UNIVERSITY OF CALIFORNIA

Los Angeles

Fault-Tolerant Cluster Management

A dissertation submitted in partial satisfaction of the

requirements for the degree Doctor of Philosophy

in Computer Science

by

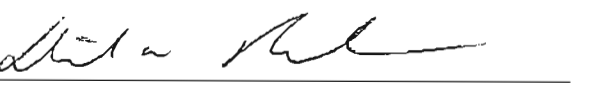
Ming Li

2006

The dissertation of Ming Li is approved.

Milos D. Ercegovac

Nathaniel Grossman



David A. Rennels

Yuval Tamir, Committee Chair

University of California, Los Angeles

2006

To my parents.

Table of Contents

List of Figures

List of Tables

ACKNOWLEDGMENTS

First, I would like to express my sincere gratitude to my advisor, Professor Yuval Tamir, for his invaluable guidance and continuous support throughout my years at UCLA. His persistent pursuit of perfection and his deep insights into various subjects have always been an inspiration to me. He has taught me so much on how to conduct qualitative research and how to improve my writing and presentation skills.

My great appreciation goes to my other committee members as well: Prof. Rennels, Prof. Ercegovac, and Prof. Grossman. I thank them for kindly agreeing to be on my doctoral committee and for their helpful advice and suggestions.

It has been a pleasure to be one of the UCLA CSL group. I would like to thank my colleagues, Israel Hsu, Navid Aghdaie, Wenchao Tao, Dan Goldberg, Machael Le, Edward Young, Donald Lam and Kahmyong Moon, for their friendship and for the enlightening discussions we had on various research topics. I have greatly benefited from these discussions and from our group meetings.

I would like to acknowledge the financial support I have received throughout my years at UCLA. This included a research assistantship as part of collaboration between NASA's Jet Propulsion Laboratory and UCLA's Concurrent Systems Laboratory under NASA's Remote Exploration and Experimentation Program as well as NASA's New Millenium Program. It also included support through a teaching assistantship from the UCLA Computer Science Department.

My deepest gratitude goes to my parents for their love, for their support and for their sacrifice over so many years. They have been the origin of my strength and will always be.

Finally, I want to thank my wife, Hongyan, for all the love she has given me, for her understanding, and for putting up with me and standing by me during the wonderful years we shared together.

VITA

December 21, 1972	Born, China
1994	B.S., Computer Science Department of Computer Science and Technology University of Science and Technology of China Hefei, China
1997	M.Eng., Computer Engineering Institute of Computing Technology Chinese Academy of Sciences Beijing, China
2000	M.S., Computer Science Computer Science Department University of California Los Angeles, California
1998-2005	Teaching Assistant/Graduate Student Researcher Computer Science Department University of California Los Angeles, California

PUBLICATIONS

M. Li and Y. Tamir (September 2004). ‘‘Practical Byzantine Fault Tolerance Using Fewer than 3f+1 Active Replicas’’, *the 17th International Conference on Parallel and Distributed Computing Systems*, pp. 241-247.

M. Li, W. Tao, D. Goldberg, I. Hsu, and Y. Tamir (September 2002). ‘‘Design and Validation of Portable Communication Infrastructure for Fault-Tolerant Cluster Middleware’’, *IEEE International Conference on Cluster Computing (Cluster*

2002), pp. 266-274.

D. Goldberg, M. Li, W. Tao, and Y. Tamir (October 2001). ‘‘The Design and Implementation of a Fault-Tolerant Cluster Manager’’, Computer Science Department Technical Report CSD-010040, University of California, Los Angeles, CA. Presented at IEEE International Conference on Cluster Computing (Cluster 2001), October 2001.

M. Li, D. Goldberg, W. Tao, and Y. Tamir (August 2001). ‘‘Fault-Tolerant Cluster Management for Reliable High-Performance Computing’’, *International Conference on Parallel and Distributed Computing and Systems*, pp. 480-485.

W. Hu, W. Shi, Z. Tang, and M. Li (February 1998). ‘‘A Lock-based Cache Coherence for Scope Consistency’’, *Journal of Computer Science and Technology (China)*, vol.13, no.2, pp. 97-109.

M. Li and Z. Tang (January 1997). ‘‘Partial Cache Locality: A New Approach of Cache Optimization’’, *Chinese Journal of Computers*, vol.20, no.1, pp. 1-8. (in Chinese).

ABSTRACT OF THE DISSERTATION

Fault-Tolerant Cluster Management

by

Ming Li

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2005

Professor Yuval Tamir, Chair

Cost-effective high-performance can be achieved using clusters of Commercial Off-The-Shelf (COTS) computers interconnected by high-speed networks. When clusters are used for critical applications and/or in hostile environment, the required system reliability can only be achieved using fault tolerance techniques that allow the system to continue to operate correctly despite component failure. Cluster management middleware (CMM) is a software layer above the operating system controlling individual nodes and below the applications. The CMM schedules tasks on a cluster, controls access to shared resources, provides for task submission and monitoring, and coordinates the cluster's fault tolerance mechanisms. Reliable operation of the cluster requires reliable, continuous operation of the management middleware.

This dissertation is focused on the key challenges in building highly reliable CMM. The system is based on centralized decision making. However, unlike most

other cluster middleware, the manager is protected by Byzantine fault-tolerant state machine replication and the ability to restore the management service to full functionality and full fault tolerance following arbitrary single faults. To this end, we use a low-cost fault-tolerant replication mechanism coupled with on-line self-diagnosis and reconfiguration. The robust replicated manager is coupled with less aggressive fault tolerance mechanisms for dealing with less critical system components and with a fault-tolerant system bootstrapping mechanism. A fault-tolerant cluster designed to operate autonomously, must include a highly-reliable trusted hardcore to control critical functions such as the initiation of a node reset. We describe the functionality required from this trusted hardcore and its interactions with the replicated cluster manager.

The result of this work is a carefully balanced integrated set of efficient practical techniques for aggressive fault tolerance. These techniques allow a highly reliable system to be built using mostly standard COTS hardware and software components. This is demonstrated in an operational system, called Ghidrah, that has been built at UCLA. This dissertation includes preliminary performance evaluation of Ghidrah and validation of the fault tolerance mechanisms by fault injection experiments.

Chapter One

Introduction

Clusters of PCs and workstations interconnected by high-speed networks have been increasingly used as a cost-effective solution for highly available services and high-performance computing (HPC). Clusters currently achieve the performance of supercomputers and enhanced reliability/availability with low-cost hardware and software.

In a typical cluster, every node is built using commercial off-the-shelf (COTS) hardware and runs a local copy of a COTS operating system. A software layer, referred to as *Cluster Management Middleware* (CMM), runs between the applications and the operating system, managing system resources for both high performance and high reliability.

The subject of this dissertation is the design and implementation of fault-tolerant cluster management middleware that allows clusters based on unreliable COTS-based hardware and software components to operate reliably even in very hostile environments. The key requirements from such middleware are high reliability, high availability, and the ability to operate continuously without operator intervention.

In this chapter, we provide a high-level overview of this thesis. First, we briefly describe the role and functionality of a cluster management system in

Section 1.1. We then discuss the motivations for building fault-tolerant cluster management middleware in Section 1.2, and present the main research contributions of this thesis in Section 1.3. In the last section, Section 1.4, we describe the organization of the rest of this thesis.

1.1. Cluster Management Middleware

A computer cluster is a group of stand-alone computers that are connected by a high-speed network and work together as a unit. As clusters are a type of distributed computing systems, so-called distributed operating systems such as Amoeba [Mull90] and MOSIX [Bara98], which are designed for distributed systems, can be used for cluster management. Such single-layer systems provide resource and processes management across cluster nodes directly from the kernel, and coordinate the operation of cluster nodes at the operating system level.

Although building the cluster management features into the operating system is potentially more efficient, it negatively impacts system portability, ability to support new hardware devices, and support for heterogeneity. Compared with custom-built ‘‘multiprocessor systems,’’ the success of cluster computing has been a direct result of the cost-effectiveness of using low-cost, widely-accepted COTS hardware and software. With the current rapid advances in COTS hardware and software, the poor portability of custom-built single-layer operating systems is highly undesirable and the case for the layered approach of ‘‘cluster computing’’ is compelling.

With the layered ‘‘cluster computing’’ approach, the system as a whole is managed by a *cluster management middleware* software layer that is interposed between applications and the OS kernel. This middleware layer creates a runtime computing environment at the user level, using standard system primitives available in most commercial operating systems for uniprocessor or small multiprocessor computers. Most existing cluster management systems are implemented as middleware on top of COTS operating systems [Ghor98, Litz88, Zhou92, Russ99, Frac02a, Cray98, Codine, SunCA3].

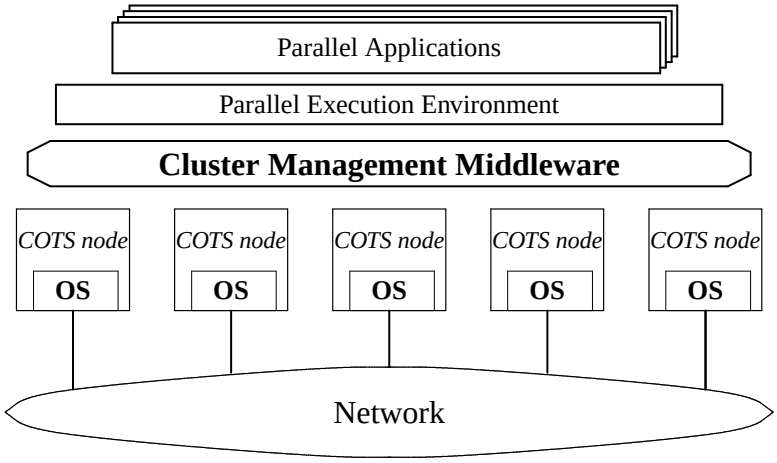


Figure 1.1: Cluster architecture

Figure 1.1 shows the typical architecture of a cluster with a cluster management middleware. In such a system, each node runs a local copy of an off-the-shelf operating system that is not designed for distributed systems, such as the standard Linux operating system. The *cluster management middleware* (CMM) runs between the applications and the operating system. It acts as an interface

between the cluster and users so users can submit their applications to the cluster. It allocates resources in the cluster to user applications and schedules application processes in a globally efficient way, so that user applications can achieve optimal performance. It also maximizes system reliability and availability, coordinates fault tolerance actions in the cluster, and provides fault tolerance supports to user applications.

The CMM is critical to the operation of the cluster: if this middleware fails, the entire cluster is useless.

1.2. Clusters for Critical Applications in Hostile Environments

The main focus of this thesis is on building highly dependable cluster management middleware for clusters operating in hostile environments and running mission-critical applications. In such hostile environments, individual cluster nodes and the communication between nodes suffer from a relatively high probability of errors due to hardware faults cause by environmental factors such as high radiation, electrical noise, or high temperatures. Examples of the need to operate in such environments include operation in space [Katz03], military application on the ground and in the air, and operation in factories near heavy machinery. Since, as discussed earlier, the nodes depend on COTS operating system kernels, errors caused by software faults must be taken into account as well [Gray86]. Examples of critical applications are applications whose results might be used to control physical devices or applications that process data that cannot be reproduced. One

specific example may be processing of data collected from different parts of a factory that might lead to a conclusion that some part of the factory must be shut down for safety reasons. Another specific example is data reduction of streaming sensor data in a spacecraft as it makes a one-time pass over a distant planet.

Since the underlying hardware and software component of clusters may be unreliable, support for critical applications implies that the system must continue to operate correctly despite component failures, i.e., the system must be fault-tolerant. Since the CMM manages all the cluster's resources, if the CMM fails the entire system fails. Thus, a necessary condition for the system to be fault-tolerant is that the CMM must be fault-tolerant.

In clusters for critical applications, another important issue is the requirement for unattended operation. In such clusters, there may not be a person present all the time who can monitor and maintain the cluster operation. The CMM system should be able to continuously self-manage the cluster without human intervention.

For these reasons, the research presented in this thesis studies the key challenges in building fault-tolerant CMM for clusters that operate in hostile environments, run critical applications, and require unattended operation. This thesis presents the design and implementation of the fault tolerance techniques we developed for building such highly reliable CMM systems.

Like many other cluster management systems [Cray98, Codine, Ghor98, Litz88, Zhou92, Russ99, Frac02a, Zhou93], the design of our fault-tolerant CMM systems is based on centralized management structure: a global, centralized

manager is in charge of making global decisions on cluster management operation such as resource allocation and task scheduling. Compared to distributed management [Gosc90, Rama98, Bara98], where management functions are distributed across all the nodes in the cluster, centralized management has the advantage of being able to achieve globally optimized decisions for management. In addition, a centralized management system is easier to design, implement and debug than a distributed management system. Additional discussion of centralized management versus distributed management is presented in Section 3.2 of Chapter 3.

The main problem with a centralized cluster manager is that the failure of the manager leads to the failure of the entire management system. Surprisingly, some cluster management systems [Ghor98, Litz88, Russ99, Frac02a] fail to deal with this problem: they just neglect this single point of failure and leave it in the system without any measure for fault tolerance. Some systems use a primary-backup approach: a cold spare or shadow copy of the manager detects the failure of the primary manager and takes over when the primary manager fails [Cray98, Codine, Gent01]. The backup manager is usually configured beforehand and the primary manager must send update information to the backup to keep their state consistent, or it has to log all its activities into reliable storage so the backup can restore the state when taking over. Some systems use different approaches, like in [Zhou93], when the manager fails, a new manager will be elected by all the working nodes and it restores the state by collecting information from all nodes.

In the management systems using primary-backup approaches, the manager failure mode is assumed to be fail-stop [Schn84], i.e., the managers never generate incorrect results. However, non-fail-stop failures do occur in real systems [Dris03]. Hence, if the fault tolerance mechanisms are based on the fail-stop assumption, reliability may be poor. Furthermore, recovery from the failure of the primary manager requires the cold backup to restore state from the storage system, apply necessary updates if its state is not up-to-date [Cray98, Codine, Gent01]. Or the system has to re-elect a new manager and the new manager collects information from all cluster nodes [Zhou93]. Recovery like these takes a long time and there is no manager available until the recovery procedure completes. This results in unacceptably long management disruptions.

Recognizing the weaknesses of existing cluster management system, we present in this thesis aggressive fault tolerance techniques to achieve highly reliable cluster management based upon active replication [Chér92] (or state machine replication[Schn90]). By replicating the management functionality across multiple nodes, we removed the single point of failure from the system: a single failed node cannot corrupt the entire system. Specifically, multiple active replicas perform all management operations and their outputs are compared and voted before any actions are taken. Hence, our CMM system can mask and tolerate failures of manager replicas without introducing long disruptions, even in the presence of non-fail-stop faults.

As a practical implementation and validation of our solution for fault-tolerant

CMM, we have implemented an operational CMM system called **Ghidrah**. Ghidrah is a general purpose CMM that is useful in many situations where high reliability for critical cluster applications is required. However, the development of Ghidrah was specifically motivated and driven by the needs at NASA for on-board high-performance computing in space [Some99, Katz03]. Space is an example of a hostile environment since, compared to the environment on Earth, radiation levels are higher, resulting in higher probability of of hardware faults. Long term robotic space exploration missions are obvious examples where continuous unattended operation is required. For such missions, on-board processing is expected to include parallel compute-intensive application for navigation, tracking, and data reduction. These application may be critical either for the survival of the spacecraft or for, what is the ultimate goal of these space missions, the collection and delivery to Earth of science data.

Ghidrah is part of the UCLA Fault-Tolerant Cluster Testbed (FTCT) project. It consists of a collection of software components that together provide continuous management functionalities. Through active replication, distributed fault diagnosis, dynamic reconfiguration and recovery, the cluster management middleware provides long-lived, robust, yet efficient management service to keep the cluster operational continuously in space, despite component failures.

1.3. Thesis Contributions

This dissertation focuses on the key issues related to the design and

implementation of fault tolerant CMM systems. In this context, it provides adaptations or improvements of several state-of-the-art fault tolerance mechanisms. It shows how these fault tolerance mechanisms can be efficiently integrated to build a practical CMM system. The main contributions of this dissertation can be summarized as follows:

- *An efficient Byzantine fault-tolerant state machine replication (BFT-SMR) algorithm that reduces replication cost and improves performance, in comparison with previous algorithms.*
- *On-line self-diagnosis and reconfiguration protocol for the replicated manager that allow the manager replicas to diagnose themselves, identify the faulty replicas, and recover from faults.*
- *A system bootstrapping protocol that enable the CMM to self-configure the cluster system without human intervention.*
- *Engineering tradeoffs leadings to a balanced set of efficient techniques for fault-tolerant CMM. Aggressive mechanisms, involving significant complexity, are used where critical to the survival of the entire system. Less aggressive simpler mechanisms are used elsewhere.*
- *Identification of the minimal functionality required from a trusted "hardcore" for unattended cluster management operation and the required interaction between the hardcore and the replicated manager.*

The CMM system we developed integrates fault tolerance techniques such as state machine replication, asynchronous consensus, atomic multicast, and

distributed fault diagnosis. The cluster manager is actively replicated across multiple nodes, and each replica performs the same management operations. The commands produced by these manager replicas are compared and voted on so that a majority of correct replicas can mask the failures of faulty replicas. Manager replicas are implemented as deterministic state machines [Schn90].

A BFT-SMR algorithm is used to maintain the consistency among the manager replicas. Existing BFT-SMR algorithms [Cast99a, Kihl98, Reit94] all require at least $3f + 1$ replicas to tolerate up to f faulty replicas. All replicas in those algorithms actively participate in the normal operation of the replicated service. Our BFT-SMR algorithm requires only $2f + 1$ active replicas for normal operation, as long as standby spare replicas are available, so that the number of active replicas plus the number of spares is at least $3f + 1$. By reducing the number of active replicas, our replication algorithm is more efficient than previous BFT-SMR algorithms, as it offers reduced replication cost and improved performance.

The replication algorithm is integrated with a distributed fault diagnosis protocol. We developed a diagnosis protocol. based on replicas comparing their internal states, that allows the manager replicas to diagnose themselves without external help. Using the diagnosis result, the manager replicas can then reconfigure themselves in a reliable manner to replace the faulty replica with a standby spare replica, thus restoring the replicated manager to full functionality and full fault resilience. Using these techniques, our management middleware provides robust and survivable cluster management.

Considering the differences on the roles and responsibilities of different components in the CMM, we applied different fault tolerance strategies to them. In our CMM, the central manager is critical to the operation of the entire system. Hence, aggressive fault tolerance techniques, such as Byzantine fault tolerant state machine replication, are used for the manager. For other components that are less critical, we applied fault-tolerance techniques that are less aggressive and involve less overhead and complexity. For example, for agents that run locally on individual cluster nodes, we implemented a simple mechanism that allows the agent to recover from crash (fail-stop) failures and maintain control over application processes that have been running on the node.

As mentioned earlier, the objective of developing these fault tolerance techniques for CMM is to ensure that critical applications submitted to the cluster are executed successfully. In order to do this, we must keep the management system operational continuously and reliably. We decouple the fault tolerance mechanisms for the management middleware from the mechanisms for applications. The CMM provides necessary supports to application-level fault tolerance, but this is separated from the fault tolerance of the CMM itself. This separation reduces complexity and allows different applications to use different fault tolerance techniques depending on their structure and reliability.requirements.

We also developed mechanisms to meet the requirements for unattended operation of the CMM. Using the self-diagnosis and reconfiguration mechanisms, the CMM can be operational continuously as long as a manager group that is fully

functioning has been constructed. In addition, we developed a system bootstrapping protocol so that the management middleware can configure itself correctly when the cluster is powered up. The bootstrapping protocol adapts to the uncertainty when the cluster just starts, and self-configures the system. The protocol is based on a Byzantine variation derived from Lamport's Paxos algorithm [Lamp98], and it allows the middleware to reliably select the nodes to run the manager replicas when the system starts.

In any system that is designed to achieve unattended operation, a trusted ''hardcore'' is needed. The necessity of the hardcore can not be avoided because, in a practical system, special operations (e.g. power-reset of a node) are required to recover from node failures cause by hardware faults or operating system faults. In addition, in the worst case, the system may catastrophic failures, such as the simultaneous failures of multiple nodes due to a burst of radiation. Recovering from such catastrophic failures may require extreme measures, such as power-resetting the entire cluster. The ability to perform these functions (e.g., node reset) must not be provided to a single unreliable COTS cluster node. Hence, there is *no way* for middleware running on the cluster nodes to perform these functions without a trusted entity to actually perform the action — ''push the reset button.'' Such a *trusted hardcore* must also survive the catastrophic failure scenario described above so that it can attempt to restore cluster operation. We define the minimal functionality required from this hardcore and the interactions between the replicated managers and this hardcore.

1.4. Thesis Organization

The rest of this thesis is organized as follows:

In Chapter 2, we review previous research work on a variety of topics in the area of fault-tolerant distributed computing that are related to our work on fault-tolerant cluster management. These areas includes asynchronous consensus, group communication, replication techniques, Byzantine fault tolerant state machine replication, and distributed fault diagnosis. We also briefly review the design and implementation of previously-developed cluster management systems and fault-tolerant distributed systems.

Chapter 3 presents an overview of our fault-tolerant cluster management middleware system. In this chapter, we discuss our choice for a centralized management structure, present the system structure of the CMM system, and the main components that form the CMM: the central manager, the agent on each node, the trusted hardcore, and an infrastructure for reliable authenticated communication. We describe the functionality of each component in the middleware, and how these components interact.

In Chapter 4 we present the state machine replication mechanism that is used to replicate the central manager for fault tolerance. In order to maintain interactive consistency [Powe88, Rush] among the manager replicas, the replication mechanism requires an atomic multicast protocol to ensure that manager replicas process input messages in the same total order. We discuss the challenges of making the atomic multicast Byzantine fault-tolerant. Starting from existing

replication algorithms, we present a new algorithm that requires fewer active replicas for Byzantine fault-tolerant state machine replication (BFT-SMR). This BFT-SMR algorithm provides the foundation for our implementation of the replicated cluster manager.

In our CMM system, the replication algorithm presented in Chapter 4 is enhanced by a manager self-diagnosis protocol. Chapter 5 presents the self-diagnosis protocol in detail. The underlying assumptions and key properties of the protocol are discussed. This chapter also describes the reconfiguration protocol that uses the results from the diagnosis to replace a faulty replica with a new replica and integrate the new replica with the existing replicas to restore the resiliency of the manager group.

In Chapter 6, we describe another important component of the CMM — the agents. We describe the fault tolerance techniques developed for the agents. We present the mechanism used to recover from agent crash failures, as well as the agent bootstrapping protocol that is used to configure the CMM system when the cluster is powered up.

The fault tolerance techniques presented in previous chapters are utilized in the implementation of the Ghidrah CMM system. Chapter 7 describes important implementation details of Ghidrah, including the internal structure of each CMM component, implementation of the communication infrastructure, the actual hardcore in Ghidrah — the spacecraft control computer, the event handling mechanism, and the group timer event mechanism for dealing with time-triggered

events on the replicated manager.

Ghidrah is currently operational on COTS hardware and software. In Chapter 7, we also present preliminary experimental results that include performance evaluation of Ghidrah as well as validation of the fault tolerance mechanisms using fault injection.

Finally, in Chapter 8, we summarize the work presented in this thesis and point out directions for future work.

Chapter Two

Related Work

The research presented in this thesis focused on building a fault-tolerant distributed system for cluster management. Our approach is to replicate the cluster management functionalities and integrate with it the state-of-the-art techniques for aggressive fault tolerance such as Byzantine fault-tolerant state machine replication, system-level fault diagnosis and reconfiguration.

This chapter examines previous work that is relevant to this thesis. It is divided into categories that correspond to the topics that are covered in this thesis. We begin, in Section 2.1, with a discussion of the main problems the must be solved when implementing fault-tolerant distributed systems. These problems includes distributed consensus, group communication, and replication. We examine previous solutions to these problems and the use of these solutions as building blocks for fault-tolerant distributed systems.

As previously discussed, the requirement of a highly reliable CMM implies that the CMM must be able to tolerate arbitrary (Byzantine [Lamp82a]) faults. Byzantine fault tolerance has been a very active research field for over two decades. Key representative results in this field are discussed in Section 2.2.

Once the failure of one of the manager replicas is suspected, the Ghidrah CMM uses distributed fault diagnosis to identify the faulty replica. In Section 2.3,

we briefly summarize the work has been done in the field of system-level fault diagnosis over the past forty years. We review different diagnosis models that have been developed, especially the distributed comparison-based models.

Many practical systems have been developed for cluster management and for distributed fault-tolerant computing. In order to compare our fault-tolerant CMM solution to these systems, in Section 2.4, we review the design of cluster management systems and fault-tolerant distributed systems that are representative in these fields

2.1. Building Fault-Tolerant Distributed Systems

A system is *fault-tolerant* if it can continue to operate correctly despite component failure. One approach to achieving fault tolerance is based on replicating critical components so that fault-free replicas can continue to perform critical functions even after some replicas fail. In a distributed system, this approach is often implemented by running multiple copies (replicas) of critical computations (processes) on different nodes (computers) of the system [Guer97]. The idea is that, even if some nodes fail, there are still computation replicas running on operational nodes to continue with the computations. Although the idea of replication is simple, robust, practical implementation of replication is often a difficult challenge.

With the replication approach, a critical component that is a single entity in the unreplicated version of the system is now a collection of replicas. All these

replicas perform the same operation and interact with other components (replicated or unreplicated) in the same way. It is then convenient to view these replicas as a group and address them as a single logical entity when other components communicate with them [Guer97]. This can be achieved using group communication [Birm93]. Group communication is used to implement efficient and reliable information dissemination among replicas so they can cooperate with consistency. A group communication system provides membership and reliable multicast services [Choc01]. The task of the membership service is to maintain agreement among processes on a set of processes that are currently operational [Mose94]. The reliable multicast service delivers messages reliably to all operational members in a group [Hadz93]. These services can be used as primitives to maintain consistent operation among replicas. Therefore, group communication is widely used as an important building block for fault-tolerant distributed systems [Guer97].

In order to maintain group membership and provide reliable multicast, a group communication system often requires that all members in a group reach common decisions on important issues, such as the set of processes that are operational and the content and order of multicast messages they received, so group consistency can be maintained. The process of this group decision-making is called consensus [Barb93]. The consensus process forms an agreement among the fault-free replicas so they can maintain the synchronization and integrity of the replica group. We will talk more about consensus in Subsection 2.1.1. Due to the importance of agreement among distributed components, the ability to solve

consensus is at the heart of a fault-tolerant distributed system [Ture92, Lamp96]. There are a number of applications of consensus in addition to group communication, such as leader election and clock synchronization.

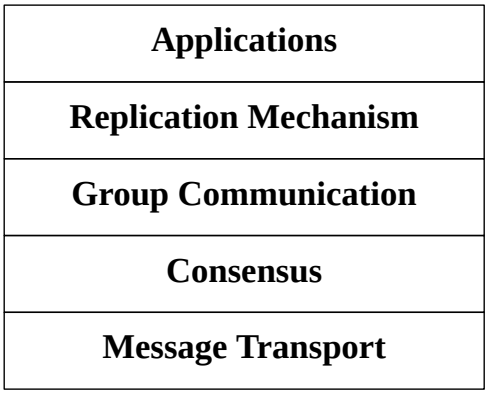


Figure 2.1: Building blocks for fault-tolerant distributed systems

So far we have discussed the important problems that need to be solved when building a fault-tolerant distributed system and the relationship between these problems: consensus, group communication, and replication. The solutions to these problems can be used as layered building blocks for fault-tolerant distributed systems, as depicted in Figure 2.1.

The bottom of this layered structure is a message transport mechanism that is use for point-to-point communication between individual components in the system. The consensus protocol is built above this communication layer as a base for cooperation and synchronization among distributed components. Group communication primitives are implemented using the consensus protocol to provide reliable delivery of messages to groups, and maintain the group membership.

Using the group communication primitives, replication can be implemented with consistency maintained among fault-free replicas. On top of the replication mechanism, a system can implement its particular application with reliability and availability. In the fault-tolerant CMM system, the application implemented on the top of this layered structure is the cluster management functionality.

In the rest of this section, we briefly examine previous research work on consensus, group communication, and replication.

2.1.1. Consensus and Paxos

As a general form of agreement in distributed systems, consensus has been extensively studied in the past two decades and resulted in a large body of work. The consensus problem is defined over a set of processes. Each process initially proposes a value and the correct processes in the set have to decide on a common value such that the following properties are satisfied:

- **Agreement** : No two correct processes decide on different values.
- **Validity** : If a correct process decides on value *v*, then *v* was proposed by some process.
- **Termination** : All correct processes eventually decide on a value.

The problem can by easily solved when there is no fault, while consensus in the presence of faults is difficult, even impossible. The FLP impossibility result presented by Fischer, Lynch and Paterson [Fisc85] asserts that there is no deterministic algorithm that solves consensus problem in an asynchronous system

with even only one crashed process.

Several approaches have been studied to circumvent the FLP result with additional assumptions about the system model. These studies have led to various algorithms that solve consensus in partially synchronous systems [Dwor88, Dole87], in timed asynchronous systems [Cris99, Fetz95], and in systems with unreliable failure detectors [Chan96a, Chan96b]. Among these algorithms, Lamport's Paxos algorithm is a well-known one. The Paxos algorithm was first introduced in [Lamp98], presented as a part-time parliamentary system on an ancient Greek island. The algorithm was then revisited in [Pris00] with a formal analysis, and further generalized and clarified in [Lamp01a, Lamp01b, Lamp05].

Paxos is an asynchronous consensus algorithm that relies on partially synchrony [Dwor88]: the bounds on message delay and relative speeds of different processes exist but they are unknown or they hold only after some unknown time. Paxos ensures termination when the system is partially synchronous and a majority of processes operate properly and communicate properly.

Paxos is a two-step consensus protocol described in terms of actions taken by three classes of agents: proposers, acceptors, and learners [Lamp01a]. The protocol proceeds in asynchronous rounds and each round is led by a proposer that guides acceptors to achieve consensus. The rounds are numbered and the numbers determines a total order on the rounds. Multiple rounds can be carried out concurrently, and each round is a distinct attempt to get a majority to accept a proposed value. It is possible that multiple rounds succeed on getting its proposal

accepted. Paxos guarantees that all accepted proposals have the same value.

This guarantee is achieved by the way that a proposer chooses a value to propose. As the first step, each proposer queries at least a majority of acceptors to learn the proposals of past rounds. An acceptor responds to the query with the proposal of the highest-numbered round it has accepted, and will not accept any proposal of rounds numbered less than the current round. The proposer then chooses the highest-numbered proposal it collected from the acceptors as its proposal and tries to get it accepted by a majority of acceptors in the second step. As any two majority sets have at least one acceptor in common, this ensures that all successful rounds decide on the same value. Once a value has been decided on, the learners learn about it from the acceptors.

Paxos guarantees the agreement and validity preperties even when there are multiple proposers running their rounds at the same time. It ensures termination when there is only one proposer and it can communicate successfully with a majority of acceptors.

Paxos provides an efficient and practical way to implement state machine replication [Lamp01a], thus it is very attractive for building fault-tolerant distributed systems. Many fault-tolerant replication algorithms —especially those achieve Byzantine fault-tolerance, which we will discuss in Subsection 2.2 —are basically variations or extensions of Paxos.

2.1.2. Group Communication

Process groups are a natural approach to implement replication for fault-tolerant distributed computing [Birm93]. Using this approach, replicas of a critical component are grouped together. The replicas operate collectively using group communication services.

Group communication provides efficient one-to-many or many-to-many communication for process groups. Among the services a group communication system may provide, *total order multicast/broadcast* (also called *atomic multicast* or *atomic broadcast*) attracts the most attention. Total order multicast is a group communication primitive that enforces reliability and a total order on the delivery of messages to a group, i.e., all messages are delivered in the same order by all correct members. There has been considerable research on total order multicast/broadcast and a large number of algorithms or protocols have been proposed. Here we take a brief look at several examples. For a comprehensive survey, one can refer to [Défa00, Défa04].

The Isis toolkit [Birm87, Birm91] is the first system for group communication. It provides services that allows processes to join process groups, broadcast messages to groups, and receive messages sent to groups. Two broadcast primitives are provided: CBCAST guarantees causally ordered message delivery, and ABCAST preserves a total order on messages. With ABCAST, a broadcast message is sent to all its destinations. Upon receiving a message, a process assigns the message a priority larger than the priority of any message it received before,

and acknowledges the sender with the assigned priority. The sender collects acknowledgments from all destinations, and then computes the maximum value of all the priorities and sends it back to all the destinations. The destinations deliver the received messages based on the decided priorities.

The group communication protocol in the Amoeba distributed operating system [Kaas91, Kaas89] implements sequencer-based atomic broadcast. A process broadcasts a message by sending it to a process designated as the sequencer. The sequencer assigns a sequence number to each message and broadcasts it to the group. Other processes in the group deliver messages in the order of their sequence numbers. A process that detects a gap in the message sequence sends a retransmission request to the sequencer. The sequencer saves all broadcast messages until it discovers that all processes in the group have received those messages. A group membership protocol based on a heartbeat mechanism is used to tolerate process failures. If the sequencer fails, the membership protocol elects a new sequencer based on the invitation algorithm described in [Garc82]. The group membership protocol may exclude fault-free processes from the group, or lead to partitioning.

In the Totem system [Amir93, Amir95], a group is organized into a logic token-passing ring. A token that contains information about reliable delivery and message order is circling around the ring. A process must possess the token before it broadcasts. When it gets the token, it increments the sequence number carried by the token and broadcasts its message with the sequence number. Retransmission

requests for missing messages are attached to the token. Totem also relies on a group membership service to handle token loss and process crash failures. Process failures are detected by an unreliable failure detector [Chan96a] implemented using timeouts. The membership protocol achieves consensus on the membership among fault-free processes and constructs a new ring. It may exclude a slow but fault-free process from the group.

2.1.3. Replication Techniques

There are two fundamental classes of replication techniques: *primary-backup replication* and *active replication* [Cher98, Guer97].

Primary-backup replication [Budh93] is also called passive replication. One replica is designated as the primary and performs the operation. Other replicas are backups and they only interact with the primary. The primary sends state update to the backups to keep the replicas consistent. If the primary fails, a fail-over occurs and one backup becomes the new primary and takes over the computation. Because it is relatively easy to implement, the primary-backup approach has been widely used in fault-tolerant systems when only benign faults such as crash-failure are considered. For example, the Harp file system [Lisk91] uses the primary-backup approach.

Active replication uses multiple identical replicas that execute simultaneously on different nodes [Ch´er92]. All the replicas are active and perform the same operation. Outputs of all replicas are voted on to produce a correct output.

Replicas are tightly synchronized to maintain consistency. Faulty replicas can be masked as long as a majority of replicas are correct. Hence, active replication can be used to tolerate non-fail-stop faults, and to avoid lengthy disruption of computation.

Active replication is usually implemented by structuring replicas as deterministic state machines, so active replication is also known as the state machine approach [Lamp82b, Schn90]. The state machine consists of a state that is the collection of all state variables used to implement the application. When an input message is delivered, the state machine performs some deterministic computation, transforms its state by modifying the state variables and produces some output messages. Processing of an input message is atomic with respect to processing of other input messages.

A replication algorithm must guarantee both safety and liveness. *Liveness* requires that the replicas complete the operation eventually. *Safety* requires that the replicas satisfy linearizability [Herl90]: the replicas behave like their one-copy equivalence that performs the operation atomically and correctly. With state machine replication, this requires *interactive consistency* [Powe88, Rush] among the replicas: correct replicas must agree on the content of input messages and the *total order* of processing the messages. Provided all correct replicas start from the same initial state and process the same input messages in the same order, they will produce the same output messages and have the same final state. Total order on input messages can be ensured by using an atomic multicast protocol (see Section

2.1.2) to deliver the messages to the replicas.

State machine replication (SMR) has been proved to be a useful approach for building fault-tolerant systems, especially for building highly reliable systems that can tolerate Byzantine faults.

2.2. Byzantine Fault Tolerance

Since the introducing of the Byzantine General problem [Peas80, Lamp82a], Byzantine fault tolerance has been widely studied both in theoretical and practical settings. Byzantine fault tolerance techniques make no assumptions about the behavior of faulty processes, therefore, can tolerate all types of faults. Research has been done in the theoretical directions such as developing consensus algorithms that are Byzantine-resilient [Brac85, Kihl03, Doud97], as well as on developing practical Byzantine fault-tolerant systems. In this section, we review previous work on Byzantine fault tolerant state machine replication (BFT-SMR).

All previous BFT-SMR algorithms for asynchronous systems require partial synchrony (see section 2.1.1) to provide both safety and liveness (see Section 2.1.3). These algorithms also assume the authenticated Byzantine fault model. In this fault model, messages communicated between processes are authenticated with digital signatures. These messages are called signed messages [Lamp82a]. A process can verify the content and the original sender of a signed message, even if the message has been relayed by other processes. Therefore, a faulty process cannot forge or spoof another process's messages. Other than this restriction, faulty

processes can behave arbitrarily.

Message authentication is usually implemented with public-key cryptosystems such as RSA [Rive78] or DSA [NIST94]. Each process can digitally sign its messages with a private key which is known only to itself. Each process can also obtain the public keys of other processes to verify the signed messages it receives.

□ **PBFT**

Castro and Liskov present a practical Byzantine fault tolerance (PBFT) algorithm for reliable services in [Cast99a]. The PBFT algorithm requires three phases for normal operations. One of the replicas (the primary) leads the other replicas (backups) to reach consensus on the order of processing client requests. All replicas are active replicas as they all execute service operations and sends replies to clients. The role of the primary is similar to the role of proposer in Paxos, and the backups play the role of acceptors. Clients send requests to the primary and wait for sufficient number of identical replies from the server replicas. A view-change protocol is triggered by timeouts, allowing the system to make progress when the primary fails. A view-change promotes a backup replica to be the new primary, but does not exclude the faulty replica from the replica group.

The PBFT algorithm ensures safety provided fewer than 1/3 of the replicas are faulty. It relies on partial synchrony to provide liveness. Messages used in PBFT are authenticated with digital signatures but public-key cryptography is not always necessary. A modification of the algorithm that uses message authentication code (MACs) during normal operations is presented in [Cast99b], which reduces the

performance overhead caused by public-key cryptography.

Compared to other work on Byzantine fault tolerance, PBFT provides a more practical and efficient solution, because it requires the minimal number of communication steps and reduces message authentication overhead.

- **Rampart**

Rampart [Reit94, Reit95] provides a practical toolkit for building Byzantine fault tolerant replicated service. Rampart includes an atomic multicast protocol that is built upon an echo multicast protocol. In the echo protocol, a process sends the digest of its multicast message to all destination processes. Processes that receive the digest ‘‘echo’’ it by sending back an echo message. When the sender process receives sufficient number of echoes, it sends a commit message with the full content of the multicast message to all processes, along with a copy of all the echo messages it has received. A process delivers the messages when it receives this commit message and verifies the echo messages. The atomic multicast protocol relies on a sequencer that decides the order of messages. The sequencer use the echo protocol to reliable broadcast the ordering information to the replica group.

Rampart uses a secure group membership protocol [Reit96] to exclude faulty replicas from the group. The membership protocol relies on an independent mechanism that allows a replica to suspect that another replica is faulty. Based on Byzantine-resilient consensus, the protocol enables correct replicas to agree on a list of replicas that are fault-free.

□ **SecureRing**

SecureRing [Kihl98] is a token-based total order broadcast protocol that can tolerate Byzantine faults. In SecureRing protocol, the replicas in the group are organized into a logical ring, and message multicast is controlled by a token. A replica can multicasts its messages only when it becomes the token holder. The token holds the ordering and authentication information for multicast messages, and is protected using public-key cryptography. SecureRing allows a single digital signature to cover multiple messages to reduce the overhead for message authentication.

SecureRing protocol also relies on a group membership protocol, which uses an unreliable Byzantine faults detector [Kihl97]. This fault detector detect faults by timeouts and by checking the authentication of messages. Once faults are detected, the membership protocol reconfigures the replica group and form a new ring consisting of only fault-free replicas.

All previous work on BFT-SMR requires at least $3f + 1$ replicas to tolerate up to f faulty replicas. In order to reduce the replication cost, Yin et al [Yin03] present an approach in which the consensus process that orders client requests and the execution of the requests are separated. With this separation, the number of replicas that execute the service operation can be reduced to $2f + 1$. However, $3f + 1$ agreement replicas that actively participate in the consensus process are still required. Our work in BFT-SMR shows that the replication costs can be further reduced by using only $2f + 1$ active replicas for *both* agreement and execution (see

Chapter 4).

2.3. System-Level Fault Diagnosis

Fault diagnosis is the process of identifying faulty units in a system. The first research work that tackled the system-level fault diagnosis problem was the PMC model proposed by Preparata, Metze and Chien [Prep67]. In this model, a system consists of a set of processing elements(PEs), and PEs can conduct test on other PEs. The PMC model use a diagnostic graph $G(V, E)$, where PEs form the set V, and each directed edge in E represents that one PE tests another PE. The testing PE sends a test to the tested PE, which executes the test and replies with the result. If the result is a passing result, the corresponding edge in G is labeled with a 0; otherwise, it is labeled with a 1. The collection of all test outcomes is called the syndrome. After the completion of all tests in G, a reliable external diagnoser analyze the syndrome to identify the faulty PEs. The PMC model assumes permanent faults and perfect test coverage, i.e., a faulty PE always fails the test from a fault-free PE.

The same testing strategy was also used in the BGM model, proposed by Barsi et al [Bars76]. This model assume that a faulty PE is always tested as faulty no matter the testing PE is faulty or not.

Malek [Male80] introduced the comparison approach for fault diagnosis. In his model the diagnostic graph is undirected, each edge represents the comparison between a pair of PEs. The edge is labeled 0 if the two PEs agree and labeled 1 if

they disagree. The assumption is that a faulty PE never agree with a fault-free PE, and two faulty PEs also disagree. As in the PMC model, the comparison syndrome is diagnosed by a centralized diagnoser. A similar approach was proposed independently by Chwa and Hakimi [Chwa81], who assumed that two faulty PEs could possibly produce the same result and their comparison outcome could be either 0 or 1.

Maeng and Malek [Maen81] extended the comparison models by having a third units to perform the comparison. The model was further extended into a generalized comparison model in [Seng92] which allows the comparator to be one of the two units being compared. In these models, although the comparisons are distributed, the comparison outcomes are still sent to a centralized supervisor for diagnosis.

Instead of relying on a trusted supervisor to diagnose the system, distributed diagnosis [Bian90, Hilt95, Hoss84, Kuhl81, Seng92] allows the components to diagnose themselves in a distributed way. With distributed diagnosis, each fault-free unit is able to independently diagnose the faulty units. This usually requires an agreement among the fault-free units.

A broadcast comparison model for distributed diagnosis is proposed in [Blou99]. In this model, the two units in a comparison pair broadcast their outputs to all units in the system. Comparisons are performed on every fault-free unit and every unit executes the diagnosis algorithm. This model assumes perfect comparison coverage: the comparison performed by a correct unit between a faulty

unit and any other unit (faulty or correct) always produces a mismatch. It also assumes that messages are broadcast in a reliable and timely manner.

Ideally, a diagnosis should be both complete and accurate. A diagnosis is said to be *complete* if all faulty units are identified. A diagnosis is *accurate* if no fault-free units are identified as faulty. Here we define *completeness* and *accuracy* in a similar way as they are defined for failure detectors [Chan96a]. In some work on fault diagnosis, accuracy is also referred as *correctness* [Shin87, Arak03].

Yang and Masson [Yang88] considered the distributed diagnosis in a soft-fail model, which covers intermittent faults and unreliable communication links. Faulty units may not always exhibit faulty behavior to others. Their algorithm allows all correct units independently diagnose the system with accuracy but the diagnosis may not be complete.

The algorithm proposed in [Shin87] uses authenticated Byzantine agreement to collect the syndrome information over several rounds of tests performed after a period of execution. This off-line algorithm provides completeness only under assumptions that a faulty processor exhibits faulty behaviors to all other processors.

On-line diagnosis algorithms, such as those proposed in [Walt94] and [Busk93], allows diagnosis to be conducted on-line as each processor collects diagnostic information without pausing the normal execution. On-line diagnosis avoids the disruption of computation caused by diagnosis.

2.4. Cluster Systems

In this section we take a quick look at a number of representative systems developed for cluster computing.

Generally speaking, a system that can be used to manage a cluster in practice should provide the following important features:

- **Transparency:** the cluster appears as a unified, powerful resource to users, i.e, the cluster management system offers a single system image (SSI) [Hwan99]. with transparent remote execution.
- **Resource Allocation:** resource management is the basic functionality required for cluster management. The management system monitors the status of all the resources in the cluster, allocates them to user jobs, and reclaims resources when they are not used.
- **Load Balancing:** the system workload is distributed across the cluster for improved performance and throughput. Balanced distribution of load leads to better job response time and resource utilization [Livn82]. Dynamic load balancing is preferred because it allows adaptive redistribution of the workload among computers to avoid load imbalance caused by workload changes [Litz88, Zhu95].
- **Gang Scheduling:** for parallel applications, significant performance benefits can be obtained if the job scheduling on different nodes is coordinated so that all the processes of a particular task are running at the same time. Effective gang scheduling leads to more efficient communication between parallel

processes and less context switch overhead [Feit90, Oust82].

- **Fault Tolerance:** the cluster needs to be continuously available even in the presence of failures. The management system should provide efficient supports for fault tolerance in applications, such as restarting, checkpointing, and process migration. The management system must be fault-tolerant itself. In clusters for critical applications in hostile environments, as discussed in Section 1.2, non-fail-stop faults need to be considered.
- **Scalability:** the cluster should be scalable. No performance bottleneck exists when the cluster is extended to a large scale.

In the rest of this section, we review a number of existing systems and examine how well they provide the features listed above. More detailed reviews of a number of research and commercial cluster management systems can be found in [Bake96] and [Kapl94].

□ **GLUnix**

Glunix (Global Layer Unix) [Ghor98] is a cluster management middleware developed for the Berkeley NOW project [Ande95]. GLUnix offers a single system image on a cluster of workstations, provides transparent remote execution service for parallel and sequential jobs. Cluster management is implemented using a centralized master process. This master process assigns tasks to node using a simple load balancing policy based on workload information periodically reported to the master by daemons running on all nodes. It also provides gang scheduling for parallel jobs.

The centralized master in GLUnix is not protected with any fault-tolerance measure. It is potentially a single point of failure in the system, and the system can not recover automatically if the master fail.

With the exception of the fault tolerance features, the basic management functionality of our CMM system is quite similar to the functionality of GLUnix.

- **Condor**

Condor [Litz88] is a cluster management system designed to provide so-called high throughput computing (HTC) for computing-intensive applications. It is a distributed batch-queue system that provides remote execution and allows utilization of otherwise wasted CPU cycles by identifying idle machines and migrating application processes to them. The process migration is supported by the checkpointing mechanism provided in Condor. A Central Manager (CM) performs the resource management by acting as a matchmaker between node-specific resources advertisements and jobs' requirements, using the ClassAd mechanism [Rama98].

Relying on the centralized CM leaves Condor with a single point of failure, although failures of the CM do not affect jobs that are already in execution. The CM has to be rebooted after it crashes and when it restarts, it collects information about all cluster nodes, and information about all running and queued jobs.

- **Chameleon**

Chameleon [Kalb99] provides an adaptive middleware infrastructure to satisfy

different levels of fault tolerance requirements of user applications. It can be used to manage a cluster for fault-tolerant applications. The Chameleon architecture is based on ARMORs, which are reconfigurable modules that control all operations in the Chameleon environment. Each type of ARMORs provides a specific fault tolerance capability.

Chameleon employs a hierarchical structure to monitor the ARMORs in the system. On the top of the hierarchy is a centralized Fault-Tolerance Manager (FTM) that is the key manager of the system. A backup FTM is used to tolerate crash failures of the FTM.

Although Chameleon is designed to manage redundant resource in a cluster, and to tolerate errors in both user applications and the middleware components, its adaptive architecture is too complicated and may not be efficient for cluster management.

□ **LSF**

Load Sharing Facility (LSF) [Zhou92, Zhou93] is a commercial product for load sharing and batch execution on heterogeneous clusters. The management is implemented using load information managers (LIMs) that distribute node load information and make task placement decisions based on the resource requirements of tasks. The task placement is performed either in a distributed way (with the potential of host overloading) or in a centralized way. LSF does not support dynamic load balancing due to the lack of supports for process migration.

In LSF, one LIM is designated as the master LIM, which is responsible for

assembling and distributing system load information, and for the centralized task placement. When the master LIM crashes, another LIM is elected as the new master by all the LIMs, using a protocol derived from the Bully algorithm [Garc82]. The new master restores its management state by collecting load information from all the LIMs.

- **Hector**

Hector [Russ98, Russ99] is a cluster runtime environment for MPI applications. It provides automatic resource allocation and transparent task migration for parallel jobs. Hector uses a central master allocator to coordinate resource allocation in the cluster, and slave allocator on every node to launch tasks and monitor local load on the node. The master allocator is not fault-tolerant. When it fails, the entire cluster goes down as all slave allocators terminate themselves.

- **STORM**

STORM [Frac02a, Frac02b] is a resource management tool designed for clusters built with a high-performance interconnection network. It integrates the management functionality with the hardware supports provided by the high-speed network to achieve fast, scalable resource allocation and gang scheduling for parallel jobs. STORM consists of a Machine Manager (MM) running on a special management node and a Node Manager running on every node. The MM is in charge of centralized resource management and is not protected with any fault

tolerance mechanism.

- **Linux-HA**

The High-Availability Linux Project [Robe04] provides a high-availability clustering solution on Linux systems. The core of Linux-HA is a package called ‘‘Heartbeat’’ [Robe00]. It provides functions such as starting and stopping resources, monitoring the nodes in the cluster, and transferring ownership of a shared IP address between nodes. This allows a standby backup to monitor a primary server and take over when the primary server fails.

- **HA-OSCAR**

OSCAR [Mugl03] was developed for Beowulf clusters. The system has a master node that receives user requests and distributes the requests to specified client nodes, detects the failures of client nodes and recover from the failures. Its high-availability extension, HA_OSCAR, uses the primary-backup approach to tolerate fail-stop failures of the master node. Fail-over from the primary to the standby backup relies on shared storage. Heartbeats on a special control network are used to detect failures.

- **MOSIX**

MOSIX [Bara98] is an enhancement of BSD Linux that supports preemptive process migration for load-balancing across a cluster. The MOSIX enhancement is implemented in the OS kernel and the modifications are transparent to the application level. The load-balancing enhancement dynamically distributes the

workload among all cluster nodes by migrating processes from overloaded node to less loaded nodes. Resource and job management in MOSIX is conducted in a decentralized fashion: each node makes its own management decisions independently, and the process migration is performed independently between a pair of nodes. With this decentralized management, the system does not have a single point of failures.

- **RAIN**

The RAIN (Reliable Array of Independent Nodes) system [Boho01] provides reliable distributed computing with fault-tolerant interconnect topologies and a reliable communication protocol. It also relies on a token-based group membership protocol as a building block for fault-tolerant applications. Only fail-stop failures are considered. A token that carries group membership information circles around the nodes. The token is also used for failure detection: if a node does not respond to its predecessor's request to pass over the token, its predecessor then decides that this node has failed. A clustering solution for firewalls, Rainwall, has been implemented based on RAIN. Rainwall supports dynamic load balancing by having lightly-load nodes requests load from heavily-loaded nodes.

Chapter Three

An Overview of the Ghidrah Cluster Management System

Cluster management middleware (CMM) schedules tasks on a cluster, controls access to shared resources, provides for task submission and monitoring, and coordinates the cluster's fault tolerance mechanisms. Thus, reliable continuous operation of the management middleware is a prerequisite to the reliable operation of the cluster. Hence, especially for supporting critical applications in hostile environments, the CMM should tolerate a wide class of faults with minimal interruptions to management operations.

Motivated by these requirements, we have designed and implemented a CMM system called Ghidrah. The Ghidrah CMM integrates a carefully-balanced set of mechanisms that allow it to continue to operate reliably despite malicious faults. This chapter provides an overview of the Ghidrah CMM system.

We first describe the design goals and system assumptions of our system in Section 3.1, then present the arguments for the centralized management structure we chose for our management system in Section 3.2. In Section 3.3, we describe the system architecture of Ghidrah and the functionality of key components. In Section 3.4, we discuss the necessity of having a trusted hardcore in the system for unattended operation. Section 3.5 describes the communication infrastructure of Ghidrah.

3.1. Design Goals and Assumptions

The main objective of the research presented in this thesis is to build a practical fault-tolerant CMM system that can provide reliable and continuous management functionality on clusters built with COTS hardware and software. The clusters are used to run critical parallel applications for high-performance computing in hostile environments.

The Ghidrah CMM system is designed to provide important management functions such as resource allocation, task control, and gang-scheduling for parallel applications. These functions must be provided with high reliability and low overhead. Furthermore, the system must support unattended operation (see Section 1.2), so it can continuously manage the cluster without human intervention.

The cluster is an asynchronous distributed system with partial synchrony (see Section 2.1.1). Processes running on cluster nodes communicate via point-to-point messages. The communication platform is unreliable: messages may be lost, delayed, duplicated, or delivered out of order by the communication platform.

Nodes in the cluster may fail such that their behavior is, essentially, arbitrary. As a result, each process running on the node may send incorrect messages, may fail to send messages it is supposed to send, may incorrectly change its internal state, or may simply crash or behave as if it crashed. When a node fails, all the processes running on the node may fail simultaneously. However, it is assumed that only one node at a time fails so that recovery action can be completed before another node fails. It is further assumed that through the use of coding and

cryptographic techniques it is possible for a receiver of a message to authenticate the identity of the sender and to verify that the message has not be corrupted during transmission.

3.2. Centralized Management

The Ghidrah CMM is based on a global, centralized manager is in charge of gathering and maintaining cluster information, and making resource allocation and job scheduling decisions. An alternative to centralized management is distributed or decentralized management: there is no global decision-maker, all nodes in the cluster act as equal peers with equal management responsibility [Gosc90]. The management functionality (resource allocation, job scheduling, and etc) is distributed across all cluster nodes, thus avoiding having a single point of failure and a centralized performance bottleneck. However, distributed management is more complex and thus more difficult to implement and debug. For example, negotiations among local managers is required to resolve contention for resources [Graf93, Rama98].

In comparison to systems based on distributed management, a system based on centralized management can manage the resources more efficiently. In particular, a centralized manager can make globally optimized decisions since it can maintain a global up-to-date view of all the resources and jobs in the cluster. For this reason, many cluster systems use a centralized management structure [Cray98, Codine, Ghor98, Litz88, Zhou92, Russ99, Zhou93].

A key potential problem with centralized management is that the failure of the manager may lead to the failure of the entire system. Therefore, in Ghidrah, the centralized manager is made reliable by replication across multiple nodes using the state machine replication approach [Cast99a].

Another possible problem with centralized management is limited scalability. However, the success of previous CMMs [Ghor98] demonstrated that the scalability of a system with centralized management is sufficient for a LAN-based system with hundreds of nodes. For larger-scale systems, the limited scalability of centralized systems may lead to unacceptable performance. Therefore, in those systems, distributed or hierarchical management architecture are often used [Chap99, Czaj98]. Since Ghidrah was not designed for clusters beyond 100 nodes, the use of hierarchical management for Ghidrah has not been investigated.

3.3. System Architecture

Figure 3.1 depicts the system architecture of the Ghidrah CMM. The middleware system consists of four components: a group of manager replicas, an agent process on each physical node, parallel applications, and the trusted hardcore. In this section, we describe the functionality of the managers and the agents, briefly talk about the applications. The trusted hardcore is discussed in the next section.

- **Managers**

As previously discussed, the Ghidrah CMM is based on centralized management. The reliability of the centralized manager directly determines the

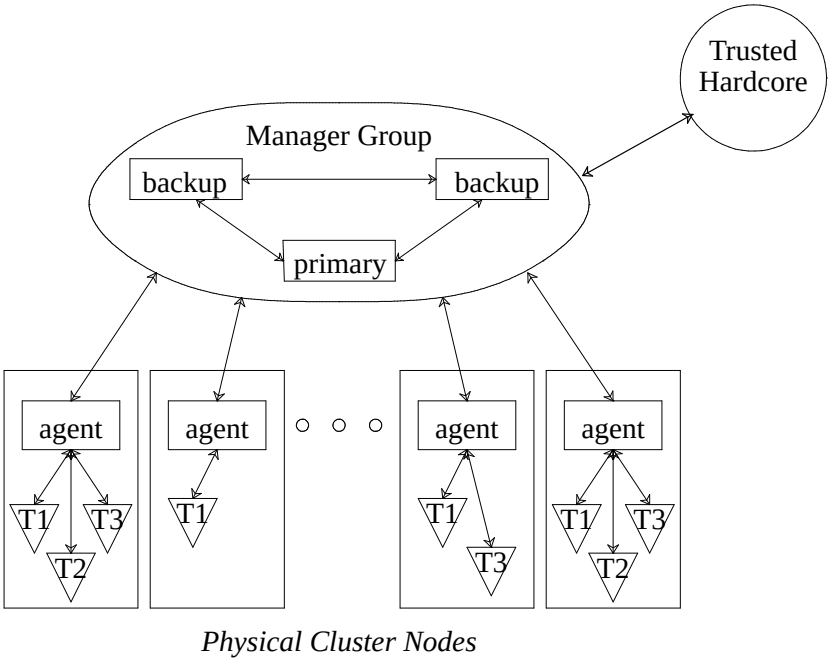


Figure 3.1: Ghidrah system architecture

reliability of the entire cluster. Hence, the centralized manager is replicated on multiple nodes using Byzantine fault-tolerant state machine replication [Cast99a] (BFT-SMR) (see Section 2.2). Our BFT-SMR algorithm is presented in detail in Chapter 4. We often refer to the replicated manager as the *manager group*.

The manager group is the centralized decision-maker and the most important part of the CMM. It is responsible for all the decisions regarding to the management of the cluster, including how to allocate resources to applications, how to schedule application tasks, and how to coordinate fault tolerance actions on all cluster nodes. In order to do that, the manager group maintains up-to-date global information about all the nodes and all the tasks that are running on the cluster. The manager group is also responsible for monitoring all cluster nodes so that it

can detect node failures and recover from them properly.

The manager group is a dynamic group: its configuration (i.e., the nodes that the manager replicas are running on) changes, with reconfigurations caused by faults. Figure 3.1 shows a manager group that consists of three manager replicas. One of the replicas is designated as the *primary* replica. The other two replicas are *backup* replicas. All three replicas independently perform the management operations and send management commands to the agents. With this setup, a single faulty replica can be tolerated at a time.

- **Agents**

An agent daemon runs on every node in the cluster. The agent acts as a proxy for the management service. It controls all the user processes that are running on the node, collects and reports to the manager group status information about the local node and user processes currently running on the node. The agent is responsible for launching processes for user tasks on the node, as well as killing and restarting processes if it is necessary. It is also the local executor of the scheduling decisions made by the managers —it stops and resumes user processes according to the scheduling decisions.

An agent acts on commands of the manager group. All the manager replicas send their commands to the agent. When receiving these commands, the agent compares and votes on them. It only accepts and acts according to a command when it has received consistent copies of the command from a majority of manager replicas.

Compared to the central manager, agents are less critical to the reliability of the entire cluster management system. For this reason, the fault tolerance techniques used on agents are less aggressive. Details about the fault tolerance mechanisms for agents are presented in Chapter 6.

- **Applications**

User applications running on the cluster are unmodified MPI parallel applications. Applications are linked with an API library that provides an interface between application processes and the local agents. We ported MPICH [Grop96] to our communication infrastructure (Section 3.5) and modified the MPI initialization code to set up the mechanisms that allow Ghidrah to control MPI applications.

The fault tolerance mechanisms for applications are separated from the fault tolerance mechanisms for the CMM. This allows different applications to utilize different fault tolerance techniques according to their requirements.

3.4. The Trusted Hardcore

Due to hardware or software faults, the hardware or the operating system on the node may fail and cause the node to be ‘‘dead’’. The only way to recover from such failures is to power reset the node. As discuss in Section 1.3, there must exist a component in the cluster that is able to perform this action. This component must be reliable and trusted; otherwise, it may become faulty and power reset a working node in the cluster, or even reset all the nodes in the cluster and cause the cluster to be unavailable.

A trusted component is also necessary for recovering from catastrophic failures, i.e., rare failures that cannot be recovered from by the managers and agents themselves. For example, the fault tolerance mechanisms used to protect the replicated managers are based on the assumption that at most f manager replicas could be faulty at a time, as described in Chapter 4 and Chapter 5. If in very rare cases, the system suffers from a burst of faults that compromises more than f manager replicas at the same time, the fault tolerance mechanisms will fail to keep the management system operational. In this situation, the entire cluster has to be power reset.

Based upon the arguments above, a *trusted hardcore* is required to keep the cluster available, even in the presence of catastrophic failures. For example, with an on-board compute cluster in a spacecraft, the trusted hardcore may be a radiation-hard highly-reliable processor, the *spacecraft control computer* (SCC), that is used mostly to control sensors and actuators on the spacecraft. This hardcore is used as a measure of last resort for recovery. This hardcore should be as simple as possible, because the more complex it is, the less reliable it will be. For this reason, it is important to minimize the functionality required from the trusted hardcore. Section 7.1.6 discusses in detail the functional requirements from the trusted hardcore as well as the interactions between the trusted hardcore and the manager replicas.

3.5. Communication Infrastructure

Communication among the CMM components are performed on top of a communication infrastructure (CI). The CI is designed to be portable across different network platforms. It provides efficient message-passing for two classes of communication on the cluster: communication between application processes and communication between key CMM components (managers, agents, and the trusted hardcore).

Both application-level communication and CMM communication requires reliability. The CI provides reliable end-to-end communication regardless of the reliability characteristics of the underlying platform [Minn01]. The communication is connection-based and provides FIFO order delivery of messages sent on a connection between two endpoints. The CI protects message integrity and guarantees that a message is delivered to its destination as long as both source and destination processes are fault-free.

The CI provides different reliability according to the different requirements of the application-level communication and the CMM communication. The application communication requires uniform reliability on all messages transmitted between application processes. On the other hand, the communication between managers, agents and the hardcore has varied reliability requirements for different message types. Some messages, such as heartbeat messages, do not require reliable transmission. Some messages require replies from the receiver at the level above the CI. For example, when managers command an agent to start an application

process, they expect a confirmation from the agent reporting that the process has been started. For this type of communication, end-to-end arguments indicate that reliability at the CI level are not needed. In order to provide efficient supports to various types of CMM communication, the CI is designed to offer adaptive reliability. Detailed implementation of the CI is discussed in Section 7.1.7.

As mentioned in Section 3.1, message transmitted between key CMM components (manager, agent, and the hardcore) are authenticated using cryptographic techniques. Message authentication is implemented in the CI. The CI provides the ability to generate and verify authentication codes, allowing the message receiver to authenticate the identity of the sender. If the identity of the sender cannot be authenticated, the receiver discards the message.

Chapter Four

Efficient Byzantine Fault Tolerance for Replicated Services

As discussed in Section 2.2, reliable services can be implemented using Byzantine fault-tolerant state machine replication (BFT-SMR) [Cast99a]. With this approach, the server replicas are implemented as deterministic state machines that are replicated across multiple nodes [Schn90, Lamp82b] (see Section 2.1.3).

All existing BFT-SMR algorithms require at least $3f + 1$ server replicas to tolerate up to f faulty replicas. All replicas in those algorithms actively participate in the agreement on a total order of client requests. Some efforts have been taken to reduce the replication cost for BFT-SMR. For example, in [Yin03] the replicas that agree on request order are separate from the replicas that actually process (execute) the requests. The solution requires only $2f + 1$ execution replicas, but it still requires at least $3f + 1$ agreement replicas.

In this chapter, we present a BFT-SMT solution that reduces the replication costs. Our solution requires only $2f + 1$ active replicas for *both* agreement and execution. In addition to the active replicas, this solution requires standby spares, such that the number of active replicas plus the number of spares is at least $3f + 1$. Standby spares are involved in the algorithms only for reconfigurations when the active replicas fail to make progress (due to faulty replicas or simply due to unexpectedly long delays). Each such reconfiguration leads to a different subset of

nodes running the active replicas. Eventually, the system reaches a configuration where the $2f + 1$ active replicas are fault-free and thus are able to make progress.

Our BFT-SMR algorithm provides the same Byzantine resiliency and service characteristics as previous algorithms under the same synchrony assumptions. The use of fewer active replicas results in reduced power assumption for processing and communication. It also has the potential to provide better performance during normal, fault-free execution since each active replica communicates with fewer replicas.

In this chapter, we discuss this new BFT-SMR algorithm as a general solution to implementing fault-tolerant services. In the CMM system, this algorithm is used to replicate the central manager to achieve high reliability. In the context of the CMM, the servers are the managers, while the agents and the trusted hardcore are the clients.

The rest of this chapter describes this new BFT-SMR algorithm. In Section 4.1, we analyze the problems and issues that we must resolve when making the replication algorithm Byzantine fault-tolerant. Section 4.2 presents our algorithm and the corresponding correctness proof for the case of $f = 1$, where there are three active replicas and a single replica failure can be tolerated. In Section 4.2.7 and 4.2.8, we also discuss extension of the algorithm to any f so it tolerates multiple simultaneous faults, and extension of the algorithm to an open replica group, respectively. In Section 4.3, we present experimental results based on simulations of our algorithm and the algorithm presented in [Cast99a], which requires $3f + 1$

active replicas. The results show that our algorithm improves the performance of fault-free execution.

4.1. Achieving BFT-SMR

A state machine replication (SMR) algorithm must guarantee both safety and liveness (see Section 2.1.3). Safety requires that all correct replicas process clients' messages atomically in the same order. The response that any correct client accepts from the replicated state machine cannot be inconsistent with the state of any correct server replica. The safety property can be ensured if the algorithm can guarantee agreements among all correct replicas on the total order of processing clients' messages.

Liveness requires that all messages sent by clients are eventually processed by all correct replicas, and a client eventually receives and accepts correct replies to its requests

In order to assure that the reply it receives is correct, a client only accepts a reply after it has received consistent replies from a majority of the server replicas. Since the number of faulty replicas is at most f, the system must have at least $2f + 1$ replicas so correct replicas are the majority.

As we mentioned in Section 2.1.3, atomic multicast protocols can be used to ensure that clients' messages are delivered to all replicas in the same total order. In order to ensure the same total order, a simple and efficient solution for atomic multicast is to have one replica act as the sequencer and be responsible for ordering

messages. All messages from clients are sent to this sequencer. Upon receiving a message, the sequencer assigns it a sequence number and relays the message with its sequence number to all other replicas. The messages from the clients (agents) are authenticated as described in Section 2.2, so a faulty sequencer cannot modify a client's message or forge a client's message. It can only discard the message or mess around with the sequence number it assigns to the message, as we will discuss later.

When a replica receives a message with an assigned sequence number from the sequencer, it delivers and processes the message in the order specified by the sequence number.

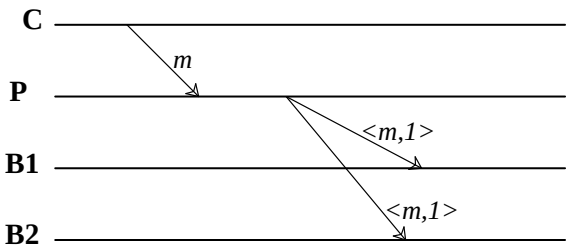


Figure 4.1: Sequencer-based atomic multicast. P is the faulty primary, $B1$ and $B2$ are two backups, and C is a client. $\langle m, 1\rangle$ is the message that consists of the client message and the sequencer number assigned by the sequencer.

Figure 4.1 depicts an example of this sequencer-based atomic multicast. Although in Figure 4.1 and other figures shown in this section, we only illustrate the case for $f = 1$, the arguments presented in this section are correct for any f.

This sequencer-based mechanism has been used in several atomic multicast/broadcast protocols, such as the group communication protocol in the

Amoeba distributed operating system [Kaas91, Kaas89]. In Amoeba, this mechanism is called the PB method [Kaas96]. There exists another version of the sequencer-based approach, which is called the BB method in Amoeba [Kaas96]. In this variant, the sender multicasts its message to the sequencer and all other replicas. When the sequencer receives the message, it sends to all other replicas a special message containing only the sequence number it assigns to the message. Each replica then delivers messages in sequence after it has received both the message and its sequence number.

The BB version of the sequencer-based algorithm reduces the workload of the sequencer because the sequencer does not have to relay clients' full messages. However, it requires nearly twice as many messages as the first version and increases the number of interrupts to replicas for receiving messages. For this reason, the multicast protocol in our state machine replication algorithm is based on the first (PB) version [Kaas96]. In our protocol, the primary replica plays the role of the sequencer. Clients send their messages to the primary replica; it then assigns sequence numbers to messages and forwards the messages to all backup replicas.

With these simple sequencer-based protocols described above, although faulty backup replicas can not sabotage the correctness of the protocol, the primary replica (or the sequencer) could be a single point of failure. In Ameoba, a group membership protocol [Kaas91] is used to deal with fail-stop failures of the sequencer. Each process periodically ‘‘pings’’ another process by sending a message to the process asking it to respond. If after a certain number of trials the

process does not respond, the process is declared as ‘‘failed’’. If the sequencer is declared as ‘‘failed’’ this group membership protocol elects a new sequencer with the invitation protocol described in [Garc82]. The protocol starts by each process establishing a group that only includes itself and considering itself as the coordinator. Each coordinator then invites other processes to join its group. If a process that is not a coordinator receives the invitation, it responds with the highest sequence number it has seen. If one coordinator invites another coordinator, the one with the higher sequence number becomes the coordinator of the joint group (if their sequence numbers are equal, the one with the lower id is chosen). When all alive processes have been invited, the coordinator of the final group is the new sequencer. This new sequencer then broadcasts a ‘‘new group’’ message to all other process and retransmits missing messages to other processes.

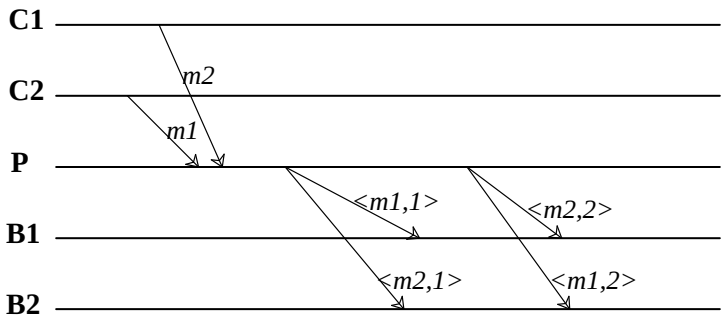


Figure 4.2: Malicious primary sending inconsistent ordering information. P is the faulty primary, $B1$ and $B2$ are two backups, and $C1$ and $C2$ are two clients. $m1$ and $m2$ are two messages from different clients, and messages such as $\langle m1, 2\rangle$ are ⟨message, sequence number⟩ pair sent by the sequencer.

The mechanism described above may fail if the sequencer is subject to non-fail-stop failures. For example, as illustrated in Figure 4.2, the malicious primary

(sequencer) relays messages it received from the clients with inconsistent sequence numbers to the two backup replicas. Each backup replica delivers and processes those messages in the order indicated by the sequence numbers that are different than the order accepted by the other backup. Thus the two correct backups would have inconsistent states and generate different outputs.

It is clear that this simple protocol with one-phase message forwarding cannot guarantee safety in presence of Byzantine faults. A fault-tolerant protocol must prevent a malicious primary from causing correct backup replicas to become inconsistent. This can be achieved by having all backup replicas exchange and compare the messages they received from the primary. In this way, if the primary is faulty and sends inconsistent ordering information, the backup replicas can detect the inconsistency and prevent their states from diverging. This additional exchanging phase leads to a two-phase protocol that is shown in Figure 4.3.

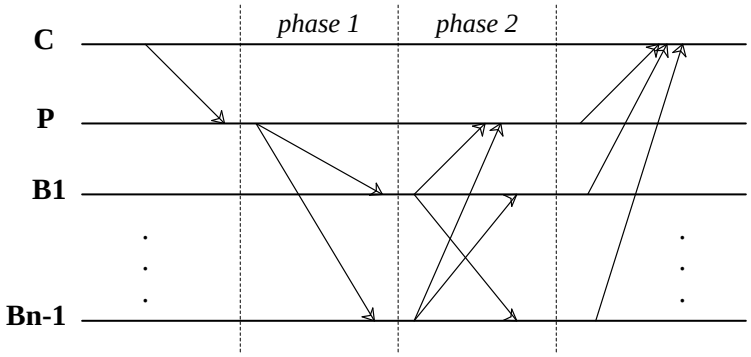


Figure 4.3: Two-phase multicast protocol. Backup replicas broadcast the ordering messages received from the primary to ensure consistency P is the primary, B_1 ... B_{n-1} are backups, and C is a client.

In this two-phase protocol, the purpose of the second phase is to ensure that a

correct replica does not accept (process) a message unless there is a total of at least $f + 1$ *correct* replicas that have received the message with the same sequence number. Thus, after receiving a message with a sequence number from the primary, each backup broadcasts this ⟨message, sequence number⟩ pair to all the replicas. A replica accepts (processes) a message with a sequence number only after it has received consistent ⟨message, sequence number⟩ pairs from at least $2f$ *backups*, possibly including itself. Since a maximum of f of the backups may be faulty, this ensures that consistent ⟨message, sequence number⟩ pairs are received from at least f correct backups. Thus, taking into account that backups initially receive the message from the primary, this protocol does ensure that each correct replica will proceed only if there are at least f other replicas that will proceed with the same ⟨message, sequence number⟩ pair.

Unfortunately, the two-phase protocol just described does not ensure liveness. Specifically, the f faulty replicas may *all* stop responding and be silent forever. Thus, if correct replicas wait for messages from $2f$ replicas, they may wait forever.

In order to ensure liveness with the two phase protocol, the number of replicas must be increased beyond $2f + 1$. If the total number of replicas is n, the number of backups is $n - 1$. Due to the liveness problem explained in the previous paragraph, a replica must be able to proceed after receiving consistent messages from $n - 1 - f$ *backups* (possibly including itself). Hence, due to the safety requirement previously discussed, $(n - 1 - f) \geq 2f$; that is, $n \geq 3f + 1$.

Due to the argument above, all previous Byzantine fault tolerant SMR

algorithms for asynchronous systems require at least $3f + 1$ replicas to tolerate f faulty replicas [Cast99a, Kihl98, Reit94]. The requirement for at least $3f + 1$ replicas holds even in a system where messages are authenticated [Brac85, Dwor88].

With at least $3f + 1$ replicas, faulty backup replicas in the two-phase protocol cannot cause correct replicas to accept incorrect message order or prevent correct replicas from making progress, because a correct replica can always receive exchanged messages from at least $2f$ backup replicas (may including the replica itself) at the end of the second phase, no matter what these faulty replicas do.

With the two-phase protocol and $3f + 1$ replicas, a faulty primary can prevent correct backup replicas from making progress. For example, for each message a faulty primary may attach different sequence numbers when it forwards the message to different backup replicas. This will prevent the backup replicas from reaching agreement at the end of second phase and thus block the backups from making progress. Even if the primary can only fail-stop, it can block the protocol if it simply stops forwarding received messages to the backup replicas.

Since with the protocol as described so far even a fail-stop primary can block progress, it is clear that the protocol is missing a critical component —a way to recover from the failure of the primary replica. Such recovery is possible based on a timeout mechanism. A client sets a timer with a timeout period when it sends a message to the primary. If it does not receive a reply before the timer expires, it sends the message to all backup replicas. When a backup replica receives the

message, it also starts a timer with a timeout period. When the timeout period expires, the backup replica invokes the reconfiguration procedure by broadcasting a message to replace the failed primary with a backup replica. When it receives messages from $2f$ other backup replicas that agree the primary change, it commits to the configuration with the new primary. This new primary resumes the ordering protocol and leads other replicas to proceed.

This reconfiguration procedure is also called ‘‘view-change’’ [Cast99a]. The use of timeout means that such a protocol relies on partial synchrony [Dwor88]: the bounds on message delay and relative speeds of different nodes exist but they are not known and they hold only after some unknown time. All BFT-SMR algorithms assume a system model with partial synchrony as they rely on timeouts.

The atomic multicast problem becomes more complex once view-changes are introduced. Specifically, the two-phase protocol cannot guarantee safety across view change when some messages sent between replicas can get delayed or lost. Consider the following scenario: the primary is faulty and it sends identical ordering message to $2f$ backup replicas but different ordering messages to the other f replicas. Since there are $2f$ replicas that have the same ordering message and broadcast the message in the second phase, some correct backup replicas will be able to receive enough consistent messages so they accept the order and process the message. For example, assume that the number of backup replicas that accept this order is f. Other correct replicas did not receive enough ordering messages because some messages sent to them were delayed or lost; as a result, these replicas

do not accept the same order and are blocked. Eventually, timeouts expires on these replicas and the view-change is triggered: the old primary is replaced by a new primary that is a replica that has not accepted a message order and received a different ordering message from the old primary as we mentioned before. After the view-change, this new primary invokes the two-phase protocol with the ordering message it has, which is inconsistent with the ordering message that has been accepted by those f replicas. Because there are still $2f + 1$ replicas (including the old primary) that have not accepted a message order yet, a correct replica can receive enough consistent messages for the new order so it accepts the new order. However, this new order is different than the order that has been accepted by f correct replicas: inconsistency exists among correct replicas and safety is violated.

The inconsistency among correct replicas as described in the above scenario is caused by the fact that when a correct replica accepts the message order at the end of the second phase, it does not know for certain that the same message order has been or will be accepted by other correct replicas. In order to avoid this problem, replicas have to exchange such information in an additional phase so that they can be assured that other correct replica will accept the same message order before they commit to this order. Such a three-phase protocol (and its variants) has been used in previous BFT-SMR algorithms that requires at least $3f + 1$ replicas. As an example, Figure 4.4 shows the three-phase protocol used in Castro and Liskov's BFT algorithm [Cast99a] (see Section 2.2).

In Castro and Liskov's algorithm, the three phases are named *pre-prepare*,

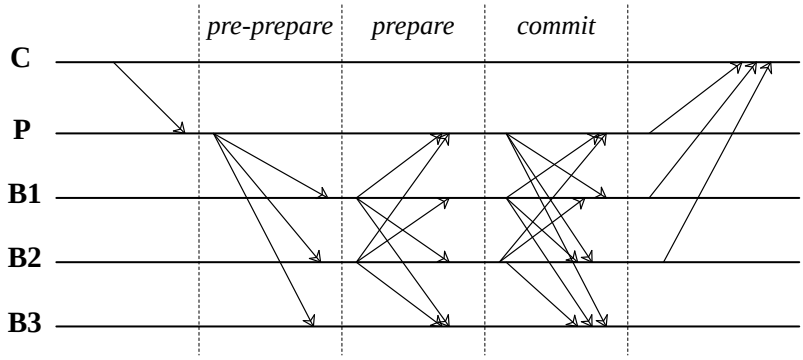


Figure 4.4: Three-phase protocol with 3f+1 replicas (Castro&Liskov[Cast99a]). B3 is a faulty backup replica.

prepare and *commit*. Backup replicas broadcast the ordering information they received from the primary as prepare messages. A replica waits until it has received $2f$ consistent prepare messages (may include its own prepare message) before it proceeds into the commit phase. In the commit phase, each replica broadcasts a commit message. A replica finally commits to a message order and process the message only when it has collected $2f + 1$ commit messages.

By having these conditions for a replica to proceed and accept a message order, this protocol ensures safety. It can also tolerate failures of the backup replicas. As illustrated in Figure 4.4, the faulty backup replica, *B*3, cannot prevent the correct replicas from making progress: the correct replicas can still collect enough messages to proceed even if the *B*3 does not send out any message. The algorithm relies on timeouts and a view-change protocol to provide liveness when the primary fails. The view-change protocol, together with the commit phase, ensures consistent message order across view changes.

4.2. BFT Replication with Fewer than 3f+1 Active Replicas

As discussed earlier, a BFT-SMR algorithm requires at least $3f + 1$ replicas to provide both safety and liveness. This requirement is based on the assertion that correct replicas cannot wait for responses from silent faulty replicas forever; otherwise, liveness cannot be ensured. In order to deal with the faulty primary, previous $3f + 1$ algorithms have to use the timeout mechanism and rely on partial synchrony (Section 4.1). With previous algorithms the $3f + 1$ replicas participate *actively* in the handling of every message, at least in the ordering of the message. However, in this section we show that this is not necessary. Specifically, we present a new BFT-SMR algorithm that requires only $2f + 1$ of the replicas to be active during fault-free operation. The total number of replicas is still $3f + 1$ but, as long as a view change is not triggered, f replicas are idle —they do not receive any messages, send any messages, or perform any processing. Under the same synchrony and fault assumptions, safety and liveness can still be achieved with this algorithm [Li04].

The new BFT-SMR protocol involves three phases and proceeds from one phase to the next under the same conditions as with Castro and Liskov's protocol [Cast99a]. The key difference is that, during normal (fault-free) operation, only $2f + 1$ replicas instead of $3f + 1$ replicas participate. Thus, in the first phase, each backup replica receives a pre-prepare message from the primary. In the second phase, each replica waits until it has received consistent prepare messages from $2f$ backup replicas (may include itself). In the third phase, each replica waits

until it has received consistent commit message from all other $2f$ replicas. Note that the conditions for a replica to proceed are precisely the same as the conditions in Castro and Liskov's algorithm for $3f+1$ replicas. By enforcing these conditions, the algorithm can still ensure safety (see the discussion in Section 4.1) with only $2f+1$ replicas as faulty replicas could not mislead correct replicas to behave incorrectly. The replicas may either generate the correct result or they may fail to make progress due to the faulty replicas. Liveness (see Section 4.1) cannot be guaranteed if any replica, the primary or a backup, is faulty.

In order to resolve the liveness problem with $2f+1$ active replicas, standby spare replicas must be available and the timeout mechanism needs to be used. The spare replicas are used only for reconfigurations that are triggered by timeouts. During a reconfiguration, spare replicas are activated to replace the faulty replicas so that correct replicas can proceed.

As discussed in the previous section, previous algorithms use the same timeout mechanism to trigger reconfigurations (view-changes) in order to ensure liveness when the primary fails. Therefore, the synchrony assumption (partial synchrony) in our algorithm is the same as in previous algorithms.

Our new algorithm reduces the replication cost for BFT-SMR. The benefit of using fewer active replicas during fault-free execution comes from the reduction in computation and communication overhead. Although it still requires the same number of replicas to be available, the standby spare replicas stay inactive most time and are only involved in reconfigurations, thus the resources that would

otherwise be consumed by them can be saved for running user applications. In addition, as only active replicas communicate during fault-free execution, with fewer replicas being active, each replica sends and receives fewer messages. This reduces the communication overhead and number of interrupts of the replicas, thus improves the normal-case performance.

On the other hand, compared to Castro and Liskov's algorithm, in which the reconfigurations are only triggered by primary failures, our algorithm is more sensitive to faults because a faulty backup replica can also cause reconfigurations. However, since faults are rare and normal-case performance is more important in practical systems, our algorithm is more efficient than previous BFT-SMR algorithms.

The remainder of this section describes our algorithm for the single-fault case, that is, $f = 1$, so there are only three active replicas and one spare replica are available. Each node with a replica is assigned a unique ID in $\{0,1,2,3\}$. As discussed in Section 4.2.7, the algorithm can be easily extended to any f.

As in [Cast99a], we rely on authenticated communication. Thus, messages are authenticated with digital signatures. We denote a message m signed by replica i as $\langle m \rangle_i$.

We consider a closed replica group in our algorithm: the set of replicas does not change, it always consists of the same four replicas. A reconfiguration changes the roles of the replicas —the designation of each replica on a particular node as primary, backup, or spare. It neither removes a replica from the group, nor brings a

new replica into the group. Issues related to extending the algorithm to an open or dynamic replica group are discussed in Section 4.2.8.

The algorithm consists of a normal-case protocol and a ‘‘view-change’’ protocol that is used to reconfigure the replicas in the presence of faults. The replicas move though a sequence of configurations called *views*. A view is identified by a unique view number: an integer v. The primary of a view is replica i, such that $i = v \ mod \ 4$. The replica identified by $(v + 3) \ mod \ 4$ is the spare, and the remaining two replicas are backups. A view change always causes a change from view v to view $v + 1$. Thus, the same configuration is reused after four consecutive views are installed, but with a new view number. The replica that is primary in the current view will be the spare of the next view. The algorithm starts with view 0, and new views are installed by the view-change protocol.

During normal operation, the standby spare replica does not send or receive any messages and does not perform any operation. In fact, the spare replica does not have to be a real process running on a node. It can be virtual and the process is spawned only when the replica is promoted to be active, as long as there is a daemon process on the node that can start the real process.

The algorithm works roughly as follows:

(1) A client sends a *group message* to the primary replica;

(2) The primary assigns a sequence number to the message and multicasts it to the backups using the total ordering protocol;

(3) All active replicas process the message and send a response or an

acknowledgment to the client;

(4) The clients accept and execute the server replicas' messages only when they receive identical copies of message from at least two different replicas.

The total ordering protocol guarantees that all correct server replicas agree on a total order for processing client's messages even in the presence of faults.

4.2.1. Client-Side Protocol

The behavior of clients in our algorithm is similar to the behavior of clients in the BFT-SMR algorithm presented in [Cast99a]. View-changes of the server replica are visible to all the clients, as the server replicas notify the clients whenever a new view is installed. A client sends its message to the replica it believes to be the current primary.

The client message is in the form of $\langle$ GROUP-MESSAGE, *cs*, —, *c* $\rangle_c$, where *c* is the client's identifier, *cs* is the channel sequence number (CSN) that is used to ensure FIFO order on messages from this client to the replica group, and "—" represents the content of the message.

A client buffers the replies it receives from active server replicas until it has received identical replies from two different server replicas. Each of these messages from the server replicas has a global send sequence number (GSSN) so the client can match messages from different server replicas.

The client sets a timeout when it sends a group message to the primary. If it

does not receive valid replies from at least two server replicas before the timeout expires, it sends the same group message to all replicas. This step indirectly triggers the reconfiguration protocol (see Section 4.2.3) and is critical for ensuring liveness [Cast99a].

4.2.2. Normal-Case Operation

The protocol for normal-case operation in our algorithm is very similar to the three-phase protocol in [Cast99a]. The only difference is that in our algorithm, only three, instead of four replicas, participate in the protocol.

When a server replica receives a group message from a client, it checks the CSN of the message. If it has already processed a message with the same CSN, it simply re-send the reply. As with other reliable communications protocols, replicas have to keep the reply message they sent to each client until they receive an acknowledgment from the client indicating that the client has received the the reply message. In the case that the group message has not been processed yet, if the replica is a backup, and it has not received from the primary a pre-prepare message (described below) for this group message yet, it relays the group message to the primary.

When the primary replica (denoted *p*) receives a group message *m* for the first time, it starts the three-phase protocol to lead the backups to agree on the total order of processing the message. In the first phase, the *pre-prepare* phase, the primary assigns a receive sequence number (RSN) *s* to the group message *m* and

multicasts a *pre-prepare* message with m to all the backups. The *pre-prepare* message has the form $\langle$ PRE-PREPARE, v, s, d_m $\rangle_p$, where v indicates the current view, and d_m is group message m's digest, generated by a collision-resistant hash function [Rive92].

After the multicast, the primary inserts the group message m and the pre-prepare message into it *message log*. Each active replica maintains a message log for recording the messages it receives and it sends to others, in case they are needed for retransmission or as proofs that are required by the protocol. We will describe how to truncate the log in Section 4.2.4.

A backup replica b accepts the $\langle$ PRE-PREPARE, v, s, d_m $\rangle_p$ message and the group message m if the following conditions are all true: it is in view v; the pre-prepare message and the group message m are all properly signed, by the primary of view v and by the client c, respectively; it has not yet accepted a *pre-prepare* message for view v with the same sequence number s but a different digest; and predicate *prepared*(m', v', s, b) (discussed below) is not true for a group message m' that is different than m for any previous view $v' \leq v$.

If backup replica b accepts the pre-prepare message, it enters the *prepare* phase by multicasting a *prepare* message $\langle$ PREPARE, v, s, d_m, b $\rangle_b$ to all other replicas (including the primary), then adds the group message m, the pre-prepare message, and its prepare message to its message log. Otherwise, it simply discards the message.

When a server replica (including the primary) receives a prepare message, it

accepts and inserts the message into its log provided that the message is properly signed, the view number in the message is the same as the replica's current view.

As in [Cast99a], the predicate *prepared*(*m*, *v*, *s*, *i*) is true if and only if replica *i* has inserted into its log the group message *m*, the pre-prepare message for *m* in view *v* with sequence number *s* and matching digest, and the prepare messages from the two backups that match the pre-prepare message (including its own prepare message if *i* is a backup). The pre-prepare message and prepare messages forms the *prepared certificate* of $\langle m, s \rangle$. This certificate proves that all active replicas have agreed to assign sequence number *s* to *m* in view *v*. It will be used in the view-change procedure as a proof of the agreement to the spare replica (see the subsection 4.2.3).

When *prepared*(*m*, *v*, *s*, *i*) becomes true, replica *i* multicasts a $\langle$ COMMIT, *v*, *s*, d_m, *i* $\rangle_i$ to all other replicas and inserts the message into its log. If a replica receives a *commit* message that is properly signed by the sender, and the view number *v* in the message is equal to its current view, it accepts the message and adds to its log.

The *committed certificate* of $\langle m, s \rangle$ is defined as the set that consists of three commit messages from different replicas (including itself) with the same RSN *s*, the same digest d_m, and the same view *v*. It proves that all active replicas in view *v* are ready to commit to *m* with sequence number *s*. We say that $\langle m, s \rangle$ is committed by replica *i* (primary or backup) in view *v* if and only if *prepared*(*m*, *v*, *s*, *i*) is true, and it has the committed certificate of $\langle m, s \rangle$, and for every client message that has

a lower RSN than *s*, it either has processed the message, or its state shows the message has been processed (the replica may get the state from another correct replica, as described later). A replica process message *m* when $\langle m, s \rangle$ is committed by it. This ensures that all correct replicas process group messages in the same total order thus produce the same result.

After processing the group message *m*, server replica *i* sends a reply $\langle$ GROUP-REPLY, *v*, *cs*, *gs*, —, *i* $\rangle_i$ or an acknowledgment $\langle$ GROUP-ACK, *v*, *cs*, *i* $\rangle_i$ to the client *a*, where "—" is the result of the execution, *gs* is the global send sequence number (GSSN) assigned to the group reply message, and *cs* is the original channel sequence number (CSN) of *m*.

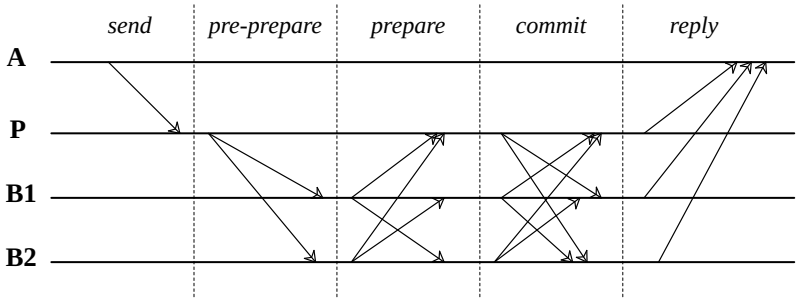


Figure 4.5: Normal case protocol for total order multicast. *P* is the primary, *B*1 and *B*2 are backups, and *C* is a client.

Figure 4.5 shows the message exchanges in the normal-case protocol. The protocol described above is presented formally in Figure 4.6 as the behavior of each server replica *i* in a view *v*. The protocol is presented in an asynchronous event-driven style. The statement of "□ **Upon** *E* **do** *A*" indicates that when an event *E* occurs, the replica performs the action described by *A*. The execution of *A* are

atomic, so that no other statements are executed before the execution of *A* completes. There are also statements like "□ **If** *C* **then** *A*", which states that whenever condition *C* becomes true, the replica executes *A*. We assume that all conditions are constantly under evaluation, so that a condition is re-evaluated whenever a variable it concerns has changed.

This algorithm provides safety: a correct server replica processes a group message *m* in the order indicated by its sequence number *s* only when it has received consistent commit messages for *m* and *s* from all replicas — *prepared*(*m*, *v*, *s*, *i*) is true for all replicas. At that moment, it knows that all correct replicas will not accept any different ⟨*m*, *s*⟩ pair in the same view. As a result, all correct replicas execute the group messages in the same total order.

As described so far, the algorithm does not ensure liveness. If a faulty replica sends to others incorrect *m* and *s* or simply does not send out messages, the correct replicas may not proceed. However, as mentioned earlier, our complete algorithm does ensure liveness since lack of progress eventually results in a timeout that triggers the view-change protocol which is described next.

4.2.3. View-Changes When Faulty Replicas Block Progress

The normal-case protocol for ordering group messages does not prevent a faulty replica (primary or backup) from blocking the protocol. This is a difference between our algorithm and Castro and Liskov's BFT-SMR algorithm [Cast99a]. In their algorithm, the normal case operation is blocked only when the primary replica

Variables

$R \equiv \{v \bmod 4, (v+1) \bmod 4, (v+2) \bmod 4\}$ /* the active replica set */
$primary \equiv v \bmod 4$
$B \equiv \{(v+1) \bmod 4, (v+2) \bmod 4\}$ /* the backup set */
$spare \equiv (v+3) \bmod 4$
rsn /* the current RSN */
s_{last} /* the RSN of the last group message processed */
$cs_{last}(a)$ /* the CSN of the last group message received from agent a */
L /* the message log */

Predicates

$prepared(m, v, s, i) \equiv m \in L \wedge \langle$ PRE-PREPARE, $v, s, d_m \rangle_{primary} \in L \wedge$
$\forall r \in B : (\langle$PREPARE, $v, s, d_m, r\rangle_r \in L)$
$committed(m, v, s, i) \equiv prepared(m, v, s, i) \wedge \forall r \in R : (\langle$COMMIT, $v, s, d_m, r\rangle_r \in L) \wedge$
$\forall s', s' < s : (\exists v', m' : (v' \leq v) \wedge committed(m', v', s', i))$

□ **Upon** RECEIVE $msg = \langle$GROUP-MESSAGE, $cs, —, c\rangle_c$ **do**
if $cs < cs_{last}(c)$ **then** discard msg
else if $cs = cs_{last}(c) + 1$ **then**
$L \leftarrow L \cup \{msg\}$
start *timer* with timout T
if $i = primary$ **then**
$rsn \leftarrow rsn + 1$
let $mm = \langle$PRE-PREPARE, $v, rsn, d_m \rangle_i$
multicast mm to B
$L \leftarrow L \cup \{mm\}$
else buffer msg

□ **Upon** RECEIVE $msg = \langle$PRE-PREPARE, $v, s, d_m \rangle_{primary}$ **do**
if $\forall v' : (v' \leq v) \wedge (\nexists\, m' : ((d_m \neq d_{m'}) \wedge prepared(m', v', s, i)))$ **then**
let $mm = \langle$PREPARE, $v, s, d_m, i \rangle_i$
multicast mm to R
$L \leftarrow L \cup \{msg, mm\}$

□ **Upon** RECEIVE $msg = \langle$PREPARE, $v, s, d_m, j \rangle_j$ **or** $msg = \langle$COMMIT, $v, s, d_m, j \rangle_j$ **do**
$L \leftarrow L \cup \{msg\}$

□ **If** $prepared(m, v, s, i)$ **then**
let $mm = \langle$COMMIT, $v, s, d_m, i \rangle_i$
multicast mm to R
$L \leftarrow L \cup \{mm\}$

□ **If** $committed(m, v, s, i)$ **and** $(s = s_{last} + 1)$ **then**
process $m = \langle$GROUP-MESSAGE, $cs, —, a\rangle_c$
$s_{last} \leftarrow s$
send $\langle$GROUP-REPLY, $v, cs, reply, i\rangle_i$ **to** c

□ **Upon** *timer* expires **do**
start *view-change*$(v+1)$

Figure 4.6: Normal-case protocol for each replica i in view v.

is faulty; a faulty backup cannot prevent other replicas from making progress, because all the $3f + 1$ replicas are active in their algorithm. In our algorithm, not only the faulty primary, but also a faulty backup can stop the system from making progress. Hence, with our algorithm, a faulty backup that blocks progress may also trigger a view change.

The purpose of the view-change procedure is to remove the faulty replica from the active replica set and replace it with a new active replica (the spare). In order to do that, the two fault-free active replica must initiate the view-change procedure and agree on the removal of the faulty replica. If a client message has been processed by at least one fault-free active replica, the algorithm must guarantee that after the view-change, other fault-free active replicas process the message in the same order. This requires that prepared certificates and committed certificates (as described in the previous subsection) are passed to the new view. In addition, the view-change procedure must ensure that the new active replica obtains correct server state. This can be achieved by having two active replicas send consistent states to the new active replica.

The view-change protocol in our algorithm moves the system into view $v + 1$ from the current view v, thus changing the current primary to standby spare, promoting a backup to be the primary, and bringing in the spare as a new active replica.

In order to initiate the view-change, correct replicas must find out that they are blocked. As in [Cast99a], a timeout mechanism is used. As mentioned earlier, a

client re-sends its group message to all replicas if it does not receive valid responses within a timeout period. Hence, if the primary is faulty and it does not forward the client message to the backups, the client will eventually retransmit the message directly to the backup replicas. Therefore, every replicas eventually receives the client message and it waits for processing the message until the three-phase protocol completes on the message, as described in the previous subsection. In order to prevent replicas from waiting indefinitely, each replica (include the primary) starts a timer with a timeout period when it receives a client message. If the timer expires and it has not processed the message, the replica starts the view-change protocol as described below.

There is no need for a replica to start a timer for every client message it receives. Instead, the replica only needs *one* timer. It starts the timer upon receiving a client message only if the timer has not already been started due to previous client messages. When the timer expires, a replica restarts the timer if it still has client messages waiting for processing. It stops the timer when it has processed all the client messages it has received.

Because it is impossible to identify which server replica (if any) is faulty, the view-change protocol always ‘‘suspects’’ the primary and removes it from the active replica set. The suspicion could be wrong, so the faulty replica may remain in the active replica set and continue blocking the ordering protocol. However, if that happens, another view-change is triggered, as described above. Since the role of the primary rotates among the four replicas, after at most three consecutive view-

changes, the faulty replica is removed from the active replica set. The correct replicas can then make progress.

The view-change protocol must ensure correct processing of group messages across views. When a standby spare replica becomes active as a result of a view change, it must obtain a state consistent with the state of the other replicas, as well as information about group messages that have been prepared at correct replicas. This is achieved by having two active replicas agree on the view-change and on a consistent state. It is possible that at the moment a group message have not been processed by all replicas, as some replicas have received all the commit messages (see the previous subsection) but others have not. In order for the replicas to reach agreement on their state, they must first reach agreement on a common position of processing group messages, i.e. the RSN of the last group message processed. Hence, the replicas exchange their last RSNs and states until two replicas reach an agreement on a common RSN and a consistent state. *All* the active replicas are supposed to participate in this procedure. Since at most one of the participants in this procedure is faulty, at least two participants are non-faulty. Hence, if agreement is reached by at least two replicas, the agreed-upon state must be consistent with a non-faulty server replica. If agreement is not reached between two replicas, then one of them must be faulty, and the correct one will reach agreement with another replica that is also correct. Therefore, such an agreement will be reached and the view-change will always proceed.

After reaching the agreement, at least two active replicas send the agreed-upon

state to the spare replica along with messages that prove its correctness, and any existing prepared certificates. Once the spare replica verifies and restores the state, the system can move into the new configuration.

When a timeout triggers the view-change on replica i in view v, it switches its operation state (mode) to "*view-change*." It continues to process normal messages as in *normal operation* mode. In addition, it multicasts a *view-change* message to all other active replicas. The message has the form $\langle$VIEW-CHANGE, $v+1$, s_i, $i\rangle_i$, where s_i is the receive sequence number (RSN) of the last message that has committed at i.

As described earlier, all active replicas start their own timers on processing a client message. If a faulty replica blocks progress, timeouts *independently* trigger view-change from view v to view $v+1$ on all correct replicas. Hence all correct replicas will eventually begin the view-change procedure. In order to prevent a faulty replica from incorrectly causing view-change on a correct replica, when a replica j receives a $\langle$VIEW-CHANGE, $v+1$, s_i, $i\rangle_i$ message, if it has not started the view-change to view $v+1$, it ignores the message. Otherwise, if the RSN of the last message it has committed to is higher than or equal to s_i, it sends back to i a $\langle\langle$VIEW-CHANGE-ACK, $v+1$, s_j, D_j, Φ, $j,\rangle_j$, $\Psi\rangle$ message, where s_j is the RSN of the last message that has committed at j, and D_j is the digest or checksum of its management state after it executes that message. The set Φ contains a set Φ_m for each group message m that prepared at j with a RSN higher than s_j, i.e., each group message that has prepared but not committed yet at j. Each Φ_m is a set

containing a valid pre-prepare message for group message m and two matching, valid prepare messages signed by different replicas with the same view, RSN and digest, i.e., the prepared certificate of m. The piggybacked Ψ is a set of Ψ_m for each group message m that has a RSN s, $s_i < s \leq s_j$. Ψ could be a null set if $s_i = s_j$. Each Ψ_m is the committed certificate of m.

When replica i receives from others a valid $\langle\langle$VIEW-CHANGE-ACK, $v+1$, s_j, D_j, Φ, $j\rangle_j$, $\Psi\rangle$ from any replica j, if it has not executed a group message with a higher RSN than s_j, it add the commit messages in each Ψ_m of Ψ to it log and executes all the group messages up to s_j provided that the committed certificate is valid. It then computes the digest of its state and compares it to D_j. If the digest are the same, replica i sends a *new-view* message to the standby spare. The new-view message has the form $\langle\langle$NEW-VIEW, $v+1$, s_i, D_i, Φ, $i\rangle_i$, S, V $\rangle$, where s_i is the RSN of the last group message it has committed (should be equal to s_j), D_i is i's state digest it just computed. S is the complete state of i corresponding to the digest D_i. The set Φ contains prepared certificate Φ_m for each group message m that prepared at i but not yet committed at i. The attached V is the view-change-ack message received from j, without the piggybacked Ψ. This view-change-ack message is used as a justification to the new-view message.

The standby spare in view v accepts a new-view message for view $v+1$ from any other replica provided: both the new-view message and the piggybacked view-change-ack message are properly signed and contain the same sequence number

and state digest; the state S matches the digest. It then restores its state from S and go through the Φ sets in both the new-view message and the piggybacked view-change-ack message. If there is at least one valid certificate Φ_m in the two Φ sets for any group message m, it adds the messages in Φ_m into its log. This ensures the predicate *prepared* to be also true at the spare for group message m, its sequence number s and view v. Once it has finished, it relays the new-view message and the view-change-ack message (without the piggybacked S) to all other replicas and installs view $v + 1$.

When a replica in view v receives the relayed new-view message for $v + 1$ from the spare replica, it first checks the validity of the message and the piggybacked view-change-ack message. If they are valid, the replica accepts them and installs view $v + 1$. The view-change protocol is presented formally in Figure 4.7.

Completion of the view-change protocol relies on the correct behavior of the spare replica. The spare replica may actually be faulty and may block the view-change. However, the single fault assumption means that if the spare is faulty none of the current active replicas are faulty. If none of the active replicas are faulty, the view-change was ‘‘incorrectly’’ triggered by premature timeouts. Since the active replicas are not really faulty, they do eventually complete the normal case operation and pending group messages are committed and executed. When they have no group message left for processing, the active replicas abort the view-change and switch back to normal mode. Therefore, either the normal-case operation, or the

Variables

$R \equiv \{v \bmod 4, (v+1) \bmod 4, (v+2) \bmod 4\}$ /* the active replica set */
$primary \equiv v \bmod 4$
$B \equiv \{(v+1) \bmod 4, (v+2) \bmod 4\}$ /* the backup set */
$spare \equiv (v+3) \bmod 4$
s_{last} /* the RSN of the last group message processed */
L /* the message log */
S /* the management state */
$D(S)$ /* digest of the management state */
$mode$ = NORMAL /* operation mode */

Predicates

$prepared(m, v, s, i) \equiv m \in L \wedge \langle$ PRE-PREPARE, $v, s, d_m \rangle_{primary} \in L \wedge$
$\forall r \in B : (\langle$PREPARE, $v, s, d_m, r\rangle_r \in L)$
$committed(m, v, s, i) \equiv prepared(m, v, s, i) \wedge \forall r \in R : (\langle$COMMIT, $v, s, d_m, r\rangle_r \in L) \wedge$
$\forall s', s' < s : (\exists v', m' : (v' \leq v) \wedge committed(m', v', s', i))$

□ **Upon** CALL *view-change*$(v+1)$ **do**
$mode \leftarrow$ VIEWCHANGE
multicast $\langle$VIEW-CHANGE, $v+1, s_{last}, i\rangle_i$ to R

□ **Upon** RECEIVE $\langle$VIEW-CHANGE, $v+1, s_j, j\rangle_j$ **do**
if ($mode$ = VIEWCHANGE) $\wedge$ $(s_{last} \geq s_{j)}$ **then**
let $\Phi = \{prepared\text{-}certificate(m, s) \mid \forall m : prepared(m, v, s, i) \wedge (s \geq s_{last})\}$
and $\Psi = \{committed\text{-}certificate(m, s) \mid \forall m : committed(m, v, s, i) \wedge (s_j < s \leq s_{last})\}$
send $\langle\langle$VIEW-CHANGE-ACK, $v+1, s_i, D(S), \Phi, i, \rangle_i, \Psi\rangle$ to j
else buffer *msg*

□ **Upon** RECEIVE $\langle\langle$VIEW-CHANGE-ACK, $v+1, s_j, D_{j,} \Phi, j, \rangle_j, \Psi\rangle$ **do**
if $s_{last} < s_j$ **then**
$\forall m, s : (s \leq s_j) \wedge (committed\text{-}certificate(m, s) \in \Psi) \Rightarrow$ process m**,** update s_{last}
if $D(S) = D_j$ **then**
let $\Phi = \{prepared\text{-}certificate(m, s) \mid \forall m : prepared(m, v, s, i) \wedge (s \geq s_{last})\}$
and $V = \langle$VIEW-CHANGE-ACK, $v+1, s_j, D_{j,} \Phi, j, \rangle_j$
send $\langle\langle$NEW-VIEW, $v+1, s_{last}, D(S), \Phi, i\rangle_i, S, V\rangle$ to *spare*

□ **Upon** RECEIVE $\langle\langle$NEW-VIEW, $v+1, s_j, D_j, \Phi_j, j\rangle_j, S_j, \langle$VIEW-CHANGE-ACK**,**
$v+1, s_k, D_k, \Phi_k, k, \rangle_k\rangle$ **do**
if $(i = spare) \wedge (s_j = s_k) \wedge (D_j = D_k = D(S_j))$ **then**
$S \leftarrow S_j$
$\forall m, s : ((prepared\text{-}certificate(m, s) \in \Phi_j) \vee (prepared\text{-}certificate(m, s) \in \Phi_j) \Rightarrow$
$L \leftarrow L \bigcup \{ prepared\text{-}certificate(m, s)\}$
multicast $\langle\langle$NEW-VIEW, $v+1, s_j, D_j, \Phi_j, j\rangle_j, \langle$VIEW-CHANGE-ACK**,**
$v+1, s_k, D_k, \Phi_k, k, \rangle_k\rangle$ to R
$v \leftarrow v+1$**,** $mode \leftarrow$ NORMAL

□ **Upon** RECEIVE $\langle\langle$NEW-VIEW, $v+1, s_j, D_j, \Phi_j, j\rangle_j, \langle$VIEW-CHANGE-ACK**,**
$v+1, s_k, D_k, \Phi_k, k, \rangle_k\rangle$ from *spare* **do**
$v \leftarrow v+1$**,** $mode \leftarrow$ NORMAL

Figure 4.7: View-change protocol for each replica i in view v.

view-change procedure, will eventually proceed, preventing the algorithm from blocking indefinitely.

After successfully changing to view $v+1$, the primary replica p of the new view first creates its own $\langle$PRE-PREPARE, $v+1$, s, $d_m\rangle_p$ message for every group message that has prepared in previous views but not yet committed at p, using the same RSN and message digest, in consecutive order. If there is a gap in the pre-prepare message set, p creates a special pre-prepare message $\langle$PRE-PREPARE, $v+1$, s, $d_{null}\rangle_p$, where d_{null} is the digest of special *null* group message. A null group message is processed like other group message but invokes no real operation. After this step, the primary switches its state back to ''*normal*'' and starts the ordering protocol for new group messages as in Section 4.2.2.

The primary replica of the previous view v becomes the spare in view $v+1$. It discards its state and cleans up its message log, then enters standby mode. Other replicas simply go back to normal state after the view change and proceed as described in Section 4.2.2.

After changing to view $v+1$, a backup replica may see a gap in the sequence numbers, from the last message it has committed to in the previous view to the first pre-prepare message it receives in the new view. In that case, it asks the primary to send the pre-prepare messages again for the missed sequence numbers, using the view number $v+1$.

4.2.4. Garbage Collection

As described before, a replica must keep messages (client messages, pre-prepare, prepare and commit messages) in its message log, in case they are needed for retransmission or as certificates during view-change. The replica can remove messages from its log when it knows that the client messages associated with these messages have been processed by all active replicas. To diffuse this kind of information, each replica has the RSN of the last client message it has processed piggybacked on the heartbeat messages it sends to other replicas periodically. When a replica sees such a RSN s sent by all other active replicas and it has processed the client message corresponding to s, it deletes all messages associated with a RSN less than or equal to s from its log.

With this garbage collection mechanism, a faulty replica can block the log truncation on other replicas by not reporting the actual RSN it has processed. When a replica's message log is full, the replica pauses the ordering protocol for any new client messages. After a while, the view-change protocol will be invoked and eventually the faulty replica will be demoted to be the spare.

The garbage collection mechanism can be improved with *stable checkpoints* as described in [Cast99a]. A replica takes checkpoints of its state when a client message with a sequence number divisible by some constant has been processed. A checkpoint is considered stable if it has a proof for its correctness. When replica i creates a checkpoint, it multicasts to other active replicas a *checkpoint* message $\langle \text{CHECKPOINT}, s, D_i, i \rangle_i$, where s is the RSN of the last client message that i has

processed before taking the checkpoint, and D_i is the digest of the state. When a replica has collected at least two checkpoint messages for sequence number *s* with the same digest *D* signed by different replicas (including its own checkpoint message), it saves the checkpoint as a stable checkpoint and the messages as the proof of correctness for the checkpoint. Because correct replicas have the same state after the processing of a client message and only one replica could be faulty, a checkpoint with at least two consistence checkpoint messages as proof must be correct.

When a replica gets a stable checkpoint, it removes from its log all client messages, pre-prepare, prepare and commit messages with a RSN less than or equal to the RSN reflected in the checkpoint. It also discards all earlier checkpoints. If during a view-change, it is required by the algorithm to send the commit messages with a RSN lower than RSN of its stable checkpoint, it sends the stable checkpoint with its proof instead (in piggybacked set Ψ). It still has to send in Ψ the messages with a RSN higher than that reflected in the checkpoint.

4.2.5. Correctness Proof

In this section we sketch the proofs of safety and liveness for the three-phase BFT-SMR algorithm. As stated before, these proofs rely on the single fault assumption and partial synchrony assumption.

4.2.5.1. Safety

The algorithm provides safety if all non-faulty active replicas agree on the same total order of client messages they commit. The proof of safety is by induction on views; the following lemmas are the keys to the induction step:

Lemma 4.1. *In any view v, if $prepared(m, v, s, i)$ is true for a correct active replica i, then $prepared(m', v, s, j)$ cannot be true for any correct active replica j and any client message m' such that $d_m \neq d_{m'}$.*

Proof. $Prepared(i, v, m, s)$ and $prepared(j, v, m', s)$ being true implies that all replicas has sent conflicting pre-prepare or prepare messages, i.e., pre-prepare/prepare messages with the same RSN s but different message digests (d_m or $d_{m'}$) in view v. This is impossible because correct primary replica only sends one pre-prepare message with a RSN in view v, and a correct backup replica only accepts one pre-prepare message and send prepare message with a RSN in view v. Since only one replica can be faulty, there cannot be prepared certificates for both $\langle m, s \rangle$ and $\langle m', s \rangle$ in view v. Therefore, $prepared(m, v, s, i)$ and $prepared(m', v, s, j)$ cannot be both true if $d_m \neq d_{m'}$. □

The pre-prepare and prepare phases in the normal case protocol guarantee that all correct active replicas prepared messages with the same digest and same RSN within a view. The assumption on the message digest function ensures the probability that $m \neq m'$ but $d_m = d_{m'}$ is negligible. Therefore, it is not possible that correct active replicas prepare to different client messages with the same RSN in a view.

Lemma 4.2. *A client message m commits at some correct replica with RSN s in view v only when prepared*(m, v, s, i) *is true for every correct active replica i in view v.*

Proof. The proof is straightforward: A correct replica can commit to $\langle m, s \rangle$ only when it has received commit message for $\langle m, s \rangle$ from every active replica in view v, and a correct replica i only sends it commit message for $\langle m, s \rangle$ when *prepared*(m, v, s, i) is true. □

The commit phase in the normal case protocol ensures that a client message commits at all correct replica in the same view with the same RSN. The view-change protocol ensures that correct active replicas also agree on the RSNs of client messages even if they committed in different views at different replicas:

Lemma 4.3. *If prepared*(m, v, s, i) *is true for all correct active replica in view v, then prepared*$(m, v+1, s, i)$ *is also true for all correct active replica in view* $v+1$.

Proof. The view $v+1$ will not be installed unless two active replicas in view v agree on the view-change and the spare replica receives both messages (*new-view* and *view-change-ack*) from these two replicas. At least one of the two replicas must be correct; the set Φ in its *new-view* or *view-change-ack* then must contains prepared certificates for all client messages that has prepared at this replica. The spare replica accepting these prepared certificates ensures that *prepared*(m, v, s, k) is also true for the spare replica k. Therefore, the fact that m prepared at every correct active replica in view v is propagated to view $v+1$. □

By induction on views, we have:

Theorem 4.1. *The algorithm satisfies Safety.*

The algorithm ensures that correct replicas commit and process client messages in the same order even if they commit across different views.

The propagation of preparation from a previous view to new views does not prevent the replicas from making progress. The primary of a new view (might be the one that a client message prepared at) will redo the ordering protocol using the same sequence number and the same client message digest if it is not faulty. If it was faulty and did not follow the prepared order, its pre-prepare would not be accepted by other replica and they would change to the next view and the prepared client message order will commit eventually.

4.2.5.2. Liveness

Because the normal-case protocol does not provide liveness when there is a faulty replica, the view-change must be conducted to move the replicas into a new view if they are not making progress. Eventually, the replicas are in a view where all active replicas are correct. The timeout mechanism we use ensures this, as long as the clock of each replica does not drift unboundedly with respect to the real time.

However, we must prevent the situation that the replicas keep doing view change, move from one view to another before they execute client messages, even when they are in a view where all active replicas are correct. It is impossible to prevent it in a pure asynchronous system, where messages can be infinitely delayed

or processes can be infinitely slow. Thus our algorithm must rely on partial synchrony to provide liveness.

Theorem 4.2. *The algorithm satisfies Liveness, under the assumption of partial synchrony.*

Proof. (sketch.) We consider two models of partial synchrony. One is a model introduced in [Dwor88] (denoted M_2 in [Chan96a]), in which there are known bounds on processing and message transmission times, but they hold only after some unknown time, called *Global Stabilization Time* (GST). With this model, if we set the timeout properly based on the known bounds, the algorithm can ensure the execution of client messages if the bounds hold long enough after GST.

In a weaker model, described and denote M_3 by Chandra and Toueg in [Chan96a], the bounds exist but are unknown, and hold only after some unknown GST. With this model, we need to adjust the timeout period to avoid premature timeouts. Initially, the replicas set their timeout period for pending client messages to a time T. If after a cycle of configurations (i.e. after four consecutive view-changes), they do not make any progress on processing client messages, they increase the timeout interval by T. Eventually, the timeout periods will be long enough to bear with the unknown bounds on processing and message transmission times. □

The faulty replica is unable to block the service by forcing frequent view-changes. It cannot directly force a view-change because a view change must be agreed by at least two replicas; so it can cause a view change only by blocking the

the ordering protocol when it is an active replica. However, it cannot be always in the active replica set for more than four consecutive views.

4.2.6. Requirement for Three Phases with 2f+1 Active Replicas

As in Castro and Liskov's algorithm [Cast99a], our algorithm also requires three phases for normal processing. As discussed in Section 4.1, with $3f + 1$ active replicas, the third phase —the commit phase —is necessary since it ensures that a group message commits at a correct replica only if the message has been ''prepared'' by at least two correct replicas. This guarantees safety across view-changes by ensuring that a message prepared in a view is always propagated to subsequent views with the same order.

Based on arguments similar to those presented above, three phases are also needed with our algorithm that uses $2f + 1$ active replicas. Specifically, if the third phase is eliminated, the result is that *prepared*(m, v, s, i) is equal to *committed*(m, v, s, i). Hence, a group message may be committed at a correct replica but its ordering information may not be propagated to subsequent views. Thus, the system may ultimately commit to a different ordering in the new view. This situation can occur *only* if the replica that committed to the ordering is the only correct replica that has completed the second phase *and* this replica is also the replica that is removed from the active replica set, i.e., it is the primary of the current view.

In the scenario described above, the primary replica in the current view

becomes a standby spare and loses its state in the next view. This, the different commitments described above do not cause inconsistency among the active replicas in the new view. Thus, the agreement on total order of group messages among active replicas is ensured even with the two-phase algorithm (Figure 4.3). However, without changes to the behavior of clients, this two-phase algorithm may cause inconsistency between the server replicas and a client. Consider the following case: a message m sent by client c completes the second phase on replica 0 (the primary) and replica 1, so both replicas process m and send their replies to c. Replica 1 is actually faulty but it sends the correct reply to c, which matches the reply from replica 0. Thus, c accepts the reply. The faulty replica 1 does not send its prepare message to replica 2 though, so replica 2 cannot complete the second phase and process m in the current view v. A timeout then triggers view-change on replica 2 and the faulty replica 1 agrees on the view-change, so the replicas move into view $v+1$ in which replica 1 becomes the primary. The standby replica 3 does not learn about the commitment of m on replica 0 during the view change. Therefore, the faulty replica 1 may successfully lead message m to commit at replica 2 and 3 with a different sequence number by sending a different pre-prepare message for m. This causes the correct replicas to proceed with a state that are inconsistent with the reply observed by client c. This violates the safety requirement on linearizability (see Section 2.1.3).

Due to the arguments above, our replication algorithm that uses $2f+1$ active replicas still requires three phases.

4.2.7. Extension of the Algorithm to Multiple Faults

So far, we have presented the details of our algorithm for the case of $f = 1$, i.e., when there is at most a single faulty replica. The algorithm is easily extended to tolerate multiple simultaneous faults. However, with $f > 1$, the view-changes can become expensive.

During a view-change, the algorithm requires that at least $f + 1$ active replicas agree on the view-change and on the state that is transferred to the new active replica. Faulty active replicas may block progress and cooperate with fault-free active replicas to incorrectly replace a fault-free primary. In the worst case, the system has to iterate through all the possible configurations via view-changes, before it reaches a ‘‘clean’’ configuration in which all active replicas are fault-free. The average time to reach it grows rapidly when f increases.

With a total of $3f + 1$ replicas, the configuration of each view consists of $2f + 1$ active replicas and f standby spares. The number of possible configurations is the binomial coefficient $\binom{3f+1}{f}$. Among all these configurations, only one can be guaranteed to consists of only fault-free replicas as active replicas. With a configuration that has only $2f + 1$ active replicas, even one faulty replica can block the ordering protocol until a view change. Therefore there is only one configuration that can ensure that the replicas make progress.

An alternative to the algorithm above is to increase the number of active replicas in each configuration to $3f$. This leads to only $3f + 1$ possible configurations. Among them there is one configuration with a faulty replica being

the standby spare. This configuration can assure progress because it includes at least $2f + 1$ non-faulty active replicas. Therefore, the system needs at most $3f$ view changes to get to this configuration. Thus, there is a tradeoff between the number of active replicas and the amount of time a group message may have to wait while the manager replicas go through a sequence of view changes.

4.2.8. Extension of the Algorithm to Open Replica Groups

So far, our BFT-SMR algorithm has been presented in the context of a *closed* replica group, in which the overall set of replicas does not change. Since a faulty replica stays in the replica group when it is removed from the active replica set, it may re-join the active replica set after additional view-changes. However, the algorithm can be easily used in the context of an *open* replica group, where the group membership changes over time. In particular, each view-change may instantiate an active replica on a *new* node that has never been used by an active replica before. Thus, a faulty replica is never ‘‘cycled back’’ into the set of active replicas. With our algorithm, such open group operation is simple since bringing a spare replica into the active replica set during a view change already involves initializing the new replica with up-to-date state from fault-free active replicas. It should be noted that a similar version of open group operation can also be derived from Castro and Liskov’s algorithm with ‘‘proactive recovery’’ [Cast00].

Since the number of physical nodes in the system is limited, the open group operation discussed above will eventually run out of new nodes on which to execute

replicas. However, in many situations, bringing a new replica into the active set may simply involve starting a new *process* as opposed to using a new physical nodes. Since most faults are transient, the new replica process is unlikely to be faulty even if it is instantiated on a node that previously executed a failed replica.

4.3. Evaluation

With fewer active replicas during normal operation there is a reduction in computation and communication overhead. Although the required number of replicas is $3f + 1$, as in existing schemes [Cast99a], the standby replicas stay dormant most of the time. Thus, the machines they reside on can be used for other applications. Alternatively, the hardware components (processors, memory, disks, etc.) can be turned off to reduce power consumption through dynamic power management (DPM) [Beni00]. Since the spare replicas are activated only on reconfigurations, which do not occur often, there would be little overhead activating and deactivating the spare replicas.

Another overhead reduction with our scheme is that with fewer active replicas, each replica sends and receives fewer messages during normal operation. This leads to reduced power consumption for communication and may result in better performance for the normal case. To evaluate this advantage, we used a simplified implementation (emulation) to compare the normal-case performance of our algorithm to the algorithm described in [Cast99a]. In both cases a single faulty replica can be tolerated and the total number of replicas is four. The evaluation was

performed on a network of 350MHz Pentium-II PCs, running Solaris 8, interconnected by a 100Mb switched Ethernet, using TCP/IP for communication. The service operation is a computation that executes in a set amount of time.

For the two algorithms we measured the average response time for requests and the throughput of the service under fault-free operation. The response time is the time interval from when a client sends its request to the primary replica to the time when it receives replies from two different replicas. The response time was measured with the system processing only one request at a time. The throughput is the number of requests per second the replicas are able to process. To show the overhead of the replication algorithms, the results are normalized to the results we measured for the same unreplicated service.

The message authentication in our experiments is based on 512-bit RSA moduli and MD5 digests using the OpenSSL 0.9.7b package. On our PCs, it takes 6.2 milliseconds to generate an RSA signature of an MD5 digest and 0.5 milliseconds to verify a signature. The generation of MD5 digest of a 1KB message takes 30 microseconds.

Figure 4.8 shows the results for a service that takes 1 millisecond to execute each request. The results show that with the full cost of authentication (‘‘1x’’), there is not much difference in response time overhead between the two replication algorithms. The reason for this is the high overhead of generating RSA signatures. Although each replica in our algorithm sends and receives fewer messages per request, it performs the same number of multicasts as in the 4-replica algorithm.

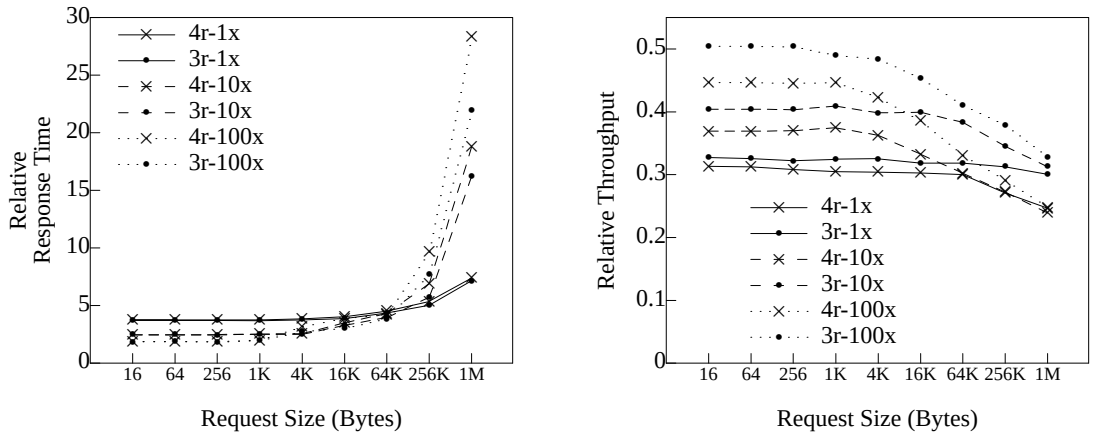


Figure 4.8: Response time and throughput relative to the unreplicated case, as request size and message authentication overhead vary. ‘‘4r’’ is the algorithm that uses four active replicas [Cast99a] while ‘‘3r’’ is our algorithm with three active replicas. Execution time for each request is 1 millisecond. Authentication overhead with speedup of 1x, 10x, or 100x relative to measured overhead.

RSA signatures are only generated once for each multicast and for the reply to the client. Hence, the two algorithms have the same overhead for signing messages. Although the 3-replica algorithm reduces the overhead for sending, receiving and verifying messages, the benefit is insignificant compared to the overhead of signing. In terms of throughput, with the full cost of authentication, our algorithm results in slightly higher performance due to the reduction in the number of messages exchanged.

If fast hardware implementations of the RSA algorithm [Blum01] is used, or faster but less secure cryptography can satisfy the requirement of the service, the overhead for message authentication is reduced. Under these conditions the overhead of signing messages is no longer the dominant factor and there is more of

a benefit to using fewer active replicas. Figure 4.8 shows the response time and throughput results with authentication that is ten and a hundred times faster. Under these conditions, the overhead of signing messages is no longer the dominant factor and there is more of a benefit to using fewer active replicas.

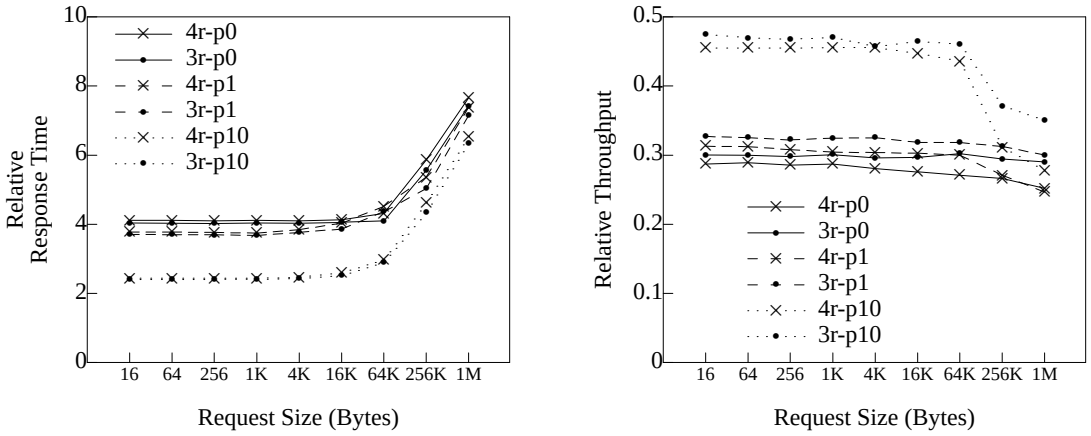


Figure 4.9: Response time and throughput relative to the unreplicated case, as request size and execution time for requests vary. ‘‘4r’’ is the algorithm that uses four active replicas while ‘‘3r’’ is our algorithm with three active replicas. The message authentication overhead is the actual measured overhead on our system. The execution time in the figure is in milliseconds, e.g., "-p1" means the execution time for each request is 1 millisecond.

Figure 4.9 shows a performance comparison between the two algorithms with various request execution times. The ‘‘-p0’’ results are presented to simulate the cases that the request execution time is insignificant. It would also represent the scenario that the execution is performed on other nodes, separated from the agreement. As the request execution time increases, the performance advantage of the 3-replica algorithm is reduced.

Chapter Five

Self-Diagnosis and Reconfiguration

With the replication algorithm presented in Chapter 4 as well as the original Castro and Liskov's algorithm [Cast99a], the goal is to have the replicas proceed with correct operation despite the failure of *up to* f replicas. If nothing else is done to recover the faulty replicas, eventually the number of faulty replicas exceeds f and these algorithms will fail. These algorithms guarantee correct behavior only if no more than f replicas are faulty during the lifetime of the system. This is unacceptable for long-lived practical systems, hence, a recovery mechanism is needed to replace the faulty replicas with fault-free replicas.

It is possible to do the recovery without knowing which replica is actually faulty. For example, the proactive recovery mechanism [Cast00, Zhou02] periodically rejuvenated replicas even though there is no reason to suspect them as faulty. With this mechanism, faulty replicas are eventually ''repaired,'' thus restoring the system to its original resiliency. However, proactive recovery incurs unnecessary cost since fault-free replicas are also rejuvenated. In addition, this mechanism can not recover from permanent hardware faults, and it requires a trusted component on each replica to ensure the initiation of the rejuvenation (for example, the watchdog timer used in [Cast00]).

Another example of repair or replacement of faulty replicas is the use of an open replica group version of the replication algorithm, as discussed in Section

4.2.8. With an open replica group, the algorithm always replaces the primary replica with a new, fault-free replica, without knowing whether the primary is the faulty one or not. Since every replica becomes the primary after all replicas that joined the group before it have been replaced, faulty replicas will be replaced eventually.

With any recovery mechanism, the system has a *window of vulnerability*, which is the period from the time that f replicas are faulty to the time that some faulty replica is replaced by a fault-free replica. During this time period, the system is ‘‘vulnerable’’ because the failure of another replica may cause the entire system to fail. If this window of vulnerability is short relative to the time interval between the failure of any two replicas, the system can theoretically survive forever despite the fact that replicas continue to fail.

The window of vulnerability can be minimized if faulty replicas can be identified and targeted for repair instead of wasting time ‘‘repairing’’ (e.g., replacing) replicas that are not faulty. This motivates the use of fault diagnosis [Prep67, Barb93] mechanisms to identify faulty replicas. This chapter is focused on the development of such a mechanism and its use for reconfiguration with the replication algorithm presented in Chapter 4.

In Section 5.1, we describe a state machine fault model for diagnosis. Based on this fault model, we present a self-diagnosis protocol in Section 5.2. Under the assumptions of the fault model, diagnosis may not be both complete (all faulty replicas are identified) and accurate (no correct replicas is declared faulty). Unlike

most other fault diagnosis solutions, we choose to favor completeness over accuracy due to the relatively low cost of inaccurate diagnosis with replicated state machines —replacement or repair of a fault-free replica.

Once the diagnosis procedure diagnoses a replica as faulty, the reconfiguration procedure will remove it from the replica group and replace it with a fault-free replica. The reconfiguration protocol is described in Section 5.3.

5.1. A State Machine Fault Model for Diagnosis

System-level fault diagnosis is based on the ability of processing elements (PEs) *testing* other PEs to uncover faults [Prep67]. In order to ensure that faulty PEs can be diagnosed, tests need to have a perfect coverage; otherwise, faulty PEs could go undiagnosed for a long period of time. An arbitrarily faulty PE may behave correctly during the diagnosis but generate incorrect outputs during normal operation. If this happens, the diagnosis test cannot uncover this faulty PE. Hence, no diagnosis procedure can have perfect coverage with respect to all possible faults [Shin87]. Therefore, the diagnosis procedure must assume a more restricted fault model in which the behavior of faulty PEs is limited. In this section, we define and justify such a restricted fault model for systems with state machine replication.

A state machine can be tested either by full state machine testing or by state validation. With full state machine testing, a state machine is tested with a given initial state and a given input stream. The state machine initializes to the state and

runs with the inputs. Its outputs and, possibility, its final state are examined to determine whether it is faulty. This testing disrupts the normal operation of the state machine. On the other hand, state validation does not require disrupting normal operation of the state machine. The state machine runs normally and at various points its internal state is examined. State validation requires some mechanism to determine whether the internal state is correct. This is, essentially, an acceptance test on the internal state. With state machine replication, this can be done by comparing the internal state of the replica under test to the state of other correct replicas.

Full state machine testing requires exercising every state transition with every possible input. For a complex state machine, such as Ghidrah's cluster manager, such testing is not feasible. The purpose of the diagnosis procedure is to identify a replica that is likely to operate incorrectly in the future. It is possible for a permanent hardware fault to cause the state machine output to be incorrect even though the internal state is correct. Hence, ideally, such permanent hardware faults should be identified during diagnosis. However, permanent hardware faults are much less likely to occur than transient hardware faults [Cast82]. Other transient faults, such as software synchronization faults, are also more likely to occur than permanent hardware faults [Gray86]. When a transient fault affects a state machine, it only increases the probability of *future* incorrect outputs if it causes the internal state to be corrupted. Identifying corrupt state is thus the key goal of the diagnosis procedure. Hence, testing based on validating the internal states of the replicas is the appropriate foundation for the diagnosis scheme.

The purpose of the diagnosis procedure is to identify a replica that is likely to produce incorrect outputs. As discussed above, this can be done by identifying a replica with incorrect internal state. During the diagnosis procedure, each replica must send its internal state to other replicas, receive state from other replicas, and perform acceptance (validation) test on the state received from other replicas. A replica can perform the validation test by comparing the received state with its own state.

A fault in a replica may prevent it from completing all the steps of the diagnosis procedure correctly even though its internal state is correct. To facilitate the discussion of the diagnosis procedure, we partition the functionality of the replica into two modules: the *state machine module* (SM) and the *diagnosis module* (DM). The SM performs the replica functions during normal operation and computes the internal state. During diagnosis, the DM in each replica performs all communication with other replicas. The DM can access the state of the local SM and send it to other replicas. It is also responsible for checking the correctness of state received from other replicas.

The SM is faulty if it generates an incorrect internal state:

- *SM fault*: the SM of the replica has an incorrect state that is inconsistent with the specification of the replicated state machine, given the known initial state and the input messages delivered to it.

The DM may exhibit two types of faults:

- *DM omission fault*: the DM omits sending a diagnosis message to one or

more correct replicas as the protocol requires.

- *DM commission fault*: the DM sends invalid diagnosis messages to one or more other replicas.

A replica is considered to be fault-free only if neither its SM nor its DM are faulty. The SM and DM may be subjected to crash failures. A crashed SM implies that the replica's internal state will not continue to change correctly over time. Hence, the internal state will quickly become incorrect. A DM that has crashed is the same as one that suffers from omission failure —it will fail to transmit the replicas state to others. Our fault model covers many failures that are not crashes. The assumptions of the model can be summarized as follows:

A1 A faulty SM eventually produces an incorrect internal state.

A2 If the internal state of the SM is incorrect, the DM, even if it is faulty, cannot send to others the correct state.

A3 Messages send by a correct DM are properly authenticated. A correct DM can correctly identify the sender of a message it receives and verify the authenticity of the message, even if the message has been relayed by others.

A4 The number of faulty replicas is no more than f at a time.

Assumption A1 implies that a complete acceptance test on the internal state, such as comparison with known correct state, will eventually yield perfect coverage. Assumptions A2-A3 limit the behavior of a faulty DM when it has commission faults. With these limitations, the commission faults can be correctly

detected by all correct replicas.

If a faulty replica exhibits only DM omission failures, it may be impossible to determine which replica is faulty. For example, if for a faulty replica r, its DM does not send its messages to f replicas, but sends correct messages to all other replicas, a correct replica that receives the messages is not able to decide whether replica r is faulty, or the f replicas that report r as faulty are actually faulty. In fact, with a pure asynchronous system model (no known bound on message latency), it is impossible for a correct replica to detect omission failures of r accurately. Specifically, there is no way to differentiate between the case that r is faulty (fails to send a message) and the case where r is correct and sends out the messages but the messages are delayed.

5.2. An Online Self-Diagnosis Algorithm

Based on the state machine fault model discussed in the previous section, we have developed a self-diagnosis algorithm that is applicable to replicated state machines in general, and to CMM manager replicas in particular. The goal of a diagnosis algorithm is to identify faulty replicas. A replica is considered faulty if its SM or its DM or both are faulty.

Ideally, the algorithm should eventually detect every failed replica (*completeness*, as defined in Section 2.3) and should never identify a fault-free replica as faulty (*accuracy*, as defined in Section 2.3). However, as explained in the previous section, in an asynchronous system it is not possible to guarantee both

completeness and accuracy. It is often assumed that inaccurate diagnosis, i.e., a correct replica is declared faulty, is unacceptable since it leads to unnecessarily eliminating valuable system resources. Hence, many previous diagnosis algorithms favor accuracy in the circumstance that completeness and accuracy cannot both be satisfied [Busk93, Shin87, Walt94].

With the state machine replication algorithm presented in Chapter 4, the consequences of inaccurate diagnosis are less severe than the consequences of incomplete diagnosis, i.e. where some faulty replicas are never identified. If some faulty replicas are never identified, they remain active and, eventually, the total number of faulty active replicas may exceed the limit (exceed f), resulting in incorrect service. When a replica is identified as faulty, it is replaced by a spare and it becomes a spare. While this involves some system overhead, normal operation can continue without interruptions. For the spare replica that is activated, the state is initialized to the state of correct replicas, thus repairing the results of any transient faults that may have previously affected it. Hence, the new replica is unlikely to be faulty. The replica that may have incorrectly been marked as faulty, becomes a spare. Eventually, it may be activated again after being ‘‘rejuvenated’’ by resetting its state. Hence, the consequences of inaccurate diagnosis are, at worst, equivalent to oblivious rejuvenation, as done by Castro and Liskov’s proactive recovery [Cast00]. Thus, the diagnosis algorithm presented in this section favors completeness over accuracy in almost all cases.

The simplest way to favor completeness over accuracy is to diagnose nodes as

faulty in a round-robin fashion. Thus with *n* replicas, every replica is diagnosed as faulty after *n* invocations of the procedure. This is essentially what is done by the algorithm presented in Chapter 4 as well as by proactive recovery [Cast00]. The potential benefit of a diagnosis algorithm that is ‘‘more accurate’’ than random choice is that it can reduce the window of vulnerability discussed in the previous section. Specifically, when something goes wrong (e.g., progress is blocked, triggering a timeout), the diagnosis algorithm is more likely than random choice to immediately identify the faulty replica, leading to the replacement of the faulty replica.

Since the diagnosis algorithm for replicas should favor completeness and the simple round-robin scheme achieves completeness, a reasonable requirement from a diagnosis algorithm is that it should not do worse than the round-robin scheme in terms of completeness (identifying faulty replicas as faulty). This desired diagnosis property (DDP), can be specified as follows:

DDP1 The maximum number of invocations of the diagnosis algorithm required to identify a faulty replicas as faulty should not be larger than the worst case number of invocations of the round-robin algorithm required to identify a faulty component as faulty.

There are two ways in which the diagnosis algorithm might be expected to do better than simple round-robin:

DDP2 The diagnosis algorithm will identify faulty replicas as faulty faster —after fewer invocations of the algorithm.

DDP3 The diagnosis algorithm will identify correct replicas as faulty less often (be ‘‘more accurate’’).

There is a conflict between DDP1 and DDP3 (completeness versus accuracy). As discussed in the previous section, a faulty replica may behave correctly during diagnosis. Hence, a diagnosis algorithm that reaches a diagnosis based on the behavior of the replicas, may complete without identifying *any* replica as faulty despite the fact that the system does contain faulty replicas. In order to strictly favor completeness, the algorithm should ensure that DDP1 is satisfied. Hence, a final step should be added to the the diagnosis algorithm so that it will pick a ‘‘victim’’ in round robin fashion if no replica has been identified as faulty up to that point. As an engineering tradeoff, this final step may not be included if the diagnosis algorithm without this final step very rarely violates DDP1 and this choice greatly improves accuracy (DDP3). This latter alternative may be chosen if, in practice, it is common for diagnosis to be triggered even if none of the replicas are faulty (e.g., due to a temporary interruption in communication).

Before describing the details of our algorithm, we discuss its basic properties. First, we present the conditions under which an invocation of the algorithm will identify all the faulty replicas in the system. The first condition deals with the network:

- During diagnosis, all messages to fault-free replicas that are injected into the network, arrive at their destination within a timeout period.

In addition to the condition above, at least one of the following conditions must

hold for *every* faulty replica:

- the SM of the replica is faulty and at the time of the invocation of diagnosis, it already has an incorrect internal state.
- the DM of the replica exhibits omission faults to *all* fault-free replicas.
- the DM of the replica exhibits commission faults to at least one fault-free replica.

Under these conditions the diagnosis is also accurate: correct replicas will not be identified as faulty.

If the conditions listed above do not hold during the diagnosis, our algorithm may not be able to correctly identify the faulty replicas. One example, as discussed earlier, is if a DM of a faulty replica has an omission failure (failure to send a diagnosis message) with respect to some correct replicas but not with respect to other correct replicas. An equivalent situation occurs if there is a DM commission failure with respect to some correct replicas but these corrupted diagnosis messages are delayed by the network beyond the timeout period. Under these conditions, the diagnosis algorithm determines that there are some faulty replicas but is not able to determine exactly which replicas are faulty. Specifically, the diagnosis algorithm is able to determine a set of replicas that includes the faulty replicas but may also include some fault-free replicas. Under these conditions, as discussed in detail later on in this section, the algorithm may have to select a subset of the suspect replicas and declare only that subset as faulty. In particular, this must be done if the total number of replicas in the suspect set is greater than the assumed maximum possible

number of faulty replicas. This selection is done in such a way that the end result cannot be worse than the oblivious round-robin scheme discussed earlier. Specifically, with n nodes in the system, after at most n invocations of the algorithm, the faulty replica is identified as faulty.

Under rare circumstances, our algorithm does violate DDP1, i.e., it does not provide the same completeness guarantee as the oblivious round-robin scheme. Specifically, if during diagnosis there are some faulty replicas but none of the replicas exhibit faulty behavior, the result of the algorithm is that none of the replicas are identified as faulty. This can occur only if for every faulty replica the internal state is correct, the DM sends the internal state correctly to all other replicas, the DM sends all diagnosis messages on time to all other replicas, and the network does not delay messages beyond the timeout. If the SM is faulty, by our fault model assumptions the internal state will eventually be incorrect so that, eventually, the faulty replica will be identified by the diagnosis algorithm. If the SM is fault-free while the DM is faulty but does not exhibit faulty behavior during diagnosis, that replica may never be diagnosed as faulty. However, as long as the SM is not faulty, this faulty node continues to operate correctly during normal operation.

The self-diagnosis procedure is invoked whenever some behavior that may be wrong is detected with the server replicas. For example, in order to detect crash failures of replicas, all replicas may exchange heartbeat messages periodically. If the heartbeat messages from a replica are missing, the other replicas will invoke the

diagnosis procedure. The diagnosis is also invoked to deal with the situation that the protocol for totally ordering messages among replicas is blocked from making progress due to faulty replicas. Timeouts will expire on correct replicas then (as described in Chapter 4) and they will invoke the self-diagnosis. The diagnosis procedure may also be invoked when a client reports that it has received inconsistent replies from the replicas, or the responses from some replicas are too late or missing. In addition, the replicas can proactively invoke the diagnosis procedure to scrub latent errors so that faults are diagnosed before they accumulate.

In the rest of this section, Subsection 5.2.1 is a high-level description of the diagnosis algorithm for the general case, i.e., for any number of replicas. Subsection 5.2.2 is a detailed presentation of the self-diagnosis algorithm for three replicas ($f = 1$), as it is actually implemented in the Ghidrah CMM system.

5.2.1. The General Algorithm

The self-diagnosis algorithm is executed by the DM in every replica. The outcome of the diagnosis are two sets: **correct-set** $\mathbf{C}$ and **faulty-set** $\mathbf{F}$. Set $\mathbf{F}$ consists of all the replicas that are diagnosed faulty, and set $\mathbf{C}$ consists of all the replicas that are considered fault-free. Any replica r belongs to one of the sets, but can not be in both sets. The size of set $\mathbf{F}$ is less than or equal to f. Each replica r also maintains a suspect list S_r for the diagnosis; the list is set to empty at the beginning of the diagnosis.

The diagnosis protocol consists of four steps. For each step, a timer is set with

a timeout period, to avoid the situation that the diagnosis procedure is blocked by a faulty replica, or by messages that are delayed by the asynchronous communication system.

A replica sends diagnosis messages in each step. In some steps, when a replica receives a diagnosis message from another replica, it forwards the message to other replicas. This is needed to deal with the situation that the direct message a replica sends to another replica is delayed or lost. With the forwarding, the message will reach the destination through redundant paths. However, for the *RSN* messages in STEP 1 below, a replica still expects to receive the message directly from the sender. If this direct message is not received before the timer expires, the receiver suspects the sender. This is needed so that when a replica exhibits an omission failure to a correct replica, it can be detected by the correct replica.

The first step in the algorithm is to have all replicas agree on a *RSN* that is used in the self-diagnosis procedure as the point for the replicas to compare states. Henceforth, this *RSN* is denoted as RSN_{sd}. Each replica i also keeps a variable denoted RSN_{sd}^{i}, as the local copy of RSN_{sd} that i will use for comparison.

STEP 1. Each replica broadcasts to all the other replicas the *RSN* of the last client message it has processed. When receiving a *RSN* from another replica, it records the *RSN* and forwards the message to the other replicas. When a replica has received *RSN*s from all replicas, it computes RSN_{sd}^{i} as the maximum of all the *RSN*s reported by all replicas (include itself) and moves to the next step.

If the timer expires before *RSN*s are received from all replicas *directly*, this replica adds the replicas from which the *direct RSN* message is missing into its suspect list S_r, and computes RSN_{sd}^i as the maximum of all the *RSN*s it has received. It then moves to the next step.

The purpose of the second step is ensure that all replicas reach the point for comparison, i.e., they all process up to RSN_{sd}.

STEP 2. If a replica's *RSN* is equal to RSN_{sd}^i, then for any other replica j, if $RSN_j < RSN_{sd}^i$, it sends all the messages in its log (see Section 4.2.2) with sequence numbers from $RSN_j + 1$ to RSN_{sd}^i to replica j as ''catchup'' messages.

If a replica's *RSN* is lower than RSN_{sd}^i, it waits for the catchup messages sent by the other replicas. When it receives a catchup message, it forwards the message to the other replicas. If it has not processed the catchup message yet, it processes the message in the same way it processes normal messages, as described in Section 4.2.2.

If a replica reaches the point where it has processed all messages up to RSN_{sd}^i (possibly utilizing to the catchup messages received), it moves to the next step.

If the timer expires and a replica has not yet received all the messages up to RSN_{sd}^i, then the replica reduces its RSN_{sd}^i to the value that corresponds to the latest message that it has received. Furthermore, every other replica that in STEP 1 reported a higher *RSN* than the

current updated RSN_{sd}^{i} is added by the replica to is suspect list S_r.

The ‘‘catchup’’ messages are needed for dealing with the situation that some correct replicas missed these messages. If catchup messages are not sent to these replicas, they may not be able to reach the comparison point and generate a correct state for comparison. The catchup messages are also used to verify the validity of the highest *RSN*: if a faulty replica reports a *RSN* that is higher than the *RSN* it actually has, it will not be able to send all the catchup messages to others and correct replicas will suspect it as faulty.

The diagnosis procedure does not disrupt the availability of the service. New incoming messages are processed continuously during the diagnosis. As a replica continues processing new messages, its state may keep changing after it reports its *RSN*. It is possible that a replica reaches a state that is beyond the state corresponding to RSN_{sd} used in the ongoing self-diagnosis session. Here the critical period is from the time that a replica broadcasts its *RSN* to others (at the beginning of Step 1) to the time that RSN_{sd} is decided. During this period, the state after processing any new client message can potentially be the state needed for comparison. Hence, after processing each of the new message, the replica has to save a copy (or checksum) of its internal state. When RSN_{sd} is finally decided, the appropriate state (or checksum of the state) can be retrieved for comparison in Step 3.

STEP 3. Each replica broadcasts to all the other replicas its state (or state checksum) at RSN_{sd}^{i} for comparison. When receiving a state message

from another replica, a replica forwards the message to the other replicas and compares its own state with the received state. If the state from replica j is different than this replica's own state, it adds j into its suspect list S_r. When the comparison has been done with respect to all other replicas, the replica moves to the next step.

In the last step, replicas exchange their suspect lists and each replica makes the diagnosis decision independently by examining the collection of the suspect lists.

STEP 4. Each replica collects suspect lists by broadcasting its own suspect list and receiving the other replicas' suspect lists. It then applies the following reduction rules repeatedly until there is no rule that can be applied anymore, or the number of replicas in set $\mathbf{F}$ is equal to f. $\mathbf{C}$ and $\mathbf{F}$ are empty before the reduction.

R1. For a replica j that is neither in $\mathbf{C}$ nor in $\mathbf{F}$, if j does not belong to the suspect list of any replica that is not in $\mathbf{F}$, and j's suspect list does not consist of any replica that is not in $\mathbf{F}$, then j is put in set $\mathbf{C}$.

R2. For a replica j that is neither in $\mathbf{C}$ nor in $\mathbf{F}$, let n_j be the number of replicas that are not in $\mathbf{F}$ and whose suspect list includes j, if n_j is larger than f minus the size of $\mathbf{F}$, then j is put in set $\mathbf{F}$.

After the reduction, if all replicas have been put in either $\mathbf{F}$ or $\mathbf{C}$, the diagnosis completes with $\mathbf{F}$ and $\mathbf{C}$ as results.

We now consider the case that there are some replicas that are not in set $\mathbf{F}$ or $\mathbf{C}$. If the size of set $\mathbf{F}$ is equal to f, the diagnosis protocol restarts. This situation can occur if the comparison step has generated too many suspicions. For example, this might be caused by a sudden increase in communication errors that leads to delays of diagnosis messages. If the cardinality of $\mathbf{F}$ is less than f, we apply some selection algorithm to pick $f - |\mathbf{F}|$ replicas and put them in $\mathbf{F}$; the remaining replicas are put in $\mathbf{C}$. The diagnosis then completes with $\mathbf{F}$ and $\mathbf{C}$ as results.

In the case described above where the cardinality of $\mathbf{F}$ is less than f but some replicas cannot be classified into to either $\mathbf{F}$ or $\mathbf{C}$, we are facing the situation that the algorithm cannot identify the faulty replicas with accuracy, i.e., the conditions we listed earlier for complete and accurate diagnosis do not hold. As mentioned earlier, the algorithm has to select a subset of the unclassified replicas and declare this subset as faulty. This selection is conducted based on the seniority of the replicas. All the replicas are ordered based on the time they joined the active replica group. When selecting from the unclassified set, the $f - |\mathbf{F}|$ replicas that are ‘‘older’’ than the others are picked. Using this age-based selection, although fault-free replicas may be selected and put into the faulty-set $\mathbf{F}$, a faulty replica will be identified as faulty, after at most n invocations of the algorithm (n is the total number of active replicas).

Once the diagnosis algorithm produces the faulty-set $\mathbf{F}$, the reconfiguration procedure is invoked to replace the replicas in $\mathbf{F}$ with new replicas.

5.2.2. The Three-Replica Diagnosis Protocol

In this section we present the detailed self-diagnosis protocol we implemented in the Ghidrah CMM system. The protocol is based on a configuration of three active replicas. The pseudo-code of the self-diagnosis protocol is shown in Figures 5.1, 5.2 and 5.3.

```
Variables:
 1   X                  /* ID of this replica */
 2   Y, Z               /* IDs of the other replicas */
 3   S                  /* the management state */
 4   CHK(S)             /* checksum of the state S */
 5   RSN_x              /* RSN of the last group message processed */
 6   RSN_y, RSN_z       /* the last RSNs reported by other replica */
 7   SDRSN              /* this replica's RSN for diagnosis */
 8   SDRSN_F            /* final SDRSN prediction*/
 9   SDRSN_P            /* predicate for whether SDRSN_F has been determined */
10   CHK_F              /* state checksum taken at SDRSN_F for comparison */
11   CHK_LIST           /* list of temporary state checksums */
12   F                  /* the replica that X suspects to be faulty */

Invocation:
13   SDRSN_F := nil;
14   SDRSN_P := False;
15   CHK_F := nil
16   CHK_LIST := ∅;
17   F := nil;
18 Start SELF-DIAGNOSIS do
19   start DiagTimer with timeout value on the self-diagnosis procedure;
20   add ⟨CHK(S), RSN_x⟩ into CHK_LIST;
21   SDRSN := RSN_x
22   broadcast ⟨CHECK-RSN, SDRSN, X⟩_x to Y and Z;
23   start RSNTimer with timeout value on receiving RSN from all replicas;
```

Figure 5.1: Pseudo-code of the self-diagnosis protocol (part A): Variables and invocation.

This protocol follows the general algorithm described earlier, but with more

```
24 □ Upon RECEIVE msg:=⟨CHECK-RSN, rsn_i, i⟩_i do
25     forward msg to the other replica;
26     if [ received CHECK-RSN messages from both Y and Z ] then
27         SDRSN_P := True;   SDRSN_F := max(SDRSN, RSN_y, RSN_z);
28         stop RSNTimer;
29         if [ SDRSN > (RSN_y /RSN_z) ] then
30             forward group messages between RSN_y /RSN_z and SDRSN to Y/Z;
31         if [ RSN_x ≥ SDRSN_F ] then
32             CHK_F := state checksum in CHK_LIST taken at SDRSN_F;
33             broadcast ⟨CHECKSUM, CHK_F, RSN_x, nil⟩_x to Y and Z;
34     if [ msg is forwarded from another backup ] then
35         if [ received ⟨CHECK-RSN, RSN_p, p⟩_p directly from primary ] then
36             SDRSN_P := True;   SDRSN_F := max(rsn_i, RSN_p);
37             if [ SDRSN_F ≤ RSN_x ] then
38                 CHK_F := state checksum in CHK_LIST taken at SDRSN_F;
39                 broadcase ⟨CHECKSUM, CHK_F, SDRSN_F, nil⟩_x to Y and Z;
40         else buffer msg;
41 □ Upon RECEIVE msg:=⟨CHECKSUM, chk_i, rsn_i, f_i⟩_i do
42     let j be the replica other than X and i:
43     if [ SDRSN_F ≠ nil ] and [ CHK_F ≠ nil ] then
44         if [ rsn_i = SDRSN_F ] then
45             if [ rsn_i > RSN_x ] and
46               [ not received group messages between RSN_x and rsn_i ] then
47                 F := i;   send ⟨SUSPECT, i⟩_x to j;
48             else if [ CHK_F ≠ chk_i ] then
49                 F := i;   send ⟨SUSPECT, i⟩_x to j;
50             else if [ f_i ≠ nil ] then
51                 if [ F = f_i ] then
52                     send ⟨SUSPECT_ACK, F⟩_x to i;   stop DiagTimer;  Done.
53                 else if [ X = f_i ] then
54                     F := i;
55                 else
56                     if [ j is older than i ] then
57                         F := j;   send ⟨SUSPECT_ACK, F⟩_x to i;
58                         stop DiagTimer;  Done.
59                     else  /* i is older than j */
60                         forward msg to j;
61                         F := i;  send ⟨SUSPECT, i⟩_x to j;
62         if [ received CHECKSUM from both Y and Z ] and [ F = nil ] then
63             stop DiagTimer;  Done.
64     else buffer msg;
```

Figure 5.2: Pseudo-code of the self-diagnosis protocol (part B): Handling diagnosis messages.

implementation details covered, and a simplified ‘‘Step 4’’, given the fact that there are only three replicas and the assumption that only one of them could be faulty. The pseudo-code describes the behavior of one replica (replica *X*) in an event-driven style.

In Figure 5.1, lines 1-12 list all the variables that are used during the diagnosis procedure. When a replica enters the self-diagnosis procedure, it first resets some of the variables to their default values (lines 13-17). It then starts two timers with proper timeout periods. Instead of using a timer for each diagnosis step as described earlier in the general algorithm, we use one global timer (*DiagTimer*) for the entire diagnosis procedure, and one timer (*RSNTimer*) for the first step — agreeing on the RSN for state comparison. The timeout period of *RSNTimer* is shorter than the timeout period of *DiagTimer*.

If the entire diagnosis procedure does not complete before *DiagTimer* goes off, the replica restarts the diagnosis protocol. It keeps a counter that is used to distinguish different incarnations of the diagnosis procedure. Every time the diagnosis is invoked or restarted, the counter is incremented. Not shown in Figures 5.1, 5.2 and 5.3, every diagnosis-related message described in the pseudo-code includes the counter value of the current diagnosis session, so that a replica can tell whether a message it receives from another replica is for the diagnosis session it currently runs. If an incoming diagnosis message has a lower diagnosis counter number than the node’s counter number, the message is discarded. If an incoming diagnosis message has a higher diagnosis counter number than the node’s counter

```
65 □ Upon RSNTimer expires do
66     if [ not received any CHECK-RSN message ] then
67         restart SELF-DIAGNOSIS;
68     else if [ received CHECK-RSN message from i ] then
69         F := j;
70         SDRSN_P := True;
71         SDRSN_F := max(SDRSN, RSN_i);
72         if [ SDRSN > RSN_i ] then
73             forward group messages between RSN_i and SDRSN to i;
74         if [ RSN_x ≥ SDRSN_F ] then
75             CHK_F := state checksum in CHK_LIST taken at SDRSN_F;
76             let j be the replica other than X and i:
77             send ⟨CHECKSUM, CHK_F, SDRSN_F, j⟩_x to i;

78 □ Upon DiagTimer expires do
79     if [ SDRSN_P = True ] and [ SDRSN_F > RSN_x ] then
80         let i be the replica that RSN_i = SDRSN_F ]:
81         send ⟨SUSPECT, i⟩_x to j;
82     else
83         restart SELF-DIAGNOSIS;

84 □ Upon RECEIVE regular group message msg: = ⟨m, rsn⟩_i from i do
85     let j be the replica other than X and i:
86     if [ i is the primary ] then
87         forward msg to j;
88     if [ rsn = RSN_x + 1 ] and [ received msg from other two replicas ] then
89         process m;  RSN_x := rsn;
90     if [ SDRSN_P = True ] then
91         if [ RSN = SDRSN_F ] then
92             broadcase ⟨CHECKSUM, CHK_F, SDRSN_F, F⟩_x to Y and Z;
93             process CHECKSUM messages buffered during this diagnosis;
94     else
95         add ⟨CHK(S), RSN_x⟩ into CHK_LIST;

96 □ Upon RECEIVE msg: = ⟨SUSPECT, f_i⟩_i do
97     if [ F = f_i ] then
98         send ⟨SUSPECT_ACK, F⟩_x to i;

99 □ Upon RECEIVE msg: = ⟨SUSPECT_ACK, f_i⟩_i
100    if [ F = f_i ] then
101        stop DiagTimer; Done.
```

Figure 5.3: Pseudo-code of the self-diagnosis protocol (part C): Handling timeouts and other messages.

number, the message is buffered. When the replica has diagnostic messages with counter numbers higher than its current number from at least $f+1$ replicas, it aborts its current diagnonsis run and begins processing the higher numbered diagnosis messages.

At the beginning of a diagnosis session, a replica starts *DiagTimer*, and takes the checksum of its internal state S and save it. It then sends to the other replicas a CHECK-RSN message to report the RSN of the last client message it has processed, and starts the *RSNTimer* (lines 19-23).

A replicas continues processing new regular client messages while the diagnosis is in progress (line 83-88). However, until the RSN that will be used for state comparison has been determined (*SDRSN_P* becomes true), the replica takes the checksum of its state after processing every client messages and saves the checksum with corresponding *RSN* (line 94), as this state could potentially be the state for comparison.

When a replica receives the CHECK-RSN message (line 24), it relays the message to the other replica that is not the sender of this message (line 25). When it has received CHECK-RSN messages from all the other replicas, it determines the RSN for state comparison (*SDRSD_F*) by picking the highest RSN reported by all replicas, including its own (line 27). If this replica is ahead of other replicas on processing client messages, it sends the ‘‘catchup’’messages to the other replicas so they can reach the comparison point (line 29-30). The replica then retrieves the state checksum that was saved on the now-known comparison point and sends a

CHECKSUM message to the others for comparison (line 31-33).

A replica expects to receive CHECK-RSN messages from all the other replicas and proceed before *RSNTimer* expires. If by time that *RSNTimer* expires, it has not received a CHECK-RSN message, it then restarts the diagnosis procedure (line 66-67). If it has received the CHECK-RSN message only form one other replica it then suspects the replica whose CHECK-RSN message is missing to be the faulty one. It determines the comparison points according to its own SDRSN and the one it has received, and sends its state checksum to the replica that has sent it the CHECK-RSN message, along with the suspicion that the other replica may be faulty (line 68-76).

To deal with a faulty replica that reports a RSN that is too high that none of the correct replicas can reach it during the diagnosis, each replica needs to justify the RSNs it has received. As the pseudo-code shows, a replica that has reported the highest RSN must send proper catchup messages once the comparison point has be determined, so all the other replicas are able to catch up with it. If a faulty replica *i* sends out an invalid RSN without the catchup messages following, other replicas will not reach the comparison point and will wait until *DiagTimer* expires. They will then suspect that *i* is faulty (line 78-80). The replica is also supposed to send to others the catchup messages before it sends out its own state checksum for comparison. Therefore, if a replicas doe not receive the catchup messages before its receives the CHECKSUM message from this replica, it suspects this replica as faulty as well (line 45-47).

When a replica receives a CHECKSUM message from another replica, and it has known the comparison point, it compares the state checksum reported in the message with its own state checksum. If the checksums mismatch, it suspects the replica to be faulty and reports the suspicion to the third replica (line 48-49).

When a replica receives from another replica a message suspecting the third one is faulty, if it also suspects that replica is faulty, it replies positively (line 51-52, 96-97). If it does not suspect that replica, then it faces the situation that it can not make accurate diagnosis. As we discussed earlier, in this case the replica chooses between the other two replicas the ‘‘older’’ one to be suspected as faulty (line 56-61).

The diagnosis procedure completes and *DiagTimer* is stopped when a replica is declared as faulty after the replicas exchanged their suspicions and acknowledgments, or when the replicas have compared their states and did not get any mismatch. In the first case, the replicas will invoke the reconfiguration protocol, which is described in the next section, to replace the faulty replica with a new replica. In the second case, the replicas conclude that all the replicas are fault-free and continue normal operation.

5.3. Reconfiguration

Once the self-diagnosis procedure has identified the faulty replicas, the fault-free replicas invoke the reconfiguration procedure to replace the faulty replicas with new replicas. A new replica joins the manager group only when it has received

reconfiguration requests from at least $f + 1$ active replicas, so reconfiguration cannot be wrongfully initiated by f faulty replicas. In this section, we describe the reconfiguration protocol implemented in Ghidrah.

The reconfiguration procedure is illustrated in Figure 5.4 as a flow chart. The procedure starts when the self-diagnosis procedure has produced the diagnosis result that one of the replica is faulty. The reconfiguration will remove this faulty replica from the group and start a new replica with correct state transferred.

At the beginning of the reconfiguration procedure, a replica first checks whether the faulty one is the primary replica (block 1 in Figure 5.4). If the faulty replica is the primary, then a new primary must be elected first. We choose the replica that is ahead of the other one on processing client messages as the new primary. The two replicas can find this out by exchanging their current *RSN*s. The replica that reports a higher *RSN* becomes the new primary (block 2-6). If they report the same *RSN*, then the replica that has a lower node ID becomes the new primary. Each replica notifies all clients about the new primary so that the clients can send their messages properly to the group (block 7). The steps described above are skipped if the faulty replica is a backup.

Next, the replica needs to decide on which node to start the new replica (block 8). The node is selected from the current set of candidates that includes all the nodes that can ran a replica and do not have a replica running already. The selection policy is deterministic and based on the node IDs, so the two replicas will independently make the same selection.

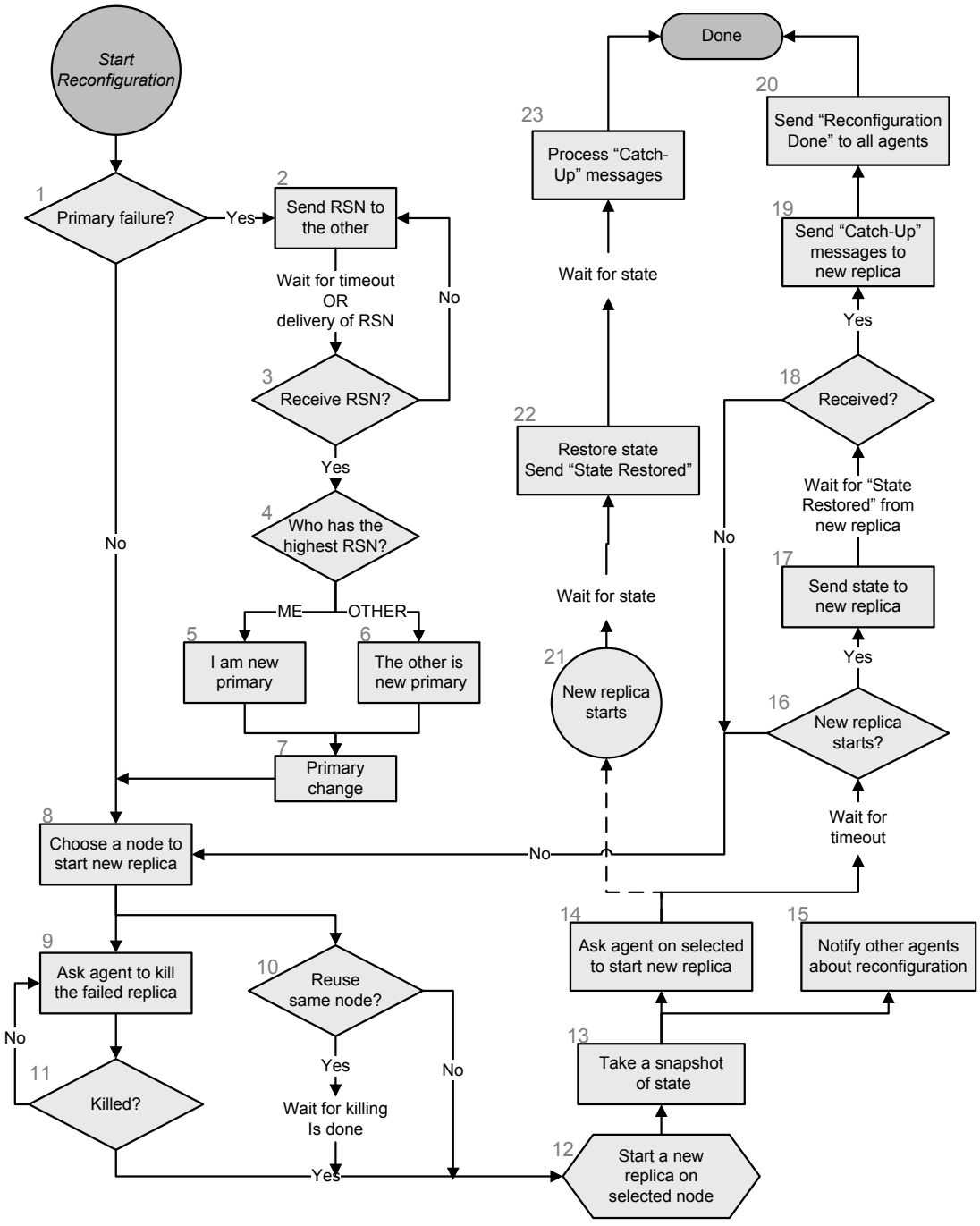


Figure 5.4: The reconfiguration procedure

In order to start a new replica on a node, there needs to be some process running on the node that can start a replica process. In the Ghidrah CMM system, the agent running on the same node is responsible for this. Each replica sends a request to the agent on the node that the faulty replica runs on, asking it to terminate the faulty replica. It also sends a request to the agent on the selected node to start a new replica. It is possible that the node selected for the new replica is actually the same node on which the old replica runs, under the rare condition that there is no other candidate available. If this is the case, the request of starting new replica must wait until the agent reports that the old replica has been terminated successfully (block 9-11). If the selected node is a different one, then starting new replica and terminating the faulty replica can be conducted concurrently.

To start a new replica, a fault-free replica first takes a snapshot of its state and calculate the checksum of the state (block 13). It then sends a ‘‘start-new-replica’’ request with the checksum to the agent on the selected node (block 14), and sends notifications to all other agent so they will know about the reconfiguration and the new replica (block 15). The agents then only expect messages from these two replica, but not from the new replica.

Upon receiving both ‘‘start-new-replica’’ requests, the agent of the selected node compares the checksums in the requests. If they match, the agent starts a new replica process and passes the checksum to it. When the new replica starts, it sends a message to the two existing replicas to tell them that it has started and is ready for

state transfer.

The new primary replica transfers the state it saved earlier to the new replica when it receives the message from the new replica (block 16-17). The new replica then verifies the state with the checksum it received from the agent, restores the state, and sends a ‘‘state-restored’’ message to the two existing replicas (block 22).

When a replica receives this ‘‘state-restored’’ message, it sends to the new replica all new group messages it has received since the reconfiguration procedure started as ‘‘catchup’’ messages (block 18-19). The new replica processes all these messages without sending actual responses to clients (block 23). After sending all the ‘‘catchup’’ messages to the new replica, a replicas broadcasts a message to all clients to notify them that the reconfiguration has completed (block 20). The system then switches back to normal and the clients now expect receiving messages from all three replicas.

This reconfiguration protocol ensure that the new replica gets the correct state so the replica group recovers to its full functionalities and fault resilience. During the reconfiguration, if the new replica fails to start or fails to restore its state, it will be considered as faulty too and the reconfiguration procedure restarts.

Chapter Six

Agents

As described in Chapter 3, in our CMM system, an agent daemon runs on every node in the cluster. The agent is a proxy that allows the manager group to control the node and application processes running on the node, and allows the application processes to interact with the management middleware. As mentioned in the previous chapter, an agent is also responsible for terminating a faulty manager replica and starting a new manager replica on the node when the manager replicas need to be reconfigured.

An agent sends its messages to the manager group using the total order multicast protocol described in Section 4.2. An agent acts as a client in the protocol. It receives individual messages from each of the manager replicas, compares and votes on these messages. It accepts and executes a manager command only if it has receive identical copies of the command from at least two manager replicas.

The survival of a particular agent or of a particular cluster node running an agent is not critical to the continued correct operation of the entire system. Hence, it is not worthwhile to use with the agent aggressive fault tolerance techniques, such as those used with the manager group. On the other hand, requiring a full node reset every time an agent process crashes, may, in hostile environments significantly decrease the availability of cluster resources. Hence, taking these

practical engineering tradeoffs into account, the Ghidrah system implements a low-cost mechanism for recovery from agent crash failures. This mechanism is described in Section 6.1.

An agent process automatically starts on each cluster node when the node is powered up. If the management system is already up, i.e., the manager group is operational, the new node must be incorporated in the running cluster so that it can be allocated by the manager group, as needed. Hence, the agent on the new node must find out what is the current manager group and contact it for further instructions. If there is no operational manager group, as is the case when the entire cluster is powered up, the agents are responsible for configuring the CMM system by executing a bootstrapping protocol. This bootstrapping procedure allows the CMM system to configure itself with a working manager group. Section 6.2 presents this bootstrapping protocol.

6.1. Recovery from Agent Crashes

An agent runs on a simplex COTS cluster node that does not have redundancy at the hardware level or at the operating system kernel level (COTS hardware and a COTS operating system are used). The hardware and operating system on a COTS node are unreliable and they may fail. If the hardware (processor, memory, I/O devices, etc) or the operating system fail, node behavior may be arbitrary. In particular, even if the execution of processes (user-level code) on the node is guaranteed to be fault-free, once the user code requests operating services, the

ultimate result may be arbitrarily incorrect. Hence, it is not possible to fundamentally increase the robustness of the node through changes to the agent's user-level code —a single fault can corrupt the node. In addition, the survival of any one particular node of the cluster is not critical to the continued correct operation of the cluster. Based on these considerations, there is no justification for using with the agent aggressive fault tolerance techniques, such as those presented in the previous two chapters.

A node in the cluster may become unusable due to many reasons, such as a kernel crash or the inability of the cluster manager to control the the node since the agent process crashes. If the cause of such failures is a transient hardware or software fault, the node may be restored through a power reset of the hardware that is followed by a reboot of the OS kernel and a restart of the agent process. However, power-resetting a node is expensive in terms of lost computation time since all running processes on the node, including all the application processes, are killed by the power reset. Hence, it is worthwhile to reduce the probability of requiring a node reset if this can be achieved using low overhead mechanisms that do not significantly increase the complexity of the management software.

Fault injection experiments indicate that, in practice, most faults that cause errors result in a process crash [Made02, Whis02]. Hence, as an engineering tradeoff, it is worthwhile to add a simple mechanism that allows recovery from agent crashes without a node reset. In this crash failure scenario, the agent process simply crashes, but the hardware and the operating system still operate correctly,

and application processes are intact. In this case, we should provide a low-cost recovery mechanism so that the management middleware can recover the crashed agent and maintain the control over the node and the application processes running on the node.

Recovery from agent crash failures requires a mechanism to detect that the agent has crashed and start a new agent process. It also needs a method that allows the new agent process to restore information about all the application processes that have been running on the node, and re-gain control over these processes, so that the applications can continue without interruptions and remain under the management of the CMM system.

One way to support recovery from agent crashes would be to maintain two agent replicas running on each node. The recovery scheme used in Ghidrah achieves the benefits of maintaining such a spare with less overhead and higher reliability. The scheme is based on a simple *agent keeper* process running on each node whose sole function is to monitor the agent process using heartbeat mechanism and initiate a new agent process when a crash failure is detected. Table 6.1 compares the complexity of the agent process and the agent keeper process. It shows that the agent keeper process is simpler and smaller than the agent process. Hence, the agent keeper is less likely to crash than a running agent and less likely to be corrupted than a spare agent replica.

In order to minimize the complexity of the agent keeper, the agent and agent keeper communicate using a shared memory segment. During normal operation the

	Agent	Agent Keeper
Code Size	25,000 lines	529 lines
Binary Size (dynamic linking)	359KB	29KB
Binary Size (static linking)	1,166KB	422KB
Runtime Size	2,740KB	950KB

Table 6.1: Comparing the complexity of agent and agent keeper.

agent increments a counter in this shared memory segment that the agent keeper reads. This counter serves as a heartbeat channel from the agent to the keeper. If for two consecutive reads of the counter the keeper determines that the agent failed to increment the counter, the keeper concludes that the agent has either crashed or hung. The keeper then kills the old agent process, if the process still exists, and starts a new agent process.

As described so far, the agent recovery mechanism fails to deal with a critical issue: who keeps the keeper alive? The solution is to have the agent monitor the keeper and restart the keeper when the keeper crashes. The keeper and the agent monitor each other and form a cycle of recovery. Hence, the agent and the keeper keep each other alive unless they both fail at the same time. If that happens, the node is considered ‘‘dead’’ and the manager group detects the failure with missing node heartbeats (see Section 7.1.5).

When a new agent process starts, it must obtain part of the state of the old agent, so that it can continue managing the local node and application processes that are running on the node. This state includes information about all processes

that have been running on the local node, such as their process IDs, and their task affiliation. The new agent can obtain this information from the manager group, since the task database maintained by the manager group includes information regarding all the application processes running on each node. Thus, when it starts, the new agent sends a request for the state to the manager group. The new agent uses the reply from the manager group to re-create the required state.

An agent has the ability to control an application process that is running on the node. It can terminate an application process, and stop or resume an application process based on scheduling commands from the managers. An agent also needs to communicate with an application process so that it can send management information to the application process. For example, it must send to an application process that belongs to a parallel task information about the node assignment of the task, i.e. on which node each process of this task is running. The application process needs this information so that it can communicate with other processes of the same task. The agent also receives from application processes management requests so that applications can request special operations of the CMM. For example, an application process may request the CMM to terminate or restart the whole task it belongs to.

As part of recovery from an agent crash failure, the new agent must re-establish communication with the application processes it controls. In Ghidrah, this communication is implemented using ‘‘named FIFOs’’. When the agent starts an application process, it creates a pair of FIFOs with names generated from the

process ID of the process. The agent and application process then communicate through these FIFOs. After recovery from an agent crash, the new agent gets the names of the FIFOs from the process IDs it obtained from the managers, and re-opens the FIFOs. In this way, the agent re-establishes the agent-application communication and obtains full control over the application processes.

6.2. A Fault-tolerant Bootstrapping Protocol

The operation of a cluster managed by Ghidrah is dependent on a functional manager group. A critical feature of the CMM system is the ability to operate continuously without human intervention. Hence, there must be an autonomous mechanism for setting up the manager group when the system is powered up. This is important since beyond the initial power up of the system, as discussed in Section 3.4, under certain conditions (e.g., too many simultaneous node failures) the entire system may be reset by the trusted hardcore.

When the cluster just starts, it could be quite unstable: some nodes might fail to start, or start but fail soon thereafter, or start with limited connectivity to other nodes. In such a scenario, a system that relies on a static configuration of the manager group (i.e. a pre-determined set of nodes to run the manager replicas) cannot guarantee reliable initiation of the manager group. The system needs to configure the manager group based upon the actual conditions when the system is powered up. In order to do this, we developed a system bootstrapping protocol for Ghidrah. This protocol adapts to the condition of the cluster and self-configures the

system to reliably start the manager group.

The system bootstrapping procedure is basically an election: instead electing one leader, it elects a set of leaders —a set of nodes to run the manager replicas. All the correct members in the cluster must agree on this set of leaders. Because such an election requires reaching consensus among the cluster nodes, our bootstrapping protocol is based on Lamport's Paxos algorithm [Lamp98, Pris00].

The classic Paxos algorithm presented in [Lamp98] considers only stopping failures. The bootstrapping protocol we developed can be considered as a variation of Byzantine Paxos algorithms [Lamp01b, Cast99a], with the assumption that no more than f agents/nodes are Byzantine faulty during the bootstrapping.

The goal of the bootstrapping procedure is for all the working nodes to agree in a *configuration*, i.e., which nodes will run manager replicas. Each node in the cluster is labeled by a unique positive integer. With the three-replica manager group in Ghidrah, a configuration is a 3-tuple (i, j, k) such that $i < j < k$ and i, j, k are the identifiers of the nodes that run the manager replicas.

The basic idea of the bootstrapping protocol is to have agents play the role of proposers (see Section 2.1.1) and ask other agents to accept and commit to their proposals. A proposer selects a manager configuration ($2f + 1$ nodes to run the manager replicas) based on its own view of the system and sends the configuration as a proposal to all other agents. If an agent (here it plays the role of an acceptor) receives from at least $f + 1$ proposers a proposal for a common configuration that is better than any configuration it has known about, it sends to others a message to

indicate that it supports this proposal and promises that it will not support any proposal that is not better than this one. A configuration C_1 is ‘‘better than’’ another configuration C_2 if C_1 is lexicographically less than C_2. For example, configuration (1,2,3) is better than (2,3,4), and (1,2,4) is better than (1,3,4). If an acceptor receives a proposal that is not better than the one it currently supports, it responds to the sender with the proposal that it supports.

When an agent receives supports for a configuration from a quorum of agents (‘‘support-quorum’’), it sends to others a proposal for the configuration, and this time it asks other agents to accept this proposal. If an agent receives such a proposal and it has not supported a better proposal, it sends out a message to announce its acceptance of this proposal. After accepting a proposal, an agent will not accept any other proposal.

If an agent receives acceptances for a configuration from a quorum of agents (‘‘accept-quorum’’), it then commits to this configuration. The protocol ensures that all correct agents will commit to the same configuration by requiring that the support-quorum of a proposal and the accept-quorum of a proposal must have at least one correct node in common. Because this node has already accepted a proposal, any proposal for a different configuration would not be able to get a support-quorum with sufficient number of nodes. Given the total number of nodes is N, and the maximum number of nodes that could be Byzantine faulty is f, the size of these quorums is set to $\lceil (N + f + 1)/2 \rceil$.

When a configuration has been committed, the agents on the nodes that are

included in the configuration start the manager replicas to form the manager group. Each committed agent then tries to communicate with the manager group and requests to join the cluster. If it does not receive group responses from the manager replicas, it then assumes that the configuration is bad and restarts the bootstrapping protocol; otherwise, it exits the bootstrapping and starts normal operation.

If an agent starts the bootstrapping protocol after a configuration has been committed, it will learn about the committed configuration from other agents. When it receives consistent messages from at least $f + 1$ other agents, it commits to this configuration as well.

Figures 6.1 and 6.2 show the pseudo-code of the bootstrapping protocol. The protocol consists of four phases. At the beginning of each phase, a timer is started with a timeout period. The agent waits in each phase until some condition becomes true; it then proceeds into the next phase. If the condition is not true when the timer expires, the agent restarts the bootstrapping procedure. This kind of behavior is described using the syntax {**do** A_1; **until** C : A_2; **timeout:** A_3}, which indicates that the agent keeps doing action A_1 until condition C becomes true; then it performs A_2. If the timeout period expires, it performs A_3.

An execution of this four-phase protocol is called a *round*. An agent keeps executing the protocol until a round has finished all four phases successfully. If a round cannot succeed, the agent restarts the protocol by initiating a new round. The agent keeps a sequence number for the current round it is executing. This sequence number is incremented every time a new round is initiated. It is attached to every

Variables:

CFG /* the current manager configuration this agent has */
SEQ /* the sequence number of current configuration */
LOG /* the log of bootstrapping messages*/

Bootstrapping Messages:

⟨ALIVE, *seq*⟩ : the sender is alive;
⟨SUPPORT, *cfg*, *seq*⟩ : the sender supports *cfg*;
⟨ACCEPT, *cfg*, *seq*⟩ : the sender has accepted *cfg*;
⟨COMMIT, *cfg*, *seq*⟩ : the sender has committed to *cfg*;
⟨JOIN, *cfg*, *seq*⟩ : an agent sends to manager group asking to join the cluster;
⟨ADMIT, *cfg*⟩ : manager group sends to an agent to admit the node;

□ **Upon** receiving any bootstrapping message from agent *i* **do**
  if already in normal operation **then**
    reply to *i* with ⟨COMMIT, *CFG*, *SEQ*⟩;
  else
    let *seq* be the sequence number of the received message:
    if *seq* ≥ the sequence number of the message from *i* saved in *LOG* **then**
      add the received message into *LOG* to replace the old message;

Figure 6.1: The Ghidrah bootstrapping protocol, part A.

bootstrapping message the agent sends out during the current round, so that messages sent in different rounds can be distinguished. The generation of this sequence number must take into account the situation that an agent is restarted or a node is rebooted. In our implementation of the bootstrapping protocol, the sequence number is actually composed of three fields: a boot ID, a restart sequence number, and a configuration sequence number. The configuration sequence number is incremented every time the agent changes its configuration without restarting the entire protocol. The restart sequence number is incremented every time the agent restarts the bootstrapping protocol. The boot ID is a unique number generated when the node is rebooted. It could be generated from the value of the hardware clock on the node.

□ **On Start**:
do *self-test*; check communication with SCC;
goto Phase 1;

□ **Phase 1**:
broadcast ⟨ALIVE, *SEQ*⟩ to all;
do
when receiving any message: reply with ⟨ALIVE, *SEQ*⟩
until messages from at least $\lceil (N + f + 1)/2 \rceil$ nodes in *LOG*
and at least $(2f + 1)$ of them can run manager replicas :
goto Phase 2;
timeout: restart;

□ **Phase 2**:
CFG := $(2f + 1)$ nodes with lowest ids so that messages from them are in *LOG*
and they can run manager replicas;
broadcast ⟨SUPPORT, *CFG*, *SEQ*⟩ to all;
do
when seeing a configuration better than *CFG*:
update *CFG* and increment *SEQ*;
goto Phase 2;
when receiving any message: reply with ⟨SUPPORT, *CFG*, *SEQ*⟩;
until SUPPPORT/ACCEPT/COMMIT messages for *CFG* from at least
$\lceil (N + f + 1)/2 \rceil$ in *LOG* :
goto Phase 3;
timeout: restart;

□ **Phase 3**:
broadcast ⟨ACCEPT, *CFG*, *SEQ*⟩ to all;
do
when receiving any message: reply with ⟨ACCEPT, *CFG*, *SEQ*⟩;
until ACCEPT/COMMIT messages for any *cfg* from at least $\lceil (N + f + 1)/2 \rceil$ in *LOG* :
change *CFG* to *cfg* and increment *SEQ*;
goto Phase 4;
timeout: restart;

□ **Phase 4**:
start manager replica if this node is in *CFG*;
send ⟨JOIN, *CFG*, *SEQ*⟩ to the manager group;
do
when receiving any message: reply with ⟨COMMIT, *CFG*, *SEQ*⟩;
until received ⟨ADMIT, *cfg*⟩ from the manager group :
terminate **bootstrapping** and enter normal operation;
timeout: restart;

□ **Whenever** at least $(f + 1)$ COMMIT messages with same *cfg* in *LOG* **do**
if not in Phase 4 **then**
change *CFG* to *cfg* and increment *SEQ*;
goto Phase 4;

Figure 6.2: The Ghidrah bootstrapping protocol, part B.

An agent saves bootstrapping messages (or the information carried by these messages) it has received from other agents in a log. It also saves copies of messages it has broadcasted in the log. It only accepts messages that are newer than the messages it has already received from the same sender. The agent decides whether a received message is newer by comparing its sequence number with the logged sequence number of the last message received from the same sender. When it accepts a new message from another agent, the receiving agent updates its log, replacing the previous most recent message received from that sender. In each phase, the agent checks its log to see if some condition has become true so that it can proceed.

As shown in Figures 6.1 and 6.2, the agent starts the bootstrapping protocol by performing a self-testing first in an attempt to determine whether the node is healthy and has the full functionality. It also tries to communicate with the trusted hardcore (sec Section 3.4) since only nodes that can communicate with the trusted hardcore can run manager replicas. The agent then enters the first phase. In this phase, it broadcasts a message to announce that it is alive and let others know about its status (e.g, whether it can run a manager replica). It then gathers information about other nodes. When it determines that there may be a valid manager configuration, it enters Phase 2 and proposes the configuration. In Phase 2, it waits until a configuration has gotten its ‘‘support-quorum’’. In Phase 3, it announces its acceptance of the configuration and waits until a configuration has gotten the ‘‘accept-quorum’’. The agent commits to the configuration in Phase 4, starts the manager replica if necessary, and attempts to communicate with the manager

group. The bootstrapping procedure completes when it receives responses from the manager group.

Once the bootstrapping procedure completes, the agent enters normal operation. If it receives a bootstrapping message from another agent afterwards, it responds with a message indicating that there is already a committed configuration and specifying what it is. If an agent that is still in the execution of the bootstrapping protocol receives at least $f + 1$ *consistent* messages of this type, it commits to the configuration specified in those messages. This allows an agent that starts late to learn about the committed configuration and complete the bootstrapping quickly.

During the bootstrapping, it is possible that a faulty agent proceeds through the bootstrapping procedure and is elected to run a manager replica on the node. This faulty agent may then fail to start the manager replica or start a manager replica that is faulty as well. The bootstrapping protocol does not need any special mechanisms to handle this situation. The reason for this is that the manager replication mechanism already has the ability to deal with a crashed or otherwise faulty replica. Hence, for example, with three replicas, if one of the replicas fails to start, the other two replicas will detect missing heartbeats from that replica, initiate self diagnosis (Chapter 5), declare the replica as faulty, and cause a new replica to be started on a different node.

Another possible situation is that an agent lies about its connection to the trusted hardcore to others so that it is elected to run a manager replica even though

it cannot communicate with the hardcore. It is also possible that the agent does not lie, but by the time it starts the manager replica, the connection to the hardcore is lost. In both cases, when the manager replica starts, it tries to communicate with the hardcore and determines that it does not have the connection. The manager replica then terminates itself so that the fault exhibits as a crash failure of the manager replica. The system will recover from the failure with the self-diagnosis and reconfiguration procedure as well.

Chapter Seven

Implementation and Experimental Evaluation

In previous chapters, we have presented several techniques for fault-tolerant cluster management. These techniques allow a highly reliable CMM system to be built using mostly standard COTS hardware and software components. To demonstrate this, we have implemented Ghidrah as an operational CMM system with these techniques integrated into the implementation.

Section 7.1 describes key aspect of the implementation of the Ghidrah CMM. In the implementation of the manager group, Ghidrah uses a simplified version of the replication algorithm presented in Chapter 4. The details of these simplifications and their implications are presented in Subsection 7.1.1. Subsection 7.1.2 describes the internal structure of the manager and agent as implemented in Ghidrah. At the core of the implementation of the manager replicas and agents is the way message and timer events are handled. The implementation framework for event handling in these programs is presented in Subsection 7.1.3. Subsection 7.1.4 describes how the consistency among manager replicas are maintained in the presence of group level time-triggered events. In subsection 7.1.5, we describe the technique used to achieve efficient heartbeat for node failure detection.

With the possible use of Ghidrah for on-board high-performance computing in space, it is expected that the trusted core will be the radiation-hard spacecraft control computer (SCC). In order to validate the operation of the entire system, we

have implemented an emulation of the SCC on a COTS computer. The functions performed by this emulated SCC are discussed in Subsection 7.1.6. The communication infrastructure (CI) of Ghidrah provides reliable authenticated communication and thus plays a critical role in the operation of the system. Subsection 7.1.7 described the implementation of the communication infrastructure.

In order to evaluate the performance of the Ghidrah CMM system and validate its fault tolerance mechanisms, we have conducted a series of experiments to measure the performance of the Ghidrah system and to test the system under faults in fault injection campaigns. The preliminary results from this experimental evaluation are presented in Section 7.2

The results of performance measurements presented in Subsection 7.2.2 show that the overhead caused by the Ghidrah CMM is quite low. For example, with a cluster of 20 nodes where each agent generate a heartbeat message every 100ms, during normal operation, the overhead of running a manager replica on a node is less than 2% of the node's CPU processing power.

The results of fault injection experiments presented in Subsection 7.2.3 indicate that although most errors caused by faults are process crash errors, non-crash errors are also occur with sufficient frequency so that they cannot be ignored. These fault injection campaigns show that, with the fault tolerance techniques presented in this thesis, the Ghidrah CMM tolerates and recovers from all failures caused by faults injected into a single manager replica.

7.1. Implementation of the Ghidrah CMM

The Ghidrah CMM system is currently operational on clusters built with COTS hardware and software. It is implemented in C/C++ on the Linux operating system with the communication infrastructure (CI) implemented using UDP socket calls. An early version has been tested on the Solaris operating system with communication implemented on a Myrinet network [Bode95] using the GM API [Li01].

Component	Lines of Code
Manager	12,900
Agent	10,600
Management Message Layer	6,200
Communication Layer (on UDP)	8,600
MPICH to CL	2,800
Emulated SCC	3,400

Table 7.1: Code size of the Ghidrah implementation. The communication layer(CL) and the management message layer(MML) are reported separately from the manager and agent programs. The actual manager program and agent program both consist of a copy of the CL and MML modules.

As a measure of the complexity of the Ghidrah implementation, Table 7.1 lists the code size of different Ghidrah components. As parts of the communication infrastructure of Ghidrah, the communication layer (CL) and the management message layer (MML) are common modules of the manager program and the agent program. Therefore, the code sizes of CL and MML are reported separately from those of the manager and agent programs. The numbers reported for the manager

and the agent do not include the code for CL and MML.

7.1.1. Practical Tradeoffs Towards a Simplified Replication Algorithm

In Chapter 4, we presented a replication algorithm that requires $2f + 1$ active replicas and a three-phase protocol during normal-case operation. This algorithm guarantees safety and liveness (see Section 4.1) as long as the total number of faulty replicas does no exceed f during the lifetime of the system.

We then presented in Chapter 5 a self-diagnosis algorithm that allows the system to identify the faulty replicas, and a reconfiguration algorithm that replaces the faulty replicas with new, faulty-free replicas. If the conditions under which the diagnosis algorithm is both complete and accurate hold, the system will keep replacing faulty replicas and thus may be able to tolerate an arbitrary number of faults during its lifetime.

The three-phase replication algorithm is expensive: during fault-free execution, to tolerate a single faulty replica, it requires 12 messages exchanged among the three active replicas for each client message. Furthermore, the three phase protocol will also increase the response time to client requests. In this section, we discuss the possibility of using replication algorithms that are based on the algorithm presented in Chapter 4 but require fewer phases. These algorithms sacrifice resiliency under extremely unlikely scenarios in order to improve efficiency. The current implementation of the Ghidrah CMM uses a one phase algorithm.

As discussed in Section 4.2.6, a simplified version of our replication algorithm, that requires only two phases during normal-case operation (Figure 4.3), can ensure that all fault-free *active* replicas process input messages in the same total order and their internal states are consistent. However, this two-phase algorithm may cause inconsistency between replicas and clients(agents), thus violating safety.

The inconsistency described in Section 4.2.6 is caused by the fact that a client accepts a reply from the server replicas after receiving consistent messages from only two replicas. As a result of a reconfiguration (view-change), the set of active replicas may include only *one* of the replicas that sent the reply accepted by the client. Since this particular replica may be faulty, it may lead the other fault-free replicas to process the client message in a different order. Therefore, the states of the fault-free replicas become inconsistent with the reply previously accepted by a client.

One way to overcome this problem is to have the client wait until it receives *three* replies from different replicas in the same view and vote on them. In this way, when the client accepts a reply, it is ensured that all active replicas have decided on the order of the client message so that there will not be inconsistency across reconfigurations. If the client does not receive replies from all three active replicas before the timeout period expires, it broadcasts the message to all active replicas, as described in Section 4.2.1, triggering a view change. The disadvantage of this solution is the requirement that the client must wait for replies from all

active replicas. Due to this requirement, if one replica is faulty and does not send the reply, the client may have to wait for a reply for a long time. Specifically, the client may have to wait until the view change completes and the new replica is initiated. This increases the interruption of normal operation due to a manger replica failure.

With the two phase algorithm, an alternative to requiring the client to receive replies from all the replicas is to rely on the diagnosis algorithm (Section 5.2). In particular, whenever the replication algorithm requires a view change, the diagnosis algorithm is invoked in order to pick which active replica to replace. If the diagnosis algorithm is alway accurate and complete, it will identify the faulty replica. If the replica that is replaced is alway the faulty replica, the inconsistency problem with the two phase protocol cannot occur.

The Ghidrah implementation takes one additional step towards simplifying the replication mechanism and uses a single phase replication protocol, as shown in Figure 4.1. It is possible to use this simple algorithm by relying on the existence of the trusted hardcore, which must be part of the system for other reasons anyway (Section 3.4). As discussed in Section 4.1, the one-phase algorithm faces a serious problem when the primary is malicious (Figure 4.2). The malicious primary may forward messages with different sequence numbers to the backup replicas, thus causing the states of the fault-free backups to diverge. In this way, the malicious primary can cause one of the backups to be identified as faulty by the self-diagnosis procedure and thus to be removed from the replica group during reconfiguration.

The faulty primary can keep doing this so that all the replicas spend all their time in self-diagnosis and reconfiguration, failing to perform their normal function. A faulty primary can also cause the state of the backups to diverge and then cause the self-diagnosis algorithm to conclude that the states of the active replicas are all different so that normal system operation cannot continue.

In all the scenarios described in the previous paragraph the replica group cannot recover by itself. However, every time a replica enters self diagnosis, it informs the trusted hardcore. The trusted hardcore assumes that self diagnosis has, in fact, been initiated if it receives these notifications from at least two replicas. If the frequency of entering self diagnosis within a time window exceeds a given threshold, the trusted hardcore power-resets the entire system. The threshold frequency and the size of the time window are parameters that need to be tuned to ensure proper system operation. There are two conflicting goals for this tuning: 1) endure that a system power reset is very unlikely to be triggered when the system is operating normally, and 2) minimize unnecessary delays in triggering a power reset when such a reset is necessary.

Another way in which the trusted hardcore is used is that if during self-diagnosis a replica receives different states from the different replicas and those states are also different from the receiver's state, the replica stops sending heartbeats to the trusted hardcore. If the trusted hardcore fails to receive heartbeats from two of the replicas, it power-resets the entire system. Based on this mechanism and the mechanism described in the previous paragraph, a malicious

primary cannot cause the system to perform the wrong function, but it can cause a power reset of the entire system.

Power-resetting the entire cluster is a costly recovery action that must not be performed often. The assumption is that the malicious behavior of a primary replica described above happens very rarely in practical systems. Based on this assumption, power-resetting the cluster is seldom required. The tradeoff between using this simple algorithm, as done in Ghidrah, and using the more complex algorithm described in Chapter 4, is a tradeoff between minimizing the overhead during fault-free execution and a high cost for recovery when this specific malicious behavior occurs. In a system that has tight hard real-time constraints, the correct choice may well be to use the algorithm described in Chapter 4.

7.1.2. Internal Structure of Key Components

This section describes the internal structure of two key components of Ghidrah —the manager and the agent.

7.1.2.1. Manager

Figure 7.1 depicts the internal structure of the manager. The functionalities of all modules that make up the manager are as follows:

- **Event Manager:** this module is the central point for detecting and gathering events, and dispatching events to proper modules for processing.
- **Timer:** it manages and schedules time-triggered events.

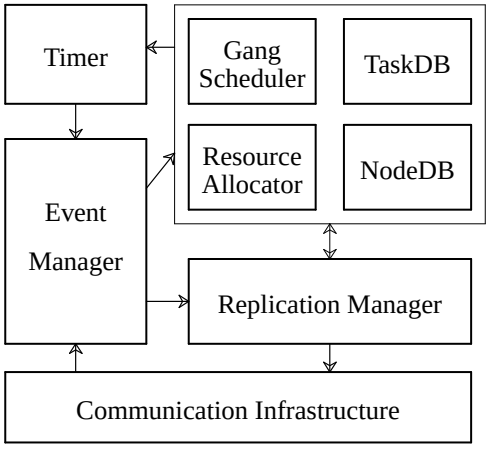


Figure 7.1: Internal structure of Ghidrah manager

- **Replication Manager:** this is the key module that implements the replication algorithm. It is responsible for ensuring consistency among the manager replicas and maintaining the replica group. It executes the BFT-SMR algorithm, as well as the self-diagnosis and reconfiguration algorithms.
- **NodeDB:** this is the database that stores and manages information about each individual node in the cluster.
- **TaskDB:** this is the database that keeps track of all the active application tasks and their processes.
- **Resource Allocator:** this module is responsible for allocating resources to application tasks.
- **Gang Scheduler:** this module performs gang scheduling on parallel applications.

7.1.2.2. Agent

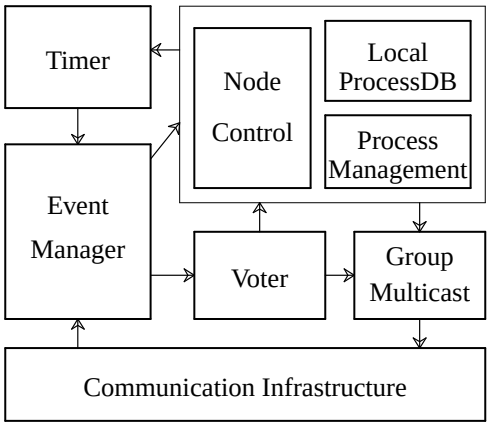


Figure 7.2: Internal structure of agent

Figure 7.2 shows the internal structure of an agent. Just like the manager, the agent also includes the Event Manager and the Timer modules, which have the same functions as in the manager. The functionalities of other agent modules are summarized below:

- **Voter:** this module performs comparison and voting on the messages received from the manager replicas.
- **Group Multicast:** this module multicasts messages to the manager replicas that make up the manager group.
- **Node Control:** this is the local entity that controls the node. It monitors the resources on the node and collects information about them. It is also responsible for sending heartbeats to the manager group. and to the keeper (Section 6.1).

- **Process Management:** this module manages and interacts with all the application processes that are running on the local node. It launches application processes, control them, and monitors the status of the processes.
- **Local ProcessDB:** this is the database that holds information about all application processes running on the node.

7.1.3. Event Handling

Ghidrah's key components, the manager replicas and the agents, are event-driven. Operations are invoked when events are triggered. Events are either message-based (triggered by an external message) or timer-based (triggered by the local timer). Event handling is based on the publish-subscribe model. Each module subscribes to the events that it is interested in from the Event Manager (Section 7.1.2) and registers a handler to each event type. The CI generates message arrive events when it receives messages from other components, and the Timer fires timer events. These events are published to the Event Manager. The Event Manager maintains an event queue of all published events and dispatches triggered events by invoking the handlers bound to them. The bindings between events and handlers are usually static and created when the modules are initialized. The Event Manager also allows dynamic bindings so a module can change and replace its handlers to events.

Figure 7.3 shows the Event Manager's main event loop. The processing of the events is priority-based. When an event is received, the Event Manager assigns a

```
BEGIN LOOP
    IF there are any expired timer events THEN
        push the event into event queue;
    IF there are any received messages THEN
        push the message into event queue;
    IF the event queue is not empty THEN
        pop out an event with the highest priority;
        process the event;
    ELSE
        block until new event received;
END LOOP
```

Figure 7.3: Event handling in event manager

priority to it and inserts it into the event queue based on its priority. After processing an event, the Event Manager polls the CI and the Timer in case there are new events triggered. When there is no event pending, the Event Manager blocks until it is woken up by an interrupt caused by new events.

Events are processed without preemption. This can be problematic if the handling of some low priority events is time consuming and during processing new events with higher priorities are received. In order to handle this case efficiently, we developed a lightweight user-level thread library that allows a low priority event handler to periodically yield the control to the Event Manager to check and process new events. After processing those event, the Event Manager will return control back to the paused handler thread to continue its processing.

7.1.4. Group Timer Events

The operation of the managers is driven not only by incoming group messages from agents, but also by events triggered by time. Examples of these time-triggered events are an invocation of the periodic gang-scheduling operation, or a timeout triggered when a message sent from the manager group to an agent is not acknowledged. The timer events must be processed by all manager replicas in the same total order with respect to other timer events and with respect to message arrival event. Hence, the timer events must pass through the sequencer (the primary replica) in the same way that messages do (Chapter 4). Thus, as with message arrival events (Section 4.2), *linearizability* with respect to these *group timer events* must be satisfied.

In order to achieve consistent total order on group timer events *and* messages, a group timer event is only triggered by the primary. The primary creates a special message for the event and assigns a sequence number (*RSN*) to the message. It then multicast this message (called group timer event message) to the backup replicas using the total order multicast protocol described in Chapter 4.

The problem with the above approach is that if the primary replica fails, the timer event may never be processed. This problem is solved by scheduling a ‘‘backup timer events’’ on each backup replica with the backup replica’s own timer. This backup timer event is scheduled with some delay (called ‘‘*late-threshold*’’) added to the original time interval of the group timer event. Under normal circumstances, the group timer event is triggered on the primary replica and then

multicasted to the backup replicas before the backup timer events are fired. The backup replicas then dismiss the backup timer event and process the event.

A backup timer event fires on a backup replica only if the backup replica fails to receive the corresponding group timer event message from the primary before the late-threshold. This implies that the primary replica may be faulty. The backup replica sends a message to the other backup replica to report the error. If the other backup replica agrees that the group timer event is late, it responds with a positive acknowledgment. The two backup replicas then decide that the primary is faulty and initiate the manager reconfiguration procedure to remove it (see Section 5.3).

There is also a possibility that a faulty primary replica sends a group timer event message to one or both of the backup replicas ahead of the scheduled time. The backup replicas detect this primary replica failure by setting an "*early threshold*" for each group timer event. When a backup replica receives a group timer event message from the primary, it checks its clock and determines whether the event is early. If it is early, the backup replica forwards the group timer event message to the other backup, along with a proposal to declare the primary as faulty. When the other backup replica receives this message, it checks the event with its clock. If it agrees that the event is early, it suspects the primary as faulty as well and sends a positive acknowledgment to the other backup's proposal. The two backups then invoke the reconfiguration procedure to remove the primary. If the second backup decides that the event is not early, it sends a negative acknowledgment to the other backup and processes the group timer event normally.

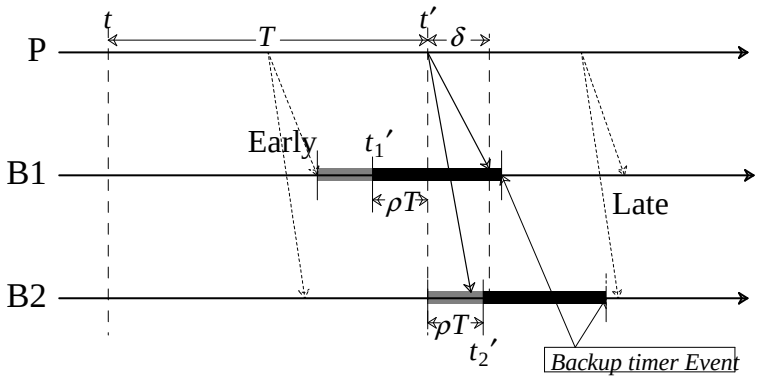


Figure 7.4: Detect late and early group timer events. P is the primary, $B1$ and $B2$ are backup replicas. The backup replicas schedule the backup timer event with a delay added to the time interval T of the group timer event. They also set the early-threshold to detect that the primary fires the group timer event too early.

The detection of late or early group timer events is illustrated in Figure 7.4. The primary replica schedules the group timer event using the original time interval T. The backup replicas schedule the backup timer events with the late-threshold θ_L added to T. θ_L is calculated based on the maximum relative drift rate ρ between the clocks of the replicas and the one-way transmission timeout delay δ, so that $\theta_L > \rho T + \delta$. The value of δ is chosen in a way such that messages are very *likely* to be delivered to destinations within the time interval of δ.

The backups also calculate an early-threshold θ_E for each group timer event such that $\theta_E = \rho T$. A group timer event message sent by the primary is considered as ‘‘early’’ by a backup if the backup receives it before $T - \theta_E$.

During a view change or a reconfiguration, when a backup replica becomes the primary, it must reset all its backup timer events back to primary group timer events with the proper original time interval T. It does this by withdrawing a

backup timer event, subtracting the late-threshold θ_L from the time interval, and re-scheduling the event as a primary timer event.

All the currently scheduled group timer events are part of state of a manager replica, so they must be included in the state comparison during a self-diagnosis and in the state transferred to the new manager replica during a reconfiguration. Because the system clocks of all manager replicas are slightly different and the primary and backups schedule group timer events differently as described above, the absolute time of a group timer event scheduled on different replicas are different. Therefore, we cannot include the absolute time of a group timer event in a replica's state. Instead, we only include the relative time interval T of such an event in the state, so that the states of fault-free replicas are consistent for state comparison and correct state is transferred to a new replica during reconfiguration.

When the new replica receives these group timer events with the state transferred to it, it schedules the events as backup timer events (the new replica is always a backup). Because for each group time event, this new replica only knows the original time interval T but does not know the exact remaining time until the real time that the group timer event should be triggered, it schedules the event using its current clock time as the starting point. This causes the problem that this new replica schedules the backup timer events later than they should be scheduled. Therefore, if the primary is late on triggering such an event, this new replica may not detect the event as late. However, if the primary does not trigger the event at all, this new replica will eventually detect this failure and agree with the other

replica that the primary is faulty. Therefore, the group timer events will not be lost.

However, since this new replica schedules these transferred group timer event late, it is possible that even if the primary triggers these events correctly, the new replica would consider the group timer event messages from the primary to be ‘‘early’’. In order to deal with this problem, when receiving a group timer event messages from the primary, this new replica does not check whether the message is early if the event is a transferred group timer event. This can lead to the problem that an early group timer event may not be detected correctly. In practice, the number of transferred group timer events are usually small, so the problem will not have significant effect. All group timer events scheduled after the new replica restores its state are handled correctly.

7.1.5. Agent Heartbeats

As described in Section 6.1, on each node, the agent keeper can recover the agent from crash failures. If both the agent and the agent keeper crash on a node, the node becomes unavailable. Heartbeats are used to detect crash failures of an agent/node. The agent sends periodic heartbeats to the manager group. Missing heartbeats serve as hints that a node has crashed. The manager group verifies the failure of the node by probing the node with a ‘‘ping’’ message to the agent. If no response from the agent is received before a timeout period expires, the manager group declares the node as ‘‘officially dead’’, excludes the node from the cluster, and takes the actions required for the loss of application processes running on the

node (e.g., informing the processes of each task running on different nodes). The manager group then sends to the trusted hardcore a request to power reset the failed node.

The implementation of the agent heartbeat mechanism is driven by two considerations: minimizing false alarms and minimizing the overhead of heartbeats. To minimize false alarms, the manager group suspects a node only when two consecutive heartbeats from the node are lost. Furthermore, the manager group then verifies the suspicion by probing the agent —sending the agent a message that requires the agent to respond. This probing minimizes that chances of false failure detection that might be triggered by lost heartbeats were due to temporary communication errors.

The heartbeat frequency is a tradeoff between fast detection of failures and overhead during normal operation. High heartbeat rate decreases the detection latency but increase the overhead introduced by heartbeat messages, especially, the overhead incurred on the manager replicas for receiving all the agent heartbeat messages. The overhead consists of the time for processing the heartbeat messages, as well as the OS context switch overhead, because the manager processes have to preempt the application processes running on the same nodes. This context switch overhead can be reduced if multiple heartbeat messages can be ‘‘bundled’’ into a single message. One way to do this is for all the agent heartbeat messages to be sent to one of the manager replicas which then bundles these messages into a single message and forwards the bundle message to other manager replicas.

Since heartbeat messages are authenticated, a faulty manager replica cannot forge agents' heartbeat messages. However, a faulty manager replica may refuse to forward the heartbeat messages, thus causing the other manager replicas to suspect some working nodes. In addition, if all heartbeat messages are forwarded by the same manager replica, that leads to an uneven distribution of the management overhead over the nodes running the manager replicas.

Based on the arguments above, the Ghidrah CMM uses the two backup manager replicas to bundle and forward agent heartbeat messages. The primary replica is not used for forwarding heartbeats since it already has the extra overhead for ordering normal group messages. Each agent alternates between the two backup replicas as destinations for its heartbeat messages. When a backup replica receives a heartbeat message from an agent, it saves the message in a buffer. A periodic timer event (not a group timer event) is scheduled on each replica. At the end of each period, a backup replica collects all heartbeat messages in the buffer, forwards them to the other two replicas as a ‘‘bundled’’ message, then clears the buffer. When a replica receives this message, it extracts every heartbeat message in the bundled message and records that a heartbeat had been received from the corresponding agent. Each replica (include the primary) checks the heartbeats from every agent at the end of the same time period as well. If two consecutive heartbeats are missing from an agent, a replica suspects a node failure and initiates probing the node.

Because the system clocks of manager replicas are different, the timer event

described above is scheduled at different real time on different manager replicas. Hence, different manager replicas suspect a node at different real time. This suspicion is local on a replica and not part of the manager state that is compared during self diagnosis. Thus, this difference among replicas does not cause false replica failure detection during self diagnosis. However, the initiation of probing a node must be synchronized on all replicas, because the probing is a group action and affects the state of the manager group. This synchronization is achieved by scheduling the probing as a group timer event (see Section 7.1.4) with a zero time period. When a manager replica suspects a node failure, it sends a message to the other replicas to announce the suspicion. When receiving this announcement, a replica forwards it to the other replica. When a replica has received the same announcement the has originated from at least two replicas (may include itself), it schedules the group timer event. The primary replica then triggers the event as described in Section 7.1.4, which initiates the probing to confirm that the node is dead.

7.1.6. The Spacecraft Control Computer

The Spacecraft Control Computer (SCC) controls the entire spacecraft and is critical to the space mission. Loss of SCC means loss of the spacecraft. For this reason, SCC is built with radiation-hard hardware, with limited processing and communication capability. In Ghidrah, SCC is used as the trusted hardcore (see Section 3.4).

As discussed in section 3.4, the main functionality of the trusted hardcore is to power reset any of the cluster nodes or the entire cluster, when necessary. In order to determine when these power reset actions are necessary, the SCC must interact with the manager replicas. It communicates with the manager replicas in the way a client communicates with replicated servers as described in Section 4.2.1.

In Ghidrah, the failure of a node is detected by the manager replicas when heartbeats are missing from the agent on the node (see subsection 7.1.5). To recover from a node failure, the manager replicas sends a command to SCC to request a power-reset on the node. When SCC receives same command from at least two manager replicas, it executes the command and resets the particular node.

The SCC also monitors the manager replicas and detects their abnormal behavior in case the replicated managers have failed. To ensure that the system has an operational manager group, all manager replicas send periodic heartbeats to the SCC. If the SCC does not receive heartbeats from at least two manager replicas for a certain period of time, it concludes that the replicated managers have failed and power-resets the entire system.

It is also possible that instead of not sending heartbeats to the SCC, the manager replicas fail in such a way that they keep invoking the self-diagnosis and reconfiguration procedure. To handle this problem, every time the manager replicas invoke self-diagnosis, they report it to the SCC. The SCC records the rate of manager diagnosis. If the rate exceeds a certain threshold, it power resets the entire cluster.

Base on the discussions above, we summarize the required functionality of the SCC, i.e., the trusted hardcore in the CMM, as follows:

- Receive and authenticate group messages from the manager group requesting a power reset of a specific node. Perform the power reset of the specific node.
- Receive, authenticate, and record group messages from the manager group with notification of self-diagnosis. Power-reset the entire cluster if the rate during a specified time window exceeds a threshold.
- Receive and authenticate periodic heartbeat messages from individual manager replicas. Power rest the cluster if heartbeats from at least two manager replicas are not received.

In the current implementation of Ghidrah, we have not implemented a fully-functional SCC with the ability to physically power-reset nodes or the cluster, because this ability requires special hardware. Instead, we implemented an emulated SCC that has the functionality of interacting with the manager replicas as described above. When a power-reset is needed, this emulated SCC emulates the reset operation by outputting a display message that says ‘‘power reset node *i*’’ or ‘‘power reset the entire system’’.

7.1.7. Implementation of the Communication Infrastructure

In section 3.5, we described the communication infrastructure (CI) that provides reliable communication in the CMM. Figure 7.5 depicts the structure of

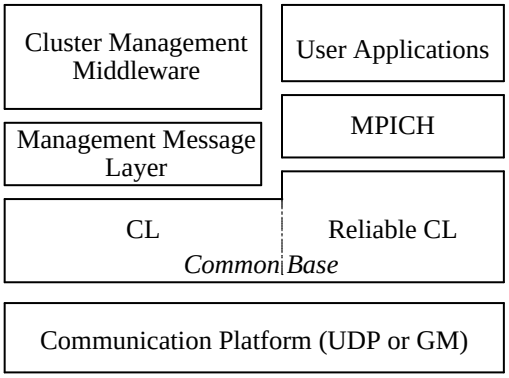


Figure 7.5: Communication infrastructure of Ghidrah

the CI implemented in Ghidrah. The infrastructure is composed of a common base called CL (Communication Layer), a RCL (Reliable CL) module that provides reliable message passing to parallel applications, and an MML (Management Message Layer) module for communication between key CMM components such as managers and agents [Li02].

CL is a light-weight layer that is portable across multiple network platforms and provides efficient message communication with minimized overhead imposed on top of the underlying network platform. High-performance communication requires avoiding system calls and eliminating local message copies [Stee94]. Hence, modern designs of NIC (network interface card) hardware allow user-level send and receive primitives, allow the NIC and host to access shared memory, and expose buffer management to the application [von98]. The design of the Ghidrah CI was done with these considerations in mind —if the underlying communication platform provides these high performance features, our CI must be able to take advantage of them. For example, like many high-speed communication systems,

CL also exposes its buffer management to applications.

CL can be easily ported to high-speed user-level network platforms as well as traditional network platforms. To demonstrate its portability, we have implemented it on top of both UDP/IP and Myrinet/GM [Bode95].

CL provides connectionless communication with no reliability guarantees. It cannot be directly used for application-level communication because applications require point-to-point reliability. Hence, we implemented the Reliable CL to provides reliable message communication to applications. RCL ensures message reliability using timeouts and retransmissions. MPICH [Grop96] is ported on top of the RCL to provide a parallel programming environment in the cluster.

The main design objective of RCL is to ensure high performance. For this reason, we decided not to implement RCL as a layer strictly above CL. Such strict layering might lead to inefficiencies because the reliable communication routines in the upper layer have no direct access to the internal date structure of lower layer. For example, if we implement RCL as a separate layer above CL, which in turn is built on top of Myrinet/GM, RCL would have to perform periodic probing to the status of the send of some control packets, such as message acknowledgments, in order to determine when the buffers can be reclaimed. Our implementation of RCL is integrated with the implementation of CL, so the reliable communication routines have full access to the internals of CL. In the example above, this allows RCL to garbage-collect the status data structures and release the control packet buffers in the callback routines that are called by GM when the packets are successfully sent

out, so there is no need for probing the the send status.

As discussed in Section 3.3, communication between managers and agents requires adaptive reliability. For this reason, we designed the MML to supports three types of messages:

- UNR —unreliable message for which no acknowledgment is required;
- ACK —reliable message that requires a positive acknowledgment;
- NACK — reliable message that does not require a positive acknowledgment (negative acknowledgments are used to flag lost messages).

Function	Operation
msg_connect(dest)	establish a connection to *dest*.
msg_disconnect(dest)	terminate the connection to *dest*.
msg_set_protocol(msg,ptcl)	set the protocol for *msg* to *ptcl* (UNR, ACK, or NACK).
msg_sync_send(dest,msg)	send *msg* to *dest* (blocking); return assigned sequence number.
msg_async_send(dest,msg,status)	send *msg* to *dest* (non-blocking); return assigned sequence number.
msg_check_status(status)	check for completion of a non-blocking send.
msg_receive()	receive all pending messages.
msg_resend(dest,seq)	retransmit the message having sequence number *seq* to *dest*.
msg_group_ack(dest)	send an ack for a multicast message.

Table 7.2: API of the management message layer (MML)

Table 7.2 lists the API of MML. MML is tightly-coupled with the management middleware and is strictly layered on top of CL. MML messages are

authenticated with digital signatures (see Section 3.5). In order to reduce the overhead of message authentication, MML uses the MD5 message-digest algorithm [Rive92], to compress a message of arbitrary length into a small, fixed-length message digest. MML signs the digest of a message instead of signing the entire message.

7.2. Experimental Results

This section reports the results of experimental performance evaluation and fault tolerance validation of the Ghidrah CMM.

7.2.1. Experimental Setup

Our experimental evaluation of the Ghidrah CMM was carried out on two clusters. The hardware configurations of the two clusters are summarized in Table 7.3.

	Cluster A	**Cluster B**
Number of Nodes	4	4
CPU	Pentium-II 350MHz	Xeon 2.66GHz
L2 Cache	512KB	512KB
Physical Memory	384MB	1GB
Network	100Mb Ethernet	Gitabit Ethernet

Table 7.3: System configurations of the two experimental clusters

The first cluster (Cluster A) consists of four nodes. Each node has a Pentium-II 350 MHz processor with 512KB L2 cache and 384MB of main memory.

The nodes are connected by a 100Mb Ethernet LAN. The second cluster (Cluster B) consists of four nodes with an Intel Xeon 2.66GHz processor and 512KB L2 cache on each node. Each node has 1GB of memory. The network is a Gigabit Ethernet LAN. All nodes in the two clusters run the Linux Red Hat 9 operating system with version 2.4 of the Linux kernel. The CMM communication infrastructure (CI) is implemented on top of UDP/IP.

7.2.2. Performance Measurements

This section presents performance and overhead measurement results for the Ghidrah CMM.

7.2.2.1. Manager Overhead in Normal Case

The processing and associated power consumption of the manager group and agents are system overhead that should be minimized. Since the manager replicas and the agents run on the same nodes that run processes of application tasks, the key measure of overhead is the fraction of CPU processing power that is not available to processes of application tasks. The overhead is caused by context switches (to invoke the manager or agent) and the time the manager or agent spends on receiving and processing messages and timer events. Since each manager replica processes many more messages and timer events that any individual agent, our evaluation focused on the manager replicas. Specifically, we measured the fraction of ‘‘processing’’ that becomes unavailable to application processes while

an active manager replica runs on the same node.

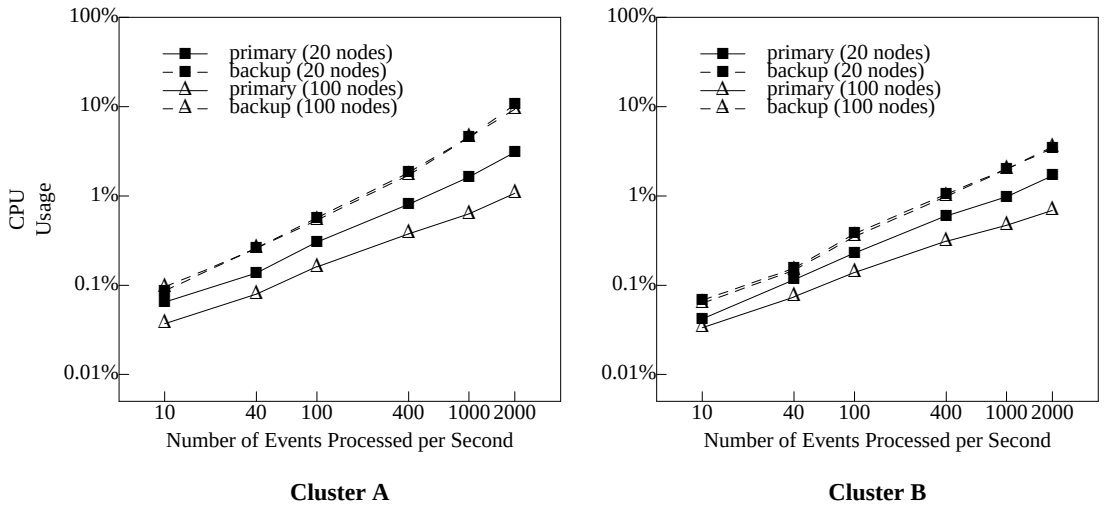


Figure 7.6: Heartbeat processing overhead on nodes running manager replicas. The Y axis (CPU Usage) is the fraction of CPU cycles that is not available for application processes.

The manager replica overhead largely depends on the frequency of heartbeats that it must handle. Figure 7.6 shows the percentage of processor time used by a manager replica for different event processing rates. The event processing rate is based on the total number of events processed by the manager group and is thus the product of the agent heartbeat rate and the number of agents. Using our small four-node clusters, we emulated larger clusters by running multiple emulated agent processes on each node, thus generating a large number of heartbeats. The first system consists of 20 agents and thus 1000 heartbeats per second are generated if each agent sends a heartbeat every 20 milliseconds. The second system consists of 100 agents and thus 1000 heartbeats per second are generated if each agent sends a

heartbeat every 100 milliseconds. Each experiment was repeated five times and the average measurement results are reported. The results concerning the backup managers are the averages of the results measured for the two backup managers.

Figure 7.6 shows that the overhead of the Ghidrah CMM is quite low under normal workload. For example, for a cluster with 100 nodes, where each agent generate a heartbeat message every 100ms, during normal operation, the overhead of running a manager replica on a node is less than 5% on Cluster A, and less than 2% on Cluster B.

Figure 7.6 also shows that ‘‘bundling’’ of heartbeat messages (see Section 7.1.5) reduces the processing overhead significantly —the overhead of the primary replica is much lower compared to the overhead of the backup replicas. For a fixed rate of heartbeats from the entire system, a larger number of nodes is handled more efficiently than a smaller number of nodes. The reason for this is that with the larger number of nodes, the larger number of agent heartbeats are bundled together by the backup replicas. Hence the primary replica is invoked less frequently by receiving the bundled heartbeats, so the overhead on context switch and receiving messages is reduced.

Another major source of manager overhead is the processing of group timer events (Section 7.1.4). Figure 7.7 shows the overhead of manager replicas for handling group timer events. This overhead includes the operating system’s timer management overhead. In addition, on the primary manager replica, it also includes the overhead caused by triggering a time event, sending group timer event

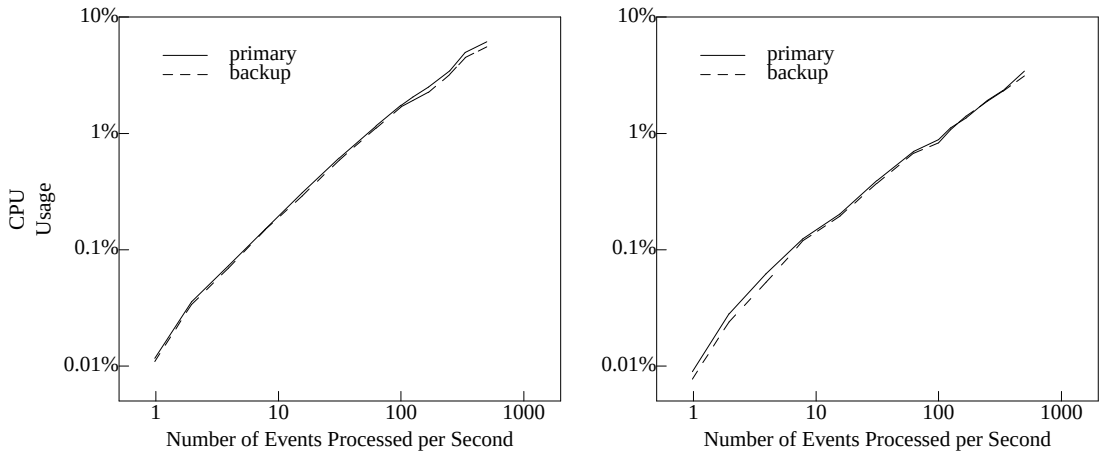


Figure 7.7: The overhead for handling timer events on manager replicas. The Y axis (CPU Usage) is the fraction of CPU cycles that is not available for application processes.

messages to the backup replicas, and invoking the callback registered to the event. For a backup manager replica, the overhead includes the time for receiving the group timer event message from the primary, checking if the event is early, dismissing the backup timer event, and invoking the callback.

7.2.2.2. Manager Recovery Time

When a manager replica fails, the system loses its ability to mask a failure of another manager replica. Hence, a new replica must be instantiated as quickly as possible. We measured the recovery time for a manager replica —the time interval from when a failed manager replica is identified by self-diagnosis to the time that a new manager starts and the manager group restores its normal operation (see

Recovery Step	**Time** (msec)	
	Cluster A	**Cluster B**
Elect new primary manager	2.08	0.17
Sync with other manager	0.08	0.01
Take snapshot	0.07	0.01
Notify agents	3.50	0.74
Start new manager	3.20 + 21‡	0.65 + 5.2‡
Transfer state	1.67	1.06
New manager restore state	2.05	0.21
Mew manager catch up	0.96	0.13
Total	**34.61**	**8.18**

Table 7.4: Primary manager replica recovery time (excluding the time to load the executable from disk). ‡ Time for initializing communication infrastructure.

Section 5.3). The results are shown in Table 7.4. The time reported excludes the operating system time to load the manager program from disk. Most of the time is spent on initializing the communication infrastructure. This includes the time for reading the node configuration, initializing the sockets and constructing the data structures in the CI (buffers, receive queues, etc.). Hence, the recovery time can be reduced significantly if the system keeps on a node an initialized ‘‘cold’’ manager replica, which is ready to accept the state from other manager replicas and become active.

During recovery from a manager replica failure, the remaining replicas continue providing the management functionality. The primary replica continues to receive messages from agents, assign sequence number and forward them to the remaining backup replica. Both replicas process the messages and send responses

back to agents. These messages will later be sent to the new manager replica as ‘‘catchup’’ messages (see Section 5.3).

However, if the failed replica is the primary, communication from the agents will fail for a short period since the agents continue to send messages to the primary until they are informed about the new primary replica. Our preliminary measurements show that, once the failure of the primary replica is detected, the time to elect a new primary and advertise the the new primary to agent is approximately 2.1 milliseconds on Cluster A, and only 0.17 milliseconds on Cluster B (first row of Table 7.4). Hence, communication from the agents to the manager group is restored after a very short interruption, requiring, at most, a retransmission of a few agent messages by the reliability features of MML (Section 7.1.7).

As shown in Table 7.4, the total recovery time of primary replica failures is about 34.6 milliseconds plus the time for the operating system to load the manager program on Cluster A. On Cluster B, the time is 8.18 milliseconds plus manager program loading time. Most of the recovery time is spent on notifying all the agents, starting the new manager replica, and transferring state to the new replica.

For recovery from backup replica failures, the step of electing a new primary is not needed. Therefore, the time reported in the first row of Table 7.4 is excluded. The approximate recovery time from backup manager failures on Cluster A is 32.5 milliseconds plus the time for the OS to load the program. On Cluster B, it is around 8 milliseconds plus the program loading time. In this case, there is no

interruption to the communication from the agents to the manager group.

7.2.3. Fault Injection Experiments

In order to evaluate the reliability of the Ghidrah CMM, we stressed its fault detection and recovery mechanisms by injecting faults into the managers and agents, as well as the communication infrastructure. In this section, we present the results of these fault injection experiments.

7.2.3.1. Process-Based Fault Injections

The process-based fault injection campaigns we conducted were based on injecting single-bit-flip faults into a process's registers and memory space. We used a simple software-implemented fault injector [Hsue97]. This fault injector is based on a device driver linked into the operating system kernel. The fault trigger is time-based [Stot00]: faults are injected at some interval (a random variable) with a given average frequency (fault rate). When a fault is triggered, depending on its configuration, the injector randomly selects a register used by the process or randomly selects an address in the stack or heap segments. It injects a fault by flipping a random bit at that register or memory location. Faults are injected to the process at the given frequency until an error is observed. Injections are resumed after the manager/agent recovers from the failure.

Table 7.5 summarizes the results of fault injections when faults are injected into the registers, the stack and heap segments of an individual manager/agent

Target	Faults Injected	Errors Observed	Error Types				
			seg. fault	illegal inst.	asser -tion	keeper miss HBs	self-diag **M** + **I** ‡
Register Injections							
manager	3054	891	523	7	59	-	13 + 289
agent	3096	630	512	5	60	53	-
Stack Space Injections							
manager	3980	191	143	4	11	-	14 + 19
agent	4037	151	128	3	16	4	-
Heap Space Injections							
manager	3769	136	67	0	12	-	7 + 50
agent	3965	98	94	0	4	0	-

Table 7.5: Single process fault injection results. ‡The rightmost column is for faults that caused the managers to enter self-diagnoses: (**M**) — missing manager heartbeats, (**I**) — agents reporting inconsistent or missing messages from a manager replica.

process. The experiments were conducted on Cluster A. In these experiments, the average fault injection rate is one injection per second, and the managers and agents send out heartbeats every 100 milliseconds. The system is loaded with a synthetic workload (10-20 jobs with different parallel degrees and execution times, submitted at random points in time). These jobs are gang scheduled with a granularity of three seconds.

The errors caused by the injected faults are classified into several categories. For both manager replicas and for agents the categories include: crashes due to segmentation faults, crashes due to illegal instructions, program aborts due to assertions in Ghidrah code. For manager replicas, the error categories also include missing manager heartbeats or incorrect/missing output from one of the manager

replicas. For agents, the error categories also include agent failures detected by the keeper due to missing heartbeats. These agent failures were not agent crashes and were caused by agent hangs or some other agent failure that prevented it from incrementing the heartbeat counter (see Section 7.1.5). In Table 7.5, the third column shows the total number of observed errors caused by the faults injected. The breakdown of these errors into the different categories is shown in the rest of the columns to the right.

We collected these results by monitoring the exit status of the manager and agent processes, as well as monitoring the operation of the agent keeper (for agent injections) and the self-diagnoses conducted by the manager replicas (for manager injections).

If injected faults cause the process to crash due to segmentation faults or illegal instructions, or to abort, a corresponding signal is delivered to the process and the process terminates with an exit status indicating which signal. We then categorize the error based on the exit status. In Table 7.5, the fourth and fifth columns report the errors due to segmentation faults and illegal instructions. The Ghidrah code includes some assertions to facilitate debugging. Examples of these assertions are checking for NULL pointers and checking whether the value of critical variables is within the legal range. The aborts triggered by these assertions are reported in the sixth column of Table 7.5.

If an agent process does not terminate with the exit status described above, but the agent keeper on the same node reports that the heartbeats from the agent are

missing so it kills the agent process and restarts a new one, we categorize the error as agent hangs. Errors of this type are reported in the seventh column of Table 7.5.

For a manager process, if it does not terminate with the exit status described above, but the other manager replicas declare it as faulty after the self-diagnosis procedure, we categorize the error as a non-crash error. Due to the faults injected, the faulty manager replica may hang or fail to send valid heartbeats to other manager replicas, Such an error is detected by the other manager replicas that miss heartbeats from the faulty replica. An injected fault may also cause a manager replica's internal state to be corrupted, so it sends incorrect output messages to an agent or it fails to send output messages to an agent. Such an error is detected by an agent that reports inconsistent or missing output from the faulty replica. In both cases, manager self-diagnosis is invoked to identify the faulty replica. We separate these non-crash errors with respect to the reason that causes the self-diagnoses. The counts of these errors are reported in the rightmost column of Table 7.5.

It should be noted that when a manager replica hangs due to faults, the error can be detected by other manager replicas as well as by an agent. The other manager replicas detect it because of missing manager heartbeats, and the agent detects it because it has received a message from the other two replica but misses the message from the faulty replica. In our implementation, the time period for the manager replicas to check for missing manager heartbeats is shorter than the timeout period an agent sets on a missing manager output message. Hence the manager replicas detect missing heartbeats and invoke the self-diagnosis before the

agent reports a missing manager output message. Therefore, this type of errors are categorized as ‘‘missing manager heartbeats’’ in the rightmost column of Table 7.5.

The results in Table 7.5 show that most register faults injected led to segmentation faults. This is because that many registers in a Pentium II processor are used for base-index addressing and for special purposes such as stack pointer. Injecting faults into these registers is likely to lead to memory accesses to invalid addresses and thus cause segmentation faults. Similarly, a large portion of faults injected into the stack space and heap space also caused segmentation faults that crashed the process. It is also shown that stack space faults and heap space faults were less likely to cause observed errors compared to register faults. For a manager replica, errors caused by faults injected into the heap space were more likely to produce incorrect outputs than errors caused by faults injected into the stack space. This is reasonable since faults in the heap space are more likely to corrupt the state of the manager replica.

Our CMM system recovered from all the errors (process failures) reported above. When faults injected into a manager replica caused the replica to fail, the diagnosis and reconfiguration procedure successfully started a new manager replica on another node. When an agent process crashed or hung, the agent keeper successfully started a new agent process.

We also measured the detection latency of errors caused by faults injected into a manager replica on Cluster A. In each experiment, we injected a single fault into the manager replica, then measured the time interval from the moment of the

injection to the moment that the injected manager replica is identified as faulty by other manager replicas. Because it is possible that a fault injected may not cause any error in the manager replica, we stopped the experiment ten minutes after the injection, assuming that the fault would never cause an error. The system clocks of all nodes that ran the manager replicas were synchronized using NTP so that we could have relatively accurate measurements of time intervals on different nodes.

Type of Errors	Detection Latency (msec)			
	min	*avg*	*max*	*80%*
crash errors	320	390	980	350-470
non-crash errors	270	590	4,900	410-620

Table 7.6: Detection latency of faults injected into a manager. The results are collected from ~500 register injections and ~1200 heap space injections. 126 crash errors and 43 non-crash errors are observed.

The detection latency measurement results are presented in Table 7.6. The table shows the minimum latency, the average latency and the maximum latency for all cases where an errors was detected. Also shown in the table is a range of detection latencies that contains eighty percent of the detected errors. This range is obtained by excluding the lowest ten percent and the highest ten percent of the detection latency results.

We separate the results we measured for crash errors from the results for non-crash errors. All crash errors were detected by the other two manager replicas with missing heartbeats. In these experiments, the heartbeat period was set to 100 milliseconds. Other manager replicas suspected the crashed replica after missing

two consecutive heartbeats from it. They then invoked the self-diagnosis procedure to verify the replica failure. During self-diagnosis, no diagnosis message was sent by the crashed replica, so the other two replicas declared it as faulty after timeout expired on waiting for RSN message from the crashed replica. In these experiment, the RSN message timeout period was set to 200 milliseconds. Based on the discussion above, a correct replica identifies a replica failure only after two consecutive 100-millisecond heartbeats are missed, triggering self-diagnosis, followed by an RSN message timeout (200 milliseconds) during self diagnosis. Therefore, the detection latency for most of the fail-stop failures was in the range of 350-470 milliseconds,

Non-crash replica failures include instances when the faulty manager replica sends incorrect outputs to an agent, as well as instances when the faulty manager replica fails to send an output message to an agent. The first case was quickly detected by the agent when it compared the manager replicas' outputs. It then reported this error to the manager replicas and self-diagnosis was invoked. Because the faulty replica was still alive and sent diagnosis messages to others, the self-diagnosis procedure identified the faulty replica quickly in these cases.

In the case where the faulty manager failed to send an output to an agent, the agent waited for the message until the timeout period for late message expired. It then reported a ''late or missing message'' error to the manager replicas. The manager replicas invoked self-diagnosis and identified the faulty replica as the faulty replica's state was corrupt due to the fault injected. The timeout period was

set to 500 milliseconds in the experiments. Most of the measured detection latencies were close to the 500-millisecond timeout period, indicating that many faults injected caused the faulty replica not to send an output message. This type of non-crash errors also include errors that the faulty replica sent a corrupt message to an agent, since this message was rejected by the communication infrastructure at the receiver side and was never delivered to the receiver.

Some of the faults that cause non-crash errors can be dormant for a long time before they cause observable errors. The longest latency we measured is close to 5 seconds. Such a fault corrupts some variable in the replica's state, but the variable is not be used soon after the corruption.

The above experiments only demonstrate what happens when faults are injected into a single process. In practice, however, all the processes are subject to faults simultaneously. Our replicated managers are designed with the assumption that no more than f of the replicas are faulty at the same moment. However, if the fault rate is too high thus more than f replicas become faulty before the replicated manager recovers from previous faults, the replicated managers may fail. In this case, SCC has to step in to power-reset the entire system, as described in Section 7.1.6.

We ran experiments on Cluster A to examine the survivability of the replicated managers under extremely high fault rate. Faults were injected into all three manager replicas simultaneously. In the original configuration of the Ghidrah system the manager group enters self-diagnosis only when some error is detected.

For these experiments, the system is modified so that the manager replicas enters self-diagnosis periodically even if no errors are detected. The reason for this modification is that, in these experiments, the workload of the manager replicas is very light so that the replicas are mostly idle. The workload consists of about 10 applications with different parallel degrees and execution times that were submitted at random points in time. The gang scheduling interval is three seconds. Therefore, a fault injected to a manager replica may not exhibit as a detectable error quickly. In this situation, it is possible that multiple replicas becomes faulty before the faults are detected. In order to reduce this possibility the manager replicas are forced to enter self-diagnosis procedure periodically with a certain frequency to scrub latent errors. If the manager replicas invoke self-diagnoses with a frequency much higher than the frequency that faults occur on them, faults can be diagnosed and recovered from before they accumulate.

Figure 7.8 show the recovery coverage of the manager replicas from errors caused by faults injected into all replicas. Here the recovery coverage is defined as the percentage of errors from which the manager replicas can recover by themselves with respect to the total number of observed errors. This percentage is plotted on the Y axis in Figure 7.8.

The X axis in Figure 7.8 shows the ratio of fault injection rate to the rate of self-diagnoses periodically invoked by the manager replicas. Here the injection rate is the rate that faults are independently injected on each individual manager replica. This ratio indicates how frequent faults occur on manager replicas compared to

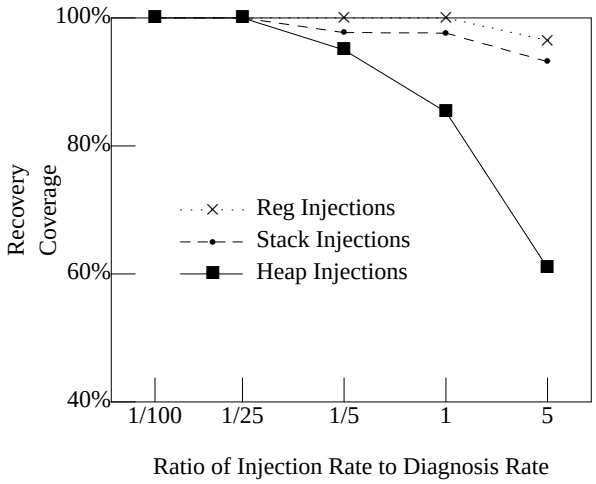


Figure 7.8: Recovery coverage of the replicated managers when faults are injected into all manager replicas simultaneously.

manager replicas invoking self-diagnoses. In the experiments, the rate of periodic manager self-diagnoses is one diagnosis per 2 seconds. The fault injection rate varies with respect to the different ratios plotted on the X axis. For example, for a ratio of 1/5, the fault injection rate is set to one fault injected per 10 seconds average on each manager replica.

For each ratio reported in Figure 7.8, about 40-50 experiments were conducted. In each experiment, faults were injected into all three manager replicas with the given injection rate until an error was detected. In some experiments, the manager replicas successfully recovered from the error with self-diagnosis and reconfiguration. In other experiments, the manager replicas failed to recover from the error and the SCC had to ‘‘power-reset’’ the cluster (see Section 7.1.6). The recovery coverage is calculated by dividing the number of experiments in which the

manager replicas recovered from the error by the total number of experiments.

The figure indicates that when the fault injection rate is much lower (less than 1/25) than the rate of manager self-diagnoses, the replicated managers always recover from the errors. When the fault injection rate is close to the self-diagnosis rate, the replicated managers still recover from *most* of the errors by themselves.

With faults injected into registers and the stack space, the replicated managers recover from most of errors even when the fault injection rate is higher than the self-diagnosis rate. The reason for this is that these faults are more likely to cause crash errors of manager replicas, compared to faults injected in the heap. Hence, a self-diagnoses is invoked soon due to missing heartbeats. The manager replicas identify the faulty replica quickly and recover from the fault before another replica becomes faulty.

With heap space faults, the recovery coverage is significantly lower than with register and stack space faults. In particular, when the *total* rate of fault injection is approximately the same as the rate of self-diagnosis, the coverage is approximately 90%. The reason for this is that these faults are more likely to corrupt replicas' internal states and the errors may not exhibit as detectable errors immediately. Hence the errors are detected only when the next periodic manager diagnosis is invoked. By that time, more replicas may become faulty due to the high fault rate, thus the replicated managers fail to recover.

These results provide useful information regarding the proper configuration of Ghidrah for different fault rate. Specifically, it is clear that the rate of scrubbing

manager replica state should be at least two orders of magnitude higher that the expected fault rate. For example, in low Earth orbit, an on-board cluster in a spacecraft may be impacted by an error every few hours. Thus, the interval between invocations of self-diagnosis should be on the order of tens of seconds. Since the time to perform self diagnosis is on the order of a few milliseconds, the overhead for doing such proactive self-diagnosis will be negligible.

7.2.3.2. Validation of the Reliability Features of the CI

In Section 7.1.7 we discussed the Communication Infrastructure (CI) that is the foundation of the Ghidrah CMM. Providing end-to-end reliable communication is a key goal of the CI. In order to evaluate and validate the reliability features of the CI, we have built a fault injector into the CI [Li02]. Based on a functional fault model for message communication, this injector can drop, delay, corrupt a message, change the order of messages, and inject duplicated messages. In the fault injection experiments, the injector randomly picks messages to inject based on a given fault rate per byte. The type of fault injected into each selected message is chosen randomly as well.

The communication in CMM includes both reliable messages and unreliable messages (mainly heartbeats). If a fault affects a reliable message transmission or the corresponding acknowledgment, the sender will retransmit the same message. Faults may also cause a process to receive duplicated messages. Faults injected into heartbeat messages from agents to the manager group may cause false alarms

regarding node failures. In particular, if the manager group misses two consecutive heartbeats, the node probing procedure will be invoked to check the health of the node. It is also possible that communication faults cause the managers to initiate the manager self-diagnosis procedure. This occurs when two consecutive heartbeat messages from a manager replica are lost or delayed, or messages from the manager replicas to agents are lost or delayed.

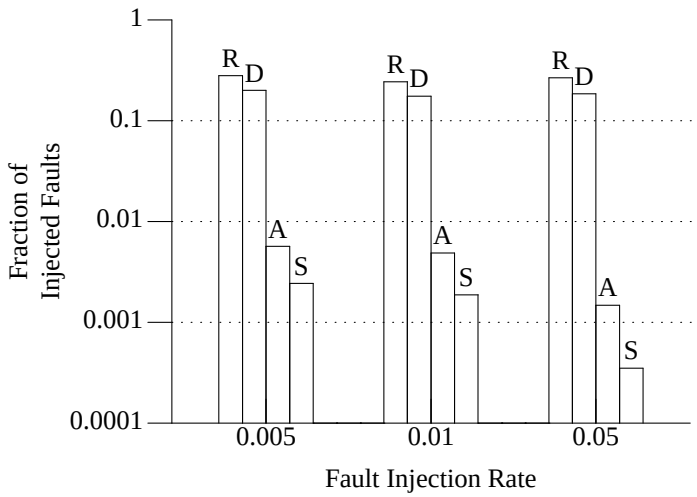


Figure 7.9: Effects of faults injected into the communication infrastructure. (**R**) Retransmissions; (**D**) Duplicated messages; (**A**) False alarms on node failure; (**S**) Manager self-diagnosis.

We ran fault injection experiments on Cluster A with the CI fault injector activated on all manager replicas and all agents. The system was loaded with a synthetic workload: 10 applications with different parallel degrees and execution times were submitted at random points in time. These applications were gang scheduled with a granularity of three seconds. Figure 7.9 shows the consequences

of communication faults under different fault injection rates. The injection rates in this case are per message. In each case, the number of consequences is normalized to the total number of faults injected. The normalized numbers of message retransmissions and duplicated messages are constant under different fault rates, indicating that the number of these events increases proportionally with the number of faults injected. However, with increasing fault rate there are smaller increases in the rate of node failure false alarms and the rate of managers entering self-diagnosis. The reason for this is that the CMM protocols are designed to mask (tolerate) some faults without resorting to high-level recovery or diagnosis actions. For example, an alarm regarding the possible failure of a node is raised only if there are *two consecutive* heartbeat messages missing or delayed. One lost message is tolerated with no high-level impact.

We also conducted experiments with communication faults injections on Cluster A to validate the fault-tolerant bootstrapping protocol described in Section 6.2. In these experiments, we injected faults into messages agents exchange during the bootstrapping procedure with a high injection rate of 50%, so that half of the messages were injected. The bootstrapping procedure completes successfully despite these faults. However, the average time to complete the bootstrapping increases to 112 milliseconds, compared to 13 milliseconds when the execution is fault-free.

Chapter Eight

Summary and Conclusions

This dissertation describes the design and implementation of fault-tolerant cluster management middleware (CMM) for clusters built with mostly COTS hardware and software and used for critical applications in hostile environments. The research presented in this dissertation identifies key issues and challenges in building such highly reliable CMM. In this context, the dissertation provides adaptations or improvements of several state-of-the-art fault tolerance techniques. These techniques are utilized in the implementation of an operational CMM system called Ghidrah.

The CMM system is based on centralized decision making, with a central cluster manager as well as an agent on every cluster node. However, unlike most other cluster middleware, the central manager in our CMM system is protected by state machine replication and the ability to restore the management service to full functionality and full fault tolerance following arbitrary single faults.

In this dissertation, we present an efficient Byzantine fault tolerant replication algorithm for reliable services. Compared to existing algorithms that all require at least $3f + 1$ active replicas to tolerate up to f faulty replicas, our algorithm requires only $2f + 1$ active replicas for normal operation. By reducing the number of active replicas, our replication algorithm is more efficient than previous BFT-SMR algorithms, as it reduces replication cost and improves performance. Furthermore,

we discussed the practical tradeoffs of using simplified versions of the replication algorithm in the CMM system.

We also developed a self-diagnosis algorithm based on a state machine fault model we defined for replicated systems. This self-diagnosis algorithm allows the system to identify the faulty replicas. Using the diagnosis result, the replicas can reconfigure themselves to replace the faulty replicas with new, faulty-free replicas. If the conditions under which the diagnosis algorithm is both complete and accurate hold, the system will keep replacing faulty replicas and thus may be able to tolerate an arbitrary number of faults during its lifetime.

In addition to the fault tolerance techniques we applied to the centralized manager, we used less aggressive fault tolerance techniques on agents, with the consideration that an agent is not critical to the continued correct operation of the entire system. We implemented in our CMM a low-cost mechanism for recovery from agent crash failures. This mechanism allows agents to recover from crash failures while maintaining control over application processes that have been running in the cluster.

We developed a fault-tolerant bootstrapping protocol that allows the CMM to automatically configure itself with a working manager group when the cluster is powered up.

We also discussed the necessity to have a trusted hardcore in a practical system in order to achieve unattended operation. We defined the minimal functionality required from this hardcore and the required interactions between the

replicated managers and this hardcore.

This dissertation also presents key implementation details of the Ghidrah CMM system, including important system engineering problems we have solved, such as the handling of replicated time-triggered events with respect to replica consistency, and the efficient heartbeat mechanism used to detect node failures. We also presented preliminary results from experiments we conducted to evaluate the performance of the Ghidrah CMM system and to validate its fault tolerance mechanisms with fault injection campaigns.

The Ghidrah CMM system provides a practical platform for future research on high-performance, fault-tolerant cluster computing. Currently, the FTCT project is heading to the following two directions:

- providing sophisticated supports to application-level fault tolerance that allows parallel applications run on the cluster with high performance and high reliability.
- development of a distributed Byzantine fault-tolerant storage system for the system to meet I/O requirements for computations as well as provide stable storage for application checkpoints.

Bibliography

[Amir93] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella, ‘‘Fast Message Ordering and Membership Using a Logical Token-Passing Ring,’’ *the 13th IEEE International Conference on Distributed Computing Systems*, pp.551-556 (May 1993).

[Amir95] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella, ‘‘The Totem Single-Ring Ordering and Membership Protocol,’’ *ACM Transactions on Computer Systems*, vol.13, no.4, pp.311-342 (November 1995).

[Ande95] T. E. Anderson, D. E. Culler, and D. A. Patterson, ‘‘A Case for NOW (Networks of Workstations),’’ *IEEE Micro*, vol.15, no.1, pp.54-64 (February 1995).

[Arak03] T. Araki, ‘‘(t, k)-Diagnosable System: a Generalization of the PMC Models,’’ *IEEE Transactions on Computers*, vol.52, pp.971-975 (July 2003).

[Bake96] M. A. Baker, G. C. Fox, and H. W. Yau, ‘‘*A Review of Commercial and Research Cluster Management Software,*’’ Technical Report, Northeast Parallel Architectures Center, Syracuse University (June 1996).

[Bara98] A. Barak and O. La’adan, ‘‘The MOSIX Multicomputer Operating

System for High Performance Cluster Computing,'' *Journal of Future Generation Computer Systems*, vol.13, no.4-5, pp.361-372 (March 1998).

[Barb93] M. Barborak, A. Dahbura, and M. Malek, ''The Consensus Problem in Fault-Tolerant Computing,'' *ACM Computing Surveys*, vol.25, no.2, pp.171-220 (June 1993).

[Bars76] F. Barsi, F. Grandoni, and P. Maestrini, ''A Theory of Diagnosability of Digital Systems,'' *IEEE Transactions on Computers*, vol.25, no.6, pp.585-593 (June 1976).

[Beni00] L. Benini, A. Bogliolo, and G. De Micheli, ''A Survey of Design Techniques for System_level Dynamic Power Management,'' *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* vol.8, no.3, (June 2000).

[Bian90] R. P. Bianchini, K. Goodwin, and D. Nydick, ''Practical Application and Implementation of Distributed System-Level Diagnosis Theory,'' *the 20th International Symposium on Fault-Tolerant Computing Systems*, pp.332-339 (June 1990).

[Birm87] K. P. Birman and T. Joseph, ''Reliable Communication in the Presence of Failures,'' *ACM Transactions on Computer Systems*, vol.5, no.1, pp.47-76 (Feburary 1987).

[Birm91] K. P. Birman, A. Schiper, and P. Stephenson, ''Lightweight Causal and Atomic Group Multicast,'' *ACM Transactions on Computer*

Systems, vol.9, no.3, pp.272-314 (August 1991).

[Birm93] K. P. Birman, ‘‘The Process Group Approach to Reliable Distributed Computing,’’ *Communications of the ACM*, vol.36, no.12, pp.36-53 (December 1993).

[Blou99] D. M. Blough and H. W. Brown, ‘‘The Broadcast Comparison Model for On-Line Fault Diagnosis in Multicomputer Systems: Theory and Implementation,’’ *IEEE Transactions on Computers*, vol.48, no.5, pp.470-493 (May 1999).

[Blum01] T. Blum and C. Paar, ‘‘High-Radix Montgomery Modular Exponentiation on Reconfigurable Hardware,’’ *IEEE Transactions on Computers*, vol.50, no.7, pp.759-764 (July 2001).

[Bode95] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su, ‘‘Myrinet: A Gigabit-per-Second Local Area Network,’’ *IEEE Micro*, vol.15, pp.29-36 (February 1995).

[Boho01] V. Bohossian, C. C. Fan, P. S. LeMahieu, M. D. Riedel, L. Xu, and J. Bruck, ‘‘Computing in the RAIN: A Reliable Array of Independent Nodes,’’ *IEEE Transactions on Parallel and Distributed Systems*, vol.12, no.2, pp.99-113 (February 2001).

[Brac85] G. Bracha and S. Toueg, ‘‘Asynchronous Consensus and Broadcast Protocols,’’ *Journal of the ACM*, vol.32, no.4, pp.824-840 (October 1985).

[Budh93] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, ‘‘The Primary-Backup Approach,’’ in *Distributed systems (2nd Ed.)* , Sape Mullender ed., ACM Press, pp.199-216 (1993).

[Busk93] R. W. Buskens and R. P. Bianchini, ‘‘Distributed On-Line Diagnosis in the Presence of Arbitrary Faults,’’ *the 23rd International Symposium on Fault-Tolerant Computing Systems*, pp.470-479 (June 1993).

[Cast82] X. Castillo, S. R. McConnel, and D. P. Siewiorek, ‘‘Derivation and Calibration of a Transient Error Reliability Model,’’ *IEEE Transactions on Computers*, vol.C-31, no.7, pp.658-671 (July 1982).

[Cast99a] M. Castro and B. Liskov, ‘‘Practical Byzantine Fault Tolerance,’’ *the Third Symposium on Operating Systems Design and Implementation*, pp.173-186 (February 1999).

[Cast99b] M. Castro and B. Liskov, ‘‘*Authenticated Byzantine Fault Tolerance Without Public-Key Cryptography,*’’ Technical Memo MIT/LCS/TM-589, MIT Laboratory for Computer Science (1999).

[Cast00] M. Castro and B. Liskov, ‘‘Proactive Recovery in a Byzantine-Fault-Tolerant System,’’ *the Fourth Symposium on Operating Systems Design and Implementation*, pp.273-287 (October 2000).

[Chan96a] T. D. Chandra and S. Toueg, ‘‘Unreliable Failure Detectors for Reliable Distributed Systems,’’ *Journal of the ACM*, vol.43, no.2,

pp.225-267 (March 1996).

[Chan96b] T. D. Chandra, V. Hadzilacos, and S. Toueg, ‘‘The Weakest Failure Detector for Solving Consensus,’’ *Journal of the ACM*, vol.43, no.4, pp.685-722 (July 1996).

[Chap99] S. J. Chapin, D. Katramatos, J. Karpovich, and A. S. Grimshaw, ‘‘Resource Management in Legion,’’ *Future Generation Computer Systems*, vol.15, no.5-6, pp.583-594 (October 1999).

[Ch´er92] M. Ch´erèque, D. Powell, P. Reynier, J.-L. Richier, and J. Voiron, ‘‘Active Replication in Delta-4,’’ *the 22nd International Symposium on Fault-Tolerant Computing Systems*, pp.28-37 (July 1992).

[Cher98] A. Cherif and T. Katayama, ‘‘Replica Management for Fault-Tolerant Systems,’’ *IEEE Micro*, vol.18, no.5, pp.64-65 (September-October 1998).

[Choc01] G. V. Chockler, I. Keidar, and R. Vitenberg, ‘‘Group Communication Specifications: A Comprehensive Study,’’ *ACM Computing Surveys*, vol.33, no.4, pp.427-469 (December 2001).

[Chwa81] K. Y. Chwa and S. L. Hakimi, ‘‘Schemes for Fault-Tolerant Computing: A Comparison of Modularly Redundant and t-Diagnosable Systems,’’ *Information and Control*, vol.49, pp.212-238 (1981).

[Cray98] Cray Reserach, Inc., ‘‘*Introducing NQE,*’’ Publication IN-2153 3.3 (March 1998).

[Cris99] F. Cristian and C. Fetzer, ‘‘The Timed Asynchronous Distributed System Model,’’ *IEEE Transactions on Parallel and Distributed Systems*, vol.10, no.6, pp.642-657 (June 1999).

[Czaj98] K. Czajkowski, I. T. Foster, N. T. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke, ‘‘A Resource Management Architecture for Metacomputing Systems,’’ *the Fourth Workshop on Job Scheduling Strategies for Parallel Processing*, pp.62-82 (March 1998).

[D´efa00] X. D´efago, ‘‘*Agreement-Related Problems: From Semi-Passive Replication to Totally Ordered Broadcast,*’’ PhD thesis, Number 2229, ´Ecole Polytechnique F´ed´erale de Lausanne, Switzerland (August 2000).

[D´efa04] X. D´efago, A. Schiper, and P. Urb´an, ‘‘Total order broadcast and multicast algorithms: Taxonomy and survey,’’ *ACM Computing Surveys*, vol.36, no.4, pp.372-421 (December 2004).

[Dole87] D. Dolev, C. Dwork, and L. Stockmeyer, ‘‘On the Minimal Synchronism Needed for Distributed Consensus,’’ *Journal of the ACM*, vol.34, no.1, pp.77-97 (January 1987).

[Doud97] A. Doudou and A. Schiper, ‘‘*Muteness Detectors for Consensus with Byzantine Processes,*’’ Technical Report 97-230, ´Ecole Polytechnique F´ed´erale de Lausanne, Switzerland (October 1997).

[Dris03] K. Driscoll, B. Hall, H. Sivencrona, and P. Zumsteg, ‘‘Byzantine

Fault Tolerance, from Theory to Reality,'' *the 22nd International Conference on Computer Safety, Reliability and Security (SAFECOMP 2003)*, pp.235-248 (September 2003).

[Dwor88] C. Dwork, N. A. Lynch, and L. Stockmeyer, ''Consensus in the Presence of Partial Synchrony,'' *Journal of the ACM*, vol.35, no.2, pp.288-323 (April 1988).

[Feit90] D. G. Feitelson and L. Rudolph, ''Distributed Hierarchical Control for Parallel Processing,'' *IEEE Computer*, vol.23, no.5, pp.65-77 (May 1990).

[Fetz95] C. Fetzer and F. Cristian, ''On the Possibility of Consensus in Asynchronous Systems,'' *1995 Pacific Rim International Symposium on Fault-Tolerant Systems*, pp.86-91 (December 1995).

[Fisc85] M. J. Fischer, N. A. Lynch, and M. S. Paterson, ''Impossibility of Distributed Consensus with One Faulty Process,'' *Journal of the ACM*, vol.32, no.2, pp.374-382 (April 1985).

[Frac02a] E. Frachtenberg, J. Fernandez, F. Petrini, and S. Pakin, ''STORM: Lightning-Fast Resource Management,'' *2002 ACM/IEEE conference on Supercomputing*, pp.1-26 (November 2002).

[Frac02b] E. Frachtenberg, F. Petrini, J. Fernandez, and S. Coll, ''Scalable Resource Management in High Performance Computers,'' *IEEE International Conference on Cluster Computing (Cluster 2002)*,

pp.305-314 (September 2002).

[Garc82] H. Garcia-Molina, ‘‘Elections in a Distributed Computing System,’’ *IEEE Transactions on Computers*, vol.31, no.1, pp.48-59 (January 1982).

[Codine] GENIAS Software GmbH, *CODINE: Computing in Distributed Networked Environments*. Germany (1995).

[Gent01] W. Gentzsch, ‘‘Sun Grid Engine: Towards Creating a Compute Power Grid,’’ *the First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pp.35-36 (May 2001).

[Ghor98] D. P. Ghormley, D. Petrou, S. H. Rodrigues, A. M. Vahdat, and T. E. Anderson, ‘‘GLUnix: A Global Layer Unix for a Network of Workstations,’’ *Software - Practice and Experience*, vol.28, no.9, pp.929-961 (July 1998).

[Gosc90] A. Goscinski and M. Bearman, ‘‘Resource Management in Large Distributed Systems,’’ *ACM Operating Systems Review*, vol.24, no.4, pp.7-25 (October 1990).

[Graf93] T. P. Graf, R. G. Assini, J. M. Lewis, E. J. Sharpe, J. J. Turner, and M. C. Ward, ‘‘HP Task Broker: A Tool for Distributing Computational Tasks,’’ *Hewlett-Packard Journal*, vol.44, no.4, pp.15-22 (August 1993).

[Gray86] J. N. Gray, ‘‘Why Do Computers Stop and What Can Be Done About It,’’ *Proc. 5th Symp. on Reliability in Distributed Software*

and Database Systems, pp.3-12 (January 1986).

[Grop96] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, ‘‘A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard,’’ *Parallel Computing*, vol.22, no.6, pp.789-828 (September 1996).

[Guer97] R. Guerraoui and A. Schiper, ‘‘Software-based Replication for Fault Tolerance,’’ *IEEE Computer*, vol.30, no.4 , pp.68-74 (April 1997).

[Hadz93] V. Hadzilacos and S. Toueg, ‘‘Fault-Tolerant Broadcasts and Related Problems,’’ in *Distributed Systems*, *2nd Edition*, Sape Mullender ed., ACM Press New York, pp.97-146 (1993).

[Herl90] M. P. Herlihy and J. M. Wing, ‘‘Linearizability: A Correctness Condition for Concurrent Objects,’’ *ACM Transactions on Programming Languages and Systems*, vol.12, no.3, pp.463-492 (July 1990).

[Hilt95] M. Hiltunen, ‘‘Membership and System Diagnosis,’’ *the 14th Symposium on Reliable Distributed Systems*, pp.208-217 (1995).

[Hoss84] S. Hosseini, J. Kuhl, and S. Reddy, ‘‘A Diagnosis Algorithm for Distributed Computing Systems with Dynamic Failure and Repair,’’ *IEEE Transactions on Computers*, vol.33, no.3, pp.223-233 (March 1984).

[Hsue97] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, ‘‘Fault Injection Techniques and Tools,’’ *IEEE Computer*, vol.30, no.4, pp.75-82

(April 1997).

[Hwan99] K. Hwang, H. Jin, E. Chow, C.-L. Wang, and Z. Xu, ‘‘Designing SSI Clusters with Hierarchical Checkpointing and Single I/O Space,’’ *IEEE Concurrency*, vol.7, no.1, pp.60-69 (January-March 1999).

[Kaas89] M. F. Kaashoek, A. S. Tanenbaum, S. F. Hummel, and H. E. Bal, ‘‘An Efficient Reliable Broadcast protocol,’’ *ACM Operating Systems Review*, vol.23, no.4, pp.5-19 (October 1989).

[Kaas91] M. F. Kaashoek and A. S. Tanenbaum, ‘‘Group Communication in the Amoeba Distributed Operating System,’’ *the 11th Interational Conference on Distributed Computing Systems*, pp.222-230 (May 1991).

[Kaas96] M. F. Kaashoek and A. S. Tanenbaum, ‘‘An Evaluation of the Amoeba Group Communication System,’’ *the 16th International Conference on Distributed Computing Systems*, pp.436-448 (May 1996).

[Kalb99] Z. Kalbarczyk, R. K.Iyer, S. Bagchi, and K. Whisnant, ‘‘Chameleon: A Software Infrastructure for Adaptive Fault Tolerance,’’ *IEEE Transactions on Parallel and Distributed Systems*, vol.10, no.6, pp.560-579 (June 1999).

[Kapl94] J. A. Kaplan and M. L. Nelson, ‘‘*A Comparison of Queueing, Cluster, and Distributed Computing Systems*,’’ NASA Technical

Memorandum 109025, NASA LaRC (June 1994).

[Katz03] D. S. Katz and R. R. Some, ‘‘NASA Advances Robotic Space Exploration,’’ *IEEE Computer*, vol.36, no.1, pp.52-61 (January 2003).

[Kihl97] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, ‘‘Solving Consensus in a Byzantine Environment Using an Unreliable Fault Detector,’’ *International Conference on Principles of Distributed Systems*, pp.61-75 (December 1997).

[Kihl98] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, ‘‘The SecureRing Protocols for Securing Group Communication,’’ *IEEE 31st Hawaii International Conference on System Sciences*, vol.3, pp.317-326 (January 1998).

[Kihl03] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, ‘‘Byzantine Fault Detectors for Solving Consensus,’’ *The Computer Journal*, vol.46, no.1, pp.16-35 (January 2003).

[Kuhl81] J. G. Kuhl and S. Reddy, ‘‘Fault-Diagnosis in Fully Distributed Systems,’’ *the 11th International Symposium on Fault Tolerant Computing*, pp.100-105 (June 1981).

[Lamp82a] L. Lamport, R. Shostak, and M. Pease, ‘‘Byzantine Generals Problem,’’ *ACM Transactions on Programming Languages and Systems*, vol.4, no.3, pp.382-401 (July 1982).

[Lamp82b] L. Lamport, ‘‘Time, Clocks, and the Ordering of Events in a

Distributed System,'' *Communications of the ACM*, vol.21, no.7, pp.558-565 (July 1982).

[Lamp98] L. Lamport, ''The Part-Time Parliament,'' *ACM Transactions on Computer Systems*, vol.16, no.2, pp.133-169 (May 1998).

[Lamp01a] L. Lamport, ''Paxos Made Simple,'' *ACM SIGACT News (Distributed Computing Column)*, vol.32, no.4, pp.18-25 (December 2001).

[Lamp05] L. Lamport, ''*Generalized Consensus and Paxos*,'' Technical Report, MSR-TR-2005-33, Microsoft Research (March 2005).

[Lamp01b] B. Lampson, ''*The ABCD's of Paxos*,'' Presented at the 20th annual ACM symposium on Principles of Distributed Computing (August 2001).

[Lamp96] B. W. Lampson, ''How to Build a Highly Available System Using Consensus,'' *the 10th International Workshopt*, pp.1-17 (October 1996).

[Li01] M. Li, D. Goldberg, W. Tao, and Y. Tamir, ''Fault-Tolerant Cluster Management for Reliable High-Performance Computing,'' *International Conference on Parallel and Distributed Computing and Systems*, pp.480-485 (August 2001).

[Li02] M. Li, W. Tao, D. Goldberg, I. Hsu, and Y. Tamir, ''Design and Validation of Portable Communication Infrastructure for Fault-Tolerant Cluster Middleware,'' *IEEE International Conference on*

Cluster Computing (Cluster 2002), pp.266-274 (September 2002).

[Li04] M. Li and Y. Tamir, ‘‘Practical Byzantine Fault Tolerance Using Fewer than 3f+1 Active Replicas,’’ *the 17th International Conference on Parallel and Distributed Computing Systems*, pp.241-247 (September 2004).

[Lisk91] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams, ‘‘Replication in the Harp File System,’’ *the 13th ACM Symposium on Operating Systems Principles*, pp.226-238 (October 1991).

[Litz88] M. J. Litzkow, M. Livny, and M. W. Mutka, ‘‘Condor - A Hunter of Idle Workstations,’’ *the Eighth International Conference on Distributed Computing Systems*, pp.104-111 (June 1988).

[Livn82] M. Livny and M. Melman, ‘‘Load Balancing in Homogeneous Broadcast Distributed Systems,’’ *ACM Computer Network Performance Symposium*, pp.47-55 (April 1982).

[Made02] H. Madeira, R. R. Some, F. Moreira, D. Costa, and D. Rennels, ‘‘Experimental Evaluation of a COTS System for Space Applications,’’ *International Conference on Dependable Systems and Networks (DSN’02)*, pp.325-330 (June 2002).

[Maen81] J. Maeng and M. Malek, ‘‘A Comparison Connection Assignment for Self-Diagnosis of Multiprocessor Systems,’’ *the 11th International Symposium on Fault Tolerant Computing*, pp.173-175

(June 1981).

[Male80] M. Malek, ‘‘A Comparison Connection Assignment for Diagnosis of Multiprocessor Systems,’’ *the Seventh Internationall symposium on Computer Architecture*, pp.31-36 (May 1980).

[Minn01] R. Minnich, ‘‘Bipolar Disorder in Cluster Networking,’’ *IEEE International Conference on Cluster Computing (Cluster 2001)* (October 2001). http://www.cacr.caltech.edu/cluster2001/program/talks/minnich.pdf.

[Mose94] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala, ‘‘Processor Membership in Asynchronous Distributed Systems,’’ *IEEE Transactions on Parallel and Distributed Systems*, vol.5, no.5, pp.459-473 (May 1994).

[Mugl03] J. Mugler, T. Naughton, S. L. Scott, B. Barrett, A. Lumsdaine, J. M. Squyres, B. des Ligneris, F. Giraldeau, and C. Leangsuksun, ‘‘OSCAR Clusters,’’ *Ottawa Linux Symposium (OLS’03)*, pp.409-419 (July 2003).

[Mull90] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren, ‘‘Amoeba: A Distributed Operating System for the 1990s,’’ *IEEE Computer*, vol.23, no.5, pp.44-53 (May 1990).

[NIST94] NIST, *Digital Signature Standard*,Federal Information Processing Standards PUB-186 (May 1994).

[Oust82] J. K. Ousterhout, ‘‘Scheduling Techniques for Concurrent Systems,’’ *the Third International Conference on Distributed Computing Systems*, pp.22-30 (October 1982).

[Peas80] M. Pease, R. Shostak, and L. Lamport, ‘‘Reaching Agreement in the Presence of Faults,’’ *Journal of the ACM*, vol.27, no.2, pp.228-234 (April 1980).

[Powe88] D. Powell, P. Ve´rissimo, G. Bonn, F. Waeselynck, and D. Seaton, ‘‘The Delta-4 Approach to Dependability in Open Distributed Computing Systems,’’ *the 18th International Symposium on Fault-Tolerant Computing*, pp.246-251 (June 1988).

[Prep67] F. P. Preparata, G. Metze, and R. T. Chien, ‘‘On the Connection Assignment Problem of Diagnosable Systems,’’ *IEEE Transactions on Electronic Computers*, vol.16, pp.848-854 (December 1967).

[Pris00] R. D. Prisco, B. Lampson, and N. A. Lynch, ‘‘Revisiting the PAXOS Algorithm,’’ *Theoretical Computer Science*, vol.243, no.1-2, pp.35-91 (July 2000).

[Rama98] R. Raman, M. Livny, and M. Solomon, ‘‘Matchmaking: Distributed Resource Management for High Throughput Computing,’’ *the Seventh IEEE International Symposium on High Performance Distributed Computing*, pp.140-146 (July 1998).

[Reit94] M. K. Reiter, ‘‘Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart,’’ *the 2nd ACM Conference on*

Computer and Communications Security, pp.68-80 (November 1994).

[Reit95] M. K. Reiter, ‘‘The Rampart Toolkit for Building High-Integrity Services,’’ *Theory and Practice in Distributed Systems (Lecture Notes in Computer Science 938)*, pp.99-110 , Springer-Verlag, Berlin Germany (1995).

[Reit96] M. K. Reiter, ‘‘A Secure Group Membership Protocol,’’ *IEEE Transactions on Software Engineering*, vol.22, no.1, pp.31-42 (January 1996).

[Rive78] R. L. Rivest, A. Shamir, and L. M. Adleman, ‘‘A Method for Obtaining Digital Signatures and Public-key Cryptosystems,’’ *Communications of the ACM*, vol.21, no.2, pp.120-126 (Feburary 1978).

[Rive92] R. L. Rivest, ‘‘The MD5 Message-Digest Algorithm,’’ *Internet RFC-1321* (April 1992).

[Robe00] A. L. Robertson, ‘‘Linux-HA Heartbeat Design,’’ *the Fourth International Linux Showcase and Conference* (October 2000).

[Robe04] A. L. Robertson, ‘‘The Evolution of the Linux-HA Project,’’ *UKUUG LISA/Winter Conference High-Availability and Reliability* (February 2004).

[Rush] J. M. Rushby, ‘‘Reconfiguration and Transient Recovery in State Machine Architectures,’’ *the 26th International Symposium on*

Fault-Tolerant Computing, pp.6-15 (June 1996).

[Russ98] S. H. Russ, J. Robinson, B. K. Flachss, and B. Heckels, ‘‘The Hector Distributed Run-Time Environments,’’ *IEEE Transactions on Parallel and Distributed Systems*, vol.9, no.11, pp.1104-1112 (November 1998).

[Russ99] S. H. Russ, K. Reece, J. Robinson, B. Meyers, R. Rajan, L. Rajagopalan, and C.-H. Tan, ‘‘Hector: An Agent-Based Architecture for Dynamic Resource Management,’’ *IEEE Concurrency*, vol.7, no.2, pp.47-55 (April-June 1999).

[Schn84] F. B. Schneider, ‘‘Byzantine Generals in Action: Implementing Fail-Stop Processors,’’ *ACM Transactions on Computer Systems*, vol.2, no.2, pp.145-154 (May 1984).

[Schn90] F. B. Schneider, ‘‘Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial,’’ *ACM Computing Surveys*, vol.22, no.4, pp.299-319 (December 1990).

[Seng92] A. Sengupta and A. T. Dahbura, ‘‘On Self-Diagnosable Multiprocessor Systems: Diagnosis by the Comparison Approach,’’ *IEEE Transactions on Computers*, vol.41, no.11, pp.1386-1396 (November 1992).

[Shin87] K. G. Shin and P. Ramanathan, ‘‘Diagnosis of Processors with Byzantine Faults in a Distributed Computing System,’’ *the 17th International Symposium on Fault-Tolerant Computing Systems*,

pp.55-60 (July 1987).

[Some99] R. R. Some and D. C. Ngo, ‘‘REE: A COTS-Based Fault Tolerant Parallel Processing Supercomputer for Spacecraft Onboard Scientific Data Analysis,’’ *IEEE Digital Avionics Systems Conference*, pp.7.B.3-1 - 7.B.3-12 (1999).

[Stee94] P. Steenkiste, ‘‘A Systematic Approach to Host Interface Design for High-Speed Networks,’’ *IEEE Computer*, vol.27, no.3, pp.47-57 (March 1994).

[Stot00] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczyk, and R. K. Iyer,, ‘‘NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors,’’ *the Fourth IEEE International Computer Performance and Dependability Symposium (IPDS-2K)*, pp.91-100 (March 2000).

[SunCA3] Sun Microsystems, Inc., ‘‘*Sun Cluster 3 Architecture: A Technical Overview,*’’ White Paper (March 2001).

[Ture92] J. Turek and D. Shasha, ‘‘The Many Faces of Consensus in Distributed Systems,’’ *IEEE Computer*, vol.25, no.6, pp.8-17 (June 1992).

[von98] T. von Eicken and W. Vogels, ‘‘Evolution of the Virtual Interface Architecture,’’ *IEEE Computer*, vol.31, no.11, pp.61-68 (November 1998).

[Walt94] C. J. Walter, N. Suri, and M. Hugue, ‘‘Continual On-Line Diagnosis

of Hybrid Faults,'' *the Fourth IFIP Working Conference on Dependable Computing for Critical Applications,* pp.233-249 (1994).

[Whis02] K. Whisnant, R. Iyer, P. Jones, R. Some, and D. Rennels, ''An Experimental Evaluation of the REE-SIFT Environment for Spaceborne Applications,'' *International Conference on Dependable Systems and Networks (DSN'02),* pp.585-594 (June 2002).

[Yang88] C.-L. Yang and G. M. Masson, ''A Distributed Algorithm for Fault Diagnosis in Systems with Soft Failures,'' *IEEE Transaction on Computers*, vol.37, no.11, pp.1476-1480 (November 1988).

[Yin03] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, ''Separating Agreement from Execution for Byzantine Fault Tolerant Services,'' *the 19th ACM Symposium on Operating Systems Principles,* pp.253-267 (October 2003).

[Zhou02] L. Zhou, F. B. Schneider, and R. van Renesse, ''COCA: A Secure Distributed On-line Certification Authority,'' *ACM Transactions on Computer Systems*, vol.20, no.4, pp.329-368 (November 2002).

[Zhou92] S. Zhou, ''LSF: Load Sharing in Large-Scale Heterogenous Distributed Systems,'' *Workshop on Cluster Computing* (December 1992).

[Zhou93] S. Zhou, X. Zheng, J. Wang, and P. Delisle, ''Utopia: A Load

Sharing Facility for Large, Heterogeneous Distributed Computer Systems,'' *Software - Practice and Experience*, vol.23, no.12 , pp.1305-1336 (December 1993).

[Zhu95] W. Zhu, C. F. Steketee, and B. Muilwijk, ''Load Balancing and Workstation Autonomy on Amoeba,'' *Australian Computer Science Communications*, vol.17, no.1, pp.588-597 (February 1995).