UCLA Computer Science Department Technical Report #140008
University of California, Los Angeles, March 2014

UNIVERSITY OF CALIFORNIA

Los Angeles

Design and Validation of a Layered Approach to Fault Tolerance for Distributed Applications

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

Israel Yi-Hsin Hsu

2014

© Copyright by

Israel Yi-Hsin Hsu

2014

ABSTRACT OF THE DISSERTATION

Design and Validation of a Layered Approach
to Fault Tolerance for Distributed Applications

by

Israel Yi-Hsin Hsu
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2014
Professor Yuval Tamir, Chair

Clusters of message-passing computing nodes provide high-performance platforms for distributed applications. Cost-effective implementations of such systems are based on commercial off-the-shelf (COTS) hardware and software components. One trend in the deployment of such systems is to scale up the number of compute nodes to deliver higher performance levels. The higher component count results in a corresponding higher rate of failure. Another trend is to deploy clusters for mission-critical applications or in harsh environments, where reliability requirements are higher than in a controlled lab setting. Both of these trends point to an increasing need to employ fault tolerance techniques to meet the reliability requirements of the applications being executed.

We present a layered approach to providing fault tolerance for message-passing applications on compute clusters that are based on COTS hardware components, COTS operating systems, and a COTS API for application programmers. This approach relies on highly-resilient cluster

management middleware (CMM) that ensures the survival of key system services despite the failure of cluster components. A key feature of this CMM is that it provides services that enable and simplify user-level implementation of fault tolerance for applications without dictating the specific techniques employed. In particular, while application-transparent techniques are supported, the CMM also supports application-specific techniques that are tailored and optimized for the characteristics and requirements of specific applications. To this end, we have developed an API that can be used in the implementation of fault tolerance by the application programmer as well as by developers of user-level libraries that provide application-transparent fault tolerance.

The effectiveness of our layered approach is demonstrated and evaluated with several applications employing different techniques for fault tolerance. The entire system is subjected to a fault injection campaign. We show that the CMM services that support fault tolerance techniques operate reliably and with very low overhead. We also show that application-specific fault tolerance techniques detect and recover from a vast majority of manifested faults while imposing much lower performance overhead than application-transparent schemes.

The dissertation of Israel Yi-Hsin Hsu is approved.

William Kaiser

Glenn Reinman

David Rennels

Yuval Tamir, Committee Chair

University of California, Los Angeles

2014

To my wife, Grace,

and my children, Anna and Daniel.

Table of Contents

Abstract ii

Acknowledgments xv

Vita and Publications xvii

Chapter One - Introduction 1

1.1. The Layered Architecture of Clusters 2

1.2. Designing Services to Support Fault-Tolerant Distributed Applications 3

1.3. Implementing Services in Cluster Management Middleware 4

1.4. Contributions 5

1.5. Roadmap of the Dissertation 7

Chapter Two - Related Work 8

2.1. Fault Tolerance Terminology 8

2.2. The Consensus Problem in Fault Tolerance 11

2.2.1. System-Level Diagnosis 11

2.2.2. The Byzantine Generals Problem 13

2.3. Implementing Fault-Tolerant Systems and Applications 14

2.3.1. Supporting Reliability Requirements of Multiple Applications 15

2.3.2. Transparency of Fault Tolerance Mechanisms to Applications 15

2.4. Implementation Using Modular Redundant Hardware 17

2.5. Implementation Using COTS Hardware and System-Level Software 19

2.6. Implementation Using COTS Hardware, COTS OS, Cluster Middleware and Application Software 22

2.6.1. Cluster Management Middleware 22

2.6.2. Checkpointing and Rollback Recovery for MPI Applications 25

2.6.3. Application-Specific Fault Tolerance Mechanisms 27
2.7. Summary 31
Chapter Three - Layered Fault Tolerance for Compute Clusters 32
3.1. End-to-End Arguments For Application-Specific Fault Tolerance 33
3.2. Involving the CMM in Implementation of Fault Tolerance Mechanisms for Applications 35
3.3. CMM Services for Supporting Application Fault Tolerance 36
3.3.1. Error Detection and Notification 37
3.3.2. Diagnosis 39
3.3.3. Error Recovery 40
3.3.4. Reconfiguration of Application Processes 40
3.4. A Set of CMM Services for Supporting Application Fault Tolerance 41
Chapter Four - Design and Implementation of the Layered Approach 42
4.1. Asynchronous Communication for Fault Tolerance Mechanisms 43
4.1.1. Ghidrah Signals 44
4.1.2. Group Requests 48
4.1.3. Simplifying the Handling of Ghidrah Signals 50
4.2. G-MPI: Extending MPI Semantics for Application Fault Tolerance 61
4.2.1. FT-MPI: Extending MPI Semantics for Recovery from Process Crashes 62
4.2.2. G-MPI: Extending MPI Semantics for Application Fault Tolerance 65
4.3. An Example G-MPI Fault-Tolerant Application 73
4.3.1. Detecting a Process Crash 74
4.3.2. Detecting a Process Hang 74
4.3.3. Recovering from a Process Hang or Crash 76
4.4. Ghidrah Cluster Management Middleware 82

4.4.1. Reliable Centralized Cluster Manager 82
4.4.2. An Agent on Each Node 84
4.4.3. Ghidrah Application Library 85
4.4.4. Radiation-Hardened Spacecraft Control Computer 86
4.4.5. Implementing Application-Directed Process Management 91
4.5. Group Communication in Ghidrah CMM 96
4.5.1. An Optimized Design for Group Communication 97
4.5.2. A Flaw in the Group Communication Protocol 110
Chapter Five - Evaluation of the Layered Approach 114
5.1. Implementing Application-Level Fault Tolerance for MPI Applications 115
5.1.1. LAMMPS 115
5.1.2. ClustalW-MPI 118
5.1.3. Nqueens and TSP 120
5.2. Implementing Application-Transparent Fault Tolerance 122
5.3. Experimental Setup 125
5.3.1. A Virtualized Distributed System 126
5.3.2. Fault Injection in a Virtualized Environment 127
5.3.3. Injection Campaign Management 130
5.3.4. Logging for Postmortem Analysis 132
5.4. Validating Ghidrah CMM 133
5.4.1. Design of Fault Injection Campaigns 134
5.4.2. Ghidrah Validation Results 138
5.5. Performance of Ghidrah Fault Tolerance Services 141
5.6. Performance Overhead of Fault Tolerance Mechanisms 145
5.7. Evaluation of Application-Specific Fault Tolerance Mechanisms 148
5.7.1. Process Crashes and Hangs 149

5.7.2. LAMMPS Application-Specific Error Detection and Recovery 151
5.7.3. ClustalW-MPI Application-Specific Error Detection and Recovery 155
5.7.4. Nqueens and TSP Application-Specific Error Detection and Recovery 157
5.8. Summary 159
Chapter Six - Conclusion 160
Bibliography 163

List of Figures

1.1 Typical layered architecture of cluster systems 3

4.1 API for Ghidrah signals 45

4.2 Group request and cluster manager's reply, with one faulty application process 49

4.3 Two events in one epoch: Cluster manager's broadcast signal causes application's group request to be dropped 54

4.4 Two broadcast requests in one epoch 55

4.5 Two events in one epoch: Error detected by cluster manager during execution of service 56

4.6 Handling two events in one epoch: Cluster manager broadcasts error notification during execution of service 57

4.7 Handling two events in one epoch: Cluster manager precedes broadcast of error notification with an acknowledgment of the application's group request 58

4.8 API for manipulating G-MPI state 66

4.9 API for G-MPI signals 66

4.10 API for G-MPI timer events 69

4.11 API for managing G-MPI signal delivery 69

4.12 Application's handler for G-MPI signal G_MPI_SIG_NTF_DIED 74

4.13 Application's code for timing a computation 75

4.14 Application's handler for G-MPI signal APP_SIG_HANG 75

4.15 Application's synchronous function for diagnosis, recovery and reconfiguration 77

4.16 Supporting functions invoked by `diagnose_recover_reconfigure()` 79

4.17 Application's handler for G-MPI signal G_MPI_SIG_REP_SPAWNED 79

4.18 Application's `main` function 81
4.19 Logical structure of the Ghidrah CMM 83
4.20 *FsmTask*, the cluster manager's finite state machine for a task 88
4.21 Continuation of *FsmTask*: states handling task requests 89
4.22 *FsmProc*, the cluster manager's finite state machine for a task process 92
4.23 *FsmAgentProc*, the *Agent's* finite state machine for a task process 95
4.24 Legend for pseudocode 99
4.25 State of a Replica 100
4.26 State of an Agent 101
4.27 Point-to-point communication procedures that are common to Agents and Replicas: sending and receiving messages 102
4.28 Point-to-point communication procedures that are common to Agents and Replicas: delivering messages, connecting, and disconnecting 104
4.29 Point-to-point communication procedures that are common to Agents and Replicas: handling acknowlnedgements and acknowledgment timeouts 105
4.30 Replica *i* reliably sends a group message replica to Agent *c* 105
4.31 Agent *c* identifies the next group message to deliver 106
4.32 Agent *c* delivers a group message, processes a group acknowledgment, and handles a group message timeout 108
4.33 Agent *c* reliably sends a group message to the Manager Group 108
4.34 Replica *i* processes a group message from Agent *c* or wrapper message from Replica *j* 109
4.35 Replica *i* delivers a group message from Agent *c* 111
4.38 A flaw exposed by a dropped group message replica 112
4.37 Agent sends a group acknowledgment only after receiving all three group message replicas 112
4.38 A flaw exposed by a dropped point-to-point message 113

5.1 Architecture of VMM-level injector 127
5.2 Architecture of OS-level injector 128
5.3 Run-times of the campaign runs that exposed flaws 140

List of Tables

4.1 Pre-defined Ghidrah signals 46

4.2 Pre-defined G-MPI signals 67

4.3 Pre-defined local G-MPI signals 70

4.4 Legend of state transitions in *FsmTask* 90

4.5 Legend of state transitions in *FsmProc* 91

4.6 Legend of state transitions in *FsmAgentProc* 96

5.1 Descriptions of CMM fault injection campaigns 134

5.2 Results of CMM fault injection campaigns 139

5.3 Response time of CMM for detecting and notifying applications of process crashes 142

5.4 Response time of CMM for handling application requests 145

5.5 Application execution times with and without error detection 146

5.6 Application execution times with and without checkpointing 147

5.7 Size of application-specific and application-transparent checkpoints 148

5.8 Possible outcomes due to CPU register fault injection 149

5.9 Time to diagnose and recover from a hung process 151

5.10 Fault injections and errors in LAMMPS without fault tolerance 152

5.11 Fault injections and errors in LAMMPS with fault tolerance 153

5.12 Ratios of incorrect results to correct results in LAMMPS undetected errors 153

5.13 Performance of LAMMPS application-specific fault tolerance mechanisms 153

5.14 Fault injections and errors in ClustalW-MPI 156

5.15 Performance of ClustalW-MPI application-specific fault tolerance mechanisms 156

5.16 Fault injections and errors in Nqueens and TSP 158
5.17 Performance of Nqueens and TSP application-specific fault tolerance mechanisms 158

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my adviser, Professor Yuval Tamir, for his guidance and mentorship through my Ph.D. experience. His experience in conducting quality research was invaluable to me. He taught me the importance and value of quality presentation and writing skills. I think I also learned a thing or two about procrastination, but only time will tell what grade I receive on that lesson. Professor Tamir was unfailing in his patience toward me, always encouraging me to press forward.

I also sincerely thank my other committee members, Professor David Rennels, Professor Glenn Reinman, and Professor William Kaiser, for being on my doctoral committee and for providing insightful comments and questions to pursue in my research.

There is a special comraderie that was built up within the four walls of the Concurrent Systems Lab that I will never forget. I enjoyed friendship with many of the graduate students who passed through, but I would like to especially thank Ming Li and Michael Le. We shared many hours of delight, despair, and debugging together... and also sweat when the facilities people assumed that our lab did not need air conditioning. Thanks for the incomparable experience!

I thank the UCLA Department of Computer Science and the staff for their support through my years here. I thoroughly enjoyed the opportunity to serve as a Teaching Assistant for the undergraduate computer systems architecture course; I learned that the best way to learn is to teach.

My Ph.D. journey would not have been possible without the faithful and limitless support

and love of my mother and my mother-in-law. I thank them both from the bottom of my heart.

Last, but not least, I thank the love of my life, my wife Grace, for her love, support, and endurance. She always stood behind me in my research pursuit, and sometimes pushed me forward, too. It has been a long journey for both of us, and I am indebted to her for her faithfulness.

VITA

1996	B.S., Computer Engineering Department of Computer Science and Engineering University of Washington Seattle, Washington
2002	M.S., Computer Science Computer Science Department University of California Los Angeles, California
2000-2007	Teaching Assistant/Graduate Student Researcher Computer Science Department University of California Los Angeles, California

PUBLICATIONS

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*, pp.266-274 (September 2002).

I. Hsu, A. Gallagher, M. Le, and Y. Tamir, ‘‘Using Virtualization to Validate Fault-Tolerant Distributed Systems,’’ *International Conference on Parallel and Distributed Computing and Systems*, pp.210-217 (November 2010).

M. Le, I. Hsu, and Y. Tamir, ‘‘Resilient Virtual Clusters,’’ *17th IEEE Pacific Rim International Symposium on Dependable Computing*, pp.214-223 (December 2011).

Chapter One

Introduction

Clusters of computers have been successfully deployed in a wide variety of environments. On one hand, increasingly large clusters for high-performance computing (HPC), with tens of thousands of processing nodes in a single system, are being used for executing long-running scientific workloads [Dong05]. As the number of components in a system increases, the probability that at a given time a component fails also increases [Schr06]. On the other hand, small clusters can be embedded in mission-critical applications or used in harsh environments such as on unmanned spacecraft for remote space exploration [Some99]. The rate of component failure is high in some embedded environments due to exposure to extreme temperatures, vibration, and high-energy particles. Both of these trends point to the increasing need of fault tolerance for applications running on clusters.

This dissertation presents and evaluates a layered architecture for software-implemented fault tolerance in clusters running distributed applications. The goal of the design is to provide a set of services to aid the implementation of fault-tolerant distributed applications in a way that minimizes the performance overhead and implementation complexity. Additionally, since the characteristics and requirements of different applications are best supported by different fault tolerance techniques, the services must support a wide variety of fault tolerance mechanisms.

Our layered approach is motivated by the fact that cluster management middleware (CMM) typically employs aggressive fault tolerance techniques due to its own reliability

requirements. There are two key reasons for this common design. First, CMM failure leads to the failure of the entire cluster. Second, only a small fraction of the cluster's resources (cycles, memory, etc) are typically spent executing CMM code, so aggressive fault tolerance techniques in the CMM have little impact on overall cluster resource utilization. The main theme of this dissertation is to leverage the high reliability of the CMM by providing services that facilitate the implementation of fault tolerance for applications.

Section 1.1 introduces the typical layered software architecture present in most compute clusters. Section 1.2 describes what kinds of services can aid the implementation of fault-tolerant distributed applications. Section 1.3 gives an overview of how these services are implemented and validated in our CMM. The contributions of this dissertation are given in Section 1.4, and a roadmap of the remainder of the dissertation is provided in Section 1.5.

1.1. The Layered Architecture of Clusters

Clusters typically host a layered software architecture as shown in Figure 1.1. At the lowest layers, a cluster is composed of commercial off-the-shelf (COTS) processing nodes connected by a fast communication network. Each node runs a conventional non-distributed operating system (OS). Software known as cluster management middleware (CMM) runs on each node on top of the OS. The CMM is responsible for managing the nodes and other system resources of the cluster. It provides services such as launching applications, spawning processes, scheduling, and responding to unexpected events (e.g., node failure).

Many clusters are configured with additional software layers that provide higher-level abstraction of the cluster's compute resources for simplifying the development of distributed applications. For example, the Message Passing Interface (MPI) [Mess95] is a *de facto*

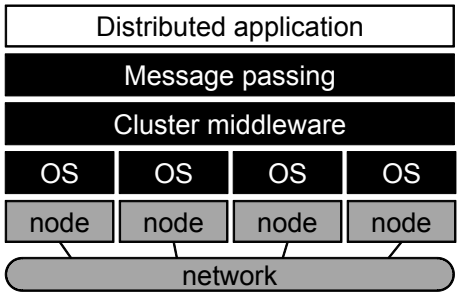


Figure 1.1: Typical layered architecture of cluster systems

standard that specifies an abstraction for message-passing processes along with a set of point-to-point and collective communication operations, freeing the application programmer from encumbering details such as remote execution, mapping of processes to nodes, and establishing message-passing connections among the processes of the distributed application. The highest layer in the cluster software architecture consists of application code.

1.2. Designing Services to Support Fault-Tolerant Distributed Applications

Our work focuses on the primitive operations that are common to many fault tolerance techniques used by distributed applications. For example, techniques to detect corruption in application-level data will vary from application to application, but typically an error detection in one process must be communicated to other processes so that they can coordinate for diagnosis and recovery. This communication is an example of a service that a reliable centralized entity can provide.

Some fault tolerance techniques require a distributed application to perform actions that must involve interaction with the CMM. For example, fault-free processes may determine that a process is faulty and that its outputs and messages are erroneous. The fault-free

processes can simply agree to ignore messages sent by the faulty process, but a more complete solution would be to terminate the faulty process. Since the CMM manages processes on behalf of applications, the application must request the CMM to terminate the faulty process.

Our work considers a variety of fault tolerance techniques for distributed applications, and derives a minimal set of services that can support the implementation of these techniques. A detailed explanation of this process is given in Chapter 3.

1.3. Implementing Services in Cluster Management Middleware

The end-to-end argument [Salt84] in system design makes a case for implementing fault tolerance in the highest layer, application code. The basic argument is that application-specific information is required in order to perform, or, at least, to perform efficiently, the key phases of fault tolerance mechanisms (e.g., error detection, error recovery). Hence, it may be best to implement fault tolerance entirely at the application level. For example, an application-transparent checkpointing and rollback error recovery scheme will involve periodic checkpointing of the entire application state. However, for a particular application, the size of the state that must be checkpointed to enable recovery may be significantly smaller than the size of the complete application state. Hence, for such an application, an application-specific checkpointing and rollback error recovery scheme will have a much lower performance overhead than an application-transparent scheme.

Despite the end-to-end argument, there are reasons to implement fault tolerance mechanisms, completely, or partially, in lower layers. First, implementing fault tolerance mechanisms in lower layers may be justified when such implementation yields better performance (lower performance overhead) than implementation at higher layers. A good

fault tolerance example of this is the use of error-correcting codes to enable retrieval of correct information despite errors. Second, there are some functions that cannot be implemented without the involvement of lower layers. For example, as part of error recovery, a distributed application may need to spawn a process on a cluster node that it was not previously using. This operation requires the involvement of the CMM.

We identify several criteria for designing CMM services to support application-level fault tolerance. First, the CMM services should be general in order to support a variety of fault tolerance techniques. Second, the CMM services should be simple. System complexity contributes to decreased reliability because the probability of system failure increases with the number of components or interactions within the system. Hence CMM services should add a minimal amount of complexity to the CMM. Third, the CMM services should be lightweight; that is, they should not require many resources or a lot of time to perform. The CMM must maintain high availability and be able to service application requests efficiently and promptly. Fourth, the CMM services should be designed in such a way that the CMM cannot fail due to malicious behavior of a failed application. Increased interaction between the CMM as a server and many applications as clients must not cause the CMM to be less reliable.

1.4. Contributions

In this research, we develop and validate a layered software architecture for implementing fault tolerance for distributed applications. This work makes the following contributions:

- A layered infrastructure that efficiently supports implementation of a range of distributed fault tolerance techniques.

- The design of a set of lightweight CMM services for supporting distributed application-level fault tolerance.
- Implementation of multiple fault tolerance techniques in MPI applications that take advantage of a trusted, reliable central authority.
- Validation of the layered approach through general and targeted fault injection campaigns.
- Comparison of performance overhead of application-specific fault tolerance schemes to that of application-transparent ones.

Performance measurements of the cluster while running MPI applications show that the CMM services designed to support application fault tolerance impose very low overhead. The CMM services to broadcast error notifications and synchronize processes of an application take 10 ms or less. For terminating and spawning processes at the request of the application, the CMM adds less than 2 ms delay on top of the time that the local node's OS uses to terminate or spawn the process.

Another key result of this research is the implementation and measurement of multiple fault tolerance techniques used to improve the reliability of existing MPI applications. We modified four MPI applications to use fault tolerance techniques to detect and recover from process hangs and crashes and also corruption in application-level state. Through fault injection campaigns, we observed that the modified MPI applications correctly detected and recovered from all process hangs and crashes, and detected and recovered from over 90% of cases of data corruption, while incurring less than 30% overhead on execution time.

1.5. Roadmap of the Dissertation

The rest of the dissertation is organized as follows.

Chapter 2 presents previous research as it relates to the layered approach. It reviews a representative set of distributed systems with respect to what fault tolerance mechanisms are implemented at what layers of the system. Chapter 3 presents the design of the layered approach, explaining the justification for the implementation of the set of supporting services at layers below the application code. Chapter 4 describes the implementation of the layered approach, covering all software layers including the CMM, application libraries, and MPI application code. Several MPI applications have been modified to use application-specific fault tolerance techniques. These modifications are presented in Chapter 5. The entire system is validated and tested with these MPI applications under fault injection campaigns, and these results are also presented in Chapter 5. Finally, Chapter 6 concludes the dissertation with a summary of the work and offers directions for future work.

Chapter Two

Related Work

The key ingredient for implementing fault-tolerant systems is redundancy. Redundant data and redundant execution can be automatically implemented by modular redundant hardware, by system-level software, or by application-level code. This chapter reviews distributed systems that implement fault tolerance for applications in all of these three layers.

The chapter begins in Section 2.1 with a brief definition of terminology used in the remainder of the dissertation. Section 2.2 reviews theoretical results on implementing fault tolerance for distributed systems. Section 2.3 discusses two factors that influence the layer at which fault tolerance mechanisms are implemented, whether involving hardware only, a combination of hardware and system-level software, or a combination of hardware, system-level software, and application software. Sections 2.4, 2.5 and 2.6 review a representative selection of fault-tolerant systems according to these factors, providing the context for the design that is presented in the remainder of the dissertation. Finally, Section 2.7 summarizes the work reviewed in this chapter.

2.1. Fault Tolerance Terminology

Research in fault-tolerant systems spans many years and many projects. The terminology used to describe faults and fault-tolerant systems has developed accordingly over the years. For example, an early proposal of terminology is given in [Ande82] while a fuller taxonomy of faults and properties of fault-tolerant systems is described in [Aviz04]. This

section defines the terminology used in the remainder of the dissertation.

Herein we adopt many of the concepts defined in [Ande82]. A *system* is a machine that is defined by its *specification*, a complete description of how the system interacts with its environment. The environment is composed of one or more other systems. A system interacts with its environment via an interface, consuming inputs and producing outputs according to its specification. The specification is composed of a finite set of *external states* and a state transition function. At the moment the external state of a system deviates from its specification, the system is said to have exhibited a *failure*.

Internally, a system is composed of a set of one or more connected components. Components themselves may be viewed as systems. The system's *design* specifies how its components interact. The internal behavior of a system is described by a tuple of its components' external states and a state transition function. The system's design also defines a function that maps internal states to external states.

In the context of the present work, the systems under consideration are distributed applications. The components of the distributed application are its application processes connected by a message-passing facility. An application's environment includes input sources, outputs, and interactions with its run-time environment such as a cluster manager.

A system failure is a symptom of a problem in the interaction among the system's components or within some of the components themselves. When a component fails, its external state deviates from its specification. This deviation is an *error* in the system. Errors are caused by *faults*, which are deviations that are internal to a system's components. A system that is *fault-tolerant* is able to detect errors in its internal state and prevent the error

from causing a system failure. In other words, the system is able to transform its internal state such that its external state does not deviate from its specification. Components and algorithms that are implemented in the system to detect and recover from errors are called *fault tolerance mechanisms*.

The implementation of fault tolerance mechanisms depends upon the class of faults to be tolerated. Faults are classified by the resulting behavior of the faulty component [Barb93]. In the most general class of faults, that of the Byzantine fault [Lamp82], the faulty component exhibits arbitrary behavior, including the ability to misrepresent its own state and manipulate its interactions to masquerade as a different component. Authenticated Byzantine fault [Lamp82] is slightly more restricted: the faulty component behaves arbitrarily, except that it cannot masquerade as another component. A hang fault occurs when the faulty component stops making progress and ceases generating output or state changes visible to other components. Finally, the most restrictive class is the fail-stop fault [Schl83], also known as a crash, in which the faulty component stops making progress and other components are immediately made aware of the fault.

Faults can also be classified by the duration of their effects. This classification depends on an assumption that a fault-free component can reliably test another component to determine whether it is faulty. Note that this classification does not describe the errors committed by the faulty component. A component that is permanently faulty does not pass any test by a fault-free component, now or in the future. A component that has an intermittent fault passes tests at some times and fails at other times. A component that has a transient fault fails tests for a short time and then passes all subsequent tests (unless another separate fault afflicts the component).

2.2. The Consensus Problem in Fault Tolerance

A key ingredient of fault tolerance mechanisms for distributed applications is achieving agreement among distributed fault-free components. For example, in order to detect a single faulty component, a majority of fault-free components must agree that they are not faulty and that another component is faulty. The Consensus problem may be defined as follows: Each component, or process, i proposes an initial value v_i. A process may irrevocably *decide* on a value that is one of the initial values. If all fault-free processes execute *Consensus* algorithm, then the following three properties are satisfied for fault-free processes: Each process must eventually *decide* on one of the initial values (validity), all processes that *decide* decide on the same value (agreement), and eventually all processes *decide* (termination) [Barb93].

Algorithms to solve Consensus have been studied for decades in a large body of work. This body of work may be divided into two orthogonal approaches that spring from two different ways of modeling systems. These areas of research are known as system-level diagnosis and the Byzantine Generals problem. Research on these two areas is surveyed in [Barb93].

2.2.1. System-Level Diagnosis

The goal of system-level diagnosis is to produce a diagnosis of which components in a system are faulty. In the seminal work [Prep67], the so-called PMC model describes a system of connected components. Each component tests a subset of other components of the system, and the test result is either that the tested component is fault-free or faulty. The *syndrome* is the collection of all test results. *System-level diagnosis* is a static analysis of a syndrome, resulting in the identification of up to t faulty components. In the original PMC model, fault-

free components that test other components report reliable test results, while faulty components can report incorrect results for tests of other components. The problem that system-level diagnosis solves is to arrive at a mapping between tester components and tested components so that the resulting syndrome reliably indicates which components are faulty, up to t faults. If such a mapping exists for a system of n components, the system is called t-diagnosable. Preparata et al. showed that for t-diagnosability, the system must have at least $n \geq 2t + 1$ components and that each component must be tested by at least t other components.

One practical weakness of the classic PMC model is that the test syndrome must be collected at a reliable centralized entity of the system. A centralized arbiter that is reliable and connected to all nodes of a distributed system can be expensive to implement, and it could also be a single point of failure. Kuhl and Reddy proposed algorithms to remove the requirement for a centralized arbiter. In [Kuhl80, Hoss84] algorithms are presented in which each component independently determines the syndrome by testing neighboring components to which they are directly connected and receiving messages containing test results from other components not directly connected to itself. Because faulty components can corrupt messages as they are forwarded to other components, there is redundancy among messages and message paths. An optimal algorithm presented in [Bagc91] required $3n \log p + O(n + pt)$ messages, where p is the number of participating fault-free components, and t is the number of faulty components.

2.2.2. The Byzantine Generals Problem

The seminal work on the Byzantine Generals problem was motivated by the design of a reliable aircraft control system known as SIFT [Wens78] in which system designers needed clock synchronization across distributed processors in the presence of a single faulty clock. This work showed that to mask the effects of f faulty clocks, at least $3f + 1$ clocks are needed.

Further work on the problem of clock synchronization resulted in the definition of a spectrum of models of computation for distributed systems. The most general of such models is the asynchronous model in which no time bound exists on the processors' relative speed nor on message latency. In contrast, the synchronous model assumes real-time bounds on process speeds and message latencies. In between these two extremes exist several partially synchronous models [Dwor88, Chan91]. The asynchronous model is the weakest, as any algorithm that can be implemented in this model will work correctly in a partially synchronous or synchronous model. However, a fundamental result of the asynchronous model of computation is that no deterministic algorithm exists to achieve consensus if just a single process experiences a crash fault, not to mention the more inclusive fault classes [Fisc85]. Intuitively, this so-called FLP impossibility result stems from the inability to distinguish a crashed process from a ''very slow'' process, which prevents any consensus algorithm from terminating.

In one partially synchronous model, bounds exist on processor speed and message latency but are not known *a priori*, and they only hold after some unknown time. It is proven in [Chan91] that in this model of computation, with crash failures, consensus is solvable with $n \geq 2t + 1$, where n is the number of processors and t is the maximum number of faulty

processors.

Lamport et al. recast the problem of achieving consensus in the midst of faulty processors as a problem of Byzantine generals of different cities agreeing to attack a designated city at a specific time by only passing messages to each other, now well-known as the Byzantine generals problem [Lamp82]. Messages could be delayed for a time, but not infinitely delayed, representing the use of the partially synchronous model of distributed computing. A subset of generals are allowed to be traitorous, meaning that they can lie in their messages, committing to attack when they would actually retreat instead. This ‘‘faulty behavior’’ is less restricted than that of crash failures. Lamport’s solutions for solving the Byzantine generals problem showed that for consensus to be achieved in the presence of f traitorous generals, there must be at least $n = 3f + 1$ generals. An algorithm was also given that involved sending $(n-1) \times (n-2) \times \cdots \times (n-f-1)$ messages.

Research in both areas of system-level diagnosis and the Byzantine generals problem share the conclusion that reaching agreement in the presence of f faulty components requires a majority of fault-free components and a large number of message exchanges (or tests and diagnosis rounds) to determine which components are faulty. In both models, the required number of components and messages is significantly reduced when a reliable centralized arbiter can be used.

2.3. Implementing Fault-Tolerant Systems and Applications

There are many fault tolerance techniques to choose from when implementing fault-tolerant distributed applications. This dissertation focuses not on the techniques themselves but on how these techniques are implemented within a layered software architecture. In

particular, fault tolerance techniques can be implemented exclusively in hardware, in a combination of hardware and system-level software, or across more layers including hardware, system-level software, application libraries, and applications themselves. The choice of the layer of implementation is dependent on applications' reliability requirements and on whether fault tolerance mechanisms need to be transparent to the applications.

2.3.1. Supporting Reliability Requirements of Multiple Applications

Some systems are designed to execute only a specific set of applications with common reliability requirements. In such systems the fault tolerance techniques can be implemented in hardware and lower layers of software, without specialization for any particular application. Furthermore, there is no need to interface with the application to determine its specific reliability requirements. All applications are executed reliably by employing the same fault tolerance techniques.

If the applications running on the system are diverse with respect to their reliability requirements, any single fault tolerance mechanism is likely to be unacceptable for use with all the applications. Specifically, it is likely to not be able to meet the reliability requirements of some applications but provide for others higher reliability than necessary with an unacceptably high performance overhead. Thus, such a system should support the simultaneous deployment of multiple fault tolerance mechanisms.

2.3.2. Transparency of Fault Tolerance Techniques to Applications

Fault tolerance implementation that is transparent to the application is not programmatically visible to the application and application developer. Because the fault

tolerance techniques are wholly implemented in layers below the application code, the application code is developed with the simplifying assumption that the system is reliable, that is, faults do not exist. Application transparency is very desirable because it frees the application programmer from the complexity of error detection and recovery code. Furthermore, application transparency of fault tolerance implementation enables legacy application code to be executed reliably without modifying the application code. Application-transparent implementation of fault tolerance techniques imposes a cost on application performance and resource usage due to the underlying implementation's assumption that applications do not have or use redundancy for fault tolerance. Therefore the implementation must create and manage redundancy on behalf of the application.

In non-transparent fault tolerance implementation, or application-specific fault tolerance, the fault tolerance technique is ''visible'' to the application developer. The application developer must provide input to the underlying system or must even implement the redundancy in application code. The benefit gained for the increased complexity in application code is that the fault tolerance technique can be more efficient. This is because the application programmer can use knowledge of application-specific characteristics to select a subset of application state for which redundancy is implemented. Thus, the overhead of maintaining redundant state is reduced.

The costs and benefits of application-transparent fault tolerance are described in more detail in Chapter 3. The following three sections describe related fault-tolerant systems in which fault tolerance mechanisms are implemented at various layers, from hardware to application code.

2.4. Implementation Using Modular Redundant Hardware

The first group of systems implement fault tolerance exclusively through the use of redundant hardware. These implementations provide fault tolerance that is transparent to all layers of software, including system-level software and applications.

□ **FTMP**

FTMP [Hopk78], was a fault-tolerant distributed computer designed for aircraft control, requiring high reliability for all programs running on the computer.

Triple modular redundancy was used for processor modules and memory modules. Each module was connected to a triply-redundant bus via a voter that voted on the inputs from the bus. Any single failed module of a triple-redundant module was automatically masked by the voter. Such failures included generation of wrong results. The system also automatically configured spare modules to replace faulty ones, based on the output of the voters. Thus, hardware failure was completely masked from software.

FTMP's design freed software developers from programming software to handle errors due to hardware failure. The cost of such a design is the inclusion of many redundant modules and custom logic connecting the modules together. Redundant and spare modules and buses cannot be used independently to increase computing and communication throughput, and the level of fault coverage is the same for all applications, regardless of the application's actual requirement for fault tolerance. For applications that are mission-critical or that involve maintenance of human life, such a design and its high cost are appropriate.

□ **Stratus**

The Stratus computer [Webb91] was computer system designed to provide fault-tolerant computing to financial service companies. Like FTMP, the Stratus computer implemented fault tolerance wholly in hardware, without any involvement of software.

A Stratus system was composed of processor boards, memory boards, and I/O controller boards that connected to a dual-redundant backplane bus. Processor boards each contained dual synchronized processors. A check circuit on the board compared the outputs of the two processors. If the comparison failed, the board would silently fail. Processor boards were paired so that each pair of boards (composed of a total of four processors) was clock-synchronized. Therefore, computation was not affected by the failure of one board of the pair because the other board continued operating correctly.

Memory boards and I/O controller boards also contained dual redundant components and self-checking hardware to enforce fail-silence. Multiple Strata systems could be connected together via a proprietary redundant bus known as StrataLINK. Each StrataLINK contained two independent links that could be used in parallel for increased communication throughput. When one link failed, failover to the remaining working link was automatic.

As in the case of FTMP, Stratus provided fault tolerance to all application software regardless of the application's actual reliability requirements, at the cost of modular redundant hardware.

□ **IBM System z10**

IBM's System z10 [Clar09] was a business computing system that used redundancy

within processors instead of modular redundant processors to implement fault tolerance. The quad-core z10 processor relied on inline error checking of functional units, parity bits and ECC to protect datapaths and state elements, and per-instruction checkpointing of architected state for rollback recovery.

The processor distinguished between ‘‘soft errors’’ and ‘‘hard errors’’. To recover from soft errors, the processor restored the checkpoint, purged its caches, and retried the failed instruction. To recover from hard errors, the core was halted and its job was migrated to another core on the processor. The error detection and recovery was implemented transparent to the application.

2.5. Implementation Using COTS Hardware and System-Level Software

In contrast to the systems in Section 2.4, the rest of the systems presented in this chapter use commercial-off-the-shelf (COTS) hardware to reduce the cost of implementing application fault tolerance. COTS hardware emphasizes performance over reliability. Reliability is achieved by employing multiple COTS hardware components, and redundancy is managed by custom system software. Specifically, these systems use system software to maintain replicas of applications for error detection and recovery.

□ **SIFT**

The SIFT system [Wens78], like FTMP, was designed to handle aircraft control. However, SIFT used multiple COTS processor modules and memory modules that were controlled by specially-programmed system-level software. SIFT’s modules were loosely coupled, only periodically synchronized through a Byzantine-fault-tolerant agreement

protocol. A global executive program replicated application processes across multiple processors.

Outputs of tasks were voted on by the global executive before being used as inputs for subsequent tasks. Output of the voters was used by the global executive to detect errors, diagnose the source of the errors, and remove faulty nodes from the system. As a critical program itself, the global executive was also replicated across multiple processors.

□ **Delta-4**

Delta-4 [Powe94] was a fault-tolerant general-purpose cluster computer. Like SIFT, the Delta-4 system was composed of COTS processing nodes. Unlike SIFT, Delta-4 was a cluster computer designed for running distributed applications.

Each Delta-4 node was equipped with a custom fail-silent network access card (NAC) that was connected to a local area network (LAN). The NAC internally used dual modular redundancy for its processors and memory to enforce a fail-silent property: If the states of the duplex modules diverged, special circuitry permanently disabled the NAC, essentially causing ‘‘failure’’ of the entire processing node. One component of the system-level software was a custom communication protocol programmed to run on the NAC processors. This protocol enabled a communication endpoint to be transparently replicated across multiple processing nodes. The protocol used totally ordered multicast and a Byzantine-fault-tolerant agreement protocol to mask inconsistent messages from faulty replicas.

The Delta-4 middleware used the NAC capability to replicate application processes across several nodes for fault tolerance. The degree of replication could be adjusted according to the reliability requirements of the application. An application that needed to tolerate f fail-

stop faults would be replicated across $f + 1$ nodes; an application requiring toleration of f Byzantine faults would be replicated across $2f + 1$ nodes. The actual replication of processes was application-transparent. Delta-4's centralized cluster manager, known as the Replica Domain Manager (RDM), was also replicated to tolerate f Byzantine-faulty replicas.

In the case that the communication protocol detected an inconsistent replica, notification was sent to the RDM which then directed reconfiguration of the application's replicas. The RDM terminated the faulty replica and replaced it with a new replica using the state of the surviving replicas. Thus, Byzantine faults in application replicas were completely masked, and faulty replicas were automatically and transparently replaced.

□ **RAIN**

The Reliable Array of Independent Nodes (RAIN) project [Boho01] was a research project focused on tolerating permanent and transient faults in the network of cluster computers. In the RAIN cluster, nodes were equipped with multiple network interface cards (NICs) for fault tolerance as well as for additional communication throughput, and the network topology was redundant to tolerate the failure of any single NIC, link, or network switch.

RAIN middleware, running on top of the Linux operating system, monitored and provided a consistent view of link status. The middleware also implemented a distributed group membership service, providing a consistent global view of fault-free nodes connected to the cluster. The group membership service tolerated nodes that were fail-stop faulty. RAIN middleware also included a library for running message-passing parallel programs, and this middleware enabled message-passing applications to use the increased communication

throughput of redundant links, similar to the redundant StrataLINK of the Stratus system. The system afforded no other fault tolerance mechanisms for message-passing parallel programs other than masking any single fault in the network, resulting in a graceful degradation of communication throughput.

2.6. Implementation Using COTS Hardware, COTS OS, Cluster Middleware and Application Software

The last group of systems are those that use COTS hardware and COTS system-level software to build very low-cost systems that support high-performance computing (HPC) applications. Fault tolerance for applications, if it exists at all, is implemented in user-level software, in middleware or in the application code itself.

HPC applications typically have looser real-time requirements than mission-critical applications, that is, they tolerate longer times to detect and recover from an error. HPC applications can trade off short recovery time for higher computation throughput. Therefore, research on implementing fault tolerance for these applications focuses on fault tolerance mechanisms that have lower performance overhead than replication of the entire application.

2.6.1. Cluster Management Middleware

Clusters are distributed computers built from COTS compute nodes connected by a COTS network, with each node running a COTS single-node OS. A middleware layer of user-level software, called the cluster management middleware (CMM), exists to manage the compute nodes and the applications running on them. At the bare minimum, CMM is responsible for allocating compute resources to distributed applications, starting applications

(including remote spawning of processes), scheduling applications for execution, and cleaning up terminated applications. Previous CMM designs have also provided various functionality for implementing application fault tolerance, which are reviewed in this section.

- **GLUnix**

One of the earliest cluster management middleware projects, GLUnix [Ghor98] provides a single-system image to the user, abstracting the cluster as a single system capable of executing parallel and sequential jobs. GLUnix strives to implement services similar to those present in single-node UNIX OS. For example, a signal sent to a parallel application is atomically broadcast to all of the processes of the application. GLUnix also implements signaling of individual processes.

GLUnix provides minimal facilities for application fault tolerance: when the middleware detects that a process of a distributed application has crashed, it immediately terminates and cleans up the surviving processes of the application. Typically, applications that must survive such a failure would employ checkpointing, relying on the user to manually restart the application with special command-line arguments that instruct the application to start executing from the checkpointed state. While this solution may be effective for clusters with very low rates of failure, it puts an unreasonable burden on the user for managing application checkpoints and restarts.

- **Chameleon**

Chameleon [Kalb99] is an object-oriented software infrastructure for supporting fault-tolerant execution of applications on COTS clusters. The infrastructure includes library of

objects that application programmers can invoke for implementing application-level fault tolerance mechanisms, such as heartbeat monitoring, application-directed checkpointing and rollback, and output fanout and input voting for supporting replicated processes. For existing parallelized message-passing applications, Chameleon demonstrated an MPI application modified to increment a progress counter that is monitored for detection of application hangs.

The CMM itself also employed the same objects for its own fault tolerance, with all CMM processes monitoring heartbeats from each other in order to tolerate CMM process crashes.

The Chameleon CMM and fault tolerance support for applications is very similar to the present work in that application-specific fault tolerance was implemented with modifications in application code. However, the interface between applications and the CMM was not clearly defined and explained. There also was not any consideration for distributed applications tolerating faults other than process crashes and hangs.

- **Starfish**

Starfish [Agba99] is a distributed heterogeneous run-time environment for MPI applications that provides application-transparent checkpointing and rollback recovery and an extended API for application-directed checkpointing and reconfiguration. Application processes can be run in a cross-platform portable virtual machine, which allows a process to be checkpointed and restarted on a node with a different native OS. The Starfish CMM provides an API for use by applications to add and remove processes during run-time, but it was not explained how these reconfigurations are integrated with MPI communications.

- **HARNESS**

HARNESS [Dong98] was a cluster manager that provided services to distributed applications for tolerating process crashes. When a process of a distributed application crashed, HARNESS notified the surviving processes of the application. HARNESS also enabled application processes to request new processes to be spawned and added to the application.

The HARNESS services were used to implement FT-MPI [Fagg05], a modified MPI API that enabled MPI applications to survive process crashes. FT-MPI is described in more detail in Subsection 2.6.3

2.6.2. Checkpointing and Rollback Recovery for MPI Applications

One common API used by HPC applications is the Message Passing Interface (MPI) [Mess95]. MPI enables portable development of parallel applications by providing a high-level abstraction of processes and communication operations among processes. The MPI standard specifies that all communications among application processes are reliable, but it does not specify any other behavior in the presence of errors. In particular, the state of the message-passing layer is undefined in the presence of crashed processes. For this reason, much of the research on providing fault tolerance for MPI applications focuses on the use of application-transparent checkpointing and rollback recovery, where rollback recovery is accomplished through termination of the application and restarting of the application with a flag to indicate that execution should begin from checkpointed state.

The challenge of checkpointing concurrent communicating processes is to avoid

committing inconsistent checkpoints. When a process delivers a message from a sending process, a dependency is formed of the receiver's state on the sender's state. A global checkpoint of a distributed applications is *consistent* if and only if, for every message in the checkpointed state that is received, the checkpointed state also includes the sending of that message; otherwise, the global checkpoint is *inconsistent*. Several protocols have been designed to ensure that checkpoints are consistent, including coordinated checkpointing [Stel96] and message logging [Bosi02, Bout03]. For a full survey of these checkpointing protocols, see [Elno02].

□ **RENEW and Egida**

Two projects, RENEW [Neve98] and Egida [Rao99] stand out from the many works on checkpointing MPI applications not only because they implemented and compared multiple checkpointing protocols but also because they designed a general framework for the implementation of new checkpointing protocols. Having a common framework for implementing different checkpointing protocols facilitates comparison among the protocols.

In both RENEW and Egida, the implementation of a checkpointing protocol forms a ''layer'' within the message-passing middleware. Below the checkpointing protocol layer, RENEW and Egida frameworks provide implementations of common low-level services related to checkpointing, such as committing state and logging events to stable or volatile storage, and tagging application messages with additional information. Above the checkpointing layer, the frameworks define a set of events to which a checkpointing protocol may react. For example, coordinated checkpointing protocols may rely on the expiration of a periodic timer to start the checkpointing process; the framework implements timer events and

invokes a timer-expired procedure that is provided by the checkpointing protocol layer. Similarly, a message logging protocol must perform actions for every message receipt; the framework invokes the checkpoint protocol layer's message-receipt procedure whenever a message is received.

The existence of the RENEW and Egida frameworks is evidence of the maturity of the research on application-transparent checkpointing for message-passing applications as well as the acceptance of checkpointing as a valuable fault tolerance technique for high performance computing despite its costs. Such costs include performance overhead to commit state and message logs to storage, communication overhead, and requirements for a large amount of stable storage. These costs can be reduced or eliminated if checkpointing is instead implemented at application-level code.

2.6.3. Application-Specific Fault Tolerance Mechanisms

We have discussed the use of replication of application processes to tolerate Byzantine faults and the use of rollback recovery in tolerating system-detected crash faults. In this subsection, we present several fault tolerance approaches that enable a distributed application to detect and recover from non-crash faults without the overhead of replicating entire processes or redundantly storing the entire application state. These mechanisms are implemented in part or wholly in application code; that is, they are not application-transparent.

- **Algorithm-Based Fault Tolerance**

Algorithm-based fault tolerance (ABFT) refers to the technique of redundantly encoding

application data, modifying the application's algorithms to operate on the encoded data, and using the encoded results to detect and correct errors and diagnose faulty components. In the seminal work [Huan84] ABFT was introduced and applied to matrix operations. Matrices are augmented with checksum rows and columns. The matrix operation operates on the augmented matrix operands, producing an augmented result matrix. The checksum rows and columns of the result matrix are used to detect and correct errors in the result matrix. For example, a ABFT matrix multiplication procedure can be parallelized to run on processors configured in 2-dimensional mesh in which inputs can be multicasted to all the processors in a row or column. Each processor computes one location in the result matrix. When the result matrix is obtained, checksum rows and columns are recalculated and compared to the result matrix's checksum rows and columns. An inconsistency is used to correct the result matrix as well as to locate the faulty processor. Reconfiguration to eliminate or repair the faulty processor was not described in the original ABFT work.

ABFT for matrix operations is scalable. For matrices of size $n \times n$ the amount of processing for the original non-fault-tolerant algorithm is $O(n^2)$. The additional redundant processing of checksum rows and columns is on the order of $O(n)$. The main drawback of ABFT techniques is that they only apply to a narrow set of algorithms, namely matrix operations.

□ **Distributed Recovery Blocks**

Recovery blocks with acceptance tests [Rand75] are commonly used to tolerate faults due to software design faults, but they can also be used to tolerate transient faults. An application is divided into blocks of code, with each block possibly having multiple

implementations, or ‘‘try blocks.’’ The result or application state following every try block is test against an acceptance test. The acceptance test indicates whether the try block executed correctly. If the acceptance test fails, application state is rolled back to the state before the try block was entered and a secondary try block is executed.

The work in [Kim89] describes a system for distributing the execution of recovery blocks across multiple processors. This idea is combined with system-level fault detection in [Hain00] for efficient application-level fault tolerance. Their fault tolerance scheme applies to a distributed application in which a frame of input data is divided to be processed by processors in parallel. Each process executes a primary try block which performs computations on its portion of the input. Then, each process executes a backup try block which performs computations on a neighboring process’s portion of the input. For lower overhead, the backup try block only computes an approximate result based on a reduced-precision version of the neighbor’s portion of input. Thus each process maintains a full-precision version of its own state as well as a reduced-precision version of one neighbor’s state. When an acceptance test on the results of a primary try block fails, the process recovers by rolling forward to the reduced-precision version of the neighbor that serves as its backup, instead of rolling back to a previous full-precision version of its state.

This scheme was applied to a real-time distributed target-tracking application in which thirty moving targets must be tracked in noisy radar data. In this application, rollback recovery was too slow to satisfy real-time requirements. Their results show that when backup try blocks computed based on just 15% of the primary process’s input, no targets were lost and no real-time deadlines were violated.

The target-tracking application only handled fail-stop faults. If each primary block is

backed up by multiple backup blocks executed by different processes, the multiple results can be compared to detect and recover from faults causing incorrect results.

- **FT-MPI**

FT-MPI is an extension of MPI that enables applications to implement application-specific recovery mechanisms to recover from process crashes. As mentioned in subsection 2.6.2, the MPI Standard states that the state of the MPI communication layer and the MPI application is undefined after a process crash. In contrast, FT-MPI [Fagg05] modifies MPI semantics so that after process crashes, surviving processes can restore a valid state for the message-passing layer, possibly replacing crashed processes with newly spawned ones, and perform recovery of application-level state.

With FT-MPI, after an MPI application process crashes, all MPI communication calls return a special error code to the application. Furthermore, if any process is blocked in a blocking MPI communication call, the operation is unblocked, and the MPI function returns the same special error code. The MPI application can then initiate roll-forward recovery of the state of the message-passing layer by calling an FT-MPI recovery function.

The implementation of FT-MPI depends on the ability of an application to detect crashes of remote processes. FT-MPI was implemented for the HARNESS CMM [Dong98], which had a facility for notifying application processes of process crash events. FT-MPI semantics only cover tolerance of process crashes. The present work builds upon FT-MPI semantics to enable MPI applications to detect and recover from other kinds of faults.

2.7. Summary

All fault tolerance activities in a distributed application—error detection, diagnosis, recovery, and reconfiguration—involve agreement among the processes of the application. In this chapter we have reviewed research on the theoretical limits of solutions for agreement in distributed systems with faulty processors, showing that toleration of f processors that exhibit fail-stop behavior requires at least $n = 2f + 1$ processors, and for processors that exhibit arbitrary behavior, that number increases to $n = 3f + 1$ processors. Furthermore, the algorithms used to reach consensus in such situations require many messages, the number of which scales super-linearly with respect to n. Much simpler algorithms can be used when an pre-designated always-reliable arbiter can assist in reaching consensus. The next chapter explains how the CMM can perform such a role.

This chapter also reviewed several CMM with respect to how they interface with fault-tolerant distributed applications. As far as we know, none of the research on CMM design presents a careful analysis of which fault tolerance services or techniques should be implemented by the CMM versus which should be implemented in higher layers of software such as application libraries and the application code itself.

Finally, this chapter reviewed presented some application-specific fault tolerance techniques, showing that such techniques may still require interaction with CMM when they are used in applications that are distributed across many compute nodes.

Chapter Three

Layered Fault Tolerance for Compute Clusters

Fault tolerance for high-performance distributed applications is increasingly important due to unreliable cluster nodes. As the number of nodes in a cluster increases, the probability of a single node failure at a given time also increases. Furthermore, the reliability of integrated circuits may decrease due to shrinking feature size and lower voltages. Clusters can also be deployed in harsh environments, where radiation and other conditions can cause malfunction in the hardware. Although much work has focused on making high-performance applications fault-tolerant, most of the work is concerned only with fail-stop faults, in which faulty processes crash. Such faults are easy to detect, and, given that a copy of fault-free application state exists, easy to recover from.

The goal of this work is to enable distributed applications running on clusters to detect and recover from process crashes, process hangs and arbitrary faulty behavior, such as the generation of incorrect results. All of these errors are easily detected by replicating applications and comparing the outputs of the replicas. However, replication imposes a great cost—at least n times the resources required to maintain n replicas. There exist many fault tolerance mechanisms that enable detection and recovery from hangs and incorrect results while using fewer resources than replication. Our work focuses on system support needed to implement such fault tolerance mechanisms for distributed applications.

This chapter describes in general terms what is needed to implement fault tolerance mechanisms for distributed applications and gives justification for implementing certain

functionality as CMM services. Section 3.1 firstly explains a general principle for the greater efficiency of application-specific fault tolerance mechanisms over fault tolerance mechanisms that work for all applications. Section 3.1 also describes exceptions to the principle, leading to an implementation that involves multiple software layers. a way that is not application-specific. Section 3.2 explains the use of CMM for implementing services that support fault tolerance mechanisms for applications. Finally, Section 3.3 proposes a set of CMM services to be implemented.

3.1. End-to-End Arguments for Application-Specific Fault Tolerance

The general solution for implementing application fault tolerance is to introduce redundancy for the application's state. For example, checkpointing is a fault tolerance mechanism that stores copies of application state made at specific points in time in an application's execution. Process-level replication is another example of a mechanism that maintains copies of the application, where replicas of applications are simultaneously executing. Both of these mechanisms can be implemented in an application-transparent way so that they work for all applications in general, regardless of the structure of the application's data or the behavior of its algorithms.

The cost of fault tolerance mechanisms in terms of resource usage and performance overhead is due mainly to the maintenance of redundant state. The amount of state to maintain can be reduced significantly by taking advantage of application-specific characteristics. For example, a checkpointing mechanism may omit committing application state that can be quickly recomputed from the checkpointed state after rollback recovery; a replication mechanism may use alternative algorithms that approximate the original algorithm

while using fewer computational resources. The gain in efficiency from using application-specific characteristics for adding redundancy comes at the cost of implementing the mechanisms for each application or group of applications with similar characteristics. Assuming that the gain in efficiency outweighs the additional cost in implementation, we can argue that all fault tolerance mechanisms should be implemented in application-specific manner as much as possible.

Implementing application-specific fault tolerance mechanisms is an instance of the end-to-end argument in system design [Salt84]. The argument for end-to-end design states that certain functionality can be implemented correctly and completely only with the knowledge of the application running at the endpoints of a communication system; providing the functionality as a feature of the communication system is not possible or may be redundant. The application of the end-to-end argument to fault tolerance mechanisms involves what layers the mechanisms should be implemented at and whether the mechanisms should be specific to the applications they are supporting. Some mechanisms such as detection of node failure operate without any dependence on application-specific knowledge and would offer no benefit from application-specific knowledge. Other mechanisms benefit from application-specific knowledge, as described above.

Given that it is sometimes desirable to implement application-specific fault tolerance mechanisms, we must consider what layers should be involved in the implementation. In typical clusters employing COTS hardware and COTS OS, there are few facilities for the hardware or OS to easily determine application-specific characteristics for implementing more efficient fault tolerance mechanisms. To implement such facilities could complicate hardware and OS for the benefit of a few kinds of applications while incurring significant cost for all

applications. Part of this cost may involve instrumenting OS code or adding measurement capability in hardware to measure and analyze events at run-time that indicate the application's behavior. We assume that modification of the hardware and OS to make such characteristics visible within these layers is expensive to the point that it negates the low-cost motivation of using COTS hardware and COTS OS in the first place. The remaining option is to implement application-specific fault tolerance mechanisms in user-level processes, involving CMM, application libraries, and the application code.

A typical CMM manages execution of applications on a general-purpose high-performance cluster using very little knowledge of application-specific characteristics. Application-specific procedures are implemented in application libraries and application code in order to keep the CMM implementation simple. However, there are some cases where involving the CMM in implementation of application-specific fault tolerance mechanisms yields benefits. The next section describes these cases.

3.2. Involving the CMM in Implementation of Fault Tolerance Mechanisms for Applications

There are two cases where involvement of the CMM in implementing application-specific fault tolerance mechanisms can be beneficial, even though there are no application-specific procedures implemented in the CMM. The first case are the functions that application-specific fault tolerance mechanisms need to perform but which must be implemented in the CMM and OS. For example, an application fault tolerance mechanism may need to terminate a faulty process and spawn a new process to replace it. The CMM manages processes on behalf of distributed applications; permitting a potentially faulty

application to perform these tasks could impact the health of other applications and of the cluster itself. The second case are the functions that can be implemented more efficiently in lower layers. For example, an OS immediately detects a process crash whereas a distributed application must use heartbeat messages and timeout events to detect a process that has crashed.

It is conceivable that some of the functionality of the second case presented above could be implemented in an application library instead of involving the CMM. The benefit of using the CMM is that the CMM has very high reliability requirements. As the manager of the cluster and all applications running on the cluster, the CMM is a critical component of the cluster and must be reliable. When the CMM fails, the entire cluster has failed, and there is nothing an application can do to guarantee correct execution. In other words, applications running on a cluster implicitly depend on the correct operation of the CMM. Adding CMM functionality which application-specific fault tolerance mechanisms can invoke simply makes this dependence explicit.

3.3. CMM Services for Supporting Application Fault Tolerance

In this section we propose a set of services to implement in CMM in order to support fault tolerance mechanisms for distributed applications. We consider general requirements of fault tolerance mechanisms performing four actions: error detection and notification, error diagnosis, error recovery, and reconfiguration of application processes. An error is detected when a fault tolerance mechanism concludes that the state of the system or application is incorrect. The error detection mechanism must notify the system or application so that normal execution can be stopped. In error diagnosis, fault tolerance mechanisms identify and

isolate the erroneous state. In error recovery, fault tolerance mechanisms restore the state to be error-free. Finally, in reconfiguration, fault tolerance mechanisms repair or replace a faulty component so that it cannot commit more errors in the future (unless it is stricken with another fault).

3.3.1. Error Detection and Notification

Detecting hangs and arbitrary incorrect behavior of a faulty application process requires application-specific knowledge. Hence, one must use application-dependent code to implement detectors for these kinds of errors. Detecting a process hang requires knowledge of how much time application algorithms take to execute. Detecting missing output, extraneous output, and incorrect output requires knowledge of the application's specifications regarding correct output. Hangs and arbitrary incorrect behavior can be detected by replicating the entire application and comparing the outputs of the replicas; this is a special case of application-dependent code where multiple instances of the application code itself are executed.

To keep the CMM simple, we refrain from implementing functionality that depend on application-specific characteristics. Therefore, we do not implement in the CMM replication or any other service that assists in detection of hangs and arbitrary incorrect behavior.

Crashes and hangs both result in a failure to make progress and can be detected by application-specific code in which application processes periodically compare the progress of the application to the passage of time. As mentioned earlier in this chapter, OS software already detects process crashes. Since the OS's detection mechanism is event-based, in contrast to the application's slower timer-based mechanism, an application would respond to

crashes more promptly if it handled crash notifications based on the OS's detection.

Although the single-node OS is not able to notify processes running on other nodes of a crash, it can notify the CMM, which in turn can notify the remaining processes of the application. Crash notification is therefore a useful service that CMM can implement without knowledge of application-specific characteristics.

An application process is prematurely terminated when the node that it is running on has failed. Since the entire node is lost, this can only be detected by the CMM or the application itself, portions of which are still running on other nodes. The CMM must detect node failures using its own mechanisms since it has to keep track of available cluster resources. Since the CMM manages cluster resources, it has the information regarding which nodes each application is using. Hence, it is very simple to add to the CMM the capability of notifying the application when one of the nodes the application is using fails.

In the above discussion we identified three sources of error detection: the operating system, for process crashes; the cluster manager, for node failures; and application-specific code, for hangs and corruption of application-level state. Once any error is detected, the distributed processes must be notified quickly for coordination in diagnosis and recovery.

Notification of error detection can be accomplished through the distributed application's message-passing facility. However, there are several reasons to separate the error notifications from normal messages related to the application's distributed computation. First, notifications of error detection may originate from entities other than the application itself. For example the cluster manager may detect a failed node, or the message-passing implementation may detect a broken connection. It would be intrusive to modify existing application code so that

these entities are treated as communication endpoints on the same level as the application's processes. Second, an error notification should be handled as soon as possible in order to prevent the spreading of corrupted state through message-passing. In particular, processes must be able to handle such notifications even if they arrive while the process is blocked. For example, a fault-free process expecting to receive a message from a sender may be blocked in a blocking message receive operation at the time of the error. The error may have affected the sender such that it fails to send the message that the receiver expects. The fault-free receiver in the blocking operation must be interrupted by the error notification so that it can execute diagnosis and recovery mechanisms.

The above discussion leads us to a solution whereby the CMM provides a reliable asynchronous communication service for distributing error notifications. The service enable communication between the cluster manager and the application, and it enables one application process to interrupt other application processes, even if they are blocked in message-passing operations.

3.3.2. Diagnosis

Diagnosis is the process of identifying the faulty components in a system. For a distributed application, the ‘‘components’’ are the individual processes. Hence, one way diagnosis can be implemented by applications is for application processes to perform system-level diagnosis [Prep67], where the processes of the application test each other. As mentioned in Section 2.2.1, producing a correct diagnosis in a distributed system is complex because faulty components can fail to send messages they are supposed to send, send more messages than they are supposed to send, or send incorrect messages. With a fault-tolerant

CMM, the cluster manager is a reliable and trusted entity that can potentially simplify the diagnosis process. For example, the reliable trusted central cluster manager can provide a very simple mechanism for identifying a majority vote among diagnoses produced by the application processes and transmitting the results to all fault-free processes.

3.3.3. Error Recovery

There are two ways to restore error-free state: rollback recovery and roll-forward recovery [Camp86]. In rollback error recovery, an earlier error-free state of the application, is restored, and computation resumes from that earlier state. The end-to-end argument applies to rollback because the application programmer has the most awareness about what application state is critical for correct rollback and at what points in the execution such state should be committed to reliable storage.

In roll-forward error recovery, the application replaces its erroneous state with newly-created correct state and continues execution. The new correct state may not have been reached in the past, and it may also not have been reached had there been no error. Knowing how to create a correct state involves application-specific knowledge, so the end-to-end argument applies to the implementation of roll-forward recovery.

3.3.4. Reconfiguration of Application Processes

Reconfiguration consists of actions taken to prevent a faulty component from causing another error. In terms of processes of a distributed application, reconfiguration may involve removal of processes diagnosed to be faulty, and replacement of faulty processes with fault-free processes. The CMM manages processes on behalf of applications to maintain the health

of the cluster and to prevent an application from interfering with the processes of another application. Therefore, implementing reconfiguration of application processes must involve communication between the application and the CMM.

3.4. A Set of CMM Services for Supporting Application Fault Tolerance

This chapter explains why some functionality related to application fault tolerance can be easily implemented in the CMM, beneath the application layer, while other functionality are best left to the application-specific code for implementation. To keep CMM algorithms simple and reliable, we avoid introducing application-specific routines to the CMM. Instead, in defining a set of services to add to the CMM we consider only functions that must be implemented by the CMM and functions that are more efficiently implemented by the CMM and system software.

We propose in this chapter the following set of CMM services that can be implemented to support application fault tolerance: (1) an asynchronous communication facility enabling communication between applications and the cluster manager and enabling interruption of application processes for error notification, (2) error detection for prematurely terminated processes due to crashes and node failures, (3) a reliable voting service, (4) a service to terminate an application process and (5) a service to spawn an application process.

Chapter Four

Design and Implementation of the Layered Approach

Having identified in Chapter 3 a set of services to implement in CMM for supporting application fault tolerance, we proceed in this chapter to define the way a distributed application communicates with the cluster manager. This chapter also presents the application programming interface (API) we designed for developers of application libraries and MPI applications. Our work on CMM is based on an existing Byzantine fault-tolerant CMM called Ghidrah. This chapter ends with a description of the Ghidrah CMM and the modifications made to implement the new CMM services.

Section 4.1 introduces the Ghidrah signal API, a feature of the Ghidrah CMM to facilitate sending and delivery of short asynchronous messages among the cluster manager and application processes. Allowing application processes and the cluster manager to generate asynchronous notifications can result in complex situations where multiple asynchronous events exist at one time and must be handled in an orderly way. Thus, in this section, we present a protocol for handling asynchronous notifications and service requests and replies among the cluster manager and the application processes. The protocol manages this complexity with the goals of minimizing the amount of state maintained by the cluster manager and simplifying error handling and recovery code in applications.

While application programmers are free to use the Ghidrah signal API described in Section 4.1, we designed another API at a higher level of abstraction for developing fault-

tolerant MPI applications. Section 4.2 begins by describing the extensions of previous work known as FT-MPI to enable MPI applications to recover from process crashes. The section then presents G−MPI, our extensions to FT-MPI to enable application-specific error detection and recovery, including detection of and recovery from process hangs and corruption of application-level state. Example source code of a fault-tolerant G-MPI application is presented in Section 4.3.

The remaining sections of this chapter discuss further implementation details. Section 4.4 first describes the Ghidrah CMM as it existed before our work on adding CMM services for application fault tolerance. Then it describes our modifications to implement group requests and process termination and spawning services. While implementing and testing the CMM services, we discovered flaws in Ghidrah's group communication protocol used for communication among CMM processes. Section 4.5 describes Ghidrah's group communication protocol, the flaws we discovered, and the modifications made to remove the flaws.

4.1. Asynchronous Communication for Fault Tolerance Mechanisms

As explained in Subsection 3.3.1, asynchronous communication is needed among application processes and between the application and the cluster manager. Here, ''asynchronous'' means that the recipient of a message is interrupted to deliver the message. This enables notification of error detection to be delivered to application processes as soon as possible, whether a process is executing a long local computation or is blocked on a message-passing operation.

Subsection 4.1.1 describes a facility of the Ghidrah CMM that enables asynchronous

communication among the application processes and the cluster manager. Then Subsections 4.1.2 and 4.1.3 develop a protocol for the usage of Ghidrah signals for application fault tolerance.

4.1.1. Ghidrah Signals

The Ghidrah CMM provides asynchronous signaling for applications called *Ghidrah signals*. The motivation for the design of Ghidrah signals is to enable application processes to interrupt and communicate with each other for coordinating the execution of fault tolerance mechanisms. Ghidrah signals also enable the cluster manager to interrupt the application in case of an error detection such as a node failure. Ghidrah signals are not intended for ‘‘normal’’ application communications. For ‘‘normal’’ communications, applications use a message passing library, such as MPI, in which application processes communicate directly through the cluster network.

A Ghidrah signal can be sent from an application process to the cluster manager or to another process of the same application. A Ghidrah signal can also be broadcast to all the processes of a particular application from the cluster manager or from one of the processes of the application. Even if a broadcast originates from an application process, *all* the processes of the application, including the sender, receive the broadcast.

Ghidrah signals that are sent from an application processes to one or all the processes of the same application are forwarded through the cluster manager (see Section 4.1.3). Hence, all signals from a particular source arrive at their destinations in the order sent (with the exception of broadcasts that are dropped, as explained later in this subsection). Furthermore, all broadcasts arrive at all the application processes in the same order. Since signals are sent

through the reliable CMM, if a sender process delivers a signal to the local CMM *agent* (see Section 4.4.2) the CMM ensures that the signal will reach its destination(s), as long as the destination node(s) is/are fault-free.

`void handler(int signum, int srcpid, int destpid, int arg)` : Function prototype for Ghidrah signal handler. Invoked in signal-handling context to deliver a Ghidrah signal to the application process. The handler is executed atomically and must not block.
`srcpid` and `destpid` are application process identifiers assigned by CMM.
If `srcpid` is GSIG_CM, then the source is the cluster manager.
If `destpid` is GSIG_BROADCAST, then the signal was broadcasted to all processes of the application.

`void G_Signal(int signum, int destpid, int arg)` : Sends Ghidrah signal of signal number `signum` with argument `arg` to `destpid`.
If `destpid` is GSIG_BROADCAST, then the signal is a broadcast request to the CMM. If performed by the CMM, signal `signum` with argument `arg` will be broadcasted to all processes of the application. `srcpid` of the broadcasted signal will be set to the process identifier of the requesting process.
If `destpid` is GSIG_CM, then the signal is a group request for CMM service `signum` with argument `arg`.

`void G_Signal_handler(int signum, void (*handler)(int, int, int, int))` : Registers function `handler` as the signal handler for Ghidrah signal number `signum`.

Figure 4.1: API for Ghidrah signals

The Ghidrah signal API, implemented in a C-language application library is described in Figure 4.1. The Ghidrah signal is a short message containing a signal number, source and destination identifiers, an optional argument, and an epoch number. The meaning of the signal number is application-defined, except for a set of pre-defined signal numbers specified in Table 4.1. The meaning of the argument depends on the signal number. The use of the epoch number is opaque to the application programmer; its usage is explained in Subsection 4.1.3.

Application processes request CMM services by sending Ghidrah signals to the cluster manager. Each service is requested with a unique pre-defined signal number; some services

Signal	Type	Meaning
GSIG_REQ_SETQUORUM	group request	request to change quorum size to `arg`
GSIG_REQ_KILL	group request	request to kill process with VPID `arg`
GSIG_REQ_SPAWN	group request	request to spawn and add a process
GSIG_REQ_SYNC	group request	request to synchronize on value `arg`
GSIG_NTF_DIED	broadcast	notification of crash of VPID `arg`, or reply for GSIG_REQ_KILL: kill of VPID `arg` completed
GSIG_REP_QUORUMSET	broadcast	reply for GSIG_REQ_SETQUORUM: quorum size set to `arg`
GSIG_REP_ACK_KILL	broadcast	acknowledgment of GSIG_REQ_KILL: kill of VPID `arg` initiated
GSIG_REP_ACK_SPAWN	broadcast	acknowledgment of GSIG_REQ_SPAWN: spawn and add a process initiated
GSIG_REP_SPAWNED	broadcast	reply for GSIG_REQ_SPAWNED: spawn and add a process completed; `arg` contains process's VPID and node address
GSIG_REP_SYNC	broadcast	reply for GSIG_REQ_SYNC: synchronized on value `arg`
GSIG_ERR_DISAGREE	broadcast	group request failed due to no quorum

Table 4.1: Pre-defined Ghidrah signals. Pre-defined group requests are signals sent from application processes to the cluster manager; pre-defined broadcasts are signals broadcasted from the cluster manager to application processes.

also require an argument included in the signal. The top section of Table 4.1 defines each service and lists the signal name the process uses to request that service.

Upon completing a requested service, the cluster manager replies with a broadcast Ghidrah signal. The reply contains a pre-defined Ghidrah signal number that corresponds to the group request that was performed, as defined in Table 4.1. For example, the table specifies that the reply for the process termination service GSIG_REQ_KILL is GSIG_NTF_DIED. The reply also contains the argument that was specified in the group request except in the case of the spawn service: the group request GSIG_REQ_SPAWN does

not specify an argument, but the corresponding reply GSIG_REP_SPAWNED contains an argument specifying the spawned process's process identifier and node address. The purpose and meaning of the two ''acknowledgment'' broadcast signals, GSIG_REP_ACK_KILL and GSIG_REP_ACK_SPAWN, are described in Section 4.1.3.

After the application process sends a request for a service to the cluster manager, the cluster manager always indicates its response in the immediately succeeding broadcast signal. The cluster manager's response will be either a reply to the application process's request, or it will be some other broadcast communication. The application process keeps a copy of each request it sends until it determines whether the request has been accepted or dropped. Since the process is allowed to have only a single outstanding request, this is not a significant overhead. The application process determines whether the cluster manager's response is a reply to its own request by comparing the contents of the broadcast signal to its own request. Specifically, the signal from the cluster manager is a reply to the process's request only if the broadcast's signal number corresponds to the request's signal number (as defined in Table 4.1) *and* the broadcast's argument is equal to the application process's request argument. Otherwise, this response indicates that the cluster manager has dropped this application process's latest request. Again, there is an exception for the spawn request: the cluster manager's response GSIG_REP_SPAWNED is a proper reply to an application process's GSIG_REQ_SPAWN request even though the arguments in the request and the reply do not match.

4.1.2. Group Requests

An application's requests for service from the CMM must be fault-tolerant. A faulty application process could request a service in error, which could possibly cause further failure. For example, a faulty process could request that a fault-free process be terminated. To tolerate such behavior, requests to the cluster manager must be sent redundantly from a quorum of processes. The group of redundant Ghidrah signals sent from a quorum of processes is called a *group request*. Redundant Ghidrah signals match if all contents of the signals are equal except for the identifier of the signals' senders. Once the cluster manager has received matching redundant Ghidrah signals from a quorum of processes, it delivers the group request and begins performing the service.

As explained in the previous section, the cluster manager broadcasts a reply for each accepted request. Hence, using the mechanism discussed in the previous section, the application process can determine whether the request it sent has been accepted or dropped.

An example of a group request and cluster manager reply is illustrated in Figure 4.2. In this example process f has sent an incorrect group request t due to an error. Processes x, y and z are fault-free; hence they send matching group requests u. When the cluster manager receives the requisite number of matching group requests (in this case, 3), it accepts the request and performs service u. Finally, it replies to the application by broadcasting a.

The cluster manager may reject a group request and indicate that rejection by broadcasting to all the processes of the application a GSIG_ERR_DISAGREE signal. Such rejection happens as soon as the cluster manager receives requests (signals) from a sufficient subset of the application's processes to determine that reaching a quorum is not possible.

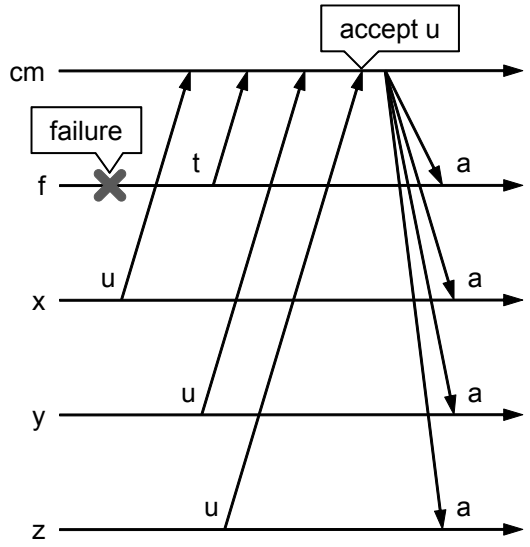


Figure 4.2: Group request and cluster manager's reply, with one faulty application process. The quorum size is 3.

Furthermore, the request will be rejected if a quorum of matching requests is not received within a preset timeout period. This timeout period begins as soon as the cluster manager receives the first group request. Typically, group requests are made by application processes that have already executed a distributed diagnosis procedure. The communication in the diagnosis procedure effectively synchronizes the processes. Hence, assuming known bounds on both message delays and differences in the relative speeds of different processes, group requests from a quorum of fault-free processes should be received by the cluster manager within some preset timeout period.

Error notifications from the cluster manager interrupt the receipt of a quorum of matching group requests, implicitly informing all of the application processes that the group request has been dropped. Broadcast requests from one of the application's processes could, potentially, also interrupt the collection of a quorum of matching group requests. However,

special mechanisms in the protocol can prevent such potential interruptions initiated by possibly faulty processes. The issues related to the interruption of group requests are explained further in Subsection 4.1.3.

The quorum size, the required number of matching group requests the cluster manager must receive, can be set by the application. When the application is started, the quorum size is initialized to the number of processes in the application. The application can reduce the quorum size according to how many simultaneously faulty processes it can tolerate. For example, an application of n processes that tolerates one faulty process would set the quorum size to $n-1$. Reducing the quorum size reduces the number of group requests the cluster manager has to receive and buffer.

4.1.3. Simplifying the Handling of Ghidrah Signals

Ghidrah signals give every application process and the cluster manager the ability to send interrupting messages at any time. Consider the use of Ghidrah broadcast signals for notifying applications processes of error detections. Errors can be detected at any time by any of the n application processes and by the cluster manager. Multiple errors can cause multiple broadcast signals to be generated; a single error detected by multiple parties can also cause multiple broadcast signals to be generated.

In general, there is considerable complexity in programming application processes to send matching group requests and consistently handle broadcasted signals. In order to successfully send matching group requests for recovery, fault-free processes need to agree on the state of the application based on the broadcast signals they receive. If there is no guarantee on the order of delivery of broadcast signals, then fault-free processes must

implement a consensus protocol to agree on broadcast signals delivered before invoking any other fault tolerance mechanism. Furthermore, for a given error, application processes do not know *a priori* how many broadcast signals will be delivered. Each handler of an error notification must be programmed to expect and handle an unknown number of additional signals that may or may not be delivered.

To avoid requiring applications to implement a consensus protocol just to handle delivery of broadcast signals, we specify Ghidrah broadcast signals to be atomic. Implementing atomic broadcast signals is easily accomplished by using the centralized cluster manager (see Section 4.1.1). When an application process sends a Ghidrah broadcast signal, it actually sends a *broadcast request* to the cluster manager. The broadcast request contains the application process's chosen signal number and argument and the constant GSIG_BROADCAST as the destination. The cluster manager sends the broadcast request to all application processes as a broadcast signal. All broadcast signals are sent by the cluster manager, and all point-to-point communication channels between the cluster manager and application processes are reliable. Therefore all processes receive the same broadcast signals in the same order.

As mentioned above, even with atomic broadcast of signals, there is still complexity in handling multiple notifications from the cluster manager and from other application processes that can arise from a single error. This complexity is due, in part, to the fact that the application processes do not know *a priori* how many notifications will be delivered for the error. The rest of this section explains a simple protocol we used to managed this complexity.

We divide the execution time of the cluster manager and the entire application logically into numbered epochs. The cluster manager enforces a general rule that in a single epoch, the

application is involved in handling only one broadcast signal or one group request with reply. An application process can have at most a single outstanding request at a time. Once a process has sent a request, it must wait for a broadcast signal indicating that either the service was performed or its request was dropped.

Epochs are implemented for each application and the cluster manager as follows. The cluster manager and each application process maintains a copy of the current epoch number. Application processes attach their current epoch number to broadcast requests and group requests. Whenever the cluster manager broadcasts a Ghidrah signal to the application, it first increments its epoch number and attaches this new epoch number to the broadcast signal. Upon delivery of the broadcast signal, the application process uses the epoch number attached to the signal to update its copy of the current epoch number.

Enforcing the restriction that an application will handle only one event per epoch simplifies the application code, but the cluster manager must still be able to handle multiple events occurring per epoch. There are two options for dealing with multiple events occurring within the same epoch: (1) The cluster manager can order the events and maintain a queue of events that have not been presented to the application or (2) the cluster manager can choose an event to handle and ignore the others, forcing the entities that created the events that were ignored to create them again in a later epoch. We chose to implement the second option in order to minimize the complexity and amount of state maintained by the cluster manager. The second option also simplifies the application code for the case that a single error generates multiple detection and notification events.

The cluster manager's rules for choosing which events to handle in an epoch are simple: (1) If the cluster manager and an application process both generate an asynchronous

communication in the same epoch—whether broadcast signal or group request—the cluster manager's communication takes priority and the application process's communication is dropped. (2) If multiple application processes generate multiple broadcast signal requests and group requests in a single epoch, the cluster manager chooses one to handle, and the rest are dropped.

To implement the above two rules, the cluster manager simply examines the epoch number attached to application processes' requests. The cluster manager drops any request whose epoch number does not match the cluster manager's current epoch number. The cluster manager accepts the first request it receives whose epoch number matches the cluster manager's current epoch, and it then increments its epoch number. Henceforth, any request sent from the same epoch as the accepted request is dropped because its epoch number no longer matches the cluster manager's current epoch number.

Figure 4.3 illustrates an example of a group request being dropped due to the cluster manager advancing to the next epoch. In the figure, the cluster manager broadcasts signal a containing epoch number i. Then in epoch i the cluster manager broadcasts signal b, causing it to advance to epoch $i+1$. The application processes also send a group request for service u during epoch i. When the cluster manager enters epoch $i+1$, it drops the group request received from process x because that group request has not been matched in a quorum of redundant group requests. In epoch $i+1$ the cluster manager drops the group requests from processes y and z because their epoch numbers do not equal $i+1$. Finally, in epoch $i+1$, processes x, y and z send a new group request for service v which is accepted by the cluster manager.

The solution of dropping requests of application processes shifts the responsibility of

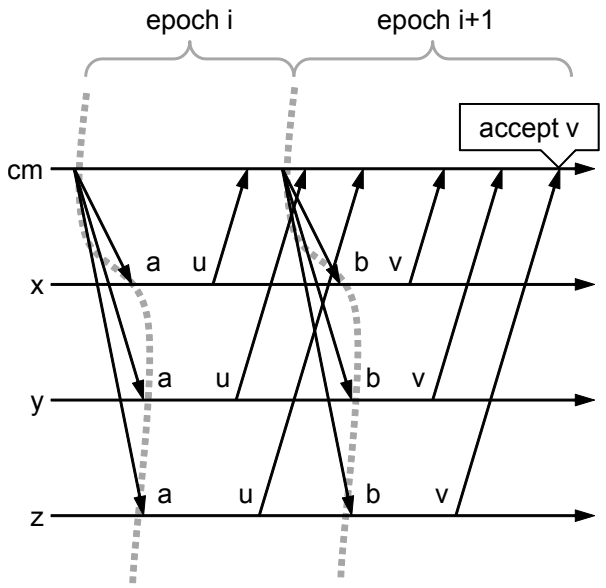


Figure 4.3: Two events in one epoch: Cluster manager's broadcast signal causes application's group request to be dropped.

queueing of multiple events from the cluster manager to the application. However, with careful implementation of application-specific fault tolerance mechanisms, queueing and retrying of broadcasts and requests may be unnecessary.

As an example of the claim made in the previous paragraph, consider the scenario illustrated in Figure 4.4, where two fault-free processes, *x* and *y*, detect an error and each request broadcast signals in the same epoch. The cluster manager broadcasts process *x*'s signal and drops process *y*'s broadcast request. Once all the application processes receive process *x*'s notification of an error detection, they all invoke an application-specific diagnosis procedure to determine what the error is and what to do to recover. During this diagnosis procedure, all fault-free processes, including processes *x* and *y*, must come to an agreement on the state of the application. That means that processes must agree that either *x* and *y* detected the same error, or there were two errors detected. Regardless of the final

determination, process *y*'s dropped broadcast request does not need to be broadcasted because its error detection was communicated to the other processes in the diagnosis procedure. (If there is a situation where process *y* needs its broadcast request fulfilled, it can generate a new broadcast request in a subsequent epoch.)

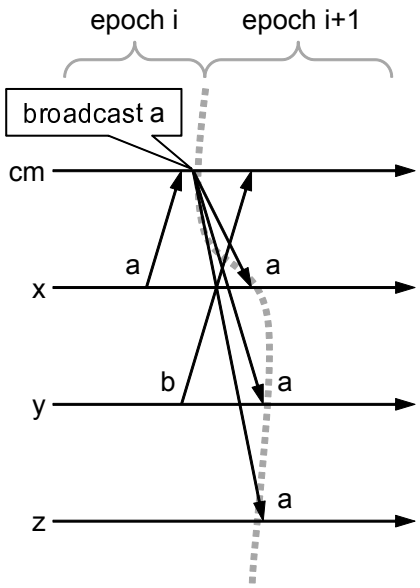


Figure 4.4: Two broadcast requests in one epoch. The cluster manager broadcasts the first broadcast request and drops the second broadcast request.

There is one situation that is not covered by the simplifying rules discussed above: the case when the cluster manager has multiple asynchronous events that it must notify the application of. This case does not occur when there are multiple node failures or process crashes, for the cluster manager already serializes these error detections and broadcasts a single notification for each detection. This case also does not occur with multiple group requests pending for an application, for the application is allowed to have a maximum of one request for service at a time. This case occurs only when the cluster manager is in the process of executing a service requested by the application when it detects an error that affects the

requesting application, such as a process crash or a node failure.

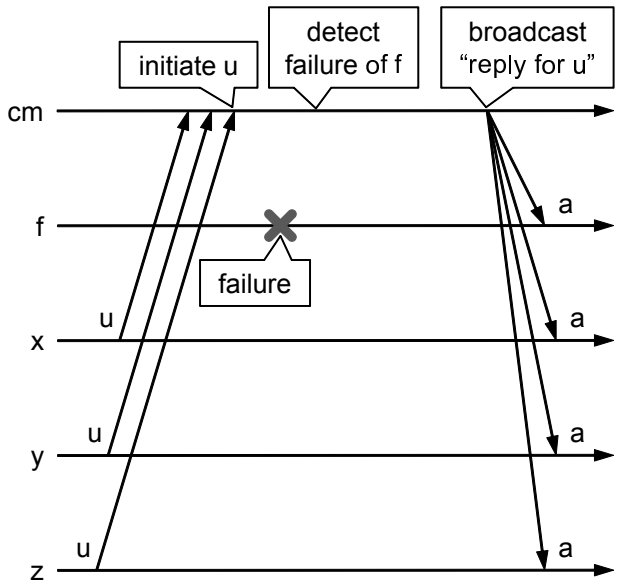


Figure 4.5: Two events in one epoch: Error detected by cluster manager during execution of service. Application processes x, y and z send group request for service u to cluster manager cm. Before cm replies with broadcast a, it detects a failure of application process f.

Figure 4.5 illustrates the case of a CMM error detection occurring while the cluster manager is executing a service on behalf of the application. There are two options for handling the two events. In the first option, the cluster manager serializes the events by postponing the error notification until after the service is complete and the cluster manager has broadcasted the service reply. (The cluster manager cannot serialize the events in the opposite order because we assume that services cannot be undone once they are initiated. This is certainly true for a service that terminates an application process.) The second option is to immediately notify the application of the error detection.

We chose to implement the second option described in the previous paragraph because it

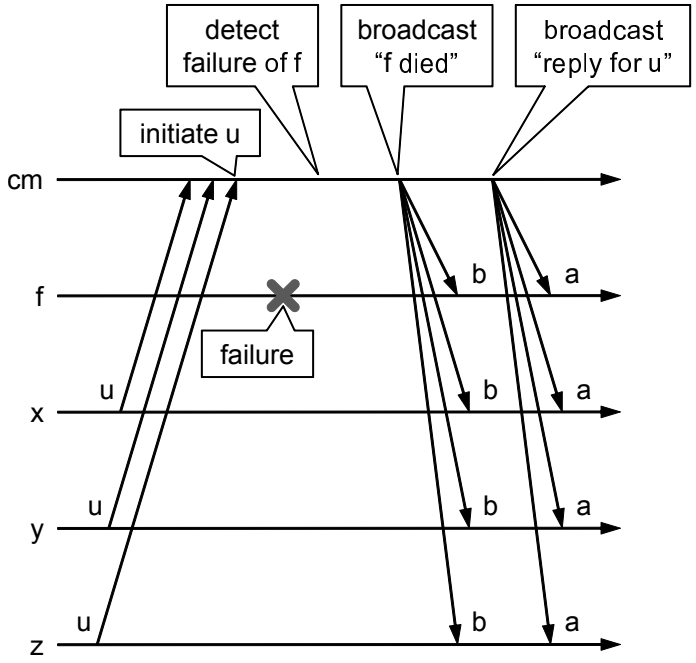


Figure 4.6: Handling two events in one epoch: Cluster manager broadcasts error notification during execution of service. The error notification in broadcast b is interpreted by the application processes as the dropping of its group request for service u.

frees the cluster manager from maintaining a queue of events to notify the application of. However, as shown in Figure 4.6, broadcasting the error notification presents an ambiguity to the application processes. Because the application processes have not received a reply regarding their group request, they will interpret the delivery of the error notification as meaning that the cluster manager has dropped their request. They will not be expecting the delivery of the cluster manager's reply when the service is complete.

To solve the ambiguity of receiving an error notification after the application's service has been initiated, the cluster manager precedes its error notification broadcast with a special acknowledgment broadcast indicating that the requested service has been initiated but not completed. Upon receiving this broadcast signal, application processes know to expect one or

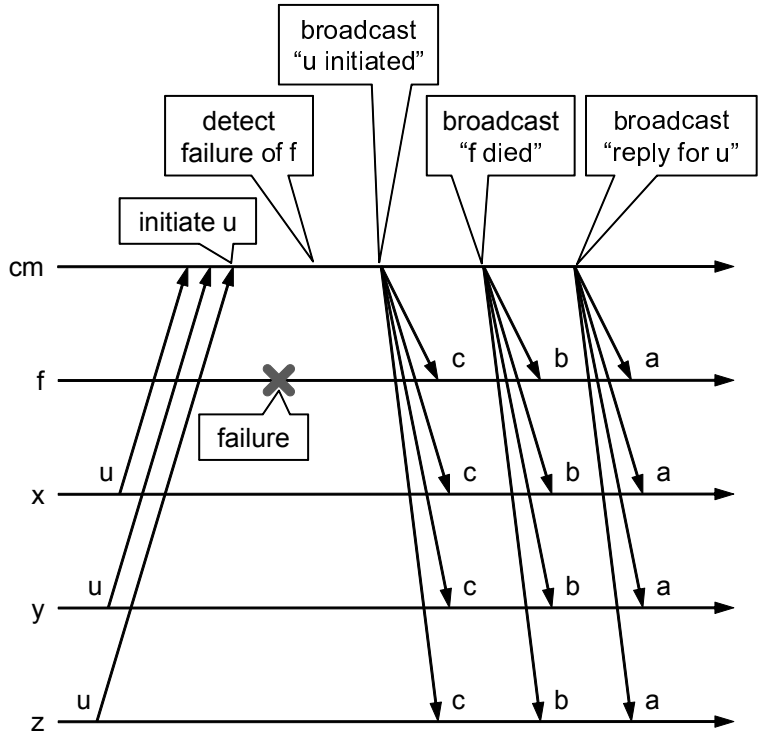


Figure 4.7: Handling two events in one epoch: Cluster manager precedes broadcast of error notification with an acknowledgment of the application's group request. The acknowledgment *c* indicates to the application that its group request for service *u* has been initiated but not completed and that an error notification will follow the acknowledgment.

more error notifications before the cluster manager's service reply is delivered. They also know not to send another group request until the reply is delivered. This solution is illustrated in Figure 4.7.

We assume that this special case of errors detected by the cluster manager during the time it is executing a service is very rare. Thus, the acknowledgment broadcast is only sent when the cluster manager has an error notification to send while a service is pending. If no errors are detected during execution of a service, the only broadcast signal that the cluster manager sends is a reply broadcast indicating that the service is complete. This minimizes the

number of broadcast signals the cluster manager must send in relation to a request for service. Furthermore, only two of the CMM services we implement are not executed by the cluster manager atomically: terminating an application process and spawning an application process. All of the other group requests are executed atomically and therefore cannot be interrupted by an error detection.

To summarize the rules of which events get handled in an epoch from the point of view of an application process, we describe the possible responses an application process should expect to receive after sending a request to the cluster manager. For broadcast requests, the application process expects to receive either its own broadcast signal or a different broadcast signal from another application process or from the cluster manager, indicating that its broadcast request was dropped.

For group requests, the application process expects to receive one of four broadcast signals: (1) G_SIG_DISAGREE, indicating that no quorum of agreeing processes exists for the current group request, (2) a reply broadcast indicating that the cluster manager accepted and performed the service that the application process requested (see Subsection 4.1.2), (3) a reply broadcast indicating that the cluster manager accepted and performed a service different from the application process's request, meaning that this application process was not part of the quorum, (4) a reply broadcast indicating that the cluster manager initiated but did not complete the service that the application process requested, or (5) a reply broadcast indicating that the cluster manager initiated but did not complete a service that the application process did not request, meaning that this application process was not part of the quorum.

There is a critical problem that the protocol described so far in this subsection does not solve. This is the problem that, as mentioned in the previous subsection, a broadcast request

from one of the application's processes could, potentially, interrupt the collection of a quorum of matching group requests. In this scenario, a single Byzantine-faulty application process can prevent a quorum of fault-free processes from making progress on a group request simply by repeatedly sending broadcast requests. In fact, this problem exists in our implementation. However, we believe that the small modification described below resolves this bug, assuming known bounds on both message delays and differences in the relative speeds of different processes.

Our solution to the problem described above is to add a rule to the cluster manager's handling of broadcast requests: If the cluster manager receives a broadcast request during an epoch in which it has already received one or more group requests, it drops the broadcast request and counts it as a group request from that process that does not match any other group request in the current epoch. This modification could, of course, allow a faulty process to block broadcast requests from fault-free processes by simply initiating a group request at the beginning of every epoch. This latter problem is resolved based on the fact that it results in a failure to reach a quorum regarding the faulty group request, leading the cluster manager to broadcast a GSIG_ERR_DISAGREE signal. A process that does not have a pending group request but receives a GSIG_ERR_DISAGREE signal, enters the application's self-diagnosis procedure. This rule eventually causes all the fault-free processes to enter self-diagnosis. The self-diagnosis procedure identifies the faulty process (the one that sent the original faulty group request) and leads to it being killed by the cluster manager. It should be noted that we have not formally proved the correctness of this mechanism and have not validated it experimentally.

4.2. G-MPI: Extending MPI Semantics for Application Fault Tolerance

While the basic API presented in Section 4.1.1 is useful in constructing fault tolerance mechanisms for distributed applications, a higher level of abstraction is helpful for developing fault-tolerant message-passing applications. Developing fault-tolerant programs involves writing code that detects, diagnoses, and recovers from errors. The recovery block [Rand75] is a basic software structure for error detecting and recovering code. Fault-tolerant applications use this structure by dividing normal application code into functional blocks which may include nested blocks. Each block is terminated by an acceptance test on the block's output. If the acceptance test fails, an error has been detected, and control is transferred to a recovery block to repair the error. When blocks execute in parallel and interact through message-passing, coordination among blocks is required so that either all of the blocks pass their acceptance tests or all of the blocks fail their acceptance tests. This coordination prevents errors from propagating from blocks that failed their acceptance tests to blocks that passed their acceptance tests.

In addition to parallel blocks communicating to coordinate their acceptance tests, distributed applications can make use of asynchronous notifications, or exceptions, to implement error recovery [Camp86]. Exception handling enables the safe interruption of concurrently executing and communicating blocks, regardless of their nesting level, so that they can coordinate for error recovery.

The higher-level abstractions described above show that implementation of fault tolerance mechanisms can be benefited by an API that combines asynchronous notifications and message-passing. This section describes our design for G-MPI, a message-passing API based on MPI that combines these two facilities.

G-MPI solves two problems with the standard MPI API that obstruct the development of fault-tolerant message-passing applications. First, MPI does not specify the state of the message-passing layer after an error has occurred. This lack of specification practically means that processes cannot reliably communicate via message-passing in an error condition. Second, MPI does not permit the membership of the set of processes in an application to be modified. Should the application programmer decide to use the new CMM's services to terminate faulty processes and spawn new processes, the state of the MPI communication context must be manipulated in a way that the MPI API does not allow. G-MPI implements additional API and slightly different semantics to solve these two problems.

4.2.1. FT-MPI: Extending MPI Semantics for Recovery from Process Crashes

Our approach to extending the MPI API builds upon the approach of FT-MPI [Fagg05], briefly introduced in Subsection 2.6.1. FT-MPI extends MPI semantics so that application processes can continue to communicate after process crashes and can modify the membership of the set of processes in response to process crashes. Hence, FT-MPI solves the two problems mentioned above for process crashes. Our implementation of G-MPI shares the same extensions to MPI semantics that FT-MPI defines. This subsection describes how FT-MPI notifies an application process of a crash, how the application specifies how to recover MPI state after the crash so that message-passing can continue, and how FT-MPI implements recovery of MPI state.

When a process crashes, the state of the MPI implementation layer, namely the MPI communication context represented by MPI communicators, becomes invalid. FT-MPI semantics permit the application to continue executing, but all MPI communication calls

cease to function, only returning the MPI error code MPI_ERR_OTHER. If an application process is executing an MPI call that is blocked at the time that the MPI communication context becomes invalid, FT-MPI specifies that the call becomes unblocked and returns MPI_ERR_OTHER. Without this unblocking behavior, the blocked call may never complete and return (see Section 3.3.1), preventing the process from executing recovery actions so that normal computation can resume.

An FT-MPI application can recover a valid communication context by having all surviving processes invoke the FT-MPI recovery function, `MPI_Comm_dup()`, with a special parameter of FT_MPI_CHECK_RECOVER. Similar to `MPI_Init()`, the FT-MPI recovery function constructs a new MPI communication context and returns a single valid MPI world communicator. All other MPI communicators that the application previously created are destroyed.

The FT-MPI recovery function is a collective call; it does not return with the new MPI communication context until all the surviving processes have called it and all the newly spawned processes have called `MPI_Init()`. For newly spawned processes, `MPI_Init()` returns a special return code—MPI_RECOVERED instead of the usual MPI_SUCCESS—to indicate that the process has joined an existing application in recovery. After the calls to `MPI_Init()` and the FT-MPI recovery function return, the application resumes normal execution. It is the application's responsibility to restore application-level state to newly spawned processes.

To direct how the FT-MPI recovery function constructs the new MPI communication context, the application specifies a *communicator mode* and a *message mode*. The *communicator mode* specifies whether the crashed processes should be replaced by newly

spawned processes and how MPI ranks should be assigned to the surviving and new processes in the new communication context. The *message mode* specifies whether messages that were in-transit between surviving processes at the time of the crash should be delivered or dropped.

The FT-MPI recovery function performs a recovery algorithm that guarantees that all processes have the same membership list for constructing the new communication context, even when there are additional process crashes. The algorithm designates one application process as the *leader* process. The *leader* communicates with all other application processes—*peons*—to determine the state of a new MPI communication context. The leader sends messages to all processes to query their survival. For an MPI application originally started with n processes the leader expects to receive a total of $r + m = n - 1$ messages (not including itself), where r is the number of surviving processes and m is the number of crash notifications from the CMM. Once it receives these messages, it requests of the CMM process spawns to replace crashed processes. When the CMM replies with acknowledgment that the processes have been spawned, the leader constructs the new MPI communication context and proposes it to the peons. The peons agree to the proposal using a multi-phase commit protocol. If at any time another process crashes, the leader restarts the entire recovery algorithm. If the leader process crashes, peons elect a new leader using an atomic test-and-set primitive that is implemented as a centralized service of the CMM. The first peon to set the value becomes the new leader.

The implementation of FT-MPI's recovery function only tolerates process crashes. Our G-MPI API also provides a recovery function, `G_MPI_Recover_comm()`, but its implementation can tolerate a Byzantine-faulty application process, as the next subsection will explain.

4.2.2. G-MPI: Extending MPI Semantics for Application Fault Tolerance

While FT-MPI enables MPI applications to recover from system-detected process crashes, it does not enable them to detect and recover from application-detected errors such as hung processes and corruption of application-level state. Furthermore, its recovery function does not tolerate a Byzantine-faulty process. G-MPI enables implementation of application fault tolerance for MPI applications by further extending the semantics and API of FT-MPI. This subsection describes the two additional facilities provided by G-MPI for applications to use: the G-MPI alert flag and G-MPI signals. The subsection ends with a description of G-MPI's recovery function, `G_MPI_Recover_comm()`, and its implementation.

In G-MPI, as in FT-MPI, an application process's MPI communication context becomes invalid when a process of the application crashes. When the MPI communication context is invalid, all MPI communication calls return the MPI_ERR_ALERT error code, and blocked MPI communication calls unblock to return the same error code. The G-MPI implementation registers an internal Ghidrah signal handler to receive process crash broadcast signals (GSIG_NTF_DIED in Table 4.1) from the CMM. This handler invalidates the application process's MPI communication context. The MPI communication context becomes valid again only upon successful exit from the G-MPI recovery function `G_MPI_Recover_comm()`.

Applications that detect application-specific errors also need a way to cause blocking MPI communication calls to unblock and return an error code. G-MPI provides a way for application processes to enable this behavior of MPI communication calls by adding a Boolean flag to the MPI state. Hence, in the G-MPI implementation, there are two components of MPI state in each application process: the MPI communication context and the new flag, called the *G-MPI alert flag*. When the G-MPI alert flag is raised, or set, all MPI

`G_MPI_Alert_raise()` : Sets the alert flag, causing all MPI calls to unblock and return with error code MPI_ERR_ALERT.

`G_MPI_Alert_clear()` : Clears the alert flag; all MPI calls execute normally.

`G_MPI_Alert_check()` : Checks state of the alert flag; returns MPI_SUCCESS if alert is not raised, or MPI_ERR_ALERT if alert is raised.

`G_MPI_Reform_comm(MPI_Comm comm, MPI_Comm *newcomm)` : Removes dead ranks and integrates new ranks into new MPI_COMM_WORLD. This function is analogous to FT-MPI's recovery function.

Figure 4.8: API for manipulating G-MPI state

`void handler(int signum, int srcrank, int destrank, int arg)` : The function prototype for GMPI signal handlers.
`srcrank` and `destrank` are MPI ranks in MPI_COMM_WORLD.
If `srcrank` and `destrank` are equal to the recipient's rank, then the signal is a local signal.
If `srcrank` is G_MPI_SIG_CLUSTER_MANAGER, then the sender is the cluster manager.
If `destrank` is G_MPI_SIG_BROADCAST, then the signal was broadcasted to all ranks of the task.

`void G_MPI_Signal(int signum, int destrank, int arg)` : Sends GMPI signal `signum` with argument `arg` to `destrank`.
If `destrank` is G_MPI_SIG_CLUSTER_MANAGER, then the signal is a group request for CMM service `signum` with argument `arg`
If `destrank` is G_MPI_SIG_BROADCAST, then the signal is broadcasted to all ranks.

`void G_MPI_Signal_handler(int signum, void (*handler)(int, int, int, int))` : Sets `handler` as signal handler for signal `signum`.

Figure 4.9: API for G-MPI signals

communication calls in that process return the MPI_ERR_ALERT error code, and blocked MPI communication calls unblock and return the same error code. When the G-MPI alert flag is clear, all MPI communication calls operate normally. Application processes manipulate and check the state of their G-MPI alert flag by using the G-MPI alert API specified in Figure 4.8. The three functions `G_MPI_Alert_raise()`, `G_MPI_Alert_clear()`, and `G_MPI_Alert_check()` are not MPI communication calls since they only change or read state that is local to the process. Hence, they will work even if the MPI communication context is invalid. The G-MPI alert flag is also implicitly cleared by

Signal	Type	Meaning
G_MPI_SIG_REQ_SETQUORUM	group request	request to change quorum size to *arg*
G_MPI_SIG_REQ_KILL	group request	request to kill process of MPI rank *arg*
G_MPI_SIG_REQ_SPAWN	group request	request to spawn and add a process
G_MPI_SIG_REQ_SYNC	group request	request to synchronize on value *arg*
G_MPI_SIG_NTF_DIED	broadcast	notification of crash of process of MPI rank *arg*, or reply for G_MPI_SIG_REQ_KILL: kill of MPI rank *arg* completed
G_MPI_SIG_REP_QUORUMSET	broadcast	reply for G_MPI_SIG_REQ_SETQUORUM: quorum size set to *arg*
G_MPI_SIG_REP_ACK_KILL	broadcast	acknowledgment of G_MPI_SIG_REQ_KILL: kill of MPI rank *arg* initiated
G_MPI_SIG_REP_ACK_SPAWN	broadcast	acknowledgment of G_MPI_SIG_REQ_SPAWN: spawn and add a process initiated
G_MPI_SIG_REP_SPAWNED	broadcast	reply for G_MPI_SIG_REQ_SPAWNED: spawn and add a process completed; meaning of *arg* is undefined
G_MPI_SIG_REP_SYNC	broadcast	reply for G_MPI_SIG_REQ_SYNC: synchronized on value *arg*
G_MPI_SIG_ERR_DISAGREE	broadcast	group request failed due to no quorum

Table 4.2: Pre-defined G-MPI signals

`G_MPI_Recover_comm()`.

Section 3.3 explains applications' need for asynchronous inter-process communication for error notification and also for interaction with the cluster manager. G-MPI provides a signaling facility called *G-MPI signals* that is analogous to Ghidrah signals defined in Subsection 4.1.1. The G-MPI signal API is defined in Figure 4.9, and the pre-defined G-MPI signal numbers are defined in Table 4.2. The only salient differences between the Ghidrah signal API and the G-MPI signal API are that the G-MPI signal API refers to processes by their MPI rank rather than by process identifiers assigned by the CMM and that the reply for the spawn request, G_MPI_SIG_REP_SPAWNED does not specify an argument. This is

because the newly spawned process is not assigned an MPI rank and included in the MPI communication context until the successful return from `G_MPI_Recover_comm()`.

As is the case with Ghidrah signal handlers, G-MPI signal handlers are invoked in a signal handling context, interrupting the main code of the application. Hence, G-MPI signal handlers execute atomically and must not block. This means that MPI communication operations cannot be safely invoked from within a G-MPI signal handler.

The invocation of G-MPI signal handlers can interrupt MPI communication calls, including blocked MPI communication calls. However, G-MPI signals do not cause blocked MPI communication calls to unblock prematurely. This is to allow application processes to handle the delivery of G-MPI signals without affecting the execution of the application's main code.

Application processes can use the G-MPI alert flag in combination with G-MPI signals to interrupt and notify peer processes of application-detected errors as follows. All processes of the application register a G-MPI signal handler for a specific application-defined G-MPI signal number. The signal handler calls `G_MPI_Alert_raise()` to raise the process's G-MPI alert flag. When an application process detects an error such as an incorrect value or a lack of timely progress, it calls `G_MPI_Signal()` to broadcast the application-defined G-MPI signal number to all processes of the application. Upon delivery of that G-MPI signal, each process raises its own G-MPI alert flag. Eventually, every fault-free application process observes the MPI_ERR_ALERT error code returned from an MPI communication call, and they can then proceed to coordinate for performing application-specific diagnosis, recovery, and reconfiguration.

`void G_MPI_set_wall_timer(long time, int arg, void **handle)` : Set a timer event to fire `time` microseconds from now; when fired, the local G-MPI signal G_MPI_SIG_LOCAL_ALARM is delivered, with argument of `arg`. The pointer `*handle` is assigned to an opaque object to be used to dismiss the timer event.

`void G_MPI_set_task_timer(long time, int arg, void **handle)` : Set a timer event to fire `time` microseconds from now in ‘‘task time’’; when fired, the local G-MPI signal G_MPI_SIG_LOCAL_ALARM is delivered, with argument of `arg`. ‘‘Task time’’ stops when the task is stopped by the cluster manager (i.e. not schedule to run in the cluster). The pointer `*handle` is assigned to an opaque object to be used to dismiss the timer event.

`void G_MPI_dismiss_timer(void *handle)` : Dismisses timer event referenced by `handle`.

Figure 4.10: API for G-MPI timer events

`void G_MPI_Enter_cs()` : Enter a critical section of code in which delivery of G-MPI signals is suspended. Arriving G-MPI signals are queued for delivery upon exiting the critical section.

`void G_MPI_Exit_cs()` : Exit a critical section of code. Any G-MPI signals that were queued during the critical section are delivered before this function returns.

`void G_MPI_Signal_wait()` : Block until delivery of a G-MPI signal.

Figure 4.11: API for managing G-MPI signal delivery

G-MPI signals are implemented using Ghidrah signals. Hence, G-MPI applications must not invoke the Ghidrah signal API directly. The G-MPI implementation registers its own Ghidrah signal handlers which translate the process identifiers of Ghidrah signals to MPI rank numbers. These Ghidrah signal handlers then invoke the application process’s G-MPI signal handlers that were registered when the process called `G_MPI_Signal_handler()`.

Since both the Ghidrah application library and the G-MPI library use POSIX signals (for delivery of Ghidrah signals and for timeout events in the reliable point-to-point message-passing protocol, respectively), it is not safe to permit the application programmer to manage POSIX signals directly. Therefore G-MPI provides additional API for managing application-level timer events and for protecting critical sections of application code from being interrupted by G-MPI signals. These functions only affect the state of the process in which

they are invoked. The API for managing timer events is defined in Figure 4.10, and the API for managing delivery of G-MPI signals is defined in Figure 4.11.

Signal	Type	Usage
G_MPI_SIG_LOCAL_CONNECTION	local	broken point-to-point communication connection involving MPI rank *arg*
G_MPI_SIG_LOCAL_ALARM	local	timer event expiration; *arg* is the argument specified in `G_MPI_set_wall_timer()` or `G_MPI_set_task_timer()`

Table 4.3: Pre-defined local G-MPI signals

In addition to the pre-defined G-MPI signals that correspond to Ghidrah signals, G-MPI defines two additional signals that are *local signals*, signals that only interrupt the process that they were generated by. These are listed in Table 4.3. Application processes can handle these G-MPI signals by registering a handler via `G_MPI_Signal_handler()`. The first local G-MPI signal, G_MPI_SIG_LOCAL_ALARM, is delivered upon expiration of a local timer event that is scheduled via `G_MPI_set_wall_timer()` or `G_MPI_set_task_timer()`. The second local G-MPI signal, G_MPI_SIG_LOCAL_CONNECTION, is delivered to an application process whenever a broken connection is detected by the process's reliable point-to-point communication protocol used to transmit MPI messages. This signal can be used by application processes to detect hung processes.

The `G_MPI_Recover_comm()` function recovers a valid MPI communication context after process crashes and spawns. Like FT-MPI's recovery function, upon successful return `G_MPI_Recover_comm()` provides a new global MPI communication context (i.e. MPI world communicator), and all previously constructed communicators are destroyed. Also like

FT-MPI, `G_MPI_Recover_comm()` is a collective operation: it does not return until all newly spawned processes call `MPI_Init()` and all other processes call `G_MPI_Recover_comm()`. The `G_MPI_Recover_comm()` specification differs from that of FT-MPI's recovery function in one important aspect: the recovery of a valid MPI communication context may fail. If this happens, `G_MPI_Recover_comm()` will return MPI_ERR_ALERT to all processes that called it, and `MPI_Init()` will return MPI_ERR_ALERT to all processes that called it. The reason for this is to allow the application to handle errors that occur while processes are agreeing on the new MPI communication context.

To enable a simple implementation of the `G_MPI_Recover_comm()` function, the G-MPI implementation layer in each application process records changes to the process membership of the application. This is done consistently by all fault-free processes, since the cluster manager reliably notifies all the processes of the application of process addition and removal using the broadcast signals GSIG_REP_SPAWNED and GSIG_NTF_DIED (see Table 4.1), respectively.

The `G_MPI_Recover_comm()` effectively performs a barrier synchronization operation. Upon return from this function, the fault-free processes must have consistent communicators. This is achieved using Ghidrah's synchronization service, GSIG_REQ_SYNC. As part of `G_MPI_Recover_comm()`, at least a quorum of fault-free processes must send a GSIG_REQ_SYNC request. The argument for this request can be any value. *All* fault-free processes must wait for the corresponding GSIG_REP_SYNC reply using `G_MPI_Signal_wait()` (Figure 4.11). Since the GSIG_REP_SYNC signal is ordered with respect to all other notifications from the cluster manager, upon receipt of this

signal, all fault-free processes have consistent membership lists.

As mentioned earlier, `G_MPI_Recover_comm()` may return MPI_ERR_ALERT, indicating failure. This can happen if, for example, the process receives a crash notification while waiting for the GSIG_REP_SYNC reply.

Using the mechanisms described above, the actions of each process in response to an error depends on the type of error that occurs: A) a process crash, detected by the manager, versus B) a process hang or a process that generates incorrect outputs. The signal handler that eventually causes MPI calls in the process to return MPI_ERR_ALERT records information that the process can use later to determine the type of error that occurred. When a process determines that there is a Type A error, it simply calls `G_MPI_Recover_comm()`, which, in turn, sends a GSIG_REQ_SYNC request, waits for a GSIG_REP_SYNC reply, and then returns with a new MPI world communicator.

When a process determines that there is *potentially* a Type B error, it must first initiate distributed diagnosis. The assumption is that at least a quorum of processes identify the faulty processes and initiate a sequence of requests to kill those processes. After each GSIG_REQ_KILL request, the process calls `G_MPI_Signal_wait()` to wait for a reply. After a process sends GSIG_REQ_KILL requests for all the processes its diagnosis results indicate that it should kill, the process calls `G_MPI_Recover_comm()`. As described above, `G_MPI_Recover_comm()` sends a GSIG_REQ_SYNC request, waits for a GSIG_REP_SYNC reply, and then returns with a new MPI world communicator.

If recovery requires spawning a new process, this is also done before calling `G_MPI_Recover_comm()`. Since GSIG_REP_ACK_SPAWNED broadcast signals are

ordered with respect to GSIG_REP_SYNC signals, all processes have consistent communicator states when `G_MPI_Recover_comm()` returns.

4.3. An Example G-MPI Fault-Tolerant Application

This section presents example G-MPI application code to illustrate the usage of G-MPI's modifications to the MPI API and semantics. The example shows how a G-MPI application detects and recovers from a single process hang or crash. It also handles a process that hangs and then crashes some undefined time later. The routines presented here can be extended to handle multiple faulty processes, provided that there are enough fault-free processes available to diagnose the faulty ones.

MPI applications can be transformed into fault-tolerant G-MPI applications by implementing two types of changes, which we term ‘‘synchronous’’ and ‘‘asynchronous’’. The ‘‘synchronous’’ section of the changes are made inline with the existing application code and consists of checking MPI functions' return codes for the MPI_ERR_ALERT error code and an application-specific function to handle error diagnosis, recovery, and reconfiguration. The ‘‘asynchronous’’ section consists of the G-MPI signal handlers implemented and registered by the application. As we will see in the example code, the asynchronous section responds to G-MPI signals by changing the state of a small set of global variables in the application process and raising the G-MPI alert flag. The synchronous section can then determine that G-MPI signals were delivered by observing the MPI_ERR_ALERT error code returned by MPI functions, and it can determine the contents of the signals by reading the global variables.

```
/* cause of G-MPI alert */
const int NONE = 0;
const int CRASH = 1;
const int HANG = 2;

volatile int rank_crashed = -1;
volatile int alert_cause = NONE;

void handle_crash(int signum, int srcrank, int arg)
{
  alert_cause = CRASH;
  rank_crashed = arg;
  G_MPI_Alert_raise();
}
```

Figure 4.12: Application's handler for G-MPI signal G_MPI_SIG_NTF_DIED

4.3.1. Detecting a Process Crash

Because the OS and CMM perform the work of detecting a process crash and notifying the surviving application processes, the code for handling the notification is very simple. The application implements a handler for the G-MPI signal G_MPI_SIG_NTF_DIED as shown in Figure 4.12. The handler simply records the cause for raising the G-MPI alert flag and the MPI rank of the crashed process in the global variables `alert_cause` and `rank_crashed`, respectively. It then raises the G-MPI alert flag. This change is in the asynchronous section of the application code. The only change in the synchronous section is to register this G-MPI signal handler.

4.3.2. Detecting a Process Hang

Detection of process hangs is handled by the synchronous section of the application code by timing the execution of application procedures, as illustrated in the application function

```
int do_computation(...)
{
  int rc;

  /* Set timeout expiration to number of microsec from now: */
  long int timeout = ...;

  /* Pointer to opaque timer event object: */
  void *timer;
  G_MPI_set_wall_timer( timeout, 0, &timer );

  /* perform computation and communication */
  rc = G_MPI_Alert_check();
  if ( rc == MPI_ERR_ALERT ) {
    G_MPI_dismiss_timer( timer );
    diagnose_recover_configure();
  }

  G_MPI_dismiss_timer( timer );
  return ...;
}
```

Figure 4.13: Application's code for timing a computation

```
const int APP_SIG_HANG = 100;

/* G-MPI signal handler for G_MPI_SIG_LOCAL_ALARM: */
void handle_alarm(int signum, int srcrank, int arg)
{
  G_MPI_Signal(APP_SIG_HANG, G_MPI_SIG_BROADCAST, 0);
}

/* G-MPI signal handler for APP_SIG_HANG: */
void handle_hang_detection(int signum, int srcrank, int arg)
{
  alert_cause = HANG;
  G_MPI_Alert_raise();
}
```

Figure 4.14: Application's handler for G-MPI signal APP_SIG_HANG

`do_computation()` in Figure 4.13. When a process's timer event expires, the asynchronous section of the application is invoked. Specifically, the application's handler for G_MPI_SIG_LOCAL_ALARM, `handle_alarm()`, is called. The handler requests broadcast of an application-defined G-MPI signal, APP_SIG_HANG, as shown in Figure 4.14. Subsequently, all processes handle the APP_SIG_HANG broadcast signal in the same way: the application handler `handle_hang_detection()` records in the global variable `alert_cause` the cause for raising the G-MPI alert flag and raises the G-MPI alert flag.

4.3.3. Recovering from a Process Hang or Crash

The synchronous section of the application code is notified of an existing error by observing the MPI_ERR_ALERT error code returned by MPI functions. Alternatively, if the application code is executing a long-running local computation that does not invoke any MPI calls, it can periodically poll for an error by calling `G_MPI_Alert_check()`. This function returns MPI_ERR_ALERT if the G-MPI alert flag is raised or MPI_SUCCESS otherwise.

Once the application code observes the G_MPI_ERR_ALERT error code, it invokes an application-defined `diagnose_recover_reconfigure()` function. If application processes need to be executing application-specific recovery from a common point in the application code, it is the application process's responsibility to unwind the call stack before invoking `diagnose_recover_reconfigure()`.

The example application's implementation of `diagnose_recover_reconfigure()` is shown in Figure 4.15. Some supporting functions called by this function are shown in Figure 4.16. The

```
MPI_Comm comm; /* MPI world communicator */
const int VOTE_TO_CONTINUE = 1;
void diagnose_recover_reconfigure()
{
  int hungrank, rc;
  do {
    if (alert_cause == CRASH) {
      request_spawn();
    } else if (alert_cause == HANG) {
      G_MPI_Alert_clear();
      hungrank = diagnose_hung_process();
      if ( G_MPI_Alert_check() == MPI_ERR_ALERT ) {
        /* G-MPI alert flag was raised during diagnose_hung_process() */
        if ( alert_cause == CRASH ) {
          request_spawn();
        }
      } else {
        if ( hungrank != -1 ) {
          request_kill(hungrank);
          request_spawn();
        } else {
          /* All processes are responsive */
          synchronized = 0;
          G_MPI_Signal(G_MPI_SIG_REQ_SYNC, G_MPI_SIG_CM, VOTE_TO_CONTINUE);
          if ( synchronized != 1 ) {
            /* Quorum of processes failed to agree to continue. */
            app_fail();
          }
        }
      }
    }
    alert_cause = NONE;
    G_MPI_Recover_comm(&comm); /* Recover MPI communication context. */
    recover_app_state(); /* Recover application-level state. */
    /* If another process fails during the calls to G_MPI_Recover_comm()
    or recover_app_state(), then handle the new alert: */
  } while (G_MPI_Alert_check() != MPI_SUCCESS);
}
```

Figure 4.15: Application's synchronous function for diagnosis, recovery and reconfiguration

`diagnose_recover_reconfigure()` function reads the global variables set by the asynchronous section of the code to determine how to proceed.

If a process crash was detected, application processes simply request a process spawn. If a process hang was detected, application processes execute a diagnosis procedure to determine which process is unresponsive.

The diagnosis procedure, called `diagnose_hung_process()` may be implemented by having all processes send and reply to short ‘‘ping’’ messages via MPI. The function returns the MPI rank number of the hung process or -1 if all processes replied. If a process is unresponsive, `diagnose_recover_reconfigure()` requests it to be terminated.

Figure 4.15 shows that the G-MPI alert flag is cleared immediately before invoking `diagnose_hung_process()` so that MPI communication can proceed. Note that the MPI communicators are still valid because no process has crashed. However, if a process subsequently crashes during execution of `diagnose_hung_process()`, then the application’s G-MPI signal handler `handle_crash()` in Figure 4.12 will raise the G-MPI alert flag again, causing the `diagnose_hung_process()` to receive the G_MPI_ERR_ALERT error code and to return without completing the diagnosis. Then the `diagnose_recover_reconfigure()` function handles a process crash instead of an unresponsive process.

As the cluster manager completes the process termination and spawn requests, it notifies application processes by broadcasting G-MPI signal replies (see Table 4.2). For process termination, the reply is the same as the notification for a process crash, so the same `handle_crash()` handler is invoked. For process spawn, the application handles the

```
void request_spawn()
{
  G_MPI_Signal(G_MPI_SIG_REQ_SPAWN, G_MPI_SIG_CM, 0);
  G_MPI_Signal_wait();
  if ( spawned != 1 ) {
    /* Delivered signal was not the spawn reply. */
    /* There is some additional error that I cannot handle. */
    app_fail();
  }
}

void request_kill(int rank)
{
  G_MPI_Signal(G_MPI_SIG_REQ_KILL, G_MPI_SIG_CM, rank);
  G_MPI_Signal_wait();
  if ( rank_crashed != rank ) {
    /* Delivered signal was not the kill reply. */
    app_fail();
  }
}

void app_fail()
{
  MPI_Abort(comm, 1);
}
```

Figure 4.16: Supporting functions invoked by `diagnose_recover_reconfigure()`

```
volatile int spawned = 0;

/* Handler for G-MPI signal G_MPI_SIG_REP_SPAWNED: */
void handle_spawn(int signum, int srcrank, int arg)
{
  spawned = 1;
}
```

Figure 4.17: Application's handler for G-MPI signal G_MPI_SIG_REP_SPAWNED

G_MPI_SIG_REP_SPAWN reply using the `handle_spawn()` handler shown in Figure 4.17.

After requests to terminate and spawn processes have been completed, the `diagnose_recover_reconfigure()` function invokes the G-MPI recovery function `G_MPI_Recover_comm()`. Upon successful construction of the MPI communication context, the `diagnose_recover_reconfigure()` function calls the application-defined `recover_app_state()` function to restore error-free application-level state. (The implementation of this function is application-specific, so it is not shown here.)

Finally, Figure 4.18 shows the example application's `main()` function, with the familiar `MPI_Initialize()` function's return code being tested for the MPI_RECOVERED return code. Note that when a newly spawned process observes the MPI_RECOVERED return code, it calls the `recover_app_state()` to restore application-level state that corresponds to the recovered state of the surviving processes. Processes can communicate via MPI in `recover_app_state()` so that the newly spawned process can learn the application's current state from surviving processes. Once application-level state has been restored, normal execution resumes.

Although we do not describe any example code for detecting and handling other types of application-detected errors (e.g. out-of-range values), the framework described in this section gives the application programmer freedom to implement arbitrarily complex application-specific detection, diagnosis, and recovery algorithms. For responding to an application-detected error that is not a process crash or hang, an approach similar to the handling of process hangs can be used. First, the application defines a new G-MPI signal number to use for broadcasting the detection of this type of error. The application also defines a new value

```
int main(...)
{
  int rc;

  /* Register all G-MPI signal handlers: */
  G_MPI_Signal_handler(G_MPI_SIG_NTF_DIED, handle_crash);
  G_MPI_Signal_handler(G_MPI_SIG_REP_SPAWNED, handle_spawn);
  G_MPI_Signal_handler(G_MPI_SIG_LOCAL_ALARM, handle_alarm);
  G_MPI_Signal_handler(APP_SIG_HANG, handle_hang);
  G_MPI_Signal_handler(G_MPI_SIG_REP_SYNC, handle_sync);

  rc = MPI_Init(...);

  if ( rc == MPI_RECOVERED )
    recover_app_state();
    if ( G_MPI_Alert_check() != MPI_SUCCESS ) {
      diagnose_recover_reconfigure();
    }
  }

  rc = do_computation();

  MPI_Finalize();
  return 0;
}
```

Figure 4.18: Application's `main` function

for the global variable `alert_cause` (see Subsections 4.3.1 and 4.3.2 and Figure 4.12). The `alert_cause` is set to this new value in the signal handler for the new G-MPI signal. Then in `diagnose_recover_reconfigure()` (where `alert_cause` is read), application processes can execute a distributed algorithm to diagnose the error and to agree on the corrective action. Alternatively, if the application employs rollback recovery of the entire state of the application no further diagnosis of the erroneous state is necessary.

4.4. Ghidrah Cluster Management Middleware

With the goal of deployment for data processing in space, the *Ghidrah* CMM includes aggressive fault tolerance capabilities [Li01, Li02]. The overall structure of *Ghidrah* is shown in Figure 4.19. The system consists of four components: a replicated centralized manager, an agent on each node, a library for user applications, and a ‘‘trusted computer’’ called the Spacecraft Control Computer (SCC). *Ghidrah* supports running multiple parallel applications with gang scheduling; in Figure 4.19 the circles represent processes of three user tasks labeled T1, T2, and T3. The *Manager Group* performs cluster-level decision making, such as scheduling and fault management. An agent on each node reports node status to the *Manager Group*, performs commands at the node on behalf of the *Manager Group*, and provides an interface between application processes and the CMM. The user-level library, linked with every user application, provides the mechanisms for setting up intra-task communication and includes the implementation of G-MPI (Section 4.2). The CMM design and implementation is focused on maintaining the basic cluster functionality despite any single faulty node. The minimum ‘‘basic cluster functionality’’ that must be maintained is the ability to submit new tasks, to continue the execution of tasks on operational nodes, and to maintain overall scheduling and monitoring of the system.

4.4.1. Reliable Centralized Cluster Manager

The most critical part of the *Ghidrah* CMM is the centralized manager. Hence, the manager employs active replication [Li01] across three processes (*Manager Replicas*, or simply *Replicas*) running on different nodes. Each manager operation is performed independently on each of the *Manager Replicas.* Messages exchanged among *Replicas*, and

between *Replicas* and agents are authenticated (signed [Lamp82]) to ensure that faulty nodes cannot forge messages from other nodes, even if the message is forwarded by the faulty node.

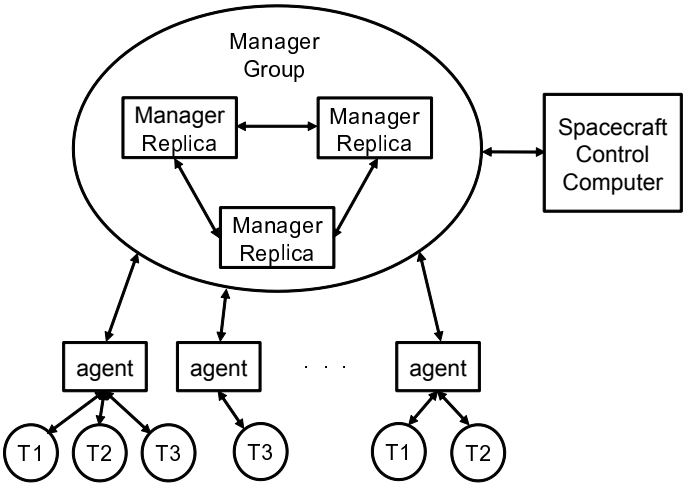


Figure 4.19: Logical structure of the Ghidrah CMM

Agents act only when receiving identical authenticated commands from at least two *Manager Replicas.* Hence, a *Replica* that stops or generates incorrect commands cannot corrupt the system. If any *Replica* or agent suspects an error in a *Replica,* a message is sent to all *Replicas* to initiate a *self-diagnosis procedure.* Self-diagnosis consists of a Byzantine-fault-tolerant agreement protocol to detect and diagnose a *Replica* with corrupted state. If two *Replicas* agree that a third *Replica* is faulty, they run a *replica recovery procedure* to command the agent on the faulty *Replica's* node to terminate the *Replica*. They also send a command to an agent on another node to spawn a new *Replica*.

The *Manager Group* detects node failures by observing a loss of periodic heartbeat messages from *Agents*. Each *Agent* sends a heartbeat message to one backup *Replica* every 75 ms, alternating between the two backup Replicas for each heartbeat. Hence, a backup

Replica should expect to receive a heartbeat from a specific Agent every 150 ms. Each backup *Replica* checks for receipt of Agent heartbeats every 225 ms (three times longer than the Agent's period for sending heartbeats). If heartbeats have not been received from an Agent in two consecutive checks, the *Manager Group* sends a probe message to elicit a response from the Agent. If the Agent does not respond to the probe message within 120 ms, the *Manager Group* declares the node as failed.

The CMM's time to detect a failed node that is executing a *Replica* is longer than for a failed node without a *Replica* running on it because the *Manager Group* diagnoses itself before turning its attention to diagnosing an *Agent's* missing heartbeats. *Replicas* send heartbeats to each other every 75 ms and check for received heartbeats every 150 ms. When a *Replica* detects one missing *Replica* heartbeat it initiates the self-diagnosis procedure to expose the faulty *Replica*. While the *Replicas* are running self-diagnosis, the timer events related to checking for missing *Agent* heartbeats are ignored until the self-diagnosis procedure is finished. In the case that a *Replica* is unresponsive during self-diagnosis, the self-diagnosis procedure times out after 400 ms. This means that on average, detection latency of a failed node running a *Replica* will be 200 ms longer than that of a failed node not running a *Replica*.

4.4.2. An Agent on Each Node

Each ‘‘agent’’ is implemented using three processes, called *Agent*, *Agent-Helper*, and *Agent-Keeper*. The *Agent* communicates directly with the *Manager Group*, including sending periodic heartbeats. It also forwards Ghidrah signals from the *Manager Group* to the destination application process. The *Agent-Helper* receives Ghidrah signals from application processes and forwards them to the local *Agent*. The *Agent-Keeper* is a very small process

that monitors heartbeats from the *Agent* and *Agent-Helper,* restarting both if the heartbeats stop.

4.4.3. Ghidrah Application Library

Ghidrah includes a user-level library to be linked to applications. The library enables application processes to be controlled by the local agent at start-up and termination and to use an asynchronous communication facility known as Ghidrah signals.

The application library implements an initialization function that causes the application process to open two named pipes at startup: one named pipe is for receiving Ghidrah signals from the *Agent* and the other named pipe is for sending Ghidrah signals to the *Agent-Helper*. The first message the process receives from the *Agent* is a *mapping message* that provides the total number of processes in the task, a flag indicating whether the task has been restarted, and a list of processes in the task and their communication endpoint addresses (including which nodes they are running on). The processes in the list are identified by unique *virtual process ID* (VPID) numbers that are assigned by the cluster manager.

The initialization function uses the mapping information to open reliable point-to-point communication channels with each of the other task processes. These communication channels are used for the application's message-passing, i.e. MPI messages. After the mapping message, all subsequent messages sent by the *Agent* to the process are *Ghidrah signals*.

The Ghidrah signals API described in Section 4.1.1 is also implemented in the application library. Ghidrah signals are sent to and received from the cluster manager via the

process's local *Agent*. Ghidrah signals are sent via named pipe to the *Agent-Helper*, which forwards it via another named pipe to the *Agent*. The *Agent* sends the signal to the cluster manager. The cluster manager sends (or broadcasts) the signal to the *Agent* on the destination process's node. That local *Agent* writes the Ghidrah signal contents to the destination process's named pipe and interrupts the application process with a POSIX SIGINT signal. The application library's handler for the SIGINT signal reads the contents of the Ghidrah signal from the named pipe and then invokes the application's handler for that Ghidrah signal, if any has been registered.

The application library also has a finalization function to be invoked by the application when it is ready to terminate. The finalization function properly closes the named pipes, thus alerting the CMM that the process is terminating normally. If the process crashes (terminates abnormally) the improper disappearance of the writer of the named pipe to the *Agent-Helper* causes the OS to deliver the SIGPIPE POSIX signal to the *Agent-Helper*. In either case, the *Agent-Helper* then notifies its local *Agent* of the process termination, and the *Agent* sends a message to cluster manager regarding the event.

4.4.4. Radiation-Hardened Spacecraft Control Computer

The SCC controls the entire spacecraft, including handling communication between the cluster and its human operators on Earth. Loss of the SCC implies loss of the spacecraft. Hence, while the entire cluster is built using commercial-off-the-shelf technology, the SCC uses radiation-hard technology and other aggressive fault tolerance techniques to ensure the survival of the spacecraft. The design of the CMM must take into account the need to interact with the SCC and can take advantage of the existence of this ‘‘hard core.’’ However, the SCC

is not designed for high performance and must not be burdened with routine operation of the cluster.

Ghidrah takes advantage of the SCC by relying on its ability to power-reset the nodes of the cluster. The *Manager Group* commands the SCC to power-reset a node if it stops receiving heartbeats from the node's *Agent*. The *Manager Group* also sends periodic heartbeats to the SCC and a report each time the self-diagnosis procedure is initiated. If the SCC stops receiving consistent heartbeats from at least two *Manager Replicas*, it power-resets all of the cluster nodes (i.e. resets the entire cluster). It is possible that failure of the self-diagnosis procedure will cause *Replicas* to erroneously initiate new rounds of self-diagnosis repeatedly. The SCC maintains a record of recent self-diagnosis initiations and triggers a reset of the entire cluster if the number of self-diagnosis initiations over a specified period of time exceeds a fixed threshold.

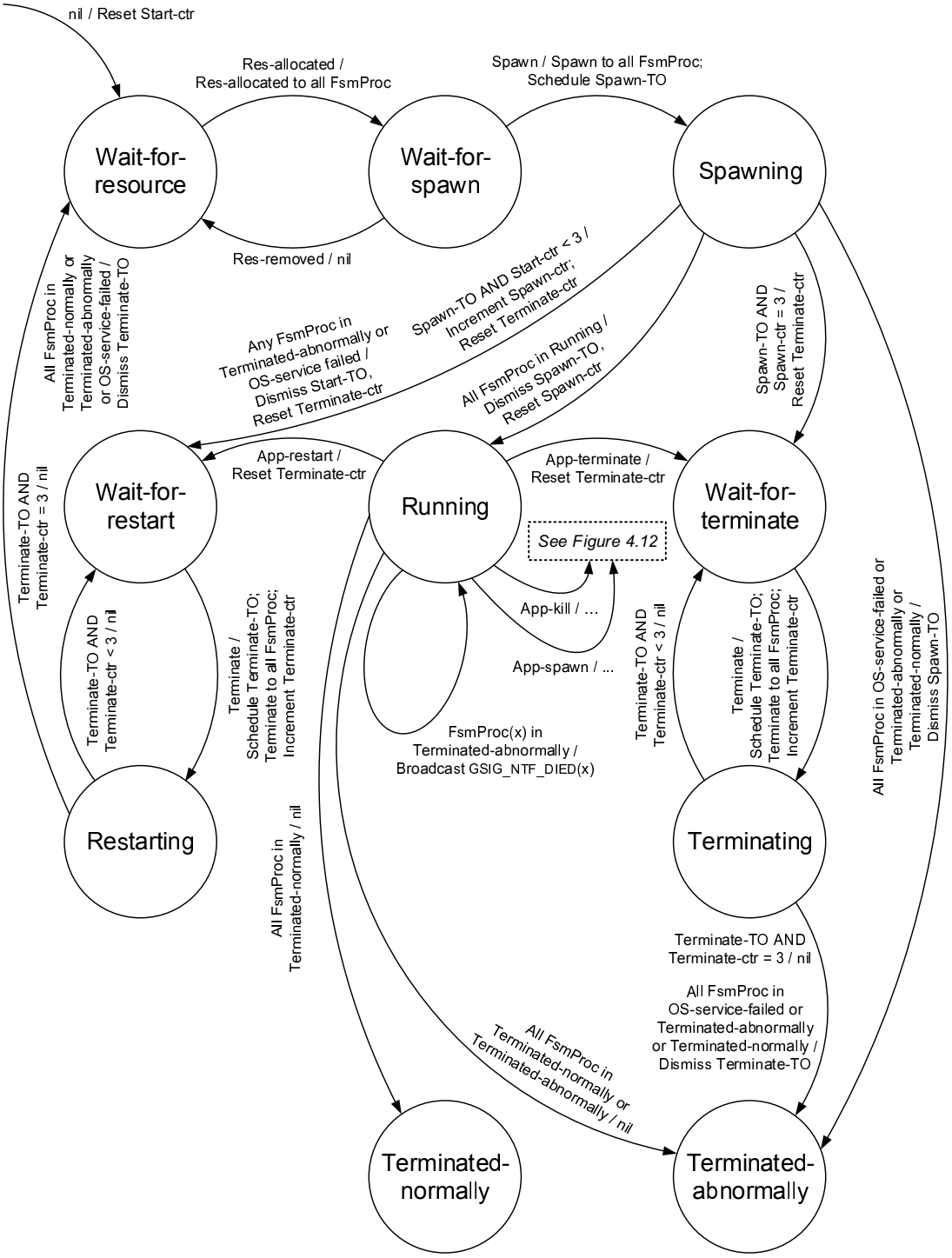


Figure 4.20: *FsmTask*, the cluster manager's finite state machine for a task.

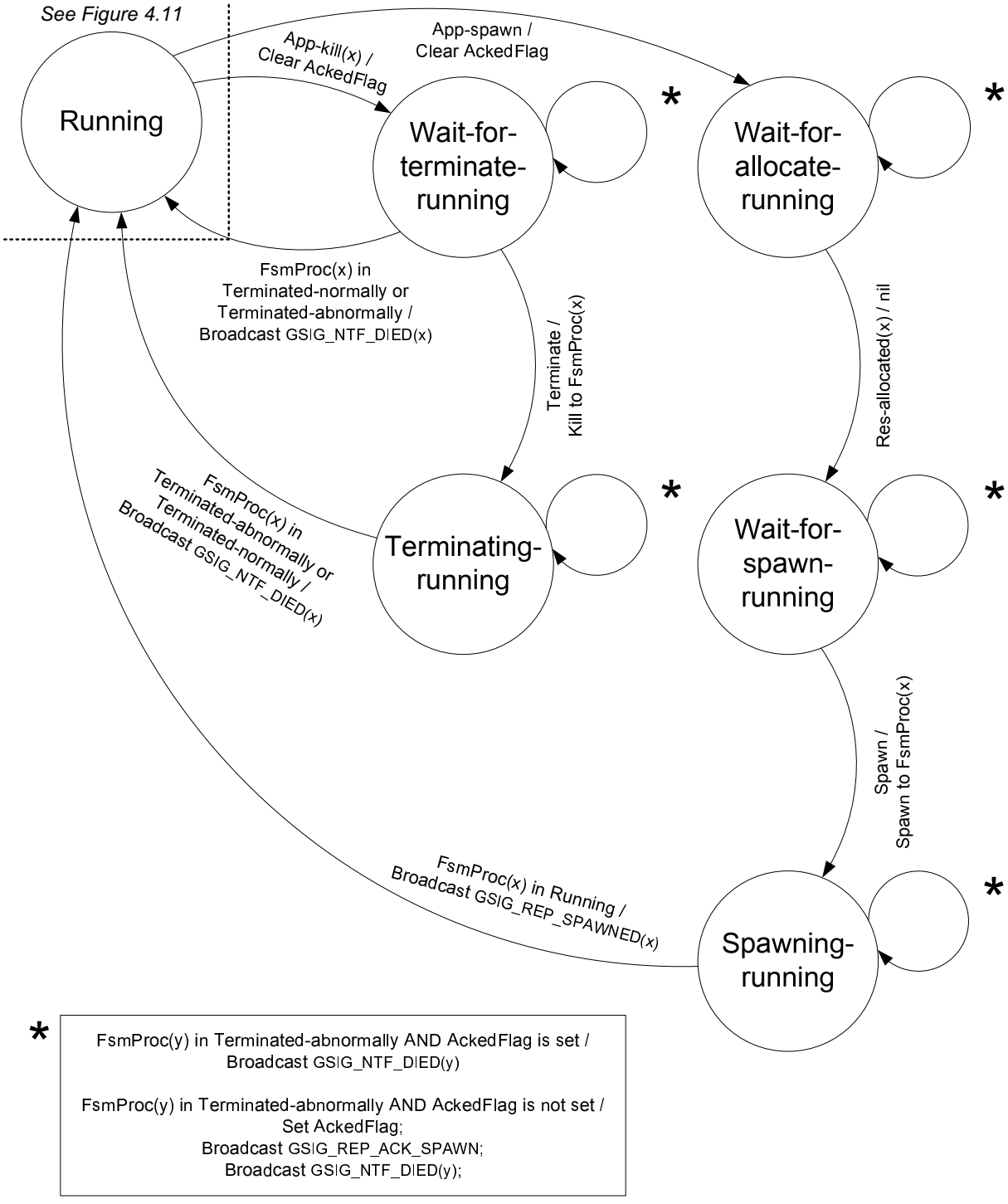


Figure 4.21: Continuation of *FsmTask*: states handling task requests

Label	Meaning
Inputs: Cluster manager events	
Res-allocated	Cluster manager has allocated compute nodes for all of the task's processes.
Res-removed	One or more compute nodes assigned to the task has been removed from the cluster (e.g. due to node failure).
Spawn	Cluster manager is ready to spawn processes for the task.
Terminate	Cluster manager is ready to terminate processes for the task.
Spawn-TO	Spawn timeout event has fired.
Terminate-TO	Terminate timeout event has fired.
App-restart	Cluster manager accepted task's group request to terminate and restart the task.
App-terminate	Cluster manager accepted task's group request to terminate the task.
App-kill(*x*)	Cluster manager accepted task's group request to terminate process with VPID *x*.
App-spawn	Cluster manager accepted task's group request to spawn a process.
Inputs: States in *FsmProc*	
Any *FsmProc* in *st1* {or *st2*}	One or more *FsmProc* is in state *st1* or state *st2*
All *FsmProc* in *st1* {or *st2*}	All of task's *FsmProc* are in state *st1* or state *st2*
FsmProc(*x*) in *st*	*FsmProc* of process with VPID *x* is in state *st*
Outputs: Commands sent to *FsmProc*	
Res-allocated	Notify *FsmProc* that resource for that process has been allocated
Spawn	Notify *FsmProc* that process should be spawned
Terminate	Notify *FsmProc* that process should be terminated
Outputs: Cluster manager actions	
Reset *ctr*	Reset value of *ctr* to 0
Increment *ctr*	Increment value of *ctr*
Schedule *ev*	Schedule timeout event *ev* to fire in the future
Dismiss *ev*	Dismiss (cancel) timeout event *ev*
Broadcast *sig*(*x*)	Broadcast Ghidrah signal *sig* with *arg* set to *x*.
Set/Clear AckedFlag	Indicate that acknowledgment of acceptance of group signal has/has not been broadcast

Table 4.4: Legend of state transitions in *FsmTask*

4.4.5. Implementing Application-Directed Process Management

The Ghidrah CMM keeps track of the state of each task and each task's processes using a set of three finite state machines (FSMs). The cluster manager maintains a FSM for each task, called *FsmTask*. This FSM controls FSMs that the cluster manager maintains for each process of the task, called *FsmProc*. Finally, *FsmProc* issues commands to *Agents* which handle the commands with a per-process FSM called *FsmAgentProc*. The FSMs specify the actions of the CMM with regard to task processes—allocating resources, spawning, scheduling, terminating, and handling process crashes and node failures. The FSMs also specify how the cluster manager handles a task's requests to terminate and spawn processes.

Label	Meaning
Inputs: Cluster manager events (See also Cluster manager outputs to *FsmProc* in Table 4.4)	
Res-allocated	Cluster manager has allocated compute nodes for this task process.
Res-removed	Cluster manager removed compute node (e.g. due to node failure)
Inputs: Messages from *Agent*	
Spawn failed	*Agent* failed to spawn process, or process did not connect to *Agent*
Spawn succeeded	*Agent* spawned process, and process connected to *Agent*
Terminated normally	Process disconnected from *Agent* and terminated normally
Terminated abnormally	Process crashed
Outputs: Commands sent to *Agent*	
Spawn	Command *Agent* to spawn process
Terminate	Command *Agent* to terminate process

Table 4.5: Legend of state transitions in *FsmProc*

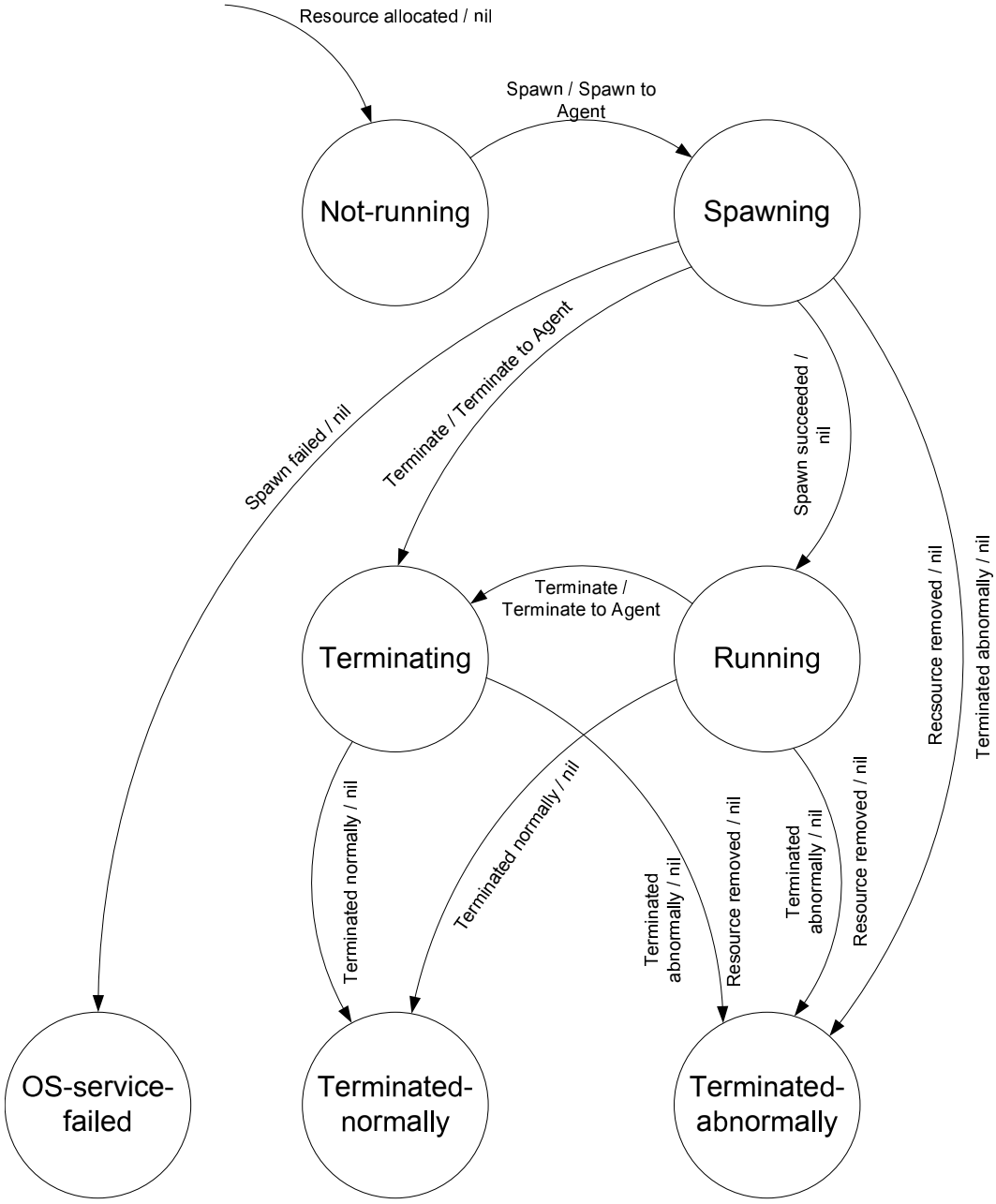


Figure 4.22: *FsmProc*, the cluster manager's finite state machine for a task process

The cluster manager's FSM for managing the state of a task, *FsmTask*, is shown in Figures 4.20 and 4.21. The states of *FsmTask* shown in Figure 4.20 follow the overall ''lifetime'' of a task within the CMM, from the first state where the task awaits allocation of compute nodes before execution, to the terminal states indicating whether the task terminated normally. The inputs to *FsmTask* are events from the cluster manager and state changes from the task's per-process *FsmProc*s. The outputs of this FSM are actions for the cluster manager and commands to the cluster manager's per-process FSMs. These inputs and outputs are defined in Table 4.4.

An instance of *FsmTask* is created and the initial transition to state *Wait-for-resource* is made when the user commands the cluster manager to execute a new task. In state *Wait-for-start*, the compute node resources have been reserved for the task and the task awaits its turn to begin executing. In state *Starting*, the cluster manager has issued commands to the compute nodes' *Agents* to create and begin executing the task processes. Once all of the task processes are confirmed to be running, *FsmTask* transitions to the state *Running*.

When all processes have terminated normally, *FsmTask* transitions to state *Terminated-normally*, after which the state of the task may be cleaned up by the cluster manager. However, task processes may terminate abnormally, request to terminate the entire task, or request to restart the entire task. These three possibilities are shown as transitions out of the *Running* state.

During the *Wait-for* states in *FsmTask*, the task waits in a queue for service by the cluster manager. Ghidrah supports running multiple tasks simultaneously and with gang scheduling. But for simplicity, Ghidrah only starts, restarts, or terminates one task at a time. Therefore, whenever a task-level service is required, *FsmTask* enters a *Wait-for* state, and the task is

placed in the cluster manager's service queue.

Associated with each task and *FsmTask* are two timeout events, *Start-TO* and *Terminate-TO* and two counters, *Start-ctr*, *Terminate-ctr*. The timeout events are scheduled to detect failures where the system takes too long to start or terminate processes of a task. The counters are used to retry the failed operations for a limited number of times. For example, when any of a task's processes fail to spawn *FsmTask* terminates all of the task's processes and retries allocation and spawning.

Two new state transitions out of state *Running* in *FsmTask* handle acceptance of the task's requests to terminate a process and spawn a process. The additional states that handle these requests are shown in Figure 4.21. The FSM clears and sets a flag called *AckedFlag* according to whether the cluster manager has acknowledged the acceptance of the task's request. As described in Section 4.1.3, *FsmProc* only broadcasts an acknowledgment of the acceptance immediately before broadcasting another signal that is generated before the request has been completed.

Each task process has two FSMs tracking its state: *FsmProc* is maintained by the cluster manager, and *FsmAgentProc* is maintained by the process's local *Agent*. The inputs of *FsmProc* are commands generated by *FsmTask* and messages from the process's *Agent*. The outputs of *FsmProc* are commands to the *Agent*. *FsmProc*'s inputs and outputs are defined in Table 4.5.

State machine *FsmAgentProc* is executed by the task process's local *Agent*. Its inputs are commands that are outputted from *FsmProc* as well as updates from the OS regarding the state of the task process. Its outputs are system calls to the OS to create and terminate

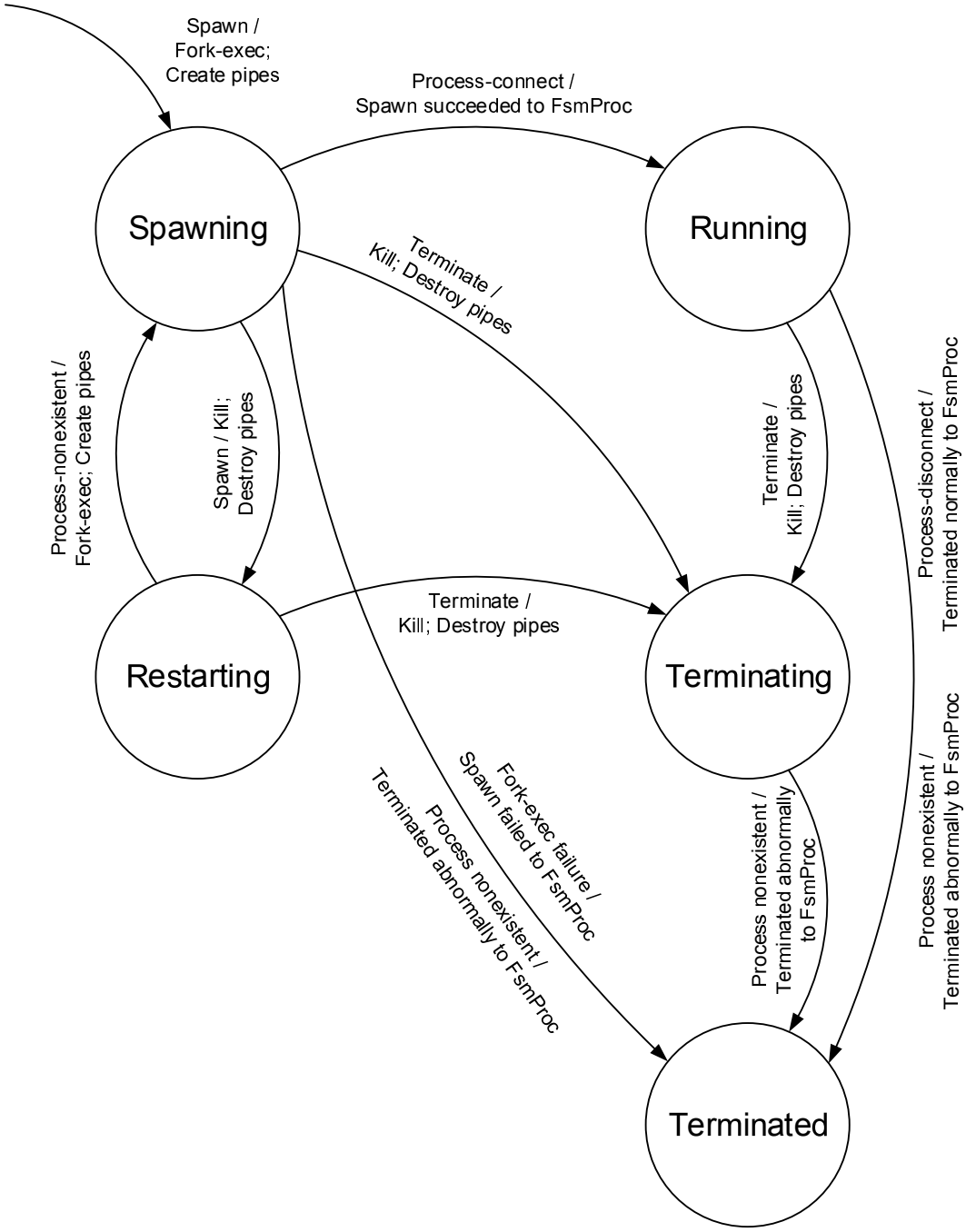


Figure 4.23: *FsmAgentProc*, the *Agent's* finite state machine for a task process

Label	Meaning
Inputs: Cluster manager events (See Cluster manager outputs to *FsmProc* in Table 4.4)	
Inputs: OS events	
Process-connect	Task process connected to pipes and requested task mapping
Process-disconnect	Task process disconnected from pipes and terminated normally
Process-nonexistent	Task process crashed without properly disconnecting from pipes
Fork-exec failure	OS returned error code in either `fork` or `exec`
Outputs: Commands sent to *FsmProc* (cluster manager)	
Spawn succeeded	Notifies the cluster manager that the process is running
Terminated normally	Notifies the cluster manager that the process terminated normally
Terminated abnormally	Notifies the cluster manager that the process has crashed

Table 4.6: Legend of state transitions in *FsmAgentProc*

processes and create and destroy the pipes that are used for communication between the *Agent* and the task process. It also outputs messages that are sent to *FsmProc* in the cluster manager. All of *FsmAgentProc*'s inputs and outputs are defined in Table 4.6.

4.5. Group Communication in Ghidrah CMM

Ghidrah CMM uses a custom group communication protocol for messages sent by the group of Manager Replicas to the Agents on each compute node. During testing of our implementation of Ghidrah group signals, a major design flaw was discovered in this group communication protocol. This section describes the group communication protocol, its design flaw, and how the flaw was fixed.

4.5.1. An Optimized Design for Group Communication

Ghidrah CMM processes send messages via reliable connection-based point-to-point communication channels. Each Agent is connected to each Replica, and all of the Replicas are connected to each other; As is typical for reliable point-to-point communication protocols, copies of sent messages are kept by the sender in case retransmission is needed. A message sent from the Manager Group to an Agent or sent from an Agent to the Manager Group is referred to as a *group message*. The group message protocols are layered on top of the reliable point-to-point channels. A group message is sent to an Agent by having each Replica independently generate and send a *group message replica* to the Agent. The Agent sends a message to the Manager Group by sending to the primary Replica. The primary Replica forwards the message to the Backup replicas.

Although the group message protocol is layered on top of the point-to-point protocol, the separation between the two layers is not strict due to several optimizations we designed for Ghidrah. The first optimization is that group messages to the Agents are not buffered for retransmission at the group message protocol layer. Instead, they are buffered and retransmitted as part of the reliable point-to-point layer. The second optimization is that for group messages from the Manager Group and from Agents are not acknowledged by point-to-point acknowledgments. Instead, they are acknowledged by *group acknowledgments* sent by the group message protocol layer. The group message protocols and the optimizations are described in more detail below.

A listing of pseudocode accompanies the description of the communication protocols in this section. A legend for the terminology and message formats used in the pseudocode appears in Figure 4.24. The key variables used to implement the protocols are listed in Figure

4.25 for the Replicas and Figure 4.26 for the Agents. Pseudocode for the point-to-point communication layer is listed in Figures 4.27 through 4.29. Pseudocode for the group communication layer is listed in Figures 4.30 through 4.32 for group messages from the Manager Group to the Agent and Figures 4.33 through 4.35 for group messages from the Agent to the Manager Group.

csn : The Channel Sequence Number (CSN) included in and defining the order of group messages from a client to the server group. *gssn* : The Global Send Sequence Number (GSSN), included in all group messages from the server group to clients, defines the order of group messages sent by the server group to all clients.

rsn : The Receive Sequence Number (RSN) assigned to messages destined for the server group by the primary server replica in order to establish a total order on all group messages.

seq : Point-to-point reliable sequence number attached to all reliably-sent messages.

⟨GROUP-MSG-CG, *csn*, $gssn_{ack}$, *payload*, *c*⟩ : A group message of type GROUP-MSG-CG from client *c* to the server group, with CSN of *csn*, acknowledgment of group-to-client message with GSSN of $gssn_{ack}$, and contents *payload*. The contents *payload* may be ∅ (empty), meaning that this GROUP-MSG-CG only acknowledges that the client has received group message with GSSN of $gssn_{ack}$.

⟨GROUP-MSG-GC, csn_{ack}, *gssn*, *payload*, *i*⟩ : A group message of type GROUP-MSG-GC from a server replica *i* to the client, containing an acknowledgment of receipt of the client's group message with CSN of csn_{ack}, the GSSN of the server group message *gssn*, and the contents *payload*. The contents *payload* may be ∅, meaning that this GROUP-MSG-GC only acknowledges that the server group has received client message with CSN of csn_{ack}.

⟨WRAPPER-MSG, *rsn*, *msg*, *i*⟩ : A message of type WRAPPER-MESSAGE containing RSN *rsn*, group message *msg* from a client, sent by server replica *i* to another server replica.

⟨PTP-MSG, *seq*, *ackseq*, *msg*, *grp*, *src*, *dest*, *cert*⟩ : A point-to-point reliable message sent with point-to-point sequence number *seq*, acknowledgment of the destination's send sequence number *ackseq*, and payload *msg* of type GROUP-MSG-GC or some other point-to-point message. If *msg* payload is ∅, then this message is only a point-to-point acknowledgment. If *msg* is of type GROUP-MSG-GC, then *grp* is set to 1; otherwise *grp* is set to 0. The message also includes a checksum and the sender's signature, together represented by *cert*.

match (*m*1, *m*2) : For any two group message replicas (GROUP-MSG-GC) *m*1, *m*2, match (*m*1, *m*2) returns *true* if and only if all fields of *m*1 and *m*2 except the last field (point-to-point source) are equal.

unwrap (*wrmsg*) : For any message of type WRAPPER-MSG *wrmsg*, unwrap (*wrmsg*) returns *wrmsg*'s *msg*

Figure 4.24: Legend for pseudocode

Replicated state:
$csn_{rcvd}(x)$: the most recent in-order CSN received from client x.
$gssn_{sent}$: the most recently used GSSN.
R : set of identifiers of the current set of active server replicas
$primary$: identifier of the current primary server replica, $primary \in R$
B : set of identifiers of the current set of backup server replicas, $R = B \cup \{primary\}$
rsn_{last} : the last in-order RSN processed by the server replica

Non-replicated state:
$conn(x)$: boolean value that is true iff a point-to-point connection with endpoint x has been established; initialized to false.
L : a log of delivered group messages, used to provide group messages that are ‘‘missing’’ in other server replicas.
PMB : Point-to-point Message Buffer stores reliably-sent point-to-point messages that have not been acknowledged
OMB : Out-of-order Message Buffer stores messages received out of order for later delivery
$seq_{rcvd}(x)$: the point-to-point sequence number most recently received in-order from source x
$seq_{sent}(y)$: the point-to-point sequence number most recently sent in-order to destination y
$seq_{acked}(y)$: the point-to-point sequence number of the message most recently acknowledged by destination y

Figure 4.25: State of a Replica

A significant part of Replicas’ performance overhead is comparison of replicated state; therefore, we optimize the group message protocol to minimize the size of replicated state. In the Ghidrah group communication protocol, sent but unacknowledged group message replicas are *not* part of the replicated state. Rather, sent group message replicas become part of the state of the point-to-point connection that each Replica has with the Agent. Thus, retransmission of a group message replica is accomplished independently by each Replica’s reliable point-to-point connection with the Agent.

Each Replica maintains a sequence number that is incremented every time a group message is generated (regardless of the destination of the message). This sequence number, called the Global Send Sequence Number (GSSN), is attached to each group message and is part of replicated state. Since fault-free Replicas are deterministic, they attach matching

$conn(x)$: boolean value that is true iff a point-to-point connection with endpoint x has been established; initialized to false.
csn_{sent} : the highest CSN sent to the server group
csn_{acked} : the highest CSN acknowledged by the server group
CMB : Client Message Buffer stores unacknowledged group messages sent to the server group
GMB : Group Message Buffer stores group message replicas admitted from server replicas
$gssn_{deliv}$: GSSN of the most recently delivered group message
R : set of identifiers of the current set of active server replicas
$primary$: identifier of the current primary server replica, $primary \in R$
B : set of identifiers of the current set of backup server replicas, $R = B \bigcup \{primary\}$
PMB : Point-to-point Message Buffer stores reliably-sent point-to-point messages that have not been acknowledged
OMB : Out-of-order Message Buffer stores messages received out of order for later delivery
$seq_{rcvd}(x)$: the point-to-point sequence number most recently received in-order from source x
$seq_{sent}(y)$: the point-to-point sequence number most recently sent in-order to destination y
$seq_{acked}(y)$: the point-to-point sequence number of the message most recently acknowledged by destination y

Figure 4.26: State of an Agent

GSSNs to matching group message replicas they send. The GSSN is used by the Agent to detect missing or out-of-sequence group messages due to a faulty Replica.

In the Agent, we distinguish among three states for the receipt of a group message replica: 1) a message being physically *received*; 2) a message being *admitted* after passing an integrity check (checksum), authentication, and a message order check (elimination of gaps in *seq*); and 3) a message being *delivered* in the Agent for processing after ensuring that a majority of Replicas sent matching group message replicas. The reliable point-to-point protocol is modified on the Agent so that, for group message replicas, a point-to-point acknowledgment is *not sent* when the group message replica is *received*. Instead, a *group acknowledgment* is sent only when the group message is *delivered* in the Agent. Since group messages must be acknowledged by group acknowledgments, point-to-point acknowledgments of group message replicas are redundant and therefore omitted. This

```
ptp_send(payload, grp, dest) {
    # src is sender's identity
    if !conn(dest) and payload ≠ ∅ then
        let ptpmsg = ⟨PTP-MSG, 0, 0, payload, grp, src, dest, 0⟩
        ptpmsg_acked ← false
        add ptpmsg to PMB
    else if conn(dest) then
        if payload ≠ ∅ then increment seq_sent(dest)
        let ptpmsg = ⟨PTP-MSG, seq_sent(dest), seq_rcvd(dest), payload, grp, src, dest, cert⟩
        send ptpmsg to dest
}
ptp_reliable_send(payload, grp, dest) {
    ptp_send(payload, grp, dest)
    add ptpmsg to PMB
    schedule RetransmitTimeout(dest, seq_sent(dest)) event
}
□ Upon receiving reliably-sent point-to-point message ptpmsg from src do {
    # dest is receiver's identity
    # ptpmsg = ⟨PTP-MSG, seq, ackseq, payload, grp, src, dest, cert⟩
    if conn(src) then
        if seq ≥ seq_rcvd(src) + 1 + window or seq ≤ seq_rcvd(src) then drop out-of-order ptpmsg
        else if seq > seq_rcvd(src) + 1 then
            add ptpmsg to OMB
            if grp = 0 then ptp_send(∅, grp, src)
        else if seq = seq_rcvd(src) + 1 then
            seq_rcvd(src) ← seq
            if grp = 0 then
                process_ptp_ack(src, ackseq)
                ptp_send(∅, grp, src)
            deliver_ptp_message(ptpmsg)
        else # seq < seq_rcvd(src) + 1
            if payload = ∅ then
                process_ptp_ack(src, ackseq)
                if ackseq < seq_sent(src) then
                    let msg be the message in PMB whose seq = ackseq + 1
                    send msg to src
}
```

Figure 4.27: Point-to-point communication procedures that are common to Agents and Replicas: sending and receiving messages

optimization reduces load on Replicas by freeing them from processing point-to-point acknowledgments of group message replicas. Since acknowledgments are sent only when the message is *delivered*, the point-to-point protocol *admits* messages that are duplicates at the point-to-point connection level.

Admitted group message replicas are buffered by the Agent, in the Group Message Buffer (GMB), until the requisite $f + 1$ matching group message replicas are admitted. (Ghidrah is currently implemented with 3 Replicas, where $f = 1$.) At that point, the group message is delivered at the Agent. The Agent then sends to the Manager Group a group acknowledgment containing the GSSN of the group message just delivered. When a Replica processes this acknowledgment, it notifies its point-to-point connection to that Agent that any buffered sent messages with GSSNs up to the acknowledged GSSN should be considered acknowledged.

As described above, the point-to-point sequence number *seq* is used to ensure that there are no gaps (lost messages) in the sequences of messages admitted by the Agent from each Replica. The Agent detects admitted duplicate messages based on their GSSN. For this purpose, the Agent maintains the GSSN of the last delivered group message ($gssn_{deliv}$). If the GSSN of an admitted group message replica is less than or equal to $gssn_{deliv}$, the Agent sends to the Manager Group a group acknowledgment containing $gssn_{deliv}$. If there are no messages in the GMB with GSSNs matching the admitted message, the admitted message is discarded. The message is also discarded if the GMB already contains a message from the same replica with a matching GSSN.

If a Replica is faulty, the Agent may not receive matching group message replicas from all the Replicas. Faulty Replicas may fail to send their group message replicas to the Agent or

```
deliver_ptp_messages_in_omb (src) {
    while there exists ptpmsg in OMB such that ptpmsg's seq = seq_rcvd(src) + 1 do
        # ptpmsg = ⟨PTP-MSG, seq, ackseq, msg, 1, src, dest, cert⟩
        remove ptpmsg from OMB
        if grp = 0 then
            process_ptp_ack (src, ackseq)
            ptp_send (∅, grp, src)
        deliver_ptp_message (ptpmsg)

connect (x) {
    initiate 3-way handshake connection establishment with x
}

□ Upon establishing connection with x do {
    conn(x) ← true
    # seq_sent(x) and seq_rcvd(x) have been initialized as part of the connection establishment protocol
    seq_acked(x) ← seq_sent(x)
    for each msg in PMB with dest = x, do
        ptp_send (msg's payload, grp, x)
        if ptpmsg_acked then remove msg from PMB
        else schedule RetransmitTimeout (dest, seq_sent(x)) event
}

disconnect (x) {
    conn(x) ← false
    dismiss all RetransmitTimeout (x, seq) events, for all seq
    remove all messages sent to x from PMB
    remove all messages received from x from OMB
}
```

Figure 4.28: Point-to-point communication procedures that are common to Agents and Replicas: delivering messages, connecting, and disconnecting

send group message replicas that do not match the correct message. If some admitted group message replicas with the same GSSN do not match, at some point the Agent can determine that it is no longer possible for it to get the requisite $f + 1$ *matching* group message replicas. If at most f server replicas can be faulty, this situation implies that the Agent is faulty. Hence, the Agent terminates itself.

```
□ Upon RetransmitTimeout(d, seq) event fires do
    for each ptpmsg in PMB such that ptpmsg's dest = d and seq is in [seq_acked(d) + 1, seq_sent(d)]
        # ptpmsg = ⟨PTP-MSG, seq, ackseq, payload, grp, src, dest, cert⟩
        if sender is client and d = primary and ptpmsg's grp = 1 and
        ptpmsg has been transmitted more than 3 times then
            for each replica j in B do
                ptp_send(payload, grp, j)
        else
            ptp_send(payload, grp, d)
    schedule RetransmitTimeout(d, seq) event

process_ptp_ack(src, ackseq) {
    s ← seq_acked(src) + 1
    while s ≤ ackseq and ptpmsg exists in PMB such that ptpmsg's seq = s do
        dismiss RetransmitTimeout(dest, s)
        remove ptpmsg from PMB
        increment seq_acked(src)
        increment s
}
```

Figure 4.29: Point-to-point communication procedures that are common to Agents and Replicas: handling acknowledgments and acknowledgment timeouts

```
group_send_to_client(payload, c) {
    if payload ≠ ∅ then
        increment gssn_sent
    let msg = ⟨GROUP-MSG-GC, gssn_sent, csn_rcvd(c), payload, i⟩
    ptp_reliable_send(msg, 1, c)
}
```

Figure 4.30: Replica *i* reliably sends a group message replica to Agent *c*

When the first group message replica with a particular GSSN is stored in the GMB, the Agent schedules a timer event associated with that message's GSSN. If the Agent admits group message replicas with the same GSSN from all the Replicas before the expiration of the timer, the timer event is canceled. At that point, all the messages with that GSSN are removed from the GMB. However, if the timer event fires, indicating that the Agent failed to receive

```
deliver_ptp_message (ptpmsg) {
    # ptpmsg = ⟨PTP-MSG, seq, ackseq, msg, grp, src, c, cert⟩
    if grp = 0 then handle msg
    else
        # msg = ⟨GROUP-MSG-GC, gssn, ackcsn, payload⟩
        if gssn > gssn_deliv then
            if ∃m ∈ GMB such that m's src = src and match (m's payload, payload) then
                # msg is a duplicate message replica
                discard msg
            else
                GMB ← GMB ∪ {msg}
                let M_gssn = {m ∈ GMB : m's msg's seq = seq}
                let M_agree = {m ∈ M_gssn : match (m's payload, payload)}
                if M_gssn = {msg} then schedule ReplicaTimeout (seq) event
                else if |M_agree| = ⌊|R| / 2⌋ + 1 then
                    deliver_group_message(msg)
                    for each m∈M_gssn do
                        # m = ⟨GROUP-MSG-GC, gssn, ackcsn, payload⟩
                        # seq = point-to-point sequence number in ptpmsg whose msg payload is m
                        # src = point-to-point sender of m
                        seq_rcvd(dest) ← seq
                        deliver_ptp_messages_in_omb (src)
                else if |M_seq| − |M_agree| > ⌈|R| / 2⌉ − 1 then client c terminates
        else # gssn ≤ gssn_deliv
            let M_gssn = {m ∈ GMB : m's gssn = gssn}
            if M_gssn = ∅ or ∃m ∈ M_gssn : m's src = src then discard msg
            else
                GMB ← GMB ∪ {msg}
                M_gssn ← M_gssn ∪ {msg}
                let M_agree = {m ∈ M_gssn : match (m's payload, payload)}
        if |M_gssn| = |R| then
            dismiss ReplicaTimeout (gssn) event
            GMB ← GMB − M_gssn
            if |M_agree| = |R| then
                seq_rcvd(src) ← seq
                if payload ≠ ∅ then client_send_to_group (∅)
                deliver_ptp_messages_in_omb (src)
            else client_send_to_group (''error report'')
}
```

Figure 4.31: Agent *c* identifies the next group message to deliver

enough matching group message replicas in a timely way, the Agent sends an error report to every Replica. Also, if the Agent receives all group message replicas but some of those messages did not match, the Agent sends a point-to-point error report to every Replica. If, at the time of firing, the GSSN associated with the timer is less than or equal to $gssn_{deliv}$ (the message has been delivered), the Agent removes from the GMB all the messages with that GSSN. However, if the GSSN associated with the timer is greater than $gssn_{deliv}$ (the message has not been delivered), the Agent reschedules the timeout event. If three such timeout events for the same GSSN fire, the Agent concludes that the messages it admitted with that GSSN were sent by faulty Replicas and removes from the GMB all the messages with that GSSN.

To support reliable group communication from the Agent to the Manager Group, each Agent maintains a sequence number called the Channel Sequence Number (CSN). The CSN is incremented and included in every group message with a non-empty payload. (The CSN is not incremented when the Agent sends a group acknowledgment.) The CSN is used by the Replicas to enforce FIFO delivery of the Agent's group messages. The Agent manages the connection to the Manager Group in the same way that it would manage a reliable point-to-point connection, except that the CSN is used to order messages instead of a low-level sequence number. At the point-to-point layer, to send a group message, the Agent sends the message to a particular Replica (usually the *primary* Replica). As an optimization, Replicas do not acknowledge the Agent's group messages using point-to-point acknowledgments. Instead, once the group message is delivered to a Replica, the Replica sends a *group acknowledgment* to the Agent. The group acknowledgment contains the CSN of the delivered message. The group acknowledgments are handled in the same way as group message replicas regarding receipt, admittance, and delivery.

```
deliver_group_message (msg) {
    # msg = ⟨GROUP-MSG-GC, gssn, ackcsn, payload, src, dest, cert⟩
    increment gssn_deliv
    process_server_group_ack (ackcsn)
    handle payload
}

process_server_group_ack (ackcsn) {
    if ackcsn = csn_acked + 1 then
        for each ptpmsg in PMB whose payload has csn = ackcsn
            # ptpmsg = ⟨PTP-MSG, seq, ackseq, payload, grp, src, dest, cert⟩
            process_ptp_ack (src, seq)
        increment csn_acked
}

□ Upon ReplicaTimeout (seq) expires do {
    let M_seq = {m ∈ GMB : m's msg's seq = seq}
    client_send_to_group (''error report'')
    if ReplicaTimeout (seq) timer has expired three times then
        GMB ← GMB − M_seq
    else
        reschedule ReplicaTimeout (seq) timer event
}
```

Figure 4.32: Agent *c* delivers a group message, processes a group acknowledgment, and handles a group message timeout

```
client_send_to_group (payload) {
    if payload ≠ ∅ then
        increment csn_sent
    let msg = ⟨GROUP-MSG-CG, csn_sent, gssn_deliv, payload, c⟩
    ptp_reliable_send (msg, 1, primary)
}
```

Figure 4.33: Agent *c* reliably sends a group message to the Manager Group

If an Agent's group message needs to be retransmitted, the Agent retransmits the point-to-point message to the primary Replica up to three times. If the message needs to be

```
deliver_ptp_message (ptpmsg) {
    # ptpmsg = ⟨PTP-MSG, seq, ackseq, msg, grp, src, i, cert⟩
    if grp = 1 then
        if src is client c then
            # msg = ⟨GROUP-MSG-CG, csn, ackgssn, payload, src⟩
            if i = primary then
                increment rsn_last
                let wrmsg = ⟨WRAPPER-MSG, rsn_last, ptpmsg, i⟩
                for each j ∈ B do
                    ptp_reliable_send (wrmsg, 0, j)
                deliver_group_message (wrmsg)
            else # i ∈ B
                let wrmsg = ⟨WRAPPER-MSG, 0, ptpmsg, i⟩
                ptp_reliable_send (wrmsg, 0, primary)
                schedule PrimaryProgressTimeout (c, csn) event
    else # grp = 0
        if src is client c then
            handle ptpmsg
        else if src is replica j ∈ R then
            # msg = ⟨WRAPPER-MSG, rsn, m, j⟩
            if i = primary then
                m ← unwrap (msg)
                deliver_ptp_message (m)
            else if i ∈ B then
                if msg's rsn ≠ rsn_last + 1 then
                    initiate self-diagnosis
                else
                    increment rsn_last
                    m ← unwrap (msg)
                    if m is of type GROUP-MSG-CG then
                        deliver_group_message (msg)
                    else
                        handle m
    }

    □ Upon PrimaryProgressTimeout (c, csn) event fires do
        if csn_rcvd(c) < csn then
            initiate self-diagnosis
```

Figure 4.34: Replica *i* processes a group message from Agent *c* or wrapper message from Replica *j*

retransmitted more than three times, the Agent sends the message using the point-to-point protocol to both backup Replicas. This ensures that the Manager Group receives the Agent's message in the case of a faulty primary Replica.

4.5.2. A Flaw in the Group Communication Protocol

Because of the increased interaction between a fault-tolerant application and the cluster manager generated by our design, the group communication protocols were exercised more thoroughly than in previous testing of Ghidrah CMM. During this testing, we discovered two flaws with the group communication protocols as described above.

In the manifestation of the first flaw, the Agent's group acknowledgment causes the point-to-point protocol to send messages out of sequence. Consider Figure 4.36, where the Replicas of the Manager Group, labeled as *pri*, *b*1, and *b*2, send a group message with GSSN 10 to an Agent. The Figure shows point-to-point messages between the Agent and each of the Replicas. Replica *b*2's group message replica is lost due to a fault. The Agent delivers the group message when after receiving matching group message replicas from *pri* and *b*1, and it sends a group acknowledgment with GSSN 10. The group acknowledgment causes all of the Replicas to dismiss the point-to-point messages involved in sending GSSN 10. Later, the Manager group sends another group message with GSSN 11 to the same agent. The point-to-point message from *b*2 that carries the group message replica is rejected by the Agent's point-to-point protocol as out of order. Furthermore, *b*2 cannot retransmit the group message replica of GSSN 10 because it has been dismissed by the group acknowledgment. Hence the point-to-point connection between *b*2 and the Agent is broken.

The fix for the flaw is to have the Agent send a group acknowledgment only after

```
deliver_group_message (wrmsg) {
    let msg = unwrap (wrmsg)
    # msg = ⟨GROUP-MSG-CG, csn, ackgssn, payload, c⟩
    if i ∈ B and PrimaryProgressTimeout (c, csn) event was scheduled then
        dismiss PrimaryProgressTimeout (c, csn) event
    process_client_group_ack (ackgssn)
    if payload ≠ ∅ then
        if csn = csn_rcvd(c) + 1 then
            increment csn_rcvd(c)
            group_send_to_client (∅, c)
            L ← L ∪ {wrmsg}
            handle payload
}

process_client_group_ack (ackgssn)
    # Find the ptp msg buffers associated with GSSN ackgssn.
    let ptpmsg be the message in PMB such that ptpmsg's group message msg's gssn = ackgssn
    # ptpmsg = ⟨PTP-MSG, seq, ackseq, msg, 1, src, dest, cert⟩
    if !conn(dest) then
        ptpmsg_acked ← true
    else
        if seq > seq_acked(dest) then
            seq_acked(dest) ← seq
            dismiss RetransmitTimeout(dest, seq)
            remove ptpmsg from PMB
}
```

Figure 4.35: Replica *i* delivers a group message from Agent *c*

receiving group message replicas from all of the Replicas. It still delivers group messages after receiving $f + 1$ group message replicas, but sending group acknowledgment is delayed. If a group message replica is lost, as in the case of *b*2's GSSN 10 message, the entire Manager Group will eventually retransmit the group message. This is shown in Figure 4.37.

The second flaw relates to an attempted optimization to combine both point-to-point messages and group messages onto one point-to-point connection. Group messages would be acknowledged with group acknowledgments and point-to-point messages that did not carry

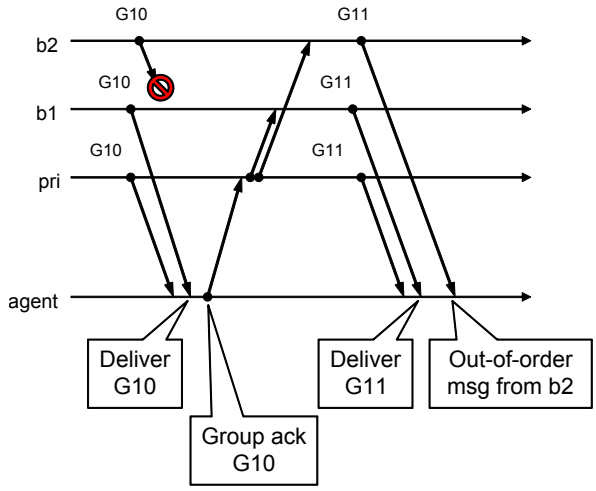


Figure 4.36: A flaw exposed by a dropped group message replica. Replica *b2* cannot retransmit group message replica G10 because it has been dismissed by the Agent's group acknowledgment.

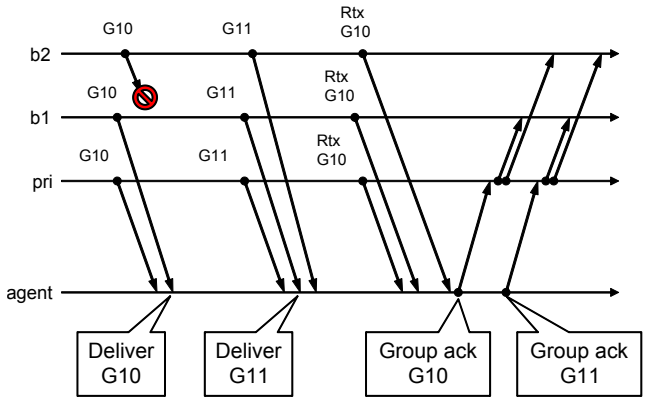


Figure 4.37: Agent sends a group acknowledgment only after receiving all three group message replicas

group message payloads would be acknowledged with point-to-point acknowledgments. The problem is illustrated in Figure 4.38, where *b*2 sends a point-to-point message to the Agent which is lost due to a fault, and then the Manager Group sends a group message with GSSN 10. Because group message replicas are buffered and retransmitted by the point-to-point

protocol in the Replicas, the group acknowledgment processed by *b*2 causes it to also dismiss all preceding point-to-point messages. Hence, *b*2's point-to-point message is permanently lost.

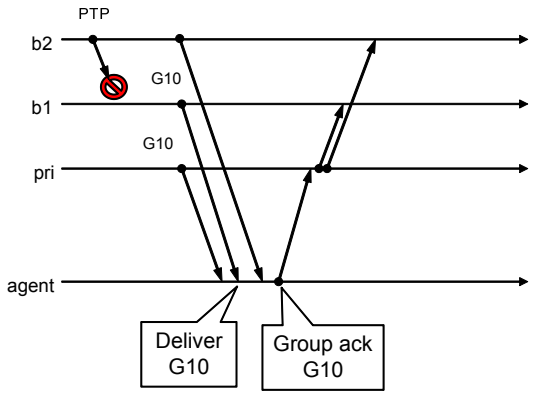


Figure 4.38: A flaw exposed by a dropped point-to-point message. Replica *b2* will never retransmit the point-to-point message because the Agent's group acknowledgment also serves as a cumulative point-to-point acknowledgment.

Another fundamental issue with combining group messages and point-to-point messages on one channel is that there is no guarantee that Agents deliver point-to-point messages and group messages in the same order that they are sent by a Replica. This is due to the different conditions of delivery between the two protocols.

Our fix for the second flaw was to separate the point-to-point messages and the group message replicas into two separate communication channels. Although this means that each Replica and Agent must maintain twice as many reliable point-to-point channels, the change does not increase the size of replicated state in the Replicas.

Chapter Five

Evaluation of the Layered Approach

We evaluated the multi-layered approach for implementing fault tolerance for distributed applications by running fault injection campaigns on MPI applications that were modified for fault tolerance. This chapter begins in Section 5.1 with a description of four MPI applications that were modified to tolerate process crashes, process hangs, and incorrect results. The new CMM services were used to implement application-specific mechanisms for error detection and recovery for these applications. Application-transparent mechanisms were also implemented for comparison, and they are described in Section 5.2.

Section 5.3 describes our experimental setup. We used software-implement fault injection (SWIFI) combined with virtualization to facilitate the execution of a variety of fault injection campaigns.

Before validating the fault-tolerant MPI applications, the fault tolerance of the Ghidrah CMM needed to be tested for reliability. If the CMM is not reliable, we cannot expect applications to run on the cluster reliably. Section 5.4 describes a set of fault injection campaigns used to exercise the fault tolerance mechanisms of the Ghidrah CMM. These campaigns exposed 11 flaws in the CMM which were fixed before higher layers of the cluster software architecture were tested.

After testing the reliability of the CMM, the response times of the new CMM services were measured. Section 5.5 presents these measurements and describes the fault injection campaigns used to obtain them.

Finally, Sections 5.6 and 5.7 discuss the performance of the fault-tolerant MPI applications and their fault tolerance mechanisms. Section 5.6 presents the performance overhead of the application fault tolerance mechanisms when no faults are injected, and Section 5.7 describes the fault injection campaigns and fault injection results for each of the MPI applications. A brief summary in Section 5.8 closes the chapter.

5.1. Implementing Application-Level Fault Tolerance for MPI Applications

To demonstrate the effectiveness of the CMM services for application fault tolerance, application-specific fault tolerance mechanisms were implemented for four MPI applications. The fault tolerance mechanisms enabled the MPI applications to detect errors including incorrect results, hung processes and crashed processes. This section describes the applications and the modifications made to implement application-specific fault tolerance mechanisms.

5.1.1. LAMMPS

Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) is a classic molecular dynamics simulation code [Plim95] developed at Sandia National Laboratories. LAMMPS is written in MPI and C++, and it is one of the applications in the SPEC MPI2007 benchmark suite [Müll10]. The simulation is parallelized by dividing the simulated three-dimensional space into regions, with the simulation of one region assigned to one process. The simulation runs in iterations; each iteration simulates one time step, involving local computations and communication among processes computing on neighboring regions. After each iteration, LAMMPS computes and outputs summary thermodynamic data of the entire

system, such as number of molecules, temperature, pairwise energy, and pressure are computed for the entire space.

LAMMPS code already implements some application-specific fault tolerance mechanisms. Because LAMMPS is designed to simulate closed systems, it checks that the number of molecules in the system remains constant. When molecules are ‘‘lost’’ due to computational error, LAMMPS outputs a warning but does not stop the simulation.

LAMMPS also implements application-specific checkpointing of the simulation state to a file. The user specifies how often checkpoints are committed. To perform rollback recovery, the user starts LAMMPS with additional arguments specifying the checkpoint file to read from.

We ran LAMMPS with two different inputs, ‘‘eam’’ and ‘‘lj’’, which exercised different portions of LAMMPS code. These inputs are examples included in the LAMMPS software distribution.

During preliminary fault injection testing we observed that small perturbations of the state of the application (e.g. through CPU register bit-flips) often did not cause observable errors in the simulation results. This is reasonable because LAMMPS simulates every molecule, and the perturbation of the state of a single molecule among thousands of molecules is unlikely to impact the result of the simulation. However, we did observe rare occasions in which a fault injection would cause the simulation to be unstable and output values that were many orders of magnitude off from the correct values. Sometimes results of the floating point computations were even ‘‘NaN’’. Therefore we chose to implement error detection mechanisms that checked for large changes in the thermodynamic data across time steps.

The error detection mechanism we implemented works as follows: When summary thermodynamic data is computed, all processes receive the results instead of just the process of MPI rank 0. Processes compare the results to those of the previous time step. If any value differs from the previous time step's value by more than a factor of 10 (larger or smaller), the process signals an error. For molecule counts, the process signals an error if there is any change to the molecule count.

The checks on the variation of the thermodynamic data are admittedly coarse, made with the assumption that sudden changes in the data from one time step to another are errors. It is conceivable that some simulation inputs correctly cause thermodynamic data to change wildly from time step to time step, and this behavior would cause the error checking to produce false error detections. Developing more intelligent error detection mechanisms that work for all possible LAMMPS input would require a deeper understanding of LAMMPS algorithms and molecular behavior.

We also modified LAMMPS to time the simulation of each time step. Since the amount of computation and communication does not vary across time steps, we implemented conservative timeouts to detect hung processes. We modified LAMMPS to time the first five time steps for computing the average time for one time step. For the remaining time steps, a timeout was set conservatively at ten times the average.

We modified LAMMPS so that processes detecting a hang or erroneous value broadcast a G-MPI signal with an application-specified signal number. Processes handle the delivery of this signal by raising the G-MPI alert flag (see Subsection 4.2.2). Handling of hung processes follows the procedure described in Section 4.3. It is possible that the diagnosis fails to identify any processes as hung, or otherwise faulty. In that case, since no component is

identified as faulty, no reconfiguration is done and all the processes of the application perform rollback recovery.

We modified the LAMMPS checkpointing facility to write checkpoint data to the process's memory instead of to disk. We wanted to avoid writing checkpoints to disk because in our virtualized environment (see Section 5.3) multiple virtual cluster nodes share one physical disk. Having multiple processes writing checkpoint data to the disk would have introduced significant delay due to resource contention on the disk. However, since memory is not stable storage (checkpoint data is lost if the application process crashes), we programmed each process to store its checkpoint data to its own memory and to send a copy of it to one other application process via MPI message-passing.

The changes we made to implement error detection and recovery mechanisms for LAMMPS added 636 lines of code to LAMMPS' original 108,993 lines of C++ code.

5.1.2. ClustalW-MPI

ClustalW is a bioinformatics application for producing multiple sequence alignments [Thom94]. ClustalW-MPI is an MPI parallelization of ClustalW [Li03]. Unlike LAMMPS, ClustalW-MPI is an integer code. ClustalW-MPI performs multiple sequence alignments in three distinct phases. For all of our runs we use a set of sequences from the BioBench benchmark suite [Alba05] containing 430 protein sequences of an average of 350 symbols in each sequence. In the first phase, all of the sequences to be aligned are read in from a file and an alignment and difference score is computed for every pair of sequences (pairwise alignment). The pairwise alignment and computation of the difference score is an embarrassingly-parallel process because alignment of each pair of sequences is independent

from alignment of every other pair. In ClustalW-MPI, one manager process assigns pairs of sequences to be aligned to many worker processes.

To detect erroneous values in the first phase, we added a routine in the manager process to check the worker processes' alignments and difference scores. The manager process uses its sequence data and a worker process's alignment to recompute the difference score. If a recomputed difference score does not match the score returned by the worker, then a corruption has either affected the manager's ability to compute the difference score, or the worker's ability to produce a valid alignment and difference score. The time to redundantly compute the difference score in the manager is $O(M + N)$ where M and N are the lengths of the two sequences and $M + N$ is the maximum possible length of the alignment. This additional cost is reasonable considering that the time for the worker to align the pair of sequences is $O(MN)$.

In the second phase of ClustalW-MPI, a neighbor-joining clustering algorithm is used on the difference scores to construct a binary ‘‘guide tree.’’ The guide tree is used to order the multiple alignment in the third phase.

The second phase is not parallelized and executes in a very short amount of time on the manager process—less than 1% of the total execution time of ClustalW using our test input. Therefore, rather than design a fast check of the results of this phase, we modified ClustalW-MPI to redundantly execute the clustering algorithm in one worker process. The manager process and worker process then compare the trees formed to detect errors in this phase.

In the third and final phase, alignments are progressively performed according to the order indicated by the guide tree, from the leaves to the root. Depending on the shape of the

tree, many of the alignments can be performed in parallel. The third phase is similar to the first phase in that workers independently align sequences in parallel and return the alignment and its alignment score to the manager.

For detection of erroneous state in the third phase, we again added a routine to the manager process to check worker processes' alignments be recomputing the alignment scores.

Hang detection is implemented in each of the three phases. In the first and third phase, the manager schedules timeout events for receiving results from worker processes, and worker processes schedule timeout events for receiving work units from the manager process. The second phase also uses timeout events between the manager process and the worker process duplicating the manager's job.

For error recovery, we implemented application-specific checkpointing of the manager process's state. Worker process's state is not checkpointed because their state can be recomputed by having the manager process resend the worker's work unit. Because the bulk of the execution time (more than 98% for our input) occurs in the third phase, we implemented checkpointing only during the third phase. The checkpoint data consists of the guide tree and the results returned by workers in the third phase.

Our implementation of fault tolerance mechanisms for ClustalW-MPI added 1059 lines of code to ClustalW-MPI's original 17,932 lines of C code.

5.1.3. Nqueens and TSP

Nqueens (eight queens puzzle using n queens on an $n \times n$ chess board) and TSP (traveling salesman problem) are message-passing applications that employ divide-and-

conquer algorithms to find solutions in an embarrassingly-parallel manner. In both applications, a single manager process assigns independent work units containing subproblems to worker processes. Worker processes solve the subproblem and return their results to the manager process. Nqueens and TSP are ‘‘toy applications’’ in the sense that they were created to use memory, compute cycles and networking bandwidth, and they do not use optimal algorithms to solve the problem at hand.

Nqueens uses a depth-first search algorithm. The manager process places the first m queens, $m < n$, on a board and then sends the board to a worker process, which continues the depth-first search to find where the remaining $n - m$ queens should be placed. Upon finding any solution, the worker process sends it to the manager process. When application-specific error detection is enabled, the manager checks that the worker process's solution is valid, that is, the board does not have any queens sharing the same row, column, or diagonal. For all of our runs, we use $n = 15$ and $m = 5$.

TSP uses a naive dynamic programming algorithm to search all paths to find the shortest path for n cities. A work unit consists of a partial path involving m cities where m varies from 0 to $n - 1$. The worker process returns all partial paths involving $m + 1$ cities whose lengths are less than the length of the shortest path found so far. Because each work unit consists of very little computation, the performance of TSP is communication-bound, with a communication bottleneck at the manager process. In our runs of TSP, $n = 13$.

For application-specific error detection, we implemented checks in the manager process on the validity of the paths returned by a worker process. Specifically, the manager process verifies that the number of cities in the path was $m + 1$ and recomputes the length of the path. It also checks that the first m cities match the subpath that is originally sent to the worker

process. If any of these tests fail the path is rejected and the work unit is assigned to another worker process.

We implemented hang detection for both TSP and Nqueens by having the manager and each worker process initially measure the time interval between communications. A timeout is conservatively scheduled at ten times the measured interval. If a manager fails to receive solutions in this amount of time, or a worker fails to receive a work assignment, the process broadcasts an application-defined G-MPI signal. Handling of the hang detection proceeds as described in Section 4.3.

We also implemented application-specific rollback recovery by checkpointing the state of the manager process in both TSP and Nqueens. The state of worker processes is not checkpointed because its state can be recomputed by having the manager resend the work unit. The manager process's checkpoint is stored in its local memory and in the memory of one worker process, transmitted via MPI message-passing.

Our implementation of application-specific fault tolerance mechanisms for TSP added 544 lines of code to TSP's original 450 lines of C++ code. Our implementation the same for Nqueens added 574 lines of code to Nqueen's original 1,824 lines of C code.

5.2. Implementing Application-Transparent Fault Tolerance

As discussed in Chapter 3, our layered approach is based on an end-to-end design argument, with fault tolerance mechanisms implemented with application-specific knowledge. To justify the effort of implementing fault tolerance mechanisms at least partially in application code, we compared the layered application-specific approach to application-

transparent mechanisms. In particular, we compared the performance overhead of the application-specific mechanisms described in the previous section to application-transparent process replication for error detection and application-transparent checkpointing for rollback recovery. This section describes our implementation of these application-transparent mechanisms.

For approximating the cost of application-transparent error detection, we implemented application-transparent replication of processes using an MPI library we developed called RepMPI. RepMPI uses the MPI Standard's ''profiling interface'' to wrap all MPI calls for implementing additional functionality. Hence, RepMPI intercepts an application's calls to MPI and transparently replicates the application's message operations across process replicas. The wrapping facility of the profiling interface enables us to implement RepMPI without making any changes to the underlying G-MPI implementation.

RepMPI transparently designates half of an application's processes to execute as duplicates of the other half. Hence, for a non-duplicated task of n processes, RepMPI requires that the task be started with $n \times 2$ processes. RepMPI ''virtualizes'' the MPI rank assignment so that, from the point of view of the application, each pair of processes shares the same logical MPI rank assignment. (From the point of view of the underlying G-MPI library, every process still possesses a unique MPI rank.) For the remainder of this section, we use the word ''rank'' to refer to the application's logical non-replicated view and the word ''process'' to refer to a replica of the rank.

When one process of rank i sends a message to a process of rank j, RepMPI transparently delivers the message to both processes representing rank j. Similarly, to complete a message receive operation, a process with rank j must receive identical messages

from both processes representing the sender with rank *i*. RepMPI implements only point-to-point communication operations; collective communication operations were not implemented.

RepMPI's communication protocol for duplicate processes is very simple. One process of each pair is designated as the primary process and the other as the backup process. When process with rank *i* sends a message to process *j*, both instances of process *i* send the message to the primary of process *j*. The primary process compares the message contents and then forwards a copy of the message to the backup process. Both the primary and backup processes send an acknowledgment to the primary process of the sender. The primary process of the sender compares the two acknowledgments and forwards one copy to the backup process.

Since we did not implement with RepMPI an application-transparent recovery mechanism, the semantics of MPI calls in RepMPI after an error is detected echo the semantics of the original MPI Standard, that is, the state of the RepMPI communication context is undefined. Specifically, when the primary process's comparison of two replicas' messages fails, the primary process broadcasts a G-MPI signal to indicate the error. RepMPI's handler for this signal invalidates the RepMPI communication context so that all of the process's communication calls to RepMPI return MPI_ERR_OTHER. With message-passing thus disabled, application processes must perform any application-specific cleanup actions (e.g. flushing I/O buffers) and then terminate.

RepMPI's communication protocol is overly simple in that it allows a primary process to be faulty in a way that is undetectable by other processes. Specifically, the primary process can misrepresent to the backup process the result of its comparison of two message replicas. It can drop messages that it receives. It can also corrupt the sending process's messages

before forwarding them to the backup process. Fully solving these problems would require use of a much more complex group communication protocol. Hence, a full correct implementation of a mechanism like RepMPI is likely to have a higher overhead than RepMPI. However, for completeness, it should be noted that there is a simple optimization of RepMPI that we have not implemented: having the backup process send a digest of each message instead of the entire message.

For application-transparent error recovery, we modified an existing checkpointing software called Distributed MultiThreaded Checkpointing (DMTCP) [Anse09]. The original version of DMTCP caused a distributed application to stop and commit a coordinated checkpoint. Each process's stack, heap, and writable mapped memory was committed to a file on disk. To make DMTCP checkpointing comparable to our applications' in-memory checkpoint, we modified DMTCP to write the checkpoint to memory and send a copy of the checkpoint to another application process.

5.3. Experimental Setup

This section describes the hardware and software configuration of the clusters used for testing and validation. This validation infrastructure has four main components: (1) a virtualized cluster, using virtual machines for compute nodes; (2) a software fault injector; (3) a fault injection campaign manager; and (4) a facility for logging information for postmortem analysis. Each of these components is described in the subsections below.

5.3.1. A Virtualized Distributed System

Since validation requires subjecting the system to a large number of faults, a meaningful validation campaign must execute for long durations. Hence, it is desirable to run multiple campaigns simultaneously on multiple distributed system instances. This may be impractical in many environments if each distributed system is actually composed of multiple physical computers. This issue provides part of the motivation for using system-level virtualization technology to run multiple nodes of the distributed system under test on a single physical computer. In particular, for the experiments we report here, the entire distributed system is consolidated on a single physical host.

Virtualization technology allocates a computer's resources, such as CPUs and I/O devices, to multiple virtual machines (VMs). Each node of the distributed system runs as a VM, and each VM runs its own OS and user-level software. A virtual machine monitor (VMM) enforces isolation among the VMs so that the activities of one VM do not affect other VMs. We use the Xen VMM [Barh03], which gives direct access to the physical computer's hardware devices to one privileged VM. The privileged VM can start, halt, or shut down unprivileged VMs. It also hosts disk images for the virtual disks in each unprivileged VM and routes the network traffic among the VMs and with other computers outside the virtualized environment.

In addition to reducing the required hardware resources, running nodes in VMs provides three main benefits over using physical machines. First, it enables implementing fault injection software partially or completely outside of the VM, minimizing intrusion on the system under test. Second, nodes implemented as VMs can be easily and quickly power-reset without specialized hardware. Finally, the virtualized environment can provide a lightweight

communication facility among VMs through shared memory that can be used for coordinating fault injections and for logging system responses.

5.3.2. Fault Injection in a Virtualized Environment

To be useful, a fault injector must be flexible in the types of faults that can be injected, the times at which faults can be injected, and the targets where faults can be injected. We have developed *Gigan* [Le08], a flexible SWIFI capable of injecting a variety of faults into OS kernels and user-level processes. *Gigan* can operate in non-virtualized systems but also has capabilities optimized for virtualized systems. *Gigan's* operation is based on *triggers* and *actions*. *Triggers* are set to fire after some threshold has been reached or some event has occurred in the target machine. Triggers can fire based on timers, instruction breakpoints, process creation/termination, and performance monitoring events (e.g., CPU cycle count) [Carr98]. Associated with each trigger is a set of one or more *actions* to be performed at the time the trigger is fired. To maximize flexibility, an action either injects a fault or sets another trigger.

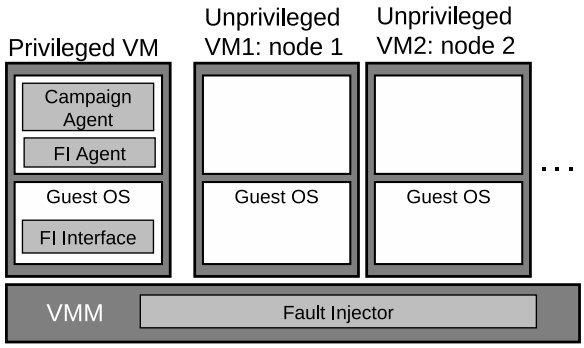


Figure 5.1: Architecture of VMM-level injector

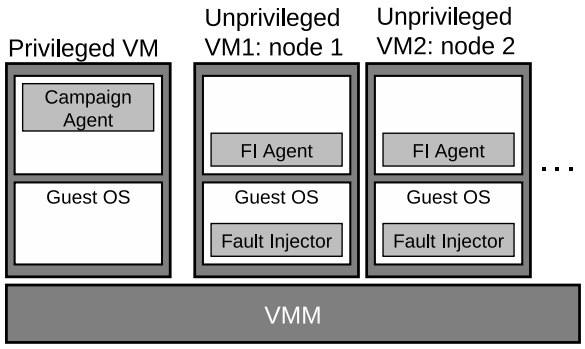


Figure 5.2: Architecture of OS-level injector

The purpose of a fault injection campaign is to validate system operation under some well-defined fault model. For example, the goal may be to validate correct operation as long as, within a period of **T** seconds, no more than **k** nodes in the system operate erroneously. Hence, for a particular experiment, the injector must not inject additional faults once the worst-case scenario being tested is reached.

Gigan implements two approaches to fault injection: the VMM-level injector (Figure 5.1) and the OS-level injector (Figure 5.2). With the first approach, the injector is implemented completely outside of the VMs. This implementation has four components: the *Fault Injector* in the VMM, the *Fault Injector Agent (FI Agent)*, the *Fault Injector Interface (FI Interface)*, and the *Campaign Agent*. The Campaign Agent tracks the progress of the campaign, gathers information about the state of the system under test, and instructs the FI Agent to pause or resume fault injections. The FI Agent creates the triggers and actions as specified by the fault injection campaign. At the beginning of each test, the FI Agent sets triggers and actions by sending commands to the Fault Injector via the FI Interface. When a trigger fires, the Fault Injector executes the associated actions.

With any fault injection, it is desirable to minimize *intrusiveness*, i.e., minimize the impact of the fault injection on the behavior of the target system. The VMM-level injector can operate without any knowledge of the internal structure of a VM running as a node in the system under test. Furthermore, it does not require any changes to such VMs and has essentially no impact on their normal operation. However, treating the VM as a ‘‘black box’’ does not allow targeting specific user-level processes or OS data structures within the VM. The only exception to this opacity is that the Fault Injector can distinguish between the VMs execution of processes and OS-level code. This exception is due to the fact that the VM OS runs in a higher hardware privilege level than the user-level processes, and the act of switching privilege levels is visible to the VMM. The VMM-level injector can use this visibility to target the VM OS by injecting faults only when the CPU is executing at the higher privilege level.

To target user-level processes, a VMM-level injector would need to be able read and parse data structures of the guest OS, such as process tables. Rather than add such complexity and OS dependency to the VMM-level injector, *Gigan* implements a second approach to fault targeting, the OS-level injector (Figure 5.2). The OS-level injector consists of an FI Agent running inside each VM to be targeted, the Fault Injector as a module in each VMs guest OS, and the Campaign Agent running in the privileged VM. As a user-level process, the FI Agent can invoke services of the guest OS to gather information on the state of user-level processes and use this information to set triggers and actions. Triggers and actions are set via system calls to the Fault Injector module. When a trigger fires, this module executes the associated actions. As with the VMM-level injector, the Campaign Agent signals the FI Agents to pause and resume fault injections. Because the FI Agent runs only to set triggers and actions and

the Fault Injector is active only when triggers are fired, there is negligible intrusion by these components on the operation of the VMs.

OS-level FI Agents must be coordinated so that fault injections into the target VMs occur as specified by the fault injection campaign. With *Gigan's* OS-level FI Agents, coordination is based on synchronized clocks and network communication. The Network Time Protocol (NTP) is used to keep the wall clocks of the VMs synchronized. In some cases, the FI Agents on the different VMs are assigned different time slots during which they can perform injection. In addition, the Campaign Agent can command FI Agents to pause or resume injection using user-level signals sent via SSH connections. The virtualized environment does enable an alternative method of coordination with potentially lower intrusion: processes on different VMs can communicate using memory pages that are shared among VMs. We chose to use time for coordination over the lower-intrusion shared memory design for two reasons. First, NTP and user-level signals use very few system resources such that the gain of even lower intrusion does not justify the added complexity of sharing memory between VMs. Second, the OS-level FI agents as designed are portable to a validation infrastructure that uses physical machines instead of VMs.

5.3.3. Injection Campaign Management

In order to be able to interpret the results of the injection campaign and use them to correct flaws in the system, each experiment (injection) must be performed starting with a fault-free system. Specifically, latent erroneous state from one injection must not be allowed to ‘‘contaminate’’ the results of the next injection. Thus, in order to maximize the speed of an injection campaign, the validation infrastructure should restore the system to a fault-free state

as quickly as possible.

Using SWIFI, the *hardware* of the system under test cannot be permanently affected. Hence, a reboot (or power reset) of a node removes any faulty state in volatile memory (CPU registers and memory). For a distributed system running on physical machines, issuing a reboot command from within the faulty node may not work because the node could be crashed or hung. Using VMs instead of physical machines enables ‘‘power-cycling’’ the nodes without the use of special hardware (e.g. Intelligent Platform Management Interface).

Power-resetting a VM is not sufficient to repair a faulty node because fault injections can cause a node to write erroneous data to disk. The validation infrastructure must isolate and remove such erroneous disk state between experiments. Furthermore, erroneous disk state may need to be saved for later analysis to assist in debugging the tested system. A possible way to meet these requirements is to move the VMs old disk image to a safe location and make a copy of a ‘‘golden’’ uncorrupted disk image for the VMs new disk image. Since VM disk images can be quite large (hundreds of MB), copying an entire disk image would introduce long delays between experiments.

To reduce the delay of restoring disk state, the validation infrastructure uses a union mount inside each VM [Okaj06]. For each unprivileged VM, the privileged VM hosts two disk images: a read-only image containing the unprivileged VMs root file system, and a read-write image for the union mount’s branch file system. The union mount of the root and branch file systems is performed by the guest OS at boot time. Subsequently, all writes to the root file system are redirected by the guest OS to the branch file system. After halting an unprivileged VM, the Campaign Agent moves the image containing the branch file system to another directory for storage and later analysis, and a new empty image is copied in its place.

The Campaign Agent then commands the VMM to power-reset the unprivileged VM.

The use of union mount involves guest OS support and is thus somewhat intrusive. However, the benefit gained by this intrusion is a significant reduction in the time spent restoring disk state before booting a VM. This is due to the fact that the size of the branch file system can be small relative to the size of the root file system. For example, the size of the root file system of each Ghidrah cluster node was 2.5GB while the size of the branch file system was limited to 50MB. Copying the 2.5GB image took up to one minute, while copying the 50MB image took at most one second.

5.3.4. Logging for Postmortem Analysis

A key goal of any fault injection campaign is to identify and correct flaws in the system's fault tolerance mechanisms. In support of this goal, the infrastructure must provide a mechanism for collecting detailed functional and timing information regarding the actions of the injector as well as the resulting actions of all the components of the system under test. As diagnosis of a flaw often requires focusing the information collected to particular components of the system, this mechanism must be configurable to easily include or exclude a variety of information sources.

To meet the above requirements, our validation infrastructure records timestamped information logged by three components: the location of fault injections, logged by the fault injector; information related to detection of and recovery from errors, logged by user-level processes running on the nodes (VMs) of the system under test; and, for each node of the system under test, usage of CPU cycles and memory of all the processes, as well as process creation and termination events, logged by a system *resource monitor* running as a user-level

process on each node. The node *resource monitor* is useful for diagnosing problems such as hung processes and memory leaks.

Each VM runs a user-level *log client* process which receives logs from various processes on that node via named pipes and sends them over the network to the *log server* on a remote physical machine. The *log server* writes the logs to its local file system for later analysis.

There is a possibility that the *resource monitor* and logging activities on each node will impact the operation of the system (undesirable *intrusiveness*). With virtualization, less intrusive resource monitoring could be implemented from the VMM. This would require significant additional complexity in the VMM — an ability to access and parse internal data structures of the OS kernel of the VMs. Additionally, instead of using *log clients* to transmit logs to the *log server*, virtualization could be exploited for lower intrusion by using shared memory pages between each of the unprivileged VMs and the privileged VM. During the design and later experimentation with our infrastructure we determined that, for our purposes, the intrusiveness of the implemented mechanisms was negligible and did not justify the added complexity of the alternative implementations. Furthermore, the present resource monitoring and logging mechanisms are portable to a distributed system running on physical machines.

5.4. Validating Ghidrah CMM

This section describes the validation of Ghidrah CMM fault tolerance. Although the focus of our work is application fault tolerance, the applications depend on the reliability of the CMM for correct execution. Therefore we used a variety of fault injection campaigns on the CMM. The fault injections exposed a number of implementation faults (flaws) in Ghidrah CMM. These flaws were fixed before running fault injection campaigns involving fault-

tolerant applications.

5.4.1. Design of Fault Injection Campaigns

A *fault injection campaign* is a set of fault injections that have common properties, such as fault injection type, target, and timing. After an injection is performed, the system under test is allowed to run for a defined period of time to observe whether the injected fault caused an error, whether the error was detected by the system, and whether the system recovered from the detected error. The campaign may also direct when and how the system is to be restored to fault-free conditions in preparation for subsequent injections.

Campaign		**Campaign Description**		
		Time slot (s)	**Selection of target node**	**Injection target**
General	1Pf	60	Round-robin	One randomly selected CMM process
	2Kf	60	Random	OS kernel
	3Kf	15	Random*	OS kernel
Ghidrah-specific	4Pf	60	Round-robin	Two randomly selected CMM processes
	5Pf	60	Round-robin	Two randomly selected CMM processes
	6Kf	60	Node not running *Manager Replica*	OS kernel
Flaw-specific	7Pt	30	Random	*Manager Replica*
	8Pt	30	Random	*Agent*

Table 5.1: Descriptions of CMM fault injection campaigns. The campaign name indicates the injection target and fault type: the first letter is the target, **P** for user-level **p**rocess or **K** for OS **k**ernel; the second letter is the fault type: **f** for bit-**f**lip or **t** for process **t**ermination. *Target node selection changed only after the target node is power-reset.

To validate Ghidrah and its fault tolerance mechanisms we designed three kinds of fault injection campaigns: 1) *general campaigns* that injected faults with minimal regard to the

design of Ghidrah, 2) Ghidrah-*specific campaigns* that injected faults based on the design of Ghidrah, and 3) *flaw-specific campaigns* that injected faults designed to reproduce conditions known to activate flaws that were exposed by the first two kinds of campaigns. Table 5.1 summarizes all of the campaigns. The general campaigns were useful in exposing a number of flaws in the Ghidrah CMM. Because the Ghidrah-specific campaigns used injections specific to the design of Ghidrah, they were able to expose more flaws that were very unlikely to be exposed by the general campaigns. Many of these flaws were exposed only after tens of hours of fault injections.

For the campaigns involving bit flips, the injected fault was a single bit flip into a randomly selected general purpose register of the x86 architecture. The selection of the bit of the register to flip was also random. Single bit flips were used because they have been shown to best capture the effect of hardware transient faults caused by particle strikes to the system [Cha96]. Since x86 processors have relatively few registers, each of these registers is frequently used. Hence, a large fraction of faults injected into registers are manifested as errors. Thus, fault injection into registers resulted in accelerated stressing of Ghidrah's fault tolerance mechanisms. Fault injection into memory was not performed because memory corruption has been shown to have a low error manifestation rate [Le08, Sieh02].

The timing of fault injections was varied in order to introduce errors at different points of execution of the system. For each campaign, time was divided into periodic slots (see Table 5.1), with one injection performed per time slot. The injection trigger was scheduled to fire after a random interval from the beginning of the time slot up to half of the time slot duration. For injections of bit flips in registers that targeted a specific process or the OS kernel, the action for the time-based trigger was not the actual injection. Instead, the action

was to set a second trigger to fire after a randomly-selected number of instructions, up to 5000 instructions, were executed by the target process or OS kernel. The second trigger's action was to perform the bit flip.

In all of the campaigns, the Campaign Agent read the SCC log so that it could power-reset nodes on behalf of the SCC. Ghidrah was designed to handle only a single node failure within the time it takes to detect and recover from such failure. Hence, as discussed in Subsection 5.3.2, the injection infrastructure had to refrain from additional injections while the node reset was in progress. Therefore, the Campaign Agent paused all fault injection during an SCC-requested node power-reset.

None of the campaigns prevented faulty state in the target node from being propagated to fault-free nodes (e.g. via a corrupt message sent to a fault-free node). However, Ghidrah was designed to detect and handle messages with corrupt payloads. Thus, it was not surprising that our experiments did not expose any fault propagation to fault-free nodes. Separate work in [Li02] performed fault injections on message payloads to validate Ghidrah's fault-tolerant communication protocols.

There were two general campaigns: Campaign 1Pf used the OS-level injector to target CMM processes; Campaign 2Kf used the VMM-level injector to target the guest OS. As discussed in Subsection 5.3.3, a node would ideally be rebooted after each injection in order to ensure a fault-free state before the next injection. However, in order to accelerate the experiments, this was not done with these campaigns. This shortcut had the potential to produce incorrect results — failure of the system under test due to a latent error from a previous injection coupled with the current injection manifesting as two simultaneous node failures. In our runs of Campaigns 1Pf and 2Kf, this circumstance did not occur because

faulty state either caused an immediate error or was overwritten before an error could be caused.

Campaign 3Kf was designed to decrease the possibility of simultaneous errors in multiple nodes while, at the same time, accelerating fault injections into the guest OS. In Campaign 3Kf, a target node was randomly selected, and it remained the target until it failed in such a way that the SCC requested it to be power-reset. Subsequently, a new target node was selected for the next injections. Thus, a single node could be targeted for multiple injections without clearing faulty state between injections, but at no time was there more than one node subject to fault injection.

The Ghidrah-specific campaigns stressed Ghidrah based on knowledge of its design. Campaign 4Pf injected bit-flip faults as Campaign 1Pf, except that in each time slot an injection was performed in two randomly selected CMM processes on a single target node. Like Campaign 1Pf, targeted processes were not terminated after each fault injection experiment to remove faulty state. In some cases, this did cause a fault injected in one time slot to be manifested as an error in a later time slot. When this impacted two *Manager Replicas* simultaneously, Ghidrah responded correctly, with the SCC resetting the entire cluster.

Demonstrating the flexibility of the infrastructure, Campaign 5Pf was a modification of 4Pf that eliminated the incorrect propagation of the effects of one injection to the following injection, without requiring a time-consuming node reboot. 5Pf was similar to 4Pf except that the targeted processes were explicitly terminated near the end of the time slot. Ghidrah detected the terminated processes and automatically started new processes to replace them. Thus, the system was quickly restored to a fault-free state by the time the next time slot

began.

Campaign 6Kf stressed Ghidrah's replica recovery procedure by attempting to cause the starting of a new *Replica* to fail. In this campaign, a bit-flip fault was injected into the OS of a node *not* running a *Replica*. This could crash or hang the node or potentially introduce latent erroneous state in the node. In the case where the target node did not hang or crash in the 15 seconds following the injection, the campaign caused the system to attempt to move a *Replica* to the target node. This was done by terminating a *Replica* process on another randomly-selected node, forcing the *Manager Group* to attempt to start a new *Replica* on the target node. If the target node failed to start the *Replica*, the *Manager Group* was expected to attempt to start a new *Replica* on another node.

The general and Ghidrah-specific campaigns exposed a number of flaws that were activated by CMM processes crashing. Campaigns 7Pt and 8Pt were flaw-specific campaigns designed to quickly reproduce these flaws by explicitly terminating CMM processes. Campaign 7Pt's injection was to terminate the *Replica* on the target node; Campaign 8Pt's injection was to terminate the *Agent* and *Agent-Keeper*.

5.4.2. Ghidrah Validation Results

The Ghidrah CMM fault injection campaigns were run on a virtualized system on a dual-socket quad-core Intel Xeon system (a total of eight cores). The VMM was Xen 3.3.0. A cluster of four nodes was run in the virtualized system. The SCC ran on a separate physical computer; this computer also used as a log client to send SCC logs to the log server. Instead of directly performing power-resets of cluster nodes, the SCC logged such actions. The logs were then read and acted upon by the Campaign Agent running on the privileged VM.

Campaign		Campaign Results			
		Total time (hours)	**# injections**	**# errors**	**# flaws exposed**
General	1Pf	422.1	23781	4943	6
	2Kf	92.6	4136	3000	2
	3Kf	76.6	7127	1458	0
Ghidrah-specific	4Pf	320.0	21403	13062	6
	5Pf	252.2	12574	12574	5
	6Kf	24.9	794	489	1
Flaw-specific	7Pt	63.5	7023	7023	1
	8Pt	30.3	1058	1058	1
Totals		1282.2	77896	43607	11

Table 5.2: Results of CMM fault injection campaigns. Note that some flaws were exposed by multiple campaigns, so the number of flaws exposed is not the sum of the rightmost column.

The results of the campaign runs are summarized in Table 5.2. Of the 43,607 fault injections that caused errors, 37 exposed a total of 11 unique flaws in Ghidrah. To qualitatively illustrate the number of injections required to expose these flaws, we define the concept of a *campaign run* as a sequence of injections beginning with the cluster starting with all nodes powering up and ending with the exposure of the flaw. The 37 flaw exposures are plotted in Figure 5.3 against the duration of the campaign run in hours. The large variability in the campaign run times are due to the randomness of injections, demonstrating the necessity of many injections. The figure also illustrates the need for varied fault types in order to cover the various parts of the system.

Flaws 1f, 2c, and 3c resulted in race conditions between normal timer events in the *Manager Group* and local events occurring in individual *Replicas* during the self-diagnosis and replica recovery procedures. Therefore, the campaigns that caused a *Replica* to fail were

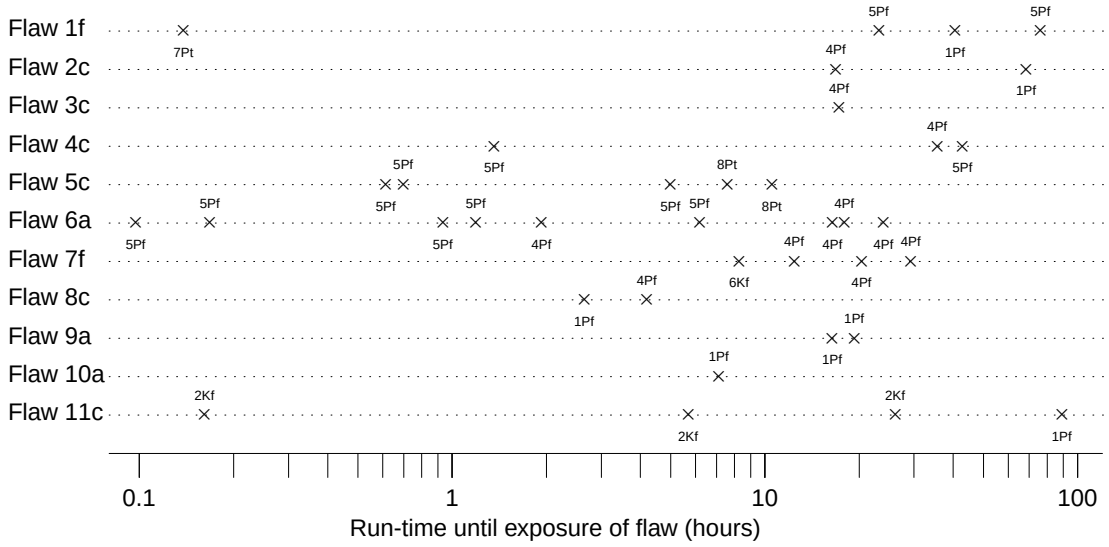


Figure 5.3: Run-times of the campaign runs that exposed flaws. Each marker is labeled with the campaign name.

more likely to expose these flaws. After these flaws were exposed by the Ghidrah-specific campaigns, Campaign 7Pt was designed to re-expose these flaws for debugging purposes. The first run of Campaign 7Pt ended with exposure of Flaw 1f in less than 15 minutes. The remaining 63 hours of Campaign 7Pt runs were performed after Flaws 1f, 2c, and 3c were fixed.

The Ghidrah implementation failed to take into account the fact that a halted *Agent* does not mean that all the other processes on that node have halted. Flaws 4c, 5c, and 6a resulted in Ghidrah processes improperly handling messages from a *Replica* on a node whose *Agent* and *Agent-Keeper* had terminated. With flaws 4c and 5c, this led to *Replica* crashes. Flaws 7f, 8c, 9a, and 10a resulted in errors in handling unexpected orderings of messages or events when CMM processes terminated or were restarted. It should be noted that these flaws were

exposed by different sets of campaigns across a wide range of campaign run-times.

Flaw 11c was unique among the 11 flaws because it was exposed primarily by injection into the OS in Campaign 2Kf. Fault injection caused the *Replica* on the target node to stop sending heartbeats to the other *Replicas* but to otherwise operate normally. This caused the other *Replicas* to repeatedly initiate self-diagnosis due to the missing heartbeats and to never detect an inconsistency in the replicated state of the *Replicas*. This flaw demonstrated the utility of injections targeting the OS even when the goal is to expose flaws in user-level programs.

Of all the components of the validation infrastructure, the logging mechanism had potentially the highest intrusion and performance overhead, depending on how much data was logged by the tested system. For most campaign runs, we ran the Ghidrah CMM with detailed logging enabled to facilitate debugging. During these runs, the four nodes combined logged an average of 5.8KB/s. The system logged more data while error detection and recovery procedures were being executed. Injections that caused errors resulted in bursts of logging activity, lasting up to 30ms, with a peak rate of 58KB/s.

5.5. Performance of Ghidrah Fault Tolerance Services

The fault injection campaigns described in Section 5.4 validated the fault tolerance mechanisms of the CMM itself. In contrast, this section presents fault injection campaigns used to trigger the fault tolerance services that the CMM provides to applications. As proposed in Section 3.4 the CMM's fault tolerance services consist of crash notifications sent by the cluster manager to application processes and services that application processes request of the cluster manager. This section describes the results of two fault injection campaigns

used to test the CMM's ability to detect process crashes and node failures and interrupt and notify the surviving application processes. This section also reports response times for CMM services requested by applications.

Fault	# of faults	CMM detection latency	Time from detection to broadcast signal	Time from broadcast signal to signal delivery	Total time
Crash of application process	613	68 (24)	9 (0)	10 (19)	87
Node failure, without manager replica	260	869 (105)	negligible	10 (17)	879
Node failure, with backup manager replica	435	1037 (109)	negligible	9 (13)	1046
Node failure, with primary manager replica	268	1052 (115)	negligible	10 (14)	1062

Table 5.3: Response time of CMM detecting and notifying applications of process crashes. All times are in ms; standard deviation is shown in parentheses.

To test Ghidrah CMM's process crash notification service, we used a four-node cluster executing LAMMPS with four processes. The fault injection campaign repeatedly chose one LAMMPS process at random to terminate, allowing LAMMPS to recover before choosing the next process to terminate. To test the CMM's detection of node failure, we used a fault injection campaign in which a randomly selected node was disconnected from the cluster network by disabling the node's network interface via Linux's `ifconfig` command. Disconnecting the node emulated the behavior of a fail-stop node failure. After the fault injection, the target node was rebooted to restore its original configuration.

Table 5.3 lists the results of both fault injection campaigns. The first row, ‘‘Crash of

application process,'' lists the results of the campaign that terminated application processes. and the second, third and fourth rows list the results for the campaign that disconnected nodes from the network. As will be explained below, the steps taken by the Ghidrah cluster manager to detect node failure differ slightly depending on whether the failed node was running a Manager Replica, and if so, whether the Replica was a primary replica or a backup replica. (See Sections 4.4 and 4.5 for a description of the Manager Replicas and their roles.) Hence, the second row, ''Node failure, without manager replica,'' reports the results of the node failure fault injection campaign for nodes that were not running a manager replica. The third row, ''Node failure, with backup manager replica,'' reports the results of the node failures for nodes that were running a backup manager replica. The fourth row, ''Node failure, with primary manager replica,'' reports the results of the node failures for nodes that were running the primary manager replica.

The second column of Table 5.3, ''# of faults,'' lists the number of fault injections performed for the row's fault injection campaign and category. The third column, ''CMM detection latency,'' represents the time from the fault injection until the first CMM component detects an error. The time is an average over all of the faults for that campaign and category. The fourth column, ''Time from detection to broadcast signal,'' represents the average time from the initial detection by the CMM component to when the cluster manager broadcasts notification signal to the application. In the case of the second fault injection campaign, the cluster manager is the first CMM component to detect the node failure, so the delay between the cluster manager's detection and its broadcast of the signal is negligible. The fifth column, ''Time from broadcast signal to signal delivery,'' represents the average time from the broadcast of the signal until the signal is delivered to the last application

process. The rightmost column, ‘‘total time,’’ is the sum of the second, third and fourth columns. The table also lists each measurement’s standard deviation in parentheses.

When a process crashes, the OS delivers a signal to the *Agent-Helper*, as described in Subsection 4.4.2. Hence the *Agent-Helper* is the first CMM component to detect process crashes.

As described in Subsection 4.4.1, the cluster manager detects node failures by timing out on heartbeats from *Agents*. Since *Agents* send their heartbeat messages to backup *Manager Replicas*, the backup *Replica* is the first component to detect an error due to node failure. The time it takes for the *Replica* to detect a node failure depends in part on how much time passes between the node failure and the next heartbeat check, which explains the variation in times for the CMM detection latency for node failures.

The CMM’s time to detect a failed node is affected by whether a Manager Replica was running on the failed node and whether the Replica was a primary or backup Replica, as described in Section 4.4. The last two rows of Table 5.3 show the increase in detection latency due to *Manager Replicas* performing self-diagnosis before probing the *Agent*. From these results, we can see that applications that use timeout events for hang detection will benefit from the CMM’s detection and notification of process crashes if the application’s timeout events are scheduled more than about 1 second apart.

The times presented in Table 5.3 show that the the CMM can promptly detect and notify application processes of crashed processes. The CMM’s detection latency for failed nodes is significantly higher than for process crashes, but much of the delay is due to the use of conservatively long timeout values. This latency can be reduced by tuning the timeout values

for the specific network and processor speeds that the cluster is running on.

Request	Time (ms)
Application-initiated broadcast signal	18
Group request: Synchronize	19
Group request: Terminate	42
Group request: Spawn, LAMMPS (12-MB executable)	112

Table 5.4: Response time of CMM for handling application requests

Table 5.4 lists the CMM response times for fault tolerance services that the application requests of the cluster manager (Section 3.4). The synchronize, terminate, and spawn services are all group requests, requiring a quorum of application processes to send matching requests to the cluster manager. The times shown in Table 5.4 do not include the amount of time the cluster manager waits to receive the requisitie number of matching group requests, since that time depends on the behavior of the application processes. Instead, the time shown is measured from the point in time that the cluster manager receives the last group signal of the quorum to the point in time that the cluster manager's response has been delivered to every application process.

5.6. Performance Overhead of Fault Tolerance Mechanisms

To measure the overhead of fault tolerance mechanisms, the applications were executed on virtualized clusters of 4 and 8 compute nodes without fault injection. Compute nodes ran in VMs, with two VMs per physical host. The physical hosts were quad-core Intel Xeon X3210 processors running at 2.13 GHz. Each core had 4 MB of L1 cache. The physical hosts were connected via Gigabit Ethernet LAN.

Application/Input	Stock (sec.)	With application-specific error detection (relative to Stock)	With RepMPI process duplication (relative to Stock)
LAMMPS/eam, 4 processes	36.50	1.01	—
LAMMPS/lj, 4 processes	56.60	1.00	—
ClustalW-MPI, 4 processes	109.11	1.15	2.40
ClustalW-MPI, 8 processes	79.65	1.21	—
Nqueens, 4 processes	202.69	1.04	2.68
Nqueens, 8 processes	117.98	1.09	—
TSP, 4 processes	28.20	1.04	2.59
TSP, 8 processes	30.64	1.05	—

Table 5.5: Application execution times with and without error detection. Times with error detection enabled are relative to stock time. Note that LAMMPS was not executed with RepMPI because it used MPI collective communications that were not implemented in RepMPI. Applications with 8 processes were not executed with RepMPI because it required a 16-node cluster.

Table 5.5 shows the performance overhead of application-specific error detection mechanisms and of applications running with duplicate processes under RepMPI. Only applications with 4 processes (duplicated by RepMPI to 8 processes) were tested due to the limited number of nodes in our clusters. This table shows that application-specific error detection mechanisms are very inexpensive. Furthermore, note that the table only compares execution times; it does not show that applications running with RepMPI use twice as many cluster nodes compared to the stock applications and applications using only application-specifc error detection mechanisms.

Applications were executed with application-specific checkpointing and with DMTCP application-transparent checkpointing. In both cases, checkpoints were scheduled so that five

Application/Input	Stock (sec.)	With application-specific checkpointing (relative to Stock)	With DMTCP aplication-transparent checkpointing (relative to Stock)
LAMMPS/eam, 4-proc.	36.50	1.03	1.26
LAMMPS/eam, 8-proc.	20.84	1.05	1.43
LAMMPS/lj, 4-proc.	56.60	1.07	1.40
LAMMPS/lj, 8-proc.	29.70	1.06	1.51
ClustalW-MPI, 4-proc	109.11	1.04	1.18
ClustalW-MPI, 8-proc	79.65	1.04	1.18
Nqueens, 4-proc.	202.69	1.07	1.19
Nqueens, 8-proc.	117.98	1.08	1.20
TSP, 4-proc.	28.20	1.00	1.07
TSP, 8-proc.	30.64	1.00	1.08

Table 5.6: Application execution times with and without checkpointing. Times with checkpointing enabled are relative to stock time.

checkpoints were committed, evenly spread across the execution time of the application. The application-specific checkpointing mechanism is described for each of the four applications in Section 5.1. Table 5.6 shows execution times for applications with and without checkpointing. The times shown for application-specific and application-transparent checkpointing are relative to the execution times of the stock (unmodified) applications without any checkpointing. For all applications, five checkpoints were scheduled per run, evenly spaced across the the application's execution time. The checkpoint sizes are shown in Table 5.7. The performance overhead for application-transparent checkpointing was many times higher than that of application-specific checkpointing because application-transparent checkpointing did not use application-specific knowledge to minimize the amount of state to be checkpointed.

Application/Input	Application-level (MB/process)	Application-transparent (MB/process)
LAMMPS/eam, 4-proc.	2.7	26.4
LAMMPS/eam, 8-proc.	1.3	24.1
LAMMPS/lj, 4-proc.	8.1	46.4
LAMMPS/lj, 8-proc.	4.0	31.8
ClustalW, 4-proc., manager	1.0	4.2
ClustalW, 4-proc., workers	0.0	8.8
ClustalW, 8-proc., manager	1.0	4.2
ClustalW, 8-proc., workers	0.0	8.8
Nqueens, 4-proc., manager	18.9	60.5
Nqueens, 4-proc., workers	0.0	10.5
Nqueens, 8-proc., manager	18.9	60.5
Nqueens, 8-proc., workers	0.0	10.5
TSP, 4-proc., manager	< 0.1	16.0
TSP, 4-proc., workers	0.0	11.8
TSP, 8-proc., manager	< 0.1	16.0
TSP, 8-proc., workers	0.0	11.8

Table 5.7: Size of application-specific and application-transparent checkpoints.

5.7. Evaluation of Application-Specific Fault Tolerance Mechanisms

This section describes the fault injection campaigns used to evaluate the application-specific fault tolerance mechanisms of the four MPI applications described in Section 5.1. Table 5.8 lists five general outcomes for faults affecting application processes. Since all four applications use the same detection and recovery mechanisms for process crashes and hangs, we cover these errors for all applications in subsection 5.7.1. Results for faults causing

corruption of application-level state are broken out by application in Subsections 5.7.2 through 5.7.4.

Outcome of fault injection	Description
Process crash	The fault caused an application process to crash (terminate abnormally, e.g. due to a segmentation fault).
Process hang	The fault caused an application process to hang (stop making progress at the application level).
Corruption of application-level state: Detected	The fault manifested as an error in application-level state that was detected by application-level error detection.
Corruption of application-level state: Undectected error	The fault manifested as an error in application-level state that was not detected by the application; the application executed to completion but outputted incorrect results.
Not manifested	The fault had no effect on the application.

Table 5.8: Possible outcomes due to CPU register fault injection

5.7.1. Process Crashes and Hangs

To test the ability of the applications to detect and recover from process crashes and hangs, we used fault injection campaigns that directly caused these errors. For process crashes, the fault injection campaign terminates a random application process at a random time, with one process termination per application execution. For process hangs, each application was instrumented with a signal handler that enters an infinite loop that does nothing. Thus, the delivery of the signal instantly causes the process to stop executing application code. The fault injection campaign signals an application process at a random time, with one signal per application execution.

As mentioned in Section 4.3, all of our applications use timeout events to detect hangs. When a timeout event fires in one process, the process broadcasts a Ghidrah signal to all processes of the task. This begins a sequence of steps shown in Table 5.9. The first step is to diagnose the hung process; we use a conservative 4-second timeout to distinguish between a slow process and a hung process. The second and third steps in Table 5.9 are to terminate the hung process and spawn a replacement process. The fourth step is to recover the MPI communication context, incorporating the newly-spawned process into a new MPI world communicator. The fifth step is to rollback application state to the latest application-specific checkpoint. The final action is to recover additional application state, such as intermediate values based on the data read from the checkpoint, after which normal computation resumes.

At least 200 fault injections involving hanging an application process were performed on each application. All of the hung processes were detected and correctly recovered from in all of the applications. The amount of time taken in each of the diagnosis and recovery procedures is listed in Table 5.9. The results show that the CMM services invoked by the applications are performed very quickly. Most of the recovery time involves application-specific diagnosis and recovery of application-level state.

At least 200 fault injections that crashed an application process were performed on each application, with one process crash per application execution. The CMM detected the process crashes and notified the application, and all applications responded to the notifications correctly and recovered successfully. The average times for each recovery step were the same as those reported for process hangs in Table 5.9, except that the first two steps, diagnosis and process termination, are not performed for process crashes.

Diagnosis & Recovery Step	LAMMPS/eam	LAMMPS/lj	ClustalW-MPI	Nqueens	TSP
1. Diagnosis	4009	4008	4009	4009	4008
2. Process termination	42	42	42	42	42
3. Process spawning	113	113	74	71	71
4. MPI communication context recovery	23	23	23	23	23
5. Restore checkpointed state	219	792	580	3254	53
6. Additional application setup	754	733	10	12	12
Total	5160	5711	4738	7411	4209

Table 5.9: Time to diagnose and recover from a hung process. All applications were executed with 4 processes. All times are in ms.

5.7.2. LAMMPS Application-Specific Error Detection and Recovery

LAMMPS spends much of its execution time in floating-point arithmetic; therefore we targeted the fault injection on floating-point registers, namely the eight 128-bit XMM registers of the x86/SSE architecture. Because the simulator works on the level of molecules, a single bit-flip in an XMM register (to simulate an SEU) is likely to affect only the behavior of a single molecule in a simulation of hundreds of thousands of molecules. In preliminary testing, flipping a single bit in one XMM register very rarely caused an observable effect on LAMMPS.

In order to perform many tests on the application-specific error detection and recovery mechanisms of LAMMPS, we needed faults to introduce some values that are significantly different from correct values. We designed the LAMMPS fault injection to overwrite the entire 128-bit content of one XMM register with a random value. In each run of LAMMPS, this fault injection was performed once at a random time. The fault injections were timed to

occur close to a point where floating-point instructions were being executed. We used *Gigan's* performance counter triggers to count a random number of floating-point instructions retired, from 10 to 50 instructions, followed by a random number of integer instructions retired, from 1 to 200 instructions, at which point the floating-point register's state was changed.

We performed the same type of fault injections described in the previous paragraph on an unmodified version of LAMMPS, without our additional fault tolerance mechanisms, to observe the fault manifestation rate and the kinds of errors produced. For each run, we compared the thermodynamic values of the final time step to the values produced when no faults were injected. When faults were manifested, the errors present in the thermodynamic values were either very similar to the correct results, differing only in the 5th significant figure or lower, or they were very different from the correct results. Different results included missing molecules, values that were millions of times greater than the correct values, and values reported as ''infinite'' or ''not a number''. We did not observe any erroneous values that fell between these two extremes. Table 5.10 summarizes the results of these fault injections.

Application/Input	**# fault injections**	**# grossly incorrect results (% of faults)**	**# slightly incorrect results (% of faults)**
LAMMPS/eam, 4-proc.	306	46 (15%)	25 (8%)
LAMMPS/lj, 4-proc.	322	13 (4%)	7 (2%)

Table 5.10: Fault injections and errors in LAMMPS without fault tolerance

Table 5.11 lists the number of fault injections performed (one per run) on the version of

Application/Input	# fault injections	# application-level detections (% of faults)		# undetected errors (% of faults)	
LAMMPS/eam, 4-proc.	4516	825	(18%)	489	(11%)
LAMMPS/lj, 4-proc.	4386	255	(6%)	225	(5%)

Table 5.11: Fault injections and errors in LAMMPS with fault tolerance

Application/Input	ratio of erroneous value to correct value: minimum, maximum			
	Temperature	Pairwise energy	Total energy	Pressure
LAMMPS/eam, 4-proc.	0.99993 1.00027	0.999996 1.000002	0.999987 1.000004	0.99993 1.0002
LAMMPS/lj, 4-proc.	0.999997 1.00009	0.999985 1.0000001	0.99996 1.0000001	0.99998 1.0025

Table 5.12: Ratios of incorrect results to correct results in LAMMPS undetected errors.

Diagnosis & Recovery Step	LAMMPS/eam	LAMMPS/lj
1. Error detection latency	176	282
2. Error notification	17	17
3. Unwind call stack	71	71
4. Diagnosis	81	81
5. Vote	23	23
6. Restore checkpointed state	219	792
7. Additional application setup	754	733
Total	1341	1999

Table 5.13: Performance of LAMMPS application-specific fault tolerance mechanisms. All times are in ms.

LAMMPS that employs the application-specific error detection mechanism described in Subsection 5.1.1, the number of faults manifested as errors detected by the application and the number of faults manifested as errors that were not detected by the application, resulting in incorrect output. Even with the more severe corruption of register state the majority of faults were still not manifested, probably due to the corrupted state never being read from the register. Also, none of the faults caused a process to hang or crash. This is not surprising since the floating-point computations in LAMMPS are not used for flow control or to address memory.

For the cases of undetected errors, the error in the results was exceedingly small. Table 5.12 shows the minimum and maximum ratios of the final erroneous thermodynamic values to the correct values. The extremely small errors imply that only very small errors remained undetected by the application-level error detection mechanism. In other words, the final results in the cases categorized as undetected failure are actually largely correct. This indicates that our application-level error detection mechanism is well-suited for these particular LAMMPS inputs and faults.

In all of the cases that the application detected the error, application-specific rollback recovery was performed correctly, and the application executed to completion with the correct results. Table 5.13 lists the average times of the error detection, diagnosis, an recovery steps for application-detected errors. The error detection latency is the time from the fault injection to the application-specific detection of an incorrect value. The second step is the broadcast of an application-specific signal for notification of the error detection. The third step includes execution up to an MPI call, receiving the MPI_ERR_ALERT error code, and throwing a C++ exception, causing the call stack to be unwound to LAMMPS' main function. The fourth step

is diagnosis for detecting any hung processes. Since no process is hung, all processes send a group request to vote to continue computation, the fifth step. The sixth step is to rollback the application state to the most recent checkpoint, and the seventh step is to perform additional application-specific computations to prepare for resumption of normal execution.

5.7.3. ClustalW-MPI Application-Specific Error Detection and Recovery

Sequence alignment involves many arithmetic operations on small integers and array addressing computations. We designed the ClustalW-MPI fault injection campaign to inject single-bit flips into a randomly-chosen integer general-purpose register of a randomly-chosen worker process while the worker was executing a recursive dynamic programming algorithm in the third phase of the multiple sequence alignment. To maximize the manifestation rate of faults, we targeted the recursive routine that implements the dynamic programming algorithm. Specifically, we used *Gigan's* breakpoint trigger on the first instruction of that routine. The fault injector chose a random time in the application execution to activate the breakpoint trigger. After reaching the active breakpoint, the fault injector waited for another random number of instructions to retire (between 5,000 and 100,000 instructions) before injecting the fault. The target x86 register was one of EAX, EBX, ECX, EDX, ESI, and EDI.

In a further effort to inject faults with a higher manifestation rate, we also ran a fault injection campaign in which all 32 bits of the chosen general-purpose register were replaced with random values. The timing of the fault injection was the same as the campaign described in the previous paragraph.

Table 5.14 summarizes the results of the two fault injection campaigns. Out of a total of 2,845 fault injections between the two campaigns, only about 5% of the faults were

Application	Fault inj. type	# fault inj.	# process crashes	# process hangs	# application-level detections	# undetected errors
ClustalW-MPI, 4-proc	Single bit-flip	2005	30 (1.5%)	57 (2.8%)	2 (0.1%)	0
ClustalW-MPI, 4-proc	Replace value	840	41 (4.9%)	21 (2.5%)	2 (0.2%)	2 (0.2%)

Table 5.14: Fault injections and errors in ClustalW-MPI

Diagnosis & Recovery Step	ClustalW-MPI
1. Error detection latency	529
2. Error notification	17
3. Diagnosis	86
4. Vote	23
5. Restore checkpointed state	519
6. Additional application setup	8
Total	1182

Table 5.15: Performance of ClustalW-MPI application-specific fault tolerance mechanisms. All times are in ms.

manifested as errors. These manifestation rates are surprisingly low compared to the rates for Nqueens and TSP (Table 5.16) and compared to what other researchers have reported for transient single-bit faults [Sagg05]. Hence, it is quite likely that there was a bug in the injection campaign. For example, perhaps the faults were actually injected while some routine other than the one we intended was executing. We were, unfortunately, unable to determine definitively whether these results are meaningful.

Although it is likely that there was something wrong with the fault injections into ClustalW-MPI, the application fault tolerance mechanisms still detected process hangs and some corruption of application-level state, and the application successfully recovered from all

detected errors.

The execution times of ClustalW-MPI's diagnosis and recovery mechanisms are listed in Table 5.15. Unlike LAMMPS, ClustalW-MPI processes do not spend time unwinding the call stack (the second step in Table 5.13). This is because of ClustalW-MPI's use of a manager process with independent worker processes. There is no need for all processes to jump to a common point in application code to handle an error because the worker processes do not perform rollback recovery. After the error is handled and recovered from, worker processes immediately resume their local computation.

5.7.4. Nqueens and TSP Application-Specific Error Detection and Recovery

Nqueens and TSP are integer codes, so we used the same fault injection campaign as the first one used for ClustalW-MPI (Section 5.7.3), that is, one fault injection per application execution of a single bit-flip in a general purpose register. In preliminary campaigns in which bit-flip faults were injected into Nqueen and TSP processes, less than 3% of the faults were manifested as errors.

To increase the probability of faults manifesting as application-level errors, we targeted the fault injection by injecting faults while certain functions in application code were being executed. Specifically, we used *Gigan's* instruction breakpoint trigger and CPU cycle performance counter to inject a fault after the targeted application process entered a local computation routine and executed for a random number of instructions ranging from 5,000 to 100,000 instructions. For example, for the Nqueens worker process, the breakpoint was on the first instruction of the depth-first search algorithm. and for the Nqueens manager process, the breakpoint was set on the function that checks the validity of a solution. Similar

breakpoint triggers were used for TSP manager and worker processes.

Application	# fault injections	# process crashes	# process hangs	# application-level detections	# undetected errors
Nqueens, 4-proc.	1026	307 (30%)	182 (18%)	68 (7%)	22 (2%)
TSP, 4-proc.	1796	446 (25%)	54 (3%)	122 (7%)	14 (1%)

Table 5.16: Fault injections and errors in Nqueens and TSP

Diagnosis & Recovery Step	Nqueens	TSP
1. Error detection latency	12	53
2. Error notification	17	17
3. Diagnosis	81	81
4. Vote	23	23
5. Restore checkpointed state	3249	68
6. Additional application setup	4	4
Total	3286	246

Table 5.17: Performance of Nqueens and TSP application-specific fault tolerance mechanisms. All times are in ms.

The results of the breakpoint-triggered fault injection campaign are shown in Table 5.16 for Nqueens and TSP. For all of the runs in which faults manifested as crashed or hung processes, application-level fault tolerance detected and recovered from the error.

Table 5.17 lists the timing for the diagnosis and recovery mechanisms of Nqueens and TSP. In both applications, all of the faults that manifested as errors detected by application-level detection were recovered from, and the application generated correct output. For Nqueens, all of the undetected errors were due to fault injection in the manager process, causing the manager to invalidate a result that was actually valid. In theses cases, the number of solutions outputted by the application was one less than the correct number. This is due to

the fact that the manager process is a single point of failure. To fix this bug, the manager process's validity check of the results could also be performed by a second process. Comparison of the results of the duplicate checks would expose an error in either process.

5.8. Summary

The results of our evaluation of the layered implementation for fault-tolerant distributed applications show that the set of services provided by the CMM enable such applications to reliably tolerate process crashes, hangs, and corruption of application-specific state. We have shown that the CMM services impose very low performance overhead on applications. We have also shown that, as expected, application-specific techniques can provide fault tolerance with much lower overhead than application-transparent techniques. Our use of the CMM services with several applications demonstrates their ability to support a variety of application-level fault tolerance techniques.

Chapter Six

Conclusion

This dissertation presents research on a layered approach for implementing fault tolerance for distributed applications on compute clusters. Several factors increase the need for application-level fault tolerance, including growing number of unreliable COTS hardware components and the use of clusters in mission-critical applications and harsh environments.

For high-performance applications, the overhead for implementing application-transparent fault tolerance mechanisms can be prohibitively high. It is often the case that application-specific mechanisms for detecting and recovering from errors can meet the application's reliability requirements with much lower overhead.

In this dissertation, we described a layered approach to implementing application-specific fault tolerance mechanisms, involving four major layers of software: the single-node OS, the cluster management middleware, the message passing library for distributed applications, and application-level code.

The key innovation in the design presented is the implementation and utilization for fault tolerance of asynchronous communication between application processes and the cluster manager as well as among application processes. This mechanism allows the asynchronous delivery of messages, known as signals, to application processes. Signals can be initiated by the cluster manager or by application processes, and they can be atomically broadcasted to all processes of the application task or sent to the cluster manager. A group protocol enables multiple application processes to redundantly send a request to the cluster manager so that the

application can tolerate failed processes that send erroneous requests.

The signals mentioned above are used by the message-passing library to implement the ability for an application to detect and recover from process failures. Specifically, broadcast signals enable application processes to notify one another of and agree on the presence of errors or failed processes. Group signals enable application processes to request the cluster manager to remove failed processes from the task and add new fault-free processes to replace them.

To facilitate the utilization of the asynchronous communication mechanism described above for distributed applications, this dissertation also presents the design and implementation of an extension to the MPI API, called G-MPI. Several MPI applications were modified to implement application-specific error detection and recovery mechanisms using the new functionality.

To test and validate our implementation, the applications were executed using a Byzantine-fault-tolerant cluster manager known as Ghidrah. A variety of faults were introduced in the running system in repeated fault injection campaigns, including modification of CPU registers, halting or termination of application processes, halting or termination of CMM processes, and halting cluster nodes. To facilitate the execution of many runs of fault injections, we used virtualized clusters, in which each cluster node is run within a virtual machine. Using virtual machines enables instantiating multiple compute nodes on a single physical machine. It also allowed the test rig to efficiently shutdown a node that was targeted with a fault injection and restore a clean state for the node for succeeding fault injections.

Results of the test runs and fault injections showed that our fault tolerance mechanisms

have very low performance overhead in fault-free execution compared to application-transparent fault tolerance mechanisms. Faults that caused processes to halt or crash were all detected and recovered from. More than 90% of faults that caused corruption in application state (i.e. incorrect values) were detected by application-specific detection mechanisms, and correct application state was recovered 100% of the time. In instances where faults that caused corruption in application state were undetected, such as some of the LAMMPS runs, the application outputted incorrect results. However, the magnitude of the difference between the incorrect results and the correct results was very small.

The results of fault injection campaigns also demonstrated the effectiveness of the interface between the application and the cluster manager, the Ghidrah signal API. Most importantly, by enabling the cluster manager to notify application processes of a process failure, the application can respond and recover from the failure more quickly than if the application only relied on application-level hang detection.

Our layered design for implementing application-level fault tolerance mechanisms has proven to be effective for some distributed high performance computing applications. Further research is needed to determine how effective our layered approach can be in real-world situations. First, the scalability of our implementation should be studied with clusters of many more nodes and applications with many more processes. Second, the use of Ghidrah signals and our extensions to MPI API should be investigated in the context of a wider variety of applications.

Bibliography

[Agba99] A. M. Agbaria and R. Friedman, ‘‘Starfish: fault-tolerant dynamic MPI programs on clusters of workstations,’’ *8th International Symposium on High Performance Distributed Computing*, pp.167-176 (1999).

[Alba05] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung, ‘‘BioBench: A benchmark suite of bioinformatics applications,’’ *IEEE International Symposium on Performance Analysis of Systems and Software*, pp.2-9 (March 2005).

[Ande82] T. Anderson and P. A. Lee, ‘‘Fault Tolerance Terminology Proposals,’’ *12th International Symposium on Fault-Tolerant Computing*, pp.29-33 (1982).

[Anse09] J. Ansel, K. Arya, and G. Cooperman, ‘‘DMTCP: Transparent checkpointing for cluster computations and the desktop,’’ *International Symposium on Parallel and Distributed Processing*, pp.1-12 (May 2009).

[Aviz04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, ‘‘Basic concepts and taxonomy of dependable and secure computing,’’ *Dependable and Secure Computing, IEEE Transactions on*, vol.1, no.1, pp.11-33 (January 2004).

[Bagc91] A. Bagchi and S. L. Hakimi, ‘‘An optimal algorithm for distributed system level diagnosis,’’ *21st International Symposium on Fault-Tolerant Computing*, pp.214-221 (1991).

[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

(1993).

[Barh03] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, ‘‘Xen and the Art of Virtualization,’’ *19th ACM Symp. on Operating Systems Principles*, pp.164-177 (October 2003).

[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.99-114 (2001).

[Bosi02] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov, ‘‘MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes,’’ *ACM/IEEE International Conference on Supercomputing*, pp.1-18 (2002).

[Bout03] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier, and F. Magniette, ‘‘MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging,’’ *ACM/IEEE International Conference on Supercomputing*, pp.25-41 (2003).

[Camp86] R. H. Campbell and B. Randell, ‘‘Error recovery in asynchronous systems,’’ *IEEE Transactions on Software Engineering*, vol.SE-12, no.8, pp.811-826 (1986).

[Carr98] J. Carreira, H. Madeira, and J. G. Silva, ‘‘Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers,’’ *IEEE Trans. on Software Engineering*, vol.24, no.2, pp.125-136 (February 1998).

[Cha96] H. Cha, E. M. Rudnick, J. H. Patel, R. K. Iyer, and G. S. Choi, ‘‘A Gate-Level Simulation Environment for Alpha-Particle-Induced Transient Faults,’’ *IEEE Transactions on Computers*, vol.45, no.11, pp.1248-1256 (November 1996).

[Chan91] T. D. Chandra and S. Toueg, ‘‘Unreliable failure detectors for asynchronous systems (preliminary version),’’ *10th Annual ACM Symposium on Principles of Distributed Computing*, pp.325-340 (1991).

[Clar09] W. J. Clarke, L. C. Alves, T. J. Dell, H. Elfering, J. P. Kubala, C. Lin, M. J. Mueller, and K. Werner, ‘‘IBM system z10 design for RAS,’’ *IBM Journal of Research and Development*, vol.53, no.1, pp.120-130 (January 2009).

[Dong98] J. Dongarra, G. E. Fagg, A. Geist, J. A. Kohl, P. M. Papadopoulos, S. L. Scott, V. S. Sunderam, and M. Magliardi, ‘‘HARNESS: Heterogeneous Adaptable Reconfigurable NEtworked SystemS,’’ *7th IEEE International Symposium on High Performance Distributed Computing*, pp.358-359 (July 1998).

[Dong05] J. Dongarra, T. Sterling, H. Simon, and E. Strohmaier, ‘‘High-performance computing: clusters, constellations, MPPs, and future directions,’’ *Computing in Science Engineering*, vol.7, no.2, pp.51-59 (March 2005).

[Dwor88] C. Dwork, N. Lynch, and L. Stockmeyer, ‘‘Consensus in the presence of partial synchrony,’’ *Journal of the ACM*, vol.35, no.2, pp.288-323 (1988).

[Elno02] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, ‘‘A survey of rollback-recovery protocols in message-passing systems,’’ *ACM Computing Surveys*, vol.34, no.3, pp.375-408 (September 2002).

[Fagg05] G. E. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, J. Pjesivac-Grbovic,

and J. J. Dongarra, ‘‘Process Fault Tolerance: Semantics, Design and Applications for High Performance Computing,’’ *International Journal of High Performance Computing Applications*, vol.19, no.4, pp.465-477 (2005).

[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 (1985).

[Ghor98] D. P. Ghormley, D. Petrou, S. H. Rodrigues, A. 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 (1998).

[Hain00] J. Haines, V. Lakamraju, I. Koren, and C. M. Krishna, ‘‘Application-Level Fault Tolerance as a Complement to System-Level Fault Tolerance,’’ *Journal of Supercomputing*, vol.16, no.1-2, pp.53-68 , Kluwer Academic Publishers (2000).

[Hopk78] A. L. Hopkins Jr., T. B. Smith III, and J. H. Lala, ‘‘FTMP—A highly reliable fault-tolerant multiprocessor for aircraft,’’ *Proceedings of the IEEE*, vol.66, no.10, pp.1221-1239 (October 1978).

[Hoss84] S. H. Hosseini, J. G. Kuhl, and S. M. 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).

[Huan84] K.-H. Huang and J. A. Abraham, ‘‘Algorithm-based fault tolerance for matrix operations,’’ *IEEE Transactions on Computers*, vol.C-33, no.6, pp.518-528

(June 1984).

[Kalb99] Z. T. 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).

[Kim89] K. H. Kim and H. O. Welch, ‘‘Distributed execution of recovery blocks: an approach for uniform treatment of hardware and software faults in real-time applications,’’ *IEEE Transactions on Computers*, vol.38, no.5, pp.626-636 (May 1989).

[Kuhl80] J. G. Kuhl and S. M. Reddy, ‘‘Distributed fault-tolerance for large multiprocessor systems,’’ in *7th Annual Symposium on Computer Architecture*, New York, NY, USA, pp.23-30 (1980).

[Lamp82] L. Lamport, R. Shostak, and M. Pease, ‘‘The Byzantine Generals Problem,’’ *ACM Transactions on Programming Language Systems*, vol.4, no.3, pp.382-401 (July 1982).

[Le08] M. Le, A. Gallagher, and Y. Tamir, ‘‘Challenges and Opportunities with Fault Injection in Virtualized Systems,’’ in *1st Int. Workshop on Virtualization Performance: Analysis, Characterization, and Tools*, Austin, TX, (April 2008).

[Li03] K.-B. Li, ‘‘ClustalW-MPI: ClustalW analysis using distributed and parallel computing,’’ *Bioinformatics*, vol.19, no.12, pp.1585-1586 (2003).

[Li01] M. Li, D. Goldberg, W. Tao, and Y. Tamir, ‘‘Fault-Tolerant Cluster Management for Reliable High-Performance Computing,’’ in *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*, pp.266-274 (September 2002).

[Mess95] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*. (June 1995).

[Müll10] M. S. Müller, M. v. Waveren, R. Lieberman, B. Whitney, H. Saito, K. Kumaran, J. Baron, W. C. Brantley, C. Parrott, T. Elken, H. Feng, and C. Ponder, ‘‘SPEC MPI2007—an application benchmark suite for parallel systems using MPI,’’ *Concurrency and Computation: Practice and Experience*, vol.22, no.2, pp.191-205 , John Wiley Sons, Ltd. (2010).

[Neve98] N. Neves and W. K. Fuchs, ‘‘RENEW: A Tool for Fast and Efficient Implementation of Checkpoint Protocols,’’ in *28th Annual International Symposium on Fault-Tolerant Computing*, IEEE Computer Society, Washington, DC, USA, p.58 (1998).

[Okaj06] J. R. Okajima, ‘‘aufs—another unionfs,’’ *http://aufs.sourceforge.net* (2006).

[Plim95] S. Plimpton, ‘‘Fast Parallel Algorithms for Short-Range Molecular Dynamics,’’ *Journal of Computational Physics*, vol.117, pp.1-19 (March 1995).

[Powe94] D. Powell, ‘‘Distributed fault tolerance—Lessons learnt from Delta-4,’’

Hardware and Software Architectures for Fault Tolerance, vol.774, pp.199-217 (1994).

[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, no.6, pp.848-854 (Decemeber 1967).

[Rand75] B. Randell, ‘‘System structure for software fault tolerance,’’ in *International Conference on Reliable Software*, New York, NY, USA, pp.437-449 (1975).

[Rao99] S. Rao, L. Alvisi, and H. M. Vin, ‘‘Egida: An extensible toolkit for low-overhead fault-tolerance,’’ in *29th Annual International Symposium on Fault-Tolerant Computing*, Washington, DC, USA, p.48 (1999).

[Sagg05] G. P. Saggese, N. J. Wang, Z. T. Kalbarczyk, S. J. Patel, and R. K. Iyer, ‘‘An experimental study of soft errors in microprocessors,’’ *IEEE Micro*, vol.25, no.6, pp.30-39 (November 2005).

[Salt84] J. H. Saltzer, D. P. Reed, and D. D. Clark, ‘‘End-to-end arguments in system design,’’ *ACM Transactions on Computer Systems*, vol.2, no.4, pp.277-288 (November 1984).

[Schl83] R. D. Schlichting and F. B. Schneider, ‘‘Fail-stop processors: an approach to designing fault-tolerant computing systems,’’ *ACM Transactions on Computer Systems*, vol.1, no.3, pp.222-238 , ACM Press (1983).

[Schr06] B. Schroeder and G. A. Gibson, ‘‘A large-scale study of failures in high-performance computing systems,’’ in *International Conference on Dependable Systems and Networks*, pp.249-258 (2006).

[Sieh02] V. Sieh and K. Buchacker, ‘‘UMLinux - A Versatile SWIFI Tool,’’ in *4th European Dependable Computing Conference*, Toulouse, France, pp.159-171 (October 2002).

[Some99] R. R. Some and D. C. Ngo, ‘‘REE: a COTS-based fault tolerant parallel processing supercomputer for spacecraft onboard scientific data analysis,’’ *18th Digital Avionics Systems Conference*, vol.2, pp.7.B.3.1-7.B.3.12 (1999).

[Stel96] G. Stellner, ‘‘CoCheck: Checkpointing and Process Migration for MPI,’’ in *10th International Parallel Processing Symposium*, pp.526-531 (April 1996).

[Thom94] J. D. Thompson, D. G. Higgins, and T. J. Gibson, ‘‘CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice,’’ *Nucleic acids research*, vol.22, no.22, pp.4673-4680 (1994).

[Webb91] S. Webber and J. Beirne, ‘‘The Stratus architecture,’’ *International Symposium on Fault-Tolerant Computing*, pp.79-85 (June 1991).

[Wens78] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, and C. B. Weinstock, ‘‘SIFT: Design and analysis of a fault-tolerant computer for aircraft control,’’ *Proceedings of the IEEE*, vol.66, no.10, pp.1240-1255 (October 1978).