RICE UNIVERSITY

Manetho: Fault Tolerance in Distributed Systems Using Rollback-Recovery and Process Replication

by

Elmootazbellah Nabil Elnozahy

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Willy Zwaenepoel, Chairman
Professor of Computer Science

Robert Cartwright
Professor of Computer Science

John Bennett
Assistant Professor of Electrical and
Computer Engineering

Houston, Texas

October, 1993

Manetho: Fault Tolerance in Distributed Systems Using Rollback-Recovery and Process Replication

Elmootazbellah Nabil Elnozahy

Abstract

This dissertation presents a new protocol that allows rollback-recovery and process replication to co-exist in a distributed system. The protocol relies on a novel data structure called the antecedence graph, which tracks the nondeterministic events during failure-free operation and provides information for recreating them if a failure occurs. The rollback-recovery part of the protocol combines the low failure-free overhead of optimistic rollback-recovery with the advantages of pessimistic rollback-recovery, namely fast output commit, limited rollback, and failure-containment. The process replication part of the protocol features a new multicast protocol designed specifically to support process replication. Unlike previous work, the new protocol provides high throughput and low latency in message delivery without relying on the application semantics.

The protocol has been implemented in the Manetho prototype. Experience with a number of long-running, compute-intensive parallel applications confirms the performance advantages of the new protocol. The implementation also features several performance optimizations that are applicable to other rollback-recovery and multicast protocols.

Acknowledgments

I dedicate this work to my parents, as an expression of love and gratitude for providing me with the best education. Without their love, sacrifices, care and encouragements, I would not have written this dissertation. And to my wife, I am grateful for her love and devotion. Over the last two years, she was an endless source of inspiration and motivation that were crucial in finishing this work.

I would like to express my appreciation to my advisor, Willy Zwaenepoel for his upholding of high standards in doing quality research. I am grateful for his comments on this manuscript, and also for his contributions to the papers that I published while at Rice. I wish also to express my gratitude to David Johnson for many constructive discussions and his comments on this manuscript and my previous papers. He also helped me understand the inner details of the V kernel. I also thank John Carter for his continuous encouragements.

I would like also to express my appreciation to the staff at the IBM T.J. Watson Research Lab for their generous support and great friendship, especially Anupam Bhide, Mike Blasgen, Eric Kronstadt, Mark Mergen, and Stephen Morgan.

Several people have helped make my life easier during my stay at Rice. I would like to thank A. Abuhashish, A. Alduhyan, L. Arbilla, A. Boriek, R. Bubenick, S. Carr, A. Cox, S. Dwarkadas, A. Elbakry, M. Elalem, K. Elsegheir, P. Keleher, U. Kremer, M. Mazina, M. Maged, D. Monk, R. Rodriguez, A. Sabry, and B. Selke. I also acknowledge with appreciation the excellent administrative support of Vicky Dean and Ivy Jorgensen. I would also like to thank Corky Cartright and John Bennett for serving on my committee.

Financial support for this research was provided by an IBM graduate fellowship, and by grants from the National Science Foundation, the State of Texas, Rice University, Cairo University (Egypt), and WGF-Switzerland.

Contents

Illustrations

Tables

Preface

The old Egyptian civilization had no exact system of chronology. The priests usually dated events according to the years of a king's reign. For this purpose, several lists of kings were maintained at the various temples throughout Egypt. Some of these lists survived the decline of the old Egyptian empire. The priest **Manetho** *(Ma-Net-Ho)* lived during the reign of the Ptolemies, circa 300 B.C., at the center of the Nile Delta in Sebennytus, a place now called Samannud. He collected the lists that were preserved in the various temples and used them to write the history of Egypt in a three volume book. This book remained the authentic source for Egypt's history for several centuries until it was lost, probably during the fire of the library of Alexandria (circa 390 A.D.).

This thesis describes a fault-tolerant distributed computing system whose operation metaphorically resembles what Manetho did to restore the history of Egypt. Like old Egypt, the system does not have access to an exact, global time service. Each individual process maintains information about its perception of the system's execution history, similar to what the priests of old Egypt did. If a failure occurs, a protocol collects the fragments of the system's execution history from the individual processes, and like Manetho, restores the full history of the system. This history is used to recover from the failure.

The system is named after Manetho because of the similarity to his work and in memory of a wonderful civilization.

Chapter 1

Introduction

This chapter describes the application domain targeted by Manetho. It also gives a short description of the new protocols presented in this dissertation and their contributions to the field of fault tolerance in distributed systems.

1.1 Network Multicomputers

Recent advances in technology are making it possible to build network multicomputers by connecting high-performance workstations using general purpose networks. Network multicomputers offer processing capacities that rival those of the mainframe computers and shared memory multiprocessors, at a fraction of their cost. They are also more scalable and easier to upgrade than shared memory multiprocessors.

Figure 1.1 shows a diagram of a typical network multicomputer, consisting of a workstation cluster and general purpose servers managed by a distributed operating system. The operating system controls the network multicomputer and makes it behave like a central computing facility. The workstation cluster acts as a compute server that provides processing resources and parallelism required for high performance computing. Examples of applications that would benefit from this environment include scientific computations, distributed simulations, and parallel combinatorial algorithms.

A network multicomputer also includes general purpose servers that offer services commonly required in a distributed computing environment. Typical examples include file, name, and authentication servers. The multicomputer would also include a job distribution server that balances the computational load on the workstation cluster.

Network multicomputers, however, are vulnerable to many failure modes. These modes make it more difficult for a network multicomputer to offer its users the familiar model of a central computing facility. Examples of such failures include communication and processor failures. These failure modes are different from those of a central

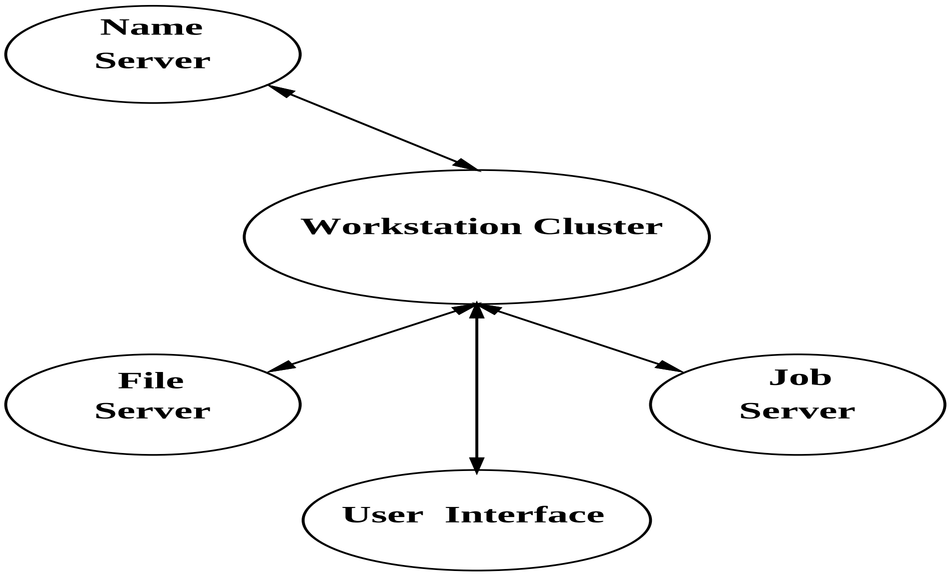


Figure 1.1 A diagram of a typical network multicomputer.

computing facility where no partial failure occurs and the system behaves like a unit that either functions properly or fails entirely. Furthermore, the likelihood of failure increases with the number of processors, and a single failure often renders the entire system unusable. This dissertation focuses on the problem of providing fault tolerance to network multicomputers.

1.2 Fault Tolerance in a Network Multicomputer

A recovery protocol for a network multicomputer has to handle failures in its two main components: the workstation cluster and the general purpose servers. These two components have different requirements with respect to fault tolerance because of the intended use of each. The workstation cluster is intended for application programs where efficiency and application transparency are important. In addition to these two requirements, general purpose servers also require high availability, since the failure of any of them affects the entire system.

1.2.1 The Workstation Cluster

The workstation cluster provides the resources for running compute-intensive application programs. Therefore, it requires a protocol that provides fault tolerance with low performance overhead. Otherwise, the resulting overhead competes with the application programs for computing resources, reducing the performance of the network multicomputer.

In addition to low overhead, it is desirable to add fault tolerance to a network multicomputer without changing the application programs. A goal of network multicomputing is to offer the simple programming model of tightly coupled parallel processors. The programs written in this model do not assume the complex failure modes that occur in distributed systems.

1.2.2 Server Programs

Unlike application programs, the failure of one general purpose server affects the entire system. For example, if a job distribution server fails, users will not be able to submit new jobs. Similarly, if a file server fails, application programs that manipulate files will not be able to continue. Therefore, a network multicomputer requires a protocol that ensures high availability for server programs, besides low overhead and transparency.

1.3 Manetho

Manetho uses rollback-recovery and process replication to provide fault tolerance for the two main components of a network multicomputer. Figure 1.2 shows a diagram of Manetho. Rollback-recovery provides transparent, low-overhead fault tolerance to the workstation cluster on which application programs run, while process replication provides the high availability required for server programs. Furthermore, Manetho's protocols allow processes that use the two recovery methods to co-exist.

1.3.1 Rollback-Recovery

In rollback-recovery, processes periodically save their states on stable storage during failure-free operation [AL89, BBG$^+$89, BCS84, BHG87, BL88, CJ91, CL85, GGL$^+$90, IM89, Jal89, Joh89, JV91, JZ87, JZ90, KMBT92, KT87, KYA86, LNP91, MS92, PP83, SBY88, SK86, SW89, SY85, TS84, WF92a, WF92b]. If a failure occurs, the processes will use the information on stable storage to restore a global consistent state

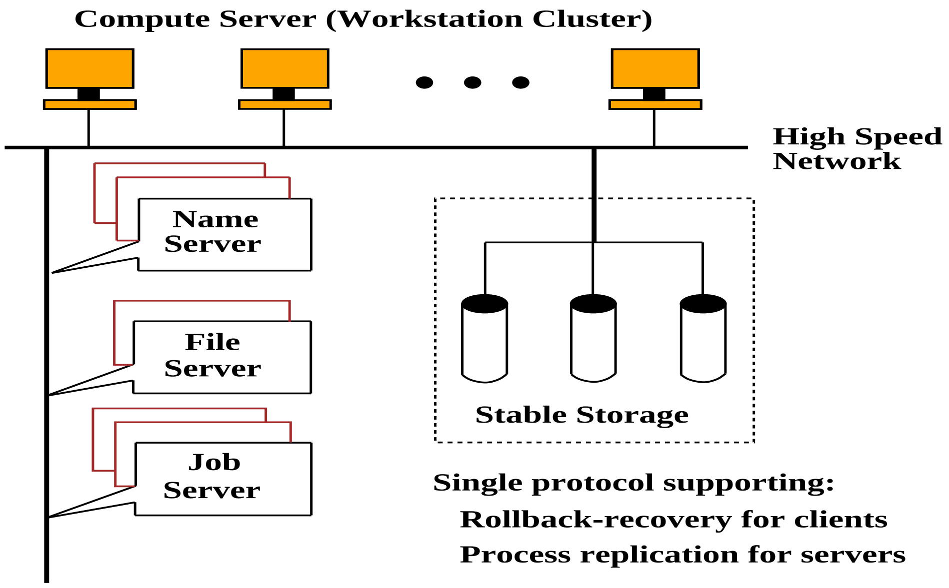


Figure 1.2 A diagram of Manetho.

and restart execution from that state, instead of restarting the computation from the beginning. A global consistent state is one that could have occurred during a failure-free execution of the application program [CL85]. Finding this consistent state may entail rolling back processes that have survived the failure [JZ90, KT87, SY85].

Rollback-recovery has the potential for providing fault tolerance with low overhead in resources and performance. Stable storage devices typically use magnetic disks and are inexpensive. The performance overhead results from saving recovery information on stable storage during failure-free operation. As discussed in this dissertation, proper implementation techniques reduce the effect of this state saving on the application's performance. Rollback-recovery also provides fault tolerance transparently. By embodying state saving and restoration techniques in the operating system, the provision of rollback-recovery is automatic and no change to application programs is required.

1.3.2 Process Replication

Manetho uses process replication to provide high availability to servers [ADL90, BF83, Bir85, Coo84, Coo85, CPR+92, SES+92]. Several copies of a server program are run on different machines. One copy is distinguished and is called the leader, while the remaining copies are called cohorts. During failure-free operation, replicas of a server program must receive the same set of messages in the same order [ADKM92, Bir85, BSS91, MSMA90, PBS89, VRB89, Woo93]. Otherwise, their executions will not remain identical. For this purpose, Manetho features a multicast protocol that ensures ordered delivery of application messages among all replicas.

Leaders represent their corresponding replicated servers when interacting with application programs or other servers. Cohorts are silent performers that receive all messages intended for the server program but do not send application messages over the network. If the leader fails, the surviving cohorts will elect a new leader among themselves and operation will continue immediately, ensuring high availability.

1.3.3 The Antecedence Graph

When a process fails, the recovery protocol may need to roll back several processes to restart from a global consistent state [CL85, SY85]. It is however undesirable to roll back replicated processes, because the services that they offer will not be available during recovery. Therefore, to allow rollback-recovery to co-exist with process replication, the recovery protocol must restart the system from a consistent state without rolling back the replicated processes. Many rollback-recovery protocols cannot give such a guarantee [CL85, JZ88, KT87, SY85].

Combining rollback-recovery and process replication in Manetho is made possible by using an antecedence graph. This data structure tracks nondeterministic events that occur during failure-free operation. During recovery, the information maintained in the antecedence graph enables the system to recreate the same sequence of nondeterministic events as before the failure. The use of the antecedence graph maintains the invariant that no process is affected by failures that occur in other processes. This invariant is the key property that allows rollback-recovery to co-exist with process replication. Failures and rollbacks that occur in other processes do not affect server processes that use replication. Therefore, a replicated process does not need to roll back its state because of the failure of another process.

Using the antecedence graph also leads to several advantages with respect to rollback-recovery. Manetho's rollback-recovery protocol incurs low overhead during failure-free operation, while protecting the processes that survive a failure from having to roll back during recovery to find the global consistent state. Only failed processes have to roll back, and only to the latest checkpoint recorded on stable storage. This limited rollback is desirable since it reduces the time required to recover from a failure. Furthermore, Manetho provides low latency in communication between the system and the outside world. The outside world consists of all entities that cannot roll back their states after a failure, such as a line printer. Any rollback-recovery protocol must ensure that the state from which the system sends a message to the outside world will never be rolled back. Typically, the recovery protocol logs information on stable storage before sending the message and thus introduces some delay. The act of logging information before sending messages to the outside world has been called "output commit" [SY85]. In Manetho, output commit is a low latency operation. A process committing output only needs to write its antecedence graph on stable storage. Unlike many other protocols, no multihost coordination is necessary.

Using the antecedence graph also leads to several advantages with respect to process replication. Manetho's multicast protocol offers high throughput and low latency message delivery. Having both these features was not possible in previous multicast protocols without relying on application semantics [BSS91, PBS89]. In contrast, Manetho's multicast is application-transparent.

A prototype for Manetho has been implemented on a network multicomputer consisting of a 10 Mbits/second Ethernet network connecting 16 Sun-3/60 workstations. The prototype serves as an example that shows how to implement the protocols in practice and also as a test bed for performance evaluation. Performance measurements confirm the low overhead of Manetho's protocols. For the application programs tested, the overhead increases the running time of the application by less than 4%.

1.4 Contributions

The contributions of this dissertation are:

1. A new recovery protocol that is based on a novel data structure called the antecedence graph. This protocol allows processes that use rollback-recovery to co-exist with those that use process replication in the same system.

2. A rollback-recovery protocol that combines the advantages of low failure-free overhead, fast output commit, and minimum rollbacks after failure. No previous protocol offered such a combination of advantages without using hardware support.

3. A multicast protocol that is designed specifically to support process replication. This application-transparent protocol delivers multicast messages with low latency and high throughput. No previous protocol combined these two advantages without relying on the application semantics.

An empirical performance evaluation using the prototype implementation confirms that the advantages of the system are realized in practice. In addition, the evaluation reveals several results that are applicable to Manetho and are relevant to any rollback-recovery or multicast protocol:

1. Contrary to previous assumptions [BL88, SY85], coordinated checkpointing is an efficient technique for implementing rollback-recovery. Moreover, the implementation shows that using coordinated checkpointing in message logging protocols improves performance and reduces the complexity of garbage collection.

2. Concurrent and incremental checkpointing [LNP90] are essential for low failure-free performance overhead.

3. Sender-based message logging [JZ87, SBY88] has proved to be superior to receiver-based logging [JV91, JZ90, SW89, SY85].

4. The implementation provides empirical evidence supporting the conventional wisdom that positive acknowledgment multicast protocols provide lower latency than positive acknowledgment protocols at the expense of a lower throughput.

The performance evaluation also includes a comparative study between Manetho and other systems. The purpose of this study is to identify the strengths and weaknesses of each system and the proper implementation techniques necessary for good performance.

1.5 Outline

Chapter 2 describes the design and proofs of correctness of Manetho's protocols. Chapter 3 presents the implementation of Manetho. Chapter 4 presents the performance evaluation of Manetho, and an empirical comparative study of Manetho with alternative protocols. Chapter 5 compares the work described in this dissertation with previous work in the field. Finally, Chapter 6 summarizes the contributions of this research and discusses avenues for further work.

Chapter 2

The Design of Manetho

This chapter describes the design of Manetho and proves the correctness of its protocols. Implementation and performance are presented in the following two chapters.

2.1 Overview

Manetho provides four protocols that are added to a distributed system to provide transparent fault tolerance:

A communication protocol: This protocol supports the maintenance of the antecedence graph, logs the messages exchanged among the processes, and controls the interactions of the distributed computation with the outside world. All processes in the distributed computation participate in the protocol.

A checkpointing protocol: Processes that use rollback-recovery periodically use this protocol to store their states. The protocol ensures that a consistent [CL85] set of process states is always available on stable storage. After a failure occurs, the failed processes restore their states from stable storage and restart execution.

A multicast protocol: Replicated processes use this protocol to ensure that they receive the same set of messages in the same order, ensuring that the executions of the replicas remain identical.

A recovery protocol: This protocol uses the antecedence graph, message logs, and checkpoints to ensure that if a failure occurs, the processes will recover to a consistent system state. Furthermore, the protocol ensures that none of the processes that survive the failure are aborted and that the outside world is not affected. The protocol tolerates any number of process and communication failures, including failures that occur during recovery. All processes that participate in the system use this protocol, whether they use rollback-recovery or replication.

The outline of this chapter is as follows. Section 2.2 and 2.3 describe the assumptions made about the system and define the antecedence graph. A description of the communication protocol follows in Section 2.4. Sections 2.5 and 2.6 describe the checkpointing and multicast protocols, respectively. A discussion of garbage collection issues follows in Section 2.7. Section 2.8 describes the recovery protocol and Section 2.9 provides proofs for its correctness. Section 2.10 summarizes the chapter.

2.2 System Model

2.2.1 Overview

A distributed system consists of a number of *recovery units* (*RU's*) that communicate only through messages [SY85]. An *RU* is the unit of failure and recovery in the system. It can correspond to a process, a machine, or any unit of failure as conveniently defined by a particular implementation. Abstractly, it represents a volatile state manipulated by a number of threads according to some application program. An *RU* may use rollback-recovery or active replication.

The distributed computation may interact with the outside world by sending and receiving messages. However, the issues of implementing the interface to the outside world are outside the scope of this dissertation. For a study of these issues, the reader is referred to the work of Pausch [Pau88].

2.2.2 Assumptions

The following is assumed about the distributed system:

1. The *RU's* do not have access to a common time base or synchronized clocks.

2. The *RU's* are fail-stop [SS83]. An *RU* that uses rollback-recovery fails by losing its volatile state and destroying its threads without transmitting incorrect messages. The replicas of a replication-based *RU* may fail in the same manner, but no more than $r-1$ failures can occur in the *RU*, where r is the number of replicas.

3. Each message has a unique identifier. The generation of unique identifiers is assumed to be deterministic.

4. The communication subsystem is assumed to be unreliable and asynchronous: a message may be lost, duplicated, or arbitrarily delayed. However, corrupted messages are detected and suppressed.

5. Each *RU* has access to a highly-available, stable storage device that survives failures [LS79].

6. Messages sent by an *RU* to the outside world may produce external effects that cannot be reversed. Furthermore, the interface to the outside world cannot be relied on to resend input messages for recovery purposes.

7. The system includes a membership protocol that maintains a list of the functioning *RU's*.

The communication subsystem does not guarantee the reliable, ordered message delivery commonly needed by application programs. These properties are easily provided by a standard end-to-end protocol [Tan88]. The Manetho runtime system treats control messages used by the end-to-end protocol as ordinary application messages. Examples of such control messages include acknowledgments, messages maintaining the connection between two machines, and messages belonging to the flow control algorithm. This assumption simplifies the presentation and makes the design independent of the communication protocol.

2.2.3 Nondeterministic Events and State Intervals

The execution of an *RU* consists of a sequence of piecewise deterministic *state intervals* [SY85]. Each state interval starts by the occurrence of a *nondeterministic* event. Such an event can be:

- The *creation* of the *RU*.

- A *message receipt* event that occurs when the *RU* receives a message from another *RU*.

- An *input* event that occurs when the *RU* receives a message from the outside world.

- An *internal* event, such as a synchronization operation between two threads in the *RU*. What constitutes the set of internal events depends on the amount of nondeterminism that a particular implementation is willing to support. For replication-based *RU's*, the outcome of an internal event is defined by the actual outcome of the event at the leader. The cohorts are forced to produce the same outcome so that their states remain identical to the state of their leader.

Figure 2.1 shows a sample execution of a distributed system consisting of three *RU's* p, q and r. The notation σ_i^p denotes the i^{th} state interval of RU p, where i is called the *index* of σ_i^p. The notation m_i^p denotes a message transmitted by RU p, where i represents the unique identifier of the message. Horizontal lines represent the progress of the execution, and an arrow between two horizontal lines denotes a message transmission. A vertical bar marks the beginning of each state interval. In the figure, state interval σ_1^r starts with an internal event that occurs in RU r, while all other events are either creation or message receipt events.

2.2.4 Incarnation Numbers

Because network delay can be arbitrarily large, a message may reach its destination after its sender has failed. If the receiver has been notified of the sender's failure before receiving the message, it will not be able to determine whether the message originated before the sender's failure or after its recovery. This is an instance of the problem of ordering the perception of failures with respect to application messages [BJ87a, SY85].

To solve this problem, each RU maintains an *incarnation number*. Each time an RU starts recovery from a failure, it increments its incarnation number and reliably sends it to all *RU's* before restarting execution. Each application message is tagged with the current incarnation number of the sender. Thus, the receiver of a message determines from the incarnation number tagging it whether it originated during the current incarnation of the sender or not.

2.3 The Antecedence Graph

2.3.1 Description

The antecedence graph of a state interval σ_i^p contains a summary of the system's execution that *happened before* σ_i^p [Lam78]. It contains a node representing σ_i^p and a node for each state interval that happened before σ_i^p. The edges of the graph represent the *happened before* relation between the state intervals. The structure of a graph

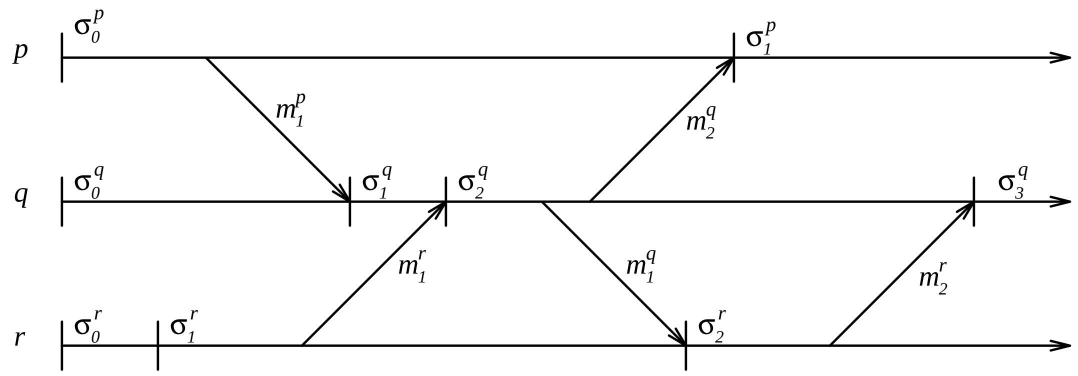


Figure 2.1 An example execution.

node depends on the type of nondeterministic event that created the corresponding state interval:

- The graph node corresponding to the initial state interval of an *RU* has no incoming edges. It indicates the creation of the *RU*.

- For a state interval created by a message receipt event, the corresponding graph node has two incoming edges: one from the node representing the previous state interval in the receiving *RU*, and one from the node representing the sender's state interval during which the message was sent. The node contains: **i)** a type field indicating a message receipt event, **ii)** the index of the state interval, **iii)** the identifier of the receiver, **iv)** the identifier of the sender, and **v)** the unique identifier of the message. The node does *not* contain a copy of the message's data.

- For a state interval created by an input event, the graph node has only one incoming edge from the node corresponding to the previous state interval in the *RU*. The node's contents are analogous to those of a state interval that starts by a message receipt event, except that the type field indicates an input event, and that the node *contains* a copy of the message.

- For a state interval created by an internal event, the corresponding graph node has one incoming edge from the node that corresponds to the previous state interval in the *RU*. The node's contents include a type field indicating the internal event's type, and the information necessary to replay the event.

The information in the antecedence graph is used to recreate the execution of the system should a failure occur.

2.3.2 Example

Figure 2.2 shows the antecedence graph of state interval σ_1^p of the example execution depicted in Figure 2.1. The graph represents the computation that occurred before σ_1^p. It does not, however, contain information about the transmission or receipt of messages m_1^q and m_2^r. The state intervals that occur because of these messages do not precede σ_1^p according to the definition of the antecedence graph.

2.4 Manetho's Communication Protocol

The *RU's* participating in Manetho use a communication protocol to maintain the antecedence graph and to control interactions with the outside world. All *RU's* follow this protocol whether they use rollback-recovery or replication.

2.4.1 Data Structures

Each *RU* maintains a set of data structures to support Manetho's communication protocol. For replication-based *RU's*, each replica maintains its own copy of these data structures. Some data structures have a persistent version that is maintained on stable storage and survives failures, and a volatile version that acts as a cache of the persistent version but loses its contents after a failure. It is also assumed that updating the volatile version is much faster than updating the persistent one [LS79]. Thus, performance considerations prohibit updating the persistent version whenever the volatile version changes. For convenience, the notation $p.x$ is used to refer to data structure x of *RU* p. A description of all data structures follows:

ID : The identifier of the *RU*.

SI : The index of the current state interval of the *RU*. It is kept in volatile storage and is incremented each time a nondeterministic event occurs in the *RU*.

MSGLOG : A volatile log containing a copy of every message *sent* by the *RU*. *MSGLOG* is updated each time the *RU* sends an inter-*RU* message.

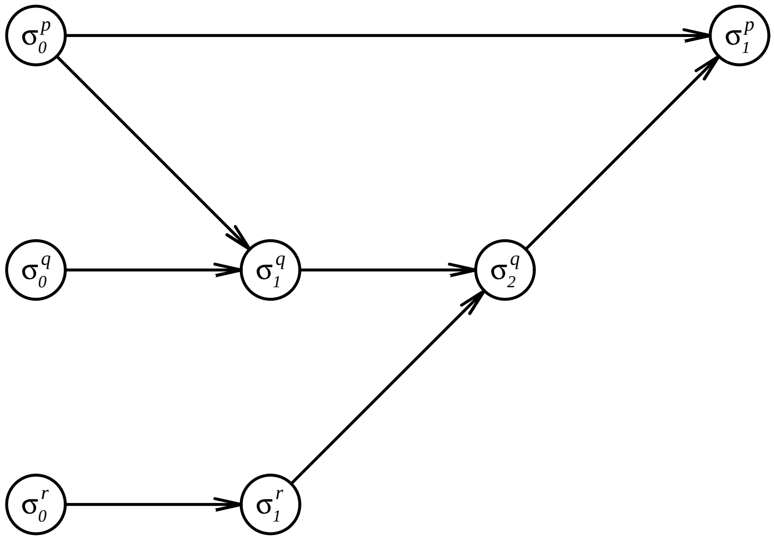


Figure 2.2 The antecedence graph of state interval σ_1^p in the example of Figure 2.1.

AG : The antecedence graph of the current state interval of the *RU*. *AG* is volatile and is updated whenever a nondeterministic event occurs in the *RU*, as described in Section 2.3.

P_AG : The persistent version of *AG*. An *RU* merges *AG* into *P_AG* when it commits output or when the size of the antecedence graph becomes too large to be piggybacked on application messages (see Section 2.4.2).

INCNUM : The incarnation number of the *RU*. It is incremented each time the *RU* starts the recovery protocol.

P_INCNUM : A persistent version of *INCNUM*.

REJECT_LIST : A volatile data structure used to inhibit receiving certain application messages during recovery (see Section 2.8).

INCVEC : A volatile vector containing the incarnation number of each *RU*.

KNOWN_SI : A volatile vector used in implementing incremental piggybacking of the antecedence graph (see Section 2.4.2). For some *RU* q, $q.KNOWN\_SI$ [p] contains the maximum index i such that $\sigma_i^p \in (q.AG \cup q.P\_AG)$.

LAST_SI : A volatile vector used in the implementing incremental piggybacking of the antecedence graph (see Section 2.4.2). For some *RU* q, $q.LAST\_SI$ [p] contains the maximum index of a state interval σ_i^q, such that q knows that such that $\sigma_i^q \in (p.AG \cup p.P\_AG)$.

P_COMMITLOG : A persistent data structure that contains the identifier of each message sent to the outside world.

2.4.2 Inter-RU Communication Protocol

Description

When an *RU* sends an inter-*RU* message, it (conceptually) piggybacks a copy of its *volatile* antecedence graph on the message. It also saves a copy of the message data in its local, volatile message log. When the message arrives at the receiver, a new state interval starts. As described in Section 2.3, the antecedence graph of the receiver is updated to reflect the creation of the new state interval. Consider the example in Figure 2.1. *RU* q starts state interval σ_2^q when it receives message m_1^r from *RU* r. *RU* q appends a node corresponding to that state interval to $q.AG$, and creates two edges to this node. The first edge is from the node corresponding to the state interval of the sender in the antecedence graph piggybacked on the message. The second edge is from the node corresponding to the previous state interval of the receiver in the local antecedence graph. Figure 2.3 shows the antecedence graph piggybacked on message m_1^r. The figure also shows the graph at *RU* q before and after receiving m_1^r.

The sender does not include the persistent version of the antecedence graph on the message. Furthermore, the sender need not include a complete copy of its volatile antecedence graph on every outgoing message. Instead, incremental piggybacking is used. By definition, for any *RU* p, $AG(\sigma_i^p)$ is a proper subgraph of $AG(\sigma_{i+1}^p)$. Thus, each *RU* q that communicates with p includes with each message sent to p the maximum state interval index j, such that σ_j^p is in $q.AG \cup q.P\_AG$. Later, when p at state interval σ_i^p sends a message to q, p piggybacks only $AG(\sigma_i^p) - AG(\sigma_j^p)$ on the outgoing message.

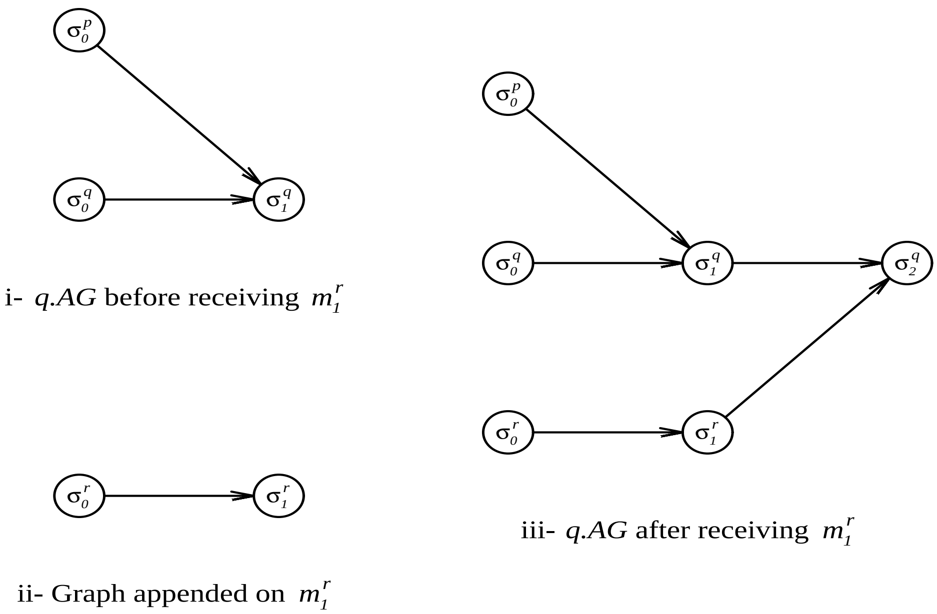


Figure 2.3 Antecedence graph maintenance at RU q before and after receiving m_1^r in the example of Figure 2.1.

In addition to incremental piggybacking, the protocol uses another optimization to reduce the size of the antecedence graph piggybacked on an application message. If $AG(\sigma_i^q)$ is a subset of $p.AG \cup p.P\_AG$, then p does not piggyback $AG(\sigma_i^q)$ to any message sent to q. Such information would be redundant, since $q.AG \cup q.P\_AG$ already contains $AG(\sigma_i^q)$.

Consider again the example in Figure 2.1. By using incremental piggybacking, the graph piggybacked on message m_2^r need not include $AG(\sigma_1^r)$, which was earlier piggybacked on m_1^r. Furthermore, $AG(\sigma_2^q)$ which is in $AG(\sigma_2^r)$ need not be transmitted to q. The antecedence graph at r during the transmission of m_2^r and the graph piggybacked on m_2^r are shown in Figure 2.4.

Manetho Message Header

The inter-RU protocol of Manetho adds a header to every application message. The complete message is then transmitted using any network protocol. The message

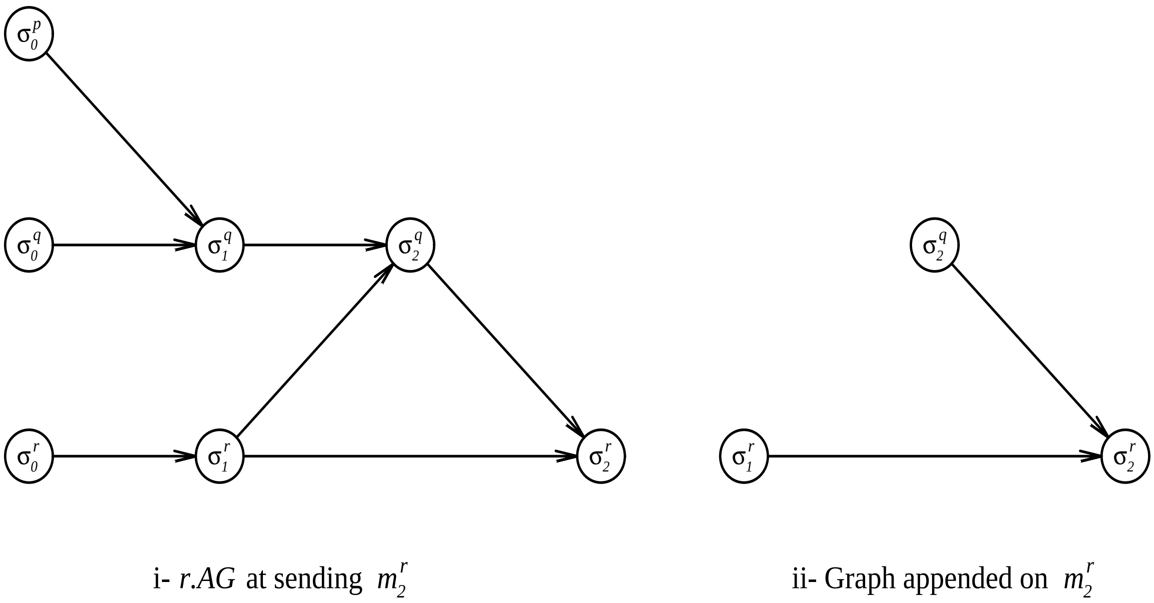


Figure 2.4 Reducing the size of the antecedence graph piggybacked to application messages: The antecedence graph piggybacked on m_2^r compared to the antecedence graph of the sender r.

header contains seven fields, as shown in Figure 2.5. A description of each field follows:

UID : A system-wide unique identifier of the message.

SENDER : The identifier of the sender of the message. All replicas of a replication-based *RU* have the same identifier.

IN : The incarnation number of the sender of the message.

SI : The state interval index of the sender.

CCN : The consistent checkpoint number at the sender when the message is sent (see Section 2.5).

KNOWN_SI : Represents the maximum state interval index of the receiver, as known to the sender.

AG : The antecedence graph piggybacked on the message, according to the message exchange protocol (may be empty).

Sending Messages

Figure 2.6 shows the procedure for preparing the header of an inter-*RU* application message before sending it. The procedure **ContructHeader()** is called with arguments s and r which are the identifiers of the sender and receiver, and m which represents the message to be sent. The sender starts by computing the antecedence graph that is piggybacked on the message. Sender s knows that the value in $s.LAST\_SI\,[r]$ is the maximum index of a state interval of s that r knows about, and therefore the corresponding antecedence graph is not included in $m.AG$. The antecedence graphs corresponding to state intervals of r that are in $s.AG$ are not included in $m.AG$ either. The sender constructs the message to be transmitted by filling in its fields with the appropriate values. **GenerateUid**() is a *deterministic* function that returns a system-wide unique identifier that serves to identify the message. The value in $s.KNOWN\_SI\,[r]$ is the maximum state interval of r known to s. It is included in the message to support incremental piggybacking of the antecedence graph at the receiver. *RU* s then calls **Transmit**(m, r) to transmit message m over the network to *RU* r. After sending the message, *RU* s copies it in the volatile message log.

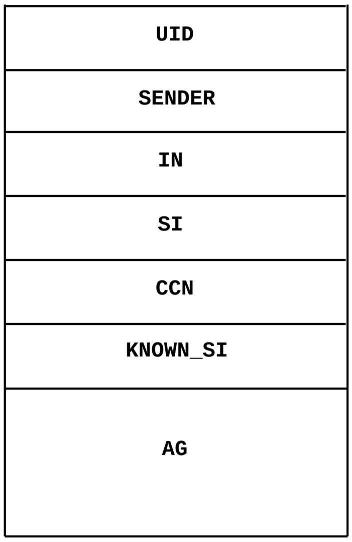


Figure 2.5 Manetho's message header.

---

procedure ConstructHeader(s, r, m)

$$
\begin{array}{lcl}
i & \leftarrow & s.SI; \\
j & \leftarrow & s.LAST\_SI\,[r]; \\
k & \leftarrow & s.KNOWN\_SI\,[r]; \\
m.AG & \leftarrow & AG(\sigma_i^s) - AG(\sigma_j^s); \\
m.AG & \leftarrow & m.AG\ - AG(\sigma_k^r); \\
m.CCN & \leftarrow & s.CCN; \\
m.UID & \leftarrow & \mathbf{GenerateUid}(); \\
m.IN & \leftarrow & s.INCNUM; \\
m.KNOWN\_SI & \leftarrow & s.KNOWN\_SI\,[r]; \\
m.SI & \leftarrow & s.SI; \\
m.SENDER & \leftarrow & s.ID;
\end{array}
$$

$\mathbf{Transmit}(m, r);$

$s.MSGLOG \leftarrow s.MSGLOG\ \cup\ (m.UID, m);$

return;

Figure 2.6 Sending an application message in Manetho.

Receiving Messages

Figure 2.7 shows the procedure for receiving a message. Procedure **Receive()** is called with arguments m and r, representing the message and the receiving RU, respectively. The procedure begins by deciding whether the message can be received. First, the incarnation number tagging the message must be the same as the entry corresponding to the sender in the incarnation number vector $r.INCVEC$. Second, the antecedence graph piggybacked on the message must not carry a node that would violate the operation of the recovery protocol. Specifically, the message will be rejected if there is a node corresponding to a state interval σ_i^p whose index i is larger than the corresponding entry for p in $r.REJECT\_LIST$. The need for this test will be explained in the description of the recovery protocol (see Section 2.8).

If the message can be received, the next step is to perform the operations necessary for maintaining the antecedence graph. The entries corresponding to the sender s in $r.KNOWN\_SI$ and $r.LAST\_SI$ vectors are updated if necessary. The update may not be necessary if the information in the antecedence graph piggybacked on the message is already known to r. This can happen, for instance, if messages transmitted from the sender s are received out of order.

Next, a new state interval is created by incrementing the state interval index $r.SI$ and updating the volatile antecedence graph. The graph piggybacked on the message $m.AG$ is first merged with $r.AG$. Then, a new node reflecting the new state interval is added to the graph by the function **AddNewNode()**. Finally, the message is delivered to the application program by calling the function **Deliver()**.

2.4.3 Output Commit

To commit output, an RU must save its volatile antecedence graph on stable storage. Figure 2.8 shows procedure **OutputCommit()**, which is run by any RU when sending a message to the outside world. The parameters of the procedure are the RU's identifier p and the output message m.

OutputCommit() begins by writing a commit record containing the unique identifier of the output message in $p.P\_COMMITLOG$. The procedure is assumed to run as an atomic action that ensures all or nothing semantics.

```
procedure Receive (m, r)
    s ← m.SENDER ;
    if m.IN ≠ r.INCVEC [s] then
        return;
    endif

    if ∃ σ_i^p ∈ m.AG such that i > r.REJECT_LIST [p] then
        return;
    endif

    if r.LAST_SI [s] < m.KNOWN_SI then
        r.LAST_SI [s]     ←   m.KNOWN_SI ;
    endif
    if r.KNOWN_SI [s] < m.SI then
        r.KNOWN_SI [r]    ←   m.SI;
    endif

    r.SI   ← r.SI + 1;
    r.AG   ← r.AG ∪ m.AG;
    AddNewNode(r.SI , r.AG , s, m.UID , m.SI);

    Deliver(m, s);
return;
```

Figure 2.7 Receiving an application message in Manetho.

```
atomic procedure OutputCommit(p, m)
    p.P_COMMITLOG   ← p.P_COMMITLOG ∪ {m.UID };
    p.P_AG          ← p.P_AG ∪ p.AG;
    p.AG            ← φ;
    Transmit(m);
return;
```

Figure 2.8 Output commit.

2.4.4 Messages from the Outside World

Input messages from the outside world are added to the volatile antecedence graph. Figure 2.9 shows the procedure **Input()** which takes the message m and the RU r as arguments. A new state interval begins by receiving the message, and a corresponding node is added to the volatile antecedence graph. *The node contains the message data.*

In practice, the RU usually replies to the outside world acknowledging the receipt of the input message. This reply invokes the output commit procedure, which saves the volatile antecedence graph in the persistent version. Therefore, there is only a small likelihood that an input message is piggybacked on any application message.

2.4.5 Discussion

Manetho's communication protocol provides fast output commit and a low runtime overhead. The overhead is low because no synchronous logging to stable storage takes place except during output commit. Therefore, the protocol behaves like optimistic rollback-recovery systems that use asynchronous logging [Joh89, SY85]. Unlike these systems, an output commit does not require several synchronization messages or several stable storage accesses. Committing output in Manetho only requires flushing the local antecedence graph in a single access to stable storage. Latency of output commit is thus lowered because of the reduction in the amount of information that has to be logged on stable storage and the elimination of multihost coordination.

2.5 Manetho's Checkpointing Protocol

2.5.1 Description

The RU*'s* that use rollback-recovery occasionally save their states on stable storage. These checkpoints reduce the amount of execution replay that would be necessary should a failure occur. The RU*'s* coordinate their checkpoints to form a system wide consistent state. A consistent state is one that could have occurred during some failure-free execution of the application program [CL85].

One distinguished RU acts as a *coordinator* and sends *marker* messages to all RU*'s* to start a consistent checkpoint. Each RU maintains one *permanent* checkpoint, belonging to the most recent consistent checkpoint. During each run of the protocol, each RU takes a *tentative* checkpoint, which replaces the permanent one only if the protocol terminates successfully [KT87]. Each RU also maintains a consistent check-

procedure Input (m, r)
$s \leftarrow m.SENDER$;
$r.SI \leftarrow r.SI + 1$;
AddNewNode$(r.SI, r.AG, s, m)$;

Deliver(m, s);
return;

Figure 2.9 Receiving a message from the outside world.

point number (CCN), which is incremented by one each time a consistent checkpoint is started. This number tags every message exchanged under Manetho's communication protocol (see Figure 2.5). The CCN enables the protocol to run in the presence of message reordering or loss [BCS84, LY87]. Additionally, each RU includes in the checkpoint the state interval index (P_SI) during which the RU took the checkpoint. Thus, if an RU p fails, it will restart at state interval $p.P\_SI$.

The protocol proceeds as follows:

1. The *coordinator* starts a new consistent checkpoint by incrementing CCN and sending *marker* messages [CL85] that contain CCN to each RU in the system.

2. Upon receiving a *marker* message, an RU starts a tentative checkpoint, if it has not started one already. The RU also starts a tentative checkpoint when it receives an *application* message whose CCN is greater than the local CCN. Since this message was sent *after* its sender has started participating in the consistent checkpoint, the receiver must save its state before receiving this message to maintain the consistency of the global checkpoint [BCS84, LY87].

3. After the tentative checkpoint has been written on stable storage, the RU sends an *acknowledgment* message to the coordinator.

4. The coordinator collects the responses from all RU's, and if all tentative checkpoints have been successful, it broadcasts a *success* message; otherwise, it broadcasts an *abort* message. When an RU receives a *success* message from the coordinator, it makes the tentative checkpoint permanent and discards the previous

permanent checkpoint. If the *RU* receives an *abort* message from the coordinator, it discards the tentative checkpoint and decrements its *CCN*.

2.5.2 Discussion

Manetho's rollback-recovery uses coordinated checkpointing, departing from the traditional design of systems based on message logging. Such systems exclusively used independent checkpointing in combination with message logging, in the belief that the cost of coordination is prohibitively expensive. In fact, the original reason for advocating message logging was to avoid the overhead of coordinating the checkpoints in a distributed system [SY85].

The decision to use coordinated checkpointing in Manetho was motivated by two important factors. First, a performance study based on actual implementation has shown that the overhead of coordinating the checkpoints is negligible. This performance study is reported in Chapter 4. Second, coordinated checkpointing eliminates the problem of garbage collection of recovery information. Because the consistent checkpoint forms a recovery line beyond which no rollback is necessary, information describing the computation that occurred before the checkpoint are removed without an explicit garbage collection protocol.

Given these two factors, coordinated checkpointing is the method of choice in conjunction with message logging. The purpose of message logging becomes the provision for faster output commit, rather than the elimination of checkpoint coordination. This observation is one of the contributions of this dissertation.

2.6 Manetho's Multicast Protocol

Manetho uses active replication to provide high availability to server applications. Replication follows the leader/cohort model [Bir85, BMST92], where an r-resilient replicated *RU* consists of a *leader* and r cohorts that execute the same application program. Each replica maintains its own copy of the *RU's* data structures described in Section 2.4. Following the terminology used in CIRCUS [Coo85], a replicated *RU* is also called a troupe.

The system converts an application message sent to a troupe into a multicast that is sent to every replica. The sender of the message is not aware that it is communicating with a troupe. To further hide the replication aspects of a troupe, only the leader sends application messages over the network on behalf of the replicas.

To prevent the executions of the replicas of the same troupe from diverging, the multicast protocol ensures that each application message is delivered in the same order at each replica.

2.6.1 Protocol Operation

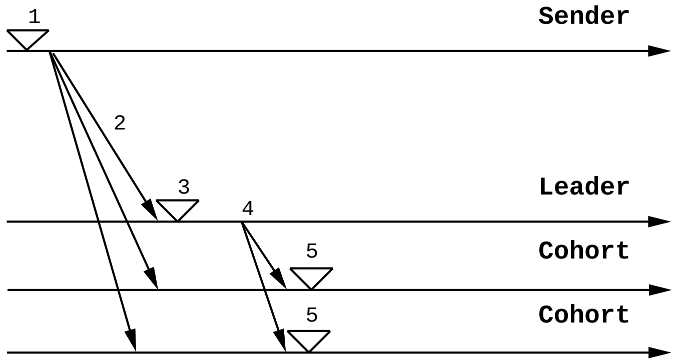
The leader of a troupe participates in the inter-*RU* communication protocol with other *RU's*. It is also responsible for representing the troupe in the recovery protocol. The cohorts do not send application messages over the network. They are silent performers that receive the same messages in the order defined by the leader and therefore their executions remain identical to the leader's execution.

When a troupe receives an application multicast, the leader defines the receipt order and sends a *sequence multicast* to its cohorts. This sequence multicast contains the receipt order, the application multicast's unique identifier, and the identifier of the sender. The leader delivers the message to the application program according to the **Receive()** procedure without waiting for the corresponding sequence multicast to reach the cohorts. The application program executing at the leader consumes the message and may send messages to other *RU's* reflecting the message receipt. Thus, there is no delay in delivering application messages at the leader.

After a cohort receives an application multicast, it expects the corresponding sequence multicast within a short period. When the cohort receives the sequence multicast, it delivers the message to the application program as described in **Receive()** in the specified order. The cohort does not acknowledge receiving the sequence multicast. Figure 2.10 shows the steps involved in receiving an application multicast.

2.6.2 Cohort Synchronization

A cohort may miss an application multicast, its corresponding sequence multicast, or both because of communication failures. To prevent a cohort from "falling behind" its leader by missing both multicasts for several consecutive application messages, the leader expects each cohort to send a periodic one-to-one synchronization message that includes the maximum state interval index known to the cohort. The leader's reply to a synchronization message contains the unique identifier, the sender's identifier, and receipt order for each application multicast that the cohort has missed, if any. The cohort uses this information to request the corresponding application message from the sender and deliver it in the order specified by the leader.



Receiving a multicast:
1- Application sends a message.
2- System transmits message in a multicast.
3- Leader delivers message.
4- Leader sends a sequence multicast.
5- Cohorts deliver message in the order defined.

Figure 2.10 Receiving a multicast.

2.6.3 Handling Communication Failures

Manetho detects and recovers lost multicasts as follows:

- When a cohort receives a sequence multicast for an application multicast that it has not received, the sequence multicast contains the identifiers of the application multicast and its sender. The cohort uses these identifiers to request a retransmission of the application multicast from the sender's message log.
- If a cohort receives a sequence multicast that is out of order, it will detect that it has missed one or more application multicasts. The cohort synchronizes with the leader by sending a synchronization-message as described in Section 2.6.2.
- When a cohort receives an application multicast, it expects to receive the corresponding sequence multicast shortly after that. If the sequence multicast is not received, the cohort requests it from the leader. This request contains the identifiers of the application multicast and the sender.
- The leader learns that it has missed an application multicast if it receives from one of its cohorts a request for a sequence multicast corresponding to an application multicast that the leader has not received. The leader requests the retransmission of the multicast from the corresponding sender's log.

- During troupe synchronization, a cohort finds the set of missed application or sequence multicasts, if any.

2.6.4 Comparison with Other Multicast Protocols

Existing application-transparent multicast protocols trade latency in delivering multicast messages to the application program against the number of control messages. In positive acknowledgment protocols, such as the original implementation of ABCAST of ISIS [BJ87b], the receivers run an agreement protocol to determine the receipt order of each application multicast. The multicast is delivered as soon as its receipt order is agreed on, at the expense of the overhead caused by the control messages necessary to reach agreement. For example, the two-phase agreement protocol of the old implementation of ABCAST requires r point-to-point messages and one overhead multicast to determine the receipt order of an application multicast sent to r receivers.

On the other hand, negative acknowledgment protocols reduce the number of control messages by piggybacking the ordering information on application multicasts [CM84, Kaa92, MSMA90]. However, reducing the number of control messages or eliminating them altogether introduces latency in achieving agreement on the receipt order of an application multicast, delaying the delivery of the multicast to the application program. For example, the r-resilient protocol by Chang and Maxemchuck requires only one overhead message per application multicast [CM84]. However, it cannot deliver a message to the application program until $r - 1$ "token transfers" have occurred, each requiring one application message to be received or a timeout to expire. Thus, an application message cannot be delivered until $r - 1$ subsequent application messages are received [CM84].

Manetho exploits the fact that active replication does not need the full generality and functionality of existing multicast protocols. Like negative acknowledgment multicast protocols, Manetho reduces the overhead during failure-free operation. A cohort does *not* acknowledge receiving application multicasts and it acknowledges the sequence multicasts only during synchronization. If multicasts are seldom lost, the overhead of acknowledgments will be eliminated, matching well with modern networks where communication failures are infrequent. Unlike negative acknowledgment protocols, Manetho's multicast avoids the latency in message delivery. The leader delivers the messages to the application program *without* waiting for the corresponding sequence multicasts to reach every cohort. Similarly, a cohort delivers the message to the application program as soon as the corresponding sequence multicast is available, even if it does *not* reach the rest of the cohorts.

The above analysis shows that Manetho is well tuned for environments where communication and host failures are infrequent. Modern networks and workstations are in line with these assumptions. However, combinations of leader and communication failures may leave the cohorts in a state inconsistent with the state of the failed leader. Consider the example shown in Figure 2.11. The leader receives a multicast m, transmits the corresponding sequence multicast, and delivers the message to the application program. The leader sends an application message m' back to the sender of m and then fails. Meanwhile, the sequence multicast carrying the receipt order was lost. The cohorts are not aware of the receipt of message m, while the sender of m received a message m' reflecting the receipt of m by the troupe. Manetho's recovery protocol resolves such inconsistencies (see Section 2.8).

2.7 Garbage Collection

The checkpointing protocol of Manetho establishes a recovery line beyond which no rollback is ever necessary. Therefore, recovery information describing events that occurred before the checkpoint is discarded, since it is not needed by the recovery protocol. This information includes the nodes of the antecedence graph belonging to events that occurred before the checkpoint, messages sent and received before the checkpoint, and older checkpoints. Thus, the *RU's* automatically discard useless recovery information without using an explicit garbage collection protocol. As described so far, this scheme does not allow the *RU's* to reclaim cross checkpoint messages, or messages received by the replicated *RU's* that do not participate in the checkpointing protocol. Cross checkpoint messages are those that have been transmitted before the sender takes the checkpoint but are received after the receiver has taken the checkpoint. The following two extensions to Manetho's communication protocol enable the garbage collection scheme to be applied on any message, regardless of which *RU* received it or when.

- Cross checkpoint messages are detected by comparing the *CCN* field in the message with the *CCN* of the receiver, and are treated as if they were input messages from the outside world.

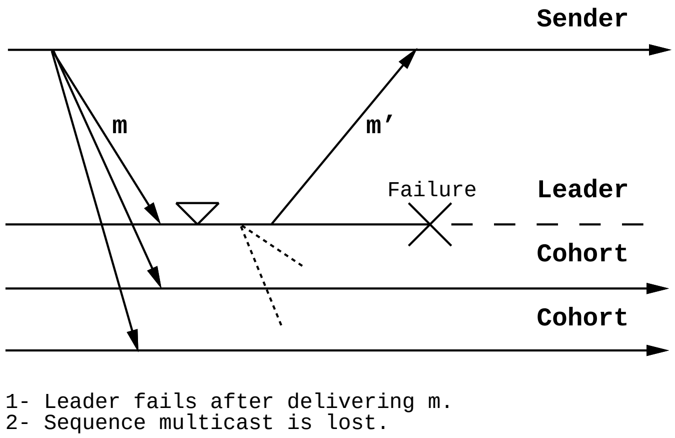


Figure 2.11 A failure scenario.

- Replication-based *RU's* participate in the coordinated checkpointing protocol by synchronizing their states. During this synchronization, the leader forces all replicas to agree on the set of messages that were received and their order. This synchronization is the equivalent of taking a checkpoint for replication-based *RU's*. When all replicas agree on the messages received, a cohort's execution always reaches the state interval during which the leader forced the synchronization. This is true because the execution is deterministic, and a description of all events that culminated in that state interval is available to the cohorts.

2.8 Manetho's Recovery Protocol

2.8.1 Overview

Consider first *RU's* based on rollback-recovery. Manetho's recovery protocol is based on a simple idea. Call a state interval σ_i^p *visible* outside p if another RU q has a node that corresponds to σ_j^p in $q.AG \cup q.P\_AG$, or if p commits output from state σ_j^p, where $j \geq i$. Thus, $AG(\sigma_i^p)$ is a subset of $q.AG \cup q.P\_AG$ or of $p.P\_AG$. If p fails, it will restore $p.AG$ using $p.P\_AG$, and then request the antecedence graphs of its visible state intervals from all other *RU's*. Thus, p reconstructs the antecedence graph of the visible state interval with the largest index. Call this state interval σ_m^p. $AG(\sigma_m^p)$ contains a description of the pre-failure execution up to σ_m^p. Using this description, p restarts from state interval $\sigma^p_{p.P\_SI}$ and recreates the events that started all state intervals σ_i^p, where $p.P\_SI \leq i \leq m$. For an internal or an input event, the graph contains the necessary information to recreate it. For a message receipt event involving a cross checkpoint message, a copy of the message and the order in which it should be received are available in the reconstructed antecedence graph. For a message receipt event involving an ordinary message, p requests the message from its sender's log using the unique message identifier indicated in the reconstructed antecedence graph. If the sender also fails, it will recreate its message log during its own recovery, and the message will then be available for replay. Thus, p reconstructs the execution up to a state that is consistent with all its visible state intervals.

2.8.2 Example

Figure 2.12 shows a failure scenario that occurs in a system of four *RU's*. In this scenario, *RU's* p, q and r fail while s continues to operate. *RU* p has also sent a message m_2^p to the outside world before the failure. At the beginning of recovery, $AG(\sigma_2^p)$ is available since it was saved in $p.P\_AG$ when output message m_2^p was committed. In addition, $AG(\sigma_1^s)$ is available in $s.AG$. Both these antecedence graphs are shown in Figure 2.13. Furthermore, assume that each of $p.P\_SI$, $q.P\_SI$ and $r.P\_SI$ is 0. Thus, p, q, and r restart execution from state intervals σ_0^p, σ_0^q and σ_0^r, respectively. A description of the steps that take place during recovery follows:

1. Each recovering *RU* starts by communicating with all other *RU's* to collect their antecedence graphs. Then, it extracts the subgraph of its most recent state interval from the collected graphs. Thus, p reconstructs $AG(\sigma_2^p)$, q reconstructs $AG(\sigma_1^q)$, and r reconstructs $AG(\sigma_2^r)$.

2. Each recovering *RU* starts execution replay, recreating the events as described in the reconstructed antecedence graph.

3. Because the execution in each state interval is deterministic, *RU* p regenerates m_1^p and inserts it in $p.MSGLOG$, but does not resend it over the network to q. If q has not failed, then the retransmission of the message does not serve any purpose. Otherwise, q is recovering and it will request the message if needed.

4. *RU* p proceeds to recreate the internal event that started σ_1^p. The necessary information to recreate the event is available in $p.P\_AG$.

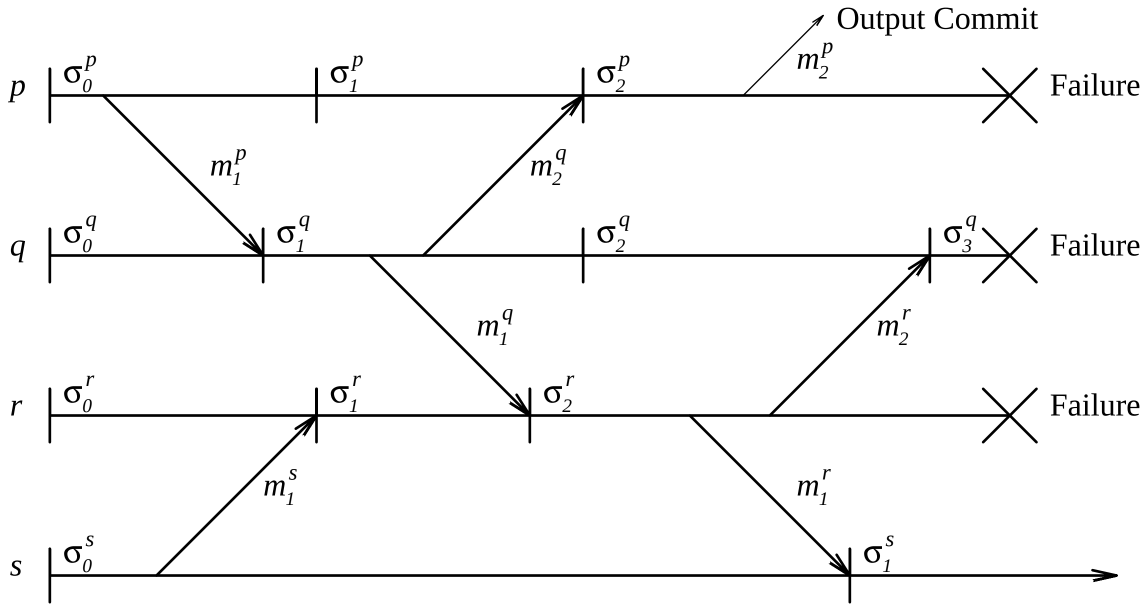


Figure 2.12 A failure scenario.

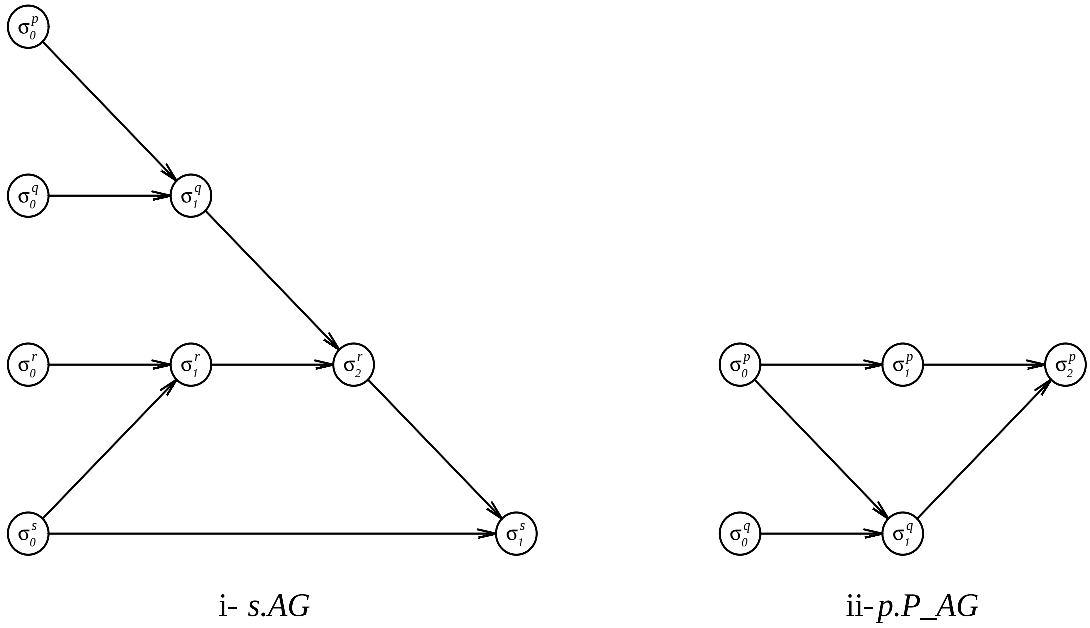


Figure 2.13 Available antecedence graphs at the beginning of recovery.

5. $RU\ p$ proceeds with the replay to find that it should receive message m_2^q to start interval σ_2^p. Therefore, p requests a replay of the message from q.

6. Meanwhile, recovery in $RU\ q$ is in progress. $RU\ q$ restarts execution and requests the replay of m_1^p as indicated by the reconstructed antecedence graph. This message is regenerated during recovery and is available in $p.MSGLOG$.

7. $RU\ q$ starts state interval σ_1^q and continues execution. It regenerates messages m_1^q and m_2^q. Again, it does not resend these messages, but only saves them in $q.MSGLOG$.

8. At this point, $RU\ q$ has replayed its execution as defined by the antecedence graph reconstructed during recovery. It now resumes normal operation.

9. Now that m_2^q has been regenerated, $RU\ q$ transmits it to $RU\ p$ in response to its request. $RU\ p$ receives the message and proceeds with its replay. It then regenerates message m_2^p. $RU\ p$ does not retransmit the message to the outside world, since there is a corresponding record in $p.P\_COMMITLOG$ showing that the message has been transmitted before the failure. Recovery of $RU\ p$ is now complete and it resumes normal operation.

10. Meanwhile, $RU\ r$'s recovery is progressing in parallel. $RU\ r$ starts by requesting message m_1^s from s. This message is available in $s.MSGLOG$ and is replayed.

11. After receiving m_1^s, $RU\ r$ proceeds to request m_1^q from $RU\ q$. This message is available in $q.MSGLOG$ and is replayed.

12. After receiving m_1^q, the recovery of $RU\ r$ is complete. It now resumes normal operation.

After executing the recovery protocol, the recovering RU's have restored a global state consistent with the state of the functioning $RU\ s$, and with the output committed by $RU\ p$. Therefore, $RU\ s$ does not need to roll back. Furthermore, the outside world has not been affected by the failure.

State intervals σ_2^q and σ_3^q have not been recreated. Information about these state intervals could not be recovered from the antecedence graph reconstructed during recovery. However, since these state intervals are not visible to any functioning RU or in the outside world, recreating these states is not necessary. The recovery protocol allows q to go through some legal execution that preserves the consistency of the system, but is not necessarily identical to the pre-failure execution.

2.8.3 Troupe Recovery

Handling troupe failures depends on whether the leader or a cohort fails.

Cohort Failure and Recovery

Handling cohort failures is simple. The leader stops accepting any message from the failed cohort and operation continues without interruption. It may be desirable to regenerate a new cohort to maintain the resilience of the troupe. This regeneration is done in parallel with the troupe operation by copying the state of the leader into a fresh cohort.

Leader Failure

To recover from a leader failure, the troupe members must elect a new leader and run Manetho's recovery protocol. The "checkpoint" used in this case is the state of the new leader. If no communication failure has occurred between the last cohort synchronization and the leader failure, this "checkpoint" is identical to the state of the failed leader before it failed, and recovery is almost instantaneous.

Running the recovery protocol is required so that the new leader can restore a state consistent with the rest of the system. The failed leader may have received and delivered several application messages before it failed. The corresponding sequence multicasts of these messages, or indeed the messages themselves, may not have reached some or all of the cohorts because of communication failures. Thus, the new leader must determine if the failed leader has received and delivered application messages that the cohorts are not aware of because of a combination of communication and leader failure.

Leader Election

If the leader of a troupe fails, the cohorts will use the following protocol to elect a new leader. This protocol is an adaptation of the *invitation* protocol [GM82] in which the winner of the election is the cohort that has the highest state interval index.

- A cohort starts leader election by sending a *recovery-multicast* to the other cohorts in the troupe. The multicast contains the cohort's current state interval index.

- When a replica receives a recovery-multicast carrying a state interval index larger than its own, it sends back a *leadership-acknowledgment* message, and aborts its own bid for leadership, if it has started one. Otherwise, when a replica receives a recovery-multicast with a state interval index smaller than its own, it starts its own leadership bid, if it has not already done so. Ties are broken arbitrarily.

- The initiator collects the responses from every member of the troupe. It retransmits the recovery-multicast until it receives a corresponding leadership-acknowledgment from every surviving member.

- The new leader increments the troupe incarnation number on stable storage.

- The new leader forces the cohorts to synchronize to inform them of its state interval index. The leader also informs the cohorts of the new incarnation number during synchronization.

If there is at least one surviving troupe member, the protocol elects a single leader and terminates [GM82]. If the initiator of the protocol fails, the protocol is simply restarted.

2.8.4 Formal Description

Figure 2.14 shows the recovery protocol. A failed RU starts recovery by calling the procedure **Recover()**. The procedure's argument is the recovering RU*'s* identifier. **Recover()** starts by restoring the RU*'s* state from stable storage for a rollback-recovery based RU, or by electing a new leader for a failed troupe. Then, the RU increments its incarnation number on stable storage, *p.P_INCNUM*. It also sets the volatile version *p.INCNUM* to the new value.

After updating the incarnation number, the RU proceeds to reconstruct its state interval index and antecedence graph from the corresponding persistent versions. The RU then executes a remote call of procedure **CollectGraph()** at each RU (except itself). These remote procedure calls may execute in parallel. The messages exchanged

procedure Recover(p)
  restore state from checkpoint or elect new leader
  $p.INCNUM \leftarrow p.P\_INCNUM + 1$;
  $p.P\_INCNUM \leftarrow p.INCNUM$;
  $p.SI \leftarrow p.P\_SI$;
  $p.AG \leftarrow p.P\_AG$;

  for all $q \in SYSTEM$, $q \neq p$ **do**
    remote call at q: **CollectGraph**(q, p, INC, AG);
    $p.INCVEC[q] \leftarrow INC$;
    $p.AG \leftarrow p.AG \cup AG$;
  Synchronize cohorts if a troupe
  for all $q \in SYSTEM$, $q \neq p$ **do**
    remote call at q: **UpdateInc**($q, p.INCVEC$);

  $p.P\_AG \leftarrow p.AG$;
  $m \leftarrow$ **max** i **such that** $\sigma_i^p \in p.P\_AG$;

  while $p.SI < m$ **do**
    Replay(p);
    $p.SI \leftarrow p.SI + 1$;
  endwhile
return;

procedure CollectGraph(q, p, INC, AG)
  $q.P\_AG \leftarrow q.P\_AG \cup q.AG$;
  $q.AG \leftarrow \phi$;
  $INC \leftarrow q.INCNUM$;
  $k \leftarrow$ **max** i **such that** $\sigma_i^p \in q.P\_AG$;
  $AG \leftarrow AG(\sigma_k^p)$;

  $q.REJECT\_LIST[p] \leftarrow k$;
return;

procedure UpdateInc($q, p, INCVEC, SYSTEM$)
  for all $r \in SYSTEM$ **do**
    $q.INCVEC[r] \leftarrow$ **max**($INCVEC[r], q.INCVEC[r]$);

  $q.REJECT\_LIST[p] \leftarrow \infty$;
return;

Figure 2.14 The recovery protocol.

for the purpose of recovery are considered "out-of-band" messages and do not follow the inter-*RU* communication protocol described in Section 2.4.2.

At *RU* q, the procedure **CollectGraph()** begins by merging the volatile version of its antecedence graph $q.AG$ into its persistent version $q.P\_AG$. It then determines σ_k^p, the state interval of p with the maximum index such that the state interval has a node in $q.P\_AG$. Next, it adds k to $q.REJECT\_LIST$, and until it receives an **UpdateInc()** call from p, *RU* q rejects any *application message* (from any sender) whose piggybacked antecedence graph contains a node of any state interval σ_i^p, where $i > k$. Then, q returns its incarnation number and $AG(\sigma_k^p)$ to p. Note that recovering *RU's* also respond to **CollectGraph()** calls.

When a **CollectGraph()** call from q returns, p merges the returned antecedence graph into $p.AG$. It also updates the entry corresponding to the incarnation number of q in $p.INCVEC$ using the returned value *INC*. After all the **CollectGraph()** calls have returned, p calls the remote procedure **UpdateInc()** at every *RU* (except itself). In this procedure, the incarnation vector $q.INCVEC$ is updated using the information that p gathered from the **CollectGraph()** calls. *RU* q also removes any restriction on accepting application messages by updating its *REJECT_LIST* vector.

RU p proceeds to determine m, the maximum state interval index in $p.AG$, and recreates the pre-failure execution up to state interval σ_m^p. The events are replayed as specified in the reconstructed antecedence graph. *RU* p requests the application messages from their senders' logs to replay message receipt events if no cross checkpoint messages are involved. For message receipt events involving cross checkpoint messages, the reconstructed antecedence graph contains a copy of the message's data. Similarly, the reconstructed antecedence graph contains the necessary information to replay input and internal events. While recreating the execution, *RU* p does not resend application messages to other *RU's* to avoid sending duplicate messages, but it updates its message log by adding a copy of each such message to the log. The messages that were transmitted and received before the failure are indicated in the reconstructed antecedence graph. *RU* p also uses $p.P\_COMMITLOG$ to prevent the retransmission of duplicate messages to the outside world.

Execution replay for troupes is identical, except that the new leader forces a synchronization step after receiving replies from all **CollectGraph()** calls. This synchronization forces each cohort to replay the execution of the old leader, if at all necessary.

2.9 Correctness

First, it must be shown that the graph reconstructed by **Recover()** is indeed the antecedence graph of σ^p_m.

Lemma 2.1 Before calling **Replay()** in **Recover()**

$$p.P\_AG = AG(\sigma^p_m)$$

Proof

$p.P\_AG \subseteq AG(\sigma^p_m)$: Initially, $p.P\_AG$ is set to $AG(\sigma^p_{p.P\_SI})$. Therefore at the end of **Recover()**:

$$g \in p.P\_AG \Rightarrow (g \in AG(\sigma^p_{p.P\_SI})) \vee (g \in p.P\_AG - AG(\sigma^p_{p.P\_SI}))$$

Case 1: $g \in AG(\sigma^p_{p.P\_SI})$:

$$p.P\_SI \leq m$$
$$\Rightarrow AG(\sigma^p_{p.P\_SI}) \subseteq AG(\sigma^p_m)$$
$$\Rightarrow g \in AG(\sigma^p_m)$$

Case 2: $g \in p.P\_AG - AG(\sigma^p_{p.P\_SI})$: $\exists q$ such that in p's **CollectGraph()** call at q, $g \in AG(\sigma^p_k)$. But $k \leq m$, therefore $AG(\sigma^p_k) \subseteq AG(\sigma^p_m)$. Thus, $g \in AG(\sigma^p_m)$.

$AG(\sigma^p_m) \subseteq p.P\_AG$: There are two cases to consider:

Case 1: $m = p.P\_SI$: Obvious.

Case 2: $m > p.P\_SI$: For $RU\ q$ where **CollectGraph()** returns $AG(\sigma^p_m)$, if $RU\ q$ has a complete representation of $AG(\sigma^p_m)$, then

$$AG(\sigma^p_m) \subseteq p.P\_AG$$

Otherwise, **CollectGraph()** at q returns a subset of $AG(\sigma^p_m)$. But q has a subset of $AG(\sigma^p_m)$ only if other *RU's* have saved the missing subgraphs on stable storage before sending the messages that should have included them. These *RU's* return the missing subgraphs during p's **CollectGraph()** calls, despite of any failure. Therefore,

$$AG(\sigma^p_m) \subseteq p.P\_AG$$

□

2.9.1 Orphan State Intervals

The recovery protocol reconstructs $AG(\sigma_m^p)$. But the execution of the previous incarnation of RU p may have proceeded beyond state interval σ_m^p. All state intervals σ_i^p that occurred in the previous incarnation of p, where $i > m$, are called *orphan* state intervals.

> **Definition 2.1** Consider the recovery during incarnation $u+1$ of RU p. Define the orphan state intervals σ_i^p as those that occurred during incarnation u of RU p, where $i > m$.

A state interval σ_i^p cannot become an orphan if $AG(\sigma_i^p)$ is in $p.P\_AG$.

> **Lemma 2.2** If $AG(\sigma_i^p) \subseteq p.P\_AG$, then σ_i^p will never become an orphan state interval.

Proof If p fails, its next incarnation will reconstruct $AG(\sigma_i^p)$ when it reads $p.P\_AG$ in **Recover()**. Therefore, $i \leq m$, and σ_i^p is not an orphan state interval. The lemma also holds for future incarnations of p, since the information in $p.P\_AG$ persists across failures. □

2.9.2 Safety

Orphan state intervals represent executions that are not replayed by the recovery protocol. There are several important properties concerning orphan state intervals. First, orphan state intervals cannot directly affect the outside world.

> **Lemma 2.3** No output is committed from an orphan state interval.

Proof Assume that orphan state interval σ_i^p at incarnation u of RU p succeeded in committing output. Therefore, $AG(\sigma_i^p) \subseteq p.P\_AG$ before incarnation $u+1$ calls **Recover()**. But $p.AG$ is initialized to $p.P\_AG$ at the beginning of **Recover()**. Therefore the following holds after reconstructing $AG(\sigma_m^p)$ during incarnation $u+1$:

$$AG(\sigma_i^p) \subseteq AG(\sigma_m^p)$$

which is a contradiction, since $i > m$. □

The second safety property of the recovery protocol concerns the effect of orphan state intervals on the RU's in the system.

Lemma 2.4 Consider the recovery of $RU\ p$. At the point where p has received the replies to all the **CollectGraph()** calls but has not yet issued any **UpdateInc()** call, no node that corresponds to an orphan state interval of p exists in the antecedence graph of any RU.

Proof When p's **CollectGraph()** call executes at any $RU\ q$, no state interval σ_i^p, where $i > m$, has a corresponding node in $q.AG \cup q.P\_AG$. After returning p's **CollectGraph()**, $RU\ q$ uses $q.REJECT\_LIST$ to discard any message whose piggybacked graph contains a node representing σ_i^p, where $m < i$, until receiving p's **UpdateInc()**. □

A message sent from an orphan state interval reflects a state that may not be recoverable. Such a message is always rejected.

Lemma 2.5 An RU always rejects messages sent from orphan state intervals.

Proof Let v be the incarnation number of $RU\ p$. Assume that $RU\ p$ at incarnation u, where $u < v$, sends a message m_j^p to $RU\ q$ from an orphan state interval σ_i^p, q always rejects m_j^p.

Let w be the incarnation number of $RU\ q$. There are two cases to consider:

Case 1: Incarnation w of $RU\ q$ executed a **CollectGraph()** call during its recovery at incarnation v of p. In this case, the reply of p to q contains v, the incarnation number of p. When q receives m_j^p, it detects that

$$m_j^p.IN = u, \quad u < v$$

and therefore discards the message.

Case 2: Incarnation v of $RU\ p$ executed a **CollectGraph()** call at incarnation w of $RU\ q$. There are two cases to consider:

case i: Message m_j^p arrives at $RU\ q$ before the **CollectGraph()** call of p executed at q. Assume that q receives m_j^p, then

$$m_j^p.AG \subseteq q.AG$$

$$\Rightarrow AG(\sigma_i^p) \subseteq (q.AG \cup q.P\_AG)$$

In this case, the reply to **CollectGraph()** call of p contains $AG(\sigma_i^p)$. Hence, σ_i^p is not an orphan by definition, since it is computed by the recovery protocol, a contradiction.

case ii: Message m_j^p arrives at $RU\ q$ after p's **CollectGraph()** executed at q. In this case, q detects that the incarnation number tagging the message is old, and rejects it.

□

The final safety property of the protocol concerns the consistency between any two *RU's* [CL85]. Lemma 2.4 shows that before p issues any **UpdateInc()** call, the antecedence graph at any $RU\ q$ does not contain any state interval σ_i^p where $i > m$. Therefore, no $RU\ q$ contains in its antecedence graph a node representing a state interval of p that will not be recovered. Furthermore, Lemma 2.5 shows that after the **UpdateInc()** calls are issued, no $RU\ q$ contains in its antecedence graph a node representing an orphan state interval of p. It remains to show that during its recovery, p does not include in its antecedence graph a node corresponding to an orphan state interval σ_i^q.

Lemma 2.6 $\forall\, i, q$ such that $\sigma_i^q \in AG(\sigma_m^p), AG(\sigma_i^q) \subseteq (q.AG \cup q.P\_AG)$.

Proof If q is functional when it returns p's **CollectGraph()** call, then the lemma is true regardless of any subsequent failures of q, since q merges $q.AG$ into $q.P\_AG$ before returning the call.

Otherwise, $RU\ q$ was recovering when it returned p's **CollectGraph()** call. There are two cases:

Case 1: $AG(\sigma_i^q)$ was returned to p by some functional $RU\ r$. There are three cases:

case i: $RU\ r$ returned p's **CollectGraph()** call before q's call executed at r. Thus, r has saved $AG(\sigma_i^q)$ in $r.P\_AG$ during p's call. Regardless of future failures of r or q, r will return $AG(\sigma_i^q)$ to q during its **CollectGraph()** call at r.

case ii: $RU\ r$ returned p's **CollectGraph()** call after q's **CollectGraph()** but before q's **UpdateInc()**. Then $AG(\sigma_i^q)$ must have been returned to q's call, since r could not have added $AG(\sigma_i^q)$ to $q.AG \cup q.P\_AG$ after q's call by Lemmas 2.4 and 2.5. This is also true if r subsequently fails, because a recovering RU does not accept application messages until it finishes recovery.

case iii: $RU\ r$ executed q's **UpdateInc()** call before p's **CollectGraph()**. Lemmas 2.4 and 2.5 show that σ_i^q cannot be an orphan state interval, and therefore $AG(\sigma_i^q)$ is in $q.P\_AG$.

Case 2: $AG(\sigma_i^q)$ is not a subgraph of the antecedence graph of the current state interval of any functional RU. Hence, either $AG(\sigma_i^q) \subseteq AG(\sigma^p_{p.P\_SI})$, in which case p returns $AG(\sigma_i^q)$ during q's **CollectGraph()** call; or p must have received $AG(\sigma_i^q)$ from some $RU\ r$ that was recovering and had $AG(\sigma_i^q) \subseteq AG(\sigma^r_{r.P\_SI})$, in which case both p and q will receive $AG(\sigma_i^q)$ from r, regardless of any subsequent failures of p, q or r.

Thus, p never includes in its antecedence graph a node corresponding to an orphan state interval during recovery. Therefore, all functional RU's are consistent. □

2.9.3 Liveness

The next step is to show that the protocol indeed restores the computation of an $RU\ p$ up to state interval σ_m^p. After reconstructing $AG(\sigma_m^p)$, the protocol replays the execution of p up to σ_m^p despite any failure.

Lemma 2.7 The recovery protocol restores the execution of all recovering RU's despite any failure.

Proof Construct graph F from $AG(\sigma_m^p)$ by removing nodes that represent state intervals in functioning RU's or state intervals that occurred before $\sigma^q_{q.P\_SI}$ of each $RU\ q$. Graph F represents the state intervals that must be recreated during recovery of p.

Every state interval in F is replayed during recovery. The proof proceeds by induction on a topological sort of F. The topological sort must exist because F is acyclic.

Base case: Consider a state interval represented by a node f at level 0 of the topological sort of F. Node f represents the state interval with the smallest index to be restored in the corresponding RU. If the event that created the state interval is an internal or an input event, or a message receipt event involving a cross checkpoint message, the information contained in node f is sufficient to recreate the event. If the event is a message receipt not involving a cross checkpoint message, then the source of the message and its identifier are indicated in the node. By the construction of F, the source of the message must be a functional RU, in which case the message is available for replay in the corresponding message log.

Induction Hypothesis: Assume that the lemma is true for all nodes at topological level k.

Induction Step: Consider a node f at topological level $k+1$. If f corresponds to a state interval created by an internal or an input event, or by receiving a cross checkpoint message, then the corresponding state interval is recreated by starting execution from the previous state interval (which is reconstructed by the induction hypothesis) and replaying the event using the information in f. If f corresponds to a message receipt (and no cross checkpoint message is involved), then the corresponding state interval is reconstructed by starting the execution from the previous state interval and requesting the message to be replayed. The message is available either because it is available at the message log of a functional RU, or because it was recreated during recovery by the induction hypothesis (the graph is acyclic, so the state interval from which the message was sent must be at a level less than or equal to k).

□

The following lemma establishes a limit on the amount of execution that has to be replayed during recovery.

Lemma 2.8 For any RU p, no event that occurred before state interval $\sigma^p_{p.P\_SI}$ needs to be recreated.

Proof Follows immediately from the construction in the proof of Lemma 2.7. □

Theorem 2.1 The recovery protocol is deadlock-free.

Proof Because a recovering RU returns the **CollectGraph()** calls without waiting for its own recovery to terminate, no deadlock occurs during a **CollectGraph()** call. Lemma 2.7 also shows that no deadlock occurs while recreating the state intervals during execution replay. □

Theorem 2.2 The recovery protocol is livelock-free.

Proof Lemma 2.5 shows that no RU accepts a message transmitted from an orphan state interval. Therefore, no livelock occurs. □

Theorem 2.3 No domino effect occurs during recovery.

Proof Lemma 2.8 shows that a recovering RU does not need to recreate the execution before $\sigma^p_{p.P\_SI}$. Furthermore, no RU needs to restart execution to regenerate the messages required for the recovery of another RU. All such messages are available in the message logs of their senders. □

2.9.4 Interactions with the Outside World

Theorem 2.4 A state interval from which output is committed will be recovered.

Proof An RU p merges $p.AG$ into $p.P\_AG$ before committing output. Lemma 2.2 shows that this state interval never becomes an orphan. Lemma 2.7 shows that the execution up to this state interval is recreated regardless of any failure in the system. □

Theorem 2.5 No output is committed from an orphan state interval or one that depends on an orphan state interval.

Proof Lemma 2.3 shows that no orphan state interval can commit output, and Lemma 2.5 shows that RU's always reject messages sent from orphan state intervals, and therefore such a state cannot have an effect on the state of any RU after recovery. □

2.9.5 Nature of the Computation After Recovery

Sections 2.9.2 and 2.9.3 have shown that Manetho's recovery protocol is safe and live. However, the computation replayed by the protocol is not necessarily identical to the one that occurred before the failure. The recovery protocol guarantees only that the replayed computation is one that could have occurred during a legal execution of the system.

> **Definition 2.2** Two distributed computations are equivalent if and only if both start from the same initial state and produce the same sequence of output.

The execution of the computation after recovery is equivalent to some legal, failure-free execution.

> **Theorem 2.6** A failure-prone computation is equivalent to a failure-free computation.

Proof Let C be a failure-prone computation. Derive the failure-free computation C' from C by removing failures, recoveries, and orphan state intervals. Theorems 2.4 and 2.5 show that output sent in C is sent in C' and vice versa.

Next, it is shown that C' is a legal execution that could have occurred had there been no failure. The state intervals constituting C' could happen in a failure-free execution that starts in the same state from which C started. The proof proceeds by induction on the state intervals of C'.

Base Case: The construction of C' is such that the initial state of each RU occurs in both C and C'.

Induction Hypothesis: Assume that the subset of C' consisting of all state intervals that "happened before" σ_i^p can occur in a failure-free execution.

Induction Step: Consider the subset of C' consisting of state interval σ_i^p and all state intervals that "happened before" σ_i^p. All state intervals that precede σ_i^p have occurred in C by the induction hypothesis. The state transition from state interval σ_{i-1}^p to σ_i^p in C' is identical to that in C, because the execution is deterministic within the same state interval, and the same event that creates σ_i^p in C is used to create σ_i^p in C'.

□

2.9.6 Discussion

Manetho's recovery protocol has properties similar to those of pessimistic message logging systems. First, it limits rollbacks to only the *RU's* that fail. *RU's* that survive a failure are not rolled back. This property is key to the co-existence of rollback-recovery and active replication in the same system, since it protects replicated *RU's* from the effects of rollbacks in *RU's* that use rollback-recovery. Second, the protocol limits the rollback to the latest checkpoint, reducing execution replay to the minimum.

Manetho's recovery protocol differs from pessimistic message logging systems in that it allows orphan states to be created. However, such states cannot have an effect on the computation or on the outside world.

2.10 Summary

Manetho provides four protocols that allow transparent rollback-recovery to co-exist with active replication. At the heart of these protocols is the antecedence graph maintenance. The antecedence graph summarizes the nondeterministic events that occur during failure-free operation. If a failure occurs, a recovery protocol will use the information in the antecedence graph to recreate the nondeterministic events that occurred before the failure. The key feature of the antecedence graph is that it protects a functional *RU* from the effects of failures in other *RU's*. Thus, replication-based *RU's* are not forced to roll back because of the failures of rollback-recovery based *RU's*, and the two methods can therefore co-exist in the same system.

Manetho's communication protocol allows output to be committed from any *RU* without multihost coordination, and reduces failure-free performance overhead by avoiding synchronous logging of recovery information except during output commit. Manetho's checkpointing protocol uses consistent checkpointing, departing from the traditional design of message logging protocols. This design decision simplifies garbage collection. The recovery protocol limits rollback to only the *RU's* that fail, and only to the last state saved by failed *RU's*. This combination of advantages is unprecedented in any transparent rollback-recovery protocol that does not rely on special hardware support. Finally, Manetho's multicast protocol delivers application messages with high throughput like negative acknowledgment protocols, but without the latency of message delivery common in them.

The design of the recovery protocol has been proven correct. It remains to show by implementation that the performance claims are realized in practice. This is the subject of the following two chapters.

Chapter 3

Implementation

A prototype of Manetho has been implemented on a network multicomputer of 16 workstations. The prototype is an example of how to implement Manetho in practice and it also serves as a test bed for evaluating performance. This chapter describes the techniques used in the implementation.

3.1 Overview

Manetho has been implemented on a multicomputer consisting of a 10 Mbits/second Ethernet network that connects 16 Sun 3/60 workstations. Each workstation has a 20 MHz MC68020 microprocessor and four Mbytes of physical memory. The workstations run a version of the V-System [Che88], to which Manetho's mechanisms are added.

The application programs running on this multicomputer use the distributed processing facilities provided by the V-System. Examples include interprocess communication primitives, transparent remote execution, and libraries to support construction of distributed programs [VDG86]. In this environment, a recovery unit consists of an address space and the threads that manipulate it (a V-System's logical host [TLC85]).* It does not include the kernel or system servers, as there is no attempt to make them recoverable. A recovery unit interacts with the outside world by receiving input from the keyboard and producing output on a workstation's display. If a failure occurs, the V-System addressing mechanism allows a recovery unit to be reinstalled and run on any available machine on the network [TLC85].

Support for Manetho consists of several user-level servers and kernel routines. Collectively, they implement the four protocols described in Chapter 2. Recovery units based on rollback-recovery include support for the communication, checkpointing, and recovery protocols. Recovery units based on replication also include support

*A recovery unit is equivalent to a Mach task [ABB+86].

for the multicast protocol. Due to the added support for Manetho, the source code files for the kernel and system servers grew by about 26% and 40%, respectively. During runtime, Manetho also causes an increase in the sizes of the loaded kernel and system servers of about 20% and 60%, respectively. The system also consumes 512 Kbytes of the physical memory on each workstation for the volatile message log.

Two dedicated servers provide stable storage for the recovery information. Each server runs the V-System on a Sun 3/140 equipped with a 16 MHz MC68020 microprocessor, eight Mbytes of memory, and a 380-Mbyte Fujitsu Eagle disk. The servers mirror each other, protecting the system from susceptibility to a single point of failure. Note that stable storage cannot be provided by using a local disk or non-volatile memory residing in the recovery unit. In such a scheme, the recovery information will not be available during an extended outage of the recovery unit, thus subjecting the system to many single-point failures.

The implementation uses several techniques to reduce the failure-free performance overhead of the system:

- Copy-on-write extends the message log into the application's address space. This scheme increases the available memory for the log and reduces the probability of blocking the application because of log overflow.

- The implementation of the antecedence graph exploits the semantics of the V-System's communication protocol to reduce the number of nodes that have to be added to the antecedence graph.

- Incremental and concurrent checkpointing techniques are used to reduce the overhead of checkpointing. These techniques reduce the amount of state to be saved and reduce the probability of blocking the application program while the checkpoint is being taken.

- The implementation of Manetho's multicast exploits the efficiency of the underlying algorithm for flow control to reduce the overhead of synchronization messages.

The implementations of the communication, checkpointing, multicast, and recovery protocols are described in Sections 3.2, 3.3, 3.4 and 3.5, in that order. Next, Section 3.6 describes the implementation of stable storage, and Section 3.7 summarizes the chapter.

3.2 Manetho's Communication Protocol

3.2.1 Volatile Message Logging

On each workstation, the kernel allocates a circular buffer of 512 Kbytes of physical memory for the message log. The routine for logging the messages resides in the kernel. Copying a message takes place immediately after issuing the command to the network interface to send it over the network. Therefore, logging a message is not in the critical path of interprocess communication. For the common case of a remote procedure call where the sender is blocked [BN84], logging the message does not affect the communication latency.

If all messages sent from a recovery unit between two consecutive checkpoints fit in the volatile log, the overhead of message logging will be small. Otherwise, the volatile log must be flushed to secondary storage to free up space. When the size of the logged data exceeds 87.5% of the log space, the log is flushed to secondary storage. Unfortunately, the workstations used in the implementation were diskless. The message log must therefore be flushed to a secondary storage server over the network. Thus, if the log is repeatedly exhausted, every application message is transmitted twice over the network. This overhead would be eliminated if each workstation had a local disk.†

Performance Enhancements

The implementation attempts to reduce the amount of data to be logged. For example, messages that carry control information pertinent to the V-System's communication protocol are not logged. Examples include message retransmissions because of timeouts, messages used for flow control, and messages used to maintain a connection between two processes on two different machines. These messages do not affect the application program and are not required during execution replay after recovery. Eliminating these messages from the volatile log yields a reduction in the size of logged data that ranges from 5% to 10% for the applications studied in Chapter 4.

Another optimization technique attempts to avoid blocking the application program when the message log becomes full. In such cases, the data of the message

†When the volatile message log overflows, it need not be flushed to stable storage as the log will be recreated during recovery if a failure occurs. Secondary storage here acts as an extension of the volatile message log similar to swap space being an extension of a machine's main memory.

are write-protected within the address space of the application using the memory management hardware [FR86]. The application program is then allowed to continue. The kernel will block the application program only if it modifies the protected data. This technique is especially effective in applications where the communication load consists of infrequent large bursts of messages. When space becomes available, the data are copied and then unprotected.

Data Compression

Compressing the volatile message log improves the utilization of the allocated buffers. Its drawback is the competition with the application program for processor cycles. To examine the effect of compression on message logging, the volatile logs were flushed to secondary storage, and a data compression algorithm was applied post mortem on the log files. The experiment used the **compress** program that is widely available on UNIX systems [Wel84]. Experiments showed that data compression is not effective with the applications studied. The reduction in data was less than 24% for all applications, a saving that could not justify the large processing overhead of compression.

3.2.2 Antecedence Graph

The antecedence graph is represented by a list of *event descriptors*, each consisting of a record of eight bytes as follows:

1. The identifier of the recovery unit in which the event occurs (one byte). This identifier is assigned to the recovery unit when it is created. The current implementation supports a total of 128 recovery units.

2. The event type if it is an internal nondeterministic event, output commit, or a message receipt from the outside world. Otherwise, the event is a message receipt from another recovery unit, and the field contains the identifier of the sender of the application message (one byte). The most significant bit of this field is reset for a message receipt from another recovery unit, and is set otherwise.

3. The identifier of the thread associated with the event, such as the receiver of a message, the caller of a system call, or the caller of a synchronization call (two bytes).

4. The index of the state interval that starts with the event (two bytes).
5. An event-dependent field that contains information to be used during execution replay (two bytes). This field holds the message unique identifier for a message-receipt event. For a system call, it contains the return code from the system.

Some events cannot be encoded within eight bytes. For example, the **GetTime()** system call returns a 32-bit value showing the time in addition to a 16-bit return code, requiring 10 bytes. Similarly, a message from the outside world may contain more data than can be encoded in eight bytes. An *extended* event descriptor is used for these events. Such a descriptor consists of a field that distinguishes the extended descriptor from ordinary ones (one byte), a field that contains the length of the descriptor (two bytes), and the data.

The antecedence graph is asynchronously flushed to stable storage whenever its size exceeds 64 bytes. This threshold was selected after experimenting with sizes of 32, 48, 64, 96, and 128 bytes. Among these, the chosen threshold offers the best tradeoff between the average size of the antecedence graph piggybacked on application messages and the frequency of saving the antecedence graph on stable storage.

Data Structures

Every recovery unit, replicated or otherwise, maintains an array of the maximum state interval index known about each recovery unit. The array prevents the addition of redundant event descriptors to the local antecedence graph. For example, assume that a recovery unit receives a message whose appended antecedence graph contains the event descriptor corresponding to state interval 42 of the recovery unit whose identifier is 11. If the entry number 11 in the array shows that the maximum state interval known is larger than 42, then the event descriptor is discarded. Otherwise, the event descriptor is added, and the array entry is adjusted correspondingly. Thus, the array allows the receiver to determine by a simple indexing operation whether an event descriptor is redundant. No hashing or linear search over the descriptors in the antecedence graph is required.

Every recovery unit also maintains a set of pointers to support incremental piggybacking of the antecedence graph. For each recovery unit in the system there is a pointer that indicates the first event in the list that has to be appended on any outgoing message to that recovery unit. The pointer is advanced beyond the last event descriptor that was piggybacked when the remote recovery unit indicates that it has received the corresponding message.

Performance Enhancements

The implementation uses the semantics of the communication protocol to reduce the number of event descriptors that have to be added to the antecedence graph. For example, no event descriptor is created when the sender receives a reply message for a blocking remote procedure call. Such an event is deterministic, as the sender cannot accept a different message while blocked waiting for a reply.

Another example occurs when the application program specifies the sender from which it should receive the next message. Again, such a message receipt is deterministic and does not require adding an event descriptor. A third example is node collapsing in the context of bulk data transfer. When several messages are received while a connection is established between two processes, only one event descriptor denotes the receipt of all consecutive messages from the same sender.

Internal Nondeterministic Events

The implementation supports internal nondeterminism caused by memory sharing between threads on a single recovery unit if access to shared memory is protected by system-visible synchronization operations. It also supports nondeterministic interactions of the application program with the workstation's kernel and system servers. The current implementation does not support other forms of nondeterminism, such as asynchronous software signals, operations whose outcome depends on the real-time behavior of the system, and unprotected access to shared memory.

Synchronized Memory Sharing

The implementation provides semaphores as the basic synchronization primitives. The kernel records the order in which P and V calls are made in the antecedence graph. Each call starts a new state interval, for which a corresponding *synchronization event descriptor* is added to the graph. The descriptor's type field indicates a synchronization operation. Remaining fields contain the recovery unit's identifier, the thread identifier, and the state interval index. Similar techniques have been used in Instant Replay for debugging parallel programs [LMC87] and in an implementation of optimistic recovery [GGL$^+$90].

System Calls

The V-System implements system calls by local remote procedure calls to the kernel and system servers. Unlike monolithic kernels, the number of system calls available to application programs is limited. They mostly cover process management (thread creation and destruction, scheduling, and synchronization), and memory management (memory allocation and deallocation).

Interactions with the kernel and system servers require special handling. Simply replaying the return values from a kernel call during recovery is not sufficient. The kernel data structures must be updated according to the outcome of the call. For example, a **CreateThread()** kernel call during recovery requires the allocation of the necessary kernel data structures. The reconstructed thread must also have the same identifier as before the failure. System calls are divided into three categories:

1. Idempotent calls that can simply be executed during recovery. These report some information about the environment in which the application program is running. An example is the **GetThreadId()** call, which returns the identifier of the calling thread. It is safe to execute idempotent calls during recovery because they always return the same value.

2. Non-idempotent calls for which the same value as during failure-free execution must be returned *without* executing the call, such as **GetTime()** which reads the system clock.

3. Non-idempotent calls that must be executed and forced to return the same result as during failure-free execution. Examples are calls that manipulate the kernel's state, such as memory allocation and deallocation calls. If the kernel cannot reproduce the same result, for example because memory cannot be allocated, recovery should be restarted on a different machine.

Idempotent system calls are not recorded in the antecedence graph. For the other two categories, the kernel records the call and its outcome as an event descriptor in the antecedence graph. The descriptor's type field indicates that the state interval is started by a system call. The descriptor contains the recovery unit's identifier, the state interval index, the identifier of the calling thread, and the values returned by the system.

Nondeterministic Events and Replication

The current implementation does not support the tracking of nondeterministic events for replicated recovery units. Such support can be implemented as in the Delta-4 project [BHV+90].

3.3 Implementation of Checkpointing

Each machine based on rollback-recovery includes a *checkpoint server* that participates in the checkpointing protocol as described in Section 2.5. This server is responsible for controlling the actions required to take a checkpoint of the recovery unit. The coordinator of the global checkpoints controls a timer that expires at the end of each checkpointing interval. Initially, the checkpointing interval is specified during the creation of the recovery unit on which the coordinator runs. It can be changed later by sending a message to the coordinator.

3.3.1 Checkpointing a Recovery Unit

When the local checkpoint server receives a request for participation in a global checkpoint, it starts a new checkpoint. The server begins by sending a message to the stable storage server to open a new version of the checkpoint file. It then sends a message to the kernel to take a snapshot of the recovery unit's state and to save it in the checkpoint file. The saved state includes a copy of the recovery unit's address space and a copy of the state of all its threads. The state also includes a copy of all data structures that are maintained by the kernel and servers for the recovery unit. These data structures are used during recovery to recreate an execution environment that is identical to the one before the failure. The saved checkpoint, however, does not contain the entire state of the kernel and servers, as there is no attempt to make them fault-tolerant.

3.3.2 The Need for Optimizations

The overhead of checkpointing consists of the costs of saving the state on stable storage and the interference between the checkpointing and the execution of the application program. The first component results from copying the state over the network to the stable storage device. The second component results from the requirement that the saved state in the checkpoint file must represent an instantaneous snapshot of the recovery unit's state at some point in time. A simple solution is to block the execution of the application program while the checkpoint is being taken [KMBT92, TS84]. This policy, however, can have a severe effect on the performance and thus saving a checkpoint in Manetho is both *incremental* and *concurrent* [Joh89, LNP90].

Incremental Checkpointing

Instead of writing the entire address space on stable storage during each checkpoint, only the memory pages that have been modified since the previous checkpoint are written. This set of pages is determined using the *dirty* bit maintained by the memory management hardware in each page table entry.

Concurrent Checkpointing

The kernel uses copy-on-write protection to allow the application program to continue execution while its checkpoint is being written on stable storage. At the start of an incremental checkpoint, the pages to be written on stable storage are write-protected using the memory management hardware [FR86]. The application program blocks only while initializing the protection of the address space. After writing a page on stable storage, the kernel removes the protection on it. If the application program attempts to modify one of these pages while it is still protected, a memory protection fault is generated. The kernel copies the page into a newly allocated page of memory, removes the protection on the original page, and allows the application program to continue. The newly allocated page is not accessible to the application program. It is deallocated after writing the original contents of the page on stable storage. If no memory is available to allocate a new page for handling the copy-on-write fault, the application program is blocked until memory can be allocated. This scheme is similar to that used by Li et al. in their implementation of checkpointing on shared memory multiprocessors [LNP90].

The interference of concurrent checkpointing with the progress of the application program is small. Blocking the application occurs only while initializing the protection on the address space, which takes only a few tens of microseconds. Servicing a copy-on-write fault takes only about 250 microseconds, during which the application program is blocked. Furthermore, often a page is written on stable storage before the application program attempts to modify it because of the locality in memory referencing. In comparison, the application program may be stopped for several seconds if it is blocked for the entire duration of taking the checkpoint.

3.3.3 Checkpointing and Replication

A checkpoint server runs on the leader of each replicated recovery unit. When it receives a request for a checkpoint from the coordinator, it forces a synchronization step as described in Section 2.6.2. After receiving an acknowledgment from all cohorts, the checkpoint server acknowledges the completion of the "checkpoint" to the coordinator. After receiving a success message, the checkpoint server at the leader sends a message to the cohorts confirming the success. This notification allows the cohorts to discard from their message logs the messages sent before the coordinated checkpoint.

3.4 Manetho's Multicast Protocol

A separate *multicast layer* is added to the V-System's communication protocol above the data link layer. It supports receiving multicast messages destined to a troupe. It operates on every network packet belonging to a multicast as a separate message. The maintenance of the antecedence graph is still performed at the transport layer for efficiency.

3.4.1 Multicast Layer at the Leader

When the troupe receives a message, the multicast layer at the leader defines the receipt order. It then passes the message to the upper communication layers for the necessary processing. However, it does not immediately send the receipt order to the cohorts in a sequence multicast as suggested in Section 2.6.1. Instead, it attempts to batch the receipt orders of 10 packets in one sequence multicast. This batching works well for bulk data transfer, since it reduces the overhead of control messages and its effect on application performance. However, if less than 10 packets were received by the time of the next timer interrupt, the pending sequence multicast is sent to ensure that the cohorts do not fall behind the leader in their executions. The timer interrupt occurs each 10 milliseconds. Thus, the sequence multicast of a message is delayed by five milliseconds on average.

3.4.2 Multicast Layer at the Cohorts

The multicast layer at a cohort suppresses the transmission of application messages. It also buffers input messages until the leader sends the corresponding sequence multicast.

The multicast layer performs a synchronization step with the leader every second, or after receiving two thousands input multicasts, whichever occurs first. This large synchronization period has been selected after experimenting with various values. It was found that communication failures were extremely rare. Specifically, the rate by which a cohort misses a packet that was received by the other replicas was less than one in 10000. This low rate is mainly a result of the excellent flow control of the V-System protocol. The implementation exploits this fact to reduce the overhead of the synchronization step.

3.5 Manetho's Recovery Protocol

Support for recovery on each machine is implemented by a user-level *recovery server* and a number of kernel routines. During failure-free operation, the recovery server responds to the requests made by other recovering units for message replay and antecedence graph collection, as described in Section 2.8. During recovery, the server controls the execution of the recovery unit until recovery is complete. As described in Section 2.8, recovery proceeds in three phases:

- Restoration of the state from an earlier checkpoint or election of a new leader.
- Reconstruction of the antecedence graph.
- Replay of the lost execution.

3.5.1 State Restoration

For recovery units that use rollback-recovery, the recovery server starts recovery by reading the recovery unit's checkpoint from stable storage and restoring its address space. The pre-failure state that was maintained by the kernel and system servers for the recovering unit is also reinstalled.

3.5.2 Leader Election

For recovery units that use replication, the recovery servers on the various replicas run the leader election protocol, as described in Section 2.8.3. The machine identifiers are used to break ties during election.

3.5.3 Running the Recovery Protocol

The implementation of the recovery protocol differs slightly from the description in Section 2.8. In the proposed algorithm, a surviving recovery unit uses a list (*REJECT_LIST*) to inhibit the reception of certain application messages while other recovery units are recovering. This inhibition extends from the time the recovery unit responds to the **CollectGraph()** call until it receives the corresponding **UpdateInc()** call. The implementation uses a simpler but less efficient approach.

When a recovery unit receives the **CollectGraph()**, it simply inhibits the receipt of *all* application messages until it receives the **UpdateInc()** call, obviating the need for the rejection list. In this scheme, the recovery protocol interferes with the normal processing of surviving recovery units. However, since failures are rare, the performance penalty is small and the desire to simplify the implementation prevails.

3.5.4 Execution Replay

The recovery server cooperates with the kernel to recreate the nondeterministic events that occurred before failure. As explained in Section 3.2.2, such events are implemented by sending messages to the kernel. During recovery, the kernel places the thread under the control of the recovery server. The server determines whether the event is next in the order specified by the information in the antecedence graph. If the event is out of order, the calling thread is suspended until the expected event occurs. Otherwise, the recovery server determines and performs the required actions to recreate the events recorded in the antecedence graph.

3.6 Stable Storage

3.6.1 Design

Since failures are infrequent, the information stored on stable storage is seldom read. Consequently, the workload on the stable storage server consists mainly of write operations that must be forced to disk. The design of the stable storage server is therefore optimized for the common case of writing files of large sizes. In particular, the stable storage server organizes the disk as a sequential log to eliminate most disk seeks [BHG87]. Read and write operations occur in multiples of disk blocks, up to eight blocks in one operation. The disk block size is eight Kbytes, which is also the size of a memory page on a Sun 3/60. The stable storage server does not maintain a conventional disk cache [OCD+88], since it would be useless given the expected workload.

To provide efficient support for incremental checkpointing, the stable storage server supports file versioning. Versions of a single file share disk blocks common between them. When a recovery unit takes a checkpoint, the server creates a new version of the corresponding file containing all disk blocks that belong to the older one. The reference count for each of these disk blocks is incremented by one. The modified pages that belong to the new checkpoint replace the older disk blocks in the new version as they are written by the recovery unit. Thus, the new version contains a contiguous image of the recovery unit's address space at the time of the checkpoint.

The maximum number of versions a file may have is limited to 255. Thus, the stable storage server can support systems where several checkpoints may be maintained for each recovery unit, such as in optimistic recovery [Joh89, SY85]. When a checkpoint is to be discarded, the corresponding version is purged.

The maximum throughput for writing to the server is 860 Kbytes/second. This maximum occurs when both server and client reside on the same machine and is included only as a reference. For the usual mode of operation, the recovery units access the server over the network. The maximum writing throughput in this case is about 670 Kbytes/second, and is limited by the performance of the V-System's communication protocol for bulk data transfer and by the bandwidth of the network. The maximum throughput for reading from the stable storage server is about 300 Kbytes/second, and is limited by the cost of reading data from disk.

3.6.2 Alternative Designs

An alternative to using a custom stable storage server is to rely on an ordinary network file server. This alternative has the advantages of simplicity and generality. However, it has a performance disadvantage. Ordinary file servers are optimized for situations where most files are small and short lived, and where the workload is dominated by reads [OCH+85]. Such optimizations would be of little value for a stable storage server in a rollback-recovery system, where the workload is write-dominated and the logs are large, long-lived, and seldom read.

A second disadvantage of relying on an ordinary file server is the lack of direct support for stable storage requirements. Examples include the requirement for checkpoint files that can be atomically updated with incremental changes, and the capability to free selected blocks from a file during garbage collection. Such requirements can be implemented by using auxiliary indexing structures, at the expense of complicating the logging logic at the clients.

3.7 Summary

This chapter described the implementation of Manetho. The implementation uses several techniques to reduce the overhead of providing fault tolerance:

1. The use of copy-on-write with message logging allows the application to proceed even when the space in the message log is exhausted. Additionally, incremental piggybacking and exploiting of the communication patterns of the application programs lead to an efficient implementation for the antecedence graph.

2. To reduce the overhead of checkpointing, concurrent and incremental checkpointing reduce the probability that the application blocks while a checkpoint is being taken. A custom stable storage device simplifies the implementation of incremental checkpointing and reduces the overhead of storing the recovery information.

3. An aggressive implementation of Manetho's multicast uses batching of sequence multicasts, and exploits the efficient control flow algorithm of the V-System to reduce the overhead of synchronization messages. These two techniques allow the multicast protocol to achieve the best tradeoff between throughput and latency.

Chapter 4

Performance Evaluation

This chapter presents an empirical performance evaluation of Manetho. It includes a discussion of the failure-free performance of rollback-recovery and multicast protocols, and the performance during recovery.

4.1 Overview

This chapter presents an empirical performance evaluation of Manetho, and compares it to protocols. The performance evaluation comes in three parts. The first part addresses the failure-free performance of Manetho's rollback-recovery. The measurements show that Manetho adds a negligible overhead over that caused by other rollback-recovery protocols, in return for faster output commit and limited rollback. Measurements also show that the implementation techniques used to reduce the performance overhead are successful.

The measurements reveal some properties of previous rollback-recovery protocols that have not been addressed in the literature. For example, the performance of receiver-based message logging is inferior to that of sender-based logging. Thus, message logging should be done at the sender whenever possible. Other issues include a comparison between coordinated checkpointing and message logging in general, and another comparison between independent and coordinated checkpointing.

The second part of the performance study compares Manetho's multicast with two well known, efficient multicast protocols, the Amoeba r-resilient multicast protocol [Kaa92], and the protocol by Chang and Maxemchuck [CM84]. They respectively represent the positive and negative acknowledgment families of multicast protocols. Results show that for supporting process replication, Manetho outperforms the other two protocols in throughput and latency. The study confirms the conventional wisdom that negative acknowledgment protocols have higher throughput and latency than positive acknowledgment protocols. Furthermore, the results show that a negative acknowledgment protocol should not be designed separately from the flow control algorithm, and that applications that stand to benefit from these protocols should not rely on synchronous communication modes.

The last part of the performance study addresses the performance of the recovery protocol. Running the recovery protocol adds only a negligible amount to the recovery time. Recovery is dominated by the time to restore the information from stable storage and replay the execution. These costs are inherent in any rollback-recovery protocol.

4.2 Failure-Free Performance of Rollback-Recovery

4.2.1 Test Suite

The performance study uses seven compute-intensive distributed applications. These applications provide a reasonable spectrum of communication loads and patterns, and memory requirements. They are also typical of the applications that would run on a network multicomputer. Each application runs in parallel on 16 recovery units. A brief description of each application follows:

`gauss`: This program performs Gaussian elimination with partial pivoting on a matrix of size 1024×1024 elements. The problem is distributed evenly among the 16 recovery units by giving each a subset of the matrix columns. At each iteration of the algorithm, the recovery unit which holds the pivot element sends the pivot column to all other recovery units.

`grid`: This program carries out an iterative computation on a grid of 2048×2048 points. In each iteration, the value of each point is computed as a function of its value in the last iteration and the values of its neighbors. This application occurs in the kernel of many fluid-flow modeling algorithms. The problem is distributed among the 16 recovery units by giving each a section of the matrix to compute. After each iteration, each recovery unit exchanges the new values on the edges of its section with the corresponding neighboring recovery units.

`matmult`: This program carries out the multiplication of a given pair of matrices of size 1024×1024. Computing the result matrix is evenly distributed among the recovery units when the execution begins. No communication is required except for reporting the solution.

`nqueens`: This program counts the number of solutions to the n-queens problem for 15 queens. The search space is evenly distributed among the participating recovery units at the beginning of the program, and no communication is required except for reporting the solution.

`prime`: This program attempts to factor a 64-digit integer, which can be useful as a probabilistic test for primality. A distributed implementation of the Pollard-Rho method is used. The computation is organized as a task queue. A master process distributes the work for each slave process and announces the factors that have been discovered for the number.

`sparse`: This program solves a linear system of equations in 48000 unknowns represented by a sparse matrix. Less than 0.25% of the elements in each matrix row are greater than zero. `Sparse` uses a variation on the iterative Gauss-Seidel method. The problem is divided such that each recovery unit solves a subset of the unknown variables. After each iteration, each recovery unit sends the new values of the unknown variables in its domain to all other recovery units.

`tsp`: This program uses a distributed branch and bound algorithm to solve the traveling salesman problem for a dense map of 18 cities. The program has a master-slave structure. A master maintains the current best solution, and a task queue containing subsets of the search space. The master assigns tasks from the queue to a number of slaves. Each slave solves its assigned task and reports the solution back to the master. The latter updates the current best solution and returns it back to the slave along with a new task.

Table 4.1 shows the running times, memory requirements, checkpoint sizes, and communication rates for the seven applications.

4.2.2 Failure-Free Application Performance with Manetho

Table 4.2 presents the failure-free overhead for each application program when running with Manetho. The overhead is expressed by the percentage increase in running time due to the provision of fault tolerance. Measurements were taken with two, five and ten minutes checkpointing intervals. These checkpointing intervals are typical for these types of long-running applications.

Program Name	Running Time (minute)	Per RU Memory (Kbyte)			Total Size (Mbyte)	Communication Rate (per second)	
		Code	Data	Check.		Messages	Kbyte
`gauss`	48	20	576	596	9.3	49.5	31.7
`grid`	59	21	2163	2184	34.1	16.0	8.3
`matmult`	137	20	2348	2368	37.0	0.2	0.2
`nqueens`	77	18	22	40	0.6	0.1	0.1
`prime`	53	38	74	112	1.8	0.8	0.3
`sparse`	57	22	1954	1976	31.6	30.5	70.5
`tsp`	73	21	27	48	0.8	0.2	0.1

Table 4.1 Running times, memory requirements, checkpoint sizes and communication rates for seven compute-bound applications.

Program	% Increase in running time		
Name	2-minute	5-minute	10-minute
`gauss`	1.1	1.1	1.0
`grid`	2.6	1.3	1.0
`matmult`	0.4	0.2	0.2
`nqueens`	0.0	0.0	0.0
`prime`	0.7	0.7	0.6
`sparse`	3.8	4.0	3.8
`tsp`	0.0	0.0	0.0

Table 4.2 The performance overhead of Manetho expressed as the percentage increase in running time for three checkpointing intervals.

Analysis

In these measurements, Manetho's failure-free overhead increases the running time by 1% on average and by 4% for `sparse`, the application with the highest memory and communication requirements. This overhead is a small price to pay for protecting the application from the effects of failures.

As expected, the overhead generally increases with smaller checkpointing intervals. This increase is due to the more frequent checkpointing activities competing with the application programs for processor cycles and network bandwidth. An exception is `sparse`, which has a high communication load. For a checkpointing interval as small as two minutes, the volatile message logs do not fill up, and the resulting overhead is mainly due to checkpointing and antecedence graph maintenance. For a checkpointing interval of 10 minutes, the overhead of checkpointing declines. However, the volatile message logs overflow and must be flushed over the network to secondary storage as described in Chapter 3. This overhead competes with the application program for network bandwidth and negates the effects of the reduction in checkpointing overhead. Nevertheless, even with a checkpointing interval as small as two minutes, the overhead is below 4%.

Overhead Components

Table 4.3 presents the individual contributions of checkpointing, message logging, and antecedence graph maintenance to the failure-free overhead for four of the applications with the highest communication loads. For the remaining applications, the overhead is so small (close to 0%) that an analysis does not produce any insight. The numbers do not always add up to the total overhead due to rounding errors.

The overhead due to checkpointing increases with the size of the application's address space, as expected. The cost of maintaining the volatile message logs in `sparse` is due to message log overflow. For the other applications, the volatile logs do not overflow between checkpoints.

Maintaining the antecedence graph accounts for less than 1% of the total overhead for the four applications. This small overhead is the extra price to be paid for fast output commit, limited rollback, and the graceful interaction between rollback-recovery and process replication. As expected, the cost depends on the frequency and pattern of interprocess communication. It is highest for `gauss`, `grid` and `sparse`, the applications with the highest communication requirements.

Program Name	% Increase in running time			
	Interval	Checkpointing Overhead	Message Logging	Antecedence Graph
`gauss`	2 min.	0.4	0.0	0.7
	5 min.	0.3	0.0	0.7
	10 min.	0.2	0.0	0.7
`grid`	2 min.	2.0	0.0	0.6
	5 min.	0.7	0.0	0.6
	10 min.	0.4	0.0	0.6
`prime`	2 min.	0.3	0.0	0.4
	5 min.	0.2	0.0	0.4
	10 min.	0.2	0.0	0.4
`sparse`	2 min.	2.2	0.7	0.9
	5 min.	0.5	2.6	0.9
	10 min.	0.3	2.6	0.9

Table 4.3 Contribution of each component to the failure-free overhead.

4.2.3 Component Costs

Cost of Antecedence Graph Maintenance

Maintaining the antecedence graph affects the performance of kernel calls, synchronization primitives, and interprocess communication. For kernel calls, Manetho's overhead is 16 microseconds, or about 3% for simple kernel calls, and less for more complex ones. For synchronization primitives, Manetho provides semaphores (see Section 3.2.2). Manetho adds 8 microseconds to the cost of a semaphore operation which was originally 15 microseconds.

The effect on interprocess communication depends on many factors, such as the size of the message and the size of the antecedence graph. When two threads on the same recovery unit communicate by a message, the addition of an event descriptor to the antecedence graph adds 8 microseconds to the cost of a message. The percentage overhead varies from 2% to 0.08% depending on the message size. The cost of remote communication depends on the number of event descriptors appended to the message. Appending n event descriptors on an application message costs $80 + 6n$ microseconds at the sender and on the communication channel. At the receiver, merging the incoming event descriptors adds $52 + 7a + 3d$ microseconds to the cost of receiving a message, where a and d are the numbers of added and discarded event descriptors, respectively.

Effect of Copy-on-Write on Logging

Manetho uses write-protection of the message data in the sender's address space when the volatile message log is full (see Section 3.2.1). In `sparse`, the volatile message logs fill up repeatedly, because communication in `sparse` occurs in large bursts. An alternate message logging implementation in which the sender blocks if the logs overflow results in an increase in running time of about 26% with a checkpointing interval of 10 minutes, compared to 3.8% for Manetho's message logging implementation. For the other applications, the message logs do not overflow and therefore copy-on-write was not used.

4.2.4 Comparison with Another Implementation of Manetho

The performance of the seven application programs was measured for an implementation of an earlier design of Manetho that uses independent checkpointing [EZ92a, EZ92b]. In this design, each recovery unit flushes its message log to stable storage whenever it takes a checkpoint. If the message log is not flushed, a domino effect may occur if two or more recovery units fail. Table 4.4 shows the percentage increase in running time for the seven compute-bound application programs with checkpointing intervals of two, five, and ten minutes. The measurements underestimate the overhead of the implementation using independent checkpointing because they do not reflect any garbage collection activity.

There is a slight performance disadvantage for the implementation using independent checkpointing, due to the cost of flushing the volatile message logs to stable storage. The difference would have been more pronounced for an application like `sparse` if a local disk provided a secondary storage device. In this case, the implementation using coordinated checkpointing would not pay the cost of flushing the volatile message logs over the network to secondary storage. In summary, the measurements show that the design using coordinated checkpointing is superior to the earlier design based on independent checkpointing.

Program Name	% Increase in running time					
	2-min.		5-min.		10-min.	
	Coord.	Indep.	Coord.	Indep.	Coord.	Indep.
gauss	1.1	1.3	1.1	1.2	1.0	1.1
grid	2.6	2.8	1.3	1.6	1.0	1.3
matmult	0.2	0.2	0.1	0.1	0.0	0.0
nqueens	0.0	0.0	0.0	0.0	0.0	0.0
prime	0.7	1.1	0.7	1.0	0.6	1.0
sparse	3.8	5.6	4.0	4.4	3.8	3.8
tsp	0.0	0.0	0.0	0.0	0.0	0.0

Table 4.4 A comparison of the performance overhead of an implementation of Manetho that uses coordinated checkpointing with a similar implementation using independent checkpointing.

4.2.5 Comparison with Other Protocols

Four other rollback-recovery protocols were implemented on the same test bed: independent checkpointing, coordinated checkpointing, optimistic sender-based message logging, and optimistic receiver-based message logging. A brief description of each of these protocols follows.

Independent Checkpointing

With independent checkpointing, each process takes a checkpoint of its state without coordination with other processes [BL88, WF92a, WF92b]. If a failure occurs, the recovery algorithm will roll back the computation to a set of checkpoints on stable storage that forms the most recent consistent state. This technique does not guarantee that a consistent set of checkpoints exists on stable storage, and therefore the computation may roll back to the initial state (the domino effect) [Ran75]. Furthermore, as checkpoints accumulate on stable storage, the system needs to invoke a garbage collection procedure.

Coordinated Checkpointing

Manetho's checkpointing protocol was chosen as a representative of coordinated checkpointing protocols.

Receiver-Based Optimistic Message Logging (RBML)

In RBML, processes use message logging in addition to independent checkpointing [BBG+89, JV91, JZ88, PP83, SW89, SY85]. The processes that participate in the distributed computation log the messages that they receive during failure-free operation on stable storage. Asynchronous logging is used to avoid the overhead of synchronous stable storage access.

During recovery from a failure, a process is restarted from a previous checkpoint and the messages in the log are replayed, restoring the process to a state that occurred before the failure. Several techniques exist for recovery, all based on computing the maximum recoverable state using the checkpoints and message logs available on stable storage [Joh89]. The performance study uses an implementation of the technique suggested by Johnson and Zwaenepoel [JZ88]. In this technique, the sender adds $O(1)$ dependence information on each message it sends. However, the implementation departs from the original design with respect to output commit. The original design runs a multihost protocol that logs the necessary messages on stable storage. However, logging the data is not necessary for output commit in these protocols, nor is it efficient. Instead, the implementation just logs the receipt order of the messages.

Sender-Based Message Logging (SBML)

The SBML protocol implemented here is modeled after the protocol of Strom et al, in which the processes log the messages at the sender and log the receipt order at the receiver [JZ87, SBY88]. The resulting protocol tolerates an arbitrary number of failures. In the implementation, each sender adds an $O(1)$ dependence information on each message it sends, as suggested by Johnson [Joh89]. Again, the implementation of output commit logs the receipt order instead of the entire messages, for the same reasons as in the RBML implementation.

Measurements

Table 4.5 shows the percentage increase in running time for four applications under the five rollback-recovery protocols. The overhead for the remaining three applications is negligible, and therefore they are omitted. The measurements do not reflect any garbage collection activities for independent checkpointing, SBML, or RBML, and thus the reported measurements slightly underestimate their failure-free performance overhead.

Analysis

1. There is no considerable difference in performance between independent and coordinated checkpointing. This result contradicts many previous claims that portrayed coordinated checkpointing as inferior because of the overhead of coordination [BL88]. In fact, the overhead of coordinated checkpointing reported here is not due to coordination messages, but rather a result of having all the machines "gang up" on the same stable storage server at the same time to save their states [EJZ92]. With a larger stable storage bandwidth and after accounting for the overhead of garbage collection of independent checkpointing, the performance difference between the two methods is likely to decrease further.

2. The three protocols based on message logging (RBML, SBML, and Manetho) have higher overhead than the protocols based only on checkpointing. The overhead of message logging and dependence tracking outweighs the overhead of coordinated checkpointing, contrary to previous claims [SW89, SY85].

3. Independent checkpointing performed worse on `gauss` than consistent checkpointing. This anomaly is due to the tight synchronization nature of the `gauss` program. Each iteration of `gauss` requires global communication among the recovery units to distribute the next pivot element and pivot column. Inevitably, the recovery unit computing the pivot column for the next iteration slows down while taking a checkpoint, delaying the generation of the next round of communication. As a result, this slowdown in one recovery unit forces a slowdown for every recovery unit participating in the computation. With independent checkpointing, the checkpoints of the recovery units are taken at different times, causing a slowdown for the entire application every time a recovery unit takes a checkpoint. With consistent checkpointing, instead, all recovery units take a

Program Name	% Increase in running time					
	Interval	Indep. Check.	Coord. Check.	SBML	RBML	Manetho
`gauss`	2 min.	0.8	0.3	1.0	2.0	1.1
	5 min.	0.3	0.2	0.5	1.3	1.1
	10 min.	0.1	0.1	0.2	1.1	1.0
`grid`	2 min.	1.6	1.8	1.9	2.3	2.6
	5 min.	0.3	0.6	0.6	0.9	1.3
	10 min.	0.2	0.3	0.4	0.8	1.0
`prime`	2 min.	0.2	0.4	0.7	1.4	0.7
	5 min.	0.3	0.3	0.7	1.3	0.7
	10 min.	0.2	0.2	0.7	1.3	0.6
`sparse`	2 min.	1.6	2.0	4.1	9.8	3.8
	5 min.	0.3	0.6	3.2	8.8	4.0
	10 min.	0.2	0.3	2.8	8.5	3.8

Table 4.5 The performance overhead of five rollback-recovery protocols.

checkpoint at essentially the same time, causing only a single slowdown of the application.

4. RBML is inferior in performance to SBML. This difference is relevant for some optimistic message logging protocols, where logging can be performed either at the receiver or at the sender.[‡] While the two approaches are semantically equivalent, logging at the sender leads to better performance for two reasons. First, copying of the message in SBML takes place after transmitting the message on the network. Therefore, the logging of the message is not in the critical path of interprocess communication. In RBML, copying the message must be performed before delivering the message to the application, which slows down interprocess communication. Second, when a recovery unit sends the same message to several recovery units, only one copy of the message needs to be made in the log in SBML. Therefore, for applications that frequently use broadcasting,

[‡]Some message logging protocols require messages to be logged at the sender, such as the Manetho protocol and the original sender-based message logging protocol.

SBML leads to more efficient usage of the space in the volatile message log, reducing the likelihood of log overflow. In RBML, a broadcast message will be logged at every receiver, increasing the likelihood of log overflow for applications that frequently use broadcast communications. The applications `gauss` and `sparse` are typical of such applications.

4.2.6 Output Commit

The real advantage of rollback-recovery protocols based on message logging is the lower latency obtained during interactions with the outside world. Table 4.6 shows the average latency of output commit for four of the protocols under study,[§] for a system of 16 recovery units. For SBML and RBML, the measurements are shown when all 16 recovery units are involved in committing output. For the coordinated checkpointing protocol, a global checkpoint must be taken before output is committed. In this case, the latency is expressed as a function of the size of the global checkpoint.

The table shows that the three message logging protocols have much better latencies than the coordinated checkpointing protocol. The delay for coordinated checkpointing is a function of the size of the checkpoint, making coordinated checkpointing impractical for applications that have considerable interactions with the outside world. Out of the three message logging protocols, the Manetho system performs best because its dependency tracking allows each process to commit output locally with a single I/O operation on stable storage. The other two message logging protocols require a multihost protocol to commit output, which entails exchanging several messages and performing several I/O operations on stable storage [Joh89, Joh93, SW89].

Output Commit Latency (Seconds)			
Coordinated Checkpointing	SBML	RBML	Manetho
1.210 + 1.3/MB	0.630	0.630	0.050

Table 4.6 The output latency for coordinated checkpointing and three protocols based on message logging.

[§] No measurements are shown for independent checkpointing since there is no known algorithm for committing output with this method.

4.2.7 Summary

The results of the performance study are summarized as follows:

- The overhead added by Manetho to application programs is small. For the applications studied, the increase in running time is within 4%. This overhead is a small price to pay for fast output commit, limited rollback, and the interaction between rollback-recovery and process replication.

- Maintaining the antecedence graph adds negligible performance overhead during failure-free operation. For the applications tested, the overhead added by maintaining the antecedence graph is within 1%.

- Manetho's output commit is better than other rollback-recovery protocols, often by an order of magnitude. This advantage is realized at the expense of a negligible increase in running time when compared to other rollback-recovery methods based on message logging.

In addition to the above results, the study provides more general results concerning rollback-recovery protocols:

- Coordinated checkpointing is an efficient technique for implementing rollback-recovery, contrary to previous beliefs. Furthermore, the performance difference between coordinated checkpointing and independent checkpointing is negligible.

- Coordinated checkpointing can be used profitably with rollback-recovery based on message logging, resulting in simpler garbage collection and lower overhead because of the reduced stable storage access.

- The results suggest that the purpose of message logging is not to eliminate the coordination of checkpoints as was suggested in previous designs. Rather, message logging is useful for applications that require interactions with the outside world, where coordinated checkpointing by itself becomes impractical.

- Using copy-on-write in implementing message logging is an important technique for reducing the overhead of message logging.

- When a choice is possible, logging the messages at the sender is preferred to logging at the receiver. Logging a message at the sender is not in the critical path of interprocess communication, and it also leads to more efficient utilization of the volatile log for applications that frequently use broadcast communications.

4.3 Failure-free Multicast Performance

This section presents an evaluation of Manetho's multicast. Results from the evaluation show that Manetho's multicast adds a small overhead compared to the performance of the unreliable unicast of the V-System, and the effect on the application programs is small. The evaluation also includes a comparison between Manetho's multicast protocol and two well known, general purpose multicast protocols. These two protocols are representatives of the positive and negative acknowledgment protocol families. They are general protocols that support active replication in addition to general purpose group communication. Both protocols were implemented in the same experimental environment as Manetho. The results show that a protocol designed specifically to support active replication outperforms general purpose multicast protocols when used for this purpose.

4.3.1 The Amoeba Broadcast Protocol

The r-resilient Amoeba broadcast protocol was selected as a representative of the positive acknowledgment family of multicast protocols because it is one of the fastest implementations of that kind known to date [Kaa92].[¶] It uses a two-phase agreement algorithm to determine the receipt order. The receivers of the multicast form a group, with a distinguished member called the *sequencer*. When the sequencer receives a message, it defines the receipt order and sends it with the message to the receiving group. The sequencer then waits for an acknowledgment from every member in the receiving group. When all acknowledgments are received, the sequencer marks the message as deliverable and allows the receivers to deliver it. During the experiment the sequencer was placed on the same site as the leader of a troupe, so that a message is delivered to the application as soon as the sequencer receives all acknowledgments.

4.3.2 Chang and Maxemchuck

The protocol of Chang and Maxemchuck was selected to represent the negative acknowledgment family because of its simplicity and efficiency [CM84]. The receivers of a multicast in this protocol form a logical ring and exchange a token along the ring's links. When the token holder receives a message, it broadcasts the receipt order and

[¶]Readers should not confuse the r-resilient version of the Amoeba broadcast which uses positive acknowledgment with the 0-resilient version which uses negative acknowledgment.

transfers the token to the next member in the ring. No explicit acknowledgment is sent when the token is received. After the token has completed one full round around the ring, the message is delivered. The transfer of the token among all receivers implies an implicit acknowledgment that the message and its receipt order have been delivered everywhere. However, token transfers occur only when an application message is received, or when a time out expires without the receipt of a new application message. As a result, the latency in message delivery increases and is sensitive to the communication rate of the application.

4.3.3 Experiments

Multicast Throughput

The throughput of a multicast protocol is the maximum rate at which it delivers data to the receivers. This rate is measured by having several client processes stream data at the maximum transmission rate to an idle troupe which does not perform any processing on the incoming data. In this manner, the application process does not consume any CPU cycles and therefore does not interfere with the experiment. Table 4.7 shows the measured throughput for the three multicast protocols under consideration, with a degree of replication up to four. The measurements shown were obtained with two client processes sending data streams. With more than two client processes, the throughput declines because of Ethernet saturation and collisions. As a reference point, the throughput of the V-System's unicast is 1250 Kbytes/second. This measurement is obtained by running the same experiment with a single receiver process using the base kernel, instead of a troupe using the kernel augmented with Manetho's multicast support.

Several conclusions are drawn from the table:

- Manetho's throughput is only 6% lower than the throughput of the V-System's unicast. This small loss of bandwidth is due mainly to the overhead of generating the sequence multicasts at the leader and transmitting them over the network to the cohorts. The cost of antecedence graph maintenance in this experiment is negligible, because of the incremental piggybacking and merging of consecutive packets from the same sender into one event descriptor, adding on average much less than one event descriptor per message (see Section 3.2). Similarly, the overhead of the messages used during the synchronization steps

Protocol	Number of Replicas		
	2	3	4
Manetho	970	970	970
Amoeba r-resilient	510	480	455
Chang and Maxemchuck	755	750	745

Table 4.7 Throughput of the three multicast protocols under study in Kbytes/second.

among the replicas is negligible because of the relative infrequency of this operation.

- Manetho's throughput is not sensitive to the number of replicas.
- The results confirm the conventional wisdom that negative acknowledgment protocols have higher throughput than positive acknowledgment protocols. The explicit acknowledgments for every message consume network bandwidth and CPU cycles at the sequencer machine, exacting a toll on the available bandwidth at the application level. To the best of my knowledge, this experiment is the first comparison of the two methods under otherwise identical circumstances.
- The throughput of the negative acknowledgment protocol was lower than expected. A subtle interaction between the multicast protocol and the algorithm for flow control of the V-System was the reason. After transmitting a batch of 32 packets, the sender's kernel waits for an acknowledgment from the receiver before sending the next 32 packets. In the experiment, there were two client processes streaming packets to the troupe. Therefore, an acknowledgment pertaining to the flow control algorithm had to be delayed for a token transfer. The token transfer occurs as soon as a packet is received from the other client process. Therefore, the delay of generating the acknowledgment is exactly the time to receive one more packet and transmitting the token. This slight delay causes the sender to spend more time waiting for the response of the receiver than anticipated in the original design of the flow control algorithm. The problem is inherent in any flow control algorithm that depends on some form of handshaking in its operation. It is not clear how the problem can be

solved, but it suggests that a multicast protocol should not be implemented or designed independently from the flow control algorithm. Analogous experience was reported in ISIS [BSS91].

Multicast Latency

The second metric of the performance of a multicast protocol is the latency. Latency is measured by the round trip time required by a null remote procedure call (RPC) issued by a client process and executed on a server. Thus, the latency can be measured by issuing a large number of null remote procedure calls and observing the average round trip time. The latency of the Amoeba multicast is measured directly in this manner. However, the multicast protocols of Manetho and Chang and Maxemchuck have variable delays. For Manetho, the delay depends on the overhead of processing the antecedence graph whose size is variable. For the protocol of Chang and Maxemchuck, the delay depends on how quickly the token circulates among the receivers of the multicast, which is a function of the incoming message rate and the value of the timeout used to force a token transfer if no subsequent application message arrives. The latency under a low load directly depends on the timeout value used to force the token transfer. Ideally, this timeout should be slightly higher than the round trip delay of an application message, to reduce the delay for the messages that are yet to be delivered. However, existing hardware and operating systems do not allow such short timeouts. In the experimental environment, this timeout is at least 10 milliseconds because of hardware limitations. Therefore, to make the comparison fair, two measurements for the protocol of Chang and Maxemchuck are presented. The first is identical to the experiments run for the other two protocols. The second measurement is for a set up where the clients do not use the RPC style to stream packets, but rather use datagrams and generate a high load. This set up is favorable for the protocol of Chang and Maxemchuck. In this experiment, the delay in delivery is measured as the average time between transmitting a datagram and receiving a corresponding reply.

Table 4.8 shows the multicast latency for the three protocols under study. The latency of Manetho's multicast is expressed as a function of n which is the number of event descriptors in the graph to be processed at the receiver. As a reference point, the latency of the V-System's unicast is about 1.3 millisecond.

The following conclusions are drawn from the table:

Protocol	Number of Replicas		
	2	3	4
Manetho	1.6 + 0.02n	1.6 + 0.02n	1.6 + 0.02n
Amoeba r-resilient	2.2	2.4	2.6
Chang & Maxemchuck (datagrams)	2.4	2.7	3.4
Chang & Maxemchuck (RPC)	10.0	20.0	30.0

Table 4.8 Latency of the three multicast protocols under study in millisecond.

- The latency of Manetho is directly proportional to the size of the graph that has to be processed at the receiver. Earlier measurements for Manetho showed that typically, less than six event descriptors are appended on every message for a system of 16 processors. Thus, the corresponding delay would be about 1.8 milliseconds.

- Positive acknowledgment protocols give predictable latency, making them suitable for applications where predictable performance is important, such as in real-time systems.

- Under a low load, RPC style communication is costly if used with negative acknowledgment multicasts. Therefore, application programs that are to benefit from such protocols should use a programming paradigm based on non-blocking, asynchronous communication.

Application Performance

The V-System includes a network RAM disk server that provides an in-memory file system and acts as a high-speed network file server. The network RAM disk server was run as a replicated troupe to provide highly available file service. No change to the code was required. Table 4.9 shows the performance of file reads, writes, and opens under the three multicast protocols studied and for different degrees of replication.

The following conclusions are drawn from the table:

- The throughputs reported (especially for write operations) are lower than the maximum allowed by the communication system. This is due to the fact that

File Operation	Protocol	Replicated RAM Disk Server		
		2	3	4
Write Throughput (Kbytes/sec)	Manetho	720	720	719
	Amoeba	404	394	360
	Chang & Maxemchuck	400	390	370
Read Throughput (Kbytes/sec)	Manetho	716	715	715
	Amoeba	790	785	780
	Chang & Maxemchuck	700	770	760
File Open (milliseconds)	Manetho	1.7	1.7	1.7
	Amoeba	2.4	2.6	2.8
	Chang & Maxemchuck	3.1	3.4	3.7

Table 4.9 Network RAM disk performance with the three protocols under study.

the file system protocol transfers data in eight Kbytes chunks over the network, and performs an explicit handshake with the client after each transfer.

- The throughput for read was highest for the Amoeba protocol, because transmitting packets from the replicated server does not require the two-phase agreement protocol. On the other hand, the application in Manetho has to pay the price of maintaining the antecedence graph, even for messages sent from the replicated server. Similarly, the negative acknowledgment protocol of Chang and Maxemchuck has to pay some overhead because of the interactions with the flow control algorithm that were outlined above.

- Surprisingly, the negative acknowledgment protocol performed worse for the write throughput. This is due to a combined effect of the application using a blocking communication paradigm after the transfer of each eight Kbytes of data, in addition to the interactions with the flow control. The blocking style of communication requires low latency in delivery to avoid a performance penalty. This result suggests that care must be taken when converting applications to run as replicated services on top of multicast protocols using negative acknowledgments. Only applications that are written using an asynchronous communication paradigm are likely to exploit the high throughput of negative acknowledgment protocols. Applications with inherent synchronization must employ a protocol that stresses low latency.

4.3.4 Summary

The study shows that a multicast protocol designed specifically to support active replication outperforms general protocols for that purpose. The measurements show that Manetho's multicast adds a small overhead compared to the performance of the V-System's unicast protocol. This small overhead translates to efficient operation at the application level, as shown for the RAM disk server.

The results provide evidence supporting the conventional wisdom that negative acknowledgment protocols have higher throughput than positive acknowledgment protocols, at the expense of higher latency. Furthermore, applications that are to benefit from negative acknowledgment protocols must not be based on a blocking communication paradigm, and interactions between the multicast protocol and the flow control algorithm can be a source of performance loss because of the inherent synchronization involved in flow control.

4.4 Performance of the Recovery Protocol

4.4.1 Rollback-Recovery

The recovery time consists of three components:

1. The time to restore the states of the failed recovery units: The checkpoints are read from the stable storage server at a rate of 300 Kbytes/second, which is dominated by the cost of reading the information from disk.

2. The time to run the protocol to reconstruct the antecedence graph and determine the events to be replayed: This time is proportional to the number of recovery units that participate in the computation. For 16 recovery units, running the protocol takes less than two seconds. Most of this time is spent writing the antecedence graph on stable storage as described in Section 2.8. Figure 4.1 shows the time required to run the recovery protocol as a function of the number of recovery units in the system.

3. The time to replay the execution: This time depends on the time of the failure within the checkpointing interval, the number of failures, and the nature of the application program.

Two failure scenarios were examined for each of the seven applications, with a checkpointing interval of 10 minutes. In one scenario, a failure occurred in the beginning of the checkpointing interval (10 seconds after the checkpoint). In the second scenario, a failure occurred at the end of the checkpointing interval (590 seconds after the checkpoint). These two extremes reflect the minimum and maximum replay times, respectively.

Figures 4.2 and 4.3 show the recovery times for `grid` as a function of the number of simultaneous failures for the two scenarios. The results shown for `grid` are similar to those for the other applications with sizable address spaces such as `gauss`, `matmult` and `sparse`. Figures 4.4 and 4.5 show the corresponding times for `tsp`. The results for `tsp` are similar to those for `nqueens` and `prime`, all of which have small address spaces.

Analysis

For applications with a small amount of state such as `nqueens`, `prime` and `tsp`, the cost of restoring the state and collecting the antecedence graph information is small compared to the replay time. For applications with a large amount of state, the relative cost of state restoration becomes more significant with the number of simultaneous failures, as more state needs to be restored. The effect of state restoration is also more prominent if a failure occurs immediately after a checkpoint. Otherwise, the effect of execution replay is more prominent. Measurements also show that when compared to the time of the execution before the failure, the time for replay increases with the number of recovering units. This increase is due to the additional overhead required by the recovery servers when requesting messages to be replayed.

Finally, the measurements show that the cost of reconstructing the antecedence graph is negligible compared to the inherent costs of rollback-recovery, namely the costs of restoring the state and replaying the execution. This result suggests that the recovery times in Manetho are not appreciably different from those in other transparent rollback-recovery schemes.

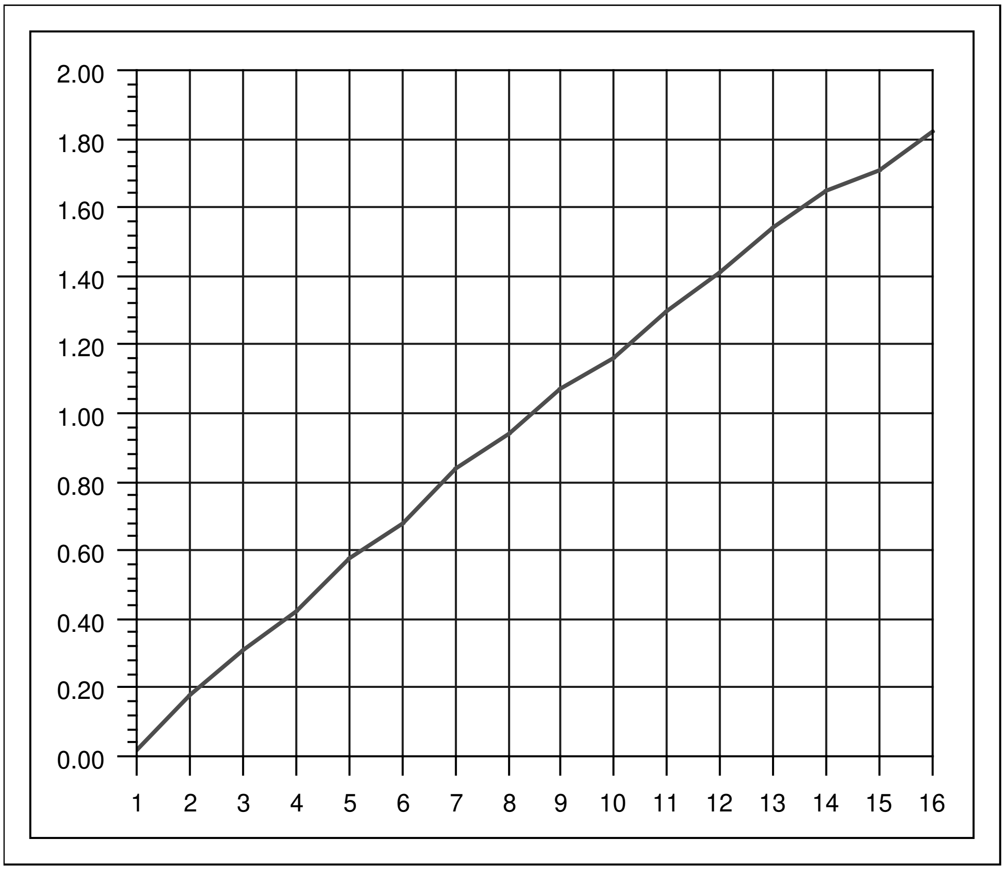


Figure 4.1 Time to run Manetho's recovery protocol as a function of the number of recovery units (in seconds).

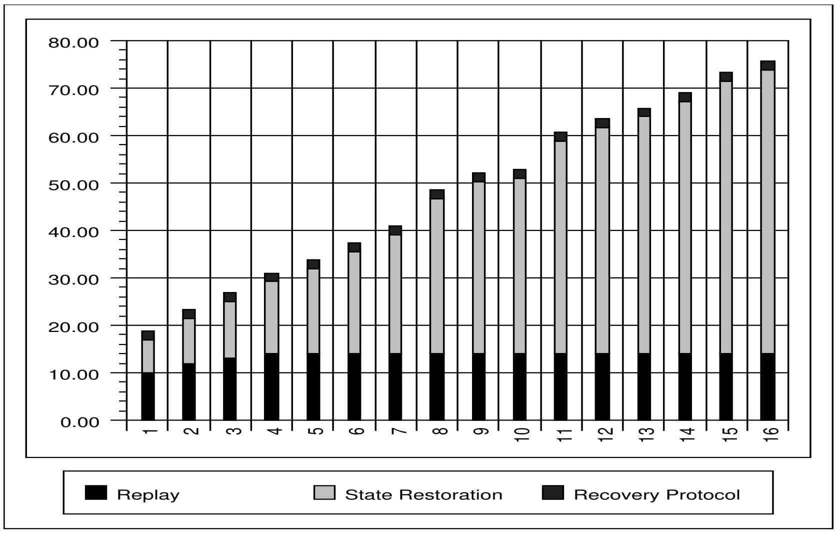


Figure 4.2 Recovery times (in seconds) for `grid` as a function of the number of simultaneous failures. Failures occur 10 seconds after taking a checkpoint.

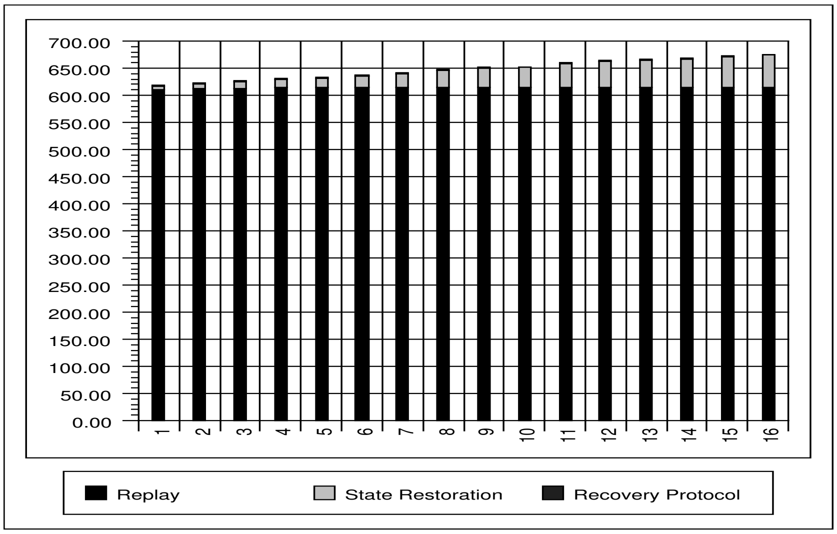


Figure 4.3 Recovery time (in seconds) for `grid` as a function of the number of simultaneous failures. Failures occur 590 seconds after taking a checkpoint.

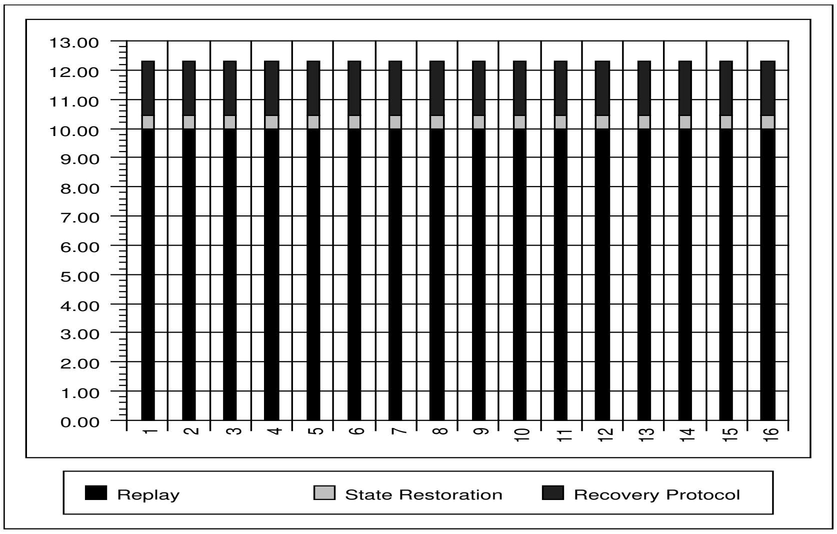


Figure 4.4 Recovery time (in seconds) for `tsp` as a function of the number of simultaneous failures. Failures occur 10 seconds after taking a checkpoint.

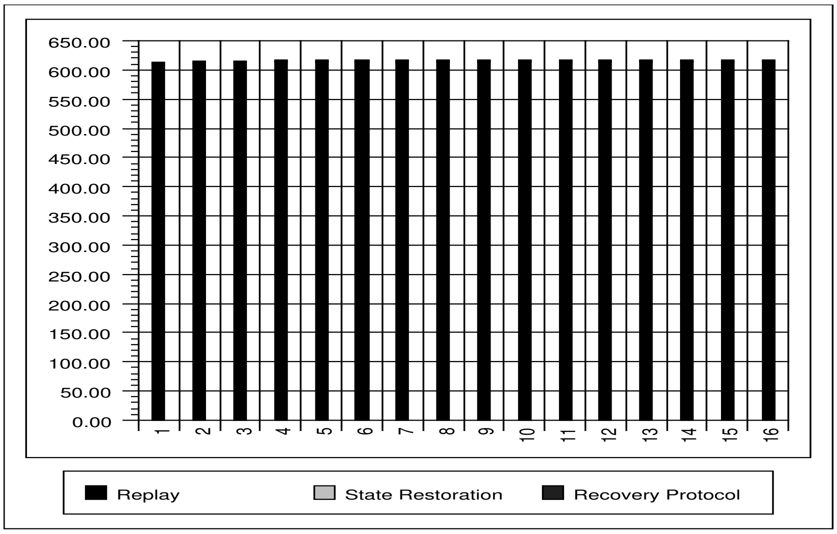


Figure 4.5 Recovery time (in seconds) for `tsp` as a function of the number of simultaneous failures. Failures occur 590 seconds after taking a checkpoint.

Effect on Running Time

The effect of a failure on the running time depends on the application. Applications fall into two categories according to the effect of a failure on the total running time:

- Applications with dynamic parallelism, such as `tsp` and `prime`: For these applications, failures may not affect the running time. In `tsp`, for instance, if a failure occurs at a slave process, the wasted time because of the failure and replay does not affect the total running time. The computation continues despite of failure.

- Applications with static parallelism and tight synchronization among the processes, such as in `gauss`, `grid` and `sparse`: For these applications, even a single failure may bring the entire computation to a halt, since the surviving processes must block waiting for the recovering process to execute recovery and replay. In these cases, the running time increases by the amount of time lost due to the failure.

Thus, while Manetho protects surviving processes from rolling back after a failure, some applications cannot benefit from this advantage because of their inherent synchronization.

4.4.2 Troupe Recovery

The cost of recovery from the failure of a troupe's leader is broken down into four components:

1. The time required to detect the failure, which is about three seconds in the implementation.

2. The time required to elect a new leader, which is less than 20 milliseconds for three cohorts.

3. The time required to run the recovery protocol, which is less than two seconds for 16 recovery units.

4. The time required to restore the cohorts to a state consistent with the system, which was negligible. In the environment used to implement Manetho, communication failures are rare. Therefore, the cohorts are always in step with the leader.

From the above analysis, the services of a replicated troupe will be unavailable for slightly more than five seconds if the leader fails. This time is dominated by the latency of detecting failures and the cost of running the recovery protocol. The failure of a cohort does not affect the availability of the service. The leader only drops the failed cohort from the list of replicas that participate in the troupe.

4.5 Summary

This chapter presented an empirical performance study of Manetho. The rollback-recovery protocol of Manetho has a small overhead. For typical compute-intensive applications, the increase in running time is about 1% on average and less than 4% for the application with the largest communication and memory requirements in the test suite. A comparison with other recovery protocols shows that Manetho's antecedence graph adds less than 1% to the failure-free performance overhead, while Manetho's output commit is lower by almost an order of magnitude.

Contrary to previous belief, coordinated checkpointing is efficient and does not differ in performance from independent checkpointing. Furthermore, coordinated checkpointing can be used with message logging to simplify garbage collection and improve performance. The study also shows that copy-on-write techniques are useful in reducing the overhead of message logging, and that sender-based message logging is superior to receiver-based message logging in terms of performance.

The second part of the performance evaluation shows that a multicast protocol designed to support active replication outperforms general protocols when used for this purpose. By exploiting the fact that the receivers of a multicast are replicas that execute the same program, Manetho's multicast protocol offers better performance in terms of latency in message delivery and throughput than alternative protocols. The price to be paid for these advantages is a loss of general applicability and a more complex recovery protocol requiring the clients of a replicated service to participate. Experiments also support the conventional wisdom that negative acknowledgment protocols outperform positive acknowledgment protocols with respect to throughput, at the cost of higher latency.

The inherent delay in negative acknowledgment protocols can have negative effects when interacting with flow control algorithms and application programs. Therefore, an algorithm for flow control must not be implemented or designed separately from the multicast protocol itself. Moreover, applications that stand to benefit from negative acknowledgment protocols should not rely on synchronous communication in their design.

Finally, the overhead of the recovery protocol is dominated by the costs of state restoration and execution replay. These costs are inherent to any rollback-recovery scheme. The cost of running the recovery protocol is negligible in comparison. For troupes, the leader and cohorts are in sync most of the time. Therefore recovery is almost instantaneous, and high availability is achieved.

Chapter 5

Related work

The research presented in this dissertation builds on a large body of previous work. This chapter presents a comparison between Manetho and other systems that provide transparent fault tolerance. Section 5.1 focuses on rollback-recovery, while Section 5.2 focuses on process replication.

5.1 Rollback-Recovery

5.1.1 Message Logging Protocols

Manetho is the first message logging protocol that uses *coordinated* checkpointing, resulting in two advantages. First, failure-free performance improves because the processes need not flush their volatile message logs to stable storage. Second, the use of coordinated checkpointing eliminates the need for a complex garbage collection algorithm. Since the global checkpoint forms a recovery line beyond which no process will roll back, recovery information describing events that occurred before the recovery line can be removed without an explicit protocol.

Previous message logging protocols exclusively used independent checkpointing in the belief that coordinated checkpointing is expensive. To the contrary, the performance measurements presented in Chapter 4 show that the overhead of coordinating the checkpoints is negligible. Moreover, the measurements show that message logging techniques have a higher overhead when they use independent checkpointing. The price of flushing the message logs on stable storage exceeds the savings resulting from eliminating the coordination of individual checkpoints to form a consistent state. Therefore, the combination of coordinated checkpointing with message logging leads to better performance than the traditional combination of message logging with independent checkpointing, unlike what is widely believed.

Optimistic Message Logging

In traditional optimistic message logging, processes take independent checkpoints and log the messages asynchronously to stable storage [Joh89, JZ90, JV91, SBY88, SY85, SW89]. The optimistic assumption is that each message will be logged on stable storage before a failure occurs and thus will be available for recovery. If a failure occurs, the processes will roll back to the maximum recoverable state and resume operation [Joh89]. The failure-free overhead of optimistic message logging is low because it does not require synchronous access to stable storage. However, there are several drawbacks. The effects of failures are not confined to the processes that fail. A process that continues to operate on a functioning machine may become an orphan and have to roll back during the recovery of other processes [SY85]. In addition, each process must maintain several checkpoints, and a garbage collection protocol is required to reclaim the space used by the message logs. Optimistic recovery also introduces delays when communicating with the outside world. Since the outside world cannot roll back, a process cannot send a message to the outside world before all messages on which the process's state depends are actually logged on stable storage. Thus, a process cannot decide locally to commit output without running a protocol with other processes. Moreover, several processes may need to flush their message logs before allowing the output to be released, which may cause delays.

Manetho offers the same performance advantages of optimistic recovery by avoiding synchronous logging, while protecting live processes from becoming orphans and allowing output to be committed locally. These advantages result from using the antecedence graph. The performance measurements show that the overhead of maintaining the antecedence graph is negligible.

Pessimistic Message Logging

In pessimistic logging, processes log recovery information synchronously on stable storage [BBG+89, Jal89, PP83]. These systems usually use hardware support to mitigate the effects of synchronous logging. Manetho's recovery protocol offers the same advantages as pessimistic logging: only one checkpoint must be maintained by each process, failures do not affect live processes, and output can be committed without multihost coordination.

Sender-Based Message Logging

Sender-based message logging is a protocol in which each message is logged in volatile memory on the machine from which the message is sent [JZ87, Joh89]. When a process receives a message, it returns to the sender a *receipt sequence number* (RSN) that indicates the order in which the message was received. When the sender receives the RSN from the receiver, it merges the RSN with the message in the volatile log, and then it sends an acknowledgment of the RSN to the receiver. The receiver cannot send any message in the interval between sending the RSN to the sender and receiving the corresponding acknowledgment. In addition, each process occasionally takes a checkpoint of its state on stable storage, but the checkpoints of individual processes are not coordinated. Each process maintains only one checkpoint of its state on stable storage, and only the process that fails has to roll back. However, sender-based message logging can tolerate only one failure at a time. It also introduces a synchronization delay in the message exchange protocol because a receiver must wait for the RSN acknowledgment before it can send any application message.

Manetho is an extension of sender-based message logging where the added antecedence graph allows recovery from an arbitrary number of failures. The antecedence graph also provides support for a limited form of nondeterministic execution, which is not possible with sender-based message logging.

5.1.2 Distributed Checkpointing

Most previous work in distributed checkpointing has concentrated on issues such as reducing the number of messages required to coordinate a checkpoint [BL88, BCS84, CJ91, LY87, LNP91, SK86, TKT89, VRL87], or limiting the number of hosts that have to participate in taking the checkpoint or in rolling back [AL89, KYA86, IM89, KT87, KT87, WF92a]. Manetho is the first system to use coordinated checkpointing with message logging. This combination can be viewed as an enhancement of coordinated checkpointing protocols to enable them to interact more efficiently with the outside world. Without message logging or dependence tracking, a protocol based solely on coordinated checkpointing must take a global checkpoint before releasing output to the outside world. The resulting latency makes such techniques unsuitable for interactive applications.

Independent Versus Coordinated Checkpointing

Bhargava et al. [BLL90] reported on the performance of independent checkpointing using simulation. They concluded that the messages used for synchronizing a checkpoint cause a large overhead. Their conclusion differs from the results presented here for two reasons. First, they use application programs with small address spaces (4 to 48 Kbytes). Second, communication is expensive in their environment. A message between two processes residing on the same machine is delivered after about 20 milliseconds from its transmission.

The study in Chapter 4 shows that the difference in performance between independent and coordinated checkpointing is marginal, and that independent checkpointing causes two important problems. First, it is susceptible to the domino effect [Ran75, Rus80], where a failure may roll back the computation to the initial state losing all work done. Second, it requires intricate garbage collection and recovery protocols, complicating the implementation and causing more overhead in processing time and in the required space on stable storage.

Implementation techniques

Johnson used a technique called precopying in implementing checkpointing [Joh89, TLC85]. In this technique, the pages to be written on stable storage are first copied to a separate area in memory if their number is below some threshold. Otherwise, a "precopying" pass is made over the entire address space where the pages modified since the most recent checkpoint are marked for writing on stable storage. The application program continues to execute and can freely modify any page. Once the kernel writes the marked pages on stable storage, the number of modified pages in the address space is reexamined. If it is above the threshold, the kernel performs additional precopying passes, up to a maximum number of passes. If the kernel reaches this maximum number, it suspends the process while writing the remaining modified pages directly from the address space on stable storage. Precopying may need to write some pages on stable storage more than once if they are modified again during a precopying pass. In addition, if the application program quickly modifies additional pages of the address space, it may have to be suspended to complete the checkpoint. The overhead introduced by copy-on-write is always less than or equal to that introduced by precopying [EJZ92].

Li et al. [LNP90] described several checkpointing methods for programs executing on shared memory multiprocessors. Their results showed that concurrent checkpointing reduces the checkpointing overhead for programs running on shared memory multiprocessors. The results presented here extend their results to distributed systems. They did not implement incremental checkpointing which proved to be an important optimization.

Kaashoek et al. [Kaa92, KMBT92] implemented consistent checkpointing to add fault tolerance to Orca, a distributed object-oriented language. Their implementation uses a broadcast protocol to order marker messages with application messages. Processes block while their checkpoints are being written on stable storage. A limited form of incremental checkpointing is used: the application code is written to the checkpoint only once, but the entire data space is written out on each checkpoint, whether modified or not.

A different technique for checkpointing relies on compiler support [LF90, LFA92]. Routines to save the state of the process are automatically inserted by the compiler in the application program. The checkpointing interval is determined statically by compiler analysis. Moreover, the analysis also can determine the program variables that have to be stored in the checkpoint, providing a means for reducing the checkpoint size. This technique provides a form of incremental checkpointing without relying on kernel support. There are two disadvantages, though. The application program must block while the state is saved, and compiler analysis may sometimes overestimate the size of the checkpoint.

5.2 Process Replication and Multicast Protocols

One contribution of the research presented here is a multicast protocol designed specifically to support process replication. In contrast, previous work in process replication relied on general purpose multicast protocols. The study presented in Chapter 4 shows that Manetho's multicast outperforms general purpose protocols in supporting process replication.

5.2.1 Process Replication

Process replication has been a subject of extensive research over the last decade. The comparison here considers only systems that operate in asynchronous distributed environments and provide replication for reliability. Systems that assume real-time requirements [SES+92], or use replication for enhancing performance [Gol91] are not considered.

CIRCUS

CIRCUS supported process replication in an asynchronous network [Coo84, Coo85]. Replicated remote procedure calls were used to implement inter-troupe communication. If identical receipt order at each replica was not required, a many-to-many RPC incurred between $r+1$ to $2r$ multicasts. Identical receipt order was possible by structuring the many-to-many RPC as a transaction. This transaction would deadlock if two members of the troupe received messages in different orders. Committing this transaction required at least r additional multicasts. In contrast, Manetho provides ordered multicast delivery with only one overhead multicast per application multicast.

CHORUS

The CHORUS system provides actors among its supported facilities [BF83]. An actor is a sequential process whose execution is divided by the programmer into processing steps that are executed sequentially. A processing step starts by receiving a message from another actor, and ends by sending messages to other actors.

To achieve fault tolerance, CHORUS uses the coupled-actors mechanism. In this design, the backup is always one processing step behind the main server. At the end of every processing step, the main server sends a copy of the request message that has been just processed to the backup. Manetho is similar to CHORUS in adopting the leader/cohort model, but Manetho's cohorts are in sync with the leader most of the time, while CHORUS's cohorts are always one processing step behind.

Clouds

The Clouds system uses transactions to support process replication [ADL90]. The computation to be replicated is written as a sequence of short transactions. For each transaction, the replicas execute the required operations in parallel. At commit time, only one replica succeeds while the remaining cohorts abort. The next transaction in program order is then run. The advantage of this approach is that it allows arbitrary nondeterministic execution in each replica. On the other hand, the application must be structured as a sequence of transactions. In comparison, Manetho limits the allowed nondeterminism, but is completely transparent to the application program.

Lazy Replication

Lazy replication advocates using the application semantics for efficient replication support [Lad89, LLSG92]. The programmer classifies the primitives of a replicated service according to their types. Non-conflicting operations are allowed to proceed in parallel at different replicas possibly in different orders. Furthermore, operations that need not be performed immediately at all replicas are performed at one site and then propagated in the background using gossips. Lazy replication offers good performance but the programmer has to specify the classes of operations and their semantic conflicts. Manetho offers similar performance without depending on the application semantics, but requires the clients of replicated services to participate in the recovery protocol.

5.2.2 Multicast Protocols

Unlike many other multicast protocols, Manetho's multicast is specifically designed for process replication. For this purpose, the combination of antecedence graph maintenance and message logging offers better throughput and latency than the protocols that have been published in the literature. The reader should bear in mind though that all multicast protocols described below are not limited to supporting active replication.

Psync and Consul

The context graph of Psync [PBS89] represents a variation on Lamport's happened-before relation [Lam78]. A group of processes participates in a conversation in which every process receives every message sent by any participant. The processes maintain a context graph that orders messages according to the context relation. Roughly, a message m_2 is sent in the context of m_1 if the sender of m_2 has sent or received m_1 before sending m_2. The processes use the context graph to deliver messages as ordered by the context relation. The context graph differs from the antecedence graph in that the former is a tool for imposing an order on message delivery, while the latter is merely a record of the order in which the messages were delivered. Furthermore, Psync orders messages sent from the same process, while message receipts are not necessarily ordered. In particular, two messages that are not related by the context order may be delivered in different orders at different processes.

The conversation abstraction requires every process to receive *every* message exchanged. Moreover, each participant must log *all* the messages it receives to tolerate failures. These restrictions force the number of processes to be small to reduce the overhead and the memory requirements to a manageable level.

For supporting recovery based on execution replay and process replication, the antecedence graph is more powerful than the context graph for the following reasons:

- The antecedence graph records the receipt orders at every process that participates in the computation. This is exactly the information required during recovery. The context graph cannot supply this information directly. Instead, a deterministic ordering function must be applied on the context graph to provide a total order on all messages. This deterministic filter introduces delay in delivering the messages. Specifically, if a message m is received, it cannot be delivered until it becomes a member of a "stable" wave, which requires that each process in the system send a message in the context of m [PBS89]. The delay thus is generally linear in the number of the processes in the conversation.

- The context graph requires that each process receive every message. Thus, a process is forced to pay the overhead of receiving all the messages whether it is the intended receiver or not. Psync also would require a large amount of volatile storage to save these messages. No such requirement exists for the antecedence graph.

- Psync does not have provisions for recording internal nondeterministic events.

For these reasons, the antecedence graph is more suitable than the context graph for supporting recovery based on execution replay.

The Consul system has been implemented on top of Psync and offers services similar to those provided by Manetho, such as rollback-recovery for client processes and replication support for server processes [MS92]. Consul is designed to exploit the semantics of the application program, so that messages can be delivered according to Psync's context order whenever possible. The context order does not guarantee that the replicas of a server receive the messages in the same order. However, since the semantics of the application are exploited, some messages could be delivered with different orders without violating the consistency of the replicas. Using context order delivery requires information about the application, but has lower latency than the total order of Psync.

In contrast with Consul, Manetho does not require knowledge about the application program but still offers low latency in delivery and ordering power equivalent to Psync's total order. In addition, Manetho does not require every host in the system to receive every message or to maintain the entire history graph of the system.

Sequencer-Based Multicasts

Having a sequencer define the receipt order of a multicast was used in the multicast protocol of Chang and Maxemchuck [CM84], the implementation of ISIS's ABCAST [BJ87b, BSS91], Amoeba's atomic broadcast protocol [KT91], and the Delta-4 system [BHV+90, CPR+92, VRB89].

The r-resilient protocol of Chang and Maxemchuck relies on negative acknowledgment and leadership transfer to achieve reliable total ordering. However, the protocol cannot deliver a multicast before $r-1$ subsequent leadership transfers occur. Like this protocol, Manetho also incurs few overhead control messages, but it avoids the delay in delivering the multicast by using the information in the antecedence graph. The performance study in Chapter 4 shows that the delay in message delivery is substantial in a broadcast protocol that follows the style of Chang and Maxemchuck, even under heavy load. The effect of this delay on the application performance can be substantial.

Manetho and the new implementation of ISIS's ABCAST [BSS91] rely on a single site to define the multicast's receipt order. ABCAST relies on an underlying transport protocol that guarantees reliable message delivery in FIFO order. This transport protocol is a major source of overhead in ISIS [BSS91]. In addition, ISIS is not application-transparent. In contrast, Manetho adopts weaker assumption about network reliability, leading to better performance. It is also application-transparent.

Amoeba's atomic broadcast uses negative acknowledgment for the 0-resilient version, and positive acknowledgment otherwise. A replicated RPC library has been implemented on top of the broadcast protocol to provide higher level support for applications [Woo93]. Amoeba's protocol is highly tuned for the 0-resilient operation mode. The r-resilient version of Amoeba's requires $r-1$ overhead messages for each application multicast. Manetho does not require such overhead messages. The performance study in Chapter 4 shows that Manetho's multicast outperforms an Amoeba-like broadcast both in throughput and latency.

The Delta-4 multicast protocol uses positive acknowledgments. Delta-4 relies on a special network adapter to provide the ordering and reliability, and to mask the overhead of acknowledgment messages from the application program. In contrast, Manetho does not depend on special network support.

Trans/Total

The atomic broadcast protocol of Melliar-Smith et al. [MSMA90] uses no control messages during normal operation. Eliminating control messages results in a delivery delay that depends mainly on the rate of incoming application broadcasts. A performance study shows that an application message may wait for seven subsequent application messages on average before it is delivered [MSMA90]. Therefore, the approach used in Trans/Total is only suitable when the rate of application multicasts is extremely high. The latency of message delivery in Manetho is small regardless of the rate of incoming application multicasts.

Transis

Transis is a protocol that combines ideas from the Trans/Total protocol with some ideas from ISIS [ADKM92]. The purpose is to exploit the broadcast facility commonly offered by local area networks to provide an efficient implementation that supports communication facilities similar to ISIS's. Manetho's multicast offers equivalent semantics to Transis's *safe* multicast, in the context of process replication. This safe multicast does not deliver the message before all receivers acknowledge its receipt, which would hurt performance.

Chapter 6

Conclusions

This dissertation presented Manetho, a new system that provides transparent fault tolerance for distributed application programs. This final chapter summarizes the contributions of the dissertation and suggests directions for further research.

6.1 Contributions

Manetho allows processes to use either rollback-recovery or active replication. Thus, client processes tolerate failures by rollback-recovery, while server processes provide high availability by replication. The system maintains an antecedence graph to make this combination possible. The graph records information about the nondeterministic events that occur during failure-free operation. The key property resulting from maintaining the antecedence graph is that each process is protected from the effects of the failures that occur to other processes. Thus, a replicated process need not roll back if a client fails. All processes participate in the recovery protocol regardless of the particular method they use.

6.1.1 A New Logging Protocol

Manetho features a novel rollback-recovery protocol based on message logging. Before this work, low failure-free performance overhead was possible in pessimistic logging protocols only by using special hardware support, or by restricting the number of failures that could be tolerated. Alternatively, optimistic recovery offered high performance at the expense of high latency in output commit, orphan creation during recovery, and cascaded rollbacks that possibly include processes that did not fail.

Manetho's rollback-recovery combines the performance advantage of optimistic systems with the failure containment and low latency in output commit of pessimistic logging. This combination is made possible by using the antecedence graph in addition to a form of sender-based message logging. No other known rollback-recovery system combines these advantages. The protocol tolerates an arbitrary number of process failures. It also tolerates failures during recovery, and arbitrary message delay.

An implementation and a performance evaluation support the performance claims. For the applications studied, the failure-free overhead increases the running time by less than 4%. The overhead is a small price for low latency in output commit, failure containment, and the elimination of cascaded rollbacks. The implementation also identified several techniques to support efficient message logging and antecedence graph maintenance.

The work also includes, for the first time, a study of the system's behavior during recovery. The costs of restoring the state and execution replay dominate the time for recovery. These costs are inherent in any rollback-recovery system. The time to run the recovery protocol is insignificant in comparison. Thus, the recovery time of Manetho's rollback-recovery is similar to that of other systems.

6.1.2 A New Multicast Protocol

Manetho features a new multicast protocol to support active process replication. The protocol uses the leader/cohort model, and relies on a combination of antecedence graph maintenance, sender-based message logging, and the deterministic execution of the replicas. This combination gives Manetho's protocol the high-throughput advantage of negative-acknowledgment protocols, without high latency in delivering application messages.

The implementation of the protocol backs the performance claims. The measured throughput differs by less than 6% from that of the unicast protocol of the V-System. Furthermore, Manetho outperforms alternative negative and positive acknowledgment protocols both in latency and throughput.

6.1.3 Integrating Rollback-Recovery and Replication

Previous work in transparent fault tolerance has addressed only rollback-recovery, without considerations for high availability issues. In contrast, Manetho's recovery protocol applies to processes that use either rollback-recovery or process replication. The system combines both methods to adapt to various application requirements. Thus, client processes can use inexpensive rollback-recovery, while server processes can use active replication to provide high availability. This combination is one of the most attractive features of Manetho, as it is becoming clear that no single recovery method suits all situations.

6.2 Future Research

Two general directions of research can follow on the work presented here. The first is to investigate some issues that were not considered in this dissertation. The second is to investigate the use of the antecedence graph in solving different problems.

6.2.1 Further Development

The work presented here left several issues unresolved. It is not clear, for example, how the system's performance would change when using more powerful processors equipped with larger amounts of memory. The effect of increasing the network bandwidth is not obvious either. With the projected increases in processor speed and memory size, one may conjecture that the cost of checkpointing will increase and dominate failure-free overhead. Similarly, the increase in processor speed and network bandwidth will decrease the absolute cost of maintaining the antecedence graph. Nevertheless, the relative effect of this decrease with respect to the increasing demands in application performance remains an open question.

The applicability of the recovery protocols presented in this dissertation to more complex systems is avenue for future research. A system such as UNIX poses several challenges. Most notably, there are many uncontrolled forms of nondeterminism in these environments. Furthermore, the monolithic kernel approach used in such systems complicates the implementation and makes execution replay during recovery difficult.

The scaling of Manetho's recovery protocol is also an issue that requires further investigation. The implementation presented here shows that the performance overhead is very low for a 16-processor multicomputer. The effect of increasing the number of processors that participate in the protocol is not clear.

Another problem worth investigating is the effect of using faster stable storage devices on the system's performance. The current technology of relying on magnetic disks to provide stable storage is reaching its limit in performance. On the other hand, the speed of processors and networks is expected to improve by several orders of magnitude in the near future. Therefore, stable storage devices are likely to create a serious performance bottleneck in any fault-tolerant system. Faster stable storage devices that rely on high-speed stable semiconductor memory are bound to be necessary in the future.

6.2.2 Applicability to Other Problems

The protocols presented in this dissertation rely on the antecedence graph and message logging to provide application-transparent fault tolerance. An avenue worth pursuing is to study the applicability of the ideas presented here for transaction-based systems. Transactions always reveal some information about the semantics of the application program. It would be useful if such information is used with some combination of dependence tracking to improve performance.

The problem of debugging distributed applications is another area where the antecedence graph can be applied directly. The history information provided by the antecedence graph can be used also for debugging purposes. Some adaptations are required, such as including message transmission events in the history description.

Finally, the multicast protocol presented in this dissertation applies only to active process replication. An open problem is whether the protocol can be adapted to support general group communication, while retaining the advantages of low latency and high throughput.

Bibliography

[ABB+86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer Usenix Conference*, pages 93–111, July 1986.

[ADKM92] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication subsystem for high availability. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[ADL90] M. Ahamad, P. Dasgupta, and R.J. LeBlanc. Fault-tolerant atomic computations in an object-based distributed system. *Distributed Computing*, 4:69–80, 1990.

[AL89] M. Ahamad and L. Lin. Using checkpoints to localize the effects of faults in distributed systems. In *Proceedings of the 8th Symposium on Reliable Distributed Systems*, pages 1–11, October 1989.

[BBG+89] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[BCS84] D. Briatico, A. Ciuffoletti, and L. Simoncini. A distributed domino-effect free recovery algorithm. In *Proceedings of the 4th Symposium on Reliable Distributed Systems*, pages 207–215, October 1984.

[BF83] J.S. Banino and J.C. Fabre. Distributed and coupled actors: A CHORUS proposal for reliability. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 128–134, October 1983.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, Massachusetts, 1987.

[BHV$^+$90] P.A. Barrett, A.M. Hilborne, P. Verissimo, L. Rodrigues, P.G. Bond, D.T. Seaton, and N.A. Speirs. The Delta-4 extra performance architecture XPA. In *Proceedings of the 20th International Symposium on Fault-Tolerant Computing*, pages 481–488, June 1990.

[Bir85] K.P. Birman. Replication and fault-tolerance in the ISIS system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 79–86, December 1985.

[BJ87a] K.P. Birman and T.A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 123–138, November 1987.

[BJ87b] K.P. Birman and T.A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[BL88] B. Bhargava and S-R. Lian. Independent checkpointing and concurrent rollback recovery for distributed systems — an optimistic approach. In *Proceedings of the 7th Symposium on Reliable Distributed Systems*, pages 3–12, October 1988.

[BLL90] B. Bhargava, S-R. Lian, and P-J. Leu. Experimental evaluation of concurrent checkpointing and rollback-recovery algorithms. In *Proceedings of the International Conference on Data Engineering*, pages 182–189, March 1990.

[BMST92] N. Budhiraja, K. Marzullo, F.B. Schneider, and S. Toueg. Optimal primary-backup protocols. Technical Report TR92-1299, Cornell University, August 1992.

[BN84] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[Che88] D.R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[CJ91] F. Cristian and F. Jahanian. A timestamp-based checkpointing protocol for long-lived distributed computations. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 12–20, September 1991.

[CL85] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[CM84] J. Chang and N.F. Maxemchuck. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[Coo84] E.C. Cooper. Replicated procedure call. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, pages 220–232, August 1984.

[Coo85] E.C. Cooper. Replicated distributed programs. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 63–78, December 1985.

[CPR+92] M. Chérèque, D. Powell, P. Reynier, J-L. Richier, and J. Voiron. Active replication in Delta-4. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, pages 28–37, July 1992.

[EJZ92] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.

[EZ92a] E.N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit. *IEEE Transactions on Computers Special Issue On Fault-Tolerant Computing*, 41(5):526–531, May 1992.

[EZ92b] E.N. Elnozahy and W. Zwaenepoel. Replicated distributed processes in Manetho. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, pages 18–27, July 1992.

[FR86] R. Fitzgerald and R.F. Rashid. The integration of virtual memory management and interprocess communication in Accent. *ACM Transactions on Computer Systems*, 4(2):147–177, May 1986.

[GGL+90] A. Goldberg, A. Gopal, K. Li, R. Strom, and D. Bacon. Transparent recovery of Mach applications. In *Proceedings of the Usenix Mach Workshop*, pages 169–184, October 1990.

[GM82] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, 31(1):48–59, January 1982.

[Gol91] A. Goldberg. *Optimistic Algorithms for Distributed Transparent Process Replication.* PhD thesis, University of California, Los Angeles, 1991.

[IM89] S. Israel and D. Morris. A non-intrusive checkpointing protocol. In *The Phoenix Conference on Communications and Computers*, pages 413–421, 1989.

[Jal89] P. Jalote. Fault tolerant processes. *Distributed Computing*, 3:187–195, 1989.

[Joh89] D.B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing.* PhD thesis, Rice University, December 1989.

[Joh93] D.B. Johnson. Efficient transparent optimistic rollback recovery for distributed application programs. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, October 1993.

[JV91] T. Juang and S. Venkatesan. Crash recovery with little overhead. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 454–461, May 1991.

[JZ87] D.B. Johnson and W. Zwaenepoel. Sender-based message logging. In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing*, pages 14–19, June 1987.

[JZ88] D.B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pages 171–181, August 1988.

[JZ90] D.B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11(3):462–491, September 1990.

[Kaa92] M.F. Kaashoek. *Group Communication in Distributed Computer Systems*. PhD thesis, Vrije Universiteit, December 1992.

[KMBT92] M.F. Kaashoek, R. Michiels, H.E. Bal, and A.S. Tanenbaum. Transparent fault-tolerance in parallel orca programs. In *Symposium on Experiences with Distributed and Multiprocessor Systems III*, pages 297–312, March 1992.

[KT87] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[KT91] M.F. Kaashoek and A.S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 222–230, May 1991.

[KYA86] K.H. Kim, J.H. You, and A. Aboulnaga. A scheme for coordinated execution of independently designed recoverable distributed processes. In *Proceedings of the 16th International Symposium on Fault-Tolerant Computing*, pages 130–135, 1986.

[Lad89] R. Ladin. *A Method for Constructing Highly Available Services and a Technique for Distributed Garbage Collection*. PhD thesis, Massachusetts Institute of Technology, May 1989.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LF90] C.C. Li and W.K. Fuchs. CATCH: Compiler-assisted techniques for checkpointing. In *Proceedings of the 20th International Symposium on Fault-Tolerant Computing*, pages 74–81, 1990.

[LFA92] J. Long, W.K. Fuchs, and J.A. Abraham. Compiler-assisted static checkpoint insertion. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, pages 58–65, July 1992.

[LLSG92] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10:360–391, November 1992.

[LMC87] T.J. LeBlanc and J.M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[LNP90] K. Li, J.F. Naughton, and J.S. Plank. Real-time, concurrent checkpoint for parallel programs. In *Proceedings of the 1990 Conference on the Principles and Practice of Parallel Programming*, pages 79–88, March 1990.

[LNP91] K. Li, J.F. Naughton, and J.S. Plank. Checkpointing multicomputer applications. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 1–10, October 1991.

[LS79] B.W. Lampson and H.E. Sturgis. Crash recovery in a distributed data storage system. Technical report, Xerox Palo Alto Research Center, April 1979.

[LY87] T.H. Lai and T.H. Yang. On distributed snapshots. *Information Processing Letters*, 25:153–158, May 1987.

[MS92] S. Mishra and R. Schlichting. Abstractions for constructing dependable distributed systems. Technical Report TR02-19, University of Arizona, 1992.

[MSMA90] P.M. Melliar-Smith, L.E. Moser, and V. Agrawala. Broadcast protocols for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):17–25, January 1990.

[OCD+88] J.K. Ousterhout, A.R. Cherenson, F. Douglis, M.N. Nelson, and B.B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[OCH+85] J.K. Ousterhout, H. Da Costa, D. Harrison, J.A. Kunze, M. Kupfer, and J.G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 15–24, 1985.

[Pau88] Randy Pausch. *Adding Input and Output to the Transactional Model.* PhD thesis, Carnegie Mellon University, August 1988.

[PBS89] L.L. Peterson, N.C. Bucholz, and R.D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[PP83] M.L. Powell and D.L. Presotto. Publishing: A reliable broadcast communication mechanism. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 100–109, October 1983.

[Ran75] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[Rus80] D.L. Russell. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, SE-6(2):183–194, March 1980.

[SBY88] R.E. Strom, D.F. Bacon, and S.A. Yemini. Volatile logging in n-fault-tolerant distributed systems. In *Proceedings of the 18th International Symposium on Fault-Tolerant Computing*, pages 44–49, June 1988.

[SES+92] S.K. Shrivastava, P.D. Ezhilchelvan, N.A. Speirs, S. Tao, and A. Tully. Principal features of the VOLTAN family of reliable node architectures for distributed systems. *IEEE Transactions on Computers Special Issue On Fault-Tolerant Computing*, 41(5):542–549, May 1992.

[SK86] M. Spezialetti and P. Kearns. Efficient distributed snapshots. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 382–388, May 1986.

[SS83] R.D. Schlichting and F.B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[SW89] A.P. Sistla and J.L. Welch. Efficient distributed recovery using message logging. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 223–238, August 1989.

[SY85] R.E. Strom and S.A. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[Tan88] A.S. Tanenbaum. *Computer Networks.* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[TKT89] Z. Tong, R.Y. Kain, and W.T. Tsai. A lower overhead checkpointing and rollback recovery scheme for distributed systems. In *Proceedings of the 8th Symposium on Reliable Distributed Systems*, pages 12–20, October 1989.

[TLC85] M. Theimer, K. Lantz, and D.R. Cheriton. Preemptable remote execution facilities in the V-system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 2–12, December 1985.

[TS84] Y. Tamir and C.H. Séquin. Error recovery in multicomputers using global checkpoints. In *1984 International Conference on Parallel Processing*, pages 32–41, August 1984.

[VDG86] V-System Development Group. *V-System 6.0 Reference Manual.* Computer Systems Laboratory, Department of Computer Science and Electrical Engineering, Stanford University, 1986.

[VRB89] P. Veríssimo, L. Rodrigues, and M. Baptista. A highly parallel atomic multicast protocol. In *Proceedings of the Sigcomm '89 Symposium*, pages 83–93, September 1989.

[VRL87] K. Venkatesh, T. Radhakrishnan, and H.F. Li. Optimal checkpointing and local recording for domino-free rollback recovery. *Information Processing Letters*, 25:295–303, July 1987.

[Wel84] T.A. Welch. A technique for high performance data compression. *IEEE Computer*, 17(6):8–19, June 1984.

[WF92a] Y.-M. Wang and W.K. Fuchs. Optimistic message logging for independent checkpointing in message-passing systems. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 147–154, October 1992.

[WF92b] Y.-M. Wang and W.K. Fuchs. Scheduling message processing for reducing rollback propagation. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, pages 204–211, July 1992.

[Woo93] M. Wood. Replicated RPC using Amoeba closed group communication. In *Proceedings of the Thirteenth International Conference on Distributed Computing Systems*, pages 499–507, May 1993.