RICE UNIVERSITY

Distributed System Fault Tolerance Using Message Logging and Checkpointing

by

David Bruce Johnson

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

Doctor of Philosophy

Approved, Thesis Committee:

Willy Zwaenepoel, Chairman
Associate Professor of Computer Science

Robert S. Cartwright
Professor of Computer Science

John E. Dennis
Professor of Mathematical Sciences

Houston, Texas

December, 1989

Distributed System Fault Tolerance Using Message Logging and Checkpointing

David Bruce Johnson

Abstract

Fault tolerance can allow processes executing in a computer system to survive failures within the system. This thesis addresses the theory and practice of transparent fault-tolerance methods using *message logging and checkpointing* in distributed systems. A general model for reasoning about the behavior and correctness of these methods is developed, and the design, implementation, and performance of two new low-overhead methods based on this model are presented. No specialized hardware is required with these new methods.

The model is independent of the protocols used in the system. Each process state is represented by a *dependency vector*, and each system state is represented by a *dependency matrix* showing a collection of process states. The set of system states that have occurred during any *single* execution of a system forms a *lattice*, with the sets of consistent and recoverable system states as *sublattices*. There is thus always a *unique* maximum recoverable system state.

The first method presented uses a new *pessimistic* message logging protocol called *sender-based message logging*. Each message is logged in the local *volatile* memory of the machine from which it was *sent*, and the order in which the message was *received* is returned to the sender as a *receive sequence number*. Message logging overlaps execution of the receiver, until the receiver attempts to *send* a new message. Implemented in the V-System, the maximum measured failure-free overhead on distributed application programs was under 16 percent, and average overhead measured 2 percent or less, depending on problem size and communication intensity.

Optimistic message logging can outperform pessimistic logging, since message logging occurs *asynchronously*. A new optimistic message logging system is presented that guarantees to find the *maximum* possible recoverable system state, which is not ensured by previous optimistic methods. All logged messages *and* checkpoints are utilized, and thus some messages received by a process before it was checkpointed may not need to be logged. Although failure recovery using optimistic message logging is more difficult, failure-free application overhead using this method ranged from only a maximum of under 4 percent to much less than 1 percent.

Acknowledgements

I would most like to thank my thesis advisor, Willy Zwaenepoel, for his advice and guidance, which helped to nurture this research. Through many discussions with him, this thesis was improved in both content and presentation. Although I argued with him on many points of writing style and organization, he was often right, and these interactions have been a valuable part of my education. I would also like to express my appreciation to the other two members of my thesis committee, Corky Cartwright and John Dennis, for their support and encouragement in my thesis work and throughout my graduate education.

Many other people and organizations have also helped to make this thesis a reality. The other members of the systems group here at Rice University, in particular Rick Bubenik, John Carter, Elmootazbellah Nabil Elnozahy, Jerry Fowler, Pete Keleher, and Mark Mazina, have contributed to this work through numerous discussions and through comments on earlier drafts of portions of this material. Pete Keleher also helped with much of the implementation of the new optimistic message logging system presented in this thesis. I would also like to thank Ivy Jorgensen, Leah Stratmann, and Lois Barrow, for excellent administrative support, often in times of crisis; and Bill LeFebvre, for help with the laser printer and `tpic`, and for being a good friend. Ken Birman, David Cheriton, Matthias Felleisen, Elaine Hill, Ed Lazowska, Alejandro Schaffer, and Rick Schlichting also provided many useful comments on earlier drafts of some material that is reflected in this work. Financial support for this research was supplied in part by the National Science Foundation under grants CDA-8619893 and CCR-8716914, and by the Office of Naval Research under contract ONR N00014-88-K-0140.

Finally, I would like to thank my parents, who have supported and encouraged me in all my endeavors. I couldn't have done it without them.

Contents

Illustrations

Tables

Chapter 1

Introduction

Providing *fault tolerance* in a computer system allows executing processes to survive failures within the system. Without fault tolerance, an application program executing in parallel on multiple processors in a *distributed system* could fail entirely if even a single processor executing part of it fails. Although the idea of adding fault tolerance to distributed systems is not new, many existing methods restrict the types of application programs that can be supported, or severely degrade the performance of these programs in the absence of failures. A *low-overhead* method of providing fault tolerance is important in order not to penalize distributed application programs that use it. If the provision of fault tolerance adds substantial overhead to the system, it may not be usable by many types of application programs. Furthermore, a *transparent general-purpose* fault-tolerance method is desirable in order to allow fault tolerance to be easily utilized by existing application programs without changes and by new application programs without programmer effort. This transparency also allows the system fault-tolerance support and the application programs that use it to evolve independently.

This thesis addresses the class of fault-tolerance methods using *message logging and checkpointing* in distributed systems. A new general model for reasoning about the behavior and correctness of these fault-tolerance methods is developed, and the design, implementation, and performance of two new fault-tolerance methods based on this model are presented. These new methods are transparent and add little overhead to the execution of the system. They support general-purpose computation and each offer unique advantages over previous message logging systems. No specialized hardware is required with these new methods.

1.1 Message Logging and Checkpointing

In general, in a system using message logging and checkpointing to provide fault tolerance, each message received by a process is recorded in a *message log*, and the state of each process is occasionally saved as a *checkpoint*. Each process is checkpointed individually, and no coordination is required between the checkpointing of different processes. The logged messages and checkpoints are stored in some way that survives any failures that the system is intended to recover from, such as by writing them to stable storage on disk [Lampson79, Bernstein87].

Recovery of a failed process using these logged messages and checkpoints is based on the assumption that the execution of the process is *deterministic* between received input messages. That is, if two processes start in the same state and receive the same sequence of input messages, they must produce the same sequence of output messages and must finish in the same state. The state of a process is thus completely determined by its starting state and by the sequence of messages it has received.

A failed process is restored using some previous checkpoint of the process and the log of messages received by that process after that checkpoint and before the failure. First, the state of the failed process is reloaded from the checkpoint onto some available processor. The process is then allowed to begin execution, and the sequence of logged messages originally received by the process after this checkpoint are replayed to it from the log. These replayed messages must be received by the process during recovery *in the same order* in which they were received before the failure. The recovering process then reexecutes from this checkpointed state, based on the same input messages in the same order, and thus deterministically reaches the state it was in after this sequence of messages was originally received. During this reexecution, the process will resend any messages that it sent during this same execution before the failure. These duplicate messages must be detected and ignored by their receivers, or must not be allowed by the system to be transmitted again during recovery.

The protocols used for message logging can be divided into two groups, called *pessimistic message logging* and *optimistic message logging*, according to the level of synchronization imposed by the protocol on the execution of the system. This characteristic of the message logging protocol used in a system also determines many characteristics of the checkpointing and failure recovery protocols required.

1.1.1 Pessimistic Message Logging

Pessimistic message logging protocols log messages *synchronously*. The protocol guarantees that any failed processes can be recovered individually without affecting the states of any processes that did not fail, and prevents processes from proceeding until the logging that is required by this guarantee has been completed. These protocols are called "pessimistic" because they assume that a failure could occur at any time, possibly before the needed logging is completed.

Previous pessimistic message logging protocols [Powell83, Borg83, Borg89] have achieved this guarantee by blocking a process when it receives a message, until that message has been logged. This ensures that if a process fails, all messages received by it since its last checkpoint are logged, regardless of when the failure occurs. Failure recovery in a system using pessimistic message logging is straightforward. A failed process is always restarted from its most recent checkpoint, and all messages received by that process after the checkpoint are replayed to it from the log, in the same order in which they were received before the failure. Based on these messages, the process deterministically reexecutes from the state restored from its checkpoint to the state it was in at the time of the failure.

Pessimistic message logging protocols have the advantage of being able to restore the system after a failure without affecting the states of any processes that did not fail. Also, since a process can always be recovered from its most recent checkpoint, only one checkpoint for each process must be saved, and the amount of reexecution necessary to complete recovery from any failure can be controlled by the frequency with which new checkpoints are recorded. The checkpointing frequency can also be used to control the amount of storage necessary to store the message log, since only messages received by a process since its most recent checkpoint must be saved. The main drawback, however, of pessimistic message logging is the performance degradation caused by the synchronization in the message logging protocol. Previous pessimistic message logging protocols [Powell83, Borg83, Borg89] have attempted to reduce this overhead by using special-purpose hardware to assist the logging.

1.1.2 Optimistic Message Logging

In contrast to pessimistic protocols, *optimistic* message logging protocols operate *asynchronously*. The receiver of a message is not blocked, and messages are logged after receipt, for example by grouping several messages and writing them to stable storage in a single operation. However, the current state of a process can only be recovered if all messages received by the process since its last checkpoint have been logged, and thus some execution of a failed process may be lost if the logging has not been completed before a failure occurs. These protocols are called "optimistic" because they assume that the logging of each message received by a process will be completed before the process fails, and are designed to handle this case most efficiently. As with pessimistic message logging, each failed process is then recovered individually, and no process other than those that failed are rolled back during recovery. However, if the assumption turns out to be incorrect, and some received messages have not been logged when a process fails, a more expensive recovery procedure is used, which may require some processes that did not fail to be rolled back as well.

Optimistic message logging protocols have the advantage of significantly reducing the overhead caused by message logging, relative to that caused by pessimistic logging protocols. Since messages are logged asynchronously after receipt, no synchronization delays are necessary during message logging. Although optimistic logging protocols require a more complex procedure to recover the system after a failure, this recovery procedure is only used when a failure occurs. Thus, optimistic message logging protocols are desirable in systems in which failures are infrequent and failure-free performance is of primary concern. The main drawback of optimistic message logging is that failure recovery may take longer to complete, since more process rollbacks may be required. Although the amount of process reexecution necessary to complete recovery cannot be controlled as directly as with pessimistic message logging, it is generally reduced by more frequent checkpointing and by logging messages sooner after receipt. Optimistic message logging may also require more space to store checkpoints and logged messages, since any process may be forced to roll back to a checkpoint earlier than its most recent, and all messages received by the process since this earlier checkpoint must be saved in the log. Although several previous optimistic message logging protocols have been proposed [Strom85, Strom88, Sistla89], none of these protocols has been implemented, and thus their actual performance cannot be evaluated.

Optimistic message logging is a type of *optimistic algorithm* or *optimistic computation*. Optimistic algorithms have also been used in other areas, including concurrency control mechanisms [Kung81, Gherfal85], the automatic parallelization of programs for multiprocessors and distributed systems [Strom87], distributed discrete event simulation [Jefferson87, Jefferson85], software development tools [Bubenik89], and network protocols for bulk data transfer [Carter89]. An optimistic algorithm has the potential to outperform a corresponding pessimistic algorithm if the assumption made by the algorithm is correct often enough to compensate for the increased complexity of allowing computations to be rolled back otherwise.

1.2 Distributed System Assumptions

The fault-tolerance methods developed in this thesis are designed to work in existing distributed systems without the addition of specialized hardware to the system or specialized programming to applications. The following assumptions about the underlying distributed system are made:

- The system is composed of a network of fail-stop processors [Schlichting83]. A fail-stop processor immediately halts whenever any failure of the processor occurs; it thus never produces incorrect output because of any failure.

- Processes communicate with one another only through messages.

- The execution of each process in the system is *deterministic* between received input messages. That is, if two processes start in the same state and receive the same sequence of input messages, they must produce the same sequence of output messages and must finish in the same state. The current state of a process is thus completely determined by its starting state and the sequence of messages it has received.

- No global clock is available in the system.

- The network includes a shared stable storage service [Lampson79, Bernstein87] that is always accessible to all active nodes in the system.

- Packet delivery on the network need not be guaranteed, but reliable delivery of a packet can be achieved by retransmitting it a limited number of times until an acknowledgement arrives from its destination.

- The network protocol supports broadcast communication. All active nodes can be reached by a limited number of retransmissions of a broadcast packet.

- Processes use a *send sequence number* (*SSN*) for duplicate message detection. For each message sent, the SSN is incremented and the new value is used to tag the message. Each process also maintains a table recording the highest SSN value tagging a message received from each other process. If the SSN tagging a new message received is not greater than the current table entry for its sender, the message is considered to be a duplicate.

1.3 Implementation Environment

These new fault-tolerance methods have been implemented, and the performance of these implementations has been measured. This work has been carried out in the environment of the V-System, a distributed operating system originally developed at Stanford University [Cheriton83, Cheriton84, Cheriton88]. The V-System runs on a collection of diskless SUN workstations connected by a 10-megabit per second

Ethernet local network [Metcalfe76, Shoch80] to a shared SUN network file server. For this implementation, version 6.0 of the V-System [Stanford86] was used, which does not support demand-paged virtual memory. Although this implementation has involved a number of system-specific issues, many of these issues would be important with any choice of implementation environment, and many of the solutions used are applicable over a wide range of similar target systems.

The V-System consists of a distributed operating system kernel and a collection of server processes. Each participating node on the network executes an independent copy of the V kernel, and these kernels cooperate to provide the abstraction of a single distributed kernel supporting transparent location-independent message-passing communication between processes. The kernel itself provides only the primitive operations required by the system, such as communication between processes, and basic process and memory management functions. All other facilities provided by the V-System are built on top of this kernel support by server processes.

Each node executes a standard collection of server processes. Perhaps the most important of these is the *team server*, which manages all programs executing on that node. This includes loading new programs and terminating existing ones when requested, scheduling the execution of programs, and maintaining a directory of currently executing programs. The *exec server* manages one or more command interpreters known as *execs*, which correspond to the *shell* in Unix [Bourne78]. Other standard servers include a *terminal server*, which controls the user interaction through the keyboard, mouse, and display of the workstation; and the *exception server*, which uniformly handles all exceptions and traps incurred by executing processes.

Process communication is structured around a synchronous request-response protocol. The `Send` operation is used to send a fixed-size 32-byte message to another process and await a response. In the destination process, the `Receive` and `Reply` operations are used, respectively, to receive the message and to return a 32-byte reply, which overwrites the original message in the sender. The receiver process may also instead use the `Forward` operation to send the message to a new receiver, which then has the responsibility to `Reply` to the original sender. A *data segment* of up to 1 kilobyte may also be appended to the message or its reply, and larger blocks of data can be moved in either direction between the sender and receiver through the `MoveFrom` and `MoveTo` operations. These operations all use retransmissions and different forms of acknowledgements to implement reliable packet delivery over the network. The `Send` operation can also be used to send a message to a *group* of processes [Cheriton85] or to send a message as a *datagram*, with no guarantee of reliable transmission.

In the V-System, multiple processes on the same node may share a single address space, forming a *team*. Multiple teams executing together may form a *logical host*, and multiple logical hosts may execute concurrently on a single node. All processes executing as part of the same logical host must be located at the same physical network address, and thus they must remain together through failure and recovery. Therefore, in terms of the distributed system model presented in Section 1.2, each logical host in the V-System is treated as a single process in implementing these fault-tolerance methods. For example, each logical host is checkpointed as a unit rather than checkpointing each process individually, and all execution within each logical host is assumed to be deterministic.

1.4 Other Fault-Tolerance Methods

Many methods other than message logging and checkpointing have been used to provide fault tolerance in computer systems. This section summarizes these other methods, dividing them into six general classes, and comparing each to the class of methods using message logging and checkpointing. Some comparison is also made to the new methods developed in this thesis, although more specific comparisons with many of these systems will be made in the following chapters as these new methods are presented.

1.4.1 Hardware Fault Tolerance

Fault-tolerance methods implemented entirely in hardware [Carter85, Siewiorek86] may be able to outperform those implemented in software. Two examples of large systems using *hardware fault-tolerance* methods are the ESS electronic telephone switching systems developed by AT&T [Clement87] and the ARPANET Pluribus IMP [Katsuki78]. Such hardware methods, though, are less flexible and cannot easily be added to existing systems. Since the fault-tolerance methods using message logging and checkpointing developed in this thesis use no specialized hardware, these hardware methods will not be discussed further.

1.4.2 Application-Specific Methods

Application-specific fault-tolerance methods [Denning76, Anderson81, Shoch82] are those designed specifically for the particular program that is to use them. These designs require knowledge of both the application program and its environment. Each type of failure that can occur in the system must be anticipated, and specific solutions for each must be programmed. Implementation of these methods may follow some general structure such as the use of *recovery blocks* [Horning74, Lee78, Randell75], or may be structured specially for each application program. However, these methods are *nontransparent*, and thus require existing programs to be carefully modified or rewritten in order to be fault-tolerant. In contrast, fault-tolerance methods using message logging and checkpointing are general-purpose and can be applied transparently to new and existing programs. Although in some cases, application-specific methods can be made to be more efficient than transparent general-purpose methods, they are limited by their lack of transparency.

1.4.3 Atomic Actions

The use of *atomic actions* to provide fault tolerance [Lomet85] is similar to the use of atomic transactions in database systems [Gray79, Lampson81, Bernstein87], but atomic actions can perform arbitrary operations on user-defined data types. By structuring application programs as a series of possibly nested atomic actions, any failure can be recovered from by simply forcing each failed action to *abort* and then reinvoking it from its beginning. Examples of systems using atomic actions for fault tolerance include ARGUS [Liskov83, Liskov87, Liskov88], TABS [Spector85b, Spector85a], Camelot [Spector86, Spector87], the original implementation of ISIS [Birman85], Clouds [LeBlanc85, Allchin83], QuickSilver [Haskin88], and Gutenberg [Vinter86]. In general, these methods can provide fault tolerance only for application programs that are specifically designed to use them. However, many existing applications do not fit this model, and it is not always practical to program new applications in this way. Since fault tolerance using message logging and checkpointing does not require the use of any specialized programming model, it can be used over a wider range of applications. Also, with atomic actions, any failed action must typically be restarted from its beginning, which may require a large amount of computation to be redone. Smaller atomic actions may be used to reduce the amount of reexecution necessary, but this increases the overhead of beginning and ending new actions. With message logging and checkpointing, the amount of reexecution can be controlled by the frequency with which new checkpoints are written, which is independent of the design of the application.

1.4.4 Functional Programming

The programming language model of *functional programming*, in which all execution is performed through the application of functions that return results and produce no side effects, can be used to facilitate the provision of fault tolerance [Grit84]. In a distributed system, these "functions" may actually be processes that are invoked by a message and return their results at completion in a message, but such processes must retain no internal state between invocations. If a process evaluating some function fails, it can simply be restarted from the beginning with the same parameters. Examples of systems using this type of functional programming for fault tolerance include DIB [Finkel87], STARDUST [Hornig84], Lin and Keller's work [Lin86], and the multi-satellite star model [Cheriton86b]. Although these methods are straightforward, they can only be used for programs that follow the functional programming model, which may not be possible or practical for some applications. Also, since failed functions must be restarted from their beginning, the amount of reexecution needed for recovery may be large. Fault tolerance methods using message logging and checkpointing do not have these limitations.

1.4.5 Active Replication

Active replication involves the concurrent execution multiple independent copies of each process on separate processors, such that each replica of the same process receives the same sequence of input, and each is expected to produce the same sequence of output. If one replica fails, the remaining replicas of that process continue the computation without interruption. The methods of *N-modular redundancy* and *N-version programming* are special cases of this class of fault-tolerance methods. Examples of systems using active replication include the new ISIS system ($ISIS_2$) [Birman87], Circus [Cooper85, Cooper84], CHORUS [Banino85, Banino82], MP [Gait85], MARS [Kopetz85b], FT Concurrent C [Cmelik88], PRIME [Fabry73], SIFT [Wensley78], and Yang and York's work on the Intel iAPX 432 [Yang85]. These methods are well suited for use in real-time systems, since failure recovery is essentially immediate. However, this ability requires extra processors to be dedicated to each program for its replicas. Message logging and checkpointing allows all processors to be used for normal work during failure-free execution, although failure recovery times may be longer. Since message logging and checkpointing is not designed to support real-time systems, this tradeoff is practical.

1.4.6 Checkpointing

Checkpointing has also been used alone, without message logging, to provide fault tolerance. In a distributed system, though, each node must be checkpointed individually, and these separate checkpoints must be coordinated in order to be able to recover the system. Some systems have used *ad hoc* rules to determine when to checkpoint each process, effectively causing each process to checkpoint each time it communicates with another process. Examples of these systems include Eden [Almes85, Lazowska81] and the Tandem NonStop system [Bartlett81, Dimmer85]. These systems force processes to checkpoint frequently, creating a large overhead for the provision of fault tolerance, as has been recognized in these two systems [Black85, Bartlett87]. A different style of using checkpointing alone is to periodically create a *global checkpoint* of the entire system, using a protocol to coordinate the checkpointing of the individual component processes. Examples of this type of global checkpointing protocol are those proposed by Chandy and Lamport [Chandy85], and by Koo and Toueg [Koo87]. Although these protocols avoid the large overhead of checkpointing on each communication, they still may be expensive to employ and may implicitly or explicitly interfere with the underlying execution of the system. In addition, since global checkpoint algorithms cause all processes to checkpoint within a short period of time, any shared network file server on which these checkpoints are recorded may become a bottleneck to their performance. With message logging with checkpointing, each process is only checkpointed infrequently, and no global coordination is required during execution.

1.5 Scope of the Thesis

This thesis contributes to both the theory and the practice of providing fault tolerance in distributed systems using message logging and checkpointing. Emphasis is placed primarily on the message logging and failure recovery protocols used in these systems, although process checkpointing is also considered. The theoretical framework developed in this thesis is independent of the particular message logging and checkpointing methods used by the system. Based on this framework, two new fault-tolerance methods using message logging and checkpointing are presented, and a proof of the correctness of each method is given. As discussed in Section 1.3, these new methods have been fully implemented, and a complete evaluation of their performance is included.

Methods for detecting failures within the system are largely independent of these protocols and are not discussed in this thesis. Likewise, methods for determining the new configuration of the system after a failure, such as finding a suitable node on which to restart any failed process, are not considered. The related problems of providing fault tolerance for real-time systems and applications [Hecht76, Kopetz85a, Anderson83], reintegrating the system after a network partition [Birrell82, Walker83], and maintaining the consistency and availability of static data such as file systems and databases [Gray79, Haerder83, Lampson79, Svobodova84] are also not directly addressed by this thesis.

1.6 Thesis Outline

Chapter 2 presents the theoretical framework that will be used in the remainder of the thesis. This framework takes the form of a general model for reasoning about the behavior and correctness of recovery methods using message logging and checkpointing. The model is independent of the particular message logging protocol on which the system is based, and is used to prove a number of properties about these systems. An important result of this model is that, at any time in a system using message logging and checkpointing, there is always a *unique maximum* state of the system that can be recovered.

Next, Chapter 3 develops a new *pessimistic* message logging protocol, called *sender-based message logging*. This protocol is designed to minimize the overhead on the system caused by pessimistic message logging. This chapter examines the design of the sender-based message logging protocol, describes a complete implementation of it, and presents an evaluation of its performance in this implementation.

Chapter 4 develops a new *optimistic* message logging protocol and two alternative recovery algorithms that guarantee to always find the *maximum* possible recoverable system state. Although previous optimistic message logging systems find *some* recoverable system state, this state may be less than the maximum. Furthermore, this new system requires less communication for the provision of fault tolerance than do these previous methods. This chapter also describes an implementation of this new system and reports on its performance.

Finally, Chapter 5 summarizes the major contributions of this thesis, and suggests several avenues for future research.

Chapter 2

Theoretical Framework

This chapter develops a general model for reasoning about the behavior and correctness of fault-tolerance methods using message logging and checkpointing, which will be used throughout this thesis. The model does not assume the use of any particular message logging protocol, and can be applied to systems using either pessimistic or optimistic message logging methods. As part of this model, a number of important properties of message logging and checkpointing methods are proven.

2.1 The Model

This model is based on the notion of *dependency* between the states of processes that results from communication in the system. That is, when a process receives a message from some other process, the current state of the receiver then *depends on* the state of the sender at the time that the message was sent, since any part of the sender's state may have been included in the message. For example, if a process sends a series of messages, each containing the next number in some sequence known only to the sender, any process that receives one of these messages then depends on the state of the sender from which the message was sent because this state determines the position in the sequence. Since there is no single notion of "real time" in a distributed system, these dependencies, together with the linear ordering of events within each process, define a partial ordering on the events within the system [Lamport78]. This model describes the state of a process by its current dependencies, and describes the state of the system as a collection of process states.

2.1.1 Process States

The execution of each process is divided into discrete intervals by the messages that the process receives. Each interval, called a *state interval* of the process, is a *deterministic* sequence of execution, started by the receipt of the next message by the process. The execution of a process within a single state interval is completely determined by the state of the process at the time that the message is received and by the contents of the message. A process may *send* any number of messages to other processes during any state interval.

Within a process, each state interval of that process is identified by a unique *state interval index*, which counts the number of messages received by the process. Processes may be dynamically created and destroyed, but each process must be identified by a globally unique process identifier. Logically, these identifiers are assumed to be in the range 1 through n for a system of n processes. The creation of a process is modeled by its receipt of message number 0, and process termination is modeled by its receipt of one final message following the sequence of real messages received by the process. All messages *sent* by a process are tagged by its current state interval index.

When a process i receives a message sent by some process j, the state of process i then depends on the state that process j had at the time that the message was sent. The state of a process is determined by its dependencies on all other processes. For each process i, these dependencies are represented by a *dependency vector*

$$\langle \delta_* \rangle = \langle \delta_1,\, \delta_2,\, \delta_3,\, \ldots,\, \delta_n \rangle \ ,$$

where n is the total number of processes in the system. Component j of process i's dependency vector, δ_j, is set to the *maximum* index of any state interval of process j on which process i currently depends. If process i has no dependency on any state interval of process j, then δ_j is set to $\perp$, which is less than all possible state interval indices. Component i of process i's own dependency vector is always set to the index of process i's current state interval. The dependency vector of a process records only those state intervals on which the process *directly* depends, resulting from the receipt of a message sent from that state interval in the sending process. Only the maximum index of any state interval of each other process on which this process depends is recorded, since the execution of a process within each state interval is deterministic, and since this state interval naturally also depends on all previous intervals of the same process.

Processes cooperate to maintain their dependency vectors by tagging all messages sent with the current state interval index of the sending process, and by remembering in each process the maximum index tagging any message received from each other process. During any single execution of the system, the current dependency vector of any process is uniquely determined by the state interval index of that process. No component of the dependency vector of any process can decrease through failure-free execution of the system.

2.1.2 System States

A system state is a collection of process states, one for each process in the system. These process states need not all have existed in the system at the same time. A system state is said to have *occurred* during some execution of the system if all component process states have each individually occurred during this execution. A system state is represented by an $n \times n$ *dependency matrix*

$$\mathbf{D} = [\delta_{**}] = \begin{bmatrix} \delta_{11} & \delta_{12} & \delta_{13} & \dots & \delta_{1n} \\ \delta_{21} & \delta_{22} & \delta_{23} & \dots & \delta_{2n} \\ \delta_{31} & \delta_{32} & \delta_{33} & \dots & \delta_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \delta_{n1} & \delta_{n2} & \delta_{n3} & \dots & \delta_{nn} \end{bmatrix},$$

where row i, δ_{ij}, $1 \le j \le n$, is the dependency vector for the state of process i included in this system state. Since for all i, component i of process i's dependency vector is always the index of its current state interval, the diagonal of the dependency matrix, δ_{ii}, $1 \le i \le n$, is always set to the current state interval index of each process contained in the system state.

Let $\mathcal{S}$ be the set of all system states that have occurred during any *single* execution of some system. The *system history relation*, $\prec$, is a partial order on the set $\mathcal{S}$, such that one system state precedes another in this relation if and only if it *must* have occurred first during this execution. The relation $\prec$ can be expressed in terms of the state interval index of each process shown in the dependency matrices representing these system states.

Definition 2.1 If $\mathbf{A} = [\alpha_{**}]$ and $\mathbf{B} = [\beta_{**}]$ are system states in $\mathcal{S}$, then

$$\mathbf{A} \preceq \mathbf{B} \iff \forall\, i\, [\alpha_{i\,i} \leq \beta_{i\,i}] \ ,$$

and

$$\mathbf{A} \prec \mathbf{B} \iff (\mathbf{A} \preceq \mathbf{B}) \ \wedge \ (\mathbf{A} \neq \mathbf{B}) \ .$$

Although similar, the system history relation differs from Lamport's *happened before* relation [Lamport78] in that it orders the system states that result from events rather than the events themselves, and in that only process state intervals (started by the receipt of a message) constitute events.

To illustrate this partial order, Figure 2.1 shows a system of four communicating processes. The horizontal lines represent the execution of each process, with time progressing from left to right. Each arrow between processes represents a message sent from one process to another, and the number at each arrow gives the index of the state interval started by the receipt of that message. The last message received by process 1 is message a, and the last message received by process 4 is message b. Consider the two possible system states $\mathbf{A}$ and $\mathbf{B}$, such that in state $\mathbf{A}$, message a has been received but message b has not, and in state $\mathbf{B}$, message b has been received but message a has not. These two system states are represented by the dependency matrices

$$\mathbf{A} = \begin{bmatrix} 2 & 0 & \perp & \perp \\ \perp & 0 & \perp & \perp \\ \perp & 0 & 2 & 0 \\ \perp & \perp & \perp & 0 \end{bmatrix} \qquad \text{and} \qquad \mathbf{B} = \begin{bmatrix} 1 & 0 & \perp & \perp \\ \perp & 0 & \perp & \perp \\ \perp & 0 & 2 & 0 \\ \perp & \perp & 2 & 1 \end{bmatrix} \ .$$

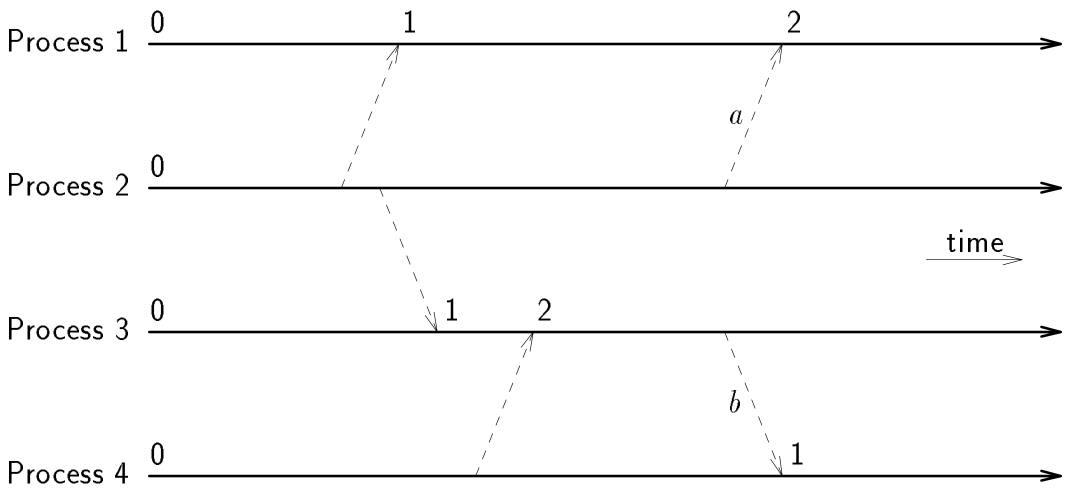

Figure 2.1 The system history partial order. Neither message a nor message b must have been received first.

System states **A** and **B** are incomparable under the system history relation. In the execution of the system, neither state **A** nor state **B** *must* have occurred first, because neither message a nor message b must have been received first. In terms of their dependency matrices, this is shown by a comparison of the circled values shown above on the diagonals of these two matrices.

2.1.3 The System History Lattice

A system state describes the set of messages that have been received by each process. For any two system states **A** and **B** in $\mathcal{S}$, the *meet* of **A** and **B**, written $\mathbf{A} \sqcap \mathbf{B}$, represents a system state that has also occurred during this execution of the system, in which each process has received only those messages that it has received in *both* **A** and **B**. This can be expressed in terms of the dependency matrices representing these two system states by copying each row from the corresponding row of one of the two original matrices, depending on which matrix has the *smaller* entry on its diagonal in that row.

Definition 2.2 If $\mathbf{A} = [\alpha_{**}]$ and $\mathbf{B} = [\beta_{**}]$ are system states in $\mathcal{S}$, the *meet* of $\mathbf{A}$ and $\mathbf{B}$ is $\mathbf{A} \sqcap \mathbf{B} = [\phi_{**}]$, such that

$$\forall\, i \left[\phi_{i\,*} = \begin{cases} \alpha_{i\,*} & \text{if } \alpha_{i\,i} \leq \beta_{i\,i} \\ \beta_{i\,*} & \text{otherwise} \end{cases} \right].$$

Likewise, for any two system states $\mathbf{A}$ and $\mathbf{B}$ in $\mathcal{S}$, the *join* of $\mathbf{A}$ and $\mathbf{B}$, written $\mathbf{A} \sqcup \mathbf{B}$, represents a system state that has also occurred during this execution of the system, in which each process has received all messages that it has received in *either* $\mathbf{A}$ or $\mathbf{B}$. This can be expressed in terms of the dependency matrices representing these two system states by copying each row from the corresponding row of one of the two original matrices, depending on which matrix has the *larger* entry on its diagonal in that row.

Definition 2.3 If $\mathbf{A} = [\alpha_{**}]$ and $\mathbf{B} = [\beta_{**}]$ are system states in $\mathcal{S}$, the *join* of $\mathbf{A}$ and $\mathbf{B}$ is $\mathbf{A} \sqcup \mathbf{B} = [\theta_{**}]$, such that

$$\forall\, i \left[\theta_{i\,*} = \begin{cases} \alpha_{i\,*} & \text{if } \alpha_{i\,i} \geq \beta_{i\,i} \\ \beta_{i\,*} & \text{otherwise} \end{cases} \right].$$

Continuing the example of Section 2.1.2 illustrated in Figure 2.1, the meet and join of states $\mathbf{A}$ and $\mathbf{B}$ are represented by the dependency matrices

$$\mathbf{A} \sqcap \mathbf{B} = \begin{bmatrix} 1 & 0 & \perp & \perp \\ \perp & 0 & \perp & \perp \\ \perp & 0 & 2 & 0 \\ \perp & \perp & \perp & 0 \end{bmatrix} \quad \text{and} \quad \mathbf{A} \sqcup \mathbf{B} = \begin{bmatrix} 2 & 0 & \perp & \perp \\ \perp & 0 & \perp & \perp \\ \perp & 0 & 2 & 0 \\ \perp & \perp & 2 & 1 \end{bmatrix}.$$

The following theorem introduces the *system history lattice* formed by the set of system states that have occurred during any *single* execution of some system, ordered by the system history relation.

Theorem 2.1 The set $\mathcal{S}$, ordered by the system history relation, forms a lattice. For any $\mathbf{A}, \mathbf{B} \in \mathcal{S}$, the *greatest lower bound* of $\mathbf{A}$ and $\mathbf{B}$ is $\mathbf{A} \sqcap \mathbf{B}$, and the *least upper bound* of $\mathbf{A}$ and $\mathbf{B}$ is $\mathbf{A} \sqcup \mathbf{B}$.

Proof Follows directly from the construction of system state meet and join, in Definitions 2.2 and 2.3. □

2.1.4 Consistent System States

Because the process states composing a system state need not all have existed at the same time, some system states may represent an impossible state of the system. A system state is called *consistent* if it *could* have been seen at some instant by an outside observer during the preceding execution of the system from its initial state, regardless of the relative speeds of the component processes [Chandy85]. After recovery from a failure, the system must be recovered to a consistent system state. This ensures that the total execution of the system is equivalent to *some* possible failure-free execution.

In this model, since all process communication is through messages, and since processes execute deterministically between received messages, a system state is consistent if no component process has received a message that has not been sent yet in this system state and that cannot be sent through the future deterministic execution of the sender. Since process execution is only deterministic within each state interval, this is true only if no process has received a message that will not be sent before the end of the sender's current state interval contained in this system state. Any messages shown by a system state to be *sent* but not yet *received* do not cause the system state to be inconsistent. These messages can be handled by the normal mechanism for reliable message delivery, if any, used by the underlying system. In particular, suppose such a message a was received by some process i after the state of process i was observed to form the system state $\mathbf{D}$. Then suppose process i sent some message b (such as an acknowledgement of message a), which could show the receipt of message a. If message b has been received in system state $\mathbf{D}$, then state $\mathbf{D}$ is inconsistent because message b (not message a) is shown to have been received but not yet sent. Instead, if message b has also not been received yet in state $\mathbf{D}$, then no effect of either message can be seen in $\mathbf{D}$, and $\mathbf{D}$ is therefore still consistent.

The definition of a consistent system state can be expressed in terms of the dependency matrices representing system states. If a system state is consistent, then for each process i, no other process j depends on a state interval of process i beyond process i's current state interval. In the dependency matrix, for each column i, no element in column i in any row j is larger than the element on the diagonal of the matrix in column i (and row i), which is process i's current state interval index.

Definition 2.4 If $\mathbf{D} = [\delta_{**}]$ is some system state in $\mathcal{S}$, $\mathbf{D}$ is *consistent* if and only if

$$\forall\, i, j \; [\delta_{j\,i} \leq \delta_{i\,i}] \; .$$

For example, consider the system of three processes whose execution is shown in Figure 2.2. The state of each process has been observed where the curve intersects the line representing the execution of that process, and the resulting system state is represented by the dependency matrix

$$\mathbf{D} = [\delta_{**}] = \begin{bmatrix} 1 & 4 & \perp \\ 0 & 2 & 0 \\ \perp & 2 & 1 \end{bmatrix} .$$

This system state is not consistent, since process 1 has received a message (to begin state interval 1) from process 2, which was sent beyond the end of process 2's current

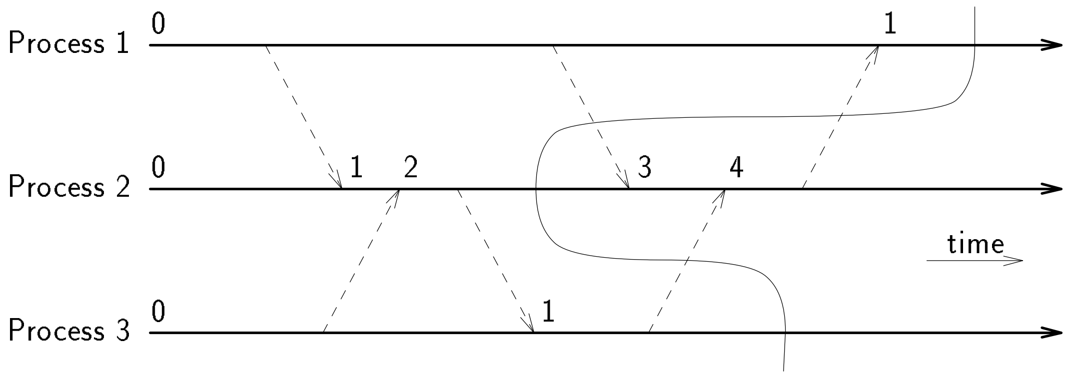

Figure 2.2 An inconsistent system state

state interval. This message has not been sent yet by process 2 and cannot be sent by process 2 through its future deterministic execution. In terms of the dependency matrix shown above, since δ_{12} is greater than δ_{22}, the system state represented by this matrix is not consistent.

Let the set $\mathcal{C} \subseteq \mathcal{S}$ be the set of *consistent* system states that have occurred during any single execution of some system. That is,

$$\mathcal{C} = \{\, \mathbf{D} \in \mathcal{S} \mid \mathbf{D} \text{ is consistent} \,\} \;.$$

Theorem 2.2 The set $\mathcal{C}$, ordered by the system history relation, forms a sublattice of the system history lattice.

Proof Let $\mathbf{A} = [\alpha_{**}]$ and $\mathbf{B} = [\beta_{**}]$ be system states in $\mathcal{C}$. By Definition 2.4, since $\mathbf{A} \in \mathcal{C}$ and $\mathbf{B} \in \mathcal{C}$, $\alpha_{j\,i} \leq \alpha_{i\,i}$ and $\beta_{j\,i} \leq \beta_{i\,i}$, for all i and j. It suffices to show that $\mathbf{A} \sqcap \mathbf{B} \in \mathcal{C}$ and $\mathbf{A} \sqcup \mathbf{B} \in \mathcal{C}$.

Let $\mathbf{A} \sqcap \mathbf{B} = [\phi_{**}]$. By Definition 2.2, and because $\mathbf{A}$ and $\mathbf{B}$ both occurred during the same execution of the system and no element in the dependency vector of any process can decrease through execution of the process, then $\phi_{j\,i} = \min(\alpha_{j\,i}, \beta_{j\,i})$, for all i and j. Thus, $\phi_{j\,i} \leq \alpha_{j\,i}$ and $\phi_{j\,i} \leq \beta_{j\,i}$, for all i and j. Since $\mathbf{A} \in \mathcal{C}$ and $\mathbf{B} \in \mathcal{C}$, $\phi_{j\,i} \leq \alpha_{j\,i} \leq \alpha_{i\,i}$ and $\phi_{j\,i} \leq \beta_{j\,i} \leq \beta_{i\,i}$. Thus, $\phi_{j\,i} \leq \min(\alpha_{i\,i}, \beta_{i\,i})$, and $\phi_{j\,i} \leq \phi_{i\,i}$, for all i and j. Therefore, $\mathbf{A} \sqcap \mathbf{B} \in \mathcal{C}$.

Let $\mathbf{A} \sqcup \mathbf{B} = [\theta_{**}]$. By Definition 2.3, either $\theta_{j\,i} = \alpha_{j\,i}$ or $\theta_{j\,i} = \beta_{j\,i}$, and $\theta_{i\,i} = \max(\alpha_{i\,i}, \beta_{i\,i})$, for all i and j. Since $\mathbf{A} \in \mathcal{C}$ and $\mathbf{B} \in \mathcal{C}$, $\theta_{j\,i} \leq \theta_{i\,i}$ for all i and j as well. Therefore, $\mathbf{A} \sqcup \mathbf{B} \in \mathcal{C}$. $\square$

2.1.5 Message Logging and Checkpointing

As the system executes, messages are recorded on stable storage in a message log. A message is called *logged* if and only if its data and the index of the state interval that it started in the process that received it are *both* recorded on stable storage. Logged messages remain on stable storage until no longer needed for recovery from any possible future failure of the system (Section 2.1.9). The predicate *logged*(i, σ) is true if and only if the message that started state interval σ of process i is logged.

When a process is created, its initial state is saved on stable storage as a checkpoint (in state interval 0). Each process is also independently checkpointed at times during its execution. Each checkpoint remains on stable storage until no longer needed for recovery from any possible future failure of the system (Section 2.1.9). For every state interval σ of each process, there must then be *some* checkpoint of that process on stable storage with a state interval index no larger than σ.

Definition 2.5 The *effective checkpoint* for a state interval σ of some process i is the checkpoint on stable storage for process i with the largest state interval index ϵ such that $\epsilon \leq \sigma$.

A state interval of a process is called *stable* if and only if it can be recreated from information currently on stable storage. This is true if and only if all received messages that started state intervals in the process after its state interval recorded in the effective checkpoint are logged. The predicate *stable*(i, σ) is true if and only if state interval σ of process i is stable.

Definition 2.6 State interval σ of process i is *stable* if and only if

$$\forall\, \alpha, \epsilon < \alpha \leq \sigma\, [\,\mathrm{logged}(i, \alpha)\,]\ ,$$

where ϵ is the index of the state interval of process i recorded in the effective checkpoint for state interval σ.

Any stable process state interval σ can be recreated by restoring the process from the effective checkpoint (with state interval index ϵ) and replaying to it the sequence of logged messages to begin state intervals $\epsilon+1$ through σ, in ascending order.

The checkpointing of a process need not be coordinated with the logging of messages received by that process. In particular, a process may be checkpointed at any time, and the state interval recorded in that checkpoint is then stable, regardless of whether or not all previous messages received by that process have been logged. Thus, if a state interval σ of some process i is stable and its effective checkpoint records its state interval ϵ, then all state intervals α of process i, $\epsilon \leq \alpha \leq \sigma$, must be stable, but some state intervals $\beta < \epsilon$ of process i may not be stable.

Each checkpoint of a process includes the complete current dependency vector of the process. Each logged message only contains the state interval index of the sending process at the time that the message was sent (tagging the message), but the complete dependency vector for any stable state interval of any process is always known, since all messages that started state intervals after the effective checkpoint must be logged.

2.1.6 Recoverable System States

A system state is called *recoverable* if and only if all component process state intervals are *stable* and the resulting system state is *consistent*. That is, to recover the state of the system, it must be possible to recreate the states of the component processes, and for this system state to be meaningful, it must be possible to have occurred through failure-free execution of the system from its initial state.

Definition 2.7 If $\mathbf{D} = [\delta_{**}]$ is some system state in $\mathcal{S}$, $\mathbf{D}$ is *recoverable* if and only if

$$\mathbf{D} \in \mathcal{C} \ \wedge \ \forall\, i\, [\, \mathrm{stable}(i, \delta_{ii})\,]\ .$$

Let the set $\mathcal{R} \subseteq \mathcal{S}$ be the set of *recoverable* system states that have occurred during any single execution of some system. That is,

$$\mathcal{R} = \{\, \mathbf{D} \in \mathcal{S} \mid \mathbf{D} \text{ is recoverable} \,\}\ .$$

Since only consistent system states can be recoverable, $\mathcal{R} \subseteq \mathcal{C} \subseteq \mathcal{S}$.

Theorem 2.3 The set $\mathcal{R}$, ordered by the system history relation, forms a sublattice of the system history lattice.

Proof For any $\mathbf{A}, \mathbf{B} \in \mathcal{R}$, $\mathbf{A} \sqcap \mathbf{B} \in \mathcal{C}$ and $\mathbf{A} \sqcup \mathbf{B} \in \mathcal{C}$, by Theorem 2.2. Since the state interval of each process in $\mathbf{A}$ and $\mathbf{B}$ is stable, all process state intervals in $\mathbf{A} \sqcap \mathbf{B}$ and $\mathbf{A} \sqcup \mathbf{B}$ are stable as well. Thus, $\mathbf{A} \sqcap \mathbf{B} \in \mathcal{R}$ and $\mathbf{A} \sqcup \mathbf{B} \in \mathcal{R}$, and $\mathcal{R}$ forms a sublattice. $\square$

2.1.7 The Current Recovery State

During recovery, the state of the system is restored to the "most recent" recoverable state that can be constructed from the available information, in order to minimize the amount of reexecution necessary to complete the recovery. The system history lattice corresponds to this notion of "most recent," and the following theorem establishes the existence of a *single* maximum recoverable system state under this ordering.

Theorem 2.4 There is always a unique maximum recoverable system state in $\mathcal{S}$.

Proof The unique maximum in $\mathcal{S}$ is simply

$$\bigsqcup_{\mathbf{D} \in \mathcal{R}} \mathbf{D} \,,$$

which must be unique since $\mathcal{R}$ forms a sublattice of the system history lattice. $\square$

Definition 2.8 At any time, the *current recovery state* of the system is the state to which the system will be restored if any failure occurs in the system at that time.

In this model, the *current recovery state* of the system is always the unique maximum system state that is currently recoverable.

Lemma 2.1 During any single execution of the system, the current recovery state never decreases.

Proof Let $\mathbf{R} = [\rho_{**}]$ be the current recovery state of the system at some time. Dependencies can only be added to state $\mathbf{R}$ by the receipt of a new message, which would cause the receiving process to begin a new state interval, resulting in a new system state. Thus, system state $\mathbf{R}$ itself must remain consistent. Since logged messages and checkpoints are not removed until no longer needed, state interval $\rho_{i\,i}$ for each process i must remain stable until no longer needed. Thus system state $\mathbf{R}$ itself must remain recoverable. Since the set $\mathcal{R}$ forms a lattice, any new current recovery state $\mathbf{R}'$ established after state $\mathbf{R}$ must be greater than $\mathbf{R}$. $\square$

The *domino effect* [Randell75, Russell80] is a well known problem that can occur in attempting to recover the state of a distributed system, and must be avoided to guarantee progress in the system in spite of failures. An example of how the domino effect can occur is illustrated in Figure 2.3. This figure shows the execution of a system of two communicating processes, with time progressing from left to right. The state of each process has been checkpointed at each time marked with a vertical bar, but no messages have been logged yet. If process 1 fails at the point shown, its state can only be recovered to its last checkpoint. Since this forces process 1 to roll back to a point before it sent its last message to process 2, process 2 is then forced to roll back to a point before this message was received. The previous checkpoint of process 2 forces process 1 to be rolled back, though, which in turn forces process 2

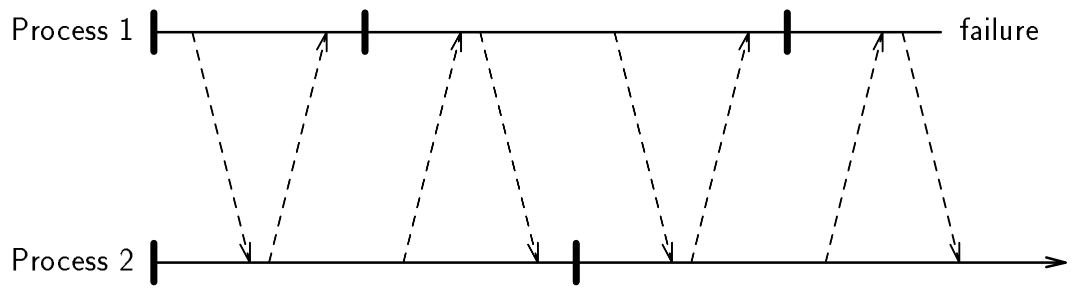

Figure 2.3 The domino effect. If process 1 fails, both processes ultimately must be rolled back to their initial states.

and then process 1 to be rolled back again. In this example, both processes are forced to roll back to their initial states, despite the presence of later checkpoints for each. This propagation of process rollbacks is known as the domino effect.

In this model, an occurrence of the domino effect would take the form of a propagation of dependencies that prevent the current recovery state from advancing. The following lemma establishes a sufficient condition for preventing the domino effect.

> **Lemma 2.2** If all messages received by each process in the system are *eventually* logged, the domino effect cannot occur.

Proof Let $\mathbf{R} = [\rho_{**}]$ be the current recovery state of the system at some time. For all state intervals σ of each process k, $\sigma > \rho_{kk}$, if all messages are eventually logged, state interval σ of process k will eventually become stable, by Definition 2.6. By Lemma 2.1, the current recovery state never decreases, and thus, by Definition 2.7, new system states $\mathbf{R}'$, $\mathbf{R} \prec \mathbf{R}'$, must eventually become recoverable and become the new current recovery state. The domino effect is thus avoided, since the current recovery state eventually increases. $\square$

2.1.8 The Outside World

During execution, processes may interact with the *outside world*, where an object is defined to be in the outside world if it does not participate in the message logging and checkpointing protocols of the system. Examples of interactions with the outside world include receiving input from a human user and writing information on the user's display terminal. All interactions with the outside world are modeled as messages either received from or sent to objects in the outside world.

Messages *received from* the outside world generally require no special treatment, and are simply logged as any other message. However, if such a message is assumed by the outside world to have been reliably transmitted, it must be *synchronously* recorded on stable storage as it enters the system, preventing the message from being lost if a failure occurs. Since the outside world knows only that the message entered the system, this ensures that the state of the outside world with respect to this message is consistent with the state of the system that can be restored during recovery.

Messages *sent to* the outside world may cause irreversible side effects, since the outside world generally cannot be rolled back. In order to guarantee that the state of the outside world is consistent with the state of the system that can be restored during recovery, any message sent to the outside world must be delayed until it is known that the state interval of the process from which it was sent will never be rolled back. The message may then be released from the system to the outside world. The following lemma establishes when it is safe to release a message sent to the outside world.

Lemma 2.3 Any message sent to the outside world by a process i in some state interval σ may be released from the system when $\sigma \leq \rho_{i\,i}$, where $\mathbf{R} = [\rho_{*\,*}]$ is now the current recovery state of the system.

Proof Follows directly from Lemma 2.1 and Definition 2.1. □

2.1.9 Garbage Collection

During operation of the system, checkpoints and logged messages must remain on stable storage until they are no longer needed for any possible future recovery of the system. The following two lemmas establish when they can safely be removed.

Lemma 2.4 Let $\mathbf{R} = [\rho_{*\,*}]$ be the current recovery state. For each process i, if ϵ_i is the state interval index of the effective checkpoint for state interval $\rho_{i\,i}$ of process i, then any checkpoint of process i with state interval index $\sigma < \epsilon_i$ cannot be needed for any future recovery of the system and may be removed from stable storage.

Proof Follows directly from Lemma 2.1 and Definitions 2.1 and 2.5. □

Lemma 2.5 Let $\mathbf{R} = [\rho_{**}]$ be the current recovery state. For each process i, if ϵ_i is the state interval index of the effective checkpoint for state interval $\rho_{i\,i}$ of process i, then any message that begins a state interval $\sigma \leq \epsilon_i$ in process i cannot be needed for any future recovery of the system and may be removed from stable storage.

Proof Follows directly from Lemma 2.1 and Definitions 2.1 and 2.5. □

2.2 Related Work

The dependency vectors used in this model are similar to those used by Strom and Yemini in their Optimistic Recovery system [Strom85]. However, Strom and Yemini's dependency vectors contain a complete *transitive closure* of all dependencies of the process, rather than just the *direct* dependencies from messages received by the process. Each message sent is tagged with the current dependency vector of the sender, and the dependency vector tagging a message received is merged with the current dependency vector of the receiving process. This distributes complete transitive dependency information to each process, but is expensive to implement since the dependency vector tagging each message is proportional in size to the total number of processes in the system.

The state interval indices used here are also similar to those used by Strom and Yemini, but Strom and Yemini tag each state interval index with an *incarnation number* indicating the number of times that the process has rolled back for recovery. This allows state interval indices in their system to remain unique, even though some state intervals may have been rolled back and repeated. Using the model presented in this chapter, some state interval indices are reused after a failure, and all processes must therefore be notified during recovery of this reuse in order to avoid confusion. This uniqueness is useful in Strom and Yemini's system because it is intended for direct implementation, but the model here is intended instead to allow reasoning about the behavior and correctness of these methods. Therefore, the incarnation number is not required in this model, but could be added in any implementation without affecting the properties proven based on the model.

The treatment of input and output with the outside world in this model is similar to the treatment used in Strom and Yemini's system, and to that proposed by Pausch in extending the transactional model to include outside world interactions [Pausch88]. For each process, Strom and Yemini define an *input boundary function*, which logs all process input as messages, and an *output boundary function*, which holds all process output until it is known that the state interval from which it was sent will never be rolled back. Strom and Yemini also point out that if it is known that all effects of some message sent to the outside world can be rolled back, then the output can be allowed immediately. Pausch's system saves all input in a *recoverable input queue*, which is essentially a message log of input received from the outside world. Output to the outside world is classified according to a number of properties such as whether it is *deferrable* (it can be delayed until the transaction commits), or is *compensable* (it can be rolled back by a complementary defined output operation). Allowing appropriate output to the outside world to occur without waiting for the current recovery state to advance could be added to the model of this chapter as well, but would do little to enhance its power for reasoning about these recovery methods. Also, adding this information to the model for each possible message would lose much of the concise expressiveness of the current model. However, this enhancement to output handling may be important in implementing systems such as these, in that it may greatly improve the efficiency of handling such special outside world messages that can be rolled back.

This same problem of delaying output occurs in other fault-tolerance methods. In systems using atomic actions (Section 1.4.3), and in more conventional database transaction systems [Bernstein87], the transaction or action cannot *commit* until all information needed to recover it has been recorded on stable storage. Thus, any side effects of the computation cannot be made permanent (and cannot be seen outside the system) until this occurs. Using shorter actions or transactions can alleviate this delay to some degree, but this increases the overhead caused by recording the recovery information on stable storage. In addition to output to a database or to the file system, Pausch [Pausch88] has extended transactions to support output to the outside world. In systems using global checkpointing without message logging (Section 1.4.6), a new global checkpoint must be created before any output can be committed. However, creating frequent global checkpoints for output commitment may substantially degrade the failure-free performance of the system.

This model also differs from that used by previous optimistic message logging systems in that it does not assume reliable delivery of messages on the network. Previous optimistic systems have used a model in which a separate channel connects each pair of processes, such that the channel does not lose or reorder messages. Thus, in their definitions of a consistent system state, Strom and Yemini [Strom85] require all messages sent to have been received, and Sistla and Welch [Sistla89] require the sequence of messages received on each channel to be a prefix of those sent on it. Since this model does not assume reliable delivery, it can be applied to distributed systems that do not guarantee reliable delivery, such as those based on an Ethernet network. If needed, reliable delivery can also be incorporated into this model simply by assuming an acknowledgement message immediately following each message receipt.

The basic assumption behind this model, that processes execute deterministically based only on their starting state and on the sequence of messages that they receive, has been called the *state machine approach* [Schneider87]. A *state machine* models a process as a set of state variables and a set of commands that operate on those variables. Each command executes deterministically, and atomically transforms the variables of the state machine to a set of new values. Commands may also output values from the state machine during execution. With systems using message logging and checkpointing, the messages received by a process are equivalent to a request specifying a command to the state machine modeling the process. This state machine approach is fundamental to recovery using message logging and checkpointing, and the model presented in this chapter is specifically designed to capture this assumption of deterministic execution. However, the model does not follow the state machine approach assumption of reliable communication and atomic execution between received messages. The increased generality of not requiring these properties leads to a more powerful model for reasoning about these systems.

A number of other recovery models have been developed, although none explicitly for systems using message logging and checkpointing. The *occurrence graph* model [Merlin78] is very general, but does not include deterministic process execution. It also does not allow any process state to become stable after the state has occurred, which is required to support optimistic message logging. Another limitation of the occurrence graph model and several other general models [Russell77, Russell80, Wood85, Chandy88] is that they do not specify how a process state becomes stable, and thus cannot be used to reason about the progress of message logging and checkpointing. Other more specialized models include those to support atomic transactions [Best81, Shrivastava82, Skeen83], real-time systems [Anderson83], and centralized systems providing recovery at multiple independent levels [Anderson78]. Many of these models also make assumptions about the underlying system, such as assuming that communication is reliable or that communication and execution are atomic, in order to simplify the properties of the model. However, these assumptions lessen the power of the model for reasoning about a broad range of real systems.

2.3 Summary

This chapter has presented a general theoretical model for reasoning about fault-tolerance methods using message logging and checkpointing. The model does not rely on the use of any particular message logging protocol, and can be applied to systems using either pessimistic or optimistic logging protocols. By using this model to reason about these types of fault-tolerance methods, properties of them that are independent of the message logging protocol used can be deduced and proven.

The model concisely captures the dependencies that exist within the system that result from communication between processes. The current dependencies of a process define its state, and are represented by a *dependency vector*. A collection of process states in the system, represented by a *dependency matrix*, define a system state. The process states that make up a system state need not all have existed at the same time. A system state is said to have *occurred* during some execution of the system if all component process states have each individually occurred during this execution. The *system history relation* defines a partial order on the system states that have occurred during any single execution, such that one system state precedes another if and only if it *must* have occurred first during the execution.

As part of this model, this chapter has proven some important properties of any system using message logging and checkpointing. First, the set of system states that have occurred during any single execution of a system, ordered by the system history relation, forms a lattice, called the *system history lattice*. The sets of consistent and recoverable system states that have occurred during this same execution form sublattices of the system history lattice. During execution, there is always a unique maximum recoverable system state, which never decreases. This state is the *current recovery state* of the system, and is always the least upper bound of all elements in this recoverable sublattice. Finally, if all messages received by processes in the system are eventually logged, the domino effect cannot occur.

Chapter 3

Pessimistic Message Logging

Pessimistic message logging protocols allow the system to be recovered simply after a failure, but previous pessimistic protocols have resorted to specialized hardware to reduce the overhead caused by the synchronization required in logging each message. This chapter presents the design of a new pessimistic message logging protocol, called *sender-based message logging*, that requires no specialized hardware and adds little additional overhead to the system. Sender-based message logging guarantees recovery from a single failure at a time in the system, and detects all cases when multiple concurrent failures prevent recovery of a consistent system state. Extensions are also presented to support recovery from multiple concurrent failures. Since sender-based message logging is a pessimistic message logging protocol, any failed process can be recovered without affecting the states of any processes that did not fail.

3.1 Overview and Motivation

Sender-based message logging differs from previous message logging protocols in that it logs messages in the local *volatile* memory on the node from which they are *sent*, as illustrated in Figure 3.1. Previous message logging protocols send an extra copy of each message elsewhere for logging, either to stable storage [Lampson79, Bernstein87] on disk or to some special backup process that can survive the failure of the receiver process. Instead, since both the sender and receiver of a message either get or already have a copy of the message, it is less expensive to save one of these copies in the local volatile memory to serve as a log. Since the purpose of the logging is to recover the receiver if it fails, a volatile copy at the receiver cannot be used as the log, but the

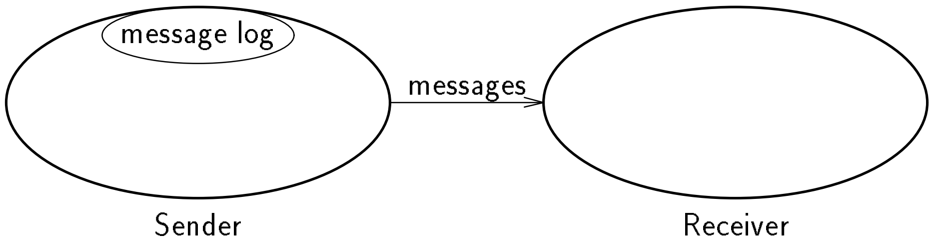

Figure 3.1 Sender-based message logging configuration

sender can easily save a copy of each message sent. It is this idea that forms the basis of sender-based message logging.

Since messages are logged in volatile memory, sender-based message logging can guarantee recovery from only a single failure at a time within the system. That is, after one process fails, no other process may fail until the recovery of the first is completed. If other processes fail during this time, some logged messages required for recovery from the first failure may be lost, and recovery of a consistent system state may not be possible with the available logged messages. Sender-based message logging detects this error if it occurs, allowing the system to notify the user or abort the computation if desired. Also, extensions to the basic sender-based message logging protocol to support recovery from multiple concurrent failures are discussed in Section 3.5.

3.2 Protocol Specification

3.2.1 Introduction

Each process participating in sender-based message logging maintains a *receive sequence number* (*RSN*), which counts messages *received* by the process. The receive sequence number of a process is always equal to the index of the current state interval of that process (Section 2.1.1). When a process receives a new message, it increments its RSN and returns this *new* value to the sender. The RSN indicates to the sender the order in which that message was received relative to other messages sent to the same process, possibly by other senders. This ordering information is not otherwise available to the sender, but is required for failure recovery because the logged messages must be replayed to the recovering process in the same order in which they were received before the failure. When the RSN arrives at the sender, it is saved in the local volatile log with the message.

The log of messages received by a process is distributed among the processes that sent these messages, such that each sender has in its local volatile log only those messages that it sent. Figure 3.2 shows an example of such a distributed message log resulting from sender-based message logging. In this example, process Y initially had an RSN value of 6. Process Y then received two messages from process X_1, followed by two messages from process X_2, and finally another message from X_1. For each message received, Y incremented its current RSN and returned the new RSN value to the sender. As each RSN arrived at the correct sender, it was added to that sender's local volatile log with the message. After receiving these five messages, the current RSN value of process Y is 11, and process Y is currently executing in state interval 11.

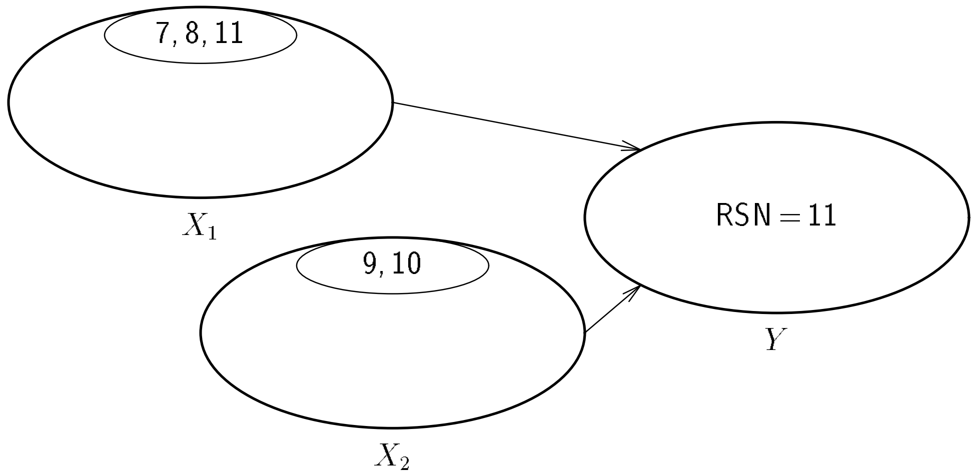

Figure 3.2 An example message log

In addition to returning the RSN to the sender when a message is received, each message *sent* by a process is also tagged with the current RSN of the sender. The RSN tagging a message identifies to the receiver the state interval of the sender from which the message was sent, and the receiver then depends on this state interval of the sender. Each process records these dependencies locally in a *dependency vector*, as discussed in Section 2.1.1. For each other process from which this process has received messages, the dependency vector records the *maximum* RSN value tagging a message received from that process. Sender-based message logging uses the dependency vector of each process during recovery to verify that the resulting system state is consistent.

3.2.2 Data Structures

Sender-based message logging requires the maintenance of the following new data structures for each participating process:

- A *receive sequence number* (*RSN*), numbering messages *received* by the process. This indicates to the sender of a message the order in which that message was received relative to other messages sent to the same process, possibly by other senders. The RSN is incremented each time a new message is received. The *new* value is assigned by the receiver as the RSN for the message and is returned to the sender. Each message sent by a process is tagged with the current RSN value of the sender. The RSN of a process thus represents the current *state interval index* of that process.

- A *message log* of messages *sent* by the process. For each message sent, this includes the message data, the identification of the destination process, the SSN (send sequence number) and RSN tagging the message (the current SSN and RSN of the sender when the message was sent), and the RSN returned by the receiver. After a process is checkpointed, all messages received by that process before the checkpoint can be removed from the logs of the processes that sent those messages.

- A *dependency vector*, recording the maximum index of any state interval of each process on which this process currently depends. For each other process from which this process has received messages, the dependency vector records the maximum RSN value tagging a message *received* from that process.

- An *RSN history list*, recording the RSN value returned for each message received by this process since its last checkpoint. For each message received, this includes the identification of the sending process, the SSN tagging the message, and the RSN returned when the message was received. This list is used when a duplicate message is received, and may be purged when the process is checkpointed.

Each of these data items except the RSN history list must be included in the checkpoint of the process. Also, the existing data structures used by a process for duplicate message detection (Section 1.2) must be included in the checkpoint. When a process is restarted from its checkpoint, the value of each of these data structures is restored along with the rest of the process state. The RSN history list need not be checkpointed, since messages received before this checkpoint will never be needed for any future recovery. For each process, only its most recent checkpoint and the log of messages received by it since this checkpoint must be saved.

3.2.3 Message Logging

Sender-based message logging operates with any existing message transmission protocol used by the underlying system. The following steps are required by sender-based message logging in sending a message m from some process X to some process Y:

1. Process X copies message m into its local volatile message log before transmitting m to process Y across the network. The message sent is tagged with the current RSN and SSN values of process X.

2. Process Y receives the message m, increments its own RSN value, and assigns this new value as the RSN for m. The entry for process X in process Y's dependency vector is set to the maximum of its current value and the RSN

tagging message m, and an entry in process Y's RSN history list is created to record that this new RSN has been assigned to message m. Finally, process Y returns to process X a packet containing the RSN value assigned to message m.

3. Process X adds the RSN for message m to its message log, and sends back to process Y a packet containing an acknowledgement for the RSN.

Process Y must periodically retransmit the RSN until its acknowledgement is received or until process X is determined to have failed. After returning the RSN, process Y may continue execution without waiting for the RSN acknowledgement, but it must not send any messages (including output to the outside world) until the RSNs returned for all messages that it has received have been acknowledged. The transmission of any messages sent by process Y in this interval must be delayed until these RSNs have been acknowledged. Process X does not experience any extra delay in execution, but does incur the overhead of copying the message and its RSN into the local message log. The operation of this protocol in the absence of retransmissions is illustrated in Figure 3.3.

This message logging is not an atomic operation, since the message data is entered into the log when it is sent, and the RSN can only be recorded after it is received from the destination process. Because the sender and the receiver execute on separate

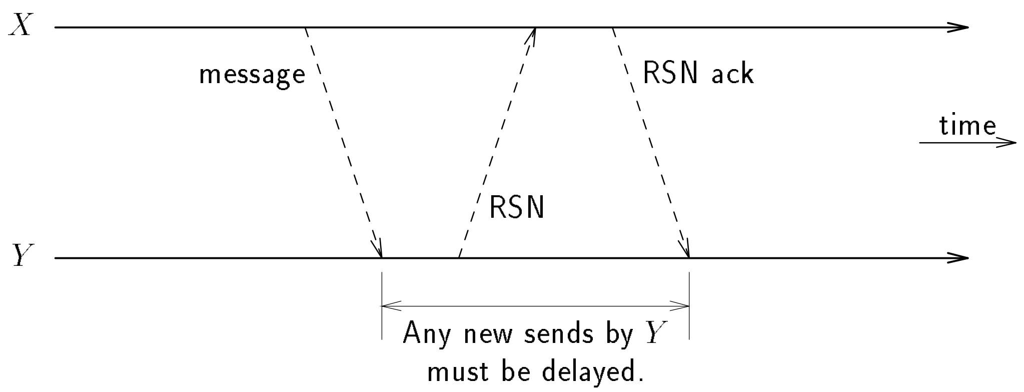

Figure 3.3 Operation of the message logging protocol in the absence of retransmissions

nodes in the distributed system, this logging cannot be completely synchronized with the receipt of the message. A message is called *partially logged* until the RSN has been added to the log by the sender. It is then called *fully logged*, or just *logged*.

Sender-based message logging is a *pessimistic* logging protocol, since it prevents the system from entering a state in which a failure could force any process other than those that failed to be rolled back during recovery. During recovery, a failed process can only be recovered up to its state interval that results from receiving the sequence of *fully logged* messages received by it before the failure. Until all RSNs returned by a process have been acknowledged, the process cannot know that its current state interval can be recovered if it should fail. By preventing each process from sending new messages in such a state, no process can receive a message sent from a state interval of another process that cannot be recovered. Therefore, no other process can be forced to roll back during recovery.

Depending on the protocol and network used by the underlying system, a process may receive duplicate messages during failure-free operation. Processes are assumed to detect any duplicate messages on receipt using the SSN tagging each message (Section 1.2). When a duplicate message is received, no new RSN is assigned to the message. Instead, the RSN assigned when the message was first received is used and is returned to the sender. When a process receives a duplicate message, it searches its RSN history list for an entry with the SSN tag and sending process identification of the message received. If the entry is found, the RSN recorded in that entry is used. Otherwise, the receiver must have been checkpointed since originally receiving this message, and the RSN history list entry for this message has been purged. The message then cannot be needed for any future recovery of this receiver, since the more recent checkpoint can always be used. In this case, the receiver instead returns to the sender an indication that this message need not be logged.

3.2.4 Failure Recovery

To recover a failed process, it is first reloaded from its most recent checkpoint on some available processor. This restores the process to the state it had when the checkpoint was written. Since the data structures used by sender-based message logging (Section 3.2.2) are included in the checkpoint, they are restored to the values they had when the checkpoint was written, as well.

Next, all *fully* logged messages that were received by the process after this checkpoint and before the failure are retrieved from the message logs of the sending processes, beginning with the first message following the current RSN value recorded in the checkpoint. These messages are replayed to the recovering process, and the process is allowed to begin execution. The recovering process is forced to receive these messages in ascending order of their logged RSNs, and is not allowed to receive any other messages until each message in this sequence of fully logged messages has been received. Since process execution is deterministic, the process then reaches the same state it had after it received these messages before the failure.

If only one process has failed, the state of the system after this reexecution must be consistent. Since the volatile message log at each sender survives the failure of the receiver, all fully logged messages received by the recovering process before the failure must be available. Since processes are restricted from sending new messages until all messages they have received are fully logged, the recovering process sent no messages before the failure after having received any message beyond the fully logged sequence that has been replayed. Thus, no other process can depend on a state interval of the recovering process beyond its state interval that has been restored using the available fully logged messages.

If more than one process has failed, though, some messages needed for recovery may not be available, and recovery of a consistent system state may not be possible. Each failed process can only be recovered up to its state interval that results from receiving the last available message in the sequence of fully logged messages. During recovery, sender-based message logging uses the dependency vector maintained by each process to verify that the system state that is recovered is consistent. For each failed process, if any other process has an entry in its dependency vector giving a state interval index of this process greater than the RSN of the last message in the available fully logged sequence, then the system cannot be recovered to a consistent state (Definition 2.4). In this case, the system may warn the user or abort the program if desired.

As the recovering process reexecutes from its checkpointed state through this sequence of fully logged messages, it resends all messages that it sent before the failure after this checkpoint was written. Since the current SSN of a process is included in its checkpoint, the SSNs used by this process during recovery are the same as those used when these messages were originally sent before the failure. Such duplicate messages are detected and handled by the underlying system using the SSN tagging each message, by the same method as used during failure-free operation (Section 1.2). For each duplicate message received, either the original RSN or an indication that the message need not be logged is returned to the recovering process.

After this sequence of fully logged messages has been received, any *partially* logged messages destined for the recovering process are resent to it. Also, any new messages that other processes may need to send to it may be sent at this time. These partially logged and new messages may be received by the recovering process in any order after the sequence of fully logged messages has been received. Again, since processes are restricted from sending new messages until all messages they have received are fully logged, no other process depends on any state interval of the recovering process that results from its receipt of any of these messages, and no other process has an entry in its dependency vector naming such a state interval of the recovering process. Since the receipt of these partially logged and new messages does not create any new dependencies in other processes, the state of the system remains consistent regardless of the order of their receipt now, by Definition 2.4.

The system data structures necessary for further participation in sender-based message logging are correctly restored, including those necessary for the recovery of any other processes that may fail in the future. These data structures are read from the checkpoint and are modified as a result of sending and receiving the same sequence of messages as before the failure. In particular, the volatile log of messages sent by the failed process is recreated as each duplicate message is sent during reexecution.

As discussed in Section 2.1.7, in order to guarantee progress in the system in spite of failures, any fault-tolerance method must avoid the *domino effect*, an uncontrolled propagation of process rollbacks necessary to restore the system to a consistent state following a failure. As with any pessimistic message logging protocol, sender-based message logging avoids the domino effect by guaranteeing that any failed process can be recovered from its most recent checkpoint, and that no other process must be rolled back during recovery.

3.2.5 Protocol Optimizations

The protocol described above contains the basic steps necessary for the correct operation of sender-based message logging. However, two optimizations to this basic protocol help to reduce the number of packets transmitted. These optimizations combine more information into each packet than would normally be present. They do not alter the logical operation of the protocol as described above, and their inclusion in an implementation of the protocol is optional.

The first of these optimizations is to encode more than one RSN or RSN acknowledgement in a single packet. This optimization is effective when an uninterrupted stream of packets is received from a single sender. For example, when receiving a *blast* bulk data transfer [Zwaenepoel85], the RSNs for all data packets of the blast can be returned to the sender in a single packet. This optimization is limited by the distribution of RSNs and RSN acknowledgements that can be encoded in a single packet, but encoding one contiguous range of each handles the most common case, as in this example.

The second optimization to the basic sender-based message logging protocol is to *piggyback* RSNs and RSN acknowledgements onto existing message packets, rather than transmitting them in additional special packets. For example, RSNs can be piggybacked on existing acknowledgement message packets used by the underlying system for reliable message delivery. Alternatively, if a message is received that requests the application program to produce some user-level reply to the original sender, the RSN for the request message can be piggybacked on the same packet that carries this reply. If the sending application program sends a new request to the same process shortly after the reply to the first request is received, the acknowledgement of this RSN, and the RSN for the reply itself, can be piggybacked on the same packet with this new request. As long as messages are exchanged between the same two processes in this way, no new packets are necessary to return RSNs or their acknowledgements. When this message sequence terminates, only two additional packets are required, one to return the RSN for the last reply message and one to return its RSN acknowledgement. The use of this piggybacking optimization for a sequence of these request-reply exchanges is illustrated in Figure 3.4. This optimization is particularly useful in systems using a remote procedure call protocol [Birrell84] or other request-response protocol [Cheriton88, Cheriton86a], since all communication takes place as a sequence of message exchanges.

The piggybacking optimization can be used in sending a new message only when all unacknowledged RSNs for messages received by the sending process are destined for the same process as the new message packet being sent, and can be included in that packet. When such a packet is received, the piggybacked RSNs and RSN acknowledgements must be handled before the message carried by the packet. When these RSNs are entered in the message log, the messages for which they were returned become fully logged. Since this packet carries all unacknowledged RSNs from the sender, all messages received by that sender become fully logged before this new

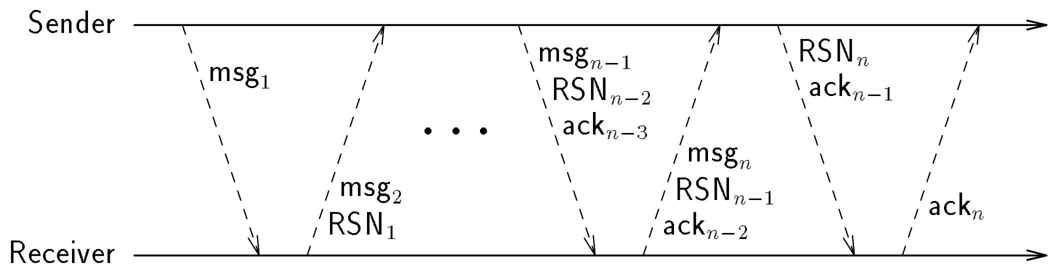

Figure 3.4 Piggybacking RSNs and RSN acknowledgements on existing message packets

message is seen by its destination process. If the RSNs are not received because the packet is lost in delivery on the network, the new message cannot be received either. The correctness of the protocol is thus preserved by this optimization because all messages received earlier by the sender are guaranteed to be fully logged before the new message in the packet is seen.

When using the piggybacking optimization, the transmission of RSNs and RSN acknowledgements is postponed until a packet is returned on which to piggyback them, or until a timer expires forcing their transmission if no return packet is forthcoming. However, this postponement may cause a delay in the transmission of new messages that would not be delayed without the use of this optimization. If a process postpones the return of an RSN for piggybacking, the transmission of a new message by that process may be delayed. If the new message is not destined for the same process as is the RSN, the message must be held while the RSN is sent and its acknowledgement is returned, delaying the transmission of the new message by approximately one packet round-trip time. Likewise, if a process postpones the return of the RSN acknowledgement for a message that it sent, new messages being sent by the original receiver process must be delayed since its earlier RSN has not yet been acknowledged. In this case, the process retransmits its original RSN to force the RSN acknowledgement to be returned, also delaying the transmission of the new message by approximately one packet round-trip time. In both cases, any possible delay is also bounded in general by the timer interval used to force the transmission of the RSN or its acknowledgement.

These two protocol optimizations can be combined. Continuing the blast protocol example above, the RSN for every data packet of the blast can be encoded together and piggybacked on the reply packet acknowledging the receipt of the blast. If there are n packets of the blast, the unoptimized sender-based message logging protocol requires an additional $2n$ packets to exchange their RSNs and RSN acknowledgements. Instead, if both protocol optimizations are combined, only two additional packets are required, one to return the RSN for the packet acknowledging the blast and one to return its RSN acknowledgement. This example using both protocol optimizations is illustrated in Figure 3.5.

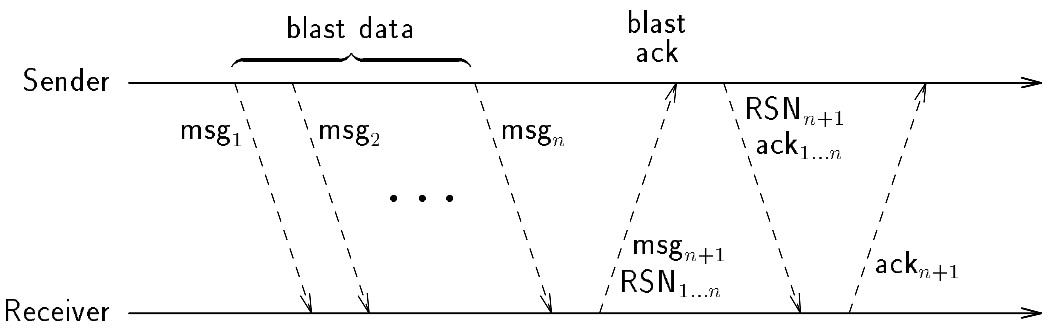

Figure 3.5 A blast protocol with sender-based message logging using both optimizations

3.3 Implementation

A full implementation of sender-based message logging has been completed under the V-System (Section 1.3). The system runs on a collection of diskless SUN workstations connected by an Ethernet network to a shared network file server. The implementation supports the full sender-based message logging protocol specified in Section 3.2, including both protocol optimizations, and supports all V-System message passing operations. As discussed in Section 1.3, all processes on the same logical host in the V-System must be located at the same physical network address, and thus this implementation treats each logical host as a single process in terms of the protocol specification. In the current implementation, only a single logical host per network node can use sender-based message logging.

3.3.1 Division of Labor

The implementation consists of a *logging server* process and a *checkpoint server* process running on each node in the system, and a small collection of support routines in the V-System kernel. The kernel records messages in the log in memory as they are sent, and handles the exchange of RSNs and RSN acknowledgements. This information is carried in normal V kernel packets, and is handled directly by the sending and receiving kernels. This reduces the overhead involved in these exchanges, eliminating any process scheduling delays. All other aspects of logging messages and retrieving logged messages during recovery are handled by the logging server process. The checkpoint server process manages the recording of new checkpoints and the reloading of processes from their checkpoints during recovery. All logging servers in the system belong to a V-System process group [Cheriton85], and all checkpoint servers belong to a separate process group.

This use of server processes limits the increase in size and complexity of the kernel. In total, only five new primitives to support message logging and three new primitives to support checkpointing were added to the kernel. Also some changes were made to the internal operation of several existing primitives. Table 3.1 summarizes the amount of executable instructions and data added to the kernel to support sender-based message logging and checkpointing for the SUN-3/60 V kernel configuration. The percentages given are relative to the size of that portion of the base kernel without checkpointing or message logging.

3.3.2 Message Log Format

On each node, a single message log is used to store all messages sent by any process executing on that node. The log is organized as a list of fixed-size blocks of message

Table 3.1

Size of kernel additions to support sender-based message logging and checkpointing

	Message Logging		Checkpointing		Total	
	Kbytes	Percent	Kbytes	Percent	Kbytes	Percent
Instructions	12.0	15.1	3.5	4.5	15.5	19.5
Data	36.0	18.5	0.0	0.0	36.0	18.5
Total	48.0	17.5	3.5	1.3	51.5	18.8

logging data that are sequentially filled as needed by the kernel, and are written to disk by the logging server during a checkpoint. Each packet sent by the kernel is treated as a message and is logged separately. The message log is stored in the volatile address space of the local logging server process. This allows much of the message log management to be performed by the server without additional kernel support. Since only one *user* address space at a time can be accessible, the message log block currently being filled is always double-mapped through the hardware page tables into the kernel address space. This allows new records to be added to the log without switching accessible address spaces, although some address space switches are still necessary to access records in earlier blocks of the log.

Each message log block is 8 kilobytes long, the same size as data blocks in the file system and hardware memory pages. Each block begins with a 20-byte header, which describes the extent of the space used within the block. The following two records types are used to describe the logging data in these message log blocks:

`LoggedMessage`: This type of record saves the data of the message (a copy of the network packet sent), the logical host identifier of the receiver, and the RSN value returned by the receiver. It varies in size from 92 to 1116 bytes, depending on the size of any appended data segment that is part of the message.

`AdditionalRsn`: This type of record saves an additional RSN returned for a message logged in an earlier `LoggedMessage` record. It contains the SSN of the message sent, the logical host identifier of the receiver, and the new RSN value returned. It is 12 bytes long.

For most messages sent, only the `LoggedMessage` record type is used. However, a message sent to a process group [Cheriton85] is delivered reliably to only one receiver, with delivery to other members of the group not guaranteed. Thus, the first RSN returned for a message sent to a process group is stored in the `LoggedMessage` record, and a new `AdditionalRsn` record is created to store the RSN returned by each other receiver of the same message. Likewise, since reliable delivery of a message sent as a datagram is not guaranteed, the kernel cannot save the `LoggedMessage` record until the RSN is returned. Instead, the RSN field in the `LoggedMessage` record is not used, and an `AdditionalRsn` record is created to hold the RSN when it arrives, if the message is received.

3.3.3 Packet Format

The network packet format used by sender-based message logging is a modified form of the standard V kernel packet format [Zwaenepoel84]. Fields have been added to each packet to carry RSNs and RSN acknowledgements. Since it must be possible to piggyback these fields on any existing packet, no special control packet type is used. Instead, if the logging protocol requires a packet to be sent when no existing packet is present on which to piggyback, an extra V kernel packet of type "no operation" is sent.

In particular, the following new fields have been added to the V kernel packet format to support sender-based message logging:

`rsn`: This field gives the receive sequence number of the sender at the time that the packet was sent, and is used to return an RSN for a received message.

`ssn2`: This field gives the SSN value of the last message whose RSN is being returned by this packet.

`rsn2`: This field gives the RSN value of the last RSN being acknowledged by this packet.

`rsnCount`: This field gives the number of RSNs being returned by this packet. If this field is zero, no RSNs are contained in this packet. Otherwise, RSNs from `rsn`−`rsnCount`+1 through `rsn`, inclusive, are being returned for received messages whose SSNs were `ssn2`−`rsnCount`+1 through `ssn2`, respectively. The size of this field is one byte, and thus a maximum of 255 RSNs can be returned in a single packet.

`rsnAckCount`: This field gives the number of RSN acknowledgements being returned by this packet. If this field is zero, no RSN acknowledgements are contained in this packet. Otherwise, RSNs from `rsn2` $-$ `rsnAckCount` $+1$ through `rsn2`, inclusive, are being acknowledged by this packet. The size of this field is one byte, and thus a maximum of 255 RSN acknowledgements can be returned in a single packet.

Also, the following additional flag bits were defined in an existing bit field in the kernel packet:

`NEED_RSN`: When this bit is set in a packet, it indicates that the receiver should return an RSN for the message contained in this packet.

`NEED_RSNACK`: When this bit is set in a packet, it indicates that an acknowledgement of the RSNs carried by this packet should be returned after this packet is received.

`DONT_LOG`: When this bit is set in a packet, it indicates that the range of messages whose SSNs are shown by `ssn2` and `rsnCount` in this packet should not be logged. This may be used when a duplicate message is received (Section 3.2.3).

These added packet fields and flag bits are contained in the V kernel packet header, but are ignored by the rest of the kernel.

3.3.4 Kernel Message Logging

Within the kernel, a layered implementation of the logging protocol is used, as illustrated in Figure 3.6. The sender-based message logging module acts as a filter on all packets sent and received by the kernel. Packets may be modified or held by this module before transmission, and received packets are interpreted before passing them on to the rest of the kernel. The messages logged by the implementation are actually the contents of each network packet passing to the Ethernet device driver through the sender-based message logging module. This organization separates the sender-based message logging protocol processing from the rest of the kernel, largely insulating it from changes in the kernel or its communication protocol. Integrating

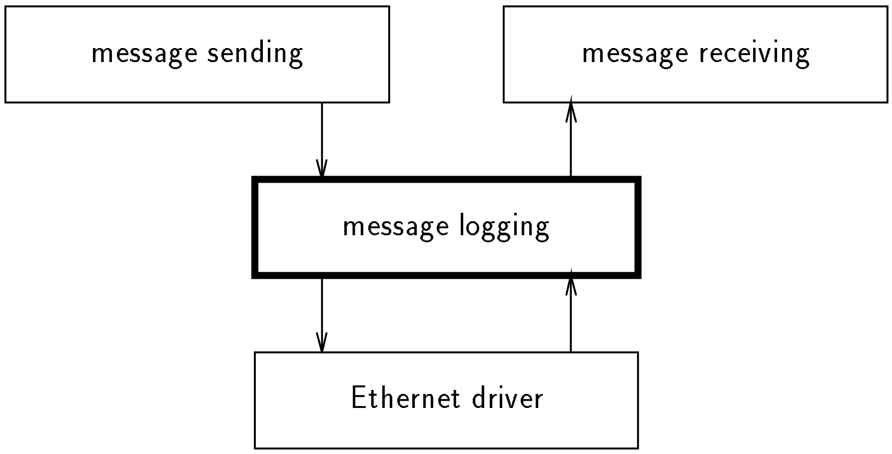

Figure 3.6 Sender-based message logging kernel organization

the kernel message logging with the existing V kernel protocol handling functions was also considered instead, but was determined not to be necessary for efficiency.

The standard V kernel uses retransmissions for reliable delivery of messages, but these retransmissions should not appear as separate messages for message logging. Also, it is important that retransmissions continue during failure recovery, rather than possibly timing out after some maximum number of attempts. Finally, with multiple processes sharing a single team address space, the appended data segment of a message could be changed between retransmissions, which could interfere with successful failure recovery. For these reasons, sender-based message logging performs all necessary retransmissions from the saved copy of the message in the `LoggedMessage` record in the log.

Most RSNs and RSN acknowledgements are piggybacked on existing packets. The piggybacking implementation makes few assumptions about the behavior of the existing V kernel protocol. When a message arrives, it is passed up to the normal V kernel protocol handling function for its packet type. If the kernel sends a reply packet (such as the data requested in a `MoveFrom`), the RSN for the request is piggybacked on this reply. Normally, if no reply is generated by the kernel, the RSN is saved, and a timer is set to force its transmission if no user-level message is returned soon. However, for certain packets for which there is not expected to be a new user-level message following, the RSN is transmitted immediately in a separate packet. An alternative implementation of piggybacking was also considered, which would use knowledge of the V kernel protocol in order to more efficiently handle piggybacking by anticipating future packets in standard packet sequences used by the kernel. However, accommodating the full generality of possible packet sequences is difficult, particularly when communicating with multiple remote logical hosts.

3.3.5 The Logging Server

At times, the logging server writes modified blocks of the message log from volatile memory to a file on the network file server. A separate file is used by each logging server. When the kernel adds a new record to some message log block, or modifies an existing record in some block, it flags that message log block in memory as having been modified. These flags are used by the logging server to control the writing of message log blocks from memory to the logging file. The writing of modified blocks is further simplified since blocks in the message log are the same size as blocks in the file system.

The logging file is updated during a checkpoint of the logical host so that the log of messages sent by this host before the checkpoint can be restored when the checkpoint is restored. Restoring this log is necessary in order to be able to recover from additional failures of other processes after recovery of this logical host is completed. As such, the file serves as part of the host's checkpoint.

The logging file is also updated at other times to reclaim space in the volatile memory of the log. The logging file may be larger than the space in volatile memory allocated to the message log. When a block has been written from volatile memory to the file, that space in memory may be reused if needed for other logging data if the kernel is no longer retransmitting messages from that block or adding new records to it. When the contents of such a block is again needed in memory, it is read from the file into an unused block in the log in memory. The logging file thus serves as a form of "virtual memory" extension to the message log, and allows more messages to be logged than can fit in the available volatile memory of the node. The kernel automatically requests modified blocks to be written to the file when the amount of memory used by the log approaches the allocated limit.

The message log contains all messages *sent* by processes executing on that node, but for most accesses to the log, one or more messages *received* by some specified logical host are needed. For example, when a logical host is checkpointed, all messages received by processes executing in that logical host are found and deleted from the logs at their senders. To facilitate such accesses to the log by receiver, the logging server maintains indices to the messages in the log, hashed by the identifier of the receiver. These indices are updated each time new blocks of the volatile log are written to the logging file, since these are exactly those blocks of the volatile log that have records describing new messages to be indexed.

These hashed indices are rebuilt by the logging server during recovery from the data in the logging file. The indices are rebuilt rather than being saved as part of the state data in the checkpoint since failure recovery should be relatively less frequent than checkpointing. During recovery, the logging server reads each block of the logging file for the recovering logical host, examining each record for messages sent before the checkpoint was recorded. Since the logging file may be updated at times other than during the checkpoint, it may contain messages sent after the checkpoint. These messages in the file are ignored during recovery since they will be sent again and logged again once the recovering host begins reexecution.

During recovery, the logging server on the node on which recovery is taking place coordinates the replay of logged messages to the recovering logical host, collecting each message from the log on the node from which it was sent. This is done by requesting each logged message from the process group of logging servers, in ascending order by the message RSNs. For each message, a request is sent to the logging server process group, naming the RSN of the next message needed. The server that has this message logged returns a copy of it, and replies with the RSN of the next message for the same logical host that it also has logged. When that RSN is needed next while collecting the logged messages, the request is sent directly to that server rather than to the process group. All servers that do not have the named message logged ignore the request. The sequence of logged messages is complete when no reply is received from a request sent to the process group after the kernel has retransmitted the request several times. As the logical host then reexecutes from its checkpointed state, the kernel simulates the arrival from the network of each message in the sequence of messages collected.

3.3.6 The Checkpoint Server

Checkpointing is initiated by sending a request to the local checkpoint server process. This request may be sent by the kernel when the logical host has received a given number of messages or has consumed a given amount of processor time since its last checkpoint. Any process may also request a checkpoint at any time, but this is never necessary. Likewise, failure recovery is initiated by sending a request to the checkpoint server on the node on which the failed logical host is to be recovered. Normally, this request would be sent by the process that detected the failure. However, no failure detection is currently implemented in this system, and the request instead comes from the user.

Each checkpoint is written as a separate file on the shared network file server. On the first checkpoint of a logical host, a *full checkpoint* is used to write the entire user address space to the checkpoint file in a single operation. On subsequent checkpoints of that logical host, an *incremental checkpoint* is used to write only the pages of the user address space modified since the previous checkpoint to the file, overwriting their previous values in the file. The checkpoint file thus always contains a complete continuous image of the user address space. Each checkpoint also includes all kernel data used by the logical host, the state of the local *team server* for that logical host, and the state of the local logging server. This data is entirely rewritten on each checkpoint, since it is small and modified portions of it are difficult to detect. Since the file server supports atomic commit of modified versions of files, the most recent complete checkpoint of a logical host is always available, even if a failure occurs while a checkpoint is being written. To limit any interference with the normal execution of the logical host during checkpointing, the bulk of the checkpoint data is written to the file while the logical host continues to execute. The logical host is then *frozen* while the remainder of the data is written to the file. This is similar to the technique used by Theimer for process migration in the V-System [Theimer85]. After the checkpoint has been completed, the group of logging servers is notified to remove from their logs all messages received by this host before the checkpoint. Because this notification is sent as a message to the process group, reliable delivery to all logging servers is not assured by the kernel, but the notification from any future checkpoint of this same host is sufficient.

A failed logical host may be restarted on any node in the network. Other processes sending messages to the recovered logical host determine its new physical network address using the existing V-System mechanism. The kernel maintains a cache recording the network address for each logical host from which it has received packets. In sending packets, after a small number of retransmissions to the cached network address, future retransmissions use a dedicated Ethernet multicast address to which all V-System kernels respond. All processes are restored with the same process identifiers as before the failure.

3.4 Performance

The performance of this implementation of sender-based message logging has been measured on a network of diskless SUN-3/60 workstations. These workstations each use a 20-megahertz Motorola MC68020 processor, and are connected by a 10 megabit per second Ethernet network to a single shared network file server. The file server runs on a SUN-3/160 using a 16-megahertz MC68020 processor, with a Fujitsu Eagle disk. This section presents an analysis of the costs involved with sender-based message logging in communication, checkpointing, and recovery, and an evaluation of the performance of several distributed application programs using sender-based message logging.

3.4.1 Communication Costs

Table 3.2 presents the average time in milliseconds required for common V-System communication operations. The elapsed times required for a `Send`-`Receive`-`Reply` sequence with no appended data and with a 1-kilobyte appended data segment, for a `Send` as a datagram, and for `MoveTo` and `MoveFrom` operations of 1 and 64 kilobytes of data each were measured. These operations were executed both with and without using sender-based message logging, and the average time required for each case is shown separately. The overhead of using sender-based message logging for each operation is given as the difference between these two times, and as a percentage increase over the time without logging. These times are measured in the initiating user process, and indicate the elapsed time between invoking the operation and its completion. The overhead for most communication operations is about 25 percent.

The measured overhead reported in Table 3.2 is caused entirely by the time necessary to execute the instructions of the sender-based message logging protocol implementation. Because of the request-response nature of the V-System communication operations, and due to the presence of the logging protocol optimizations described

Table 3.2

Performance of common V-System communication operations
using sender-based message logging (milliseconds)

Operation	Message Logging		Overhead	
	With	Without	Time	Percent
`Send`-`Receive`-`Reply`	1.9	1.4	.5	36
`Send`(1K)-`Receive`-`Reply`	3.4	2.7	.7	26
Datagram `Send`	.5	.4	.1	25
`MoveTo`(1K)	3.5	2.8	.7	25
`MoveTo`(64K)	107.0	88.0	19.0	22
`MoveFrom`(1K)	3.4	2.7	.7	26
`MoveFrom`(64K)	106.0	87.0	19.0	22

in Section 3.2.5, no extra packets were required during each operation, and no delays in the transmission of any message were incurred while waiting for an RSN acknowledgement to arrive. Although, two extra packets were required after all iterations of each test sequence in order to exchange the final RSN and RSN acknowledgement, this final exchange occurred asynchronously within the kernel after the user process had completed the timing.

To better understand how the execution time is spent during these operations, the execution times for a number of components of the implementation were measured individually by executing each component in a loop a large number of times and averaging the results. The time for a single execution could not be measured directly because the SUN lacks a hardware clock of sufficient resolution. Packet transmission overhead as a result of sender-based message logging is approximately 126 microseconds for messages of minimum size, including 27 microseconds to copy the message into the log. For sending a message with a 1-kilobyte appended data segment, this time increases by 151 microseconds for the additional time needed to copy the segment into the log. Of this transmission time, 38 microseconds occurs after the packet is transmitted on the Ethernet, and executes concurrently with reception on the remote node. Packet reception overhead from sender-based message logging is approximately 142 microseconds. Of this time, about 39 microseconds is spent processing any piggybacked RSNs, and about 45 microseconds is spent processing any RSN acknowledgements.

These component measurements agree well with the overhead times shown in Table 3.2 for each operation. For example, for a `Send-Receive-Reply` with no appended data segment, one minimum-sized message is sent by each process. The sending protocol executes concurrently with the receiving protocol for each packet after its transmission on the network. The total sender-based message logging overhead for this operation is calculated as

$$2\left((126 - 38) + 142\right) = 460 \text{ microseconds.}$$

This closely matches the measured overhead value of 500 microseconds given in Table 3.2. The time beyond this required to execute the logging protocol for a

1-kilobyte appended data segment `Send-Receive-Reply` is only the additional 151 microseconds needed to copy the segment into the message log. This closely matches the measured difference for these two operations of 200 microseconds. As a final example, consider the 64-kilobyte `MoveTo` operation, in which 64 messages with 1 kilobyte of appended data each are sent, followed by a reply message of minimum size. No concurrency is possible in sending the first 63 data messages, but they are each received concurrently with the following send. After the sender transmits the last data message, and again after the receiver transmits the reply message, execution of the protocol proceeds concurrently between the sending and receiving nodes. The total overhead for this operation in microseconds is calculated as

$$\begin{array}{lrl}
 & 63\,(126+151) & \text{for sending the first 63 packets} \\
 & (126-38)+151 & \text{for sending the last packet} \\
 & 142 & \text{for receiving the last packet} \\
 & (126-38) & \text{for sending the reply packet} \\
+ & 142 & \text{for receiving the reply packet} \\
\hline
 & 18,062\ . &
\end{array}$$

This total of 18,062 microseconds agrees well with the measured overhead of 19 milliseconds given in Table 3.2.

In less controlled environments and with more than two processes communicating, the communication performance may degrade because the transmission of some messages may be delayed waiting for an RSN acknowledgement to arrive. To evaluate the effect of this delay on the communication overhead, the average round-trip time required to send an RSN and receive its acknowledgement was measured. Without transmission errors, the communication delay should not exceed this round-trip time, but may be less if the RSN has already been sent when the new message transmission is first attempted. The average round-trip time required in this environment is 550 microseconds. Although the same amount of data is transmitted across the network for a `Send-Receive-Reply` with no appended data segment, this RSN round-trip time is significantly less than the 1.4 milliseconds shown in Table 3.2 for the

`Send`-`Receive`-`Reply`, because the RSN exchange takes place directly between the two kernels rather than between two processes at user level.

The effect of the two protocol optimizations on the performance of these communication operations was measured by using a sender-based message logging implementation that did not include either optimization, to execute the same tests reported in Table 3.2. All RSNs and RSN acknowledgements were sent as soon as possible without piggybacking, and no packet carried more than one RSN or RSN acknowledgement. For most operations, the elapsed time increased by an average of 430 microseconds per message involved in the operation, over the time required for logging with the optimized protocol. Comparing this increase to the measured RSN round-trip time of 550 microseconds indicates that about 120 microseconds of the round-trip time occurs concurrently with other execution. This includes the time needed by the V kernel protocol functions and the time for the user process to receive the message for which this RSN is being returned and to form the reply message. The times for the 64-kilobyte `MoveTo` and `MoveFrom` operations, and for the datagram `Send`, however, increased by an average of only 260 microseconds per message. This is less than the increase for other operations because multiple sequential messages are sent to the same destination, with no intervening reply messages, and thus many messages sent are not forced to wait for an RSN round-trip. The increase is still substantial, though, because each RSN and RSN acknowledgement must be sent in a separate packet and must be handled separately.

To put this measured average communication overhead of 25 percent from sender-based message logging into perspective, the expected performance of other pessimistic message logging techniques may be considered. If messages are logged on some separate logging node on the network, without using special network hardware, the overhead should be approximately 100 percent, since all messages must be sent and received one extra time. If each message is synchronously written to disk as it is received, the overhead should be several orders of magnitude higher, due to the relative speed of the disk. Such an approach does allow for recovery from any number of failures at once, though.

3.4.2 Checkpointing Costs

Table 3.3 shows the measured elapsed time for performing checkpointing in this implementation, based on the size of the address space portion written to the checkpoint file. During a checkpoint, each individual contiguous range of modified pages is written to the file in a separate operation, and for a given address space size, the time required to write the address space increases as the number of these separate write operations increases. For the measurements reported in Table 3.3, the total address space size was twice the size of the modified portion, and the modified pages were each separated from one another by one unmodified page, resulting in the maximum possible number of separate write operations. The hardware page size on the SUN-3/60 is 8 kilobytes.

Within these checkpointing times, the logical host is frozen, and its execution is suspended, for only a portion of this time, as discussed in Section 3.3.6. The total time frozen is highly dependent on the behavior of the application program being checkpointed, and is affected primarily by the rate at which the logical host modifies new pages of its address space during the checkpoint before the host is frozen. For the application programs measured, the logical host is frozen typically for only a few tens of milliseconds during each checkpoint.

Table 3.3

Checkpointing time by size of address space portion written (milliseconds)

Kilobytes	Pages	Full	Incremental
8	1	140	140
16	2	170	170
32	4	200	220
64	8	260	300
128	16	460	500
256	32	780	880
512	64	1470	1570
1024	128	2870	2980

The measurements in Table 3.3 illustrate that the elapsed time required to complete a checkpoint is dominated by the cost of writing the address space to the checkpoint file. The elapsed time required to complete the checkpoint grows roughly linearly with the size of the address space portion written to the file. The extra cost involved in an incremental checkpoint results primarily from the fact that the modified portions of the address space are typically not contiguous. For each discontiguous range of modified pages, a separate write operation must be performed. A full checkpoint, on the other hand, performs only a single write operation for the entire address space. The extra overhead per write operation has been measured to be approximately 5 milliseconds.

Other costs involved in checkpointing are minor. A total of approximately 17 milliseconds is required to open the checkpoint file and later close it. Checkpointing the state of the kernel requires about 0.8 milliseconds, and checkpointing the team server requires about 1.3 milliseconds. The time required to checkpoint the logging server varies with the number of message log blocks to be written to the logging file. A minimum of about 18 milliseconds is required, and increases about 25 milliseconds per message log block written.

3.4.3 Recovery Costs

The costs involved in recovery are similar to those involved in checkpointing. The address space of the logical host being recovered must be read from the checkpoint file into memory. The state of the kernel must be restored, as well as the states of the team server and the logging server. In addition, the sequence of messages received by this logical host after the checkpoint must be retrieved, and the logical host must complete the reexecution necessary to restore its state to the value it had after receiving these messages before the failure.

Table 3.4 shows the measured times required for recovery based on the size of the address space of the logical host being recovered. These measurements do not include the time required for the logical host to reexecute from the checkpointed state, since this time is highly dependent on the particular application being recovered. In

Table 3.4

Recovery time by address space size (milliseconds)

Kilobytes	Pages	Time
8	1	2580
16	2	2600
32	4	2620
64	8	2670
128	16	2760
256	32	2950
512	64	3320
1024	128	4080

general, this reexecution time is bounded by the interval at which checkpoints are recorded. As with the cost of checkpointing, the measured recovery times increase approximately linearly with the size of the address space being read. There is also a large fixed cost included in these recovery times, due to the necessary timeout of the last group send of the request to collect the next logged message. In the current implementation, this timeout is 2.5 seconds.

3.4.4 Application Program Performance

The preceding three sections have examined the three sources of overhead caused by the operation of sender-based message logging: message logging, checkpointing, and failure recovery. Distributed application programs, though, spend only a portion of their execution time on communication, and checkpointing and failure recovery occur only infrequently. To analyze the overhead of sender-based message logging in a more realistic environment, the performance of three distributed application programs was measured using this implementation. The following three application programs were used in this study:

`nqueens`: This program counts the number of solutions to the *n-queens problem* for a given number of queens n. The problem is distributed among multiple processes by assigning each a range of subproblems resulting from an equal division of the possible placements of the first two queens. When each process finishes its allocated subproblems, it reports the number of solutions found to the main process.

`tsp`: This program finds the solution to the *traveling salesman problem* for a given map of n cities. The problem is distributed among multiple processes by assigning each a different initial edge from the starting city to include in all paths. A branch-and-bound technique is used. When each new possible solution is found by some process, it is reported to the main process, which maintains the minimum known solution and returns its length to this process. When a process finishes its assigned search, it requests a new edge of the graph from which to search.

`gauss`: This program performs *Gaussian elimination with partial pivoting* on a given $n \times n$ matrix of floating point numbers. The problem is distributed among multiple processes by giving each a subset of the matrix rows on which to operate. At each step of the reduction, the processes send their possible pivot row number and value to the main process, which determines the row to be used. The current contents of the pivot row is sent from one process to all others, and each process performs the reduction on its rows. When the last reduction step completes, each process returns its rows to the main process.

These three programs were chosen because of their dissimilar communication rates and patterns. In the `nqueens` program, the main process exchanges a message with each other process at the start of execution and again at completion, but no other communication is performed during execution. The subordinate processes do not communicate with one another, and the total amount of communication is constant for all problem sizes. In the `tsp` program, the map is distributed to the subordinate processes, which then communicate with the main process to request new subproblems and to report any new results during each search. Since the number of subproblems is bounded by the number of cities in the map, the total amount of communication performed is $O(n)$ for a map of n cities. Due to the branch-and-bound algorithm used, though, the running time is highly dependent on the map input. Again, there is no communication between subordinate processes. The `gauss` program performs the most communication of the three programs, including communication between all processes during execution. The matrix rows are distributed to the subordinate processes and collected at completion, each pivot row is decided by the main process, and the contents of the pivot row is distributed from one subordinate process to all others. The total amount of communication performed is $O(n^2)$ for an $n \times n$ matrix.

These three application programs were used to solve a fixed set of problems. Each problem was solved multiple times, both with and without using sender-based message logging. The maps used for `tsp` and the matrices used for `gauss` were randomly generated, but were saved for use on all executions. For each program, the problem was distributed among 8 processes, each executing on a separate node of the system. When using sender-based message logging, all messages sent between application processes were logged. No checkpointing was performed during these tests, since its overhead is highly dependent on the frequency at which checkpoints are written.

The overhead of using sender-based message logging ranged from about 2 percent to much less than 1 percent for most problems in this set. For the `gauss` program, which performs more communication than the other programs, the overhead was slightly higher, ranging from about 16 percent to about 3 percent. As the problem size increases for each program, the percentage overhead decreases because the average amount of computation between messages sent increases. Table 3.5 summarizes the performance of these application programs for all problems in this set. Each entry in this table shows the application program name and the problem size n. The running time in seconds required to solve each problem, both with and without using sender-based message logging, is given. The overhead of using sender-based message logging

Table 3.5

Performance of the distributed application programs using sender-based message logging (seconds)

		Message Logging		Overhead	
Program	Size	With	Without	Time	Percent
`nqueens`	12	5.99	5.98	.01	.17
	13	34.61	34.60	.01	.03
	14	208.99	208.98	.01	.01
`tsp`	12	5.30	5.19	.11	2.12
	14	16.40	16.13	.27	1.67
	16	844.10	841.57	2.53	.30
`gauss`	100	12.41	10.74	1.67	15.55
	200	71.10	66.40	4.70	7.08
	300	224.06	217.01	7.05	3.25

for each problem is shown in seconds as the difference between its two average running times, and as a percentage increase over the running time without logging.

Table 3.6 shows the average message log sizes per node that result from solving each of these problems using sender-based message logging. For each problem, the average total number of messages logged and the resulting message log size in kilobytes are shown. These figures are also shown averaged over the elapsed execution time of the problem. These message log sizes are all well within the limits of available memory on the workstations used in these tests and on other similar contemporary machines.

The degree to which these three programs utilize of the piggybacking optimization of the logging protocol is summarized in Table 3.7, by the percentage of messages sent, averaged over all processes. For each problem, the three possible cases encountered when sending a message are shown individually. If no unacknowledged RSNs are pending, the message is sent immediately with no piggybacked RSNs. If all unacknowledged RSNs can be included in the same packet, they are piggybacked on

Table 3.6

Message log sizes for the distributed application programs (average per node)

Program	Size	Total Messages	Total Kilobytes	Per Second Messages	Per Second Kilobytes
`nqueens`	12	8	1.9	1.30	.32
	13	8	1.9	.23	.06
	14	8	1.9	.04	.01
`tsp`	12	43	5.5	8.09	1.04
	14	48	6.1	2.91	.37
	16	59	7.3	.07	.01
`gauss`	100	514	95.4	41.44	7.69
	200	1113	292.8	15.66	4.12
	300	1802	593.7	8.04	2.65

Table 3.7

Application program piggybacking utilization (percentage of messages sent)

Program	Size	None Pending	Piggybacked	Delay For RSN Ack
`nqueens`	12	42.9	44.4	12.7
	13	41.3	46.0	12.7
	14	41.3	46.0	12.7
`tsp`	12	19.9	62.4	17.6
	14	22.3	63.5	14.2
	16	33.0	58.3	8.7
`gauss`	100	20.7	30.0	49.2
	200	29.4	28.6	42.0
	300	29.0	31.0	40.0

it and the message is sent immediately. Otherwise, the packet cannot be sent now and must be delayed until all RSNs for messages previously received by the process have been acknowledged. The utilization of piggybacking was lowest in the `gauss` program, since its communication pattern allowed all processes to communicate with one another during execution. This reduces the probability that a message being sent is destined for the same process as the pending unacknowledged RSNs, which is required in order to piggyback the RSNs on the message. Likewise, for the `nqueens` and `tsp` programs, piggybacking utilization was lower in the main process than in the subordinate processes due to the differences in their communication patterns. For all problems, more than half the messages could be sent without delaying, either because no unacknowledged RSNs were pending or because they could be piggybacked on the message.

These same tests were executed again using a sender-based message logging implementation that did not include the protocol optimizations. In this implementation, no messages could be sent with piggybacked RSNs. Instead, the percentage of messages that were sent with piggybacked RSNs using the optimized protocol was in this case divided approximately equally between those sent while no unacknowledged RSNs were pending and those forced to delay for RSN acknowledgements. Because piggybacking may postpone the return of an RSN acknowledgement, some messages may be forced to delay with piggybacking that could be sent immediately without piggybacking. Although this effect could not be measured directly, these average measurements indicate that this does not commonly occur.

To evaluate the effect of checkpointing on the overhead of sender-based message logging, each application program was executed again with checkpointing enabled. The largest problem for each program was solved again with checkpoints being written every 15 seconds by each process during execution. A high checkpointing frequency was used to generate a significant amount of checkpointing activity to be measured. For the `nqueens` and `tsp` programs, the additional overhead was less than 0.5 percent of the running time with sender-based message logging. For the `gauss` program, checkpointing overhead was about 2 percent. This overhead is higher than for the other two programs because `gauss` modifies more data during execution, which must be written to the checkpoint. In all cases, the average additional time required for each checkpoint was much less than the total time required for checkpointing reported in Table 3.3, because most time spent during checkpointing waiting for writes to the disk to complete could be overlapped with execution of the application before the logical host was frozen.

3.5 Multiple Failure Recovery

Sender-based message logging is designed to recover from only a single failure at a time, since messages are logged in volatile memory. That is, after one process (or logical host in the V-System) has failed, no other process may fail until the recovery from the first is completed. Sender-based message logging can be extended, though, to recover from more than one failure at at time.

Sender-based message logging uses the dependency vector of each process to verify that the resulting system state after recovery is consistent. As discussed in Chapter 2, if a process X depends on a state interval of some failed process Y that resulted from Y receiving a message beyond those messages logged for Y that are available for replay, the system state that can be recovered by the existing sender-based message logging protocol is not consistent. To recover a consistent system state in this situation would require each such process X to be rolled back to a state interval before it received the message from Y that caused this dependency.

With the existing sender-based message logging protocol, this consistent system state can be recovered only if the current checkpoint for each such process X that must be rolled back was written before the corresponding message from each process Y was received. By rolling each such process X back to before this dependency was created, the resulting system state must be consistent because no process then depends on a state of some other process that cannot be reached through its deterministic execution (Definition 2.4). In this case, each process X can be rolled back by forcing it to fail and recovering it using this checkpoint. However, if the current checkpoint was written after the message from Y was received, process X cannot roll back far enough to remove the dependency.

If a consistent system state can be recovered in this way, to preserve as much of the existing volatile message log as possible, each of these processes must be rolled back one at a time after the reexecution of the original failed processes is completed. As the original failed processes reexecute using the sequences of messages that could be replayed, they resend any messages they sent before the failure, and thus recreate much of their original volatile message log that was lost from the failure. Then, as each of these additional processes is forced to fail and recovered from its earlier checkpoint, it will recreate its volatile message log during its reexecution as well. By rolling them back one at a time, no additional logged messages needed for their reexecution from their checkpointed states will be lost.

If the checkpoint for some process X that must be rolled back was not written early enough to allow the process to roll back to before the dependency on process Y was created, recovery of a consistent system state using this method is not possible. To guarantee the recovery of a consistent system state even in this case, the sender-based message logging protocol can be modified to retain on stable storage all checkpoints for all processes, rather than saving only the most recent checkpoint for each process. Then, the existence of an early enough checkpoint for each process X that must be rolled back is ensured. Although not all checkpoints must be retained to guarantee recovery, the existing sender-based message logging protocol does not maintain sufficient information to determine during failure-free execution when each checkpoint can safely be released. The domino effect is still avoided by this extension since the data in the checkpoints is not volatile. Once a process has rolled back to a checkpoint, all messages sent by it before that time have been logged as part of the checkpoints of that process.

3.6 Related Work

The sender-based message logging protocol differs from other message logging protocols primarily in that messages are logged in the local *volatile* memory of the *sender*. Also, sender-based message logging requires no specialized hardware to assist with logging. The TARGON/32 system [Borg89] and its predecessor Auros [Borg83] log messages at a backup node for the receiver, using specialized networking hardware that provides three-way atomic broadcast of each message. With this networking hardware assistance and using available idle time on a dedicated processor of each multiprocessor node, the overhead of providing fault tolerance in TARGON/32 has been reported to be about 10 percent [Borg89]. Sender-based message logging causes less overhead for all but the most communication-intensive programs, without the use of specialized hardware. The PUBLISHING mechanism [Powell83] proposes the use a centralized logging node for all messages, which must reliably receive every network packet. Although this logging node avoids the need to send an additional copy of each message over the network for logging, providing this reliability guarantee seems to be impractical without additional protocol complexity [Saltzer84]. Strom and Yemini's Optimistic Recovery mechanism [Strom85] logs all messages on stable storage on disk, but Strom, Bacon, and Yemini have recently proposed enhancements to it using ideas from sender-based message logging [Johnson87] to avoid logging some messages on stable storage [Strom88].

Another difference between sender-based message logging and previous *pessimistic* logging protocols, which is not related to the logging of messages at the sender, is in the enforcement of the requirements of a pessimistic logging protocol. All pessimistic logging protocols must prevent the system from entering a state that could cause any process other than those that failed to be rolled back during recovery. Previous pessimistic logging protocols [Borg83, Powell83, Borg89] have required each message to be logged *before it is received* by the destination process, blocking the receiver while the logging takes place. Sender-based message logging allows the message to be received before it is logged, but prevents the receiver from *sending new messages* until all messages it has received are logged. This allows the receiver to execute based on the message data while the logging proceeds asynchronously. For example, if the message requests some service of the receiver, this service can begin while the message is being logged.

Optimistic message logging methods [Strom85] have the potential to outperform pessimistic methods, since message logging proceeds asynchronously without delaying either the sender or the receiver for message logging to complete. However, these methods require significantly more complex protocols for logging. Also, failure recovery in these systems is more complex and may take longer to complete, since processes other than those that failed may need to be rolled back to recover a consistent system state. Finally, optimistic message logging systems may require substantially more storage during failure-free operation, since logged messages may need to be retained longer, and processes may be required to save additional checkpoints earlier than their most recent. Sender-based message logging achieves some of the advantages of asynchronous logging more simply by allowing messages to be received before they are fully logged.

Some of the simplicity of the sender-based message logging protocol results from the concentration on recovering from only a single failure at a time. This allows the messages to be logged in volatile memory, significantly reducing the overhead of logging. This assumption of a single failure at a time was also made by Tandem NonStop, Auros, and TARGON/32, but without achieving such a reduction in fault-tolerance overhead. The addition of the extensions of Section 3.5 to handle multiple failures causes no additional overhead during failure-free operation, although to guarantee recovery using the second extension requires that all checkpoints be retained on stable storage. Also, the recovery from multiple concurrent failures using these extensions may require longer to complete than with other methods, since processes must be rolled back one at a time during recovery.

3.7 Summary

Sender-based message logging is a new transparent method of providing fault tolerance in distributed systems, which uses *pessimistic* message logging and checkpointing to record information for recovering a consistent system state following a failure. It differs from other message logging protocols primarily in that the message log is stored in the *volatile* memory on the node from which the message was *sent*. The order in which the message was *received* relative to other messages sent to the same receiver is required for recovery, but this information is not usually available to the message sender. With sender-based message logging, when a process receives a message, it returns to the sender a *receive sequence number* (*RSN*) to indicate this ordering information. The RSN of a process is equivalent to its current state interval index. When the RSN arrives at the sender, it is added to the local volatile log with the message. To recover a failed process, it is restarted from its most recent checkpoint, and the sequence of messages received by it after this checkpoint are replayed to it in ascending order of their logged RSNs.

Sender-based message logging concentrates on reducing the overhead placed on the system from the provision of fault tolerance by a pessimistic logging protocol. The cost of message logging is the most important factor in this system overhead. Keeping the message log in the sender's local volatile memory avoids the expense of synchronously writing each message to disk or sending an extra copy over the network to some special logging process. Since the message log is volatile, the basic sender-based message logging protocol guarantees recovery from only a single failure at a time within the system. Extensions to the protocol support recovery from any number of concurrent failures.

Performance measurements from a full implementation of sender-based message logging under the V-System verify the efficient nature of this protocol. Measured on a network of SUN-3/60 workstations, the overhead on V-System communication operations is approximately 25 percent. The overhead experienced by distributed application programs using sender-based message logging is affected most by the amount of communication performed during execution. For Gaussian elimination, the most communication-intensive program measured, this overhead ranged from about 16 percent to about 3 percent, for different problem sizes. For all other programs measured, overhead ranged from about 2 percent to much less than 1 percent.

Chapter 4
Optimistic Message Logging

Optimistic message logging protocols have the potential to outperform pessimistic message logging protocols during failure-free operation, since the message logging proceeds asynchronously. However, previous systems using optimistic message logging have required large amounts of additional communication, and have utilized complex algorithms for finding the current recovery state of the system. This chapter presents a new fault-tolerance method using optimistic message logging, based on the theoretical development of Chapter 2, that requires significantly less communication and uses a new efficient algorithm for finding the current recovery state. This new method also guarantees to always find the *maximum* possible recoverable system state, by utilizing logged messages *and* checkpoints. Previous recovery methods using optimistic message logging and checkpointing have not considered the existing checkpoints, and thus may not find this maximum state. Furthermore, by utilizing these checkpointed states, some messages received by a process before it was checkpointed may not need to be logged.

4.1 Overview

Theorem 2.4 showed that in any system using message logging and checkpointing, there is always a unique maximum recoverable system state. This maximum state is the *current recovery state*, which is the state to which the system will be restored following a failure. However, the model of Chapter 2 does not address the question of how the current recovery state can be determined from the collection of individual process state intervals that are currently stable.

The algorithm used by the system to find the current recovery state is called the *recovery state algorithm.* A straightforward algorithm to find the current recovery state would be to perform an exhaustive search, over all combinations of currently stable process state intervals, for the maximum consistent combination. However, such a search would be too expensive in practice. This chapter presents two alternative algorithms that find the current recovery state more efficiently. These algorithms are independent of the protocols used by the system for message logging, checkpointing, or failure recovery. In particular, they make no assumptions on the order in which checkpoints and logged messages are recorded on stable storage. These algorithms are each assumed to run centrally on the shared stable storage server on which all checkpoints and logged messages are recorded. Either of these algorithms can be used with the logging system developed in this chapter.

This chapter presents a *batch* recovery state algorithm and an *incremental* recovery state algorithm. The batch algorithm finds the current recovery state "from scratch" each time it is invoked. No internal state is saved between executions of the algorithm. This algorithm can be used at any time to find the new current recovery state, after any number of new process state intervals have become stable. The incremental algorithm, instead, must be executed once for *each* new process state interval that becomes stable. It begins its search for the new current recovery state with the previously known value, and updates it based on the fact that a single new process state interval has become stable. Internal state saved from previous executions of the algorithm is used to shorten the search.

Unlike the sender-based message logging system described in Chapter 3, messages in this system are logged by the *receiver* rather than by the *sender.* As each message is received, it is copied into a buffer in volatile memory. Occasionally, the contents of this buffer is written to stable storage, which logs all messages received since the buffer was last written. Writing this buffer to stable storage is done *asynchronously,* and does not block execution of any user process. Since all message logging is done by the receiver, no interaction with the sender of a message is needed to complete the logging of any message.

4.2 Protocol Specification

The design of this system follows the model presented in Chapter 2. However, many details of a practical system are unspecified there. This section reviews the features of the model that make up the protocols used by the system, and supplies the details necessary to form a complete specification.

4.2.1 Data Structures

The following new data structures are maintained by each process:

- A *state interval index*, which is incremented each time a new message is received. Each message sent by a process is tagged with the current state interval index of the sender.

- A *dependency vector*, recording the maximum index of any state interval of each other process on which this process currently depends. For each other process from which this process has received messages, the dependency vector records the maximum state interval index tagging a message *received* from that process.

- A buffer of messages *received* by the process that have not yet been logged on stable storage. For each message in the buffer, the message data, the identification of the sending process, and the SSN (send sequence number) and state interval index tagging the message are recorded in the buffer. This buffer is stored in the local volatile memory of the receiving node. Messages are logged by writing this buffer to stable storage.

Each of these data items except the message buffer must be included in the checkpoint of the process. Also, the existing data structures used by a process for duplicate message detection (Section 1.2) must be included in the checkpoint.

4.2.2 Process Operation

During execution, each process maintains its own current *state interval index*, which is equal to the number of messages received by the process. Each message received by a process is first checked to determine if it is a duplicate, using the SSN tagging the message. Any duplicate messages are ignored. If the message is not a duplicate, the state interval index of the process is incremented before the process is allowed to examine the message. Messages sent by a process are also tagged with the current state interval index of the sender. When a new message is received by a process, the dependency vector of the receiving process is updated by setting its entry for the sending process to the maximum of its current value and the state interval index tagging the message received.

Each new message received is also saved in a message buffer in the volatile memory of the receiving node. Messages are saved in this buffer until they are logged by writing the contents of the buffer to the message log on stable storage. This buffer may be written at any time, such as when the amount of volatile memory used by buffered messages exceeds some defined threshold, or after some defined time limit on the delay between receiving and logging a message is reached. Likewise, the process may be checkpointed at any time, such as after receiving some given number of messages or consuming some given amount of processor since its last checkpoint. No synchronization is required between message logging, checkpointing, and the communication and computation of any process.

Each logged message includes the index of state interval started in the receiver by that message. Each checkpoint includes the state interval index of the process at the time that the checkpoint was written, and the complete current dependency vector of the process. Once new messages received by a process are logged or new checkpoints of a process are recorded on stable storage, some new state intervals of that process may become stable, by Definition 2.6.

4.2.3 Failure Recovery

After a failure, the state of each surviving process is recorded on stable storage as an additional checkpoint of that process. Also, any surviving messages that have been received but not yet logged are recorded on stable storage in the message log. The recovery state algorithm is then used to determine the current recovery state of the system, indicating the state to which the system will be restored as a result of this failure. The surviving process states and received messages must be written to stable storage before determining the current recovery state, since only information recorded on stable storage is used by the recovery state algorithm. This allows the algorithm to be restartable in case another failure occurs during this recovery.

To restore the state of the system to the current recovery state, the states of all failed processes must be restored. Each failed process is first restarted from the effective checkpoint for its state interval in the current recovery state. Any messages received the process since that checkpoint are then replayed to it from the log, and using these logged messages, the recovering process deterministically reexecutes to restore its state to the state interval for this process in the current recovery state. In addition to restoring the state of each failed process, any other process currently executing in a state interval beyond its state interval in the current recovery state must be rolled back to complete recovery. Each such process is forced to fail and is restored to its state interval in the current recovery state in the same way as other failed processes. If additional processes fail during recovery, the recovery is simply restarted, since all information used in the algorithm is recorded on stable storage.

4.3 The Batch Recovery State Algorithm

The *batch* recovery state algorithm is a simple algorithm for determining the current recovery state of the system. It can be executed at any time, and considers all process state intervals that are currently stable in finding the maximum consistent system state composed of them. In the current implementation, which will be described in Section 4.5, the algorithm is executed only when beginning recovery after a failure.

However, it could also be executed when a group of new messages are logged by some process, or could be executed periodically in order to allow output to the outside world to be released more quickly.

Conceptually, the algorithm begins its search for the current recovery state at the highest point in the portion of the system history lattice that has been recorded on stable storage, and searches backward to lower points in the lattice until a consistent system state is found. This system state must then be the current recovery state of the system. In particular, the following steps are performed by the algorithm:

1. Make a new dependency matrix $\mathbf{D} = [\delta_{**}]$, where each row i is set to the dependency vector for the maximum stable state interval of process i.

2. Loop on step 2 while $\mathbf{D}$ is not consistent. That is, loop while there exists some i and j for which $\delta_{ji} > \delta_{ii}$, showing that state interval δ_{jj} of process j depends on state interval δ_{ji} of process i, which is greater than process i's current state interval δ_{ii} in $\mathbf{D}$.

   (a) Find the maximum index α less than δ_{jj} of any *stable* state interval of process j such that component i of the dependency vector for this state interval α of process j is not greater than δ_{ii}.

   (b) Replace row j of $\mathbf{D}$ with the dependency vector for this state interval α of process j.

3. The system state represented by $\mathbf{D}$ is now consistent and is composed entirely of stable process state intervals. It is thus the current maximum recoverable system state. Return it as the current recovery state.

A procedure to perform this algorithm is shown in Figure 4.1. Here, *DM* is a dependency matrix used by the algorithm to represent the system state that is currently being considered. For each stable process state interval α of each process i, the vector DV_i^α represents the dependency vector for that state interval. The result of the algorithm is a vector *CRS*, such that for each i, $CRS[i]$ records the state interval index for process i in the new current recovery state.

for $i \leftarrow 1$ **to** n **do**
  $\alpha \leftarrow$ maximum index such that stable(i, α);
  set row i of matrix DM to DV_i^α;

while $\exists\, i, j$ such that $DM[j,\, i] > DM[i,\, i]$ **do**
    $\alpha \leftarrow$ maximum index less than $DM[j,\, j]$ such that
      stable$(j, \alpha)\ \wedge\ DV_j^\alpha[i] \leq DM[i,\, i]$;
    replace row j of DM with DV_j^α;

for $i \leftarrow 1$ **to** n **do** $CRS[i] \leftarrow DM[i,\, i]$;

Figure 4.1 The batch recovery state algorithm

Theorem 4.1 The batch recovery state algorithm completes each execution with $CRS[i] = \rho'_{i\,i}$, for all i, where $\mathbf{R}' = [\rho'_{*\,*}]$ is the current recovery state of the system when the algorithm is executed.

Proof If the algorithm terminates, the system state found must be consistent, as ensured by the predicate of the **while** loop, by Definition 2.4. This system state must also be recoverable, since each process state interval included in it by the **for** loop at the beginning of the algorithm is stable, and only stable process state intervals are used to replace components of it during the execution of the **while** loop. Since logged messages and checkpoints are not removed from stable storage until no longer needed for any future possible failure recovery, following Lemmas 2.4 and 2.5, *some* recoverable system state must exist in the system. The search by the algorithm starts at the maximum system state, say $\mathbf{D}$, composed of only stable process state intervals. Therefore, some recoverable system state $\mathbf{R}' \preceq \mathbf{D}$ must exist. Thus, by Theorem 2.1, the algorithm must find some recoverable system state, and therefore must terminate.

The following loop invariant is maintained by the algorithm at the beginning of the **while** loop body on each iteration:

> If the matrix DM represents system state $\mathbf{D}$, then no recoverable system state $\mathbf{R}'$ currently exists such that $\mathbf{D} \prec \mathbf{R}'$.

Before the first iteration of the loop, this invariant must hold, since the system state $\mathbf{D}$ is the maximum system state that currently exists having each component process state interval stable. On each subsequent iteration of the loop, if the loop predicate finds some i and j such that $DM[j, i] > DM[i, i]$, then process j (in state interval $DM[j, j]$) in system state $\mathbf{D}$ depends on state interval $DM[j, i]$ of process i, but process i in $\mathbf{D}$ is only in state interval $DM[i, i]$. Therefore, the system state represented by $\mathbf{D}$ is not consistent. The loop invariant is maintained by choosing the *largest* index $\alpha < DM[j, j]$ such that state interval α of process j is stable and does not depend on a state interval of process i greater than $DM[i, i]$. Since on each iteration of the loop, the state interval index of one process decreases, along with its corresponding dependency vector, the system state $\mathbf{D}$ represented by DM after each iteration precedes its value from the previous iteration. Thus, no recoverable system state $\mathbf{R}'$ currently exists such that $\mathbf{D} \prec \mathbf{R}'$.

When the **while** loop terminates, the system state $\mathbf{D}$ represented by DM must be a recoverable system state. Thus, by the loop invariant, $\mathbf{D}$ is the *maximum* recoverable system state that currently exists, and must therefore be the current recovery state of the system. The algorithm then copies the state interval index of each process in $\mathbf{D}$ into the corresponding elements of CRS. □

4.4 The Incremental Recovery State Algorithm

Although the batch recovery state algorithm is simple, it may repeat a substantial amount of work each time it is invoked, since it does not save any partial results from previous searches. The *incremental* recovery state algorithm attempts to improve on this by utilizing its previous search results. Whereas the batch algorithm searches downward from this highest point in the lattice, the incremental algorithm searches upward in the lattice, from the *previous* current recovery state, which is the maximum known recoverable system state.

The algorithm is invoked once for each new process state interval that becomes stable, either as a result of a new checkpoint being recorded for the process in that state interval, or because all messages received since the effective checkpoint for that interval are now logged. For each new state interval σ of some process k that becomes stable, the algorithm determines if a new current recovery state exists. It first attempts to find *some* new recoverable system state in which the state of process k has advanced to state interval σ. If no such system state exists, the current recovery state of the system has not changed. The algorithm records the index of this state interval and its process identifier on one or more lists to be checked again later. If a new recoverable system state is found, the algorithm searches for other greater recoverable system states, using the appropriate lists from its earlier executions. The new current recovery state is the maximum recoverable system state found in this search.

4.4.1 Finding a New Recoverable System State

The heart of the batch recovery state algorithm is the function *FIND_REC*. Given *any* recoverable system state $\mathbf{R} = [\rho_{**}]$ and some stable state interval σ of some process k with $\sigma > \rho_{k\,k}$, *FIND_REC* attempts to find a new recoverable system state in which the state of process k is advanced at least to state interval σ. It does this by also including any stable state intervals from other processes that are necessary to make the new system state consistent, applying the definition of a consistent system state in Definition 2.4. The function succeeds and returns **true** if such a consistent system state can be composed from the set of process state intervals that are currently stable. Since the state of process k has advanced, the new recoverable system state found must be greater than state $\mathbf{R}$ in the system history lattice.

Input to the function *FIND_REC* consists of the dependency matrix of some recoverable system state $\mathbf{R} = [\rho_{**}]$, the process identifier k and state interval index $\sigma > \rho_{k\,k}$ of a stable state interval of process k, and the dependency vector for each stable process state interval θ of each process x such that $\theta > \rho_{x\,x}$. Conceptually, *FIND_REC* performs the following steps:

1. Make a new dependency matrix $\mathbf{D} = [\delta_{*\,*}]$ from matrix $\mathbf{R}$, with row k replaced by the dependency vector for state interval σ of process k.

2. Loop on step 2 while $\mathbf{D}$ is not consistent. That is, loop while there exists some i and j for which $\delta_{j\,i} > \delta_{i\,i}$. This shows that state interval $\delta_{j\,j}$ of process j depends on state interval $\delta_{j\,i}$ of process i, which is greater than process i's current state interval $\delta_{i\,i}$ in $\mathbf{D}$.

   Find the minimum index α of any *stable* state interval of process i such that $\alpha \geq \delta_{j\,i}$:

   (a) If no such state interval α exists, exit the algorithm and return **false**.

   (b) Otherwise, replace row i of $\mathbf{D}$ with the dependency vector for this state interval α of process i.

3. The system state represented by $\mathbf{D}$ is now consistent and is composed entirely of stable process state intervals. It is thus recoverable and greater than $\mathbf{R}$. Return **true**.

An efficient procedure for the function *FIND_REC* is shown in Figure 4.2. This procedure operates on a vector *RV*, rather than on the full dependency matrix representing the system state. For all i, $RV[i]$ contains the diagonal element from row i of the corresponding dependency matrix. When *FIND_REC* is called, each $RV[i]$ contains the state interval index of process i in the given recoverable system state. The dependency vector of each stable state interval θ of each process x is represented by the vector DV_x^θ. As each row of the matrix is replaced in the algorithm outline above, the corresponding single element of *RV* is changed in *FIND_REC*. Also, the maximum element from each column of the matrix is maintained in the vector *MAX*, such that for all i, $MAX[i]$ contains the maximum element in column i of the corresponding matrix.

```
function FIND_REC(RV, k, σ)

        RV[k] ← σ;
        for i ← 1 to n do MAX[i] ← max(RV[i], DV_k^σ[i]);

        while ∃ i such that MAX[i] > RV[i] do
                α ← minimum index such that
                        α ≥ MAX[i] ∧ stable(i, α);
                if no such state interval α exists then return false;
                RV[i] ← α;
                for j ← 1 to n do MAX[j] ← max(MAX[j], DV_i^α[j]);

        return true;
```

Figure 4.2 Procedure to find a new recoverable state

Lemma 4.1 If the function *FIND_REC* is called with a known recoverable system state $\mathbf{R} = [\rho_{**}]$ and state interval σ of process k such that $\sigma > \rho_{kk}$, *FIND_REC* returns **true** if there exists *some* recoverable system state $\mathbf{R}' = [\rho'_{**}]$, such that $\mathbf{R} \prec \mathbf{R}'$ and $\rho'_{kk} \geq \sigma$, and returns **false** otherwise. If *FIND_REC* returns **true**, then on return, $RV[i] = \rho'_{ii}$, for all i.

Proof The predicate of the **while** loop determines whether the dependency matrix corresponding to RV and MAX is consistent, by Definition 2.4. When the condition becomes false and the loop terminates, the matrix must be consistent because, in each column i, no element is larger than the element on the diagonal in that column. Thus, if *FIND_REC* returns **true**, the system state returned in RV must be consistent. This system state must also be recoverable, since its initial component process state intervals are stable and only stable process state intervals are used to replace its entries during the execution of *FIND_REC*.

The following loop invariant is maintained by function *FIND_REC* at the beginning of the **while** loop body on each iteration:

> If a recoverable system state $\mathbf{R}' = [\rho'_{**}]$ exists such that $\mathbf{R} \prec \mathbf{R}'$ and $\rho'_{ii} \geq RV[i]$, for all i, then $\rho'_{ii} \geq MAX[i]$.

The invariant must hold before the first iteration of the loop, since any consistent system state must have $RV[i] \geq MAX[i]$, for all i, and any state $\mathbf{R}'$ found such that $\rho'_{ii} \geq RV[i]$ must then have $\rho'_{ii} \geq RV[i] \geq MAX[i]$. On each subsequent iteration of the loop, the invariant is maintained by choosing the *smallest* index $\alpha \geq MAX[i]$ such that state interval α of process i is stable. For the matrix to be consistent, α must not be less than $MAX[i]$. By choosing the minimum such α, all components of DV_i^α are also minimized because no component of the dependency vector can decrease through execution of the process. Thus, after replacing row i of the matrix with DV_i^α, the components of MAX are minimized, and for any recoverable (consistent) state $\mathbf{R}'$ that exists, the condition $\rho'_{ii} \geq MAX[i]$ must still hold for all i.

If no such state interval $\alpha \geq MAX[i]$ of process i is currently stable, then no recoverable system state $\mathbf{R}'$ can exist, since any such $\mathbf{R}'$ must have $\rho'_{ii} \geq MAX[i]$. This is exactly the condition under which the function *FIND_REC* returns **false**. $\square$

Suppose state interval σ of process k depends on state interval δ of process i, then the function *FIND_REC* searches for the minimum $\alpha \geq \delta$ that is the index of a state interval of process i that is currently stable. For the set of process state intervals that are currently stable, the dependency on state interval δ of process i has been *transferred* to state interval α of process i (including the case in which $\alpha = \delta$), and state interval σ of process k is said to currently have a *transferred dependency* on state interval α of process i.

> **Definition 4.1** A state interval σ of some process k, with dependency vector $\langle \delta_* \rangle$, has a *transferred dependency* on a state interval α of process i if and only if:

(1) $\alpha \geq \delta_i$,

(2) state interval α of process i is stable, and

(3) there does not exist another stable state interval β of process i such that $\alpha > \beta \geq \delta_i$.

The transitive closure of the transferred dependency relation from state interval σ of process k describes the set of process state intervals that *may* be used in any iteration of the **while** loop of the function *FIND_REC*, when invoked for this state interval. Although only a subset of these state intervals will actually be used, the exact subset used in any execution depends on the order in which the **while** loop finds the next i that satisfies the predicate.

4.4.2 The Complete Algorithm

Using the function *FIND_REC*, the complete incremental recovery state algorithm can now be stated. The algorithm, shown in Figure 4.3, uses a vector *CRS* to record the state interval index of each process in the current recovery state of the system. When a process is created, its entry in *CRS* is initialized to 0. When some state interval σ of some process k becomes stable, if this state interval is in advance of the old current recovery state in *CRS*, the algorithm checks if a new current recovery state exists. During the execution, the vector *NEWCRS* is used to store the maximum known recoverable system state, which is copied back to *CRS* at the completion of the algorithm.

When invoked, the algorithm calls *FIND_REC* with the old current recovery state and the identification of the new stable process state interval. The old current recovery state is the maximum known recoverable system state, and the new stable state interval is interval σ of process k. If *FIND_REC* returns **false**, then no greater recoverable system state exists in which the state of process k has advanced at least to state interval σ. Thus, the current recovery state of the system has not changed, as shown by the following two lemmas.

```
if σ ≤ CRS[k] then exit;

NEWCRS ← CRS;

if ¬FIND_REC(NEWCRS, k, σ) then
    for i ← 1 to n do if i ≠ k then
        β ← DV_k^σ[i];
        if β > CRS[i] then DEFER_i^β ← DEFER_i^β ∪ { (k, σ) };
    exit;

WORK ← DEFER_k^σ;
β ← σ − 1;
while ¬stable(k, β) do
        WORK ← WORK ∪ DEFER_k^β; β ← β − 1;

while WORK ≠ ∅ do
        remove some (x, θ) from WORK;
        if θ > NEWCRS[x] then
            RV ← NEWCRS;
            if FIND_REC(RV, x, θ) then NEWCRS ← RV;
        if θ ≤ NEWCRS[x] then
            WORK ← WORK ∪ DEFER_x^θ;
            β ← θ − 1;
            while ¬stable(x, β) do
                    WORK ← WORK ∪ DEFER_x^β; β ← β − 1;

CRS ← NEWCRS;
```

Figure 4.3 The incremental recovery state algorithm, invoked when state interval σ of process k becomes stable.

Lemma 4.2 When state interval σ of process k becomes stable, if the current recovery state changes from $\mathbf{R} = [\rho_{**}]$ to $\mathbf{R}' = [\rho'_{**}]$, $\mathbf{R} \prec \mathbf{R}'$, then $\rho'_{k\,k} = \sigma$.

Proof By contradiction. Suppose the new current recovery state $\mathbf{R}'$ has $\rho'_{k\,k} \neq \sigma$. Because only one state interval has become stable since $\mathbf{R}$ was the current recovery state, and because process k in the new current recovery state $\mathbf{R}'$ is not in state interval σ, then all process state intervals in $\mathbf{R}'$ must have been stable before state interval σ of process k became stable. Thus, system state $\mathbf{R}'$ must have been recoverable before state interval σ of process k became stable. Since $\mathbf{R} \prec \mathbf{R}'$, then $\mathbf{R}'$ must have been the current recovery state before state interval σ of process k became stable, contradicting the assumption that $\mathbf{R}$ was the original current recovery state. Thus, if the current recovery state has changed, then $\rho'_{k\,k} = \sigma$. $\square$

Lemma 4.3 When state interval σ of process k becomes stable, if the initial call to *FIND_REC* by the recovery state algorithm returns **false**, then the current recovery state of the system has not changed.

Proof By Lemma 4.2, if the current recovery state changes from $\mathbf{R} = [\rho_{**}]$ to $\mathbf{R}' = [\rho'_{**}]$ when state interval σ of process k becomes stable, then $\rho'_{k\,k} = \sigma$. However, a **false** return from *FIND_REC* indicates that *no* recoverable system state $\mathbf{R}'$ exists with $\rho'_{k\,k} \geq \sigma$, such that $\mathbf{R} \prec \mathbf{R}'$. Therefore, the current recovery state cannot have changed. $\square$

Associated with each state interval β of each process i that is in advance of the known current recovery state is a set DEFER_i^β, which records the identification of any stable process state intervals that depend on state interval β of process i. That is, if the current recovery state of the system is $\mathbf{R} = [\rho_{**}]$, then for all i and β such that $\beta > \rho_{i\,i}$, DEFER_i^β records the set of stable process state intervals that have β in component i of their dependency vector. All *DEFER* sets are initialized to the empty set when the corresponding process is created. If *FIND_REC* returns **false** when some new process state interval becomes stable, that state interval is entered in at least one *DEFER* set. The algorithm uses these *DEFER* sets to limit the search space for the new current recovery state on its future executions.

If the initial call to *FIND_REC* by the recovery state algorithm returns **true**, a new greater recoverable system state has been found. Additional calls to *FIND_REC* are used to search for any other recoverable system states that exist that are greater than the one returned by the previous call to *FIND_REC*. The new current recovery state of the system is the state returned by the last call to *FIND_REC* that returned **true**. The algorithm uses a result of the following lemma to limit the number of calls to *FIND_REC* required.

> **Lemma 4.4** Let $\mathbf{R} = [\rho_{**}]$ be the existing current recovery state of the system, and then let state interval σ of process k become stable. For any stable state interval θ of any process x such that $\theta > \rho_{x\,x}$, no recoverable system state $\mathbf{R}' = [\rho'_{**}]$ exists with $\rho'_{x\,x} \geq \theta$ if state interval θ of process x does not depend on state interval σ of process k by the transitive closure of the transferred dependency relation.

Proof Since state interval θ of process x is in advance of the old current recovery state, it could not be made part of any recoverable system state $\mathbf{R}'$ before state interval σ of process k became stable. If it does not depend on state interval σ of process k by the transitive closure of the transferred dependency relation, then the fact that state interval σ has become stable cannot affect this.

Let δ be the maximum index of any state interval of process k that state interval θ of process x is related to by this transitive closure. Clearly, any new recoverable system state $\mathbf{R}' \neq \mathbf{R}$ that now exists with $\rho'_{x\,x} \geq \theta$ must have $\rho'_{k\,k} \geq \delta$, by Definitions 4.1 and 2.4, and since no component of any dependency vector decreases through execution of the process. If $\delta > \sigma$, then system state $\mathbf{R}'$ was recoverable before state interval σ became stable, contradicting the assumption that $\theta > \rho_{k\,k}$. Likewise, if $\delta < \sigma$, then $\mathbf{R}'$ cannot exist now if it did not exist before state interval σ of process k became stable, since state interval δ must have been stable before state interval σ became stable. Since both cases lead to a contradiction, no such recoverable system state $\mathbf{R}'$ can now exist without this dependency through the transitive closure. $\square$

The **while** loop of the recovery state algorithm uses the *DEFER* sets to traverse the transitive closure of the transferred dependency relation backward from state interval σ of process k. Each state interval θ of some process x visited on this traversal depends on state interval σ of process k by this transitive closure. That is, either state interval θ of process x has a transferred dependency on state interval σ of process k, or it has a transferred dependency on some other process state interval that depends on state interval σ of process k by this transitive closure. The traversal uses the set *WORK* to record those process state intervals from which the traversal must still be performed. When *WORK* has been emptied, the new current recovery state has been found and is copied back to *CRS*.

During this traversal, any dependency along which no more **true** results from *FIND_REC* can be obtained is not traversed further. If the state interval θ of process x that is being considered is in advance of the maximum known recoverable system state, *FIND_REC* is called to search for a new greater recoverable system state in which process x has advanced at least to state interval θ. If no such recoverable system state exists, the traversal from this state interval is not continued, since *FIND_REC* will return **false** for all other state intervals that depend on state interval θ of process x by this transitive closure.

> **Lemma 4.5** If state interval β of process i depends on state interval θ of process x by the transitive closure of the transferred dependency relation, and if no recoverable system state $\mathbf{R} = [\rho_{**}]$ exists with $\rho_{x\,x} \geq \theta$, then no recoverable system state $\mathbf{R}' = [\rho'_{**}]$ exists with $\rho'_{i\,i} \geq \beta$.

Proof This follows directly from the definition of a transferred dependency in Definition 4.1. Either state interval β of process i has a transferred dependency on state interval θ of process x, or it has a transferred dependency on some other process state interval that depends on state interval θ of process x by this transitive closure. By this dependency, any such recoverable system state $\mathbf{R}'$ that exists must also have $\rho'_{x\,x} \geq \theta$, but no such recoverable system state exists since $\mathbf{R}$ does not exist. Therefore, $\mathbf{R}'$ cannot exist. $\square$

Theorem 4.2 If the incremental recovery state algorithm is executed each time any state interval σ of any process k becomes stable, it will complete each execution with $CRS[i] = \rho'_{i\,i}$, for all i, where $\mathbf{R}' = [\rho'_{*\,*}]$ is the new current recovery state of the system.

Proof The theorem holds before the system begins execution since $CRS[i]$ is initialized to 0 when each process i is created. Likewise, if any new process i is created during execution of the system, it is correctly added to the current recovery state by setting $CRS[i] = 0$.

When some state interval σ of some process k becomes stable, if the initial call to *FIND_REC* returns **false**, the current recovery state remains unchanged, by Lemma 4.3. In this case, the recovery state algorithm correctly leaves the contents of *CRS* unchanged.

If this call to *FIND_REC* returns **true** instead, the current recovery state has advanced as a result of this new state interval becoming stable. Let $\mathbf{R} = [\rho_{*\,*}]$ be the old current recovery state before state interval σ of process k became stable, and let $\mathbf{D} = [\delta_{*\,*}]$ be the system state returned by this call to *FIND_REC*. Then $\mathbf{R} \prec \mathbf{D}$, by Lemma 4.1. Although the system state $\mathbf{D}$ may be less than the new current recovery state $\mathbf{R}'$, because the set of recoverable system states forms a lattice, then $\mathbf{R} \prec \mathbf{D} \preceq \mathbf{R}'$.

The **while** loop of the recovery state algorithm finds the new current recovery state by searching forward in the lattice of recoverable system states, without backtracking. This search is performed by traversing backward through the transitive closure of the transferred dependency relation, using the information in the *DEFER* sets. For each state interval θ of each process x examined by this loop, if no recoverable system state exists in which the state of process x has advanced at least to state interval θ, the traversal from this state interval is not continued. By Lemmas 4.4 and 4.5, this loop considers all stable process state intervals for which a new recoverable system state can exist. Thus, at the completion of this loop, the traversal has been completed, and the last recoverable system state found must be new current recovery state. The algorithm finally copies this state from *NEWCRS* to *CRS*. $\square$

4.4.3 An Example

Figure 4.4 shows the execution of a system of three processes in which the incremental recovery state algorithm is used. Each process has been checkpointed in its state interval 0, but no other checkpoints have been written. Also, a total of four messages have been received in the system, but no messages have been logged yet. Thus, only state interval 0 of each process is stable, and the current recovery state of the system is composed of state interval 0 of each process. In the recovery state algorithm, $CRS = \langle 0, 0, 0 \rangle$, and all *DEFER* sets are empty.

If message a from process 2 to process 1 now becomes logged, state interval 1 of process 1 becomes stable, and has a dependency vector of $\langle 1, 1, \perp \rangle$. The recovery state algorithm is executed and calls *FIND_REC* with $\sigma = 1$ and $k = 1$, for state interval 1 of process 1. *FIND_REC* sets RV to $\langle 1, 0, 0 \rangle$ and MAX to $\langle 1, 1, 0 \rangle$. Since $MAX[2] > RV[2]$, a stable state interval $\alpha \geq 1$ of process 2 is needed to make a consistent system state. However, no such state interval of process 2 is currently stable, and *FIND_REC* therefore returns **false**. The recovery state algorithm changes $DEFER_2^1$ to $\{ (1, 1) \}$ and exits, leaving CRS unchanged at $\langle 0, 0, 0 \rangle$.

Next, if process 2 is checkpointed in state interval 2, this state interval becomes stable. Its dependency vector is $\langle 0, 2, 1 \rangle$. The recovery state algorithm calls *FIND_REC*, which sets RV to $\langle 0, 2, 0 \rangle$ and MAX to $\langle 0, 2, 1 \rangle$. Since no state interval $\alpha \geq 1$ of

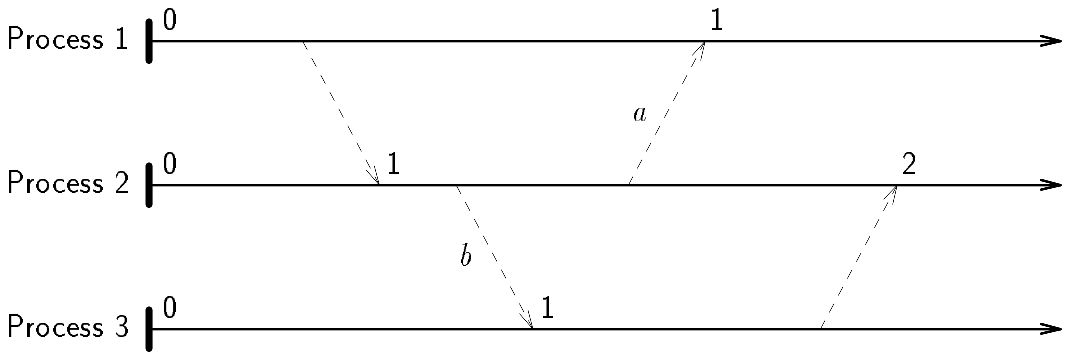

Figure 4.4 An example system execution

process 3 is stable, *FIND_REC* returns **false**. The recovery state algorithm sets $DEFER_3^1$ to $\{(2,2)\}$ and exits, leaving *CRS* unchanged again.

Finally, if message b from process 2 to process 3 becomes logged, state interval 1 of process 3 becomes stable, and has a dependency vector of $\langle \perp, 1, 1 \rangle$. The recovery state algorithm calls *FIND_REC*, which sets *RV* to $\langle 0, 0, 1 \rangle$ and *MAX* to $\langle 0, 1, 1 \rangle$. Since $MAX[2] > RV[2]$, a stable state interval $\alpha \geq 1$ of process 2 is required. State interval 2 of process 2 is the minimum such stable state interval. Using its dependency vector, *RV* and *MAX* are updated, yielding the value $\langle 0, 2, 1 \rangle$ for both. This system state is consistent, and *FIND_REC* returns **true**. The maximum known recoverable system state in *NEWCRS* has then been increased to $\langle 0, 2, 1 \rangle$.

The *WORK* set is initialized to $DEFER_3^1 = \{(2,2)\}$, and the **while** loop of the algorithm begins. When state interval 2 of process 2 is checked, it is not in advance of *NEWCRS*, so the call to *FIND_REC* is skipped. The sets $DEFER_2^2$ and $DEFER_2^1$ are added to *WORK*, making $WORK = \{(1,1)\}$. State interval 1 of process 1 is then checked by the **while** loop. Procedure *FIND_REC* is called, which sets both *RV* and *MAX* to $\langle 1, 2, 1 \rangle$, and therefore returns **true**. The maximum known recoverable system state in *NEWCRS* is updated by this call, to $\langle 1, 2, 1 \rangle$. The set $DEFER_1^1$ is added to *WORK*, but since $DEFER_1^1 = \emptyset$, this leaves *WORK* empty. The **while** loop then terminates, and the value left in $NEWCRS = \langle 1, 2, 1 \rangle$ is copied back to *CRS*. The system state represented by this value of *CRS* is the new current recovery state of the system.

This example illustrates a unique feature of the batch and incremental recovery state algorithms. These algorithms uses checkpointed process state intervals in addition to logged messages in searching for the maximum recoverable system state. In this example with the incremental recovery state algorithm, although only two of the four messages received during the execution of the system have been logged, the current recovery state has advanced due to the checkpoint of process 2. In fact, the two remaining unlogged messages need never be logged, since the current recovery state has advanced beyond their receipt.

4.5 Implementation

This optimistic message logging system has been implemented on a collection of diskless SUN workstations under the V-System (Section 1.3). These workstations are connected by an Ethernet network to a shared SUN network file server. The protocols discussed in Section 4.2 are used for message logging and failure recovery, and the *batch* recovery state algorithm discussed in Section 4.3 is used to find the current recovery state when needed. The *incremental* recovery state algorithm has not yet been implemented.

This implementation is similar to the implementation of sender-based message logging described in Section 3.3. All V-System message-passing operations are supported, and all processes executing as part of the same V-System logical host are treated as a single process in terms of the protocol specification. As with the sender-based message logging implementation, this implementation currently supports the use of message logging by only a single logical host per network node.

Each logical host is identified by a unique *host index*, assigned by the message logging system when the logical host is created. The host index is an integer in the range of 1 through n, where n logical hosts are currently using optimistic message logging. This index serves as the process identifier used in the model of Chapter 2 and in the protocol and algorithm specifications of this chapter. The logical host identifier assigned by the V-System is not used as the index, since logical host identifiers are assigned sparsely from a 16-bit field. Assigning the new host index simplifies using it as a table index within the implementation.

4.5.1 Division of Labor

Each participating node in the system runs a separate *logging server* process, which manages the logging of messages received by that node, and a separate *checkpoint server* process, which manages the recording of checkpoints for that node. Also, a single *recovery server* process, running on the shared network file server of the system, is used to implement the recovery state algorithm. The recovery server process also assigns the host index to each new logical host participating in the message logging protocol. The kernel records received messages in the message buffer shared with the local logging server process, and notifies the logging server when the buffer should be written to stable storage.

Table 4.1 summarizes the amount of executable instructions and data added to the kernel to support optimistic message logging and checkpointing for the SUN-3/60 V kernel configuration. These percentages are relative to the size of that portion of the base kernel without message logging or checkpointing. In addition to several small changes in the internal operation of some existing kernel primitives, only four new primitives to support message logging and three new primitives to support checkpointing were added to the kernel.

4.5.2 Message Log Format

For each node in the system, the message log for all processes executing on that node is stored as a single file on the network file server. Until each message is written to this file, it is saved in a message buffer in the volatile memory of the logging server on the node that received the message. Messages in the message log file and in the message buffer are stored in the same format.

Table 4.1

Size of kernel additions to support optimistic message logging and checkpointing

	Message Logging		Checkpointing		Total	
	Kbytes	Percent	Kbytes	Percent	Kbytes	Percent
Instructions	3.7	4.6	3.5	4.5	7.2	9.1
Data	7.4	3.8	0.0	0.0	7.4	3.8
Total	11.1	4.0	3.5	1.3	14.6	5.3

Each message in the log is stored in a `LoggedMessage` record of a format similar to the `LoggedMessage` records used in the sender-based message logging implementation (Section 3.3.2). The `LoggedMessage` record saves the data of the message (a copy of the network packet received), the host index of the sending logical host, the index of the state interval of that host during which the message was sent, and the index of the state interval started in the receiving logical host by the receipt of this message. To simplify finding the dependency vector for stable process state intervals, the `LoggedMessage` record also contains the value of the dependency vector entry for this sender before it was updated after the receipt of this new message. That is, the receiver saves the current value of its dependency vector entry for this sender in the `LoggedMessage` record, and then sets this entry to the maximum of its current value and the state interval index from which this message was sent. No other record types are used in the message log.

These `LoggedMessage` records are packed into 8-kilobyte blocks, which are the same size as hardware memory pages and data blocks on the file system. As each message is received, a new `LoggedMessage` record is created to store it. The records are created sequentially in a single block, until the next record needed will not fit into the current block. A new block is then allocated, and the new record is created at the beginning of that block. These blocks are saved in memory until they are written to the message log file by the logging server. Each message log block begins with a short header, which describes the extent of the space used within the block. The header also contains the index of the state interval started in this logical host by the receipt of the *first* message in the block, and the complete dependency vector for the state interval started in this logical host by the receipt of the *last* message in the block.

4.5.3 Packet Format

The standard V kernel packet format is used for all messages, but the following three fields have been added to carry the additional information needed to support the logging protocol:

`stateIndex`: This field gives the state interval index of the sender at the time that the message in this packet was sent.

`hostIndex`: This field gives the unique host index assigned to the sending logical host by the recovery server.

These new fields are used only by the message logging protocol, and are ignored by the rest of the kernel. No new V kernel packet types are used.

4.5.4 The Logging Server

As new messages are received by a logical host, they are saved in the *message buffer* by the kernel. This buffer is stored in the volatile memory of the local logging server process, which periodically writes the contents of this buffer to the message log file on the network storage server. The kernel requests the logging server to write this buffer to the file each time a specified number of new messages have been received, and when beginning recovery after any failure in the system. The system can also be configured to trigger this writing after a specified number of new message log blocks in memory have been filled, or after some maximum period of time has elapsed since receiving the earliest message that has not yet been logged. This limits the amount of memory needed to store the message buffer, and helps to limit the amount any process may need to be rolled back to complete recovery from any failure.

After writing these messages from the buffer to the log file, the logging server also sends a summary of this information to the single recovery server process on the file server. For each message logged, this summary gives the state interval index started in this host by the receipt of this message, the host index of the sender, and the value of the receiver's dependency vector entry for this sender before it was updated from the receipt of this new message (that is, the maximum index of any state interval of the same sender on which this process depended before this message was received). This summary is used by the recovery server to determine which new process state intervals have become stable through message logging.

4.5.5 The Checkpoint Server

The checkpoint server operates the same as the checkpoint server used by sender-based message logging described in Section 3.3.6, except that more than one checkpoint for a given host is maintained in the same checkpoint file. In particular, the format of each checkpoint is the same, and the memory of the host is written to the file as either a *full* checkpoint or an *incremental* checkpoint. A full checkpoint writes the entire address space to the file in a single operation, and is used for the first checkpoint of each logical host. Thereafter, an incremental checkpoint is used to only write the pages of the address space that have been modified since the last checkpoint was completed. Most of this address space data is written concurrently with the execution of the host, and the host is then *frozen* while the remainder of the checkpoint data is written.

The checkpoint file begins with a list describing each checkpoint contained in the file. Space in the file to write each section of the checkpoint data is dynamically allocated within the file by the checkpoint server. Although only a portion of the host address space is written to the file on each checkpoint, the entire address space is always contained in the file, since any blocks not written have not been modified since the previous checkpoint of this host. For each checkpoint, a bitmap is also written to the file, describing the address space pages written for this checkpoint. This bitmap enables the complete address space to be reconstructed from the most recent version of each page written on all previous checkpoints in the file. After the checkpoint has been completed, the recovery server is notified of the new process state interval made stable by this checkpoint. This notification also includes the complete current dependency vector of the host at the time of the checkpoint.

4.5.6 The Recovery Server

The recovery server implements a simple form of failure detection for each host that has registered for a host index. The V-System `ReceiveSpecific` operation is used to poll each host periodically. Recovery from any failures detected is performed under the control of the recovery server, using the current recovery state determined by the batch recovery state algorithm. The input to the algorithm comes from the summaries of logged messages sent by the logging server on each host, and from notification of each new checkpoint sent from the checkpoint server on each host.

Failure recovery proceeds as described in Section 4.2.3. Each failed logical host is restored on a separate available node in the system. Each other process that must also be rolled back because it is currently executing in a state interval beyond its state interval in the current recovery state is restored on the node on which the process is currently executing. For each host to be restored, the recovery server sends a request to the checkpoint server on the node on which recovery is to take place. In response, each checkpoint server completes the recovery of that individual host in much the same way as recovery using sender-based message logging is performed, as described in Section 3.3.6. However, the sequence of messages to be replayed from the message log to the recovering host is read directly from the message log file by the local logging server, since all messages are logged by their receiver rather than by their sender.

Again, as the host reexecutes from its checkpointed state, it will resend any messages that it sent during this period before the failure. The treatment of these duplicate messages, though, differs from that used with sender-based message logging. Until the last message being replayed to the recovering host has been received, any messages sent by the host are ignored by the local kernel and are not actually transmitted on the network. With sender-based message logging, these messages must be transmitted, in order to recreate the volatile message log at the sender by the return of the original RSN from the receiver. All messages sent while the list of messages to be replayed is nonempty must be duplicates, since all execution through the end of this state interval (and the beginning of the next) must duplicate execution from before the failure. However, after the last message being replayed has been received and this list is empty, it cannot be known at what point during this last state interval the failure occurred. Therefore, any further messages sent by the recovering host are transmitted normally, and any duplicates are detected at the receivers by the existing duplicate detection method of the underlying system, using the send sequence number (SSN) tagging each message sent.

4.6 Performance

The performance of this optimistic message logging system has been measured on the same network of workstations used to evaluate the performance of sender-based message logging, as described in Section 3.4. Each workstation used is a diskless SUN-3/60, with a 20-megahertz Motorola MC68020 processor, and the network used is a 10 megabit per second Ethernet. The file server used on this network is a SUN-3/160, with a 16-megahertz MC68020 processor and a Fujitsu Eagle disk.

4.6.1 Communication Costs

The measured overhead of using optimistic message logging with common V-System communication operations averaged about 10 percent, with a maximum of about 18 percent, depending on the operation. The operations measured are the same as those reported in Section 3.4.1 for sender-based message logging. These performance measurements for optimistic message logging are shown in Table 4.2. The time for each operation is shown in milliseconds, both with and without using optimistic message logging. The performance is shown for a `Send-Receive-Reply` with no appended data segment and with a 1-kilobyte appended data segment, for a `Send` of a message as a datagram, and for `MoveTo` and `MoveFrom` operations of 1 and 64 kilobytes of data each. All times were measured at the user-level, and show the elapsed time between invoking the operation and its completion. For each operation, the overhead of using optimistic message logging is also shown. These overheads are shown as an increase in the elapsed time required to complete the operation with logging, and as a percentage increase in elapsed running time.

For each packet sent in these operations, all message logging overhead occurs at the host receiving that packet, and is composed mainly of the time needed to allocate space in the message buffer and to copy the new packet into it. The lack of overhead

Table 4.2

Performance of common V-System communication operations using optimistic message logging (milliseconds)

Operation	Message Logging		Overhead	
	With	Without	Time	Percent
`Send-Receive-Reply`	1.6	1.4	.2	14
`Send`(1K)-`Receive-Reply`	3.0	2.7	.3	11
Datagram `Send`	.4	.4	.0	0
`MoveTo`(1K)	3.3	2.8	.5	18
`MoveTo`(64K)	90.0	88.0	2.0	2
`MoveFrom`(1K)	3.3	2.8	.5	18
`MoveFrom`(64K)	89.0	87.0	2.0	2

at the sender is demonstrated by the measured overhead time of 0 milliseconds for sending a message as a datagram. Since no packets are returned by the receiver for this operation, overhead at the receiver does not affect the sending performance. The overhead of copying the packet into the message buffer at the receiver is approximately 27 microseconds for a packet of minimum size, and approximately 162 microseconds for a packet of maximum size.

4.6.2 Checkpointing Costs

The performance of checkpointing using optimistic message logging is similar to its performance using sender-based message logging, described in Section 3.4.2. No difference in the cost of writing the checkpoint file was measurable, as the extra overhead of allocating space in the checkpoint file for multiple versions of the checkpoint was negligible. The additional time required to notify the recovery server of the process state interval made stable by this checkpoint was also negligible, since it involves only a single `Send-Receive-Reply` over the network. The checkpointing performance has been summarized in Table 3.3 in Chapter 3.

4.6.3 Recovery Costs

As with the checkpointing performance, the performance of failure recovery using optimistic message logging is similar to its performance using sender-based message logging, described in Section 3.4.3. The time required to read the checkpoint file and restore the state is the same as that required when using sender-based message logging. These times were summarized in Table 3.4 in Chapter 3. However, the time to collect the messages to be replayed from the log during recovery using optimistic message logging differs from that required by sender-based message logging. With sender-based message logging, these messages must be collected from the separate logs at the hosts that sent them, but with optimistic message logging, they are read directly from the single message log file written by the receiving host. These times vary with the number of messages to be replayed, but are also negligible compared to the time required to read and restore the checkpoint file, and to complete any reexecution based on them. The running time for the recovery state algorithm itself is also negligible.

4.6.4 Application Program Performance

The overall overhead of optimistic message logging on the execution of distributed application programs was measured using the same set of programs that were used in the evaluation of sender-based message logging, reported in Section 3.4.4. Specifically, the performance of the following three programs was measured:

`nqueens`: This program counts the number of solutions to the *n-queens problem* for a given number of queens n.

`tsp`: This program finds the solution to the *traveling salesman problem* for a given map of n cities.

`gauss`: This program performs *Gaussian elimination with partial pivoting* on a given $n \times n$ matrix of floating point numbers.

The set of problems solved with these programs is also the same as that used in evaluating sender-based message logging.

Table 4.3 summarizes the performance of these application programs for all problems in this set. Each entry in this table shows the application program name and the problem size n. The running time in seconds required to solve each problem, both with and without using optimistic message logging, is given. The overhead of using optimistic message logging for each problem is shown in seconds as the difference between its two running times, and as a percentage increase over the running time without logging. The maximum measured overhead was under 4 percent, and for most problems, overhead was much less than 1 percent.

4.7 Related Work

Two other methods to support fault tolerance using optimistic message logging and checkpointing have been published in the literature, although neither of these have

Table 4.3

Performance of the distributed application programs
using optimistic message logging (seconds)

		Message Logging		Overhead	
Program	Size	With	Without	Time	Percent
`nqueens`	12	6.04	6.03	.01	.17
	13	34.99	34.87	.12	.34
	14	210.82	210.73	.09	.04
`tsp`	12	5.21	5.19	.02	.39
	14	16.26	15.94	.32	2.01
	16	844.53	843.45	1.08	.13
`gauss`	100	10.73	10.37	.36	3.47
	200	68.84	67.36	1.48	2.20
	300	217.40	215.86	1.54	.71

been implemented. The work in this thesis has been partially motivated by Strom and Yemini's Optimistic Recovery system [Strom85], and recently Sistla and Welch have proposed a new method using optimistic message logging [Sistla89], based in part on some aspects of both Strom and Yemini's system and on an earlier version of this work [Johnson88]. The system in this chapter is unique among these in that it always finds the *maximum* recoverable system state. Although these other systems occasionally checkpoint processes as this system does, they do not consider the checkpoints in finding the current recovery state of the system. The two recovery state algorithms presented in this chapter include these checkpoints and logged messages in the search for the current recovery state, and thus may find recoverable system states that these other algorithms do not. Also, by including these checkpointed process states, some messages received by a process before it was checkpointed may not need to be logged, as illustrated in the example of Section 4.4.3.

In Strom and Yemini's Optimistic Recovery system [Strom85], each message sent is tagged with a *transitive* dependency vector, which has size proportional to the number of processes in the system. Also, each process is required to locally maintain its knowledge of the message logging progress of each other process in a *log vector*, which is either periodically broadcast by each process or appended to each message sent. The new system presented in this chapter tags each message only with the current state interval index of the sender. Information equivalent to the log vector is maintained by the recovery state algorithm, but uses no additional communication beyond that already required to log each message. Although communication of the transitive dependency vector and the log vector allows control of recovery to be less centralized and may result in faster commitment of output to the outside world, this additional communication may add significantly to the failure-free overhead of the system.

Sistla and Welch have proposed two alternative recovery algorithms based on optimistic message logging [Sistla89]. One algorithm tags each message sent with a transitive dependency vector as in Strom and Yemini's system, whereas the other algorithm tags each message only with the sender's current state interval index as in the system of this chapter. To find the current recovery state, each process sends information about its message logging progress to all other processes, after which their second algorithm also exchanges additional messages, essentially to distribute the complete transitive dependency information. Each process then locally performs the same computation to find the current recovery state. This results in $O(n^2)$ messages for the first algorithm, and $O(n^3)$ messages for the second, where n is the number of processes in the system. In contrast, the recovery state algorithms presented here require no additional communication beyond that necessary to log each message on stable storage. Again, the additional communication in their system allows control of recovery to be less centralized. However, the current recovery state must be determined frequently, so that output to the outside world can be committed quickly. The increased communication in Sistla and Welch's algorithms may substantially increase the failure-free overhead of the system.

Some mechanisms used to support atomic actions (Section 1.4.3) resemble the use of optimistic message logging and checkpointing. Logging on stable storage is used to record state changes of modified objects during the execution of a transaction. Typically, the entire state of each object is recorded, although *logical logging* [Bernstein87] records only the names of operations performed and their parameters, such that they can be reexecuted during recovery, much the same as reexecuting processes based on logged messages. This logging may proceed asynchronously during the execution of the transaction, but must be forced to stable storage before the transaction can commit. This is similar to the operation of optimistic message logging. Before the transaction can commit, though, additional *synchronous* logging is required to ensure the atomicity of the commit protocol, which is not necessary with message logging and checkpointing methods. However, this extra logging can be reduced through the use of special commit protocols, such as the *Presumed Commit* and *Presumed Abort* protocols [Mohan86].

4.8 Summary

This chapter has presented a new transparent *optimistic* message logging system that guarantees to find the *maximum* possible recoverable state in the system. This is achieved by utilizing all logged messages *and* checkpoints in forming recoverable system states, using the model presented in Chapter 2. Previous systems using optimistic message logging and checkpointing [Strom85, Sistla89] have considered only logged messages in forming recoverable system states, and thus may not find the maximum possible state. Also, but utilizing the checkpointed states, some messages received by a process before it was checkpointed may not need to be logged. Furthermore, this system adds less communication overhead to the system than do these previous methods.

Each message sent by a process includes the current state interval index of the sender. No other communication overhead is incurred beyond that necessary to log each message and to record the process checkpoints. The current recovery state of the system is determined using a *recovery state algorithm*, and two alternative algorithms were presented in this chapter: a *batch* algorithm and an *incremental* algorithm. The batch algorithm finds the current recovery state "from scratch" each time it is invoked, considering all process state intervals that are currently stable. It uses no internal state information saved from previous executions of the algorithm. On the other hand, the incremental algorithm begins its search for the current recovery state with the previously known value, and utilizes information saved from its previous executions to shorten its search. It must be executed once for *each* new process state interval that becomes stable, and updates the current recovery state based on the fact that a single new process state interval has become stable. Each algorithm runs centrally on the shared network file server on which all logged messages and checkpoints are recorded, reducing the network communication requirements of the algorithm. These algorithms are restartable if the file server fails, since all information used by them has been recorded on stable storage.

This optimistic message logging system has been completely implemented under the V-System, using the batch recovery state algorithm. Measured on a network of SUN-3/60 workstations, the overhead of this system on individual communication operations averaged only 10 percent, ranging from about 18 percent to 2 percent, for different operations. The overhead on distributed application programs using this system ranged from a maximum of under 4 percent to much less than 1 percent.

Chapter 5
Conclusion

This thesis has examined the class of fault-tolerance methods using *message logging and checkpointing* in distributed systems. These methods are transparent and general-purpose, and can be used by application programs without the need for special programming. This final chapter summarizes the research contributions of this thesis, and explores related avenues for future work.

5.1 Thesis Contributions

The contributions of this thesis fall mainly into three areas: the theoretical framework and model developed in Chapter 2, the new pessimistic message logging system presented in Chapter 3, and the new optimistic message logging system presented in Chapter 4.

5.1.1 Theoretical Framework

The model developed in this framework precisely defines the properties of systems using message logging and checkpointing to provide fault tolerance, and is independent of the particular protocols used in the system. The execution of each process is divided into discrete *state intervals* by the messages received by the process, and the execution within each state interval is assumed to be *deterministic*. To represent process and system states in the model, the *dependency vector* and the *dependency matrix* were introduced. These new structures allow the relevant aspects of process and system states to be concisely represented, and allow mathematical expressions of important properties of these states. A *stable* process state interval is one that can be recreated from information recorded on stable storage, a *consistent* system state is one that could have been observed at some instant during preceding failure-free execution of the system from its initial state, and a *recoverable* system state is a consistent system state in which all process state intervals are stable. This model is the first general model developed specifically to support reasoning about systems using message logging and checkpointing to provide fault tolerance.

An important result of this model is a theorem establishing the existence of a *unique* maximum recoverable system state at all times in any system using message logging and checkpointing. The proof of this theorem relies on other results of the model, namely that the set of system states that have occurred during any single execution of the system forms a *lattice*, with the sets of consistent and recoverable system states as sublattices. At any time, this unique maximum recoverable system state is defined as the *current recovery state*, which is the state to which the system will be recovered if a failure occurs at that time. Other results of the model include sufficient conditions for avoiding the *domino effect* in the system, for releasing output to the outside world, and for removing checkpoints and logged messages from stable storage when they are no longer needed for any possible future recovery in the system. This model was also used in this thesis to prove the correctness of the new message logging methods presented in Chapters 3 and 4.

5.1.2 Pessimistic Message Logging

The *sender-based message logging* system developed in Chapter 3 uses a new *pessimistic* message logging protocol designed to minimize its overhead on the system. This protocol is unique in several ways. First, sender-based message logging logs each message in the local *volatile* memory of the machine from which it was *sent*, which avoids the expense of sending an extra copy of each message either to stable storage on disk or to some other special backup process elsewhere on the network. The protocol also greatly relaxes the restrictions of a pessimistic logging protocol, while still ensuring the same guarantee that any failed process can be recovered individually, without rolling back any processes that did not fail. This is achieved by preventing the process from sending any new messages until all messages it has received have been logged, thus allowing the process to begin execution based on the received message while the logging proceeds concurrently. Previous pessimistic logging systems have instead prevented the process from receiving the original message until it has been logged, thus delaying the execution of the receiver for the entire duration of the logging. Furthermore, sender-based message logging is the first system to recognize that the message data and the order in which the message was received can be logged separately. This allows more flexibility in the design of message logging protocols. Finally, sender-based message logging appears to be the first pessimistic message logging protocol implemented that does not require the use of any specialized hardware to reduce the overhead caused by the logging.

5.1.3 Optimistic Message Logging

The optimistic message logging system developed in Chapter 4 is also unique in several ways. First, it guarantees to always find the *maximum* recoverable system state that currently exists, by utilizing logged messages *and* checkpoints. Previous systems using optimistic message logging and checkpointing have considered only the logged messages, and thus may not find the maximum recoverable system state possible. Furthermore, by utilizing these checkpointed process states, some messages received by a process before it was checkpointed may not need to be logged. This system also requires substantially less additional communication than previous optimistic logging systems. Only a small *constant* amount of information is added to each message, and no additional communication is required beyond that needed to record the messages and checkpoints on stable storage. Previous systems have required information proportional in size to the number of processes in the system to be appended to each message, or have required substantial additional communication between processes in order to determine the current recovery state of the system. Finally, this system appears to be the first complete implementation of a fault-tolerance method using optimistic message logging and checkpointing.

5.2 Suggestions for Future Work

The recovery state algorithms developed in Chapter 4 are only two of many possible algorithms that can be used to determine the current recovery state in a system using message logging and checkpointing. The choice of which algorithm is best for any particular system depends on many factors, such as the number of processes in the system and the frequency of output from the system to the outside world. More work needs to be done to quantify these differences and to evaluate the appropriate algorithm for any given system.

These two recovery state algorithms execute centrally on the shared stable storage server of the system. In order to avoid delays when this server fails, and in order to reduce any possible bottleneck that this centralized facility presents to performance, a *distributed* recovery state algorithm seems to be desirable. However, guaranteeing to always find the maximum possible recoverable system state requires complete knowledge of all stable process state intervals, which may be expensive to distribute to all processes. It may be possible to limit the knowledge required by each process, while still achieving an efficient distributed recovery state algorithm.

Another desirable feature of a recovery state algorithm is to ensure correct and efficient operation in large, perhaps geographically dispersed networks, in which partitions of the network can occur. The current algorithms require coordination between all processes in the system in order to determine the current recovery state and to complete failure recovery. In systems with large numbers of processes, this global coordination is expensive, and if network partitions can occur, such coordination may not always be possible. One solution to this problem is to use a hierarchical decomposition of the system, dividing the system into separate components that are each responsible for their own recovery. Conceptually, each component might be treated as a single process at the next higher level in the hierarchy, thus reducing the number of entities that must be coordinated.

The problem of failure detection is also an area deserving of further study. The simple polling method that is used by most current systems, including the optimistic message logging system developed here, is adequate for small networks with highly reliable communication. However, this polling becomes very expensive in larger networks, and becomes impossible if a network partition occurs. Strom and Yemini's incarnation number in each state interval index partially solves this problem, since processes can then discover a failure later during regular communication by comparing the incarnation number in each message, but this scheme does not correctly handle the occurrence of a network partition. The same failed process could be recovered in each component of the partition, and each would receive the same incarnation number. New theoretical models are also required to prove the correctness of any failure detection mechanism.

Finally, although the assumption of deterministic process execution is reasonable for many applications, some processes may occasionally be nondeterministic for short periods. For example, if two processes share memory, asynchronous scheduling of the processes may cause nondeterministic behavior. In order to support nondeterministic processes with message logging and checkpointing, it must be possible to detect when nondeterministic execution has occurred, and new algorithms and protocols must be developed to correctly restore the system to a consistent state following a failure. New theoretical developments will also be required to reason about systems that allow nondeterministic execution.

Bibliography

[Allchin83] J. E. Allchin and M. S. McKendry. Synchronization and recovery of actions. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 31–44. ACM, August 1983.

[Almes85] Guy T. Almes, Andrew P. Black, and Edward D. Lazowska. The Eden system: A technical review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59, January 1985.

[Anderson78] Thomas Anderson, Peter A. Lee, and Santosh K. Shrivastava. A model of recoverability in multilevel systems. *IEEE Transactions on Software Engineering*, SE-4(6):486–494, November 1978. Also reprinted in *Reliable Computer Systems*, edited by Santosh K. Shrivastava, pages 381–395, Springer-Verlag, New York, 1985.

[Anderson81] T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[Anderson83] Thomas Anderson and John C. Knight. A framework for software fault-tolerance in real-time systems. *IEEE Transactions on Software Engineering*, SE-9(3):355–364, May 1983. Also reprinted in *Reliable Computer Systems*, edited by Santosh K. Shrivastava, pages 358–377, Springer-Verlag, New York, 1985.

[Banino82] J. S. Banino and J. C. Fabre. Distributed coupled actors: A CHORUS proposal for reliability. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 128–134. IEEE Computer Society, October 1982.

[Banino85] J. S. Banino, J. C. Fabre, M. Guillemont, G. Morisset, and M. Rozier. Some fault-tolerant aspects of the CHORUS distributed system. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 430–437. IEEE Computer Society, May 1985.

[Bartlett81] Joel F. Bartlett. A NonStop kernel. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 22–29. ACM, December 1981.

[Bartlett87] Joel Bartlett, Jim Gray, and Bob Horst. Fault tolerance in Tandem computer systems. In *The Evolution of Fault-Tolerant Computing*, edited by A. Avižienis, H. Kopetz, and J.C. Laprie, volume 1 of *Dependable Computing and Fault-Tolerant Systems*, pages 55–76. Springer-Verlag, New York, 1987.

[Bernstein87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Massachusetts, 1987.

[Best81] Eike Best and Brian Randell. A formal model of atomicity in asynchronous systems. *Acta Informatica*, 16(1):93–124, August 1981. Also reprinted in *Reliable Computer Systems*, edited by Santosh K. Shrivastava, pages 266–297, Springer-Verlag, New York, 1985.

[Birman85] Kenneth P. Birman. Replication and fault-tolerance in the ISIS system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 79–86. ACM, December 1985.

[Birman87] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 123–138. ACM, November 1987.

[Birrell82] Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, April 1982.

[Birrell84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Black85] Andrew P. Black. Supporting distributed applications: Experience with Eden. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 181–193. ACM, December 1985.

[Borg83] Anita Borg, Jim Baumbach, and Sam Glazer. A message system supporting fault tolerance. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 90–99. ACM, October 1983.

[Borg89] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[Bourne78] S. R. Bourne. The UNIX shell. *The Bell System Technical Journal*, 57(6):1971–1990, July–August 1978.

[Bubenik89] Rick Bubenik and Willy Zwaenepoel. Performance of optimistic make. In *1989 ACM SIGMETRICS and PERFORMANCE '89 International Conference on Measurement and Modeling of Computer Systems: Proceedings*, pages 39–48. ACM, May 1989.

[Carter85] W. C. Carter. Hardware fault tolerance. In *Resilient Computing Systems*, edited by T. Anderson, chapter 2, pages 11–63. Collins, London, 1985.

[Carter89] John B. Carter and Willy Zwaenepoel. Optimistic implementation of bulk data transfer protocols. In *1989 ACM SIGMETRICS and PERFORMANCE '89 International Conference on Measurement and Modeling of Computer Systems: Proceedings*, pages 61–69. ACM, May 1989.

[Chandy85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[Chandy88] K. Mani Chandy. Theorems on computations of distributed systems. Technical Report Caltech-CS-TR-88-6, California Institute of Technology, April 1988.

[Cheriton83] David R. Cheriton and Willy Zwaenepoel. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 129–140. ACM, October 1983.

[Cheriton84] David R. Cheriton. The V kernel: A software base for distributed systems. *IEEE Software*, 1(2):19–42, April 1984.

[Cheriton85] David R. Cheriton and Willy Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[Cheriton86a] David R. Cheriton. VMTP: A transport protocol for the next generation of communication systems. In *Proceedings of the 1986 SigComm Symposium*, pages 406–415. ACM, August 1986.

[Cheriton86b] David R. Cheriton and Michael Stumm. The multi-satellite star: Structuring parallel computations for a workstation cluster. Technical report, Department of Computer Science, Stanford University, Stanford, California, 1986.

[Cheriton88] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[Clement87] George F. Clement and Paul K. Giloth. Evolution of fault tolerant switching systems in AT&T. In *The Evolution of Fault-Tolerant Computing*, edited by A. Avižienis, H. Kopetz, and J.C. Laprie, volume 1 of *Dependable Computing and Fault-Tolerant Systems*, pages 37–54. Springer-Verlag, New York, 1987.

[Cmelik88] R. F. Cmelik, N. H Gehani, and W. D. Roome. Fault Tolerant Concurrent C: A tool for writing fault tolerant distributed programs. In *The Eighteenth Annual International Symposium on Fault-Tolerant Computing: Digest of Papers*, pages 56–61. IEEE Computer Society, June 1988.

[Cooper84] Eric C. Cooper. Replicated procedure call. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, pages 220–232. ACM, August 1984.

[Cooper85] Eric C. Cooper. Replicated distributed programs. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 63–78. ACM, December 1985.

[Denning76] Peter J. Denning. Fault tolerant operating systems. *ACM Computing Surveys*, 8(4):359–389, December 1976.

[Dimmer85] C. I. Dimmer. The Tandem Non-Stop system. In *Resilient Computing Systems*, edited by T. Anderson, chapter 10, pages 178–196. Collins, London, 1985.

[Fabry73] R. S. Fabry. Dynamic verification of operating system decisions. *Communications of the ACM*, 16(11):659–668, November 1973.

[Finkel87] Raphael Finkel and Udi Manber. DIB—A distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, April 1987.

[Gait85] Jason Gait. A distributed process manager with transparent continuation. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 422–429. IEEE Computer Society, May 1985.

[Gherfal85] Fawzi F. Gherfal and S. Mamrak. An optimistic concurrency control mechanism for an object based distributed system. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 236–245. IEEE Computer Society, May 1985.

[Gray79] J. N. Gray. Notes on database operating systems. In *Operating Systems: An Advanced Course*, edited by R. Bayer, R. M. Graham, and G. Seegmüller, chapter 3. F., pages 393–481. Springer-Verlag, New York, 1979.

[Grit84] D. H. Grit. Towards fault tolerance in a distributed multiprocessor. In *The Fourteenth International Conference on Fault-Tolerant Computing: Digest of Papers*, pages 272–277. IEEE Computer Society, June 1984.

[Haerder83] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[Haskin88] Roger Haskin, Yoni Malachi, Wayne Sawdon, and Gregory Chan. Recovery management in QuickSilver. *ACM Transactions on Computer Systems*, 6(1):82–108, February 1988.

[Hecht76] H. Hecht. Fault-tolerant software for real-time applications. *ACM Computing Surveys*, 8(4):391–407, December 1976.

[Hornig84] David A. Hornig. *Automatic Partitioning and Scheduling on a Network of Personal Computers*. Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania, November 1984.

[Horning74] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell. A program structure for error detection and recovery. In *Operating Systems*, edited by E. Gelenbe and C. Kaiser, volume 16 of *Lecture Notes in Computer Science*, pages 171–187. Springer-Verlag, Berlin, 1974. Also reprinted in *Reliable Computer Systems*, edited by Santosh K. Shrivastava, pages 53–68, Springer-Verlag, New York, 1985.

[Jefferson85] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[Jefferson87] David Jefferson, Brian Beckman, Fred Wieland, Leo Blume, Mike DiLoreto, Phil Hontalas, Pierre Laroche, Kathy Sturdevant, Jack Tupman, Van Warren, John Wedel, Herb Younger, and Steve Bellenot. Distributed simulation and the Time Warp operating system. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 77–93. ACM, November 1987.

[Johnson87] David B. Johnson and Willy Zwaenepoel. Sender-based message logging. In *The Seventeenth Annual International Symposium on Fault-Tolerant Computing: Digest of Papers*, pages 14–19. IEEE Computer Society, June 1987.

[Johnson88] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 171–181. ACM, August 1988.

[Katsuki78] D. Katsuki, E.S. Elsam, W.F. Mann, E.S. Roberts, J.G. Robinson, F.S. Skowronski, and E.W. Wolf. Pluribus—An operational fault-tolerant multiprocessor. *Proceedings of the IEEE*, 66(10):1146–1159, October 1978.

[Koo87] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[Kopetz85a] H. Kopetz. Resilient real-time systems. In *Resilient Computing Systems*, edited by T. Anderson, chapter 5, pages 91–101. Collins, London, 1985.

[Kopetz85b] H. Kopetz and W. Merker. The architecture of MARS. In *The Fifteenth Annual International Symposium on Fault-Tolerant Computing: Digest of Papers*, pages 274–279. IEEE Computer Society, June 1985.

[Kung81] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 2(2):213–226, June 1981.

[Lamport78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lampson79] Butler W. Lampson and Howard E. Sturgis. Crash recovery in a distributed data storage system. Technical report, Xerox Palo Alto Research Center, Palo Alto, California, April 1979.

[Lampson81] Butler W. Lampson. Atomic transactions. In *Distributed Systems: Architecture and Implementation*, edited by B. W. Lampson, M. Paul, and H. J. Siegert, chapter 11, pages 246–265. Springer-Verlag, New York, 1981.

[Lazowska81] Edward D. Lazowska, Henry M. Levy, Guy T. Almes, Michael J. Fischer, Robert J. Fowler, and Stephen C. Vestal. The architecture of the Eden system. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 148–159. ACM, December 1981.

[LeBlanc85] Richard J. LeBlanc and C. Thomas Wilkes. Systems programming with objects and actions. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 132–139. IEEE Computer Society, May 1985.

[Lee78] P. A. Lee. A reconsideration of the recovery block scheme. *The Computer Journal*, 21(4):306–310, November 1978. Also reprinted in *Reliable Computer Systems*, edited by Santosh K. Shrivastava, pages 69–79, Springer-Verlag, New York, 1985.

[Lin86] Frank C. H. Lin and Robert M. Keller. Distributed recovery in applicative systems. In *Proceedings of the 1986 International Conference on Parallel Processing*, edited by Kai Hwang, Steven M.

Jacobs, and Earl E. Swartzlander, pages 405–412. IEEE Computer Society, August 1986.

[Liskov83] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[Liskov87] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler. Implementation of Argus. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 111–122. ACM, November 1987.

[Liskov88] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[Lomet85] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Reliable Computer Systems*, edited by Santosh K. Shrivastava, pages 249–265. Springer-Verlag, New York, 1985. Reprinted from *ACM SIGPLAN Notices*, 12(3):128–137, March 1977.

[Merlin78] P. M. Merlin and B. Randell. State restoration in distributed systems. In *The Eighth International Conference on Fault-Tolerant Computing: Digest of Papers*, pages 129–134. IEEE Computer Society, June 1978. Also reprinted in *Reliable Computer Systems*, edited by Santosh K. Shrivastava, pages 435–447, Springer-Verlag, New York, 1985.

[Metcalfe76] Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[Mohan86] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems*, 11(4):378–396, December 1986.

[Pausch88] Randy Pausch. *Adding Input and Output to the Transactional Model*. Ph.D. thesis, Carnegie-Mellon Univerisity, Pittsburgh, Pennsylvania, August 1988. Also available as Technical Report CMU-CS-88-171, Department of Computer Science, Carnegie-Mellon University, August 1988.

[Powell83] Michael L. Powell and David L. Presotto. Publishing: A reliable broadcast communication mechanism. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 100–109. ACM, October 1983.

[Randell75] Brian Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[Russell77] David L. Russell. Process backup in producer-consumer systems. In *Proceedings of the Sixth Symposium on Operating Systems Principles*, pages 151–157. ACM, November 1977.

[Russell80] David L. Russell. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, SE-6(2):183–194, March 1980.

[Saltzer84] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[Schlichting83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant distributed computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[Schneider87] Fred B. Schneider. The state machine approach: A tutorial. Technical Report TR 86-800, Cornell University, Ithaca, New York, June 1987. To appear in *Proceedings of a Workshop on Fault-Tolerant Distributed Computing*, *Lecture Notes in Computer Science* series, Springer-Verlag, New York.

[Shoch80] John F. Shoch and Jon A. Hupp. Measured performance of an Ethernet local network. *Communications of the ACM*, 23(12):711–721, December 1980.

[Shoch82] John F. Shoch and Jon A. Hupp. The “worm” programs—Early experience with a distributed computation. *Communications of the ACM*, 25(3):172–180, March 1982.

[Shrivastava82] S. K. Shrivastava. A dependency, commitment and recovery model for atomic actions. In *Proceedings of the Second Symposium on*

Reliability in Distributed Software and Database Systems, pages 112–119. IEEE Computer Society, July 1982. Also reprinted in *Reliable Computer Systems*, edited by Santosh K. Shrivastava, pages 485–497, Springer-Verlag, New York, 1985.

[Siewiorek86] D. Siewiorek. Architecture of fault-tolerant computers. In *Fault-Tolerant Computing: Theory and Techniques*, edited by Dhiraj K. Pradhan, volume 2, chapter 6, pages 417–466. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[Sistla89] A. Prasad Sistla and Jennifer L. Welch. Efficient distributed recovery using message logging. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing.* ACM, August 1989.

[Skeen83] Dale Skeen and Michael Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering*, SE-9(3):219–228, May 1983.

[Spector85a] Alfred Z. Spector, Jacob Butcher, Dean S. Daniels, Daniel J. Duchamp, Jeffrey L. Eppinger, Charles E. Fineman, Abdelsalam Heddaya, and Peter M. Schwarz. Support for distributed transactions in the TABS prototype. *IEEE Transactions on Software Engineering*, SE-11(6):520–530, June 1985.

[Spector85b] Alfred Z. Spector, Dean Daniels, Daniel Duchamp, Jeffrey L. Eppinger, and Randy Pausch. Distributed transactions for reliable systems. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 127–146. ACM, December 1985.

[Spector86] Alfred Z. Spector, Joshua J. Bloch, Dean S. Daniels, Richard P. Draves, Dan Duchamp, Jeffrey L. Eppinger, Sherri G. Menees, and Dean S. Thompson. The Camelot project. Technical Report CMU-CS-86-166, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, November 1986.

[Spector87] Alfred Z. Spector. Distributed transaction processing and the Camelot system. In *Distributed Operating Systems: Theory and Practice*, edited by Yakup Paker, Jean-Pierre Banatre, and Müslim Bozyiğit, volume 28 of *NATO Advanced Science Institute Series F: Computer and Systems Sciences*, pages 331–353. Springer-Verlag, Berlin, 1987. Also available as Technical Report CMU-CS-87-100, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, January 1987.

[Stanford86] V-System Development Group, Computer Systems Laboratory, Departments of Computer Science and Electrical Engineering, Stanford University, Stanford, California. *V-System 6.0 Reference Manual*, May 1986.

[Strom85] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[Strom87] Robert Strom and Shaula Yemini. Synthesizing distributed and parallel programs through optimistic transformations. In *Current Advances in Distributed Computing and Communications*, edited by Yechiam Yemini, pages 234–256. Computer Science Press, Rockville, Maryland, 1987. Also available as Research Report RC 10797, IBM T. J. Watson Research Center, Yorktown Heights, New York, July 1984.

[Strom88] Robert E. Strom, David F. Bacon, and Shaula A. Yemini. Volatile logging in n-fault-tolerant distributed systems. In *The Eighteenth Annual International Symposium on Fault-Tolerant Computing: Digest of Papers*, pages 44–49. IEEE Computer Society, June 1988.

[Svobodova84] Liba Svobodova. File servers for network-based distributed systems. *ACM Computing Surveys*, 16(4):353–398, December 1984.

[Theimer85] Marvin N. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the V-System. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 2–12. ACM, December 1985.

[Vinter86] Stephen Vinter, Krithi Ramamritham, and David Stemple. Recoverable actions in Gutenberg. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 242–249. IEEE Computer Society, May 1986.

[Walker83] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In

Proceedings of the Ninth ACM Symposium on Operating Systems Principles, pages 49–70. ACM, October 1983.

[Wensley78] John H. Wensley, Leslie Lamport, Jack Goldberg, Milton W. Green, Karl N. Levitt, P. M. Melliar-Smith, Robert E. Shostak, and Charles B. Weinstock. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, October 1978.

[Wood85] W. G. Wood. Recovery control of communicating processes in a distributed system. In *Reliable Computer Systems*, edited by Santosh K. Shrivastava, pages 448–484. Springer-Verlag, New York, 1985.

[Yang85] Xiao-Zong Yang and Gary York. Fault recovery of triplicated software on the Intel iAPX 432. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 438–443. IEEE Computer Society, May 1985.

[Zwaenepoel84] Willy Zwaenepoel. *Message Passing on a Local Network*. Ph.D. thesis, Stanford University, Stanford, California, October 1984. Also available as Technical Report STAN-CS-85-1083, Department of Computer Science, Stanford University, October 1985.

[Zwaenepoel85] Willy Zwaenepoel. Protocols for large data transfers over local area networks. In *Proceedings of the 9th Data Communications Symposium*, pages 22–32. IEEE Computer Society, September 1985. Also reprinted in *Advances in Local Area Networks*, edited by Karl Kümmerle, John O. Limb, and Fouad A. Tobagi, Frontiers in Communications series, chapter 33, pages 560–573, IEEE Press, New York, 1987.