Sender-Based Message Logging: Low-Overhead Fault Tolerance for Distributed Systems

David B. Johnson

Rice COMP TR86-43

December 1986

Department of Computer Science
Rice University
P.O. Box 1892
Houston, Texas 77251-1892

(713) 527-8101

Abstract

This paper proposes the development of a new low-overhead fault-tolerance mechanism called *sender-based message logging* that can allow processes in a distributed system to continue execution in spite of faults occurring in the system. Unlike many previous fault-tolerance mechanisms, sender-based message logging can be applied transparently to existing distributed applications and does not require the use of specialized hardware not normally found in existing distributed systems. This new mechanism also attempts to minimize any performance degradation on the system resulting from the provision of fault tolerance.

With sender-based message logging, each process is individually checkpointed on occasion, and all messages that a process receives are logged in a message log *in the sender's local volatile memory*. Then, if a process fails, it can be restarted from its most recent checkpoint, and the messages that have been logged for it since that checkpoint can be resent to it, thus restoring its internal state to the point at which the failure occurred. The frequency of checkpointing can be tuned to balance its expense against the time needed for recovery, but the logging must be done for each message that each process receives, placing a continuous overhead on the operation of the system even when no faults occur. By storing the message log in the sender's local volatile memory, this overhead can be minimized, and the message log can be maintained inexpensively without adding extra network communication to the system.

Contents

1 Introduction

This paper proposes the development of a new low-overhead fault-tolerance mechanism that can allow processes in a distributed system to continue execution in spite of failures within the system. This new mechanism, called *sender-based message logging*, can be applied transparently to existing distributed applications, including both simple server processes and complex collections of cooperating processes across many machines. Additionally, this new mechanism does not require the use of any special hardware not normally found in existing distributed systems and attempts to minimize any negative affect of the fault tolerance on the performance of distributed applications.

Although a number of mechanisms for achieving fault tolerance in distributed systems have been proposed by others, many of these have encountered problems and have not been entirely successful [Serlin85]. Many of the current fault-tolerance mechanisms create a large overhead on the system in both increased computation and communication costs, causing applications processes to run significantly slower than without the fault tolerance. Other fault-tolerance mechanisms restrict the types of applications that can be run, such as requiring them to be structured as a series of atomic transactions. Also, specialized processor communications hardware is required by many of these mechanisms, making them expensive and difficult to apply to existing computer systems.

Distributed systems present many challenges to the design and implementation of fault-tolerance mechanisms over more conventional centralized systems. The possible failure modes for a distributed system are more complex than for a centralized computer, making the provision of fault tolerance more difficult, yet more important. Since the independent processors of a distributed system may fail at different times, the possibility of a *partial failure* exists, where only some of the processors cooperating in a computation fail, leaving the others to attempt to continue from a possibly inconsistent state. Also, since the time required for communication between the processors in a distributed system can be significant, it may be difficult and expensive to determine the global state of the entire system, making recovery of a consistent state after a failure more difficult. Additionally, since the network hardware used in such systems typically provides only best-effort attempts at message delivery, any required reliability of communication between processors must be implemented above the basic network level [Tanenbaum81, Chang84]. The added complexity of the network protocols needed for this reliability will make any new communication required for the provision of fault tolerance more expensive.

A number of researchers are currently investigating techniques for utilizing more of the potential inherent in distributed systems, such as the ability to combine the computing power of many processors in the system toward the solution of a single problem. The provision of inexpensive fault tolerance in these distributed systems can help make such ambitious applications more manageable. For example, in the Computer Science Department at Rice University, we are currently studying the parallel execution of programs on a collection of personal workstations connected by a local network [Almes85b]. We have so far developed a number of prototype applications programs, including a parallel implementation of the UNIX[1] *make* facility and several methods of parallel compilation [Johnson85, Boehm86]. These prototypes are good examples of the types of applications that can benefit from the low-overhead fault-tolerance mechanism proposed here.

The type of fault tolerance considered by this paper is the survival of process execution despite faults occurring within a distributed system. The problems of maintaining the consistency and availability of static data such as file systems and databases are not directly addressed by this paper; these issues are well represented by other current and previous research [Gray79, Haerder83,

[1] UNIX is a trademark of AT&T Bell Laboratories.

Lampson79, Svobodova84]. Also, the constraints of real-time systems and applications are not considered in detail by this paper since these systems require special treatment that does not generalize well to other classes of systems. The specific methods for achieving fault tolerance in real-time systems and applications have been studied by others [Hecht76, Kopetz85a].

Section 2 of this paper provides an introduction to the different concepts and terminology that are used in the field of fault-tolerance research. The goals for this research are then presented in Section 3, followed by an overview of the sender-based message logging mechanism in Section 4. Section 5 describes the basic protocols to be used in sender-based message logging, and Sections 6 and 7 give some ideas on the creation of a prototype implementation of these protocols and the problems of testing their correctness and measuring and evaluating their performance. In Section 8, some issues that are related to the subject of this research and which might be considered within its scope are discussed. An extensive survey of other software solutions to fault tolerance is presented in Section 9, giving examples of existing systems for each mechanism discussed and a comparison of each mechanism with the sender-based message logging mechanism proposed here. Finally, conclusions are given in Section 10, including a discussion of the major contributions expected from this research.

2 Concepts and Terminology

Fault tolerance is a broad area covering many issues. Like any field of specialization, a large collection of terminology has evolved to describe the concepts and ideas used in this area. Unfortunately, there is much confusion in the literature and in the computing industry about the definitions of these terms. For example, Anderson at the University of Newcastle upon Tyne has commented:

> A continuing source of difficulty in discussing issues relating to the reliability of systems and the provision of fault tolerance is the absence of any agreed terminology for the relevant concepts. Indeed, the concepts themselves are perhaps a little more elusive than many people realise. [Anderson85, page 1]

To help to resolve these conflicts and standardize usage in the future, IFIP Working Group 10.4 on Reliable Computing and Fault Tolerance and the IEEE Computer Society Technical Committee on Fault-Tolerant Computing have attempted "to propose clear and widely acceptable definitions for some basic concepts" of fault-tolerant systems [Laprie85]. Following their work, Sections 2.1 and 2.2 present some definitions from this area that will be useful in the remainder of this paper.[2] Section 2.3 then discusses the problem of restoring a consistent state in a distributed system following a failure and describes a phenomenon known as the domino effect that must be avoided by any distributed fault-tolerance mechanism. Finally, in Section 2.4, a model of processor failure in a computer system is presented.

2.1 Failures, Faults, and Errors

Every system is assumed to have some form of agreed *service specification* defining the service expected from the system by its users, although this specification can be informal. When the service delivered by the system differs from this specification, a *failure* has occurred. The cause of a failure,

[2]Most of this material has also been covered by others [Anderson81, LeLann81, Randell79], although their presentations differ somewhat from the proposed IFIP/IEEE definitions.

whether mechanical or algorithmic, is called a *fault*. Examples of faults include *physical faults* such as short-circuits, overheating, and power failures, and *human-made faults* such as programming mistakes and violations of defined operating procedures.

When a fault occurs, the internal state of the system may be changed; any part of the system state that is liable to lead to a failure is called an *error*. An error is termed a *latent error* immediately after the fault that causes it, but becomes an *effective error* when it is *activated*. A failure occurs when an effective error changes the behavior of the system such that it no longer meets its service specification. In other words, an error is the effect in the system of a fault, and a failure is the effect of an error on the service provided by the system. These concepts are recursive in the sense that a failure in a subsystem is a fault in the enclosing system. For example, if a memory word in a computer changes value due to some electromagnetic disturbance, a latent error is created. The error does not become effective until it is activated by being read by some program. If this error then causes the program to violate its specifications, a failure has occurred. If this computer is part of a larger system such as a distributed system, this failure of the computer is a fault within the system as a whole.

The frequency and duration of system failures can have a large affect on the user's perception of the *dependability* of the system, where system dependability is a measure of the degree to which the users of the system can justifiably rely on the service given. From the point of view of its users, a system can be in either of two states. When the service delivered by the system meets its service specification, the system is said to be in *service accomplishment*, but when delivered service deviates from this specification, the system enters the *service interruption* state. As failures and subsequent recoveries occur, the system alternates between these two states. The time spent in each of these states affects the measured dependability of the system. The *reliability* of a system is a measure of the continuous service accomplishment from some reference starting time. This is equivalent to the elapsed time from this starting point until a failure occurs and is commonly expressed as the *Mean Time Between Failures* or *MTBF*. System *availability* is a measure of the service accomplishment with respect to service interruption over some interval, or the fraction of the time that the system meets its specifications. Reliability and availability are the two main measures of system dependability.

2.2 Fault Tolerance

Several approaches to improving the dependability of a system are possible. One way is to make the system very easy to repair so that the duration of service interruption periods can be minimized, thus improving the availability measure of system dependability. A complimentary approach is to construct the system of sufficient quality such that failures are infrequent, thus lengthening the periods of service accomplishment and also improving system availability. Since lessening the likelihood of system failures in this way would also increase the reliability measure of system dependability, this approach has added benefit.

It is this approach of increasing system dependability through decreasing the frequency of failures on which *fault tolerance* concentrates. Fault tolerance is the provision, through redundancy, of service complying with the system's service specification in spite of faults having occurred or occurring within the system. Since a fault may create an error in the system, the job of fault tolerance is to detect and either to repair or to compensate for these errors before they can cause the failure of the entire system.

Fault tolerance is carried out through *error processing*, which is actually composed of two individual phases called *latent error processing* and *effective error processing*. The goal of latent

error processing is to keep existing latent errors within the system from becoming effective and is typically achieved by reconfiguring the system so that the error will not be activated. Since fault tolerance is concerned with allowing a system to continue service accomplishment in spite of failures within the system, and since latent error processing deals with errors before they can cause failures, this phase of error processing is used little in fault-tolerance mechanisms. With effective error processing, however, the goal is to transform an effective error back into a latent state before a failure can occur. It is this idea which forms the basis of most fault-tolerance mechanisms.

The effective error processing phase of fault-tolerance can be achieved either through *error compensation* or through *error recovery*. Error compensation attempts to allow the system to continue service accomplishment with the effective error still present in the internal system state. For example, it is common for magnetic disks on computer systems to use error correction codes to allow data to be read from the disk even if there is a small error on the portion of the disk containing the needed data. Conversely, when error recovery is used, the erroneous internal state is replaced by some error-free state, and the computation is continued from this substituted state. Extending the example above, if the disk became heavily damaged, the error correction codes would no longer be able to compensate for the error. An error recovery approach to solving this problem would be to restore a tape backup copy of the failed disk onto a new disk.

During error recovery, any error-free state can be substituted for the system state that is in error. One possible choice for this is to use some state that has been saved that the system was in before the error became effective. Because this method attempts to simulate a backward movement through time by substituting some previously occupied state for the current erroneous state, this method is called *backward error recovery*. Alternatively, the *forward error recovery* method constructs some new state that the system has never been in before and substitutes this for the current state that is in error. Since creation of this new error-free state requires specific knowledge of the system and the type of error that occurred, forward error recovery can not be used as the basis for a general-purpose error recovery mechanism for unanticipated faults. For this reason, general-purpose fault-tolerance mechanisms use backward error recovery, exclusively.

Since backward error recovery restores the system to a state that it was in before the error occurred, it is possible that the same error will occur again in the same way after recovery. An error that has a low likelihood of recurring is referred to as *volatile*; all other errors are called *solid*. For backward error recovery to be successful, it is important for the error that necessitated the recovery to be volatile. Fortunately, most errors in typical computer systems can be considered volatile when appropriate system reconfiguration is used during recovery. For example, if a processor in a distributed system fails due to some faulty component, an effective error is created in the system as a whole, assuming the processor was in use at the time. If the programs that were running on the failed processor are restarted from saved states on another processor in the system, it is unlikely that the new processor will fail in the same way that the original one failed.

2.3 The Domino Effect

After a failure, it is important to restore the system to a consistent state. If the system consists of only a single independent process, the state of this process can simply be saved periodically, and any one of these saved states can be used to restart the process following a failure. In a system of many cooperating processes, however, the problem becomes more difficult since state changes in one process can be reflected in the states of many other processes in the system. For example, when one process sends a message to another process, the states of these two processes are then dependent on each other. If it is necessary to recover the receiver because of a failure, and if the only saved

state for the receiver occurred before the message was received, then it may also be necessary to roll back the state of the sender. Also, if it is necessary to recover the sender to a point before the message was sent, it may be necessary to roll back the state of the receiver as well, since the received data may no longer be valid. In some situations, these dependencies between the states of different processes can lead to a phenomenon called the *domino effect* [Randell75, Russell80, Russell77] when trying to recover from the failure of a single process. This term describes a potentially unbounded series of process rollbacks that may be required to achieve a consistent state following a failure. In order to guarantee progress in spite of failures in the system, any fault-tolerance mechanism must avoid the domino effect.

To illustrate how the domino effect can occur, Figure 1 shows the execution of two cooperating processes, P_1 and P_2. The two horizontal lines represent the execution of the two processes, with time progressing from left to right. The dashed lines between the the two processes show messages sent from one process to the other (the direction of the message exchange is unimportant). The state of each process has been independently saved as a checkpoint at each time marked with a heavy tick mark. If process P_1 fails, recovery is easy since its most recent saved state will still be consistent with current state of P_2. Conversely, if process P_2 fails, its most recent saved state precedes the last message exchange with P_1, leaving the states of the two processes inconsistent. It is therefore necessary in this case to roll back the state of P_1 to a point before the message exchange, but this saved state precedes other communication with P_2. In this example, the failure of process P_2 will produce a period of uncontrolled rolling back of both processes that will ultimately result in both processes being restored to their initial states.

2.4 A Model of Processor Failure

In real computing systems, a processor may fail in many different ways. Since a ***failure*** of one processing unit in a system of many processors is only a ***fault*** in the system as a whole, fault-tolerance mechanisms can be designed to allow the system to continue service accomplishment after such a failure. In order to be able to reason about the consequences of a processor failure and to devise fault-tolerant protocols to compensate for these failures, it is useful to have a model to work with of how a processor fails. A common and useful model for processor failures is that of a *fail-stop processor* [Schlichting83]. Most of the fault-tolerance mechanisms described in this paper,

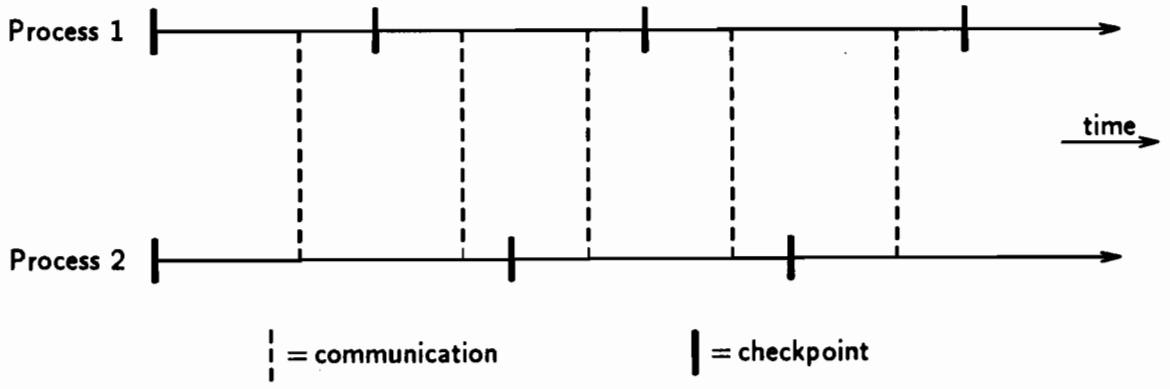

FIGURE 1: The domino effect: If process 2 fails, both processes ultimately must be rolled back to their initial states.

including the sender-based message logging mechanism being proposed here, assume this model of processor failure to some degree.

Under normal operation, a fail-stop processor behaves the same as a normal equivalent processor without the fail-stop property. But, when a fault that would lead to a failure occurs in a fail-stop processor, the processor simply halts. That is, the only way in which a fail-stop processor can provide service that is not in agreement with its service specification is to stop executing instructions completely. It never continues executing after such a fault or produces incorrect results.

Although the fail-stop model is only an abstraction of real processor failures, it does reflect fairly accurately many common ways in which modern processing units actually fail. Modern computer hardware employs a wide variety of error checking mechanisms such as CRC codes on solid-state memory and parity on bus transfers. These techniques, coupled with the advances in VLSI manufacturing methods in the last decade, greatly reduce the likelihood that the processor could produce an undetectably incorrect result for some machine instruction that is executed. Since the normal procedure for the hardware upon detecting such errors is to produce some type of "machine check" exception, halting the processor on these errors is easily managed by the operating system.

One type of processor failure that the fail-stop model does not represent well is a failure in the software executing on the machine. An error in the software that causes the service provided by the processor to diverge from its service specification could be caused by many problems, such as invalid user input or a latent error in the running program (a "bug"). In these cases, a normal processing unit will typically not halt immediately as a fail-stop processor would, but will simply continue running, producing incorrect results. Although the task of producing correct software is a difficult one, these types of errors can be handled by a careful programmer, and these questions are well covered by other areas of research such as integrated programming environments, automatic program testing and verification, and very-high-level programming languages. Hardware errors, however, are typically more unexpected than software errors, particularly those caused by external events such as power failures. Therefore, fault-tolerance mechanisms usually concentrate on handling hardware failures and assume that software errors either will not occur or will be handled reasonably by the same techniques that are applied to hardware errors.

3 Research Goals

The aim of this research is to develop a useful, low-overhead mechanism to achieve fault tolerance in a distributed system. This mechanism should allow processes in a distributed system to continue execution in spite of failures occurring in the system. Like most other fault-tolerance mechanisms, this mechanism will be designed to be able to handle only a single failure at a time, although this restriction may be relaxed in a later stage of the research. This section will discuss some of the important goals of this research.

An important goal for this new fault-tolerance mechanism is that it be *transparent* to both programs and programmers, allowing it to be immediately applicable to existing distributed applications without changes. One aspect of transparency that is sometimes neglected is that it should affect the performance of applications programs as little as possible. In a distributed system with many processors, one of the most useful measures of an application's performance is the elapsed running time required to solve a given problem with the program, and minimizing the impact of fault tolerance on this time will make the fault-tolerance mechanism more generally useful. Another important aspect of transparency is that fault tolerance should not restrict new processes and

processors from joining or leaving the system at any time. Since this is possible in most existing distributed systems, any fault-tolerance mechanism that restricts this flexibility would not be able to be used transparently in these systems.

Another goal of this research is that the resulting fault-tolerance mechanism should not rely on any specialized hardware that is not commonly found in existing distributed systems. Many current fault-tolerance mechanisms for distributed systems require special-purpose hardware, usually to assist with the additional communication between the processors required for the fault tolerance. Although these same mechanisms can usually be implemented without this special hardware by using appropriate protocols in software on top of standard hardware, the results would generally be slower than with the specialized hardware. If the special-purpose hardware can easily be added to the system, a more efficient fault-tolerance mechanism can be achieved. However, since most existing distributed systems do not have such hardware, a fault-tolerance mechanism that does not rely on it will be able to be used in a wider class of distributed systems.

4 Design Overview

The design of the sender-based message logging mechanism is based on a simple model distributed system. This model system is composed of a local network of fail-stop processors as described in Section 2.4. Any process in the system can send a message to any other process, regardless of which processor within the network they are running on, but these messages are assumed to be the only method by which processes can communicate. The processes in the system must be *deterministic* in the sense that, if two processes start in the same state, and both receive the identical sequence of inputs, then they will produce the identical sequence of outputs and will finish in the same state. Since messages are assumed to be the only form of interaction between processes, the state of a process is completely determined by its starting state and by the sequence of messages that it has received.

There are many ways of saving the state of a process so that its execution can be recovered following a failure. One of the simplest ways is to periodically write a *checkpoint* containing all of the process data and system data associated with the process to stable storage. This method can be very expensive, though, for processes with a large address space. Also, for systems containing many cooperating processes, this leaves open the possibility that the domino effect could occur, as described in Section 2.3. Since cooperating processes in a distributed system may be executing on several different processors separated by a local network, it is difficult and expensive to record a consistent global state for the system, making the prevention of the domino effect in a distributed system harder than in a centralized system.

The sender-based message logging mechanism uses a technique called *message logging* to independently track the state of individual processes between checkpoints. This technique records all messages exchanged between processes in a message log. The log provides enough information to reconstruct a consistent state after a failure, thus avoiding the the possibility of the domino effect. Although other fault-tolerance mechanisms using message logging have been designed and implemented [Borg83, Powell83b, Birman85], most of these systems either require special-purpose network hardware for the message logging or suffer from poor performance. Sender-based message logging uses a new low-overhead method to accomplish the message logging where all messages are logged in the address space of the sending process, as shown in Figure 2. This logging method results in a simple, highly-efficient fault-tolerance mechanism that works naturally in a distributed system.

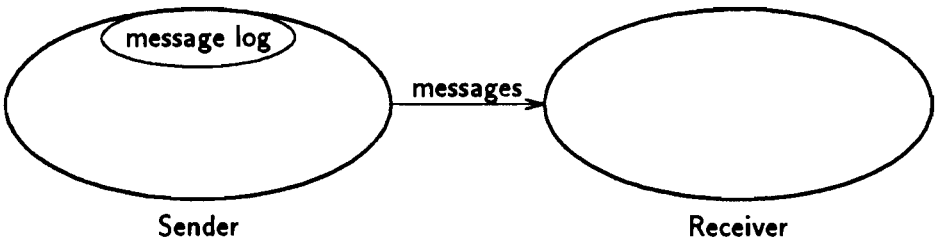

FIGURE 2: Process and message log configuration

In sender-based message logging, each process in the system is periodically checkpointed, although there is no coordination between when different processes are checkpointed, and the frequency of the checkpointing can be tuned to balance its expense against the time needed for recovery. Additionally, as discussed above, every message sent between processes is logged. Then, if a process fails, it is first restarted from its most recent checkpoint. To complete the recovery of the process, though, its state must be updated to account for any messages that it received since this checkpoint was originally taken. To accomplish this, the messages that were logged since that checkpoint are replayed to the process. While re-executing based on these logged messages from the checkpointed state, any messages that the process sends to other processes will be duplicates of those sent before the failure, and will be discarded. After completing this re-execution, the state of the process will again be consistent with that of the rest of the system, and recovery will be complete.

Most processes are both the sender of some messages and the receiver of other messages. Since the message log records all messages that have been *received* by a process, it is the receiver role of the process that is important for this recovery. If a process that has sent some messages fails, its state can be recovered from the log of messages that it has *received* in the same way as a process that had not sent any messages. The purpose of the message log is to recover the receiver of a message; the sender of the message is recovered based only on what messages it has received itself.

An informal proof of the overall correctness of the design follows easily from the definition of deterministic processes given above, since all processes in the system are assumed to be deterministic. A process that is recovering from a failure will begin execution in some state that the original process had before the failure. In particular, this will be the state saved in the most recent checkpoint of the original process taken before the failure. The recovering process will then receive, from the message log, the same input messages in the same order that were received by the original process before the failure after this checkpoint was taken. Since all processes are deterministic, the recovering process will then produce the same output messages (which will be discarded as duplicates) that the original process produced after the checkpoint and will complete recovery in the same state as the original process was in right before the failure. The recovered process will then be an exact reconstruction of the original process and can continue to communicate with other processes exactly as the original would have done if it had not failed.

5 The Basic Protocols

The mechanism of sender-based message logging is actually a combination of several protocols operating together to achieve fault tolerance. Protocols are necessary to accomplish the message

logging and the periodic checkpointing, as well as for fault detection and recovery. This section presents a preliminary design for these protocols.

5.1 Message Logging

5.1.1 Introduction

The method to be used for logging the messages exchanged between processes in the sender-based message logging mechanism is derived from an analysis of the minimum-cost method of doing the required logging. When one process sends a message to another, both the sender process and the receiver process naturally get (or already have) a copy of the message. In order to avoid sending the message elsewhere to log it as is done in existing message logging systems, it would be convenient if either the sender or the receiver process could serve as the log by saving a copy of the message in its own local volatile memory. Unfortunately, the receiver can not log the message in this way since the purpose of the logging is to be able to recover the receiver if it fails. The sender, though, could save a copy of each message in its volatile memory and provide it again when needed for recovery after a failure.

For recovery from this message logging to be successful, though, not only the data of each message is needed, but also the order in which each message was received relative to any other messages sent to the same recipient by other processes. Since each sender process only knows about its own messages, this ordering information is not normally available to be recorded in the log by the sender. To correct this problem, the receiver returns this ordering information to the sender for inclusion in the log.

5.1.2 The Receive Sequence Number

Each process maintains a sequence number of messages that it has received, called the *receive sequence number* or *RSN*. Each time a process receives a message, it increments the current value of its RSN and then returns this value to the sending process of that message. The RSN must be returned to the sender before the receiving process is allowed to look at the data of the message it has received. When the sending process gets this sequence number from the receiver process, it should enter it in the log in its local memory along with the message data already recorded. The sender can consider the return of the RSN as an acknowledgment that its message was received, and the return network packet for the RSN can be merged with or take the place of any existing acknowledgment message currently required by the system. There is no requirement that the sender must wait for the return of the RSN before continuing execution following the send, but it must be prepared to enter the RSN in the message log when it does arrive.

To illustrate this message logging method, the situation that would occur after a single sender process has sent three messages to the same receiver is shown in Figure 3. Here, the receiver initially had a receive sequence number value of 3. Then, the sending process sent three messages one at a time to the receiver, and got the sequence numbers 4, 5, and 6 back. For each message, the sender entered the message data and then the RSN value in the message log in its local address space. After these three messages, the current value of the receive sequence number for this receiver will be 6, and the next message to be received by this process will be assigned an RSN of 7.

The situation is slightly more complicated, though, when more than one sender process is communicating with the same receiver. In this case, the log of messages that have been received by this process since its last checkpoint will be distributed across the set of processes that sent these

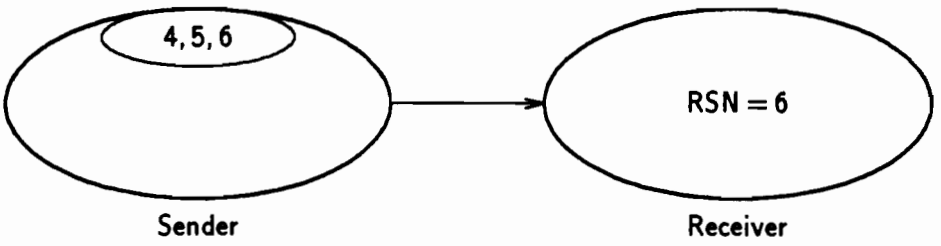

FIGURE 3: Example message log with one sender

messages. When multiple senders are involved, each sender will have only the messages that it sent recorded in its local message log for that receiver process. The message log for this receiver must be constructed when needed from these partial logs in the order of each message's recorded receive sequence number.

An example message log with more than one sender process is shown in Figure 4. Here, there are two sender processes that have communicated with the same receiver process since its last checkpoint was taken. The receiver initially had a receive sequence number value of 6. Two messages were then sent to this receiver by the first sender process, followed by two from the second sender process, and then one more from the first sender. For each message, the receiver returned the new value of its RSN to the correct sender, and these RSN values were entered in the sender's log along with the message data. After receiving these five messages, the value of the RSN in the receiver process will be 11.

5.1.3 Partially Logged Messages

The act of logging a message under sender-based message logging is not atomic since the message data and the receipt-ordering information can not be recorded in the log at the same time. It is therefore possible for the receiver to fail while some messages are only partially logged at the sender. For purposes of recovery, though, a message is not considered to be logged until both the

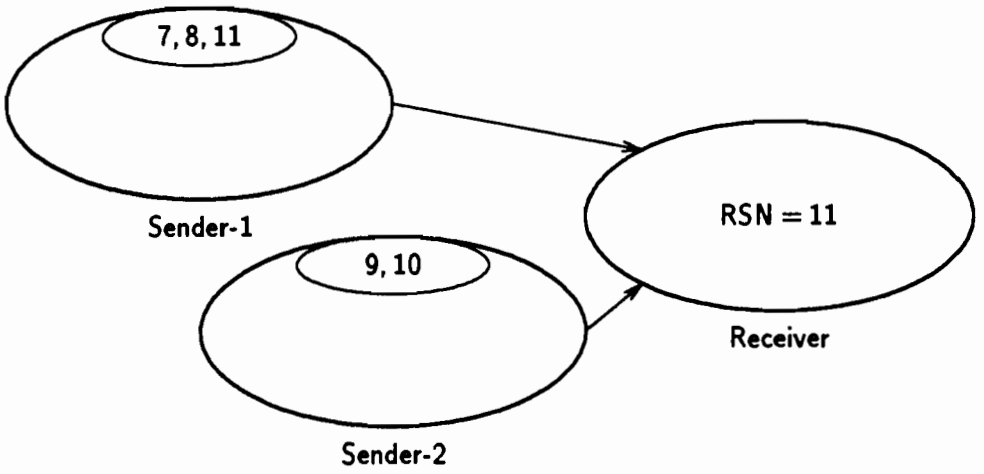

FIGURE 4: Example message log with multiple senders

message data and the RSN are both entered in the log. Since the message data is entered into the log before the message is sent, and since the RSN can only be entered after the message has been received by the target process, the only possible type of partially logged message is one with the RSN not yet entered. Any partially logged messages will not be replayed from the message log to the receiving process during its recovery, but these messages must still be sent to the process when its recovery from the message log is complete. The sender-based message logging protocols are designed so that these partially logged messages can be sent to the process in any order following recovery from the log.

There are three possible causes for a message to be only partially logged, with the RSN not yet entered. One possibility is that the receiver has not sent the RSN back to the sender process, either because it has not received the message or because it has received it and not had a chance to send the RSN; this includes the case when the message has not even been sent yet but has been copied into the message log by the sender. If the message has been received, it can not have affected the state of the receiver process yet, since the RSN must be returned before the receiver is allowed to examine the message data.[3] Thus, as long as the RSN has not been sent by the receiver, it is unimportant whether the message was actually received yet. The message can not have affected the state of the receiver process, and it therefore does not matter what order the message is ultimately received in. Any such message can simply be resent to the receiver process at any time after recovery from the logged messages is complete.

The second possible reason that a message might be only partially logged is that the packet containing the RSN is still in transit on the network from the receiver machine to the sender machine. With any reasonable network hardware and network interfaces, though, there is a limit on how long it can take a packet to arrive at its destination after being sent. Since any protocol that can detect the failure of a processor must rely on the arrival (or lack of arrival) of network packets, it must take longer to decide that the receiver has failed than it can take for the packet containing the RSN to arrive if was sent. That is, if the RSN packet was sent before the receiver failed, it must arrive at the sender before the sender can discover the failure. Thus, there should be no partially logged messages caused by the RSN still being in transit when the failure is detected. The RSN will either arrive normally, or the packet containing it will be lost on the network, leading to the final cause for a partially logged message.

The remaining possible cause for a message being only partially logged is that the network packet containing the RSN being returned was lost in transit. Although the error rate on typical local networks is very low, there is a real possibility that a packet containing an RSN may occasionally be lost on the network, either due to a transmission error or such problems as a buffer overrun in the network interface. The changes to the message logging protocol that are needed to handle this possibility of a packet containing an RSN being lost are taken up in the next section.

5.1.4 Lost RSN Packets

If the network connecting the processors of a distributed system utilizing sender-based message logging was perfectly reliable, the message logging protocol as described so far would be all that was needed to correctly implement this mechanism. Unfortunately, as mentioned previously, typical local networks provide only best-effort delivery of individual packets, so the chances of loosing an

[3]The only state that may be affected by the receipt of the message is confined to the implementation of the message logging protocol on the receiving machine. Since this state will be lost if the receiver fails, any changes to it from this message are harmless. None of these state changes can propagate off the receiving machine, and all of this state will be reconstructed along with the recovering process after the failure.

RSN packet are real. The state of a process is *unrecoverable* until all messages it has received are fully logged at their senders by the successful return of the RSN packet. If a process fails in an unrecoverable state, its state is *lost*. When one process sends a message to another, the state of the receiver *depends on* the state of the sender at the time the message was sent since any part of its state may have been included in the message. The state of a process becomes an *orphan state* and the process becomes an *orphan process* if it depends on a state that is lost.

To illustrate how such an orphan state can occur, Figure 5 shows an example situation consisting of three processes, P_1, P_2, and P_3. Here, process P_1 has sent a message to P_2. This message was assigned an RSN value of 20 by P_2, but the network packet containing the RSN being returned by P_2 was lost and never delivered to P_1. Then, P_2 sent a new message to P_3, which was assigned an RSN of 101 by P_3, and this RSN was successfully returned and logged by P_2. The message sent to P_3 by P_2 could have contained any information in P_2's state, and thus the state of P_3 then *depends on* the state of P_2 at the time the message was sent. Since the RSN returned by P_2 was lost and not logged by P_1, this state of P_2 is unrecoverable at this time because P_1 does not know the correct order of receipt for the message that it sent relative to any other messages that P_2 may have received. Thus, if P_2 fails after sending the message to P_3, the state of P_3 becomes an orphan state since it depends on this unrecoverable state.

One way to solve the problem of lost RSN packets and to avoid entering orphan states involves a simple modification to the message logging protocol to guarantee the delivery of all RSN packets. In this solution, the sender of a message must acknowledge back to the receiver process when it gets the RSN, and the receiver process must continue to retransmit with some timeout the RSN packet until it gets this acknowledgement. This extra acknowledgement can proceed asynchronously with the regular execution of both the sender and the receiver processes. However, any process that attempts to send a new message and has unacknowledged RSN packets from previous messages that it received must be delayed until all of these acknowledgements arrive. The operation of this modification of the message logging protocol is illustrated in Figure 6. It avoids the orphan state problem since it does not allow any process to create a dependency in another process when it is unsure that its own state is recoverable. By delaying new sends until all RSN packets have been acknowledged, no state changes that resulted from the receipt of messages can be propagated to other processes until those received messages are completely logged in their senders. Since this extra

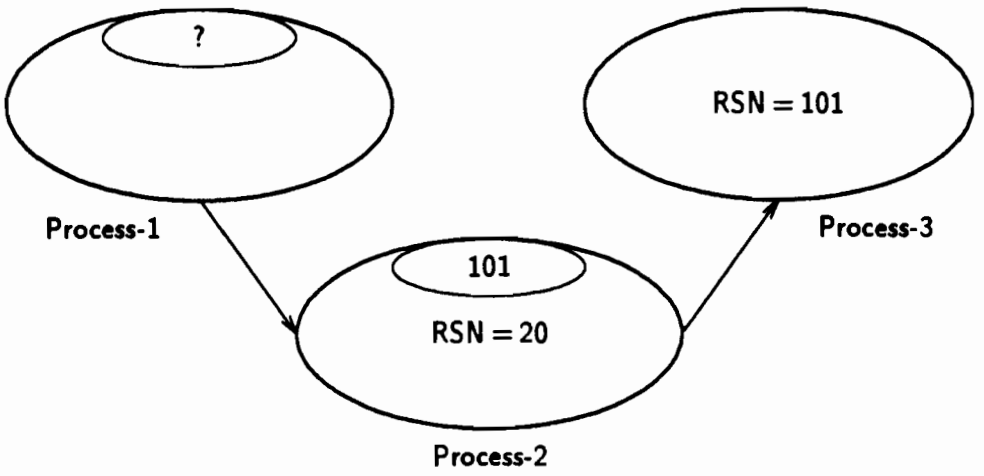

FIGURE 5: Example orphan state caused by a lost RSN packet

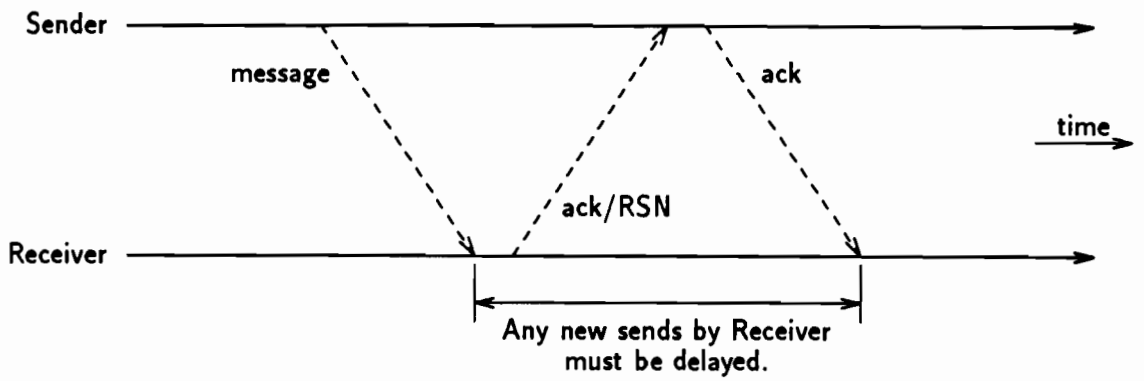

FIGURE 6: Operation of the message logging protocol to avoid orphan states

acknowledgement can proceed completely asynchronously, there should be little delay in normal processing, except when the receiver process attempts to send new messages shortly after receiving a message.

An alternative way to solve the problem of lost RSN packets is to allow the possible orphan state to be entered, but to maintain enough information within the system to detect when the orphan state does occur and to be able to recover from it. That is, this solution will allow the receiver process to send new messages without guaranteeing that previous messages that has it received have been fully logged. Any processes that have received messages from this process since it received the first message that is only partially logged will be orphaned if this process fails, since the state of a process is only recoverable once all of the messages that it has received have been fully logged. There are thus two requirements to the integration of this solution into the message logging protocol as already described. First, a method is needed to detect that a process is in an orphan state or a possible orphan state, and second, enough information needs to be maintained to be able to recover a process when it is discovered to be in an orphan state following the failure of another process.

If a process sends a message when there are partially logged messages for it at some other process, the process that receives this message may enter an orphan state. When it receives this message, its state then depends on the state of the sender process, but since the sender has some partially logged messages, the sender's state is unrecoverable at that time. If the logging for all of the sender's partially logged messages eventually is completed, no harm will come from this situation, and the state of the receiver will not actually become an orphan state. However, if the sender fails while the receiver is in this potential orphan state, with its state depending on the unrecoverable state of the sender, the receiver's state will become an orphan. Therefore, what is needed to detect the possibility or existence of an orphan state is to determine if a process has received any messages that were sent after the sender received a message that is only partially logged.

In order to implement this orphan detection algorithm, it must be possible to order the messages that a process has sent. Every process must, therefore, maintain a *send sequence number*, or *SSN*, which is a sequence number of messages sent by that process. If the current value of the SSN of the receiver is returned along with its RSN to the sender every time a process receives a message, the sender will know the sequence number of the most recent message sent by the receiver before the message was received. The SSN should easily be able to be included in the same network packet that is carrying the RSN from the receiver to the sender. When the sender gets the SSN back from

the receiver, it should enter the SSN in the message log along with the message data that is already there and the RSN that was returned with this SSN.

With this send sequence number information in the message log, the decision as to whether the failure of a given process can cause the state of some other process to become an orphan is straightforward. If there are no partially logged messages among those logged for the given process, its state is completely recoverable, and thus, its failure can not orphan the state of any other process. If, however, there are partially logged messages, any process that has received a message with a higher SSN than the one logged with the last *fully* logged message before any partially logged messages may become orphaned if this process fails. It can not be known exactly which messages were sent by this process after it received the first partially logged message, since the RSN and SSN for it have not been logged. Thus, this algorithm must assume that any message sent after the last fully logged message was received (those with a higher SSN) could contain information received in this partially logged message.

There are two times at which this orphan-detection algorithm must be used. First, when a process fails, each other process must decide if its state has been orphaned by that failure. This can easily be accomplished by a broadcasting protocol that enquires about the RSNs and SSNs logged at each process for the failed process. If a partially logged message is discovered (with a missing RSN), and if the process doing this check has received a message from the failed process with a higher SSN than the one on the last fully logged message, then the state of this process is now an orphan state.

The second time at which this orphan-detection algorithm must be used is before a process is checkpointed, since if the process does become an orphan, a checkpoint from before the state was orphaned will be needed for recovery. After recovering the state of a process following a failure, the states of any orphaned processes can be recovered by forcing them to fail one at a time and recovering them from their checkpoints and message logs in the same way as is used for normal process failures. Some of the messages logged for an orphaned process may be recorded in the volatile memory of the failed process, but this log information will be reconstructed during the recovery of that process. After the failed process and all orphans are recovered, their states will be consistent as of the time that the last fully logged message was received before the failure.

Either of these alternative methods for solving the problem of lost RSN packets could be used in the sender-based message logging protocols, and each has certain advantages and disadvantages. The first method, which uses acknowledgements of the RSN packets to guarantee that no process can ever enter an orphan state as a result of a lost RSN packet, does not require any extra time on recovery after a failure to also recover the states of any orphaned processes, but does require the small overhead of one extra asynchronous packet for each message received. Also, this method will require new sends by a process to be delayed until all previous RSN packets from this process have been acknowledged. The second method for solving the lost RSN packet problem, which relies on recovering processes whose states become orphaned following a failure, requires a slightly more complex protocol for recovery and checkpointing and can take longer to recover the system as a whole if there are orphaned processes, but requires essentially no extra overhead during normal operation of the system. The choice of which of these methods should be incorporated into the sender-based message logging protocols is not clear at this time and should be considered carefully.

5.2 Checkpointing

Checkpointing in sender-based message logging servers two purposes. First, it limits the amount of recomputation that must be done by a process when it recovers after a failure. To recover a process,

it is restarted from its most recent checkpoint, and only messages that it originally received after this checkpoint need to be resent to it during recovery. Also, since only those messages originally received since the most recent checkpoint are needed for recovery, earlier messages can be discarded each time a checkpoint it taken. Thus, checkpointing also servers to limit the size of the message logs that must be stored.

As discussed in the overview of the sender-based message logging design in Section 4, the frequency with which the checkpointing is done can be tuned to balance its expense against the time needed for recovery. Although a process can be checkpointed at almost any time,[4] there are two parameters, which correspond to the two purposes of the checkpointing, that could be evaluated in determining when a checkpoint "optimally" should be taken for process [Borg83]. First, a checkpoint should be taken when ever the process has executed for some specified amount of CPU time since its last checkpoint, thus placing an upper bound on the amount of work that must be redone in case of a failure. Second, when the size of the message log for the process reaches some specified size, the process should be checkpointed so as to limit the total space needed for the log. The limits used for these two parameters will depend on the amount of available spare CPU time, the amount of free memory to store the message log, and the cost of taking a checkpoint.

The checkpoint for each process can be stored either on disk or in the volatile memory of some other processor in the system. Rather than writing the entire address space of a process to the checkpoint each time one is taken, it is more efficient if only those portions of the address space that have been modified since the last checkpoint are written out. If a hardware memory management unit or dynamic address translation hardware is available (such as is required to support virtual memory), the "modified" bit on the page table entry for each page of the address space can be used to identify the changed portion of the address space. The use of this technique is not required, though, but can be used to make the checkpointing more efficient.

5.3 Fault Detection

When a process fails, this fault in the system must be detected before recovery of the process can be initiated. Since the process itself has failed, this fault detection must be accomplished by the remaining processes in the system. When a process failure is detected, it is important that the failed process is only restarted once, so there must be cooperation between the other processes in the system in this failure detection. Only one process should decide to restart the failed process, and this process should control and coordinate the recovery including the replay of messages from the log.

One method for detecting failures would be to use some periodic polling protocol to check that each process that a process knows about or is communicating with is still alive. With the sender-based message logging protocol described in Section 5.1, the set of processes that must be polled in this way is the union of the set of those processes that have messages logged for the polling process and the set of processes that this process has messages logged for. This first set of the union is simply the set of processes that this process has received messages from since its own last checkpoint, and the second set of the union can easily be determined by looking in the local message log to see which processes do indeed have messages logged there for them. Thus, the set of processes that it is important to poll for fault detection is known always to each process. If some

[4] When using the second method discussed in Section 5.1.4 for solving the problem of lost RSN packets, which relies on being able to recover processes whose states are orphaned by a failure, a process should not be checkpointed when its state could become an orphan.

number of these periodic polls to a given process are not answered, the process can be assumed to have failed. Any normal message traffic to or from each process in this polling set can serve the place of a scheduled periodic poll, thus saving overall message traffic.

Another method for fault detection would be to detect faults only when a process attempts to send a message to some failed process. If the intended recipient of a message does not respond with the receive sequence number within some timeout, it might be reasonable to assume that the process has failed. However, this could also occur if the target process never existed, so it is important to be careful to determine which is the true cause of this unresponsiveness. Since a common method used in distributed systems for determining when the user is attempting to send a message to a nonexistent process is to simply send the message and detect the fact that no process replies, some method must be designed to enable the sender to tell this situation from one where the recipient has only failed. Perhaps this method of fault detection could be used together with the first method suggested above to allow for faster detection of faults before the polling period has expired.

5.4 Fault Recovery

To recover a failed process, it should first be restarted from its most recent checkpoint. Then, to update its state to the point at which the failure occurred, any messages that have been completely logged in the message log should be resent to it in order of the receive sequence numbers. Only messages for which both the message data and the RSN were recorded in the log should be resent at this time; messages for which the RSN was never logged should be ignored at this time. There will be a separate message log stored at each process that sent messages to the failed process after its last checkpoint, and these processes must cooperate to resend these logged messages to the recovering process in sequence-number order. It should be possible to easily locate the logged messages in sequence number order by using a broadcasting protocol. For example, the recovering process could send a broadcast message asking for the next logged message to be sent to it until all logged messages have been received.

As the recovering process executes from its checkpointed state to the state it had at the time of the failure, it will attempt to resend any messages that it sent during this period before the failure. Since these messages will be duplicates of the ones that were sent before the failure, they have already been received and acted on by the processes that they were sent to. Therefore, it is important to make sure that these duplicate messages are detected and not acted on again by their recipients. A simple method for accomplishing this is to add a send sequence number to each message.[5] If the next sequence number to use when sending a message is part of the checkpoint, the sequence numbers used by the process during recovery will be the same as those used before the failure. Duplicate messages can then be detected on receipt by checking if the message's sequence number is less than the next sequence number expected by the recipient. Since most distributed systems already require such sequence numbers for duplicate detection due to retransmissions, handling these additional duplicate messages should require little or no modification of existing systems.

When a process fails, it may be restarted on some processor other than the one it was running on when it failed. When other processes that were communicating with this process before the failure attempt to send new messages to it across the network, they need to determine the network

[5]This is the same as the send sequence number suggested in the alternative solution to the problem of lost RSN packets presented in Section 5.1.4.

address of the new processor on which it is executing now. Since the recovery operation will need to broadcast messages across the network in order to find the different machines at which the message log is distributed, these broadcasts should be able to serve as notification of the new network address for the process. Recent work on process migration under the V-System [Theimer85] and under DEMOS/MP [Powell83a] should also be useful in refining the details of how this process location should be done, since this problem is basic to any process migration mechanism.

6 Prototype Implementation

The V-System developed at Stanford University [Cheriton83, Cheriton84] appears to be a good base on which to build a prototype implementation of the sender-based message logging mechanism. The V-System is a fairly complete distributed operating system designed to support efficient interprocess communication. It runs on a collection of Sun Workstations[6] connected by a 10-megabit per second Ethernet local network [Metcalfe76, Shoch80]. Each machine runs an independent copy of the V-System kernel, and these kernels cooperate to provide transparent, location-independent message-passing communication between processes. A final point that makes the V-System a good choice for this prototype implementation is that a reasonably-sized network of machines running it will be readily available during the period that this research is being conducted.

In order to produce a successful prototype implementation of the sender-based message logging mechanism, it will be necessary to address a number of system-specific issues. Some of these issues would be equally important with any choice of implementation system, but some of them are important only because the V-System differs slightly from the model system assumed in the design of the sender-based message logging protocols (Section 4). Although these issues are specific to the choice of the V-System as the system in which to implement this prototype, the decisions made for them should be applicable over a wide class of similar target systems. This section will discuss some of these issues that will be involved in such an implementation.

Although the V-System supports the type of transparent message-passing communication between processes that was assumed in the protocol design, it also supports some additional communications mechanisms. Perhaps the most important of these other mechanisms is the provision of a limited form of shared memory between cooperating processes on the same machine. Under the V-System, the notions of *process* and *address space* are separated, allowing a *team* of processes to inexpensively share data within a single *team space*. This will force the implementation of sender-based message logging to consider the team in many places in the protocols where the process would normally be considered. For example, the protocol to deal with the problem of a lost RSN packet suggested in Section 5.1.4 that forces a process to delay sending new messages until all previous RSN packets have been acknowledged will need to be modified to consider the entire team of processes. If any of the processes on the team of the sending process has any unacknowledged RSN packets, the send must be delayed. Also, since checkpoints record the state of the address space of a process, the implementation of checkpointing for the V-System will need to record the state of the entire team rather than taking an individual checkpoint for each process on the team.

Another difference between the communications primitives provided by the V-System and those assumed in the design of the sender-based message logging protocols is that the V-System primitives are at a slightly higher level of abstraction. Rather than providing only simple message-passing primitives, the V-System provides *Send*, *Receive*, and *Reply* operations for sending a message to

[6]Sun Workstation is a registered trademark of Sun Microsystems, Inc.

another process and getting a reply back, and *MoveFrom* and *MoveTo* operations for copying sections of memory between processes. Although these primitives are actually composed of simple combinations of the more basic message-passing operations, it may be possible to exploit the relative frequency of the message-passing patterns used by them to more efficiently support the sender-based message logging protocols. For example, since request messages usually have a reply message shortly following them, it may be possible to combine the functions of these two messages with some of the packets required for the message logging protocol, such as returning the RSN for the original message in the packet containing the reply message. Unfortunately, this reply message is not returned until after the receiving process has been allowed to look at the message data, so some refinement of the logging protocol may be needed to support this in the most efficient way. Also, it is possible in the V-System to designate certain messages, or rather the operations invoked by those messages, as *idempotent*, which means that the behavior of the operation, including its return message and side-effects, will be the same regardless of the number of times that the operation might be invoked, as long as the initial state is the same each time. It might be possible to treat such idempotent messages specially in the message logging protocol to take advantage of this property.

The V-System kernel provides only a somewhat minimal set of primitives required for operation, with all other functions built on top of this with server processes, much in the same way that a computer backplane provides only what is necessary to plug in individual boards to provide the remaining functionality required. Although this "software chassis" [Cheriton84] structure is largely a philosophical decision about how such systems "should" be structured, it seems reasonable to follow this model as much as possible when adding fault tolerance to the system. Therefore, it must be decided how much of the mechanism should be implemented inside the kernel and how much should be done in user or server processes.

Another question that must be decided is exactly what information should be included in each checkpoint. Since address spaces in the V-System belong to teams of cooperating processes rather than to single processes, each checkpoint will have to cover an entire team. Associated with each team are certain system state variables inside the kernel, such as a data structure describing the team itself and one for each process belonging to the team. It will be necessary to identify exactly portions of these and other system state variables should also be included in each checkpoint. The recent research by Theimer, Lantz, and Cheriton on process migration in the V-System [Theimer85] will be helpful in this regard, since this same question of identifying the exact components that make up the state of a process in the V-System had to be answered for their work.

7 Measurement, Evaluation, and Testing

In order to achieve a level of confidence in the design of sender-based message logging and in its prototype implementation, it will be necessary to examine it in operation and to measure and evaluate its performance. This measurement, evaluation, and testing should cover both the correctness of the mechanism and the implementation (whether it can indeed survive faults in the system) and the performance of the resulting system (whether the fault tolerance negatively affects the behavior of the system, such as the elapsed execution time of distributed applications). Other factors such as the time required for recovery following a failure and the amount of communication and storage required by the fault-tolerance mechanism will also be measured.

There are, unfortunately, few standard tools available for performance monitoring and measurement of distributed systems. It will thus be necessary to design and develop many of the tools and procedures necessary to measure this prototype implementation. One of the phases of the research

currently being conducted locally by the Department of Computer Science at Rice University is to design and develop such monitoring tools [Almes85b]. It is expected that the development of sender-based message logging will benefit from these tools and will further help to develop them.

Testing the correctness of the sender-based message logging protocols and the prototype implementation will be difficult since the inherent concurrency in distributed systems and distributed applications makes possible such a large number of different execution orderings within the system. Simply running a number of applications on the system and forcing various components to fail during these executions will provide some solid evidence of correctness, but does not actually provide proof. Due to the large number of possible execution orderings, there will be some possibility that a situation that exposes some problem with the system will not be encountered by such testing. A method needs to be developed to assist with this testing process that will allow more precise control over the configuration of the system and the component being failed.

One of the more important measurements to make for a performance evaluation of this mechanism will be to determine its overhead on distributed applications, both when failures occur in the system and in the absence of failures. One of the most useful measures of the overhead in such a system is the difference that adding the fault tolerance makes on the required elapsed running time of typical distributed applications. Also important will be a measure of the average and maximum amounts of storage space required to hold the fault-tolerance information such as the message log, and the amount of extra communications overhead placed on the network connecting the distributed system.

8 Other Issues

There are a number of other issues related to the design of sender-based message logging that could be explored within this research. This section discusses some of these issues and how they relate to the design of these protocols.

The design of sender-based message logging presented in this paper concentrates on the communication between deterministic processes in a distributed system. However, real programs at times communicate with the "outside world", for example, to take input from the user, to report results to the user, or to read and write permanent files on disk. These are all instances of the communication of a process with some physical device, rather than with another process. To handle this type of communication in sender-based message logging, the protocols must guarantee that any interactions with these devices in the outside world that are re-executed during recovery are identical to what they were originally before the failure. If device communication is performed as a message between a process and the device just as messages are sent between regular processes, the requirement for correct communication with the outside world is simply that no device in the outside world can be allowed to enter an unrecoverable state as describeed in Section 5.1.4. This is because, if this unrecoverable state should become an orphan state with the failure of the process doing the communication, the state of the device in the outside world can not be rolled back and recovered as the state of a process can.

The idea of process migration has received considerable attention from researchers in distributed systems, and a number of successful implementations have been created [Powell83a, Theimer85]. With process migration, individual processes in a distributed system can be moved from one processor to another over the network without affecting their execution or the execution of cooperating processes. This can be used to more evenly balance the load on the processors in the system and to remove "guest" processes from a personal workstation that had been running there while the

workstation's owner did not need his machine. With a fault-tolerance mechanism, when a failed process is recovered, it may be restarted on a processor other than the one on which it originally ran, particularly when a long-term hardware failure on that processor was the cause of the process failure. With such an ability already needed for fault tolerance, process migration could easily be accomplished on demand by forcing the target process to fail and restarting it on the processor to which it should be migrated. Consideration of this useful extra ability of fault-tolerance mechanisms will be given in the design and implementation of sender-based message logging.

Different message logging mechanisms have been used successfully by others as the basis for debugging systems for distributed programs [LeBlanc85a, Curtis82]. In such systems, the message log allows individual processes to be re-executed under control of a conventional single-process debugger whenever the behavior of that process is questioned. Replaying the messages from the log guarantees that its behavior while being examined under the debugger will be the same as it was when it first executed and the messages were logged. Individual processes can thus be examined and debugged without disturbing their interactions with other processes. It seems reasonable that sender-based message logging could provide this type of debugging support with little extra effort over that needed for its basic function in fault tolerance.

9 Related Work

Many different mechanisms have been used to achieve fault tolerance in both commercial and research systems. This section presents a survey of the different fault-tolerance mechanisms that have been implemented in software and describes representative systems for each mechanism. The list of systems presented here is not intended to be an exhaustive survey of existing fault-tolerance implementations but merely to give examples of the different fault-tolerance mechanisms discussed. Also, since there is some overlap in between these mechanisms, and because some of the example systems presented here use more than one mechanism, each example described below has been placed under only the presentation of the single fault-tolerance mechanism that it primarily exemplifies. The following fault-tolerance mechanisms and representative systems will be discussed:

- Application-specific mechanisms: recovery blocks
- Functional programming: the multi-satellite star
- Atomic actions: ARGUS
- N-modular redundancy: SIFT
- Active backup processes: Circus
- Checkpointing: Tandem NonStop
- Message logging: Auragen's Auros
- Optimistic recovery: Strom and Yemini's work

9.1 Application-Specific Mechanisms

The earliest attempts to build fault-tolerant systems did not provide general-purpose mechanisms that could be transparently used by any programs in the system. Rather, they relied on *application-specific mechanisms*, requiring the programmer to explicitly design the fault tolerance into each program. For example, the early Xerox work with the "worm" programs [Shoch82] relied on the programmer to embed the appropriate application-specific mechanisms into each worm so that it could survive processor failures. Surveys of several other such methods have been written, including ones by Denning [Denning76] and Anderson [Anderson81]. The example application-specific mechanism that will be discussed in this section is a mechanism known as *recovery blocks* [Randell75, Russell80].

A recovery block consists of a sequence of different implementations of some task, where the first implementation is called the *primary module*, and the others are referred to as *alternate modules*. Each module within the block has an acceptance test associated with it that decides if the results of the execution of that module are to be accepted. To execute the recovery block, the primary module is first executed and its acceptance test is then evaluated. If this test succeeds, the execution of the block is complete, and control is transferred to the first instruction outside the entire block. However, if the acceptance test is not passed or if a failure occurred during the execution of the primary module, the state of the system is restored to its initial value on entry to the block, and the first alternate module is executed. This continues if needed with each alternate module in turn until one of the acceptance tests succeeds or until there are no more modules in the block to try.

The primary disadvantage of using these application-specific mechanisms is that a great deal of knowledge about the system and the types of errors expected must be programmed into each application. For example, with recovery blocks, the programmer is responsible for providing enough modules within each block to handle the number of failures that could occur while the block is executing. A system using these techniques would require existing distributed programs to be carefully modified or even rewritten in order to run with fault tolerance.

9.2 Functional Programming

Functional programming is a style of programming where all computation is performed by evaluating *functions* that produce no side effects other than their return values. Probably the most popular example of a functional programming language is LISP. In a distributed computation, if the individual processes follow this same style in their communications with each other (whether they communicate with remote procedure calls or through messages), they are *functional* in nature. Specifically, such processes must not retain any state information between invocations and must not operate on any data other than what is passed to them when they are invoked or what is compiled into each program.[7] Because of these restrictions on the behavior of a functional process, recovering from a failure in one is simple and straightforward since the process can simply be reinvoked with the same parameters, after perhaps reloading it on a different processor [Grit84].

Several systems have been developed that take advantage of this easy fault-tolerance recovery allowed by functional programming. The DIB system [Finkel85], a package for distributed implementation of backtracking problems developed for the Crystal project at the University of

[7]This is a slightly stronger restriction than required for *idempotence*, which requires only that the results and the side effects be the same regardless of the number of times it is invoked. All functional programs are idempotent, but the reverse is not necessarily true.

Wisconsin, is designed so that the subproblems that are allocated to other processors are purely functional. David Hornig's STARDUST system at Carnegie-Mellon University [Hornig84] consists of a functional language and execution environment that make use of the properties of functional programming for fault tolerance. Keller and Lindstrom of the University of Utah have discussed the advantages of using functional programming concepts in distributed computing and have shown how to apply functional programming to the implementation of a distributed database system [Keller85]. Also at the University of Utah, Lin and Keller have developed a mechanism called functional checkpointing [Lin86] to support fault-tolerance recovery in a distributed application consisting of functional subtasks. The representative system that will be discussed in detail in this section is the *multi-satellite star* work done by Cheriton and Stumm at Stanford University [Cheriton86] under the V-System [Cheriton83, Cheriton84].

The Multi-Satellite Star is a master-slave model designed to use a network of personal workstations to run compute-intensive applications. In this model, a program is structured as a set of *subtasks* that can be executed on some number of logical processors called *satellite modules.* These subtasks are controlled by and assigned to specific satellite modules for execution by a master module called *star central.* Star central starts some of the subtasks executing on the satellite modules, and when one of these subtasks completes, the satellite on which it was running requests a new subtask assignment from star central. These subtasks are required to be *functional* processes.

Star central monitors the execution of the subtasks on the satellite modules and detects when one of the satellites has failed. Since the subtasks are required to be purely functional, each has no effect on the overall computation until it returns its results to star central. This fact allows star central to recover a failed subtask by simply restarting it on another processor when convenient. The failure is noted in a list of pending subtasks, and when a satellite module next requests work, the failed subtask is restarted from the beginning on that satellite. Normally, the time lost due to a failure is simply the amount of time that the subtask had been executing before the failure (since that work must now be redone) plus the time taken by star central to detect the failure. Since star central itself does not observe the necessary functional programming restrictions, it can not be covered by the simple fault tolerance of this model and must be handled by other mechanisms.[8]

In addition to the simple nature of fault-tolerance recovery for functional programs, there are several other advantages to this style of programming. With most fault-tolerance mechanisms, it is necessary to be certain that the original process has actually failed and is no longer executing before restarting it to recover from a failure. With purely functional processes, however, since there are no side effects to the execution, the results will be the same (except for efficiency) if the original has actually not failed. Also, ordinarily sequential invocations of functional processes may be run in parallel with the same results, yielding greater efficiency if sufficient extra processors are available [Friedman78]. However, since a process must be entirely restarted if it fails, a large amount of work may need to be redone; a fault-tolerance mechanism that allows for recovery points *within* a computation could avoid this possibly excessive recomputation. Also, some computations may not be easily expressible functionally, and in particular, not all existing distributed applications obey the necessary functional restrictions. Therefore, despite the elegance of this mechanism when it can be applied, functional programming can not form the basis of a general-purpose fault-tolerance mechanism that does not require existing programs to be rewritten.

[8] Fault tolerance for star central itself is achieved through a checkpointing mechanism similar to that described in Section 9.6. Since star central resides entirely on a single processor though, the actual use of checkpointing for this is simplified.

9.3 Atomic Actions

The mechanism of *atomic actions* is derived from a concurrency control mechanism commonly used in database systems, called *atomic transaction* [Gray79, Lampson81, Bernstein81, Haerder83]. An atomic transaction is a set of processing steps that transforms the database (or other data) from one consistent state to another and that must execute without interfering with other concurrently executing atomic transactions. If the transaction completes normally (it is said to *commit*), exactly all of the changes to the data made by the transaction will be completed, and these changes will survive any future failures in the system. If the transaction does not complete normally (it is said to *abort*), none of the data changes made by the transaction will be retained, and the data will be restored to its initial state in which the transaction began. If a failure in the system occurs while the transaction is executing, it must be possible either to complete the transaction after recovery or to force it to abort, reversing any changes to the data that might have taken place. This idea of atomic transaction has been extended to form atomic actions by allowing arbitrary operations on user-defined data types, rather than only simple read and write operations on static data. This extended mechanism can be used in more general-purpose computations, but still retains all of the failure recovery properties of atomic transactions.

The mechanism of atomic actions has been used in the construction of several fault-tolerance systems. The Archons project at Carnegie-Mellon University [Jensen82, Schwarz84] defines the operations on instances of abstract data types (called "resources" in their model) to behave as atomic actions. Also at Carnegie-Mellon University, a distributed transaction system named TABS [Spector85b, Spector85a] has been constructed, which uses atomic actions for fault tolerance. The Clouds project at Georgia Institute of Technology [LeBlanc85b, Allchin83] uses a programming language named Aeolus to create atomic actions that execute on a local network of workstations. The Gutenberg system [Vinter86], a distributed operating system developed at the University of Massachusetts, provides Recoverable Communicating Actions, a form of atomic actions, for building fault-tolerant distributed applications. The system that will be used as an example for further discussion in the remainder of this section is the ARGUS system developed by Barbara Liskov and others at Massachusetts Institute of Technology [Liskov82, Liskov83, Weihl83, Liskov79].

In ARGUS, the atomic actions (referred to simply as *actions* in ARGUS) are executed by collections of processes called *guardians*, which manage and control access to the abstract data types of the system. Instances of abstract data types in ARGUS are called *objects*, and some of these may be declared to be *atomic objects*. Only the values of atomic objects are preserved by the system across failures. To guarantee that multiple concurrent actions within a guardian do not interfere with each other, ARGUS adopts the usual notion of read and write locks on atomic objects [Bernstein81]. When a write lock is granted on an object, a *version* of that object is created, and all changes to this object during the action operate on the version. If the action commits, the version replaces the original object, but if the action aborts, the version is destroyed.

So that any atomic objects modified by an action can survive failures, their values are saved to stable storage when the action commits. If a failure occurs, any actions currently executing in the failed guardian are forced to abort, and the guardian is restarted (on some other processor if necessary). When a guardian restarts in this way, only the state of its atomic objects can be restored from stable storage. The values of any other data that the guardian needs must be restored by a special action executed by the system at the time the guardian is restarted. Therefore, only redundant data that can be recreated after a failure (for example, an inverted index into a database) should be stored in non-atomic objects. Once the remainder of the guardian's data is restored in this way, new actions are allowed to begin execution as normal.

An advantage in using atomic actions to provide fault tolerance is that they are fairly well understood since they are a natural outgrowth of the atomic transactions used in database systems. However, the use of this mechanism would require most existing distributed applications to be rewritten to structure them as a set of (possibly nested) atomic actions on abstract data types. Also, it is left up to the programmer to decide which objects within the guardians need to be atomic in a particular application. Finally, such a mechanism may be slow, since there is a large amount of communication and synchronization overhead needed to support the atomic actions.

9.4 N-Modular Redundancy

A mechanism frequently used in hardware implementations of fault tolerance is *N-modular redundancy* (or *NMR*) [Carter85], where a given component is replicated some number of times, and a special voting circuit determines the correct answer based on the answer supplied by the majority of the replicated components. In order to detect errors, the replication factor N must be at least two, but it can in fact be much larger than this, limited primarily by the complexity of the voting circuit used. A special case of N-modular redundancy is *triple-modular redundancy* (or *TMR*) and is the most common replication factor used. Although these mechanisms were originally designed for hardware implementation, several systems have been built where the voting is done in software from software-generated answers. When each of these replicated software components runs a different implementation of the program, this technique is known as *N-version programming.*

A number of example systems exist that use some form of N-modular redundancy to achieve fault tolerance. The PRIME time-sharing system [Fabry73] designed at the University of California at Berkeley uses "dynamic verification" in software to check operating system decisions produced by replicated processors. The MOMENTUM system developed by the British firm Information Technology Limited [Smith85] uses a dual-system architecture where each "half-system" communicates with and checks the other through a special interprocessor bus. Xiao-Zong Yang at Harbin Institute of Technology and Gary York at Carnegie-Mellon University have experimented with a software implementation of triple modular redundancy on a system of Intel iAPX 432 processors [Yang85]. The remainder of this section will discuss the SIFT (Software Implemented Fault Tolerance) system developed at SRI International [Wensley78] as a representative system using an N-modular redundancy fault-tolerance mechanism.

The SIFT system was developed for NASA as part of a study in designing and implementing a reliable computer system for advanced aircraft control. The system consists of at least three processing modules connected to each other over replicated buses. Under SIFT, applications software is structured as a set of *tasks*, which are each composed of a sequence of *iterations*. Each task is run simultaneously on multiple processors, with each copy synchronizing with the others at each loop iteration. The parameters for each loop iteration are determined through software majority voting conducted by an *executive software* module on the processor on which the applications software task is executing. At each step, each task obtains the parameters for the next iteration from the executive software, executes the loop body, and returns the results to the executive. The executive waits for all of the replicated copies of the task to finish one iteration, decides on the correct results based on a majority vote of the outputs of all tasks, and finally starts the program on the next iteration.

One advantage of the N-modular redundancy mechanism is that it does not need to rely on the fail-stop model of processor failure. Since all results are voted on before acceptance, errors that cause unexpected results other than simply halting the processor can be detected and rejected. Also, if N-version programming is used, software errors in the application processes can be tolerated as

well. Since the replicated processes are all (at least very closely) up to date, the computation can continue almost uninterrupted after the failure of one process, making this mechanism very useful in real-time systems. One problem with this technique, though, is that it requires a large amount of extra computing power to support. Since the applications software must actively execute on each of the N processors (the level of replication), at least a factor of N extra computing power must be available.

9.5 Active Backup Processes

The *active backup processes* fault-tolerance mechanism executes multiple concurrent copies of each process much in the same way as is done with the N-modular redundancy mechanism described above. However, rather than using a majority voting algorithm from the results of the replicated processes, this mechanism simply chooses a single answer from one process to use as the "correct" answer. The particular answer chosen is usually simply the first answer returned, but some systems use a designated "primary" process for all answers until that process fails, at which time another processes is chosen as primary. The name of this mechanism derives from the fact that the computation is actively running on multiple processors, all of which serve as backups to the primary process in case the primary fails.

Several fault-tolerance systems have been developed that use active backup processes to implement fault tolerance. The CHORUS system at INRIA in France [Banino85] uses a "coupled actors" mechanism [Banino82] where each server actor (essentially a process) is paired with a backup actor, which performs the same computations as the master but is one processing step behind the master. MP [Gait85], a distributed process manager developed by Tektronix, runs an identical copy of each process on each workstation that is willing to accept the process, and is responsible for coordinating the inputs and outputs of this process set. The MARS system developed at the Technical University of Berlin [Kopetz85b] is a generalized architecture for real-time applications that masks component failures through replication at the subsystem level. The remainder of this section will discuss the Circus system developed by Eric Cooper at the University of California at Berkeley [Cooper85] as a representative system using the active backup processes fault-tolerance mechanism.

The Circus system replicates each program (or *module*) on multiple processors to form a *troupe*. The individual modules that compose the troupe are called *troupe members*. In Circus, it is assumed that troupe members are *deterministic* such that if each of the members starts in the same state, and each receives identical inputs, then they will all produce identical outputs and will all finish in the same state. This is the same determinism requirement that is made in the sender-based message logging mechanism being proposed in this paper. To ensure this determinism between multiple concurrently executing troupes, Circus uses a replicated atomic transaction protocol. The replication provided by the troupe is transparent to the programmer and is completely controlled and coordinated internally by Circus.

Module invocations are written as standard procedure calls, but internally, these calls are translated by the system into *replicated procedure calls* [Cooper84] that invoke the same procedure in each of the troupe members. When a client troupe makes a procedure call to a server troupe, *one-to-many* communication is used; when the server troupe replies, the communication is *many-to-one*. Since the troupe members are required to be deterministic, all members will execute the procedure call the same way, and each will return the same results. When a troupe sends a message to another troupe, one copy will be sent by each of the sending troupe's members since they are all actively executing. Normally, Circus uses a *unanimous* approach to handle these duplicate messages and to keep the sending troupe members synchronized in which it waits for all of the

duplicates to arrive before processing the message a single time.[9] Since this limits the progress of the program to the speed of the slowest member of each troupe, Circus also provides an alternate *first-come* approach where computation on a received message is allowed to begin when the first of the duplicate messages arrives. With this approach, servers must buffer replys to replicated procedure calls so that the same reply can be returned to all client troupe members, and clients must be willing to discard extra replies from replicated servers.

One advantage of the active backup processes mechanism is the speed with which recovery can be accomplished in case one of the processes fails, making this mechanism very useful in real-time systems. In the Circus system for example, when the unanimous approach to handling duplicate messages is used, all members of a troupe are always very nearly in the same state (within the same replicated procedure call, if one is active). If one of the troupe members fails, the rest of the troupe can continue immediately as if the failed member had never existed. However, since all members of each troupe actively execute, they each place a demand on the computational and other resources of the processor on which they are executing. Thus, the total resources required to execute a program is increased by a factor at least as large as the degree of replication over what would be required without the fault tolerance. Also, since entire troupes must communicate with each other, the level of communication on the network and the amount of overhead in each process to handle this communication is greatly increased.

9.6 Checkpointing

A mechanism commonly used to allow large sequential batch computations to recover from simple failures is the idea of *checkpointing*. A *checkpoint* of a process is an static record of the state of that process at some instance in time. When multiple concurrent processes communicate with each other, though, the use of checkpointing to achieve fault-tolerance becomes more difficult. Since the state of a process can change almost continuously as it communicates with other processes, the checkpoints must be updated regularly in order to remain current. The checkpoints are typically stored on disk, but the disk must be accessible to some other processor after the processor whose checkpoints are stored on it fails. In a system with many independent processors, the checkpoints can also be stored in the address space of other processors as inactive backup processes. Such checkpoints can be activated and begin execution if the primary process fails but are normally only static representations the same as more conventional disk checkpoints.

Although many systems use some form checkpointing to help provide fault tolerance, most of these only use this as a small part of a larger mechanism. A few systems have have been developed, however, that use only checkpointing to achieve fault tolerance. The Eden system at the University of Washington [Almes85a, Lazowska81] creates checkpoints as passive representations of Eden objects (each object is implemented by a separate process), and these checkpoints are automatically activated and begin execution as needed if the existing active object fails. The Tandem NonStop[10] system [Bartlett81, Dimmer85] creates process checkpoints on disk to provide fault tolerance; this system will be discussed in the remainder of this section as a representative system using the checkpointing mechanism.

---

[9] Circus normally compares these duplicate messages to check that they are all identical, as they should be since the troupe members are deterministic. This comparison is only used as a form of error detection, though; if majority voting was done to determine the "correct" message contents, though, this would be an example of the N-modular redundancy mechanism discussed in Section 9.4.

[10] Tandem, NonStop, Dynabus, and Guardian are trademarks of Tandem Computers Incorporated.

The Tandem NonStop system consists of a network of up to 255 nodes, each consisting of from two to sixteen independent processors. These processors are interconnected by replicated Dynabus interprocessor buses, which support message passing communication between the processors. Each processor runs an independent copy of the operating system called Guardian, which implements a process backup mechanism called a *process pair*. A process pair is composed of an active *primary* process on one processor and an inactive *backup* process on a second processor. The backup process is a normal, complete process in the system, but is kept inactive until needed. The backup process itself is the checkpoint of the primary process, and periodically, the primary process synchronously checkpoints its state to the backup process, making the state of the backup process an exact copy of the primary process. The checkpointing is synchronous in the sense that the primary process (and the backup process) are blocked from execution while the checkpoint is taking place.

If the primary process fails, the backup process becomes the new primary process and is activated. Since was maintained as an exact image of the primary process by the checkpoints, all that is necessary to activate it is for the operating system to begin scheduling it for execution. It will begin execution in the same state which the old primary had at the time its last checkpoint was taken. The new primary process can then optionally create a new backup process for itself on some new processor.

One problem with using checkpointing for fault tolerance is the frequency with which checkpoints must be made as the state of the primary process changes during execution. Since the process can only be recovered to the state of its last checkpoint, care must be taken to create a new checkpoint each time some important state change takes place, such as the receipt of a message from some other process. If checkpointing is not done carefully in these systems, a phenomenon called the *domino effect* [Randell75, Russell80, Russell77] may occur, as described in Section 2.3. To prevent the domino effect, checkpoints must record all communication with other processes. These frequent checkpoint operations can place a significant load on both the computational and communications resources of the system.

9.7 Message Logging

The *message logging* mechanism discussed in this section is the general basis for the sender-based message logging design being proposed in this paper. Here, each process is individually checkpointed periodically, but no attempt is made to keep these checkpoints completely current with every state change in each process. Instead, all messages between processes are logged, and these recorded messages are used to update the last checkpointed state in case of a failure. The state of a process can always be reconstructed from its most recent checkpoint, plus the messages which have been logged for it since that checkpoint. For this recovery to work correctly, the process being recovered must be *deterministic* in the sense that, if the process during recovery starts out in the same state that the failed process had at some point (the checkpoint), and it receives the identical input messages that the failed process received (the message log), it will produce the same output messages and reach the same state as the failed process at the time that the failure occurred; this is the same requirement of determinism assumed in the sender-based message logging design. Also, since the messages output by the process during recovery were already output by the failed process, these messages must not be not be acted on again by the other processes in the system.

This idea of message logging has been applied in a number of recent fault-tolerant systems. The ISIS system at Cornell University [Birman85] logs remote procedure call invocations in passive backup processes called *cohorts*, which become activated and re-execute the calls from the log in case of failure. The PUBLISHING mechanism, implemented within the DEMOS/MP system at

the University of California at Berkeley [Powell83b], uses a centralized recording process to log message traffic from the network so that messages for a failed process can be replayed to recover it. Message logging has also proven useful in the construction of debugging mechanisms for distributed systems [LeBlanc85a, Curtis82]. In the remainder of this section, the Auros[11] operating system [Borg83], developed by Auragen Systems Corporation, will be used as an example of a system using message logging.

The Auros operating system is a distributed version of UNIX[12] supporting message passing between processes independent of their location in the network. Each user process to be run with fault tolerance has an inactive backup process located on some other processor. Every message sent to a primary processes is also sent to a backup of that process, as well as to the backup process of the sender. Specialized hardware and support software guarantee that each message is either received by all three of these processes or by none of them, and that the messages will be received in the same order at all three of these processes with respect to messages from other senders.

At the primary process that the message was sent to, the message is simply used in the normal execution of the program as if no fault tolerance was involved. The copies of the message sent to the two backup processes, though, are an important part of the message logging mechanism. At the recipient's backup process, the message is saved in the normal incoming message queue which the kernel maintains, ready to be read if the backup should need to be activated on a failure of its primary; at the sender's backup, the message is used to keep a count of the number of messages which its primary has sent since the last checkpoint. Thus, if a primary process fails, its backup process has a queue of messages that need to reread to restore the backup to the state that the primary process had when it failed, and it has a count of messages that the primary process sent after its last checkpoint, which the backup should not resend during recovery.

Periodically, a primary process is automatically checkpointed to its backup process; in the Auros system, this is called *synchronization*. This normally occurs whenever the primary process has executed for some specified amount of time since its last checkpoint, or when it has sent some specified number of messages since the checkpoint. Completing the checkpoint allows the message queue at the backup process to be flushed since the state of the backup then represents the state of the primary after it had received all of these messages. If the primary fails after this checkpoint, none of the messages logged at the backup before the checkpoint need to be reread. The checkpoint also causes the backup process to clear its count of messages sent by the primary since the last checkpoint. After this, if the backup process must take over for the primary on a failure, none of its output messages will be duplicates of those already sent by the primary.

One problem with the message logging mechanism is that, if checkpoints are not performed frequently enough, the message log at the backup process can become very large, making storage management at the backup difficult. Also, if the message log is allowed to get too long, recovery after a failure can become slow since each of the logged messages must be reread by the backup, and each of these messages could initiate a large amount of computation by the recovering process. Performing the checkpoints automatically based on the execution time and number of messages sent as is done in Auros can help solve this problem, though.

The message logging mechanism does have some attractive advantages over the earlier mechanisms described. First, if the automatic checkpointing thresholds are tuned properly, the overhead in the system due to the checkpoints can be made negligible. Since less frequent checkpoints will make the time needed for recovery longer, this would not be suitable for real-time systems, but

[11] Auros is a trademark of Auragen Systems Corporation.

[12] UNIX is a trademark of AT&T Bell Laboratories.

such systems already require special treatment in other ways as well. Since Auros was designed for on-line transaction processing, short delays during recovery are allowable. Also, since the backup processes are inactive, the amount of overhead due to their presence in the system can be kept low.

9.8 Optimistic Recovery

A recent modification of the message logging techniques described above is the idea of *optimistic recovery* developed by Strom and Yemini at IBM [Strom85, Strom84c]. This mechanism, designed for the NIL project [Strom83, Strom84a], uses *asynchronous* message logging (together with checkpointing) rather than the *synchronous* logging used by others. This allows processes to perform computations based on received messages *before* those messages have actually been logged. This should allow for better response from the system since a receiving process need not wait for the message to be logged before it may begin processing it.

In general, an *optimistic algorithm* is one that temporarily assumes (guesses) that some highly-likely event will indeed occur, and proceeds with its computations as such. These computations, though, must not be made permanent until it is known that the assumption was correct. In case the assumption proves to be wrong though, the algorithm must be able to roll back any computations that depend on the assumption to their initial value at the time the assumption was made. Optimistic recovery gets its name from the optimistic assumption that it makes that the asynchronous message logging will "catch up" with the computations that are based on the delivery of those messages before a failure occurs. Optimistic algorithms have also been used in several other areas, including concurrency control mechanisms [Gherfal85] and the automatic parallelization of programs for multiprocessors and distributed systems [Strom84b].

Optimistic algorithms have the potential for performing better than non-optimistic ones since the optimistic assumptions that they make will usually be correct. If this happens sufficiently often, performance can be increased by using the optimistic algorithm. Since an optimistic algorithm is generally more complex and requires more overhead than the corresponding non-optimistic one, however, this performance improvement relies on the fact that the net gain in performance (the weighted probability average of the performance improvement when the assumption is correct and the performance loss when the assumption turns out to be incorrect) can overcome the costs associated with the more complicated algorithm.

In optimistic recovery, in order to be able to identify those computations that would need to be rolled back if a failure should occur, a *dependency vector* is appended to each message transmitted. This vector tracks the dependency of the state of the sending process on the states of the other processes with which it has communicated. The computation of each process is divided into *state intervals*, where a new state interval is begun when the process removes a new message from its incoming message queue and begins to process it. The dependency vector is simply a list for each other process of the index of the most recent state interval on which the state of the sending process now depends. When a process receives a new message, it merges its own dependency vector with the vector contained in the new message. The dependency vector for each process is also updated when a message that it has received (which means that it depends on the state of the sending process at the time the message was sent) has been logged.

If a process fails before some of the messages that it has seen have been logged, its current state can not be recovered. The process must be rolled back to the beginning of its most recent state interval for which the message that started it and all earlier messages received by that process have been logged. The state intervals that can not be recovered are called *lost*, and any computations (or state intervals) performed by other processes that depend on these lost states are called *orphans*.

Each of these processes that are now in orphaned states must also be rolled back to a state that does not depend on any lost states. So that other processes can know the status of the message logging for messages received by each process, a *log vector* is periodically broadcast by each process, which shows the logging status for all processes as far as the sending process knows it.

It is not clear how successful optimistic recovery will be since no implementation of it has been reported yet. The claim of increased performance due to the lack of synchronization between message logging and computation may be overshadowed by the increased complexity of the protocols. The dependency vector that must be added to each message and the periodic broadcasting of the log vector seem to greatly increase the communications cost in the system, although this communication is completely asynchronous with respect to the computation performed by each process. Also, since message logging may be arbitrarily far behind computation, a large number of checkpoints for each process may need to be kept (rather than just keeping the most recent). This will increase both the cost of performing each checkpoint and the amount of space on stable storage required to hold the checkpoints.

9.9 Summary

This section has presented a survey of the primary software mechanisms that have been used in existing systems or been proposed by others to achieve fault tolerance. These categories represent a fairly complete characterization of the existing solutions which have been reported in the literature. It is interesting, though, to note a broader characterization of the techniques presented here.

The first three mechanisms presented (application-specific mechanisms, functional programming, and atomic actions) can not be transparently applied to many existing distributed applications as sender-based message logging can. With the application-specific mechanisms (discussed in Section 9.1), this lack of transparency is caused by the large amount of knowledge of the possible failures and of the environment which must be designed into each program. With functional programming and atomic actions (presented in Sections 9.2 and 9.3), the problem is that a specialized model of computation is used in each to make it easier to achieve the fault tolerance. Since many existing distributed applications do not follow these computational models, these applications would need to be modified or rewritten to make use of these fault-tolerance mechanisms. In fact, these three fault-tolerance mechanisms generally rely on other mechanisms such as checkpointing to save the state of different processes, and as such are not fundamentally different from the other mechanisms discussed in this survey. They have been included here, though, because they use these other mechanisms in different ways and have often been considered to be independent fault-tolerance mechanisms by other researchers.

The remaining fault-tolerance mechanisms presented in this survey can be applied transparently to existing distributed applications, but they differ in their use of the available hardware in the system when no faults occur. The mechanisms of N-modular redundancy and active backup processes (described in Sections 9.4 and 9.5) use a *static* allocation of the extra computing power in the system. With static allocation, extra processors are assigned to each application for the duration of their execution, and the application processes actively execute redundantly on all of these processors. Since an up-to-date copy of each process exists on these processors, recovery after a failure can be almost immediate, making these mechanisms particularly well suited to real-time systems. Since sender-based message logging is not designed to operate in real-time systems, it can tolerate a slightly longer recovery time, making the dedication of these extra processors to each application unnecessary.

Conversely, the checkpointing, message logging, and optimistic recovery mechanisms (discussed in Sections 9.6, 9.7, and 9.8) use a *dynamic* allocation of the available computing power, as does sender-based message logging. These mechanisms execute each process actively on only one processor at a time, but maintain sufficient information elsewhere in the system to allow for recovery on another processor in case of failure. If this recovery information can be maintained efficiently, such a dynamic allocation can allow the extra computing power to be used to execute additional applications, off-loading the other processors in the system in the absence of failures. The sender-based message logging mechanism proposed here utilizes a method for maintaining this recovery information that should be simpler and more efficient in than those used in these other dynamic allocation mechanisms.

10 Conclusion

This paper has proposed research in the design and development of a new, low-overhead fault-tolerance mechanism for distributed systems called *sender-based message logging*. This mechanism can transparently allow processes in a distributed system to continue execution after a failure. It places very little overhead on the system in computation and communication costs and does not rely on the presence of special-purpose hardware to achieve fault tolerance. This appears to be the first fault-tolerance mechanism to be studied which uses the sender process itself as a message log rather than sending each message to a special logging process.

This research should result in a specific, detailed design for this new fault-tolerance mechanism. From this design plan, a prototype implementation will be completed and evaluated. Experience gained from this implementation should prove valuable in refining the design and should give evidence as to its correctness and efficiency. Finally, this prototype will be tested with existing distributed applications to verify its transparency and to measure the overhead of the fault tolerance on their execution. When completed, this work should result in a simple, low-overhead mechanism that can be utilized to achieve fault tolerance over a broad class of distributed systems and distributed applications.

References

[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.

[Almes85a] 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.

[Almes85b] Guy T. Almes and Willy Zwaenepoel. *Understanding and Exploiting Distribution.* Technical Report Rice COMP TR85-12, Department of Computer Science, Rice University, Houston, Texas, February 1985.

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

[Anderson85] T. Anderson. Fault tolerant computing. In T. Anderson, editor, *Resilient Computing Systems*, chapter 1, pages 1–10, Collins, London, 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.

[Bernstein81] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

[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. An earlier version is available as Technical Report 85-668, Department of Computer Science, Cornell University, Ithaca, New York, March 1985.

[Boehm86] Hans-Juergen Boehm and Willy Zwaenepoel. *Parallel Attribute Grammar Evaluation.* Technical Report Rice COMP TR86-39, Department of Computer Science, Rice University, Houston, Texas, May 1986.

[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.

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

[Chang84] Jo-Mei Chang and N. F. Maxemchuck. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[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.

[Cheriton86] David R. Cheriton and Michael Stumm. The multi-satellite star: structuring parallel computations for a workstation cluster. To appear in *Distributed Computing*, 1986.

[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.

[Curtis82] R. Curtis and L. Wittie. BugNet: a debugging system for parallel environments. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 394–399, IEEE Computer Society, October 1982.

[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 T. Anderson, editor, *Resilient Computing Systems*, 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.

[Finkel85] Raphael Finkel and Udi Manber. DIB—a distributed implementation of backtracking. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 446–452, IEEE Computer Society, May 1985.

[Friedman78] Daniel P. Friedman and David S. Wise. Aspects of applicative programming for parallel processing. *IEEE Transactions on Computers*, C-27(4):289–296, April 1978.

[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 R. Bayer, R. M. Graham, and G. Seegmüller, editors, *Operating Systems: An Advanced Course*, 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.

[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.

[Jensen82] E. Douglas Jensen. Decentralized executive control of computers. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 31–35, IEEE Computer Society, October 1982.

[Johnson85] David B. Johnson and Willy Zwaenepoel. Macropipelines on a network of personal workstations. In preparation, Department of Computer Science, Rice University, Houston, Texas, 1985.

[Keller85] Robert M. Keller and Gary Lindstrom. Approaching distributed database implementations through functional programming concepts. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 192–200, IEEE Computer Society, May 1985.

[Kopetz85a] H. Kopetz. Resilient real-time systems. In T. Anderson, editor, *Resilient Computing Systems*, 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.

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

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

[Laprie85] Jean-Claude Laprie. Dependable computing and fault tolerance: concepts and terminology. In *The Fifteenth Annual International Symposium on Fault-Tolerant Computing: Digest of Papers*, pages 2–11, IEEE Computer Society, June 1985.

[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.

[LeBlanc85a] Richard J. LeBlanc and Arnold D. Robbins. Event-driven monitoring of distributed programs. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 515–522, IEEE Computer Society, May 1985.

[LeBlanc85b] 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.

[LeLann81] Gerard LeLann. Error recovery. In B. W. Lampson, M. Paul, and H. J. Siegert, editors, *Distributed Systems: Architecture and Implementation*, chapter 15, pages 371–376, Springer-Verlag, New York, 1981.

[Lin86] Frank C. H. Lin and Robert M. Keller. Distributed recovery in applicative systems. In Kai Hwang, Steven M. Jacobs, and Earl E. Swartzlander, editors, *Proceedings of the 1986 International Conference on Parallel Processing*, pages 405–412, IEEE Computer Society, August 1986.

[Liskov79] Barbara Liskov. Primitives for distributed computing. In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 33–42, ACM, December 1979.

[Liskov82] Barbara Liskov. On linguistic support for distributed programs. *IEEE Transactions on Software Engineering*, SE-8(3):203–210, May 1982.

[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.

[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.

[Powell83a] Michael L. Powell and Barton P. Miller. Process migration in DEMOS/MP. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 110–119, ACM, October 1983.

[Powell83b] 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.

[Randell79] B. Randell. Reliable computing systems. In R. Bayer, R. M. Graham, and G. Seegmüller, editors, *Operating Systems: An Advanced Course*, chapter 3. E., pages 282–391, Springer-Verlag, New York, 1979.

[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.

[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.

[Schwarz84] Peter M. Schwarz and Alfred Z. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2(3):223–250, August 1984.

[Serlin85] Omri Serlin. Fault tolerant blues. *Datamation*, 31(6):82–88, 15 March 1985.

[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.

[Smith85] A. P. Smith. The MOMENTUM high resilience system. In T. Anderson, editor, *Resilient Computing Systems*, chapter 11, pages 197–207, Collins, London, 1985.

[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.

[Strom83] Robert E. Strom and Shaula Yemini. NIL: an integrated language and system for distributed programming. In *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, pages 73–82, ACM, June 1983. Also available as Research Report RC 9949, IBM T. J. Watson Research Center, Yorktown Heights, New York, April 1983.

[Strom84a] Rob Strom and Shaula Yemini. *The NIL Distributed Systems Programming Language: A Status Report.* Research Report RC 10864, IBM T. J. Watson Research Center, Yorktown Heights, New York, December 1984.

[Strom84b] Rob Strom and Shaula Yemini. *Synthesizing Distributed and Parallel Programs through Optimistic Transformations.* Research Report RC 10797, IBM T. J. Watson Research Center, Yorktown Heights, New York, July 1984.

[Strom84c] Robert E. Strom and Shaula Yemini. Optimistic recovery: an asynchronous approach to fault-tolerance in distributed systems. In *The Fourteenth International Conference on Fault-Tolerant Computing: Digest of Papers*, pages 374–379, IEEE Computer Society, June 1984.

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

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

[Tanenbaum81] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[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.

[Weihl83] William Weihl and Barbara Liskov. Specification and implementation of resilient, atomic data types. In *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, pages 53–64, ACM, June 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.

[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.