TRANSPARENT PROCESS ROLLBACK RECOVERY: SOME NEW TECHNIQUES AND A PORTABLE IMPLEMENTATION

A Thesis

by

ERNEST LLOYD ELLENBERGER

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 1995

Major Subject: Computer Science

TRANSPARENT PROCESS ROLLBACK RECOVERY: SOME NEW TECHNIQUES AND A PORTABLE IMPLEMENTATION

A Thesis

by

ERNEST LLOYD ELLENBERGER

August 1995

ABSTRACT

Transparent Process Rollback Recovery: Some New Techniques and a Portable Implementation.

(August 1995)

Ernest Lloyd Ellenberger, B.S., Case Western Reserve University

Chair of Advisory Committee: Dr. Nitin H. Vaidya

Processes in a distributed system can be made *transparently recoverable* through the use of process checkpointing, which periodically introduces a relatively large but temporary overhead, or message logging, which introduces a smaller overhead for every message sent, and requires process execution to be deterministic. Research prototypes have shown promising performance results [12, 7], but an implementation has not been readily available. To address that deficiency, this thesis describes the design, implementation, and failure-free performance of a new transparent recovery system for standard Unix workstations, which provides a basis for future experimental work.

The system incorporates both coordinated checkpointing and family-based message logging with the *logging site* technique for reducing message logging overhead when some processes share a common memory address space (recently suggested independently by Vaidya [30] and Alvisi and Marzullo [2]). Performance measurements show the overhead of the consistent checkpointing and family-based message logging implementations to be reasonably small for a representative distributed application.

This thesis also presents a new approach for efficient output commit and recovery when some processes are *intermittently nondeterministic.* The approach requires the application program to explicitly mark the beginning and end of each period of nondeterminism. Message logging is used during deterministic execution, but is disabled and replaced with optimistic checkpointing during periods of nondeterminism. The commit algorithm avoids communication with deterministic processes by using causal dependency information that makes a distinction between nondeterministic and deterministic state intervals.

Finally, the concept of *reactive replication* for message logging is introduced. During failure-free operation, reactive replication uses a low-overhead protocol that can tolerate only a single simultaneous failure, such as family-based message logging. If a failure occurs, the logged data necessary to recover the failed process is immediately copied to stable storage or to the volatile storage of another processor; a second failure can be tolerated only after the completion of that copy operation. This technique assumes that such a copy operation is faster than the recovery protocol.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

Introduction

I.A Outline

As the size of a distributed system and the running time of a distributed computation increase, so does the probability that a failure will occur during the computation. Techniques for providing *transparent rollback-recovery* to processes in distributed systems aim to hide fault-tolerance issues from applications while imposing acceptably low overhead. The contributions of this thesis are in the following areas:

- The **efficient coexistence of message logging and nondeterministic execution** is addressed in Chapter III. Nondeterministic process execution, such as unsynchronized modification of a shared resource by multiple processes, can be made recoverable by checkpointing but not by message logging. Message logging protocols generally exhibit better output commit performance than protocols that use checkpointing alone, however. A recovery technique that uses message logging during deterministic execution and checkpointing during nondeterministic execution of the same process is presented. An orphan-free message logging protocol is assumed to be in use at all deterministic processes, so that output commit only requires the nondeterministic processes upon which the state interval being committed is dependent to be made stable (i.e. checkpointed). An output commit algorithm that uses that observation to reduce communication and checkpointing overhead is presented, as is a method for recovering the system state after a failure.

- **Reactive replication in message logging** is a technique for allowing a message logging protocol to tolerate multiple overlapping failures with no more failure-free overhead than would be required to tolerate only a single overlapping failure, with the restriction that every two

The journal model is *IEEE Transactions on Computers*.

failures must be separated by some minimal interval of time that is sufficient for the non-failed process to prepare for another failure by replicating data crucial to the recovery of the failed process. The reactive technique can improve performance only if the replication completes before recovery completes and before a second failure occurs. Reactive replication and its use in family-based message logging are described in Chapter IV.

- An **implementation** suitable for experimenting with new recovery protocols. The implementation includes a library of routines for Unix that provide a message passing abstraction, as well as the failure-free portion of protocols for coordinated checkpointing, family-based message logging, and a modified version of family-based logging that maintains the logs for all processes on a processor in shared memory. Chapter V describes the design and implementation of the system.

- Empirical performance measurements of the implementation are given in Chapter VI.

The main focus of the work for this thesis has been on creating an implementation suitable for use in evaluating recovery methods, as opposed to evaluating any specific recovery method. The implementation is intended as a platform for empirically evaluating recovery methods and is structured so that adding new recovery protocols is straightforward. The software itself is freely-available.

I.B System Model

The system consists of a set of processes executing on a set of processors. Multiple processes may reside on a single processor. Processes on the same processor communicate through message passing. The processes execute at arbitrary speeds relative to each other and do not have synchronized clocks. The processes fail according to the fail-stop model [24]. FIFO channels are assumed, but the order of receipt of messages at a single process from two other processes is random (that random order does not make a process itself nondeterministic, however). Communication is assumed to be reliable. and stable storage (such as a disk guaranteed to survive failures) is assumed to be available.

CHAPTER II

Related Work

II.A Rollback Recovery in Distributed Systems

A distributed system consists of a set of processes that communicate only by message passing. A program that executes in a distributed system is called a *distributed application* and is said to perform a *distributed computation*. As the size of a distributed system and the running time of a distributed computation increase, so does the probability that a failure caused by some external event will result in the loss of at least one process's state during the computation.

In the absence of any provisions for recovery, even a single such failure can place the distributed computation into a state from which execution cannot proceed correctly, and from which the only recourse is to restart the computation from the beginning. Techniques for providing *transparent rollback-recovery* to processes in distributed systems aim to hide fault-tolerance issues from applications while imposing acceptably low overhead. Recovery techniques reduce the amount of computation that will be lost if a failure occurs, at the expense of increased overhead during failure-free execution. Two different approaches to recovery have been investigated in recent years: *backward* recovery, and *forward* recovery. Backward recovery protocols record information during failure-free execution that can be used to restore the system to a recent state; if a failure occurs, the failed process, and possibly some other processes, will be *restarted* from the most recent saved state. The recovery protocol will generally be able to restore a state with relatively little delay, compared to the time that would be required to re-execute all processes from the beginning of execution. Forward recovery protocols employ multiple concurrently-executing *replicas* of each process, and failures cause negligible delay in execution as long as at least one replica of each process survives every failure. The short recovery time of the forward approach comes at a substantial cost in computation resources, because each replica must execute the computation. In contrast, the backward approach imposes only a small computation overhead, but cannot guarantee immediate recovery and is hence not well-suited to real-time applications. This thesis deals with checkpointing and message logging, which are both backward-error-recovery techniques.

In distributed systems, the recovered global state must be *consistent*. A consistent global state is one that could occur during a failure-free run of the application program [5]. If the receipt of a message m by some process is recorded in the global state, then the sending of m must also be recorded in the state for the state to be consistent.

II.B Checkpointing in Distributed Systems

Numerous checkpointing and recovery protocols have been developed for recovering a system to a consistent state. There are two distinct approaches:

- *Coordinated checkpointing* [5, 27, 17, 18, 7] requires all processes to synchronize their checkpoints so that the state contained in the union of all the checkpoints (a *global checkpoint*) is a consistent global state. The recovery protocol can restore the system to a consistent global state by rolling back each process at most once, since a consistent global state is guaranteed to exist. Checkpoints older than the most recent can be discarded. The overhead of coordination and the nearly simultaneous occurrence of all checkpoints, which is likely to result in contention for the network and stable storage, are the primary drawbacks of coordinated checkpointing.

- *Independent checkpointing* [3, 12, 14, 22, 29, 32, 33] allows the processes to take checkpoints independently, whenever they choose. The recovery protocol must attempt to construct a consistent global state from the checkpoints that are available, so it may be necessary to roll a process back past multiple checkpoints (the occurrence of such is called the *domino effect* [21] because rollbacks may propagate in a chain reaction). Since independent methods do not guarantee that the most recent checkpoints of all processes form a consistent global state, they must maintain more than one previous checkpoint for each process to increase the probability that a consistent global checkpoint will exist. Garbage collection is thus necessary to limit the number of stored checkpoints. An obvious conclusion is that the failure-free overhead of independent checkpointing should be less than that of coordinated checkpointing, but measurements of implementations of both techniques in the Manetho system [10] indicate that the two techniques perform similarly.

II.C Message Logging

The messages exchanged in a distributed computation entirely define the computation if the computation is assumed to be *deterministic*. A computation is deterministic up to a given point in its execution if its state at that point depends only on its initial state and the sequence of messages it has received up to that point. It follows that such a computation can be recorded by logging the messages it exchanges to a storage medium that survives failures.

Pessimistic message logging protocols [4, 20] wait (and block the application from proceeding on the local processor) until the message logging operation finishes successfully. The log must contain any message that the system is to process, so pessimistic protocols generally use either an atomic *deliver and log* or *send and log* operation.

Optimistic message logging techniques [29] are *non-blocking* because they log messages *asynchronously*, without waiting for the logging operation to complete before allowing the local application to continue. The benefit of optimism is an expected reduction in failure-free overhead because of the elimination of waiting for logging operations, at the expense of a more complicated and slower recovery protocol which may roll back non-failed processes. Optimistic message logging can be done either by the sender [28, 13, 10], the receiver [29, 14, 26, 16], or both (the latter is generally not done).

The *Sender-based message logging* protocol [13, 12] requires that each message be logged in the volatile storage of its sender, and the sender must wait until an acknowledgment has been received from the receiver before sending another message. This protocol is pessimistic because of this synchronization. Independent checkpoints are taken periodically, and only one checkpoint need be retained for each process. A crucial assumption of sender-based message logging is that only a single process will fail at a time. The protocol can detect the occurrence of multiple simultaneous failures, but cannot recover from them. In the event of a single failure, only the failed process need be rolled back. When rollback occurs, the process is restored from its checkpoint, and the messages it had received after the checkpoint was taken are replayed to it by their senders. Strom, Bacon, and Yemini suggest an extension of sender-based logging to tolerate n simultaneous faults [28, 12].

The Manetho message logging protocol [9, 7] extends the idea of sender-based message logging by appending to each message an *antecedence graph* that represents all the events (including message receipt and nondeterministic events) that "happened before" the message. The antecedence graph contains sufficient information recover from an arbitrary number of simultaneous failures.

The Manetho protocol is non-blocking, yet it does not suffer from the drawbacks of traditional optimistic protocols. Recently, Alvisi et al. have given a definition of a class of message logging protocols, called *family-based* protocols, that are neither optimistic nor pessimistic but are are non-blocking and do not roll back non-failed processes. Family-based message logging (which is also called *optimal* logging) [1, 2] logs the contents of each message in the volatile storage of the sender and logs the processing order in the volatile storage of the next process that receives a message from the receiver (the receiver's processing order does not affect any other processes until it has sent a message that depends on that order). Family-based protocols require some communication to retrieve logged information from other processes during recovery, so pessimistic protocols are the best choice for applications that require fast recovery [11]. In general, no single protocol is optimal for all possible applications.

Family-based protocols can tolerate f overlapping (i.e. concurrent) process failures, where f is an integer between one and the number of processes in the system (n). Larger values of f require more information be appended onto application messages, but these protocols do not introduce any additional non-application messages. The Manetho protocol is a family-based protocol with $f = n$ [2].

Many existing recovery techniques combine both checkpointing and message logging. During failure-free operation, the instantaneous state of each process can be saved periodically in a checkpoint, and messages can be logged between checkpoints. Since taking a checkpoint requires much more overhead than logging a message, increasing the time between checkpoints can increase performance. Checkpoints can be used to bound the size of message logs or to allow nondeterministic computation.

A distributed application may need to communicate with *external* processes that can neither be rolled back nor expected to re-send messages. For example, the actions of an automated teller machine cannot generally be rolled back. Hence any process that does not comply with the requirements of the recovery protocol is considered to be part of the *outside world.* A message sent to the outside world is called an *output message*, and a message received from it is called an *input message.* Before an output message can be transferred to the outside world the recovery system must ensure that the message's send event will never be "undone." The term *output commit* refers to any procedure that ensures that the sending of such an output message will never need to be undone. The outside world cannot be relied upon to re-send a message either, so an input message must also be treated specially. Pessimistic protocols can commit output messages immediately because they maintain "up to date" message logs at all times. Family-based protocols must log any ordering information for a message before it can be committed, thereby requiring a generally small but rarely zero delay. Finally, optimistic protocols must ensure that no other process on which the output message depends will ever be rolled back, and the communication necessary to guarantee that property implies a possibly large output commit latency.

II.C.1 Performance

In some recent measurements of implementations of message logging protocols in the Manetho system [10], coordinated checkpointing without message logging was found to perform better in terms of failure-free overhead than all of the message logging protocols measured. Furthermore, coordinated checkpointing with message logging always performed better than independent checkpointing with message logging. However, the time to commit output can be reduced by a message logging protocol. The output commit latency of the Manetho message logging protocol is considerably less than that of a pure coordinated-checkpointing protocol for the implementations measured [10]. From these measurements it seems that the complexity of message logging is only outweighed by performance benefits if output commit latency is an important aspect of performance. The performance measurements of the Manetho system should not be taken as the last word on the performance of these techniques, however. The measurements are of the impact of the Manetho protocol on the execution times of four different benchmark application programs (these same programs will be used in this thesis and are described in Chapter VI).

II.D Family-Based Message Logging

Family-based logging (FBL) protocols [1, 2] never block the sender of a message and never introduce orphan processes during recovery. FBL protocols can be *optimal* in the sense that they introduce no additional messages compared to an equivalent protocol that does not log messages but uses a simple acknowledgment scheme to tolerate transient link failures.

The information necessary to recreate the delivery of a message m is only logged when the delivery of m at process p can causally affect some other process q, i.e., only after p has sent a subsequent message m' to q. A message m's data is logged only in the volatile storage of the sender, and the delivery order information for the message, which consists of the tuple $<m.source, m.ssn, m.dest, m.rsn>$ (called m's *determinant*, abbreviated as $\#m$), is logged at the receiver's receiver, i.e., at the next process q that delivers a message m' from the receiver p after p delivers m. The determinant $\#m$ is piggybacked on subsequent messages sent by process q.

To tolerate f overlapping failures, the determinant $\#m$ must be logged at every process that delivers a message from q until the total number of different receiver processes that has logged $\#m$ (denoted by $|m.log|$, where $m.log$ is the set of processes at which $\#m$ is logged) becomes equal to f. Once $|m.log|$ becomes greater than or equal to f, the determinant $\#m$ no longer need be piggybacked on messages, and subsequent receivers need not add it to their logs. The value of $|m.log|$ is not always known by any single process, however, so FBL protocols must estimate it based on the limited information that they do have about the other processes at which $\#m$ has been logged. If a process overestimates $|m.log|$ then it will unnecessarily log $\#m$. A process must never underestimate $|m.log|$. Alvisi and Marzullo [2] describe three classes of estimation protocols that differ in the amount of additional information that is piggybacked on messages, and they show that the accuracy of a process's estimate of $|m.log|$ can be increased by increasing the amount of piggybacked information. However, they do not address the possibility of introducing additional messages (which would violate their definition of "optimal") into the protocol to increase the estimate's accuracy. There is a possibility that the cost of the piggybacked information is greater

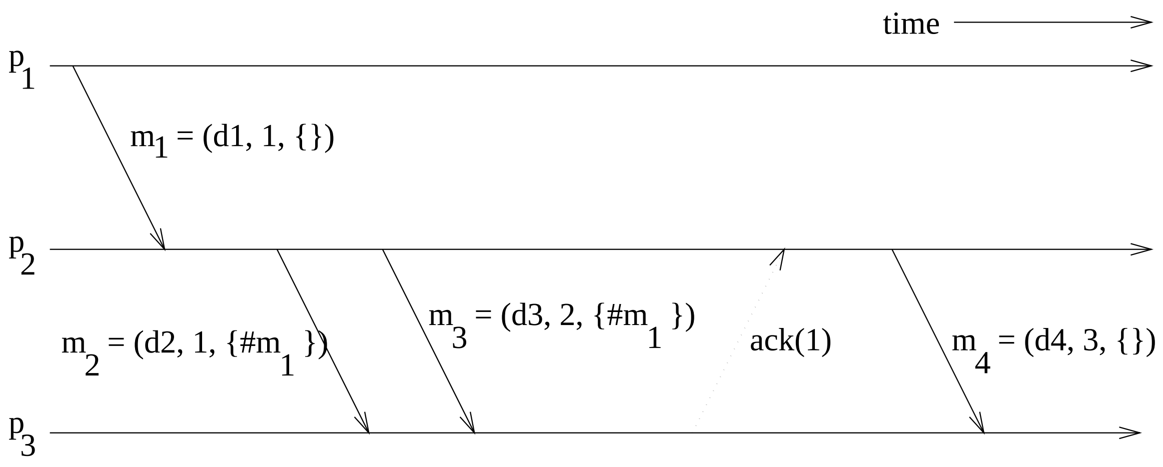

Fig. 1. Family-Based Logging for $f = 1$.

than the cost of additional messages for values of $f > 2$.

An example execution of a family-based logging protocol that can tolerate a single overlapping failure ($f = 1$) is shown in Figure 1. In the figure, each application message m is a triple $(\mathit{data}, \mathit{ssn}, \mathit{piggyback})$, where *data* is m's data, *ssn* is m's send sequence number, and *piggyback* is the set of determinants piggybacked on m. For $f = 1$, the determinant (i.e. receipt order information) of a message m_i is $\#m_i = <m_i.\mathit{source}, m_i.\mathit{ssn}, m_i.\mathit{rsn}>$.

The determinant $\#m_1$ is piggybacked on all messages sent by process p_2 until p_2 learns that $\#m_1$ has been received (and hence logged) by at least one other process. The acknowledgment *ack(1)* of m_2 (p_2's SSN 1) informs p_2 that $\#m_1$ (the information piggybacked on m_2) has been received and logged by p_3. Since $f = 1$, $\#m_1$ need not be piggybacked on any subsequent messages p_2 sends. The acknowledgments of m_2, m_3, and m_4 have not yet arrived and are not shown in the figure.

Alvisi [2] extends FBL to $f > 1$ overlapping failures by requiring a process to piggyback a determinant $\#m$ on every message until the process learns that at least f processes (instead of just one process, as is done in the case of $f = 1$) have received (and hence logged) $\#m$. In Figure 1, $\#m_1$ would be piggybacked on m_4 for $f \geq 2$, because p_2 only receives a single acknowledgment before sending m_4. Furthermore, p_3 would also piggyback $\#m_1$ on any messages it sends until it learns that $\#m_4$ is logged at at least f processes. Alvisi's FBL protocols for $f > 1$ use a mechanism that maintains “weak dependency vectors”, which are updated from the determinants piggybacked on each received message, to determine when it is safe to stop piggybacking a given determinant on outgoing messages.

The FBL protocol presented by Alvisi [2] uses a data structure called a *DetLog* (determinant log) at each process to record the determinants logged at the process, and a *SendLog* at each process to record the message data of each message sent by the process.

CHAPTER III

Recovery and Efficient Output Commit in the Presence of Nondeterminism

III.A Introduction

Nondeterministic process execution, which might arise from unsynchronized modification of a shared resource or any other random choice, can be made recoverable by checkpointing but not by message logging. Message logging protocols generally exhibit better output commit performance than protocols that use checkpointing alone, however [10]. This chapter presents a recovery technique that uses message logging for deterministic state intervals and checkpointing for nondeterministic state intervals of the same process and explicitly tracks dependencies on nondeterministic processes. Because the technique only requires nondeterministic processes to checkpoint as part of an output commit operation, but not necessarily at any other time, a failure can introduce orphan processes that must be rolled back. Processes should checkpoint at certain times such that a balance is achieved between rollback distance and checkpointing overhead. Additional checkpoints over those required as part of output commits is likely to decrease rollback distance, but the choice of when such checkpoints should be taken is an orthogonal issue that is not dealt with here.

An orphan-free message logging protocol is assumed to be in use at all deterministic processes, so that output commit only requires the nondeterministic processes upon which the state interval being committed is dependent to be made stable (i.e. checkpointed). An output commit algorithm that uses that observation to reduce communication and checkpointing overhead is presented, and a method for recovering the system state after a failure is described.

III.B Definitions

This chapter uses Johnson's [15] definitions of nondeterministic execution and stable and committable state intervals. A state interval is called *stable* if and only if its process can be restored to some state in the interval, either from the information available on stable storage or available from other processes. A state interval σ of process i is called *committable* if and only if it will never be rolled back, which is true if and only if there exists some recoverable system state in which process i is in some state interval $\alpha \geq \sigma$ [12, 15]. Before a message can be sent to the outside world, the state interval from which it is being sent must be committable.

A process state interval is *nondeterministic* if it cannot necessarily be reproduced by restarting the process in the state it had at the beginning of the interval [15]. During nondeterministic execution, a process creates a new state interval before sending a message. A nondeterministic state interval can only be made stable by a checkpoint. A nondeterministic state interval $\alpha \leq \sigma$ becomes stable immediately after a successful checkpoint of σ if both α and σ are in the same process (checkpoints are discussed in Section III.H).

The application must inform the recovery system of an impending switch to nondeterministic execution by executing the **BeginND** event before becoming nondeterministic. The **EndND** event is assumed to be executed by a process some time after a **BeginND** to mark the end of nondeterministic execution. The **EndND** event does not necessarily need to occur, but can bound potential rollback distance after a checkpoint, since the subsequent deterministic execution can be made recoverable by message logging.

All state intervals of a process after a **BeginND** but before the matching **EndND** are defined to be nondeterministic unless they happened before a checkpoint of the process. All state intervals after an **EndND** but before a subsequent checkpoint are nondeterministic until a subsequent checkpoint is taken, assuming that a second **BeginND** does not happen before the checkpoint. A *nondeterministic process* is one that has executed **BeginND** before its current state interval, but has not executed **EndND** between its most recent **BeginND** and its current state interval.

A nondeterministic process is required to take a checkpoint upon the request of the commit algorithm. Nondeterministic processes need not checkpoint at any other particular time, and checkpoints of deterministic are never required. The recovery algorithm uses the most recent available checkpoints for nondeterministic processes to construct the most recent recoverable state, so frequent checkpointing reduces rollback distance. However, the choice of a particular checkpointing policy or periodic interval is not addressed by the technique presented here. The optimum interval may depend on numerous parameters including application characteristics, and is beyond the scope of this work.

The output commit algorithm is described in Section III.F, and the recovery algorithm is described in Section III.H

III.C Motivation

A given state interval σ can be committed by ensuring that all (remote) process state intervals upon which σ causally depends are stable. Johnson's **Commit** algorithm [15] communicates with the minimum number of other processes necessary to commit a given state interval, where the set of processes that must be stable (and hence must be communicated with) is determined by the dependencies of the process being committed.

The idea behind the technique presented here is that only the *nondeterministic* state intervals upon which σ is dependent must be stable for σ to be committed if all deterministic processes are always stable. In particular, a deterministic process is always stable if it participates in an orphan-free (e.g. pessimistic [4, 20] or optimal [9, 7, 1, 2]) message logging protocol. Reducing the set of state intervals that must be made stable will, in general, reduce the number of processes that must be communicated with to commit a state interval, and the reduced communication will yield improved performance.

III.C.1 The Need for a New Type of Dependency Tracking

To make a state interval σ committable, it is necessary to know the identities of all remote process state intervals upon which σ is dependent. Ordinary direct dependencies would suffice for identifying all processes upon which σ is dependent. However, to find all such processes by using ordinary direct dependencies, it is necessary to communicate with each process indicated in the dependency, and then follow the dependencies of those directly dependent processes, and so on, until all dependent processes are found. But the point of the output commit approach suggested

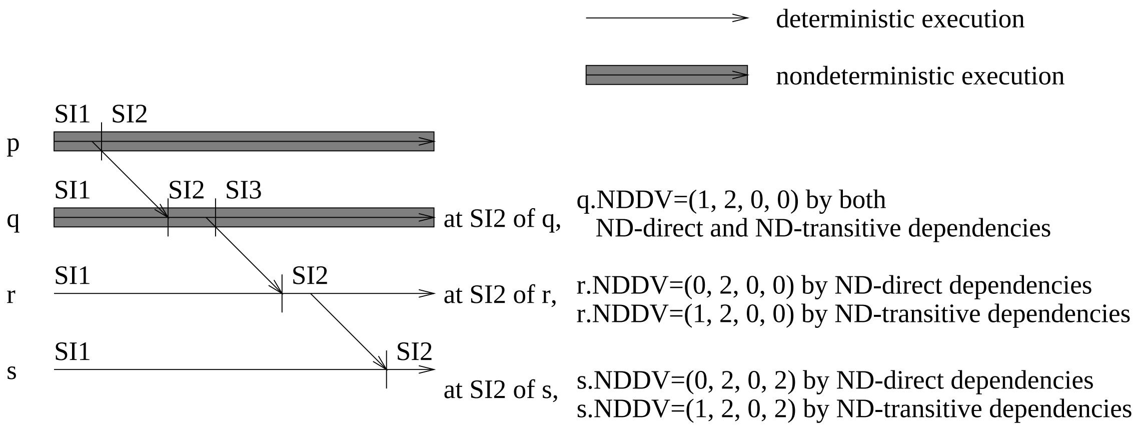
deterministic execution
nondeterministic execution
p
q
r
s
SI1
SI2
SI1
SI2
SI3
SI1
SI2
SI1
SI2
at SI2 of q,
q.NDDV=(1, 2, 0, 0) by both ND-direct and ND-transitive dependencies
at SI2 of r,
r.NDDV=(0, 2, 0, 0) by ND-direct dependencies
r.NDDV=(1, 2, 0, 0) by ND-transitive dependencies
at SI2 of s,
s.NDDV=(0, 2, 0, 2) by ND-direct dependencies
s.NDDV=(1, 2, 0, 2) by ND-transitive dependencies

Fig. 2. The *NDDV* dependency vectors at processes q, r, and s

here is to avoid communication with deterministic processes upon which σ is dependent, since they are always stable and never need to checkpoint. Thus a type of dependency tracking that includes all necessary nondeterministic state intervals but excludes deterministic state intervals is necessary. Ordinary transitive dependencies are inappropriate for use here as well, because they record *all* dependencies.

III.C.2 ND-Transitive Dependencies

The first type of dependency tracking introduced here is called ND-transitive. With ND-transitive dependencies, *all* nondeterministic processes upon which a process p is causally dependent are always known to p. The example execution in Figure 2 compares the dependency vector *NDDV* of dependencies on nondeterministic state intervals produced by the ND-transitive definition to the *NDDV* produced by the ND-direct definition.

A commit algorithm can use ND-transitive dependencies to immediately inform the processes that have a state intervals upon which the state interval being committed is dependent that those state intervals must now be made stable.

III.C.3 ND-Direct Dependencies

ND-transitive dependency tracking provides more information than is strictly necessary to commit a given state interval. The *ND-direct* dependency tracking introduced here provides only the information necessary to "look up" the full set of state intervals upon which a state interval is causally dependent. ND-direct dependencies impose less overhead in terms of message size but greater overhead in terms of output commit latency as compared to ND-transitive dependencies.

The information that ND-direct dependencies provide is the identities of all nondeterministic processes upon which p is causally dependent such that the dependency chain leading back to a nondeterministic process consists of only deterministic processes. A commit algorithm can use that information to which tracks transitive dependencies on deterministic processes back to the "most recent" nondeterministic process that is a causal ancestor of p. Figure 2 compares the dependency vectors ($NDDV$) produced by ND-direct and ND-transitive dependencies.

The difference between ND-direct dependencies and ordinary direct dependencies is that in general there may be a causal chain of any number of deterministic processes separating a process from its most recent nondeterministic causal ancestor. Hence direct dependencies alone are not sufficient to track the most recent causal ancestor, and the ND-direct definition is introduced.

A commit algorithm that uses only direct dependencies was suggested by Johnson [15]. The basic idea is as follows: A commit algorithm can use ND-direct dependencies to commit a state interval σ by first informing the remote processes that have state intervals upon which σ is ND-directly dependent that they must make those state intervals stable. That activity constitutes the first *round* of the commit algorithm.

In addition, those remote processes return the dependencies of those state intervals to the process executing the commit algorithm, and the commit algorithm then executes a second round to inform any new processes that have state intervals (upon which σ is indirectly dependent) not previously made stable that those state intervals must now be made stable. After that, additional rounds may be executed until the initiator finds that all state intervals upon which σ is causally dependent have been made stable. [1] Ordinary transitive or ND-transitive dependencies provide full dependency information, and a commit algorithm that uses them only executes a single round. In summary, the absence of full transitive dependency information can be overcome if sufficient dependency information exists for the commit algorithm to determine the transitive dependencies by executing multiple rounds.

Although a commit algorithm that uses ND-direct dependencies must communicate with other remote processes to determine the identities of all nondeterministic processes upon which σ is dependent, those remote processes are limited to be *only* nondeterministic processes. The motivation for introducing ND-direct dependencies is that less information is exchanged in maintaining them than in maintaining ND-transitive dependencies, and that ND-direct dependencies still limit communication during a commit of σ to only the nondeterministic processes upon which σ is dependent.

The ND-direct dependencies suggested here provide sufficient information for a multi-round commit algorithm to determine the identities of all nondeterministic state intervals upon which a process is dependent. A multi-round algorithm is likely to exhibit greater latency than the single-round algorithm, although ND-transitive dependencies are required by a single-round algorithm.

Both ND-direct and ND-transitive dependencies allow the commit operation to avoid communication with the deterministic processes upon which the state interval being committed is dependent. Existing commit algorithms [15] do not distinguish between deterministic and nondeterministic processes when deciding which processes to communicate with. Hence the method suggested here will have lower communication overhead than existing commit algorithms.

The rest of this chapter is organized as follows. Methods for tracking the nondeterministic state intervals upon which a process depends are described in Section III.D, and the use of those nondeterministic dependencies to improve the performance of Johnson's Commit algorithm is described in Section III.F. Section III.G assesses the degree to which the overhead of a consistent checkpointing algorithm, in terms of communication and number of checkpoints, can be reduced when only

[1] Johnson claims that up to N rounds may be executed, where N is the number of processes in the system, but on the average the number of rounds will be small [15].

nondeterministic processes need be checkpointed.

The technique suggested here allows nondeterministic processes to send messages without checkpointing, which means that a process that delivers a message from a nondeterministic process is susceptible to being orphaned by the failure of the sender until the sender takes a subsequent checkpoint. Here it is optimistically assumed that a failure which will cause a temporary possible orphan process to be rolled back is unlikely to occur. When output must be sent to the outside world, a commit algorithm must be used to ensure that the state interval from which the output message is sent will never be orphaned.

III.D Maintaining Dependencies on Nondeterministic Processes

The causal dependencies of a process (more precisely, the causal dependencies between state intervals of different processes) can be maintained by direct dependency tracking, which adds a single process number to each message, or transitive dependency tracking [29], which adds a vector of length N to each message. Both methods track dependencies on *all* processes, with no distinction between deterministic and nondeterministic processes.

Here it is assumed that messages may be sent from non-stable nondeterministic state intervals; a process that delivers such a message is a possible orphan until the sender becomes stable (i.e. is checkpointed).

Figure 3 shows an example execution in which messages are sent from nondeterministic state intervals and the consequent nondeterministic dependencies. Dependencies on deterministic state intervals are not labeled in the figure, and the example does not contain any ND-transitive dependencies.

To allow a process to determine the nondeterministic processes upon which it depends, modified dependency tracking methods can be introduced. Two possible approaches, **ND-Direct** dependency tracking and **ND-Transitive** dependency tracking, are introduced here:

A *process state interval* is identified by a pair $<p, \sigma>$, where σ is a state interval index of process p.

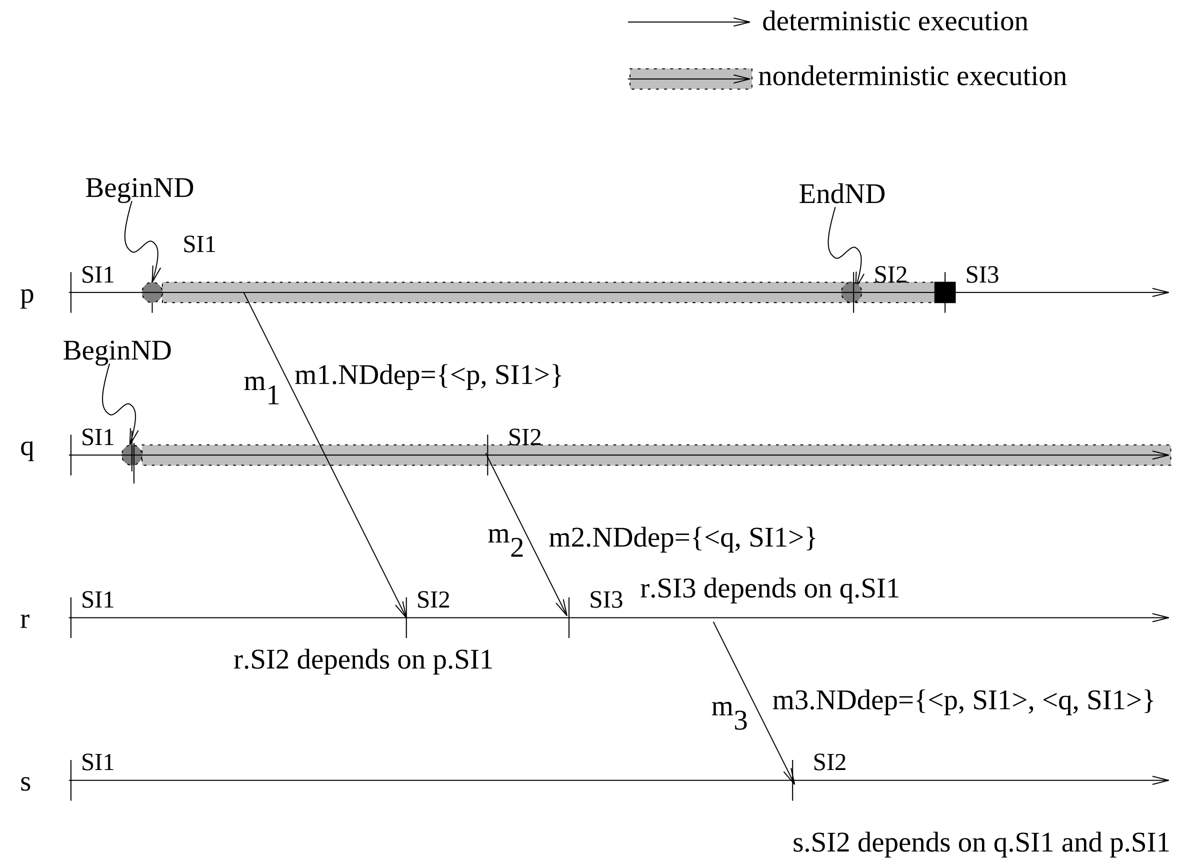

Fig. 3. All nondeterministic dependencies in this execution are labeled.

ND-Direct State interval σ is said to be ND-directly dependent on α if σ is either directly or transitively dependent on a nondeterministic state interval α, but none of the intermediate process state intervals in a chain of transitive dependencies are nondeterministic. This definition conceptually captures "direct dependencies on nondeterministic processes", where a causal chain of deterministic dependencies leading back to a nondeterministic state interval is defined as an direct dependency on a nondeterministic state interval.

ND-direct dependencies can be maintained by a vector at each process and update rules for the vector that add a set of processes identifiers to each message (in contrast, ordinary direct dependencies which require that only a single process identifier be added to each message).

ND-Transitive ND-transitive dependencies correspond to ordinary transitive dependencies and can be maintained by update rules that add a vector of length N to each message.

Specifically, state interval σ is said to be ND-transitively dependent on α if σ is either directly or transitively dependent on a nondeterministic state interval α; any or all of the intermediate process state intervals in a chain of transitive dependencies may be nondeterministic.

A process can simultaneously track both its deterministic and its nondeterministic dependencies by either maintaining two separate dependency vectors (one for deterministic dependencies, the other for nondeterministic dependencies) or by maintaining a single unified dependency vector *DV* that has a boolean value *DV[j].ND* associated with each vector entry, such that *DV[j].ND* is true if and only if state interval *DV[j].SI* is nondeterministic.

III.D.1 Data Structure

Each process p maintains a vector *NDDV* such that $NDDV[j]$ is the maximum nondeterministic state interval index of process j upon which p is currently dependent, or $\perp$ if p does not depend on a nondeterministic state interval of j. In the case of ND-direct dependencies, the word *depend* here means "ND-directly dependent," and in the case of ND-transitive dependencies, *depend* means "ND-transitively dependent."

The *NDDV* could alternatively be represented with a ***stateful dependency vector*** method, where a state variable is associated with each entry of the vector. Each vector entry also contains a state interval index, as in ordinary dependency vectors. For example, to represent *NDDV* with stateful vectors, the state variable would be a boolean variable such that a true value indicates that the dependency is nondeterministic and a false value indicates that the dependency is on a deterministic state interval. For the dependency vector update rules, two vectors received from the same process can be compared to determine which vector reflects a more recent state by simply comparing the two vectors' state interval indexes.

III.D.2 Maintaining ND-Direct Nondeterministic Dependencies

This section describes a method by which each process's ND-direct dependencies can be determined and recorded in the NDDV vector. A set *NDdep* of process state interval identifiers upon which the sender process is currently ND-dependent is added to each application message for use in maintaining the *NDDV* vector at each process.

This set of processes is necessary for maintaining *NDDV*, and the ND-direct dependencies provide sufficient information for a process p to determine its ND-transitive dependencies (that can be done by communication with the processes upon which p is ND-directly dependent, since those processes know all the processes upon which they are ND-directly dependent, and can inform the original process of all such dependencies).

In the worst case, all other processes in the system could be members of this set, in which case the set degenerates to a dependency vector. In the average case, however, the size of the set will be roughly proportional to the number of non-stable nondeterministic processes from which the sender process has delivered messages.

Figure 4 shows the *NDdep* set that must be added to each message to track ND-direct dependencies in a sample execution. In the figure, process p is informed of its ND-direct dependency on u when it receives a message directly from u. Process r is likewise informed of its ND-direct dependency on q when it receives a message from q. Processes r and s do not become aware of their dependency on

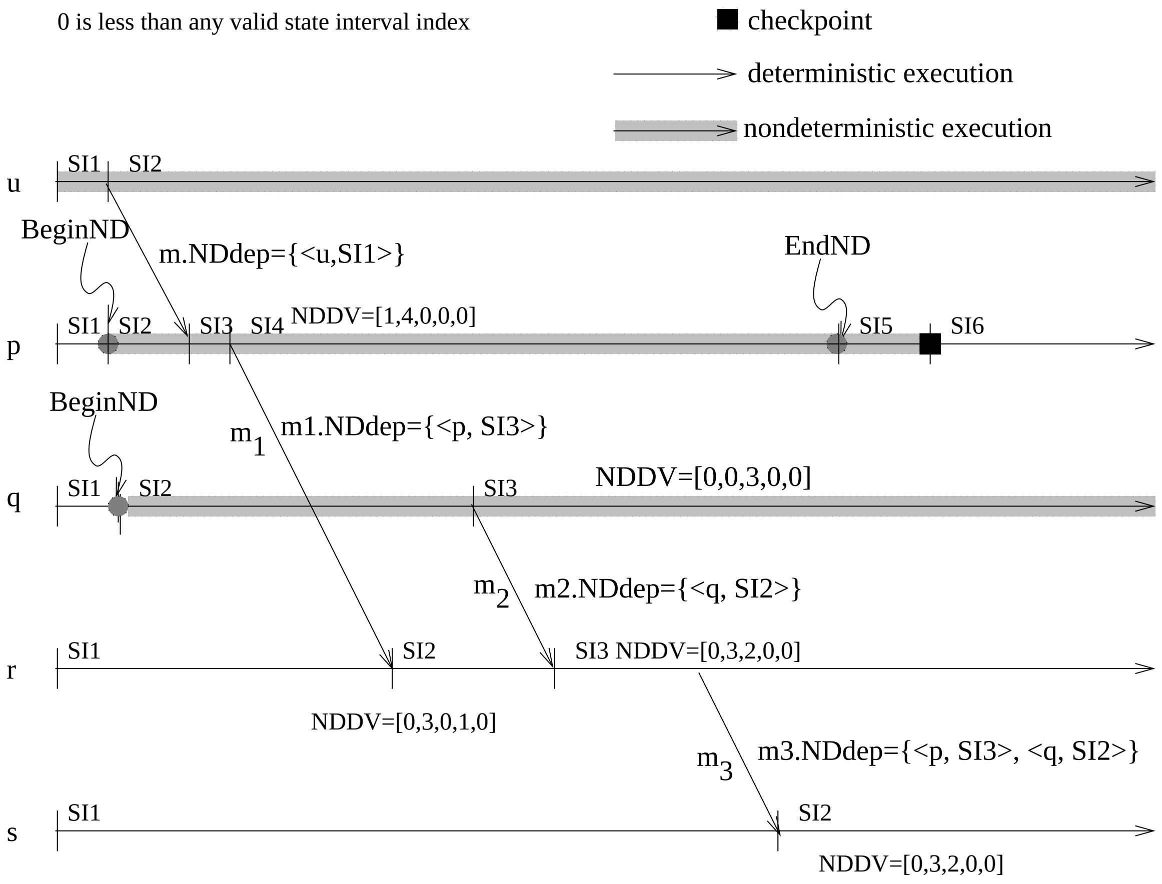

Fig. 4. Maintaining ND-direct dependencies with the *NDdep* set.

u.SI1 in this execution, because dependency chains containing multiple nondeterministic processes not tracked explicitly. That is an ND-transitive dependency, and it can be determined by a commit algorithm that uses inter-process communication to follow the chain of ND-direct dependencies.

The procedure **ND-Transitive-Send-Msg** shown in Figure III.D.1 implements ND-transitive dependencies by adding a process state interval identifier to each outgoing message for each valid entry in *NDDV*. This procedure must be called before *m* is sent to the destination process.

ND-Direct-Send-Msg, shown in Figure III.D.2, implements ND-direct dependencies by adding a process state interval identifier to each message for each valid entry in *NDDV only if* the calling process is deterministic. Hence, when ND-transitive dependencies exist, **ND-Direct-Send-Msg** adds less information to messages sent by a nondeterministic process than does *ND-Transitive-*

```
ND-Transitive-Send-Msg($m$)
if $local\ process\ is\ nondeterministic$ then
    $m.NDdep \leftarrow <LocalProcessId, LocalProcessSI>$
else
    $m.NDdep \leftarrow \emptyset$

for each $pid$ such that $NDDV[pid] > \perp$ do
    $m.NDdep \leftarrow m.NDdep \cup <pid, NDDV[pid]>$
```

Figure III.D.1: **ND-Transitive-Send-Msg** adds the set *m.NDdep* to each message sent by the application.

Send-Msg.

```
ND-Direct-Send-Msg($m$)
if $local\ process\ is\ nondeterministic$ then
    $m.NDdep \leftarrow <LocalProcessId, LocalProcessSI>$
else
    $m.NDdep \leftarrow \emptyset$
    for each $pid$ such that $NDDV[pid] > \perp$ do
        $m.NDdep \leftarrow m.NDdep \cup <pid, NDDV[pid]>$
```

Figure III.D.2: **ND-Direct-Send-Msg** adds the set *m.NDdep* to each message sent by the application.

The **ND-Receive-Msg** procedure shown in Figure III.D.3 updates the calling process's *NDDV* vector to include the set of process state intervals that were added to the message when it was sent. The same **ND-Receive-Msg** can be used to maintain either ND-direct or ND-transitive dependencies; the type of dependency is determined entirely by the sender.

```
ND-Receive-Msg($m$)
for each $e \in m.NDdep$ do
    $NDDV[e.pid] \leftarrow \max(NDDV[e.pid], e.SI)$
```

Figure III.D.3: **ND-Receive-Msg** updates *NDDV*, the vector that records ND-direct dependencies.

III.E Disabling Message Logging During Nondeterministic Execution

A process that is currently nondeterministic need not log the order in which it receives messages, because nondeterministic execution is never replayed. In family-based logging that means that a determinant need not be created for a message received by a nondeterministic process. The send log entry for a message sent to a nondeterministic process can also be purged, but the sender will generally not know that the receiver is nondeterministic until after the message has been received.

A process that is currently deterministic must log the order information of every message it receives, even if the message was sent from a nondeterministic process, because the nondeterministic process can replay the messages it sent deterministically. Nondeterministic processes can participate in the recovery protocol because the recovery system is assumed to always be deterministic.

III.F Output Commit Based On Nondeterministic Dependencies

An output commit algorithm that uses nondeterministic dependencies avoids communication with deterministic processes, which are made recoverable by the message logging protocol. The remainder of this section describes the differences between a version of the Commit algorithm that uses ND-direct dependencies and a version that uses ND-transitive dependencies.

III.F.1 Commit Algorithm Behavior When Only ND-Direct Dependencies are Maintained

Johnson's **Commit** algorithm [15] can be used with no modifications to the algorithm itself. The only change is in way dependencies are tracked (which are used to maintain the DV vector in Johnson's algorithm).

III.F.2 Commit Algorithm Behavior When Only ND-Transitive Dependencies are Maintained

In this case, Johnson's algorithm would have access to full transitive dependencies. The algorithm will still function correctly: it starts with more accurate (i.e. full transitive) dependency information, and does not need to collect that information in multiple rounds [15]. The transitive dependencies decrease output commit latency, but the number of messages is not decreased compared to direct dependencies

III.F.3 Correctness of the Commit Algorithm

An output commit algorithm is correct if it ensures that the given state interval is committable, which is true if the conditions given in Theorem 2 are satisfied. The correctness of the entire Commit algorithm itself need not be shown, since Johnson's **Commit** algorithm is used with no changes except for the modification of the definition of dependency vectors.

It suffices to show that a process can be made committable even when only nondeterministic dependencies are available, and that the nondeterministic dependencies are in fact maintained according to their definition. Considering the latter point first, the correctness of the ND-transitive dependency tracking protocol is established by Theorem 1. A proof of the ND-direct protocol would be similar and is not given here.

Theorem 1 *For a process i using the ND-transitive dependency tracking protocol given by the procedures* **ND-Transitive-Send-Msg** *(Figure III.D.1) and* **ND-Receive-Msg** *(Figure III.D.3), the value of NDDV[j] at i is the maximum state interval index of process j upon which i is currently ND-transitively dependent.*

Proof: *The first* `if` *statement in the* **ND-Transitive-Send-Msg** *procedure ensures that for every message m that a process p sends, the value of $m.NDdep$ is equal to $<p, \sigma>$, where σ is the state interval index from which the message was sent, if and only if σ is a nondeterministic state interval. The* `for` *statement in the procedure* **ND-Receive-Msg** *ensures that for every message m received by a process q, q updates its dependency vector entry NDDV[p] for every process $p \in m.NDdep$ to be the maximum of the entry's current value and state interval index value associated with p in $m.NDdep$ Hence q's NDDV[p] is updated with the most recent nondeterministic state interval index of process p for every $p \in m.NDdep$.*

The receiving process q can become ND-transitively dependent on a state interval of a remote process r in two ways:

Case 1: Directly: the sender of m is itself nondeterministic ($q = r$). In this case, the first `if` *state-*

ment of **ND-Transitive-Send-Msg** *added the sender process and state interval* $<r, \sigma_r>$ *to* $m.NDdep$.

Case 2: Indirectly, through a chain of deterministic dependencies: In this case, the first process in the chain that received a message from r must have correctly set NDDV[r] (by Case 1). When that first process sent a message m' *to the second process in the chain, the* `for` *loop in* **ND-Transitive-Send-Msg** *must have included* $<r, \sigma_r>$ *in* $m'.NDdep$. *Similarly, r must have been included* $<r, \sigma_r>$ *in the NDdep of every message in the causal chain from r to q, and so q must receive* $<r, \sigma_r>$ *in the last message in the chain.*

In both cases, NDDV[r] is set to σ_r *by* **ND-Receive-Msg**. *Because that is done for every message received by every process, the value of NDDV[j] at process i is the maximum state interval index of process j upon which process i is currently ND-transitively dependent.*

Now it remains to be shown that a process can be made committable by using the nondeterministic dependencies. Intuitively, that claim is justified because only nondeterministic processes need be checkpointed by the commit algorithm, and the ND-transitive dependencies have been shown to correctly record the set of nondeterministic processes upon which each process is dependent. Given a correct dependency vector, the **Commit** algorithm will operate correctly. A slightly more rigorous justification of the claim is provided below for completeness.

Theorem 2 states that the nondeterministic dependency vector NDDV, along with the identities of all nondeterministic state intervals and the state interval indexes of all checkpoints, is sufficient information to commit a state interval. [2] The identities of nondeterministic state intervals are assumed to be correctly maintained (a data structure can be maintained monitoring of all **BeginND** and **EndND** events), and recording the state interval indexes of checkpointed intervals is trivial.

Theorem 2 is similar to a theorem introduced by Johnson [15]. The difference between the two stems from the different environment assumed here. In particular, here a deterministic state interval

[2]Recall that nondeterministic (but not deterministic) state intervals must be made stable in the environment assumed in this chapter.

is *always* stable, but a nondeterministic state interval is only stable if it happened before a checkpoint of its process.

Theorem 2 *A state interval σ of some process i is committable if and only if*

(a) *some later state interval $\alpha > \sigma$ of process i is committable, or*

(b) *for all $j \neq i$, state interval NDDV[j] of process j is committable, and*

(σ of i is nondeterministic) $\Rightarrow$ (σ happened before a checkpoint of i).

(NDDV is the vector of nondeterministic dependencies defined in Section III.D.1, and state interval index $\perp$ is defined to be always committable).

Proof Outline: *(The "if" direction is proved, since only it is relevant here)*

(a) *Since some later state interval $\alpha > \sigma$ is committable, α will never be rolled back, and hence an earlier state interval σ will not be rolled back. Thus σ is committable.*

(b) *Since no later state interval is committable, process i could roll back to σ because of a failure. First σ is shown to be stable, then σ is shown to be stable. If σ is deterministic, then the implication is trivially true (in this case recovery can proceed by orphan-free message logging). If σ is nondeterministic, then σ is committable only if it will never be rolled back (by definition), and it will never be rolled back because a checkpoint of state interval $\alpha \geq \sigma$ is known to exist. In either case, process i can be restored to σ. When process i is restored to σ, σ will be part of a consistent global state only if every process state interval that σ depends on will never be rolled back. That condition is satisfied because state interval NDDV[j] of process j is known to be committable, for every $j \neq i$.*

III.G Coordinated Checkpointing

Only a subset of the processes in the system will ordinarily be nondeterministic, but in general it is impossible to determine the identities of all nondeterministic processes without communicating with *all* processes in the system, both deterministic and nondeterministic. However, since deterministic processes need not take checkpoints, [3] they can immediately discard any "take checkpoint" messages (e.g. marker messages) they receive. Thus it is possible to take a coordinated checkpoint that consists only of checkpoints of nondeterministic processes, but communication with all processes is necessary to find all the nondeterministic processes.

III.H Recovery

If a failure occurs, a recovery algorithm must be executed to restore the system to a consistent state. All processes are assumed to participate in an orphan-free message logging protocol during deterministic execution. Checkpoints are not assumed to exist, although the available checkpoints are used in recovery. A checkpoint of a nondeterministic process creates a new state interval σ_c, and all state intervals $\alpha \leq \sigma_c$ of the process are defined to be *stable* if the checkpoint completes successfully (a more precise definition of a stable state interval in this environment is given below in Equation III.1).

A single failed deterministic process is recoverable by the message logging recovery protocol. A single failed nondeterministic process must be rolled back to its most recent *stable* state interval; the rollback of the failed process may create orphan processes that must also be rolled back.

Figure 5 shows an example execution in which the failure of a nondeterministic process p causes the two processes that are causally dependent on p's nonstable nondeterministic execution to become orphan processes because the messages sent to q and r were sent during nondeterministic execution after p's most recent checkpoint. A failed nondeterministic process must roll back to its most recent deterministic state interval, which is the most recent state interval of the process that is either checkpointed or that precedes a **BeginND** event. In this example, p rolls back to the beginning of state interval 2, which is stable because it is checkpointed by checkpoint C1. Both q and r must be rolled back to state intervals that do not depend on p's nonstable nondeterministic execution (the precise definition of a stable state interval in the presence of nondeterminism is given in Section

[3]Recall that some form of orphan-free (e.g. pessimistic of family-based) message logging is assumed to be in operation, so that the most recent state of any deterministic process is committable if all the nondeterministic state intervals it depends on are committable.

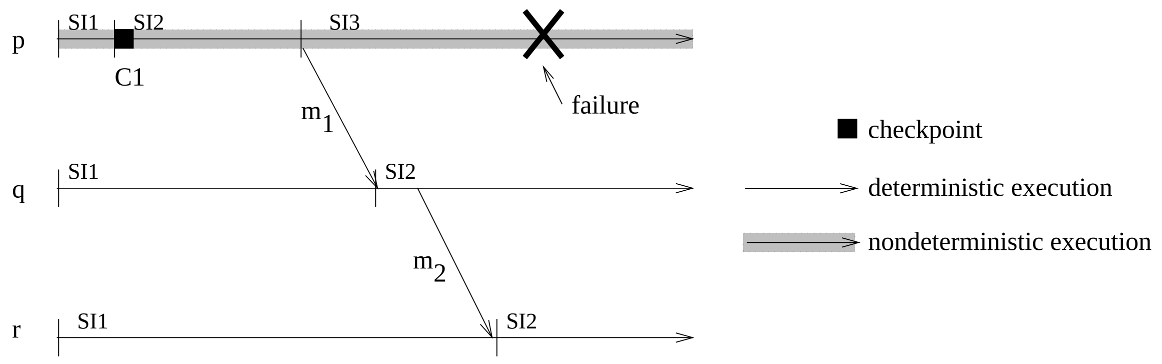

Fig. 5. An example failure scenario.

III.H.1). In this example, q and r must roll back to their initial state intervals. If multiple overlapping failures occur, rollback distance may be greater; some failed processes may be rolled back beyond their most recent stable state intervals. A rollback method and the characteristics of the restored system state are described below.

III.H.1 Definitions

The notion of a *stable* state interval is now defined to include nondeterministic processes: A deterministic state interval is always stable in the presence of orphan-free message logging, whereas a nondeterministic state interval is stable if and only if it happened before a checkpoint of its process. This definition of a stable state interval in the presence of nondeterminism is captured by the predicate $stable(i, \sigma)$:

$$stable(i,\sigma) \Leftrightarrow (not(ND(i,\sigma))) \vee (\sigma \leq MAXCKPTSI(i)) \qquad \text{(III.1)}$$

where *MAXCKPTSI(i)* is the index of the most recent checkpointed state interval of process i and $ND(i, \sigma)$ is true if and only if state interval σ of process i is nondeterministic:

$$ND(i,\sigma) \Leftrightarrow (\perp < BeginNDSI(i) \leq \sigma) \wedge (EndNDSI(i) < BeginNDSI(i)) \qquad \text{(III.2)}$$

BeginNDSI(i) and *EndNDSI(i)* are the state interval indexes of the most recent **BeginND** and **EndND** events in process i, respectively. If there has been no **BeginND** or no **EndND** event in process i, then the respective value of *BeginNDSI(i)* or *EndNDSI(i)* is $\perp$.

III.H.2 Characterization of the Recoverable System State

The most recent recoverable state is the most recent consistent system state in which no process depends on a nonstable state interval. Since only nondeterministic execution can create a nonstable state interval, the most recent recoverable system state can be characterized more specifically as the most recent consistent system state in which no process depends on a nondeterministic state interval that did not happen before a checkpoint of its process.

Using the definition of a stable state interval given in Equation III.1 with Johnson's system history lattice method [14], it can be shown that:

1. A unique most recent recoverable system state always exists.

2. A recovery algorithm such as Johnson's batch state recovery algorithm [12] or *FINDREC* algorithm [12, 14] will find the most recent recoverable system state. Also note that these recovery algorithms can be modified to restrict their search to nondeterministic dependencies in the same way that Johnson's Commit algorithm is restricted to communicate with only nondeterministic processes in Section III.F.

3. The recoverable system state always eventually advances with time, and hence the recovery algorithm is domino-effect-free.

III.H.3 An Alternate Recovery Algorithm

This section presents an on-line distributed algorithm for recovery of a process that was nondeterministic when it failed. Unlike Johnson's recovery algorithms, this algorithm has not been shown to always recover the maximum recoverable state. This algorithm is similar to many well known algorithms, such as that of Koo and Toueg [17].

The recovery algorithm is as follows:

- When a process p begins to recover, it determines its most recent stable state interval σ and broadcasts the message "p rolling back to σ" (if, in fact, rollback is necessary, since it may not be necessary if the process is not dependent on any nondeterministic state intervals).

- When a process q receives the message "p rolling back to σ", q consults its *NDDV* to determine if it has been orphaned by p. Specifically, if σ is less than or equal to q's *NDDV*[p], then q rolls back to its most recent state interval α such that α is not dependent on *NDDV*[p], and then q broadcasts the message "q rolling back to α".

All of the broadcast operations can be replaced with point-to-point communication by changing the algorithm, so that instead of the broadcast, the log of sent messages (which is maintained by FBL) is used to determine the set of processes that are causally dependent on the local process.

III.H.4 Transparently Detecting Nondeterminism

This section suggests an approach that, in the absence of output messages, eliminates need for **BeginND** and **EndND** . During failure-free execution, the delivery order information (i.e. determinant) and data contents of each message must be saved; then, if a failure occurs, re-execution will produce a sequence of messages that can be compared to those saved during failure execution. If the messages generated during re-execution correspond exactly to those previously saved, then nondeterminism has not affected execution. Otherwise, if there is even a single difference in the message data or ordering information, then a process must have executed nondeterministically during the original failure-free run, and hence cannot be re-executed through the state interval from which the first differing message was sent; a previous checkpoint or deterministic state interval must be found. The recovery algorithm can continue to roll back processes for which current checkpoints are not available until there are no differences between the messages generated during re-execution and those that were saved during failure-free execution. The performance of such a recovery algorithm may be poor, however. The algorithm may not find the most recent recoverable state, and the number of rollbacks that occur in the search for a system state in which all state intervals are deterministic may be excessive.

III.H.5 Summary

The method suggested in this chapter allows intermittently nondeterministic processes to use message logging during deterministic execution and minimizes communication during when committing a nondeterministic process state interval. The choice of when nondeterministic processes should checkpoint (outside of a commit operation) is left to the application, and the problem of determining such an optimal checkpointing interval in general is an area for future work.

CHAPTER IV

Reactive Replication In Message Logging Protocols

IV.A The Problem

This chapter describes an approach that allows a message logging protocol that to tolerate multiple overlapping failures with no more failure-free overhead than would be required to tolerate only a single overlapping failure, with the restriction that every two failures must be separated by some minimal interval of time that is sufficient for the non-failed process to prepare for another failure by replicating data crucial to the recovery of the failed process.

During failure-free operation, this reactive approach uses a low-overhead protocol that can tolerate only a single simultaneous failure (e.g. family-based logging [1] or sender-based logging). If a failure occurs, the logged data necessary to recover the failed process is immediately copied (i.e. replicated) to stable storage or to the volatile storage of another processor. Hence the requirement that no two overlapping failures occur is relaxed to a requirement that a second failure does not occur until the necessary logged data has been copied. Reactive replication is only beneficial if certain timing constraints are satisfied during recovery, which means in practice that timeouts have to be well-chosen and message delay has to be small. Since the need for fast failure detection imposes a limit on communication delay, this reactive approach is not practical for systems in which message delay is large.

The primary benefit of reactive replication is small failure-free overhead. Its drawbacks are the bounds its imposes on communication delay, and, if a failure occurs, the overhead of replicating logged data.

[1] FBL's overhead (in terms of the amount of piggybacked information) is lower when $f = 1$ than when $f > 1$.

IV.B Timing Assumptions

The reactive replication technique consists of a *failure detection* phase followed by a *replication* phase. When a failure is detected, the replication phase is initiated. [2]

In family-based logging, the information necessary to recover the failed process is likely to be distributed across the volatile memories of several processes at which it is *sympathetically* logged (in the terminology of Alvisi [2]) on behalf of the failed process. If any of that information is lost, recovery will be impossible. Hence, the replication phase creates a copy of that information in such a way that another failure of a given type (e.g. a single simultaneous process crash, or f simultaneous crashes) can be tolerated without losing it. Processes that have volatile information logged locally on behalf of the failed process must not fail before the completion of the replication phase.

The reactive approach can enhance performance only when recovery of a crashed process takes longer than the sum of the time to detect the crash and the time to replicate the information necessary to recover the crashed process.

For a given failure-free process i and failed process p in a given execution, the real-time duration between the failure of p and the notification of i that the failure has occurred is t^i_{detect}, the real-time for which the replication phase executes is $t^i_{replicate}$, and the sum of those two times is t^i_{react}.

$$t^i_{react} = t^i_{detect} + t^i_{replicate} \tag{IV.1}$$

From the global perspective, the total real-time required to execute the reaction protocol, T_{react}, is the time required to complete its execution at all processors:

$$T_{react} = \max_i(t^i_{react}) \tag{IV.2}$$

Reactive replication is feasible when the reaction protocol's execution time is less than both (a) $T_{recover}$, the time to recover the failed process (recovery is carried out concurrently with the reaction protocol) and (b) $T_{CommitAll}$, the time that would be required to commit the entire system by a

[2]Multiple concurrent initiators are not considered here.

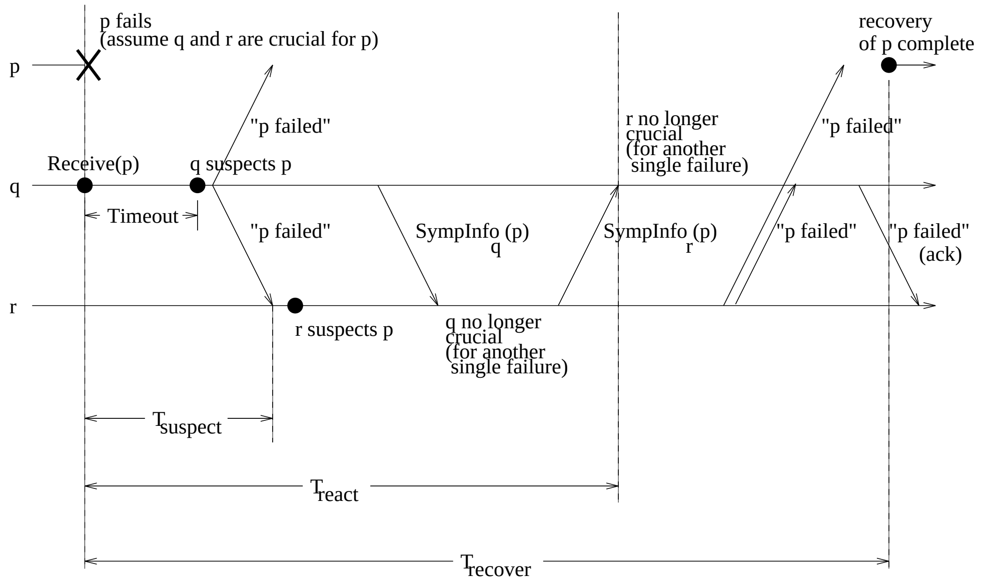

Fig. 6. A sample execution showing reactive replication's response to a failure of process p.

conventional technique (such as a coordinated checkpoint). Figure 6 illustrates these quantities for an example execution. The fail-stop model is in use in that execution, so the Sim-FS protocol is operating. The detection phase is complete when all non-failed processes have been informed of the failure, and the replication phase is complete when SympInfo messages have been received from all processes that are crucial to the recovery of p. Finally, the Sim-FS protocol continues to execute upon the completion of the replication protocol.

IV.C The Relationship between Failure Detection and Reactive Replication

Replication occurs when a failure is detected. This section addresses the problem of failure detection in an asynchronous system. Background on the fail-stop model and a method due to Sabel and Marzullo [23] for implementing it in asynchronous systems is given in Appendix A.

The semantics of the fail-stop model require that every process in the system eventually learns of a failure. Thus a protocol that implements simulated fail-stop is likely to send messages in a pattern that is directly usable by a reactive failure detection protocol. In particular, a "simulated fail stop" (Sim-FS) protocol suggested by Sabel and Marzullo [23] requires that a process broadcast a message when a suspected failure is detected. The "inform" phase of reactive failure detection can be implemented with no additional messages compared those required by Sabel and Marzullo's Sim-FS protocol by using that "failure suspected" broadcast message to inform every process that any information logged on behalf of the suspected process should be replicated (i.e. the replication phase should be initiated for the suspected process). The approach is straightforward: when a process p receives a "failure of process q" suspected, p initiates the reactive replication protocol for q.

The primary drawback of such a reactive technique is the need for a failure to be detected very soon after it occurs. Fail-stop failure detection can be implemented in an asynchronous environment [23], but in practice the latency between failure and detection must be small. Since the simulated fail-stop failure detection protocol is based on communication timeout, the reactive method described here is only practical in asynchronous distributed systems when sufficiently small timeout values are used to implement failure detection. Timeout values must be chosen so that failures are detected quickly, but false failures are not reported too often. The problem of choosing timeout values is not addressed here.

IV.D Definitions

With respect to a crashed process p, a *sympathetic process* is a process that has sympathetically logged information on behalf of p.

$\mathit{SympInfo}_q(p)$ denotes the information sympathetically logged in process q's volatile storage on behalf of process p. A process is either *crucial* or *non-crucial*:

A *crucial process* q is a sympathetic process for which at least one piece of the logged information $\mathit{SympInfo}_q(p)$ is not logged at any other process.

A *non-crucial* process q is either a nonsympathetic process or a sympathetic process for which every piece of logged information $\mathit{SympInfo}_q(p)$ is logged at at least one other process.

In the time interval between the crash and recovery of a process p, the message logging protocol (as described here) cannot tolerate the failure of any processes that are crucial to p. In terms of these definitions, reactive replication attempts to make all crucial processes non-crucially sympathetic before one of the crucial processes can fail.

Detection of the failure of a crucial process is straightforward in a fail-stop environment [3] The recovery protocol will learn of any crash that occurs during recovery, and will be able to determine whether such a crashed process is crucial to the recovery in progress.

IV.E Tolerating Violation of the Timing Assumptions

A simple way to deal with the failure of a crucial process is to employ a second recovery protocol that has greater overhead but can tolerate a greater number of simultaneous failures than the message logging protocol. Clearly, the second protocol should be executed relatively infrequently so that a good tradeoff between failure-free overhead and rollback distance is obtained [31]. For example, global consistent checkpoints could be taken periodically (but rarely), and in the event of a failure that message logging cannot tolerate (i.e. failure of a crucial process), the system would be rolled back to the most recent global checkpoint. Another possible approach would be to periodically flush all processes' volatile message logs to stable storage in a consistent manner.

IV.F Abstract Reactive Replication Protocol

This section summarizes the two phases of reactive replication independently of any particular message logging protocol.

IV.F.1 Failure Detection Phase

If a failure occurs (i.e. a process crashes), the failure must be detected, and all processes must learn of the failure In general, a failure cannot be detected in an asynchronous environment. However, messages can be periodically sent to a process p to determine if it is crashed, and if p does not reply within a reasonable amount of time, then it can be assumed to have crashed, in which case p can be removed from the system and replaced by a new process restored to p's saved state. False crash detections will cause unnecessary rollbacks.

[3]It is assumed here that the fail-stop model's property that all processes are informed of a failure is desirable.

IV.F.2 Replication Phase

As described above, $\mathit{SympInfo}_i(j)$ denotes the information contained in the volatile storage of process i that is necessary to recover process j from a failure.

Upon detection of a crash (e.g. upon receipt of a "crash notification" message), each process i for which $\mathit{SympInfo}_i(j) \neq \emptyset$ must either

- Immediately log $\mathit{SympInfo}_i(j)$ to stable storage

  or

- Immediately log $\mathit{SympInfo}_i(j))$ to the volatile storage of another non-crashed processor. The method for selecting another processor is not addressed here.

  Each process p such that the original crashed process's volatile log contained some information necessary to recover p is subject to an additional limitation: if one of them crashes (after the original failure has been detected), it must wait until the original crashed process recovers before it can recover.

IV.F.3 The Contents of $\mathit{SympInfo}_i$ for Family-Based Logging

In family-based logging [1, 2], the information necessary to recover a process p is:

1. the data contents of every message that has been sent to p
2. the determinant $\#m$ of every message m that p has delivered [4]

When a failure is detected, the above information must be replicated immediately (either to stable storage or the volatile storage of another non-failed processor). Hence $\mathit{SympInfo}_i(j)$ for family-based logging consists of: (a) the data contents of all messages that process i has sent to process j and (b)

[4]A different approach would be for each process q to replicate all information in its logs (msg data and determinants). That would allow a future failure of q to be tolerated; it would have to be done for every process (except the crashed one). Hence all the logged information in the system would be replicated. The technique described in the text is supposed to replicate only the information necessary to tolerate a subsequent failure of a process that has information necessary for the recovery of the originally-failed process.

all determinants e for which $e.dest = j$. Specified as a set,

$$SympInfo_i(j) = \{\{m | m.dest = j\}, \{\#m' | m'.rsn \leq \sigma^j_{crash}\}\} \tag{IV.3}$$

where m is a message (including the message data $m.data$) and σ^j_{crash} is the state interval in which the failure occurred. This specification of *SympInfo* includes the information sufficient to completely determine the deterministic process j's execution up until the crash. This set contains values of two different types, determinants and messages, so that all the information can be contained in a single set (when a particular element of the set is referred to, the type (determinant or message) will be clear from the context) . This information is necessary for the operation of the family-based logging recovery algorithm [2].

IV.G A Simple One-Round Sim-FS Protocol for the Detection Phase

This section describes a protocol that can be used for both general failure detection and as the failure detection phase of reactive replication. The following one-round protocol from Sabel and Marzullo [23] implements a version of asynchronous simulated fail stop (properties **sFS2a-d**, as described in Appendix A) that is indistinguishable from fail-stop. It is assumed that a failure suspection mechanism exists (e.g. timeout), and that no more than t failures are suspected in any run. Here, the suspection message (called $SUSP_{i,j}$ in Appendix A) and the acknowledgment message ($ACK.SUSP_{i,j}$) are both of the form "j failed".

- When process i first suspects the failure of process j, i sends the message "j failed" to all processes (including itself). Process i waits for messages of the form "j failed" from other processes and takes no other action except for acknowledging "x failed" messages until it completes the protocol or crashes.
- When process i has received messages of the form "j failed" from more than $\frac{n(t-1)}{t}$ processes (including itself), i executes $failed_i(j)$ (that is, i decides that j has failed).
- When process x receives a message of the form "x failed", x executes $crash_x$ (that is, x decides to crash itself).

- When process x receives a message of the form "y failed" and x does not suspect the failure of y, x suspects the failure of y and executes the first step of the protocol.

Since this protocol uses $\frac{n(t-1)}{t}$ for its quorum set size, it requires that $n \geq t^2$.

IV.H A Reactive Protocol For Family-Based Logging

A reactive protocol suitable for use with FBL is as follows:

- **Detection Phase:** When process i detects the failure of process j, i broadcasts "j failed" to all other processes to inform them that they should prepare to tolerate further failures.

- **Replication Phase:** When a process k receives a "j failed" message, process k *reacts* to the failure by immediately replicating any volatile information $\mathit{SympInfo}_k(j)$ that it might have to either stable storage or the volatile storage of another process (the latter process should reside on a separate non-failed processor).

  The value of *SympInfo* for family-based logging is described in Section IV.F.3. $\mathit{SympInfo}_k(j)$ can be determined from process k's local volatile logs using the **Determine-SympInfo** procedure shown in Figure IV.H.1. Note that **Determine-SympInfo** places values of two different types (determinants from DetLog and messages from SendLog) in the SympInfo set; there is no relation between any two values at the time they are added. It remains to be shown that every message m (including $m.\ data$) and corresponding determinant $\#m$ that must be in SympInfo at a given process have indeed been added by the time the **Determine-SympInfo** terminates.

  The determinant log *DetLog* contains every determinant logged at the local process, and the send log *SendLog* contains the message data of every message that has been sent by the local process. These logs are maintained by the family-based logging protocol [2].

  Determine-SympInfo must be executed at each process k that has sympathetically logged information on behalf of j. The broadcast-based protocol described here simply executes it at every process in the system. The set *SympInfo(j)* can be logged either incrementally, as it

```
DETERMINE-SYMPINFO(j)
    $SympInfo(j) \leftarrow nil$
   $\triangleright e$ is is a determinant; $e = \#m$ for some message $m$
    for each $e \in DetLog$
        if $e.dest = j$ then
            $SympInfo(j) \leftarrow SympInfo(j) \cup \{e\}$

    for each $m \in SendLog$
        if $m.dest = j$ then
            $SympInfo(j) \leftarrow SympInfo(j) \cup \{m\}$
```

Figure IV.H.1: The **DETERMINE-SYMPINFO** procedure.

is being determined (i.e inside the for loop), or all at once, after it has been determined (i.e. after the for loop).

IV.H.1 Correctness

Reactive replication is carried out correctly (assuming that the timing assumptions are met) if **DETERMINE-SYMPINFO** correctly constructs the $SympInfo_p(j)$ for a failed process j set at every nonfailed process p. The set is correctly constructed because the first `for` loop of **DETERMINE-SYMPINFO** adds all determinants $\#m$ in the local process p's determinant log for which $m.dest = j$ to SympInfo, and the second loop adds all messages m in the local processes send log for which $m.dest = j$. By the definition of DetLog and SendLog, the necessary determinants and messages for the local process's SympInfo set must be available in those logs, and hence the SympInfo set is constructed correctly.

IV.H.2 Efficiency Considerations

If multiple processes share a common processor and share a single per-processor log, the number of "j failed" messages sent by the protocol can be reduced by treating each processor as a "logging site" [5] and not sending the "j failed" message to processes on the same logging site as the sender. That is justified by the fact that processes on the same logging site share the same log, and there is no reason for more than one of them to replicate the logged information. The term "logging site" is

[5]The logging site concept is described in more detail in Section V.C.3.6

due to Alvisi and Marzullo [2].

The broadcast-based protocol suffers from at least two sources of potentially wasteful message complexity: It may duplicate message traffic that the failure detection protocol has already used to decide which process failed, and some processes that receive the "j failed" broadcast message may have no information relevant to j (i.e. $SympInfo_i(j) = \emptyset$, in which case the overhead incurred in sending and delivering the broadcast message to that process is wasted. If the number of processes n is large, the overhead of broadcast will be significant. A benefit of the broadcast-based protocol is its simplicity: each process is only responsible for the relevant information in its own volatile storage, which means that a process does not need to send any messages related to the location of the relevant information. Furthermore, this broadcast's overhead is negligible when it is done as part of the failure detection protocol's broadcast.

As part of its implementation of fail-stop semantics, the Sim-FS failure detection protocol does a broadcast to inform all processes of the identity of a process that is suspected to have failed. That broadcast corresponds directly to the failure detection phase of reactive replication.

Thus the broadcast-based protocol's message overhead can be reduced by merging the failure detection protocol (Sim-FS) with the failure-detection phase of the reaction protocol, so that the messages used by the detection protocol trigger the reaction directly. Specifically, the Sim-FS protocol can be used as the sole means of failure detection in the system, and the "crash notification message" that initiates the replication phase is simply the "j crashed" message sent by Sim-FS.

One version of Sim-FS waits for acknowledgments from every other process before deciding that a process has crashed, whereas another version waits for acknowledgments from only a subset of all processes. Since both versions broadcast the "j failed" message, the difference in waiting for acknowledgments is only important here if the final decision of the detection protocol is used to initiate the replication phase of reactive replication, as opposed to the initial *suspect* event. Since the reactive protocol given here allows the first "j failed" to initiate the replication phase, the difference between the two acknowledgment policies is not addressed further here. This assumption means that *every* failure suspection will result in a corresponding replication. The number of replications due to false detections could be reduced by waiting until the detection protocol terminates before beginning replication, but then the detection time (t_{detect}) would increase from just the time to deliver a single broadcast to the total time to execute the detection protocol. A good choice for the timeout value can reduce the number of both false notifications and replications.

An obvious approach to reducing the overhead of the broadcast-based protocol is to replace the broadcast with point-to-point communication directed by the relevant information. If that could be done, then the message overhead could be reduced to only that necessary to inform the nodes with relevant information of the failure. However, the broadcast is necessary for reactive replication, because the information available at a single arbitrarily-chosen surviving processes is not sufficient to determine the set of all processes that have sympathetic information. The overhead of one broadcast at the time of a failure should be small. However, if the timing assumptions are violated, and the delivery of the broadcast is delayed sufficiently long, another process may fail before the replication completes, in which case some crucial sympathetic information may be lost.

IV.I Summary

Reactive replication may be beneficial for certain specific application areas where the timing constraints are known to be met. If special hardware were available to provide nearly instantaneous failure detection, reactive replication would be quite viable for use in tolerating multiple overlapping failures with no more overhead than is necessary to tolerate a single overlapping failure.

CHAPTER V

Implementation

V.A Introduction

The implementation consists of a library of routines for Unix that provide a message passing abstraction, which is general enough for practical use yet simple enough that recovery protocol implementation is straightforward, as well a the failure-free portion of the recovery protocols described in Section V.A.1.

V.A.1 Recovery Protocols

The following protocols have been implemented:

- A **coordinated checkpointing** protocol similar to the Manetho coordinated checkpointing protocol [7]. The Manetho protocol has been extended to checkpoint a given set of processes to allow coordinated checkpoints that include only the processes on a single logging site. Only the processes on the logging site of the process that initiates a coordinated checkpoint are included in that coordinated checkpoint. One process at each logging site is designated as the coordinator of its site, and sends marker messages to only the processes on that site. A consistent checkpoint of all processes in the system can be taken by executing the coordinated checkpointing protocol where the set of processes to be checkpointed contains only the set of site coordinators. Unix process checkpointing is done with the *libckpt* library [19], which has been extended with minor modifications that allow multiple checkpoints of different instances of a single program to be saved and stored simultaneously.

- The **family-based message logging** protocol π_α suggested by Alvisi and Marzullo [2], which is able to tolerate multiple overlapping failures. In addition, a version of π_α that uses logging sites [2] to maintain a single volatile log for each processor in shared memory (to reduce the amount of piggybacked information) has also been implemented.

The actual recovery portions of the protocols have not been implemented; failure-free performance can be meaningfully measured without them, and they are left as future work.

V.A.2 Overview of the Implementation

The message passing abstraction consists of a set of functions for use in writing distributed programs. The most important functions are **Exec**, which creates a new *child* process given the name of an executable program file and establishes a connection between the *parent* (creator) and child processes, **SendMsg**, *ReceiveMsg*, and *ForwardMsg* for passing messages between connected processes, and **ConnectProcesses**, for establishing a connection between two child processes.

These functions call the recovery protocols in an application-transparent way, so that any distributed program that uses them can automatically be made fault-tolerant without placing any additional programming burden on the person writing the application. The cost of the fault-tolerance is in overhead added to the application's failure-free performance, in terms of running time and storage usage.

V.A.2.1 Functional Layers

Three major layers of abstraction are present in a program that uses the recovery system, as shown in Figure 7. At a low level is Unix itself, upon which the message passing, recovery protocol, and checkpointing libraries are directly built. The message passing library automatically calls the recovery protocol functions in the recovery system. Finally, the user application calls the message passing library, but need not call the recovery protocol or checkpointing library. If the application wishes to provide application-specific information to the recovery system, such information would be provided by direct calls from the application to the recovery system. Such application-specific hints are beyond the scope of this thesis.

V.B Message Passing Library

The message passing library provides an environment suitable for writing distributed applications. Any program can call the message passing functions subject to the requirements given in this

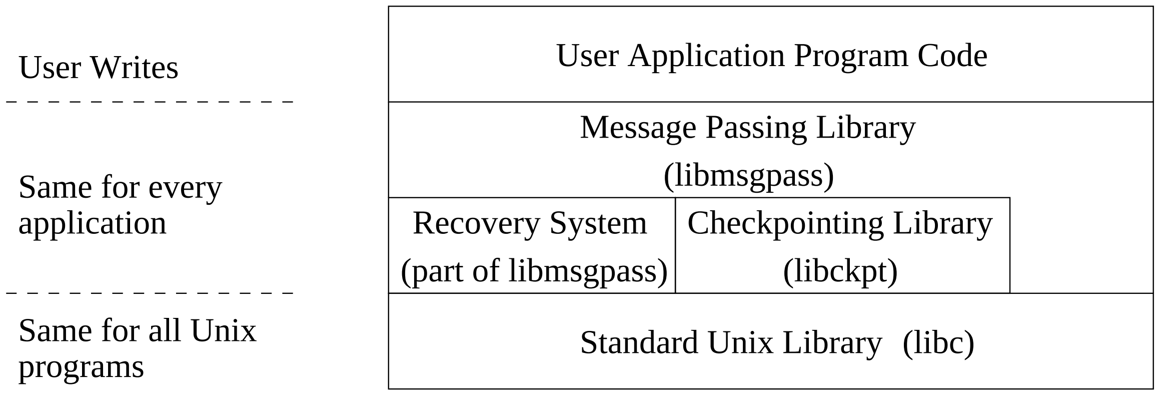
User Writes
Same for every application
Same for all Unix programs
User Application Program Code
Message Passing Library (libmsgpass)
Recovery System (part of libmsgpass)
Checkpointing Library (libckpt)
Standard Unix Library (libc)

Fig. 7. The layers of abstraction in a process that uses the recovery system.

section. The application should have a master-slave structure, where a single master process creates all the slave processes and then explicitly establishes connections between those slaves that are to communicate directly with each other. The functions provided by the message passing library are summarized in Table V.B.1.

V.B.1 Requirements for the Master Process

Any program can be a master process, subject to the following restrictions. The creation of slave processes and establishment of connections between them must be done during the *system initialization phase*. The master must explicitly end this initialization phase before the actual computation begins. These restrictions on the structure of the program and the time at which processes may be created and connected could be removed in the future. These restrictions are present to simplify the implementation of the message passing library and are reasonable for relatively simple programs such as the benchmarks used here. After the initialization phase, the master may execute freely and call **SendMsg**, **ReceiveMsg**, or **ForwardMsg** at any time.

V.B.2 Requirements for the Slave Process

Any program can be a slave process if it calls that calls **MsgPass_SlaveInit** to initialize the message passing library before calling any of the library's other functions. In practice, a slave program should communicate with the master through some common protocol. Since the master creates slaves with **Exec** by giving the pathname of an executable program file, it is possible to use a different program for each slave process. In the benchmarks applications used here, each slave process is an instance of the same slave program, and the master program is different from the slave program.

V.B.3 External Function Interface

The message passing library's external interface consists of the functions listed in Figure V.B.1. Each message is represented as a variable of type *Message*, as described in Section V.B.4.

V.B.4 Structure of a Message

The Message type, which is an argument to SendMsg and is returned by ReceiveMsg, is simply a C structure whose first field (i.e. member) is of type MessageHeader. The MessageHeader type is a structure with the following fields:

srcAddr, dstAddr – the source and destination ProcessId's of the message, respectively. These fields are filled in automatically by the message passing library.

forwardedFrom – if the message was not forwarded, this field has the value NOT_FORWARDED; otherwise, this field contains the ProcessId of the process from which the message was forwarded. The application can ignore this field.

msgType – an integer value that identifies the message's type. The application must explicitly set a value for this field. The application is free to use any value, but note that a message with a type value less than or equal to Max_SystemMsgType is treated as a *system* message, which means that the message is not delivered to the application. That is, ReceiveMsg will never return a system message. Also, some type values have already been allocated for specific purposes; see the file `messages.h`.

dataLen – an integer that gives the number of bytes of the message structure following the header field that are to be sent with the message. The application must explicitly set this field. (the value zero can be used for a message that has no data. The upper bound on dataLen is given

MsgPass_MasterInit() must be called by the master process before the master calls any other message passing library functions.

MsgPass_MasterInitComplete() should be called by the master process after it has finished creating and connecting slave processes.

MsgPass_SlaveInit() must be called by a slave process before the slave calls any other message passing library functions.

ProcessId MsgPass_GetLocalProcessId() returns the local process's ProcessId.

int MsgPass_GetNumProcessesInSystem() returns the total number of processes to which the local process is currently connected, including the master.

SendMsg(ProcessId dest, Message *msg) sends the given message to the process identified by *dest*.

ForwardMsg(ProcessId dest, ProcessId originalSrc, Message *msg) sends the given message to the process identified by *dest*; the message will appear to process *dest* to have been sent from process *originalSrc* instead of from the local (forwarding) process.

Message *ReceiveMsg(ProcessId source, int type) blocks the local process until a message of the given *type* is received from the process identified by *source*, and then returns that message; if such a message was received before the function was called, then the first such previously-received message is returned immediately. If ReceiveMsg is called with *source* = *ANYPROCESS*, then the first message of the given type available from any process is returned. Similarly, if *type* = *ANYTYPE*, then the first message from the given process is returned. If both *ANYPROCESS* and *ANYTYPE* are specified, the first message available is returned.

int Exec(char *argv[], char *hostname) creates a new child process running the program given by *argv[0]* on the processor given by *hostname* and establishes a connection between the child process and the process that called Exec (the parent). Exec assigns a systemwide-unique ProcessId to the child process and returns that value. After Exec returns, the parent can address the child using the returned ProcessId, and the child can address the parent using the special ProcessId value 0. The *argv* argument is an array of strings that begins at *argv[0]* and ends at the first element of the *argv* array that is not equal to NULL. The first n non-NULL array elements become the new process's own argv command line arguments, and the new process's argc is set to n (the number of elements in the process's argv). If *hostname* == "", then the processor allocator is used to select a host as described in Section V.B.7. In the current implementation, Exec should only be called by a master process, and an application should have only one master process.

ConnectProcesses(ProcessId p1, ProcessId p2) establishes a connection between processes *p1* and *p2*. The calling process must already be connected to *p1* and *p2*. In the current implementation, this function should only be called by the master process, and only during the master's initialization phase.

Figure V.B.1: Message passing library interface functions

by the constant MAX_MSG_PAYLOAD_SIZE, which is currently set to 16384 bytes (see the file `config.h`).

SSN, RSN – the send sequence number and receive sequence number, respectively, of the message. These are filled in automatically by the message passing library.

CCN – this field is only present if the coordinated checkpointing recovery protocol has been compiled into the message passing library. See Section V.C.4 for a description of the CCN value.

piggybackLen, piggybackPtr – These fields are only present if the piggybacking code is compiled into the message passing library. Piggybacking is described in Section V.C.3.5.

A sample message structure is shown in Figure V.B.2. The application can define its own messages, but every message structure should begin with a MessageHeader field. There can be any number of additional fields. The implementation of SendMsg and ReceiveMsg cast every message to a fixed-size structure that has an array of bytes immediately following the header field. The size of that array, which is given by MAX_MSG_PAYLOAD_SIZE in `config.h`, determines the maximum message data size.

```
typedef struct ProcessCreatedMsg {
  MessageHeader hdr;
  int newProcessId;
  int newProcessCtrlPort;
  unsigned long newProcessHost;
} ProcessCreatedMsg;
```

Figure V.B.2: An example message structure definition.

V.B.5 An Example Application Program

This section shows an example application that uses the message passing library. In this application a master process starts a number of slave processes and sends a single message to each slave. Each slave receives a single message from the master, sends a message to each remote slave, and finally receives a message from each remote slave. Figure V.B.3 shows the master program, which would be started as an ordinary Unix program (e.g. from a shell prompt), and Figure V.B.4 shows the slave program, which is started by the master.

```
#include <msgpass.h>
main(int argc, char *argv[])
{
   int i, j, NumSlaves;
   Message msg;   /* Message is a generic message type defined in msgpass.h */

   MsgPass_MasterInit(&argc, &argv);
   NumSlaves = 4;

   /* the full pathname should be used by the application */
   argv[0] = ``/user/erniee/mp.solaris/gauss/slave'';
   argv[1] = NULL;
   /* start processes 1, 2, 3, and 4 */
   for (i=0; i < NumSlaves; i++)
      Exec(argv, ``'');

   /* establish a connection between every pair of slave processes */
   for (i=1; i <= NumSlaves; i++)
      for (j=i+1; j <= NumSlaves; j++)
         ConnectProcesses(i, j);

   MsgPass_MasterInitComplete();

   msg.hdr.msgType = 10001;    /* user-defined types should be >= 10000 */
   msg.hdr.dataLen = 0;
   for (i=i; i= < NumSlaves; i++)
      SendMsg(i, msg);
}
```

Figure V.B.3: An example master program

V.B.6 Internal Operation

This section presents high-level details of the message passing library's internal operation.

V.B.6.1 Basic Data Types

The following data types are used by the message passing library:

ProcessId – values of this type identify processes (an integer)

ProcessorId – values of this type identify processors (an integer)

```
#include <msgpass.h>
#define SAMPLE_TYPE 10001
typedef struct SampleMessage {
   MessageHeader hdr;
   int number;
} SampleMessage;

main(int argc, char *argv[])
{
   int i, NumSlaves;   ProcessId LocalPid;   SampleMessage *msgptr, msg;

   MsgPass_SlaveInit(&argc, &argv);
   NumSlaves = GetNumProcessesInSystem() - 1;
   LocalPid = GetLocalProcessId();

   /* receive a message from the master */
   msgptr = (SampleMessage *) ReceiveMsg(0, SAMPLE_TYPE);
   /* ReceiveMsg returns a pointer to heap storage that the caller must free */
   free(msgptr);

   /* initialize the msg variable */
   msg.hdr.type = SAMPLE_TYPE;
   msg.number = 1;
   msg.hdr.dataLen = sizeof(msg.number);

   /* send a message to each remote slave */
   for (i=1; i <= NumSlaves; i++)
      if (i != LocalPid) {
         msg.number = LocalPid * i;
         SendMsg(i, msg);
      }
   /* receive NumSlaves-1 messages (assume one from each remote slave) */
   for (i=1; i < NumSlaves; i++) {
      /* receive order is random here because a specific
       * source address is not requested in the call to ReceiveMsg.
       * Alternatively, the call ReceiveMsg(i, SAMPLE_TYPE)
       * could be used to make receive order deterministic.
       */
      msgptr = (SampleMessage *) ReceiveMsg(ANYPROCESS, SAMPLE_TYPE);
      printf(``received msg from pid %d with number=%d\n'',
               msgptr->hdr.srcAddr, msgptr->number);
      free(msgptr);
   }
}
```

Figure V.B.4: An example slave program

V.B.6.2 Basic Data Structures

The message passing library maintains the following information at each process to support process creation, connection establishment, and sending and receiving messages:

LocalProcessId is the process identifier (a numeric value, of type *ProcessID*) of the local process. Each process has a unique ProcessID by which all other processes identify it. ProcessID's are used as the source and destination addresses for message passing; SendMsg's destination address argument and ReceiveMsg's source address argument are ProcessID's.

LocalProcessorId is the processor identifier (a numeric value, of type *ProcessorID*) of the processor on which the local process is running.

LocalHostName is a string that contains the host name of the processor on which the local process is running. This name is an Internet host name (such as *almond.cs.tamu.edu* in the current implementation.

LocalIPAddr is the Internet Protocol address of the processor on which the local process is running. Any one of the three variables LocalProcessorId, LocalHostName, or LocalIPAddr is sufficient to identify the local processor; the three different representations are present because each is used at some place in the implementation.

The *ConnectionTable* contains an entry for each connection to a remote process. The ConnectionTable is currently implemented as an array indexed by ProcessId values, so *ConnectionTable[j]* contains the connection information for the local process's connection to process j. The value of *ConnectionTable[LocalProcessId]* is undefined. Specifically, The ConnectionTable is an array of structures of type ConnectionTableEntry, each of which contains three elements:

- *sockfd* – the socket to use in communicating with process j
- *serverPort* – the port on which process j accepts new connections (in the local host's byte order)

  - HostAddr – a structure of type sockaddr_in that contains the IP address of the processor on which process j resides. This structure also contains j's port (in network byte order).

The above variables are set during the system initialization phase.

V.B.6.3 Process Creation Protocol

The Exec function passes the following information to each new process via command-line arguments:

- the new process's ProcessId
- the IP address and server port number of the master process
- the ProcessorId of the new process's processor
- the number of processes that already exist on the new process's processor

The slave's server port is the sum of the master's server port and the new process's ProcessId. The Exec function waits for a "ChildRunning" message from the new process before returning. The entire Exec protocol is summarized in Figure 8.

V.B.6.4 Inter-Slave Connection Establishment Protocol

The inter-slave connection establishment protocol executed by the ConnectProcesses function is summarized in Figure 9. Referring to the figure, the final two messages guarantee that the receiving process assigns the correct remote process address to the connection, since the "accept from" and "connect to" messages may be received in a different order than intended by the master at different processes. If the "accept from" message shown here were to be received by $p1$ after $p2$ attempted to connect, the connection would still be established because ConnectSocketToServer will block until AcceptSocketFromClient is called.

An alternative to the ConnectProcesses function would be a ConnectToProcess function that takes a single argument j, where j identifies a remote process to which a connection will be established. This alternative approach would allow a process to establish a connection on its own

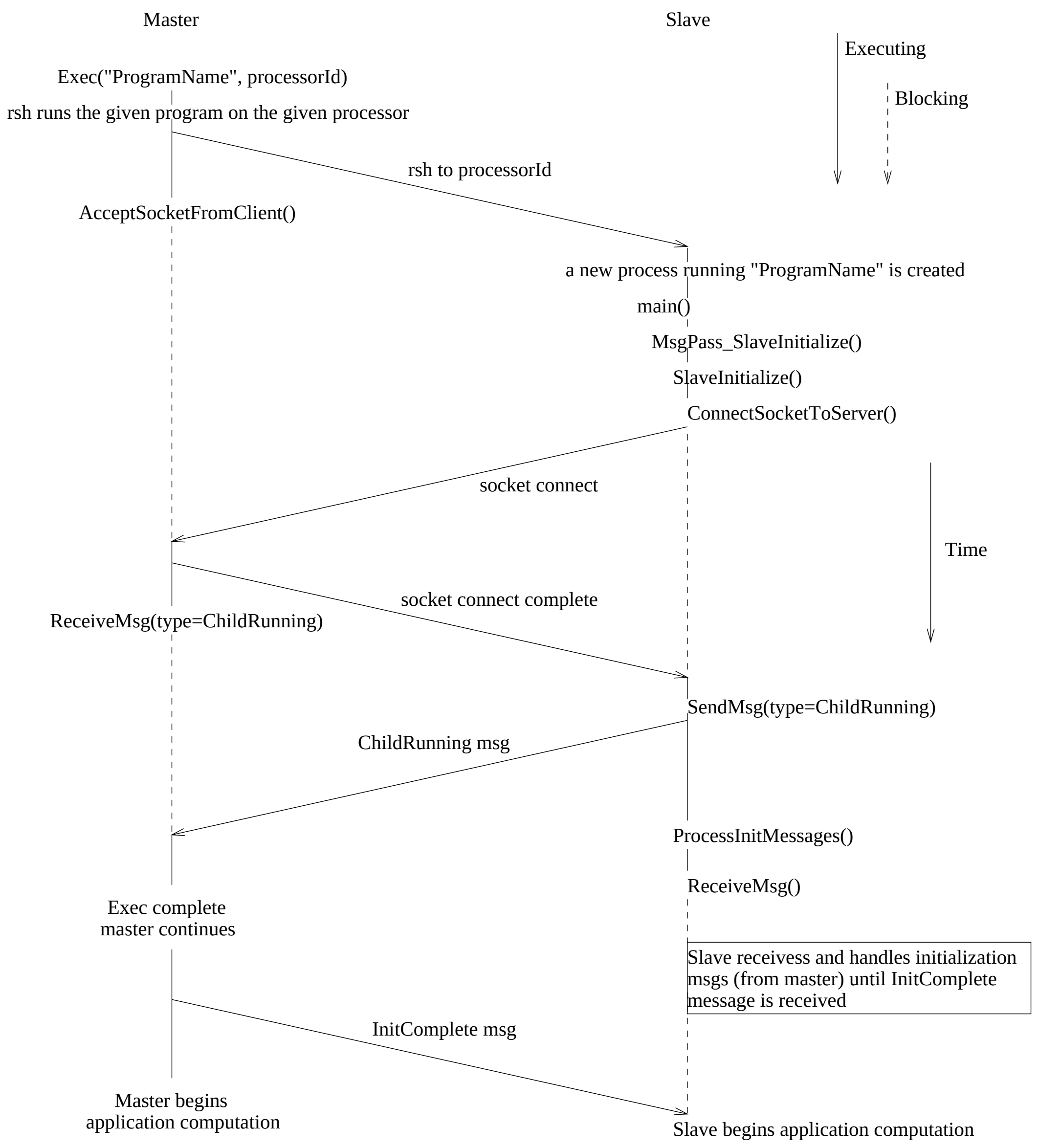
Master
Slave
Executing
Blocking
Exec("ProgramName", processorId)
rsh runs the given program on the given processor
rsh to processorId
AcceptSocketFromClient()
a new process running "ProgramName" is created
main()
MsgPass_SlaveInitialize()
SlaveInitialize()
ConnectSocketToServer()
socket connect
Time
socket connect complete
ReceiveMsg(type=ChildRunning)
SendMsg(type=ChildRunning)
ChildRunning msg
ProcessInitMessages()
ReceiveMsg()
Exec complete
master continues
Slave receivess and handles initialization msgs (from master) until InitComplete message is received
InitComplete msg
Master begins
application computation
Slave begins application computation

Fig. 8. Creating a new process with Exec

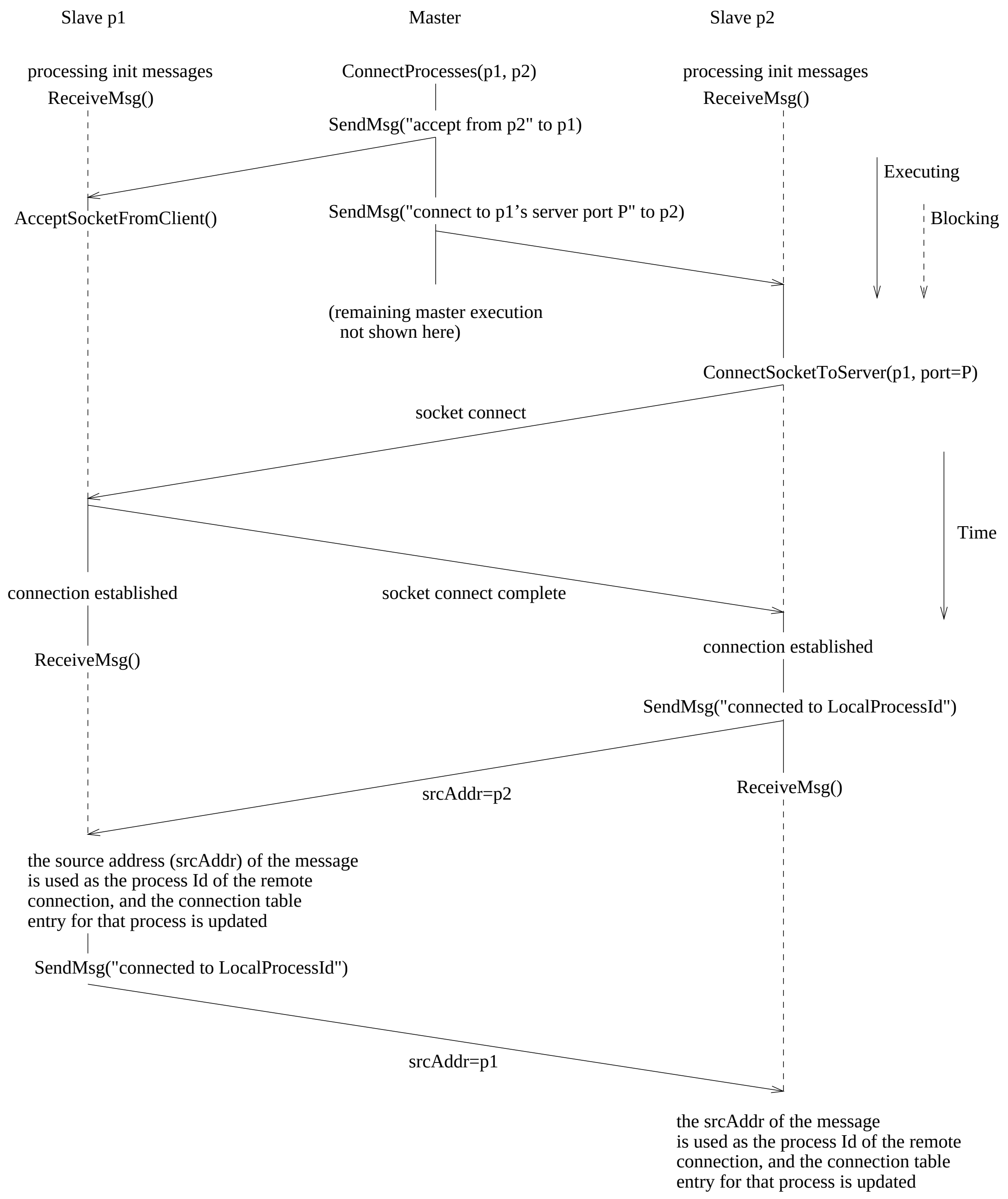
Slave p1
Master
Slave p2
processing init messages
ReceiveMsg()
ConnectProcesses(p1, p2)
processing init messages
ReceiveMsg()
SendMsg("accept from p2" to p1)
Executing
Blocking
AcceptSocketFromClient()
SendMsg("connect to p1's server port P" to p2)
(remaining master execution not shown here)
ConnectSocketToServer(p1, port=P)
socket connect
Time
connection established
socket connect complete
ReceiveMsg()
connection established
SendMsg("connected to LocalProcessId")
ReceiveMsg()
srcAddr=p2
the source address (srcAddr) of the message is used as the process Id of the remote connection, and the connection table entry for that process is updated
SendMsg("connected to LocalProcessId")
srcAddr=p1
the srcAddr of the message is used as the process Id of the remote connection, and the connection table entry for that process is updated

Fig. 9. Establishing a connection between two slave processes.

initiative, without the need for a third process, and would allow connection establishment at any time during execution. This approach would be straightforward to implement, but has not been pursued here because it is not required by the benchmark applications. The ConnectToProcess function would send a connection request message, designated as a system message, so that the message passing library itself would process the request message (the processing would involve a connection establishment protocol). The request message would not interfere with the application's operation because the message passing library does not deliver system messages.

V.B.7 The Processor Allocator

The message passing library is equipped with a rudimentary processor allocation mechanism that keeps track of the available processors in the system and the number of processes allocated to each processor.

The names of the available processors are all given in a *host file* that is read when the allocator is initialized. The processor on which the allocator is initialized (i.e. the master's processor) is also part of the pool of available processes, even if its name does not appear in the host file. There is no distinction between the master and slave processes for the purposes of allocation. Since a processor corresponds to a workstation, each processor name is actually a workstation's Internet host name.

The allocator provides the function GetNextAvailProcessor, which allocates processors according to a simple policy of selecting the processor to which the least number of processes have previously been allocated. Ties are broken consistently according to the order in which the processors appear in the host file.

V.B.7.1 Processor Allocator Data Structures

The processor allocator uses the following data structures (the terms processor and host are used interchangeably here):

NumFunctioningHosts : the number of hosts (i.e. processors) currently available to the allocator (an integer).

FunctioningHosts : the names of the currently-available hosts (an array of strings)

NumProcessesAllocated : the number of processors currently allocated to each host, initially all zero (an array of integers)

V.B.7.2 Processor Allocator Operation

The processor allocator must be initialized with a call to InitializeProcessorAllocator, which returns the processor ID assigned to the calling process's processor. InitializeProcessorAllocator initializes FunctioningHosts to contain the hosts given in the file named **`hostfile`**. A name that is listed in the file multiple times will only be included in FunctioningHosts once. The processor on which the master is started is not treated specially; if it appears in the host file, it will be treated the same as any other processor.

After initialization, each successive call to the function GetNextAvailProcessor will return the name and ProcessorId of the processor to which the least number of processes have been allocated. Thus, when GetNextAvailProcessor is called repeatedly, it will simply cycles through the hostfile.

If a specific processor is desired, the function AllocateToSpecificProcessor can be called to inform the allocator that the caller is going to use a given processor. AllocateToSpecificProcessor increments the NumProcessesAllocated value that corresponds to the given processor name, and returns the ProcessorId that corresponds to that name. The simple algorithm currently used by GetNextAvailProcessor will not necessarily continue according to the "next least loaded" policy, since it does not search the NumProcessesAllocated array for the minimum value. AllocateToSpecificProcessor is not presently used by the message passing library, so such improvements to the allocator were not pursued.

The processor allocator functions assume that they are called by a single master process, and hence do not perform any inter-process coordination.

V.C Recovery System

The term *recovery system* refers to the collection of recovery protocols that are integrated with the message passing library. The application need not be aware of the recovery system, though, for simplicity, the recovery system code is presently stored in the same library as the message passing code.

V.C.1 Interface to the Message Passing Library

The recovery protocol library is separated from the message passing library by the set of interface functions shown in Figure V.C.1. These functions are part of the recovery library and are called automatically by the message passing library for each message sent, received, or forwarded, as appropriate.

Recov_Initialize() is called by the message passing library to initialize each process

Recov_HandleReceive(Message *msg) is called by the message passing library for each message received

Recov_HandleSend(Message *msg) is called by the message passing library for each message sent

Recov_HandleForward(Message *msg) is called by the message passing library for each message forwarded; presently, Recov_HandleForward just calls Recov_HandleSend.

Figure V.C.1: Recovery system interface functions

Each recovery system interface function calls a corresponding function that implements the appropriate action for a specific recovery protocol. So, for example, **Recov_HandleSend(msg)** calls **FBL_HandleSend(msg)**, where the latter function implements the section of the family-based logging protocol that must be executed for each message sent. This level of indirection isolates the recovery system from the message passing library by allowing the specific recovery protocols to be changed without any changes to the message passing library functions. The diagram in Figure 10 illustrates these three levels of abstraction.

Adding a new recovery protocol to the system is simply a matter of writing the Initialize, HandleReceive, and HandleSend functions for the new protocol, and modifying Recov_Initialize, Re-

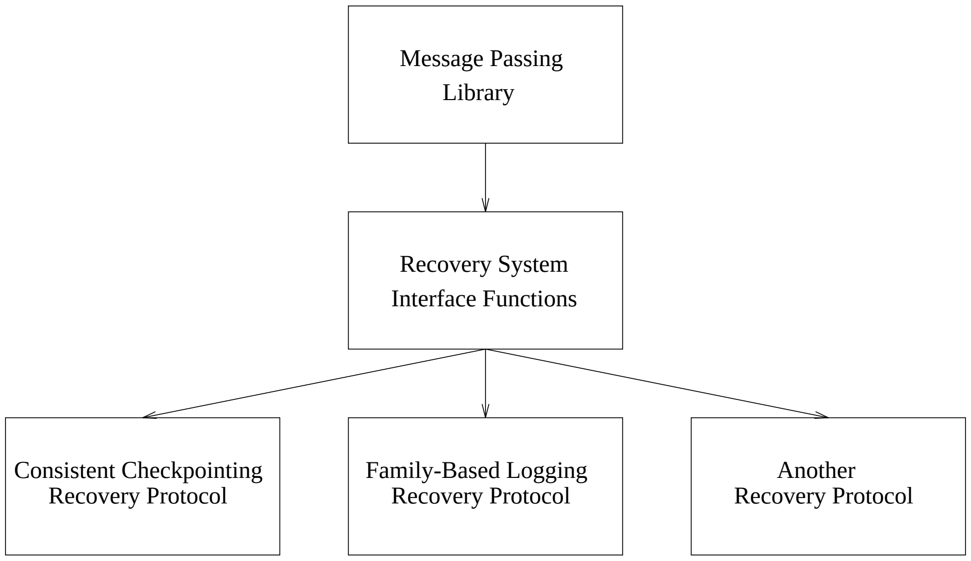
Message Passing Library
Recovery System Interface Functions
Consistent Checkpointing Recovery Protocol
Family-Based Logging Recovery Protocol
Another Recovery Protocol

Fig. 10. Generic recovery system interface functions.

cov_HandleSend, and Recov_HandleReceive, respectively, to call the new functions.

Multiple recovery protocols can operate simultaneously if each interface function calls multiple recovery protocol functions. Recovery protocols can be easily added to or removed from the recovery system by adding or removing a function call in each of the interface functions. Presently the choice of recovery protocols is made at compile time by conditional-compilation (`#ifdef`) directives in these functions, so it is necessary to recompile to add or remove a recovery protocol. Alternatively, the recovery protocol could be determined at run time for each process, but that possibility is left as future work. Finally, since the specific recovery protocol can be changed without changing the high-level interface, a change to the recovery system code should rarely require any change to the message passing library.

V.C.2 Internal Operation

This section describes the details of the consistent checkpointing and family-based logging implementations. Both protocols communicate with the message passing library via the interface functions (Initialize, HandleSend, and HandleReceive) described in Section V.C.1.

V.C.2.1 Basic Data Types

The following data types are used by the recovery system code:

- StateIntIndex – values of this type (integers) identify state intervals, send sequence numbers, and receive sequence numbers
- LoggingSiteId – this type is equivalent to the ProcessorId type
- IntSet – a set of integer values (used for keeping track of sets of ProcessId's, ProcessorId's, or LoggingSiteId's)

V.C.3 Family-Based Message Logging Implementation

This section describes the portion of the recovery system that implements the family-based message logging protocol π_α suggested by Alvisi and Marzullo [2]. They give a precise pseudocode version of π_α, so the main implementation issues are the choices of data structures to represent their abstract set data structures (e.g. set of determinants, set of messages, information piggybacked on a message, and set of ProcessId's).

Since both the determinant log and send log must be of finite size, a long-running application will fill them up. The best way to avoid log overflow is to periodically garbage collect the logs to bound their length. Another approach is to flush the logs to stable storage, when they become full. Prevention of log overflow is beyond the scope of this thesis [1]

A problem with the implementation of logging sites described here is that they store logs in shared memory that has a system-imposed maximum size (e.g. one megabyte) that can only be increased by the system administrator. Consequently, the implementation limits the size of the log of sent messages to 860 kilobytes and size of the log of determinants to 100 kilobytes. [2]

[1] Alvisi's dissertation (currently in progress) addresses FBL garbage collection techniques.

[2] The maximum log sizes sizes are determined by constants SENDLOG_SIZE_BYTES and DETLOG_SIZE_BYTES in the file family-msglog.c.

V.C.3.1 Representing Sets of Integers

The InsSet type is implemented as a fixed-length array of bits. An IntSet contains the element with value *i* if the *i*th bit of the bit vector is set. If the range of element values is known to be between zero and the number of bits in a machine word, then the bit vector can be represented as a single machine word for quick access. Otherwise, the bit vector consists of multiple bytes and the byte that contains a given bit must be calculated.

V.C.3.2 Determinant Log

The determinant log is an array of DetLogEntries, where a DetLogEntry is a determinant plus a set containing the ProcessId's of the processes at which this determinant is known to be logged.

```
typedef struct DetLogEntry {
     ProcessId source, dest;
     SequenceNum ssn, rsn;
     IntSet logged_at;
} DetLogEntry;
```

The determinant log is a fixed-length block of memory referred to as DetLog (its size is set by the value of the constant DETLOG_SIZE_BYTES, which is currently 100 kilobytes).

If logging sites are not in use, DetLog is simply declared as an array

```
/* DETLOG_MAXENTRIES = DETLOG_SIZE_BYTES/sizeof(DetLogEntry) */
DetLogEntry DetLog[DETLOG_MAXENTRIES];
```

If logging sites are in use, DetLog is a pointer that is set in FBL_Initialize to point to a region of shared memory:

```
DetLogEntry *DetLog;    /* initialized in FBL_Initialize */
```

DetLogEntries are added to the DetLog with the help of the DetLog_CurrentOffset, which is an integer that can be stored either in the local process's memory (no logging sites) or in shared memory (when logging sites are in use).

```
        int *DetLog_CurrentOffset = NULL;
```

Specifically, DetLog_CurrentOffset points to an integer value which is the number of determinants currently stored in the DetLog. Each time a determinant is added, the value is incremented. The function AddToDetLog adds a determinant to the log by copying the given determinant (newEntry) to the log. If logging sites are in use, each process is assigned a region of shared memory to which only that process writes and from which all other processes on the processor read without synchronization. A region is implemented as a Unix shared memory segment identified by a small integer. This partitioning of the shared log ensures mutual exclusion for modification operations. AddToDetLog updates the region that is assigned to the local process. The count of log entries is stored in shared memory along with the log, and is incremented only after each new log entry has been made. The AddToDetLog function's C language implementation is shown in Figure V.C.2.

```
AddToDetLog(DetLogEntry *newEntry)
{
  currentDetLogEntryp = DetLog + *DetLog_CurrentOffset;

  if (currentDetLogEntryp >= DetLogEndPtr) {
    Error("AddToDetLog: DetLog is full\n");
    /* free up some space (not implemented) */
  }
  memcpy(currentDetLogEntryp, newEntry, sizeof(DetLogEntry));
  (*DetLog_CurrentOffset)++; /* added one DetLogEntry */
}
```

Figure V.C.2: The FBL AddToDetLog function adds a determinant to DetLog

Finally, the PrintDetLog function is shown in Figure V.C.3 as an example of a function that retrieves determinants from DetLog. If the log is shared among multiple processes, then the regions of all processes on the local processor must be traversed. Each process has an array DetLog_ShmIds that contains the shared memory segment identifiers of all regions on the local processor (DetLog_ShmIds is initialized in FBL_Initialize).

```
PrintDetLog()
{
  DetLogEntry *detLog, *detLogEntryp;
  ProcessId p;
  int *detLog_CurrentOffset, detLogBytes;

  printf("--- DetLog contains:\n");

#ifdef LOGSITE_ENABLED
  for (p=0; p < MAX_PROCESSES; p++) {
    if (DetLog_ShmPtrs[p] != NULL) {
      Debug((Level3, "Portion of shared log owned by process %d:\n", p));

      detLog_CurrentOffset = (int *) DetLog_ShmPtrs[p];
      detLog = (DetLogEntry *) (DetLog_ShmPtrs[p] + DETLOG_HEADER_SIZE);
      currentDetLogEntryp = detLog + *detLog_CurrentOffset;
#else
      detLog = DetLog;
#endif

      for (detLogEntryp = detLog; detLogEntryp < currentDetLogEntryp;
   detLogEntryp++) {

PrintDetLogEntry(dbgLevel, detLogEntryp);
Debug((dbgLevel, "\n"));
      }
#ifdef LOGSITE_ENABLED
    }
  }
#endif
}
```

Figure V.C.3: PrintDetLog shows how the DetLog can be traversed.

V.C.3.3 Message Log

The SendLog resembles the DetLog, but the SendLog entries are variable length to accommodate variable length message data. A send log entry is of the form $e = (data, ssn, dv, dest)$, and is implemented with the SendLogEntry structure shown in Figure V.C.4. The SendLog itself is declared in the same way as the DetLog.

```
typedef struct SendLogEntry {
  int dataLen; /* length in bytes of the data element */
  /* total size of a SendLogEntry is sizeof(SendLogEntry)+dataLen */
  char *data;
  SequenceNum ssn;
  StateIntIndex dv[MAX_LOGGING_SITES];
  ProcessId dest;
} SendLogEntry;
```

Figure V.C.4: SendLogEntries make up the SendLog, the log of sent messages.

Since SendLog entries may differ in length, accessing arbitrary elements involves starting at the beginning of the SendLog and stepping through each entry, which requires advancing a pointer by the number of bytes occupied by the entry. To allow new elements to be added quickly, the number of bytes currently used in the SendLog is stored in a variable (called SendLog_CurrentOffset, which can be either in local or shared memory, analogous to DetLog_CurrentOffset), so the memory location at which the next entry should be added can be quickly calculated. Alternatively, a pointer to the next available memory location in SendLog could be maintained, but that approach would not interact well with the shared memory SendLog, which might appear in different memory locations at different processes. The function AddToSendLog is similar to AddToDetLog.

V.C.3.4 Interface with the Message Passing Library

The functions FBL_HandleSend, FBL_HandleReceive, and FBL_HandleAck closely follow the pseudocode given by Alvisi and Marzullo for handling a message send, receive, and acknowledgment, respectively. To keep the amount of piggybacked information small, family-based logging requires information about message acknowledgments. Unfortunately, that is not available from the TCP protocol used here. Some approaches that the recovery system could use to determine when a message has been acknowledged are:

1. An explicit acknowledgment message can be sent by the recovery system (using SendMsg) in response to each application message. This is the approach currently used in the implementation. The FBL_HandleAck function is currently called directly by the message passing library for each such acknowledgment message received. Alternatively, the explicit acknowledgments could be sent less frequently. That possibility is left as a source of future work.

2. A weaker form of implicit acknowledgment, such as a "maximum received sequence number" that the sender piggybacks on every application message to indicate the messages that it has received from the destination process. This approach is feasible for FIFO channels if each pair of processes exchanges messages at similar rates in both directions. If some processes exchange messages rarely, the technique could benefit from an extension that transitively passes the maximum RSN's.

3. The message passing library could be changed to use a user-level transport protocol based on UDP instead of the current TCP transport protocol. Then the user-level transport protocol could directly call FBL_HandleAck.

V.C.3.5 Piggybacking

Piggybacking refers to the addition of information by a recovery protocol to application messages. The piggybacked information is invisible to the application. The only information that family-based logging piggybacks is a set of determinants to be logged at the receiver process. The implementation of piggybacking uses the PiggybackData structure to refer to the information to be piggybacked:

```
typedef struct PiggybackData {
  ListOfDeterminants dets;
  /* additional elements can be added here for use by other protocols */
} PiggybackData;
```

The message passing library sends and receives the piggybacked data as part of the messages, but the details of format of the piggybacked data are handled entirely by the recovery system through the use of "data marshaling" functions. When the message passing library is ready to write piggybacked data (in SendMsg), it calls the WritePiggybackData function for each application (i.e. non-system) message. The ReadPiggybackData function "unmarshals" the piggybacked data and must read data in the format written by WritePiggybackData. The message passing library calls ReadPiggybackData for each application message. The SizeofPiggybackData function must return the number of bytes occupied by a given PiggybackData variable. When a message is sent, the number of bytes of data piggybacked on the message is set in a field of the message header to be the value returned by SizeofPiggybackData for the PiggybackData variable associated with the message. Extending the recovery system to piggyback additional data is simply a matter of adding the new data's type to the PiggybackData structure and adding the code necessary to marshal and unmarshals the data to WritePiggybackData, ReadPiggybackData, and SizeofPiggybackData.

V.C.3.6 Logging Sites

The logging site implementation takes advantage of an idea suggested independently by Alvisi and Marzullo [2] and Vaidya [30] to reduce message logging overhead when multiple processes share the same processor. Vaidya's suggestion assumes that at most one overlapping failure will occur, and can be adapted for family-based logging as follows:

- Determinants need not be piggybacked onto messages sent between processes on the same processor; instead, determinants are stored in a log in shared memory accessible to all processes on the same processor.

- When a message is sent between processes on different processors, all determinants in the shared log are piggybacked onto the message and the shared log is cleared when the message is acknowledged. After that acknowledgment, those determinants have been logged and can be retrieved to replay the sender's execution if the sender fails.

- A coordinated checkpoint is periodically executed at each processor; only the processes that reside on the same processor are included in the coordinated checkpoint.

V.C.3.7 Implementation of Logging Sites

Alvisi and Marzullo [2] suggest a similar approach that can be used with any family-based logging protocol, although they do not consider checkpointing. They use the term *logging site* to refer to a memory address space shared by multiple processes, and suggest that a logging site can correspond to a processor shared by multiple processes. Whereas in ordinary FBL the set $m.log$ is the set of processes at which the message m is logged (see Section II.D), in FBL with logging sites, $m.log$ is the set of logging sites at which $\#m$ is logged.

Modifying an FBL protocol to use logging sites involves replacing most references to Processid's with references to LoggingSiteIds, and changing the HandleSend function so that the determinant $\#m$ is not piggybacked on a message sent to a process q when q is known (by the sender) to be on a logging site at which $\#m$ is already logged [2].

Specifically, Alvisi and Marzullo introduce the function $L(p)$ to denote the logging site associated with process p (this function is constant since processes are assumed to not move between logging sites) and the function $P(l)$ to denote the set of processes associated with logging site l. Finally, the function $P^{\cup}(S)$ denotes all processes associated with the logging sites in the set S:

$P^{\cup}(S) = \bigcup_{l \in S} P(l)$

Now the FBL HandleSend function can be modified so that a determinant $\#m$ will not be piggybacked on a message m' sent to a process q when $q \in P^{\cup}(m.log)$. Furthermore, throughout the FBL protocol (for example, when manipulating the *logged_at* set that is maintained for each entry in the DetLog), references to a process p are replaced with $L(p)$.

Assuming that some logging sites are associated with multiple processes, the performance tradeoffs are as follows:

1. Reduction in communication overhead, since the number of piggybacked determinants is reduced. Computation overhead may be slightly increased, however, because the $P^{\cup}$ and L

functions must be evaluated often. An efficient implementation should be able to reduce the overhead of evaluating those functions so that they so not significantly impact performance.

2. Reduction in the total amount of memory used on a processor, since all processes on the processor share a single log. Without logging sites, each process would have its own private log. If there are n processes on a given processor, and each process uses a log size of k bytes, then nk bytes will be required without logging sites, while only k bytes will be required with logging sites. The drawback of the shared logging site is that the partitioning scheme, which is necessary to ensure mutually-exclusive log updates, imposes some computation and memory overhead.

Thus, if shared memory accesses are as fast as ordinary memory accesses, logging sites should be beneficial when communication and memory are relatively expensive compared to computation. According to the performance results given in Section VI.C, logging sites do reduce the number of determinants piggybacked on each message, but can actually decrease performance in some cases. There is most likely still room for performance improvement in the implementation, which has not been extensively optimized for performance.

V.C.3.8 Implementation in the Recovery System

The logging site implementation keeps track of logging sites in a table called LogSite_L_array. The logging site table is initialized by calls to LogSite_AssociateProcess(l, p) to associate process p with logging site l. For simplicity, the master is assumed to be the only process that calls LogSite_AssociateProcess, and all such are assumed to occur during system initialization. That limitation could be removed by extending LogSite_AssociateProcess to broadcast the new association to all existing slaves. This problem is a case of the more general problem of maintaining globally-shared data in a distributed environment. The problem is not significant for simple application programs and is not addressed further here. Figure V.C.5 shows the LogSite_AssociateProcess function. MasterInitialize calls LogSite_AssociateProcess to associate the master process with the processor on which the master is running (which is always defined to be ProcessorId 0), and Exec calls LogSite_AssociateProcess each time a new process is created.

```
LogSite_AssociateProcess(LoggingSiteId l, ProcessId p)
{
  /* set LogSite_L_array[p] to l for fast lookup of L(p) later */
  LogSite_L_array[p] = l;
}
```

Figure V.C.5: LogSite_AssociateProcess

LogSite_AssociateProcess is only executed at the master. When the master creates a new process, it sends its logging site information to the new process in a *LogSiteInit* message, which contains a LoggingSiteTable. When a process receives a LogSiteInit message, it copies the LoggingSiteTable from the message to its own local LoggingSiteTable variable. It is assumed that no further LogSite_AssociateProcess calls will occur. That assumption could be lifted, however, by modifying LogSite_AssociateProcess to broadcast every new association to all existing processes. The explicit initialization phase currently used prevents any inconsistencies that might arise from such dynamic state updates.

The function $L(p)$, which returns the logging site associated with a given process p, is implemented by a lookup into the array *LogSite_L_array*, in which $L(p) = \mathit{LogSite\_L\_array}[p] = l$ for a given valid ProcessId p and LoggingSiteId l. With LogSite_L_array, $P(l)$ and $P^{\cup}(S)$ can be specified as follows:

$$P(l) = \{p \mid \mathit{LogSite\_L\_array}[p] = l\}, \tag{V.1}$$

$$P^{\cup}(S) = \{p \mid \mathit{LogSite\_L\_array}[p] \in S\}. \tag{V.2}$$

$P^{\cup}(S)$ can be determined by the algorithm shown in Figure V.C.6, which is presently implemented by the code shown in Figure V.C.7. This implementation is reasonably efficient if MAX_CARDINALITY is small.

```
$found \leftarrow$ **false**
**for each** $l \in S$ **do**
    **if** $LogSite\_L\_array[p] = l$ **then**
        $found \leftarrow$ **true**
        stop the loop
```

Figure V.C.6: Determining $P^{\cup}(S)$

```
isMember = FALSE;
for (l=0; l < MAX_CARDINALITY; l++)
  if (GETBIT(S, l) && LogSite_L_array[p] == l) {
    isMember = TRUE;
    break;
  }
```

Figure V.C.7: Implementation of $P^{\cup}(S)$

V.C.4 Coordinated Checkpointing Implementation

The coordinated checkpointing protocol used in the implementation is based on that of Manetho [7], with modifications to allow coordinated checkpointing to include only the processes on the same processor. Different processors can choose to checkpoint independently of each other, so checkpointing need not involve inter-processor communication. The system will still be recoverable if message logging is done for inter-processor messages.

A *coordinator* process is a distinguished process that is responsible for initiating the consistent checkpointing protocol.

At each process p there is a set called ProcessesToCkpt that contains the ProcessId's of all the processes to which the local processor must send marker messages. When the master process creates a new process q, q is added to the master's ProcessesToCkpt set only if q is the first process created on q's processor. Hence, at each processor, the first process created on that processor P is designated the processor's *processor coordinator* of P.

For each subsequent process p created on processor P, the master sends a message "CoordinateProcess p" to the processor coordinator. When the processor coordinator receives "CoordinateProcess p", it adds p to its ProcessesToCkpt set.

The master process is the only process at which a periodic "alarm timer" automatically initiates a coordinated checkpoint. A coordinated checkpoint is initiated by calling the function TakeConsistentCheckpoint (which is done automatically at the master according to the checkpointing period). TakeConsistentCheckpoint increments the local process's CCN (consistent checkpoint number) variable, and then calls Checkpoint, which checkpoints the local process and all processes for which the local process is the coordinator. Figure V.C.8 shows the code for TakeConsistentCheckpoint and Checkpoint.

With this approach, the checkpointing policy used by processor coordinators can be modified easily to allow them to autonomously determine when to initiate a coordinated checkpoint of the processes on their processor (in that case the master would no longer send marker messages).

```
TakeConsistentCheckpoint()
{
  CCN++;
  Checkpoint();
}

Checkpoint()
{
  checkpoint_here(); /* libckpt function that checkpoints local process */

  /* if we are the coordinator for any processes, checkpoint them
   * by sending marker messages to them
   */
  if (numCkptProcesses > 0)        /* # of processors in ProcessesToCkpt */
    ConsistentCheckpoint(ProcessesToCkpt);
}
```

Figure V.C.8: TakeConsistentCheckpoint and Checkpoint

The function ConsistentCheckpoint sends a marker message to each process in the ProcessesToCkpt set, as shown in Figure V.C.9.

Finally, the function ConsCkpt_HandleSend is called by Recov_HandleSend for each message sent, and ConsCkpt_HandleReceive is called by Recov_HandleReceive for each message received. ConsCkpt_HandleSend and ConsCkpt_HandleReceive are given in Figure V.C.10.

The marker message specified in the protocol is implemented by sending one byte of out-of-band

```
ConsistentCheckpoint(ProcessSet ProcessesToCkpt)
{
  int i;
  MarkerMsg marker;
  ProcessId dstPid;

  marker.hdr.dataLen = sizeof(MarkerMsg) - sizeof(MessageHeader);
  marker.hdr.msgType = Msg_Marker;
  marker.hdr.CCN = CCN;

  for each p in ProcessesToCkpt {
    /* send out-of-band data to asynchronously notify receiver
     * that a marker message is arriving
     */
    send_oob_to_fd(ConnectionTable[p].sockfd, OOB_MARKER);

    /* send the marker message */
    SendMsg(p, (Message *) &marker);
  }
}
```

Figure V.C.9: ConsistentCheckpoint

```
ConsCkpt_HandleSend(Message *msg)
{
  if (OnSameProcessor(msg->hdr.dstAddr, LocalProcessId))
     msg->hdr.CCN = CCN;
  else
     msg->hdr.CCN = 0;  /* CCN=0: never cause a ckpt on a remote processor */
}

ConsCkpt_HandleReceive(Message *msg)
{
  if (msg->hdr.CCN > CCN) {
    /* msg's CCN > local CCN: initiate a tentative checkpoint */

    CCN = msg->hdr.CCN;

    Checkpoint();
  } else if (msg->hdr.CCN < CCN) {
    /* msg is a cross-checkpoint message; log it */
  }
}
```

Figure V.C.10: ConsCkpt_HandleSend and ConsCkpt_HandleReceive

data (with the value OOB_MARKER) immediately followed by an ordinary message passing message of type **MarkerMsg**. Out-of-band data is a feature of TCP; when out-of-band data is received, the receiving process is immediately asynchronously interrupted and informed of the data's arrival. This asynchronous interrupt ensures that processes that have not called receive will process the marker message. The arrival of a marker message causes a SIGURG Unix signal, which causes the operating system to call the SIGURG signal handler function, which is defined by the message passing library to call the appropriate message handling function based on the value of the out of band data. The only value currently recognized is OOB_MARKER, which causes HandleMarkerMessage to be called. HandleMarkerMsg is shown in Figure V.C.11.

If an application message arrives before a marker message with the same CCN value, then the protocol specifies that the application message will cause the receiver to checkpoint. The corresponding marker message should have no effect, so the implementation discards a marker message that has a CCN less than or equal to the receiver's CCN.

```
HandleMarkerMsg(MarkerMsg *markerMsg)
{
  if (markerMsg->hdr.CCN > CCN) {
    CCN = markerMsg->hdr.CCN;
    Checkpoint();
  } else
    printf("HandleMarkerMsg: discarding marker msg\n");
}
```

Figure V.C.11: HandleMarkerMsg

The message passing library (SendMsg, specifically) automatically sets the CCN value in the **MarkerMsg**'s header CCN to be the sender's CCN (application message CCN's are set in the same way). The **MarkerMsg** type contains a redundant CCN field that is explicitly set by ConsistentCheckpoint (as shown in Figure V.C.9; this CCN field of **MarkerMsg** is present for clarity only and could be removed.

Cross-checkpoint ("lost") messages can be detected by comparing the CCN tagged on each message to the receiver process's CCN and logged at the receiver (i.e. treated as input messages, as in Manetho [7]).

V.D Summary

The successful incorporation of multiple different recovery protocols (coordinated checkpointing and family-based message logging) into the recovery system has shown that the system is indeed extensible.

An application programmer can select any combination of recovery protocols by setting configuration options when the application is compiled. Protocols that are not desired are not compiled into the application. The application is not required to call any recovery system functions, so the recovery system is transparent to the application programmer. Instructions for running applications and configuring the system are given in Appendix B.

As described in the next chapter, an actual application program that originally ran under Manetho [7] on the V distributed operating system [6] has been successfully ported, with no significant changes, to use the message passing library and recovery system.

CHAPTER VI

Performance Evaluation

VI.A Introduction

The performance of the recovery system implementation is evaluated here by comparing the execution times of benchmark application programs executing with the recovery system to the execution times of the same programs executing without the recovery system. All measurements are of failure-free execution.

The benchmark application used here, called *gauss* [10, 7], has been converted from Manetho to use the message passing library. This application has a master-slave structure, in which a single master process initializes the slaves and reports the total running time upon their completion. `gauss` performs Gaussian elimination with partial pivoting. The problem is divided evenly between all processes. At each iteration, the process that has the pivot element sends the pivot column to all other processes. This application is compute-bound, i.e., computation is the performance bottleneck, as opposed to communication. This chapter calls the size of the matrices the "problem size" For example, a problem size of 128 means that 128 equations are being solved, and that each of the matrices used by the program (of which there are three) contains 128x128 elements (a matrix element is of type `double`, and the size of a `double` is eight bytes). When the problem size is 128, each slave sends approximately 200 messages during an execution.

The primary performance metric is the failure-free overhead of the recovery protocol, in terms of application execution time, number of messages, and message size.

All benchmarks were run on Sun SPARCstation-5s running Solaris 2.4. Each workstation has 32 megabytes of memory, and the network is a 10 megabit per second Ethernet. The term *processor* used here corresponds to a single workstation, and each workstation has a single CPU. The program code was compiled with the Sun `acc` compiler using the "-O" optimization option. The master process was run on a separate processor in all cases, so even the "one processor" cases actually used

Failure-Free Application Execution Time (seconds)		
Configuration	Problem Size	Execution Time
4on1	128	8.4
4on2	128	11.9
4on2	1024	170.1
4on4	1024	92.67
8on8	1024	134.12

TABLE I

Execution time of the *gauss* application with no recovery protocols.

two processors (one for the slaves, and another for the master).

The following system configurations were tested:

`4on1` Four slaves, all executing on a single processor

`4on2` Four slaves executing on two processors. Hence there are two slaves per processor.

`4on4` Four slaves, each executing on a separate processor.

`8on8` Eight slaves, each executing on a separate processor.

Table I gives the failure-free execution time of the *gauss* application for each of these configurations. For a problem size of 1024, the `4on2` configuration requires 84 percent more time to run than the `4on4` configuration. That is likely because of the large memory size of the 1024 problem size (2.3 megabytes). Hence the execution time increases as the number of processes per processor increases. Furthermore, the execution time *increases* as the number of processors are added, as shown by the increase in execution time from `4on4` to `8on8`. With `gauss`, as the number of processors increases, the increased communication overhead outweighs the decreased computation overhead at each processor, so speedup is negative.

VI.B Coordinated Checkpointing

The performance of the coordinated checkpointing implementation is summarized in Table II. The Disk Type column indicates the type of disk to which the checkpoints were written. For the

Percent Increase in Failure-Free Application Execution Time			
Disk Type	Configuration	Ckpt Size (MB)	Percent Increase
local	4on4	2.3	22.92
	8on8	1.24	10.27
NFS	4on4	2.3	63.4
	8on8	1.24	265.5

TABLE II

The overhead of coordinated checkpointing, with a problem size of 1024 and intercheckpoint interval of 60 seconds.

local disk, the checkpoints were written to /var/tmp, whereas for NFS the checkpoints were written (all simultaneously) to a single shared network file system disk (on a SPARCstation-5 server). The number of checkpoints, which is approximately three for these tests, is the running time divided by the inter-checkpoint interval. None of the `libckpt` library's special options, such as forked or incremental checkpointing, were used in these tests.

VI.C Family-Based Message Logging

Table III summarizes the increase in failure-free execution time caused by family-based message logging (FBL) and family-based message logging with logging sites (FBL+LOGSITE). In both cases FBL is configured to tolerate at most one simultaneous failure. Logging sites maintain one (shared) log on each processor, instead of the per-process logs maintained by ordinary family-based logging. No checkpointing was done in these message logging tests. The percent increase shown for the `4on1` logging site case (6.9) was found by measuring an older implementation that used fewer shared memory segments but performed a file locking operation for each log access. The `4on1` case could not be tested with the current implementation because of an arbitrarily-imposed limit on the maximum number of shared memory segments per process. The test could be done for the current implementation, with the help of root access privileges, which are necessary to increase the maximum number of allowed segments.

The average number of determinants piggybacked by FBL on each application message is shown

Percent Increase in Failure-Free Application Execution Time			
Configuration	Problem Size	FBL (f=1)	FBL+LOGSITE (f=1)
4on1	128	11.9	<6.9
4on2	128	11.8	71.5

TABLE III

The overhead of the implementations of family-based message logging.

Average number of determinants piggybacked per message			
Configuration	Problem Size	FBL (f=1)	FBL (f=1) + LOGSITE
4on1	128	18.47	0
4on2	128	17.9	3.3

TABLE IV

Number of determinants piggybacked on each application message (each determinant is 20 bytes)

in Table IV. When all four processes are on the same processor, the logging site implementation never piggybacks any determinants. [1]

VI.D Profiling

Profiling is a method that collects statistics at run-time for the purpose of identifying functions that dominate execution time. This section gives the results of profiling the failure-free execution of one of the **gauss** application's slave process with no recovery protocols, with family-based logging, and with family-based logging augmented with logging sites. The profiling results are shown as tables produced by the Unix **gprof** utility. Each table entry is the name of a function, and the functions are sorted in decreasing order by percentage of total execution time. Time for which the process was blocked or put to sleep by the operating system is not included in the times produced by **gprof**. Thus profiling identifies the functions that are most computationally-intensive and that use the most CPU time, but total execution time may also include additional time during which the process was waiting for an event or resource.

[1]The single processor case cannot actually tolerate a failure and is included here to represent a case where many processes share one processor in a multi-processor system.

The table columns have the following meanings:

self seconds – the total number of seconds spent in this function, not including time spent in any other function

calls – the number of times this function was called

percent time – the percentage of the total running time of the process used by this function

cumulative seconds – a running sum of the number of seconds used by this function and all those listed above it

self ms/call – the number of milliseconds for which a call of this function executed, on the average

total ms/call – the number of milliseconds for which a call of this function executed, including the time for which its descendants executed, on the average

For the case of nonrecoverable execution, Table V gives the profiling results for four slaves sharing a single processor and Table VI shows the results for four slaves with two processors (two slaves per processor). Table V shows that the write and read system calls dominate execution time. The sigprocmask function, which is also a source of significant overhead is used to disable and enable signal interrupts. The Compute function is part of the underlying computation executed by the slave process, and the mcount function is part of the profiling system. Notice that writing (sending) is slower than reading (receiving). There is no obvious explanation for that difference, although a protocol could be tuned to take advantage of the difference if it is a common phenomenon.

The results of profiling family-based logging are presented in Table VII, which is for a run with four slaves sharing a single processor, and Table VIII, which is for a run with four slaves on two processors. Logging sites should in principle be able to improve upon the performance shown in these two tables, since in both cases multiple processes share a single processor.

According to Table VII, the most expensive FBL protocol functions are Update_Logged_Dets and Log_New_Dets, but neither of those functions is overly expensive compared to the total execution time. The implementation cannot exclude determinants that are known to be stable from consideration for piggybacking. That is, the latter two functions must scan all determinants in the DetLog to find those that must be piggybacked. Performance could be improved by a data representation capable of excluding a determinant from the scan after the first time it is found to be stable, if such a representation could be implemented with low overhead.

Finally, the results of profiling family-based logging with logging sites are presented in Table IX, which is for a run with four slaves on two processors. The `4on1` case could not be tested because of an arbitrarily-imposed limit on the maximum number of shared memory segments per processor as described in Section VI.C. However, measurements of a previous implementation that used fewer shared memory segments but performed locking show that performance increases noticeably as the number of piggybacked determinants per message approaches zero.

Referring to Table IX, logging sites reduce the time spent in Update_Logged_Dets by more than an order of magnitude. Log_New_Dets is also sped up. Hence determinant processing overhead in general is reduced by the use of the logging site concept. The most obvious area for improvement is IntSet_Cardinality, which currently runs in time $O(MAX\_SET\_CARDINALITY)$ but could be changed to run in time $O(1)$ by storing the cardinality explicitly with each set. The total execution time of IntSet_Cardinality is sufficiently small that the change was not considered important enough to implement.

Overall performance has not been improved by the logging site implementation, although the read and write system calls execute for less time with logging sites (because less data is piggybacked on each message).

VI.E Analysis

Checkpointing has a small impact on performance when checkpoints are written to local disks, but a much greater impact when checkpoints are written over the network simultaneously to a shared disk. In the case of `8x8`, writing the checkpoints to the shared disk slowed the application down by a factor of 2.66. That same test shows that the overhead of eight processes writing checkpoints of

1.24 megabytes each is much greater than the overhead of four processes writing checkpoints of 2.3 megabytes each. This unexpected result is likely attributable to decreased throughput of the NFS server caused by the greater number of simultaneous clients. Data block sizes transferred over the network may get smaller and the number of interruptions of the server probably gets larger as more clients are added. A detailed analysis of the NFS server itself would provide an explanation for this result, but such an undertaking is beyond the scope of this thesis.

Checkpointing overhead can be reduced by decreasing the checkpointing rate, which is artificially high here because of the short application run time. Forked and incremental checkpointing, which are provided by **`libckpt`** [19], would also improve performance,

The overhead of the family-based message logging implementation is around twelve percent for the application tested, which is higher than average overhead of one to four percent of the Manetho protocol [7, 10]. This implementation has not been tuned for performance, and can likely be improved. The functions that deal with sets of integers (especially IntSet_Cardinality) can be improved. Furthermore, the DetLog data structure could be replaced with a representation that excludes determinants that are known to be stable from consideration in scans of the determinant log.

In the **`4on1`** configuration, logging sites reduced the amount of piggybacked information to zero and hence improved the performance of family-based logging. In the **`4on2`** configuration, although logging sites reduced the amount of piggybacked information, they increased execution time by nearly seventy percent. The overhead is in the computations performed to implement the logging site protocol, as described in Section V.C.3.7. Further efforts should be made to improve the implementation's performance before discounting the logging site technique, however. There is a good chance that their computational overhead can be reduced to the point where they never slow down the application.

VI.F Summary

The logging site implementation can in some cases improve the performance of family-based logging.

The logging site protocol eliminates computational overhead in processing determinants, but the overall performance gain is not large for the problem sizes tested. Larger problem sizes and values of f should be tried to measure the full potential of the logging site concept. Specifically, a larger value of f will cause many more determinants to be piggybacked per message, and the logging site concept should be able to dramatically reduce the piggybacking overhead. An environment in which communication is expensive, such as a wide-area network, could certainly benefit from the reduction in piggybacked data that logging sites provide. A very communication-intensive application would also be likely to benefit from logging sites.

Although the implementation does allow both message logging and checkpointing to be used at the same time, the performance overhead of the combination was not measured because it is likely to be the sum of the overheads of the two protocols. Hence that measurement is left as future work. The tests run here are short, but their results are nonetheless informative. Longer-running tests should also be tried, but they are left for future work. Longer-running tests can easily be run for checkpointing by increasing the problem size. For message logging, however, a garbage collection method will have to be implemented to deal with the problem of log overflow. The implementation also allows arbitrary values of family-based logging's f value (the maximum number of tolerable overlapping failures), but the tests here are all for $f = 1$. The performance implications of larger values of f should be investigated.

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
============================================================================
 36.9       1.40     1.40      812     1.72     1.72  _write [8]
 26.4       2.40     1.00      796     1.26     1.26  _read [12]
  9.0       2.74     0.34     3254     0.10     0.10  _sigprocmask [14]
  7.9       3.04     0.30      388     0.77     0.77  _poll [19]
  5.3       3.24     0.20       15    13.33    13.33  _open [22]
  2.6       3.34     0.10       16     6.25     6.25  _close [26]
  2.6       3.44     0.10       10    10.00    10.00  _ioctl [27]
  2.6       3.54     0.10        6    16.67    16.67  _stat [29]
  2.4       3.63     0.09      199     0.45    11.66  ReceiveMsg [6]
  2.1       3.71     0.08      128     0.63     0.63  Compute [35]
  0.5       3.73     0.02      786     0.03     0.03  realloc [48]
  0.3       3.74     0.01     2112     0.00     0.00  __fabs [62]
  0.3       3.75     0.01     1417     0.01     0.01  _free_unlocked [52]
  0.3       3.76     0.01      262     0.04     3.89  SendMsg [11]
  0.3       3.77     0.01      133     0.08     0.08  strlen [60]
  0.3       3.78     0.01       18     0.56     0.56  _sigfillset [63]
  0.1       3.79     0.01                             _mcount (443)
  0.1       3.79     0.01                             _moncontrol [69]
  0.0       3.79     0.00     2173     0.00     0.00  _mutex_lock_stub [191]
  0.0       3.79     0.00     2108     0.00     0.00  _mutex_unlock_stub [192]
  0.0       3.79     0.00     1629     0.00     0.00  _sigemptyset [193]
  0.0       3.79     0.00     1627     0.00     0.10  BlockSig [23]
  0.0       3.79     0.00     1627     0.00     0.00  UnblockSig [85]
  0.0       3.79     0.00     1627     0.00     0.00  _sigaddset [194]
  0.0       3.79     0.00     1627     0.00     0.00  _waitid [195]
  0.0       3.79     0.00     1087     0.00     0.02  malloc <cycle 2> [50]
  0.0       3.79     0.00      794     0.00     0.01  free [58]
  [remaining functions contribute little to execution time]
```

TABLE V

Profiling results for a nonrecoverable run (no recovery protocols present). Problem size=128, and there are four processes, all on one processor.

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
============================================================================
 52.1       1.60     1.60      804     1.99     1.99  _write [7]
 13.0       2.00     0.40       15    26.67    26.67  _open [11]
  6.5       2.20     0.20     3270     0.06     0.06  _sigprocmask [18]
  6.5       2.40     0.20      798     0.25     0.25  _read [19]
  6.5       2.60     0.20      389     0.51     0.51  _poll [20]
  5.2       2.76     0.16      199     0.80     7.74  ReceiveMsg [8]
  3.3       2.86     0.10                             _getmsg [29]
  2.3       2.93     0.07      128     0.55     0.55  Compute [30]
  0.7       2.95     0.02      782     0.03     2.14  writen [6]
  0.7       2.97     0.02      781     0.03     0.40  readn [12]
  0.7       2.99     0.02      331     0.06     0.06  memcpy [39]
  0.3       3.00     0.01     1635     0.01     0.01  _sigaddset [49]
  0.3       3.01     0.01     1088     0.01     0.02  malloc <cycle 1> [41]
  0.3       3.02     0.01      786     0.01     0.01  realloc [48]
  0.3       3.03     0.01      425     0.02     0.04  search_pending_queue [44]
  0.3       3.04     0.01       58     0.17     0.28  fread [43]
  0.3       3.05     0.01        2     5.00     5.02  _endutxent [47]
  0.3       3.06     0.01        2     5.00    31.97  _ttyname_r [31]
  0.3       3.07     0.01                             _mcount (441)
  0.0       3.07     0.00     2175     0.00     0.00  _mutex_lock_stub [184]
  0.0       3.07     0.00     2112     0.00     0.00  __fabs [185]
  0.0       3.07     0.00     2110     0.00     0.00  _mutex_unlock_stub [186]
  0.0       3.07     0.00     1637     0.00     0.00  _sigemptyset [187]
  0.0       3.07     0.00     1635     0.00     0.07  BlockSig [28]
  0.0       3.07     0.00     1635     0.00     0.00  UnblockSig [76]
  0.0       3.07     0.00     1635     0.00     0.00  _waitid [188]
  0.0       3.07     0.00     1418     0.00     0.00  _free_unlocked [54]
  [remaining functions contribute little to execution time]
```

TABLE VI

Profiling results for a nonrecoverable run (no recovery protocols present). Problem size=128, and there are four processes, two per processor.

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
============================================================================
 34.9       2.04     2.04     1203     1.70     1.70  _write [8]
 25.6       3.54     1.50     1182     1.27     1.27  _read [11]
  7.4       3.97     0.43     4666     0.09     0.09  _sigprocmask [18]
  5.1       4.27     0.30      388     0.77     0.77  _poll [21]
  5.1       4.57     0.30       11    27.27    27.27  _ioctl [23]
  5.0       4.86     0.29      195     1.49     1.53  Update_Logged_Dets [24]
  3.2       5.05     0.19      195     0.97     1.07  Log_New_Dets [27]
  2.7       5.21     0.16      199     0.80    19.15  ReceiveMsg [6]
  1.7       5.31     0.10       16     6.25     6.25  _close [37]
  1.6       5.41     0.10                             _mcount (481)
  1.4       5.49     0.08      128     0.63     0.63  Compute [42]
  1.0       5.55     0.06    79228     0.00     0.00  IntSet_Cardinality [47]
  0.9       5.60     0.05      193     0.26     0.32  FBL_HandleSend [46]
  0.7       5.64     0.04      189     0.21     0.77  FBL_HandleAck [32]
  0.5       5.67     0.03    36776     0.00     0.00  _thr_main_stub [56]
  0.5       5.70     0.03     1165     0.03     1.49  readn [10]
  0.2       5.71     0.01    18156     0.00     0.00  _fflush_u [69]
  0.2       5.72     0.01     4872     0.00     0.00  IntSet_Add [70]
  0.2       5.73     0.01     4101     0.00     0.00  IntSet_SetEmpty [71]
  0.2       5.74     0.01     3890     0.00     0.00  GetIndexOfArrMax [72]
  0.2       5.75     0.01     3371     0.00     0.00  _free_unlocked [65]
  0.2       5.76     0.01     2651     0.00     0.01  malloc <cycle 2> [54]
  0.2       5.77     0.01     2333     0.00     0.00  UnblockSig [77]
  0.2       5.78     0.01     2333     0.00     0.00  _sigaddset [78]
  0.2       5.79     0.01     2217     0.00     0.01  realloc [61]
  0.2       5.80     0.01     1168     0.01     1.90  writen [7]
  0.2       5.81     0.01      389     0.03     0.03  Update_Eval_Help [75]
  [remaining functions contribute little to execution time]
```

TABLE VII

Profiling results for FBL (no logging sites). Problem size=128, and there are four processes, all on one processor.

```
 %   cumulative   self              self     total
time   seconds   seconds    calls  ms/call  ms/call  name
============================================================================
37.5       1.91     1.91     1203     1.59     1.59  _write [8]
19.6       2.91     1.00     1184     0.84     0.84  _read [12]
 8.0       3.32     0.41     4670     0.09     0.09  _sigprocmask [17]
 6.2       3.64     0.32      195     1.62     1.63  Update_Logged_Dets [19]
 5.0       3.89     0.26      195     1.31     1.42  Log_New_Dets [20]
 3.9       4.09     0.20      389     0.51     0.51  _poll [24]
 3.9       4.29     0.20       11    18.18    18.18  _ioctl [25]
 3.1       4.45     0.16                             _mcount (479)
 2.2       4.56     0.11      199     0.55    15.77  ReceiveMsg [6]
 2.0       4.66     0.10       16     6.25     6.25  _close [36]
 2.0       4.76     0.10       15     6.67     6.67  _open [37]
 1.4       4.83     0.07    80700     0.00     0.00  IntSet_Cardinality [40]
 1.2       4.89     0.06      128     0.47     0.47  Compute [45]
 1.0       4.94     0.05      193     0.26     0.34  FBL_HandleSend [41]
 0.4       4.96     0.02    18339     0.00     0.00  fflush [55]
 0.4       4.98     0.02     2335     0.01     0.01  UnblockSig [60]
 0.4       5.00     0.02     1526     0.01     0.01  memcpy [59]
 0.2       5.01     0.01    18346     0.00     0.00  _fflush_u [70]
 0.2       5.02     0.01     4055     0.00     0.00  _mutex_unlock_stub [75]
 0.2       5.03     0.01     4021     0.00     0.00  IntSet_SetEmpty [71]
 0.2       5.04     0.01     2652     0.00     0.01  malloc <cycle 2> [58]
 0.2       5.05     0.01     2195     0.00     0.00  realloc [68]
 0.2       5.06     0.01     1945     0.01     0.01  GetNthLargestElement [72]
 0.2       5.07     0.01     1168     0.01     1.78  writen [7]
 0.2       5.08     0.01      389     0.03     0.54  _select [22]
 0.2       5.09     0.01      195     0.05     0.05  Update_D [74]
 0.2       5.10     0.01      190     0.05     0.58  FBL_HandleAck [33]
 [remaining functions contribute little to execution time]
```

TABLE VIII

Profiling results for FBL (no logging sites). Problem size=128, and there are four processes, all on one processor.

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
=============================================================================
 36.0       1.60     1.60     1186     1.35     1.35  _write [8]
 18.0       2.40     0.80     1185     0.68     0.68  _read [13]
 11.7       2.92     0.52     4654     0.11     0.11  _sigprocmask [15]
  9.0       3.32     0.40       20    20.00    20.00  _open [16]
  6.8       3.62     0.30      389     0.77     0.77  _poll [20]
  4.1       3.80     0.18                             _mcount (492)
  3.2       3.94     0.14      128     1.09     1.09  Compute [28]
  2.7       4.06     0.12      200     0.60    12.45  NewReceiveMsg [6]
  1.4       4.12     0.06    60217     0.00     0.00  IntSet_Cardinality [35]
  1.4       4.18     0.06      193     0.31     0.37  FBL_HandleSend [33]
  0.9       4.22     0.04     2028     0.02     0.02  realloc [40]
  0.7       4.25     0.03      189     0.16     0.58  FBL_HandleAck [31]
  0.5       4.27     0.02     1162     0.02     1.59  writen [7]
  0.5       4.29     0.02      195     0.10     0.18  Log_New_Dets [42]
  0.5       4.31     0.02      195     0.10     0.11  Update_Logged_Dets [46]
  0.2       4.32     0.01    36417     0.00     0.00  _thr_main_stub [63]
  0.2       4.33     0.01    17968     0.00     0.00  _fflush_u [60]
  0.2       4.34     0.01     2985     0.00     0.01  _free_unlocked [44]
  0.2       4.35     0.01     2202     0.00     0.02  malloc <cycle 1> [39]
  0.2       4.36     0.01     1093     0.01     0.01  memcpy [61]
  0.2       4.37     0.01      968     0.01     0.01  IntSet_SetEmpty [62]
  0.2       4.38     0.01      389     0.03     0.82  Select [18]
  0.2       4.39     0.01      389     0.03     0.80  _select [19]
  0.2       4.40     0.01      262     0.04     4.09  SendMsg [9]
  0.2       4.41     0.01      261     0.04     0.06  AllocateNewMsg [53]
  0.2       4.42     0.01      190     0.05     3.23  WritePiggybackData [14]
  0.2       4.43     0.01      128     0.08     6.83  ForwardMsg [12]
  [remaining functions contribute little to execution time]
```

TABLE IX

Profiling results for FBL with logging sites. Problem size=128, and there are four processes, two per processor.

CHAPTER VII

Conclusion

The primary goals of this thesis work were to create a new implementation suitable for evaluating recovery protocols in a distributed environment, and to investigate and develop new techniques for efficient, transparent recovery.

A message passing library and the failure-free portion of a recovery system for standard Unix workstations has been implemented, and the successful incorporation of multiple different recovery protocols (coordinated checkpointing and family-based message logging) into the recovery system has shown that the system is indeed extensible. The ease with which an application originally written for a distributed operating system was converted to use the message passing library shows that the implementation is reasonably complete and usable by application programmers.

The recovery system implementation also includes the first known implementation of the *logging site* technique [2, 30] for maintaining message logs in shared memory.

Performance measurements of the implementation confirm previous claims that the overhead of checkpointing can be low [8]. As has been observed by others, checkpointing performs well when checkpoints are written to local disks, but can perform abysmally when several large checkpoints are written to a shared disk simultaneously [31].

The overhead of the family-based message logging implementation is slightly greater than expected, but still reasonable. The logging site technique improves performance slightly when many processors share a processor, but actually increases failure-free run time slightly when few processors share a processor. The implementation's performance can most likely be improved to bring those results closer to expectations, and may improve as the problem size and value of f (the maximum number of overlapping failures) increase.

A technique has been presented for allowing a process to alternately use message logging during deterministic execution and checkpointing during nondeterministic execution, and reduces dependency tracking overhead to that necessary to record dependencies on nondeterministic processes to improve output commit performance. The concept of *stateful* dependency tracking, in which each vector entry consists of both a state interval index and a state value, was defined as a generalization of the nondeterministic dependency tracking.

The idea of reactive replication in message logging was also presented. This technique allows a message logging protocol to tolerate multiple overlapping failures with no more failure-free overhead than would be required to tolerate only a single overlapping failure, with the restriction that every two failures must be separated by some minimal interval of time that is sufficient for the non-failed process to prepare for another failure by replicating data crucial to the recovery of the failed process. If the timing assumptions are met, the reactive technique eliminates the need to increase the amount of information piggybacked by family-based logging when increasing the number of simultaneous failures that can be tolerated.

Although performance was not measured experimentally for the either the nondeterministic or reactive method, both methods decrease communication overhead and hence are likely to increase failure-free performance.

VII.A Future Work

The implementation should be extended to include the recovery portion of message logging and coordinated checkpointing. Furthermore, the nondeterministic and reactive methods remain to be implemented. The recovery system implementation should be tested with longer-running tests.

The problem of detecting failures in practice on a network of workstations should be addressed. Since TCP timeout values are relatively large (greater than one minute), failure detection will likely have to do its own polling of remote processors. It may be possible make use of the existing NFS (Network File System) timeout mechanism, which uses frequent polling and has a relatively short timeout value. The choice of timeout value could be made based on trial and error, or an analytical model could be constructed to determine an optimal value.

A final area for future work is the choice of when to checkpoint to minimize both failure-free overhead and rollback distance when the beginning and end of each period of nondeterminism are known exactly, as they are with the **BeginND** and **EndND** events. Some additional application-provided information, such as an estimate of the frequency and duration of future intervals of nondeterminism, would be useful in choosing the best time at which to checkpoint.

APPENDIX A

The Simulated Fail-Stop Protocol (Sim-FS)

A.A Introduction

This appendix summarizes the work of Sabel and Marzullo [23] relevant to simulating the fail-stop model in asynchronous systems. The Sim-FS protocol can be used with the reactive replication technique, as described in Section IV.G. This appendix is present to provide some background information on the Sim-FS protocol.

The fail-stop model [25] requires that two conditions be satisfied in any run of the system:

FS1 The failure of a process is eventually detected by all processes that do not crash

FS2 No false failures are detected

Sabel and Marzullo show that in an asynchronous system with crash failures, it is impossible to implement both **FS1** and **FS2**. However, they describe a model, called *simulated fail-stop* (**sFS**) that is "indistinguishable" from fail-stop and can be implemented in an asynchronous system. The **sFS** model consists of the **FS1** condition and four new conditions that are weaker than, and replace, **FS2**. The four new conditions are:

sFS2a If process i detects that process j has crashed, then eventually j will crash even if i's detection was erroneous

sFS2b The failed-before relation must always be acyclic

sFS2c A process never detects its own failure

sFS2d Once i detects the failure of j, then any subsequent messages sent by i to any process k will not be received until k has also detected the failure of j.

A.A.1 Lower Bounds for Protocols Implementing sFS

Sabel and Marzullo give the following lower bounds on message complexity and replication for failure detection protocols that implement **sFS**.

In a *one-round* protocol, a process i exchanges one round of messages with other processes before deciding that a process j has failed. Thus a single process cannot unilaterally detect the failure of another process. If a unilateral decision were allowed, then a limit would be imposed on the processes that another process could detect as faulty.

When a process suspects that another process may have failed (e.g. because of a communication timeout), the suspecting process initiates a failure detection protocol. A round consists of two phases: in the first phase, a message is sent by process i to all other processes, followed by a second phase in which processes send an acknowledgment message to i. The first message is called $SUSP_{i,j}$ and the acknowledgment message is $ACK.SUSP_{i,j}$. When the failure-detection protocol completes, the initiator i will either become crashed itself or decide that j has failed.

A one-round protocol for **sFS** must ensure that cycles do not occur in the failed-before relation, which means that in any run there must be at least one process that participates in all failure detections. Furthermore, a process a that suspects the failure of process b cannot communicate with b directly, because b may have crashed. Instead, the failure detection protocol must enlist the help of other processes. Process a must receive information from enough other processes to be sure that process b has not decided that a has failed, and a must also distribute information to enough other processes to be sure that if a decides that b has failed, then b will not subsequently decide that a has failed. The information that a must distribute is "a suspects the failure of b." Since a must know that this information has been received by other processes, it must receive messages from other processes acknowledging that the failure of b is suspected.

The *quorum set* Q_{ij} of $failed_i(j)$ is defined as the set of processes from which i has received acknowledgment messages relating to its suspicion of j's crash. The set Q_{ab} must be large enough to ensure that after b hears from Q_{ba}, b will not execute $failed_b(a)$. That requirement is satisfied when the intersection of the sets Q_{ab} and Q_{ba} is not empty. Sabel and Marzullo call that property the *Witness Property (W)* because the quorum sets for any two failure detections must share a common *witness* process (that means there must be a process w that is in the quorum set of all failure detections).

Let t denote the maximum number of crashes in any run, including those that arise from erroneous suspicions. Note that Sabel and Marzullo define t to apply to an entire run. However, in a system with processes that recover, t should be the maximum number of simultaneously-crashed but not recovered processes (i.e. overlapping crashes). The Witness Property constrains t as described below.

A one-round protocol can ensure that W holds by requiring a process to wait for a response from every other process, except those that are suspected to have failed, before detecting a failure. A process that never fails and is never suspected of failing will be a witness to every failure detection that is executed. Although this protocol only requires that $t < n$, it also requires the initiating process to wait an amount of time proportional to the number of processes in the system (n).

In a second possible implementation, a process is required to wait for a fixed, predetermined number of responses before detecting a failure. The size of the quorum for which a process must wait is thus reduced at the expense of a stronger restriction on the number of failures that can occur. Specifically, when the quorum set is of fixed and equal size for each failure detection, the size of each quorum set must be strictly greater than $\frac{n(t-1)}{t}$. Furthermore, if that minimum quorum size is used in a one-round protocol, then $n \geq t^2$ must hold [23].

APPENDIX B

Running the System

This appendix contains some information about using the implementation in practice.

B.A Operation

A program that uses the message passing library must initialize itself as a master and start slaves by calling Exec. Although the same program could be used as a master and slave (with an internal if statement to distinguish the two cases), it seems simpler to separate the master and slave into two different programs. Hence, the application is started by running the master program after everything has been compiled. The names of the remote workstations to use must be specified in the file `hostfile`, one per line.

If the preprocessor symbol EXEC_IN_XTERM was defined when exec.c was compiled, then a separate xterm window will be created for each slave process. In this case the remote workstations must have permission to open the local workstation's display, which can be set with `xhost +`. If the master is being run on a different workstation than the local display, then the environment variable MPDISPLAY should be set to the local display name (there does not seem to be any reason not to simply use DISPLAY instead of MPDISPLAY; exec.c would have to be modified to do that):

```
setenv MPDISPLAY vanilla:0
```

tells the message passing library to open the windows on `vanilla`.

To enable debugging, the line

```
DBG=-g -DRUN_DEBUGGER
```

should be present in the `Makedefs` file in the root of the source tree. When RUN_DEBUGGER is defined, each child (slave) process will be started under the debugger specified in `msgpass/config.h` (DBX or GDB). When each slave window opens, it is necessary to type the run command printed by the master (or simply copy the command using the X paste buffer). To debug the only master, RUN_DEBUGGER need not be defined, and the master can be started by `dbx master` and then `run arg1 arg2 ... argN`.

If the application is run twice within a short period of time, the socket may not be bindable. The present solution for that problem is an environment variable that allows the port number to be changed manually:

```
setenv SERVPORT XXXX
```

After typing that at a shell prompt, the application can be run. The value XXXX should be greater than the previous value plus the number of processes in the previous run. An alternative approach is to modify the code that does the bind in `msgtcp.c` to keep trying larger port numbers until the bind succeeds.

Profiling can be done by adding the line

```
PROFILE=-pg
```

to the `Makedefs` file, defining `PROFILE` in exec.c, and recompiling. When the application is run, each slave must run in a separate directory, because the profiling data will be dumped to the current directory when a slave exits. The solution used here is to create a subdirectory named with the ProcessId of each slave that will be created. These subdirectories must be created manually, and the `PROFILE` symbol in exec.c causes the slaves to be started in them. In each directory there must be a symbolic link to the slave binary in the parent directory. The slaves will need to be told where to find there data files, which can be done by specifying names like `../a128` when starting the master, or by making a symbolic link to each necessary file in each subdirectory.

```
cd gauss
for i in 1 2 3 4
   mkdir $i
   cd $i
   ln -s ../slave .
   cd ..
done
```

Now, after the application finishes, there will be a file called **`gmon.out`** in each slave subdirectory. To translate **`gmon.out`** to a readable file:

```
cd 1
gprof slave > profile.N.S.config
```

where N, S, and config would identify the specific run (i.e. problem size N, S slaves, and recovery protocol configuration). The profiling information will not include times for which the process was blocked by the operating system and not using the CPU.

To enable specific recovery protocols, edit the **`Makedefs`** file:

```
RECOV=-DORDINARY_CONSCKPT -DORDINARY_FBL -DLOGSITE_ENABLED
```

enables all the protocols that presently have been implemented. A protocol can be removed by removing the appropriate `-D` option. The `libckpt` library is linked in if the substring `CKPT` appears in `RECOV`.

The `Debug` macro used in the code is defined in `debug.h` to call the function DebugPrint (defined in `util.c` when PRODUCE_DEBUG_OUTPUT is defined, and to do nothing otherwise. The first argument to `Debug` is a numeric verbosity level; the convention adopted here is that a smaller verbosity level should produce less output. The current verbosity level is controlled by VERBOSITY_LEVEL in `config.h`. Debug statements with a verbosity level argument greater than the current verbosity level will produce no output. The output appears in the log file associated with DebugFP.

Finally, a summary of RCS commands appears below. RCS revisions are stored as files in the RCS subdirectory of the current directory, which is a symbolic link to a corresponding RCS directory in the actual source tree. The current directory should be part of a "build" (i.e. "work" tree).

co file – check out the most recent version of the file without acquiring a lock (the checked out file will not be writable)

co -l file – checkout the most recent version of the file and acquire a lock (makes the file writable)

co -rX.Y file – check out version `X.Y` without acquiring a lock

ci file – check a file in and unlock it (the file will be deleted from the current directory)

ci -u file – check a file in and unlock it (the file will remain in the current directory, but will not be writable)

rcs -i file – initialize a newly-created file for use with RCS

rcs -u file – unlock the file (does not change the file's access permissions)

rcs -l file – lock the file (does not change the file's access permissions)

rcs -oX.Y file – delete (i.e. "obsolete") revision `X.Y`

rcsdiff file – show differences between the file in the current directory and its most recent checked in version

rlog file – show the file's RCS log

REFERENCES

[1] Lorenzo Alvisi, Bruce Hoppe, and Keith Marzullo. Nonblocking and orphan-free message logging protocols. In *Digest of Papers: The 23rd International Symposium on Fault-Tolerant Computing*, pages 145–154, June 1993.

[2] Lorenzo Alvisi and Keith Marzullo. Optimal message logging protocols. Technical Report TR94-1457, Cornell University Department of Computer Science, September 1994. Some sections are to appear in ICDCS-15 with the title *Message Logging: Pessimistic, Optimistic, Causal and Optimal*, and others have been submitted to FTCS-25 with the title *Trade-Offs in Implementing Optimal Message Logging Protocols.*

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

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

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

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

[7] Elmootazbellah N. Elnozahy. *Manetho: Fault Tolerance in Distributed Systems Using Rollback-Recovery and Process Replication.* PhD thesis, Rice University, Houston, Texas, October 1993.

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

[9] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit. *IEEE Transactions on Computers*, 41(5), May 1992.

[10] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. On the use and implementation of message logging. In *Digest of Papers: The 24th International Symposium on Fault-Tolerant Computing*, June 1994.

[11] Yennun Huang and Yi-Min Wang. Why optimistic message logging has not been used in telecommunications systems. Submitted to FTCS-25.

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

[13] D. B. Johnson and W. Zwaenopoel. Sender-based message logging. In *Digest of Papers: The 17th International Symposium on Fault-Tolerant Computing*, June 1987.

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

[15] David B. Johnson. Efficient transparent optimistic rollback recovery for distributed application programs. In *Proceedings of the 12th Symposium on Reliable Distribted Systems*, October 1993. Also available as Computer Science Technical Report CMU-CS-93-127, Carnegie Mellon University.

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

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

[18] Kai Li, Jeffrey F. Naughton, and James S. Plank. Checkpointing multicomputer applications. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 2–11, September

1991.

[19] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under Unix. In *Proceedings of the Winter Usenix Conference*, 1995.

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

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

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

[23] Laura S. Sabel and Keith Marzullo. Simulating fail-stop in asynchronous distributed systems. Technical Report TR94-1413, Cornell University Department of Computer Science, June 1994. Versions appear in Proceedings of the 13th Annual Symposium on Principles of Distributed Computing, August 1994, and Proceedings of the 13th Symposium on Reliable Distributed Systems, October 1994.

[24] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–234, August 1983.

[25] Fred B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145–154, May 1984.

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

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

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

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

[30] Nitin H. Vaidya. Some thoughts on distributed recovery (preliminary version). Technical Report 94-044, Texas A&M University Department of Computer Science, June 1994.

[31] Nitin H. Vaidya. A case for two-level distributed recovery schemes. In *Proceedings of SIGMETRICS/Performance*, 1995. Also available by anonymous ftp to ftp.cs.tamu.edu, directory pub/vaidya.

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

[33] Jian Xu and Robert H. B. Netzer. Adaptive independent checkpointing for reducing rollback propagation. In *Proceedings of the 5th IEEE Symposium on Parallel and Distributed Processing*, December 1993. Also available as Technical Report CS-93-25, Brown University Department of Computer Science.