UNIVERSITY OF CALIFORNIA

Los Angeles

Application-Transparent Error Recovery Techniques for Multicomputers

A dissertation submitted in partial satisfaction of the

requirements for the degree Doctor of Philosophy

in Computer Science

by

Tiffany Michelle Frazier

1995

© Copyright by

Tiffany Michelle Frazier

1995

The dissertation of Tiffany Michelle Frazier is approved.

Kirby A. Baker

Richard Muntz

David A. Rennels

Algirdas A. Avizienis

Yuval Tamir, Committee Chair

University of California, Los Angeles

1995

To my husband, Greg.

Table of Contents

Chapter One - Introduction 1

1.1. Basic Concepts 4

1.2. Application-Specific Techniques 7

1.3. Application-Transparent Techniques 9

1.3.1. Message Logging 9

1.3.2. Coordinated Checkpointing 11

1.4. Scope of the Thesis 14

1.5. Thesis Organization 18

Chapter Two - System Model 24

2.1. Computational Model 24

2.1.1. Fault-Free Operation 25

2.1.2. Operation with Failures and Recovery 26

2.2. Assumptions 29

2.3. Interprocess Communication 30

2.4. Error Detection 31

Chapter Three - Message Logging Techniques 35

3.1. Previous Work 36

3.2. Data Structures for Asynchronous Message Logging 40

3.3. Basic Approach of Message Logging 41

3.4. Recovering a Consistent Global State 45

3.5. Algorithms 50

3.5.1. Normal Operation and Checkpointing 50

3.5.2. Recovery 52

3.6. Proofs of Correctness 55

3.6.1. Deleting Regenerated Messages 56

3.6.2. Handling Orphan Messages 58

3.6.3. Messages Following Ordered Playback 62

3.6.4. Summary 68

3.7. Other Issues 69

3.8. Summary 71

Chapter Four - Coordinated Checkpointing Techniques 73

4.1. Previous Work 76

4.2. Basic Approach of Coordinated Checkpointing 79

4.2.1. Recording a Consistent Global State 79

4.2.2. Global Checkpointing 82

4.2.3. Recording Interacting Sets of Processes 83

4.2.4. Maintaining a Consistent Global State 83

4.2.5. Recovery From Consistent Global States 90

4.3. Coordinating Interacting Sets as Trees 93

4.3.1. Coordinating the Checkpoint/Recovery Sessions 95

4.3.1.1. Multiple Independent Checkpoint/Recovery Sessions 98

4.3.1.2. Multiple Coordinators 105
4.4. Synchronous Coordinated Checkpointing 114
4.4.1. Supporting Dynamic Virtual Channels and Processes 115
4.4.2. Synchronous Checkpointing of Interacting Sets 118
4.4.3. Concurrent Invocations of the Algorithms 121
4.4.4. Checkpointing with Packet-Switching 122
4.5. Recovery 123
4.5.1. Recovery From Communication Errors 123
4.5.2. Recovery From Node Failures 124
4.5.3. Handling Failures during Checkpoint Sessions 125
4.6. Asynchronous Coordinated Checkpointing 128
4.6.1. Asynchronous Checkpointing: 1 Coordinator, 1 Session 133
4.6.2. Asynchronous Checkpointing: N Coordinators, 1 Session 135
4.6.3. Asynchronous Checkpointing: N Coordinators, N Sessions 136
4.6.4. Summary 139
4.7. Minimizing the Cost of Checkpointing Using the Virtual Memory System 140
4.8. Conclusions 142
Chapter Five - Execution-Driven Simulation of Error Recovery Techniques 144
5.1. Introduction 145

vi

5.2. Testbed Design Considerations 147

5.3. The Testbed 148

5.3.1. The Interconnection Network 150

5.3.2. Simulating the Execution of Application Processes 152

5.3.3. The Operating System 155

5.3.4. The Host Processor (I/O Object) and Stable Storage (Disk Object) 156

5.3.5. Summary 157

5.4. The Recovery Schemes Simulated 157

5.5. Experiences with the Applications 160

5.6. Experiences with the Recovery Schemes 164

5.6.1. Message Logging 166

5.6.2. Coordinated Checkpointing 169

5.6.3. Optimizations 171

5.7. Simulator Performance 173

5.8. Previous Work 174

5.9. Summary and Conclusions 175

Chapter Six - Performance Evaluation 177

6.1. Detailed Itemization of Recovery Scheme Overhead 177

6.1.1. Overhead Of Checkpointing/Recovery of Independent Processes 178

6.1.2. Overhead of Asynchronous Message Logging 184

6.1.2.1. The Cost of Logging Messages to Disk 186

vii

6.1.2.2. Other Message Logging Techniques 187

6.1.3. Overhead Of Coordinated Checkpointing of Interacting Sets 194

6.2. Performance Evaluation Using DERT 198

6.2.1. Performance of Asynchronous Message Logging 200

6.2.1.1. Overhead Due to Network Bandwidth 209

6.2.1.2. Overhead Due to Disk Bandwidth 211

6.2.1.3. Reducing the Number of Orphan Processes 213

6.2.1.4. Committing Outputs to the External World 216

6.2.2. Performance of Coordinated Checkpointing 218

6.2.2.1. Overhead Due to Network Bandwidth 227

6.2.2.2. Overhead Due to Disk Bandwidth 229

6.2.2.3. Simulations with Increased Checkpoint Frequency 230

Chapter Seven - Conclusions 236

Bibliography 242

viii

List of Figures

1.1 Inconsistent and Consistent Checkpoints 5

1.2 The Domino Effect 6

2.1 Inconsistent Post-Failure States 27

3.1 Example of an incorrect state following recovery 44

3.2 The pre-failure execution of process Q 47

3.3 Recovery Up To the First Orphan Message 56

3.4 A Logged Message Can Become an Orphan 59

4.1 Merging Checkpoint Sessions 112

4.2 Disk Server Process Table 126

4.3 Preventing Checkpoints From Becoming Inconsistent 134

4.4 A Deadlocked Checkpoint Session 139

5.1 Structure of the Simulation Testbed 149

5.2 Application Execution Time With Message Logging 168

5.3 Application Execution Time With Coordinated and Independent Checkpointing 170

6.1 Application Execution Time with Message Logging 206

6.2 Application Execution Time with Message Logging, cont. 207

6.3 Application Execution Time with Coordinated and Independent Checkpointing 221

6.4 Application Execution Time with Coordinated and Independent

Checkpointing, cont. 222

6.5 Application Execution Time with Increased Checkpoint Frequency I 231

6.6 Application Execution Time with Increased Checkpoint Frequency II 232

List of Tables

1.1 Application-Transparent Error Recovery Techniques 19

5.1 Application Statistics 162

5.2 Out-Of-Sync Problem 164

5.3 Simulation of the Copy-On-Write Mechanism 172

5.4 Simulation Time Per Application Assembly Instruction 173

6.1 O/S Processing Overhead Per Application Message 202

6.2 O/S Processing Overhead and Communication Frequency 203

6.3 Total Messages Sent in Each AML Simulation 210

6.4 Percentage of Time that AML Keeps the Disks Busy 212

6.5 Disk Bandwidth Requirements of AML 213

6.6 Number of Orphan Processes Resulting from a Single Failure 215

6.7 Checkpointing Statistics for Coordinated and Independent Checkpointing 223

6.8 Total Messages Sent in Each Coordinated Checkpointing Simulation 228

6.9 Disk Bandwidth Requirements of Coordinated Checkpointing 229

6.10 Checkpointing Statistics (Increased Frequency II) 233

xi

ACKNOWLEDGMENTS

I must acknowledge the large number of people who helped to make this dissertation a reality. First among them is my advisor, professor Yuval Tamir, from whom I learned an immense amount. His guidance has been invaluable. Also important to my work was the advice and support of my cohorts in the VLSI Systems Laboratory: Marc Tremblay, Yoshio Turner, G Janakiraman, Sanjay Jain, Greg Frazier, and Hsin-Chou Chi. Whether the problem was coding, finding a reference, formatting a document, or a need for lively technical discussions, these were the people I turned to. Verra Morgan, I must acknowledge, for always being there whenever the little things, or big things, seemed about to throw a wrench into the works. I would also like to acknowledge my family, whose love and tireless support powered me through the long years of graduate school: my husband, Greg; my daughters, Mikaela and Joanna; my mother and her husband, Karen and Klaus Lukas; my father and his wife, Larry Gaber and Polly Archer; and my mother-in-law, Sue Frazier. Finally, I acknowledge my grandmother and grandfather, Irma and Gordon Kritzer, whose pride in me always meant so much.

VITA

November 23, 1964	Born in Ayer, Massachusetts
1981-1985	Edward and Hazel Felber Scholarship
1986	B.S. Computer Science University of Wisconsin-Madison Madison, Wisconsin
1986-1987	Teaching Assistant University of California, Los Angeles Los Angeles, California
1987-1989	Research Assistant University of California, Los Angeles Los Angeles, California
1989-1990	GTE Fellowship
1990	Teaching Assistant University of California, Los Angeles Los Angeles, California
1991	M.S. Computer Science University of California, Los Angeles Los Angeles, California
1991-1994	Programmer Analyst, Computer Science Department University of California, Los Angeles Los Angeles, California
1991, 1993-1994	Dean's Research Assistant University of California, Los Angeles Los Angeles, California

PUBLICATIONS AND PRESENTATIONS

1. T. M. Frazier and Y. Tamir, ‘‘Application-Transparent Error-Recovery Techniques for Multicomputers,’’ *The Fourth Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA (March 1989).

2. Y. Tamir and T. Frazier, ‘‘Application-Transparent Process-Level Error Recovery for Multicomputers,’’ *Hawaii International Conference on System Sciences-22*, Kailua-Kona, Hawaii (January 1989).

3. Y. Tamir and T. M. Frazier, ‘‘Error-Recovery in Multicomputers Using Asynchronous Coordinated Checkpointing,’’ Computer Science Department Technical Report CSD-910066, University of California, Los Angeles, CA (September 1991).

ABSTRACT OF THE DISSERTATION

Application-Transparent Error Recovery Techniques for Multicomputers

by

Tiffany Michelle Frazier

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1995

Professor Yuval Tamir, Chair

This dissertation addresses the problem of providing application-transparent error recovery to restore a valid system state in general-purpose scalable multicomputers following hardware failures. Message logging and coordinated checkpointing are the two main classes of error recovery techniques that provide transparent checkpointing and rollback of distributed applications. Several new algorithms are presented, their correctness proven, and their performance evaluated.

A new message logging algorithm uses *volatile message logs* to recover as much of the failed process' pre-failure execution as possible. Checkpointing of process states and logging of messages proceeds asynchronously with respect to process execution. This scheme is shown to be *correct* — in the presence of failures it can be used to recover one or more processes of a distributed application, achieving a *valid* system-wide application state.

Two new algorithms based on coordinated checkpointing of interacting sets of processes are presented and proven correct. These algorithms checkpoint multiple processes ''simultaneously'' such that their saved states are consistent. Logging of messages is not required. The first algorithm requires synchronous checkpointing —

while processes are checkpointed to stable storage, their execution is suspended. The second algorithm, *asynchronous coordinated checkpointing*, suspends process execution only during the taking of a *local volatile checkpoint*. Like many *independent checkpointing* techniques, the fraction of time a process is suspended is independent of network delays and the number of processes that constitute the application.

To evaluate the performance of these techniques we built DERT, a Distributed Error Recovery Testbed. DERT is built on top of an event-driven, multithreaded simulation kernel. Actual compiled distributed applications are executed on a simulated multicomputer. Instrumentation of the application assembly code is used to account for execution time and allow accurate simulation of periodic checkpointing. The testbed is fully functional — it can use one of several recovery schemes to restore one or more processes when simulated failures occur. Using this testbed we measured the impact of the recovery schemes and optimizations of these schemes on the error-free execution time of coarse-grained and fine-grained applications.

Chapter One
Introduction

Multicomputer systems, consisting of hundreds of processors interconnected by high-speed links, are now technologically and economically feasible[Dall88, Seit85, Whit85]. Such systems can achieve high performance for many applications at a low cost by exploiting parallelism. A large multicomputer consists of thousands of large VLSI chips and hundreds of printed circuit boards. If the entire system is likely to fail as a result of the failure of any of its components, the expected mean time between system failures is only a few hours[Peat81]. Such a high system failure rate is unacceptable for many applications that require continuous correct operation for tens or hundreds of hours. This problem will be exacerbated by the continued scaling of VLSI ICs, which will cause reliability to become a more dominant problem in the future, resulting in shorter chip lifetimes[Wood86]. Hence, the system must be able to operate correctly despite the failure of some of its components, i.e., it must be *fault tolerant*. In order to ‘‘tolerate’’ hardware faults, the system must be able to *detect* the error caused by the fault, *locate* the faulty component, *restore* a valid system state (since the state may have been corrupted by the fault), and (possibly) *reconfigure* the system so that it does not continue to use the faulty components[Rand78]. The topic of this thesis is distributed *error recovery*: the restoration of a valid system state following a hardware failure.

The scope of this thesis is limited to the support of ‘‘general-purpose’’ applications which have no hard real-time constraints. In such an environment

fault tolerance through hardware redundancy and error masking, e.g. TMR (triple modular redundancy), is not appropriate. Such techniques involve a very high level of overhead relative to a system with no fault tolerance (over 200 percent for TMR) which is not reasonable for non-real-time systems.

Software error recovery techniques based on dynamic redundancy can be broken into two classes: forward and backward error recovery. *Forward error recovery* techniques are inappropriate for general-purpose multicomputers because they require detailed knowledge of the behavior of the applications. Such techniques examine the failed state of an application and then correct, or ‘‘adjust’’, the failed state so that it meets specified acceptance criteria — hence no work is lost.

Backward error recovery techniques involve *checkpointing* and *recovery/rollback*. The states of all application processes that can fail are periodically saved (checkpointed) to stable storage, e.g. reliable disks. In the event of a hardware failure all processes which were affected or could have been affected by the error are *rolled back* to their checkpoints. Only backward error recovery techniques are suitable for a environment supporting a variety of applications. However, existing fault tolerance schemes using backward error recovery tend to have high levels of overhead and/or restrict the behavior of application software and message delivery protocols. This thesis presents new backward error recovery techniques that have low levels of overhead (which can be ‘‘tuned’’ to the needs of different applications), ensure efficient use of the communication network, and do not restrict the behavior of applications.

Given the difficulty of writing large distributed applications it is highly

desirable that the *virtual machine* visible to the programmer not be made more complex by the fault tolerance scheme. Hence, we restrict our attention to *application-transparent* techniques, where no restrictions are made on the behavior and structure of application software. Hence, non-application-transparent techniques such as recovery blocks and recovery schemes used in transaction-based systems are only briefly mentioned. We divide existing application-transparent error recovery techniques into two main classes: Message Logging and Coordinated Checkpointing. The overhead of these schemes includes the time and storage overhead incurred during normal operation, restrictions made on application processes and the message delivery system used, and time lost during recovery.

In the proposed techniques, checkpointing sessions are initiated by ‘‘timers’’ associated with each process[Bari83]. The frequency of checkpointing can thus be tuned to the specific needs of a task: a higher frequency of checkpointing results in higher overhead but in less work being lost when recovery is necessary. Thus the fault tolerance scheme can be ‘‘configured’’ to meet the requirements of each application instead of penalizing all applications equally in order to satisfy the requirements of the most demanding application.

Since the goal of multicomputers is to achieve high performance, the fault tolerance scheme must not slow down the system significantly, even with applications that are ‘‘communication-intensive’’ (exploit fine-grain parallelism). Thus performance and storage overhead during normal operation should be minimized at the cost of a potentially expensive recovery.

A recovery scheme, intended for use on a large multicomputer, should have

the following attributes:

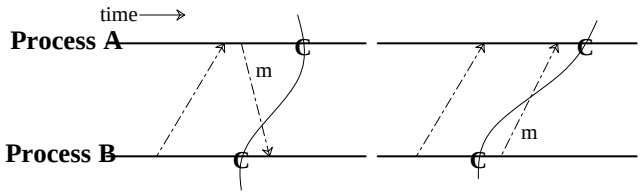
- *application-transparent*
- *minimal restrictions on application behavior*
- *low-overhead*: determined by the frequency of checkpointing, the ‘‘cost’’ of saving each checkpoint, extra operations required during normal computation for error detection and maintenance of bookkeeping information, and, secondarily, the ‘‘cost’’ of recovery.
- *n-fault-tolerant*: since nodes may be multitasking, it is desirable for the recovery scheme to be n-fault-tolerant, i.e., be able to handle the case where *n* processes in the system fail simultaneously. Such a recovery technique can consequently also handle multiple simultaneous node failures.
- *distributed*: the recovery scheme must not have a single point of failure, i.e., it cannot depend on any single resource.
- *suitable for a variety of applications*: computation-intensive and communication-intensive applications.

1.1. Basic Concepts

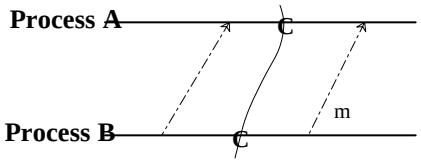
In a system which supports only *non-interacting* processes, checkpointing involves periodically suspending each process, copying the process' state to stable storage, and then allowing the process to resume normal computation. When a node fails, all information on the node is assumed to be lost. The checkpointed states of all the processes that were on the failed node are restored from stable storage to working nodes. The processes may then resume computation from these

older states. In order to be able to tolerate node failures that occur during checkpointing, the old checkpoint of a process is erased only after the new checkpoint has been completely written to stable storage.

Checkpointing and recovery are more complex when processes are allowed to interact via messages. When one or more processes are rolled back care must be taken to ensure that each pair of processes are *consistent* with each other and hence that the entire system state is *valid*. Specifically, every pair of processes must agree which messages have been sent and which have not[Chan85], such that no messages are lost or duplicated (see Figure 1.1).



a) *Inconsistent checkpoints*



b) *A consistent checkpoint*

Figure 1.1: In a), if A and B roll back to their latest checkpoints their states will be inconsistent. In the first case B expects a message from A which A's rolled back state has already sent (lost message). In the second case B will resend a message which A has already received (duplicate message). In b), the latest checkpoints of A and B are consistent with each other.

A set of checkpoints, one per process being rolled back, which are consistent with each other is called a *recovery line*[Rand78]. Graphically a recovery line can

be represented as a line which connects a set of checkpoints and intersects no communication lines (see Figure 1.1b). If processes are checkpointed independently, ignoring interactions with other processes, the recovery algorithm must find a recovery line amongst the available checkpoints[Wood81]. The major problem with this method (referred to as ‘‘independent checkpointing’’ in this thesis) is the fact that there is no guarantee that a recovery line exists and *domino effect*[Rand78] can occur, where, in the worst case, the entire system must be rolled back to its initial state (see Figure 1.2). Furthermore, many generations of checkpoints for each process have to be maintained on disk. This is necessary since, in the best case, it is desirable to roll back to a recent checkpoint (to minimize lost computations) while in the worst case it may be necessary to roll back to the initial process state.

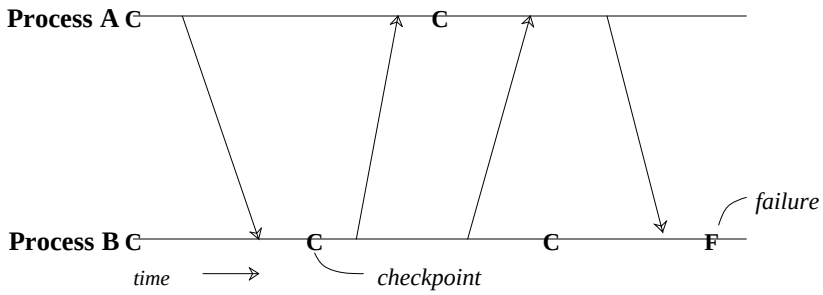


Figure 1.2: Process B fails and is rolled back to its latest checkpoint. Since B will later expect a message from A, A must also be rolled back, requiring B to roll back to a previous checkpoint. In this case, the entire system must roll back to its initial state.

1.2. Application-Specific Techniques

Many error recovery techniques have been proposed which avoid domino effect, but which restrict the actions of the application or require the application to contain special features. The two main areas of such application-specific techniques are Recovery Blocks[Rand75, Russ80, Wood81, Lee84, Kim84, Kim86] and Transaction Processing[Gray81, Nett86, Chin83, Skee81]. Both methods are similar in that they require applications to be structured as atomic actions, or transactions.

Recovery blocks, a language construct, provide facilities to detect failures (acceptance tests), and to reconfigure (one or more alternate algorithms which can be executed in place of the primary algorithm if an error does occur). The application programmer specifies where a recovery block starts (at which point a checkpoint is taken), primary and alternate algorithms which (assuming no software design errors) are functionally interchangeable, and an acceptance test at the end of the block. If the acceptance test fails the previously taken checkpoint is restored and an alternate code block is executed.

Using *conversations*[Rand75] recovery blocks can be applied to interacting processes. Processes that wish to exchange messages must enter a conversation (checkpointing their states) and then may interact freely, but only amongst themselves. Exiting the conversation requires that all participating processes pass their acceptance tests — and no process may proceed until all have done so.

Alone, however, neither recovery blocks nor conversations avoid the worst-case domino effect. One way to avoid the domino effect is for the application programmer to specify sequences of sets of recovery points that form recovery

lines. If an error is detected, the system rolls back to the last programmer-specified recovery line. This approach is not an option if the fault tolerant characteristics of the systems are to be kept hidden from the programmer. Further, in a large general-purpose system, keeping many generations of checkpoints for each process is not a viable option due to required disk space. Hence, recovery schemes that may require indefinite storage of all checkpoints (until the task terminates[Wood81]) are not a practical alternative for our target systems.

In transaction-processing techniques application processes are broken into transactions which always take the system from one consistent state to another. Each transaction terminates by committing or aborting its updates to stable storage. Such a system can ‘‘readily’’ handle recovery by treating hardware failures as aborts. While the system processes performing recovery can be transparent to the user application processes must be structured out of transactions. While some transactions can be executed in parallel *serializability* is required and can significantly degrade the level of concurrency achievable in a system. Transaction-processing recovery techniques are widely used in database systems.

A third technique, N-version (or ‘‘diverse’’) programming[Aviz85], worthy of mention, is an implementation of software masking redundancy. With n-version programming multiple versions of software are independently developed versions and then executed concurrently. A software driver supplies inputs to the different versions and then compares and votes on the outputs. This technique carries a large software design and development cost and has been used in real-time control environments.

As suggested by the name, application-specific techniques are designed for

use with certain types/classes of applications. Since we want to support a wide variety of general-purpose applications on a multicomputer the scope of this thesis is limited to application-transparent techniques. The next section reiterates the classes of application-transparent recovery techniques, briefly describes different schemes that fall in each class, and compares these schemes to the error recovery techniques presented in this thesis.

1.3. Application-Transparent Techniques

There are two major application-transparent approaches to avoiding domino effect in distributed checkpointing and rollback schemes:

I) Message Logging, where messages as well as process states are saved in order to allow the state of a restored process to be ‘‘adjusted’’ so that it is consistent with other processes in the system[Stro85, John87, Stro88, John90a, Sist89, Jalo89, Elno92a].

II) Coordinated Checkpointing, where processes are not checkpointed independently but are, instead, checkpointed in a coordinated way with some or all of the other processes in the system such that if recovery is necessary the restored states are guaranteed to be consistent[Bari83, Tami84, Koo87, Tami89].

1.3.1. Message Logging

Message logging techniques checkpoint process states and (log) interprocess messages onto stable storage. When a process fails and is rolled back, its *message log* is played back to it, so that when the message log has been depleted, the process is in a state consistent with the non-failed processes in the system.

Message logging techniques require application processes to be *deterministic*: given a process state and a sequence of inputs (message log), the process will generate the same outputs. Processes can be checkpointed *independently*, thus minimizing disruptions to the entire system, and a single failed process can be recovered, potentially without interfering with the operation of other processes.

Many papers have been published which use the *message logging* technique as the basis for application-transparent recovery algorithms [Borg83, Powe83, Stro85, John87, Stro88, John90a, Sist89, Jalo89, Elno92a]. These algorithms have proved to be complex and difficult to understand. In this thesis a new message logging algorithm based on [Stro85] is presented. A correctness model derived from the ideas presented in [Lamp78, Chan85] is used as the foundation for a complete set of proofs. This model is also used to thoroughly explain the technical details of asynchronous (optimistic) message logging and to make corrections to previously published algorithms. The new algorithm uses both stable *and* volatile message logs to recover more of the failed process(es)' lost computation. The use of volatile message logs also reduces the number of non-failed processes (orphans) that are forced to roll back in order to recover a consistent system state. This work makes two major contributions: 1) a set of proofs that verify the correctness of this optimized algorithm and 2) a performance analysis of the algorithm using execution-driven simulation.

A complete message logging algorithm is presented. This recovery algorithm is intended for use on a general-purpose distributed system where communication between processes is via asynchronous message passing (e.g. a multicomputer). It is required that the algorithm be distributed, application-transparent, provide *n*-

fault tolerance, and place minimal restrictions on application behavior. The techniques presented in [John87, Sist89] assume that no additional failures will occur during recovery while the optimistic technique presented in [John90a] is a centralized algorithm and hence not appropriate for a scalable multicomputer. In order to support the most general communication model no synchronization (pessimism) [Borg83, Powe83, John87, Jalo89] should be imposed on the message passing protocol, for the purposes of recovery.

A complete set of proofs verify the correctness of the presented asynchronous message logging algorithm. The proofs given in previously presented papers have 1) only shown that their algorithms hold certain properties[Stro85, Stro88, Jalo89], 2) made stringent assumptions[Sist89], or 3) were the basis for a centralized algorithm[John88].

1.3.2. Coordinated Checkpointing

Coordinated checkpointing techniques checkpoint a set of processes together (the entire system or a subset thereof) in such a way that each pair of process states *on stable storage* are consistent with each other [Chan85]. The recovery algorithm is guaranteed to be able to find a recovery line and hence be able to recover a valid system state. Recovery and checkpointing are likely to disrupt more of the system with coordinated checkpointing than with message logging (or, more generally, independent checkpointing), since more than one process is involved. However, there is no need to store message logs and very little bookkeeping information is required.

In error-recovery schemes based on *global* checkpointing[Tami84, Tami87,

Chan85] a consistent ‘‘snapshot’’ of the entire system state is periodically checkpointed and when an error is detected the last checkpointed state is restored. This scheme has very low performance overhead for long batch-oriented applications but is not appropriate for more interactive environments.

With coordinated checkpointing[Bari83, Koo87, Tami89] a consistent set of processes (an *interacting set*) are checkpointed or rolled back together. Disruptions to system operation are reduced compared to global checkpointing schemes and the overhead during normal operation is reduced compared to message logging techniques.

The new low-overhead synchronous and asynchronous coordinated checkpointing error recovery schemes presented in this thesis allow the system to recover from multiple simultaneous failures. Checkpointing involves saving a consistent snapshot of the states of an interacting set of processes in such a way that a valid global checkpoint of the system state is maintained in stable storage[Chan85, Tami84]. This allows the system to be *n-fault-tolerant*, keep only one generation of checkpoints on disk, and avoid the domino effect. In the proposed schemes checkpointing and recovery are done at the level of processes with no system-wide central coordination. Many checkpointing and recovery sessions may be active simultaneously. Unrelated sessions do not interfere with each other, while the actions of related ones are properly coordinated. Processes which are not part of the interacting set need not participate in checkpointing/recovery and may continue to do useful work.

The major disadvantage of synchronous coordinated checkpointing is the requirement to block computation of the processes being checkpointed during the

entire checkpoint session. This periodic *synchronous* checkpointing of multiple process states disrupts system operation and can result in significant performance overhead.

The asynchronous coordinated checkpointing scheme achieves the benefits of checkpointing processes independently, but without suffering from the domino effect. The key feature of this scheme is the use of volatile checkpoints *in combination with* the coordinated checkpointing technique to minimize disruption to normal operation due to checkpointing. A volatile checkpoint is simply a copy of the process state in local volatile memory. Checkpointing begins by copying the changed state of a process to local volatile storage, after which the process may resume execution. As discussed in Section 4.6, volatile checkpoints allow much of this local copying to be avoided through the use of a slightly modified virtual memory system. The rest of a checkpointing session involves identification of the set of processes to be checkpointed ‘‘simultaneously’’ and transfer of the volatile checkpoints to stable storage. Thus, a process is suspended only a small fraction of the time it takes to complete a checkpointing session. As in many independent checkpointing techniques, this fraction of time is *independent of network delays and the number of processes that constitute the application*. The cost, compared to synchronous coordinated checkpointing, is an increase in the complexity of the checkpointing algorithm which must ensure that the volatile checkpoints are saved as part of a consistent global state even though the processes being checkpointed have resumed execution.

No message logging and minimal bookkeeping information is required. Also, no hardware support (e.g. clock synchronization) is assumed. Since processes are

suspended from execution only a small, locally determined, amount of time, our scheme should scale well and support both multi-tasking nodes and communication-intensive applications with less overhead than previously published coordinated checkpointing techniques.

Two recently published algorithms [Bhar88, Wang92] have stated ‘‘independent checkpointing’’ as a primary goal and advantage. These papers presented algorithms that, in order to avoid any synchronization penalty, do not guarantee forward progress of the recovery line maintained on stable storage. We argue that coordinated checkpointing is more general and can perform better than independent checkpointing — during recovery *and*, as shown in simulation results, during normal operation in a system without failures. Our claim is not necessarily true if the ‘‘independent’’ checkpointing algorithm actually relies on some form of clock synchronization (low-level coordination) to limit its overhead as in the simulation results given in[Wang92].

1.4. Scope of the Thesis

This thesis examines techniques to provide transparent recovery from hardware failures in general purpose multicomputers. Both of the major classes of transparent recovery, message logging and coordinated checkpointing, are considered in detail. A system model based on[Lamp78] is described. This model provides the framework for proving the correctness of both message logging and coordinated checkpointing algorithms.

A new asynchronous message logging algorithm is presented. This algorithm

is based upon a previously published algorithm, but includes corrections, an optimization and several important details. The major contributions of this algorithm are 1) the accompanying proof of correctness and 2) the performance analysis. Previously published proofs were only partial proofs or made simplifying assumptions that rendered the algorithm impractical.

Two new coordinated checkpointing algorithms are also presented. The first is a synchronous algorithm that checkpoints and recovers interacting sets of processes. The correctness of this algorithm is shown. The basic idea behind this synchronous coordinated checkpointing algorithm is not new but many important details of the presented algorithm have not previously been addressed. This algorithm is then extended to be asynchronous — unlike in synchronous coordinated checkpointing, processes do not have to be suspended for the duration of their checkpoint session. Instead, normal execution is resumed as soon as possible after taking a volatile checkpoint, such that the process is suspended a fraction of time that is independent of network delays and the number of processes in the application. The cost is more complex send and receive primitives during the first phase of the checkpoint session. No asynchronous coordinated checkpointing algorithm has been previously presented.

Performance analysis is an important part of the work in the thesis. No attempt has previously been published to compare, through simulation or implementation, the overhead of the schemes belonging to the two major classes of transparent error recovery. Also, no such comparison has been made between schemes within the same class. Further, all previously published performance analyses and implementations of the techniques in these two classes (known to this

author) have been restricted to ethernet-based multiprocessors and have not considered general-purpose, scalable, multicomputers. This thesis contributes a first-order performance analysis of the schemes within and between the two classes. Most of the performance analysis, however, is performed on a simulation testbed.

The execution-driven simulator was first used to analyze the performance of the message logging techniques on a failure-free system. Real parallel applications were used along with low-overhead send and receive primitives similar to those used on current multicomputers. In this environment the message handling overhead required by asynchronous message logging algorithms has a substantial impact on application execution time. Two simple optimizations we proposed (elimination of the orphan detection filter and postponement of the dependency vector update at the receiver) offered significant improvement, while sending dependency vector differences often increased overhead. However, the message logging algorithm still scales poorly for three of the six applications and may require extensive hardware support to eliminate this trend.

We also showed that our simple optimization to reduce the likelihood of orphan processes occurring (by recording *RSN*s at the senders) can be beneficial to recovery performance due to the probability that, if one orphan process occurs, continued message transmission by this orphan is likely to result in many orphans. Future work involves validating this observation through performance analysis of the recovery procedure, comparing the message logging technique to other application-transparent error recovery schemes, examining possible hardware support, and working on methods to reduce the $O(n)$ dependency vector

requirement without losing the benefits of the message logging algorithm.

Secondly, simulations we ran using the same applications show that asynchronous coordinated checkpointing can be an appropriate error recovery technique for high-performance multicomputers in which fine-grain parallelism is exploited. Additionally, it is amenable to highly-effective optimizations that can be supported with relatively small enhancements to a conventional virtual memory system. With these optimizations, a volatile checkpoint can be taken by only a few dozen instructions without physically copying any pages in local memory. Hence, the disruption to normal processing will last, at most, tens of microseconds. The amount of data that has to be sent to stable storage is minimized and consists only of pages that have been modified since the last checkpoint and are resident in local memory when the checkpointing session is initiated.

Using general assumptions our simulations showed that, even *without* this optimization, the amount of overhead incurred by asynchronous coordinated checkpointing is nearly independent of multicomputer scale (with constant problem size). Asynchronous coordinated checkpointing performs at least as well as synchronous coordinated checkpointing and better when the checkpoint interval is decreased. This is because the overhead of the asynchronous algorithm is independent of network delays and the size of the interacted set (or the number of processes that constitute the application), unlike the synchronous algorithm. Furthermore, for communication-intensive applications (or those with large volatile checkpoints) the coordination of checkpointing processes is *desirable* in order to overlap checkpointing overhead; in several of our simulations independent checkpointing (without message logging) performed worse than even *fully*

synchronous coordinated checkpointing. Lastly, with coordinated checkpointing recovery is guaranteed to be free of the domino effect, while independent checkpointing techniques (without message logging) do not not guarantee progress of the recovery line.

1.5. Thesis Organization

Many distributed application-transparent error recovery techniques have been proposed [Rand78, Russ80, Wood81, Bari83, Hoss83, Powe83, Borg83, Tami84, Stro85, Koo87, John87, Stro88, Bhar88, Sist89, Jalo89, Tong92, Wang92, Elno92a] — Table 1.1 characterizes representative examples of the major approaches to application-transparent error-recovery. This table shows, for each listed recovery scheme:

1) Assumptions about the system model (e.g. does it require reliable communication and/or fail-stop nodes?).
2) The overhead incurred during normal operation (e.g. are messages logged?, is bookkeeping information transmitted with each message?).
3) The overhead incurred during checkpointing sessions (e.g. do checkpointing sessions involve multiple processes or are they independent?).
4) Recovery features (e.g. is the algorithm n-fault-tolerant?, how many processes have to be rolled back?).
5) Does the algorithm suffer from domino effect? (i.e. can it guarantee progress of a recovery line?).

Contrary to what is stated in the table (in the third row, where the second column is

Recovery Scheme	Assumes	Overhead During Normal Operation	Overhead During Checkpointing	Recovery Features
				Domino Effect?
Synchronous Message Logging (J&Z)	Reliable comm, fail-stop nodes	●Each message: add'l ovrhd incl. O/S processing; logged in volatile memory. ●Process is suspended if a 2nd message is sent before the 1st is logged.	Suspend process for asynch chkpt. (independent)	●Single process failures only.
				No.
Asynchronous Message Logging (S&Y,F&T††)	Reliable comm, fail-stop nodes	●Each message: add'l ovrhd incl. O/S processing; logged on stable storage.	Suspend process for asynch chkpt. (independent)	N-fault-tolerant; Restore failed processes and any orphans.
				May roll processes back > 1 times.
"Chase" Protocols (Wood)	Reliable comm, fail-stop nodes†	Maintain prop-list and PRI-list ●Each message: new PRI-list entries.	Suspend process for asynch chkpt. (independent)	N-fault-tolerant; Restore recovery line.
				●Yes — no guarantee recovery line will progress.
Independent Checkpointing (W&F)	Reliable comm, fail-stop nodes	●Each message: add'l ovrhd incl. O/S processing; logged in volatile memory.	Suspend process for asynch chkpt. (independent)	N-fault-tolerant; Restore local recovery line.
				●Yes — no guarantee recovery line will progress.
Global Checkpointing (C&L,T&S)	Fail-stop nodes	No.	●Suspend system for synch chkpt.	N-fault-tolerant; Restore entire system.
				No.
Synchronous Coordinated Checkpointing (K&T,T&F††)	Fail-stop nodes (K&T: Rel. Comm.)	Keep track of Buddies list.	●Suspend interacting set for synch chkpt.	N-fault-tolerant; Restore failed processes and their interacting sets.
				No.
Asynchronous Coordinated Checkpointing (T&F††)	Fail-stop nodes	Keep track of Buddies list.	Suspend process for asynch chkpt. (interacting set)	N-fault-tolerant Restore failed processes and their interacting sets.
				No.

Table 1.1: Main characteristics of application-transparent error recovery schemes. Major drawbacks are indicated with a (●).

marked with a †), Wood[Wood81] does not actually require that nodes be fail-stop but instead assumes that errors are detected by the applications themselves through the use of acceptance tests; the algorithm is, however, compatible with either assumption. Work presented in this thesis is highlighted in the table with (††).

In Chapter 2, the system model used in this thesis is described. A correctness model based on the concepts in[Lamp78] is presented. *Correct* execution is defined for both systems without failures and systems where failures can occur. A brief description as to how message logging and coordinated checkpointing techniques ensure correct execution of application processes, in the event of a hardware failure, is given. The details of the hardware model of the target multicomputer are then described in Section 2.2 and the error detection used is discussed in Section 2.4.

Message logging as a method for providing application-transparent error recovery is examined in detail in Chapter 3. Previous work is described in Section 3.1. The data structures which must be maintained for each process when using asynchronous message logging are enumerated in Section 3.2. The basic approach of asynchronous message logging algorithm, as originally proposed in[Stro85] is described in Section 3.3. Section 3.4 discusses, in detail, how the correctness model presented in Section 2.1 applies to asynchronous message logging algorithms. Our new asynchronous message logging algorithm, which has an optimization to reduce the number of orphans that occur as a result of a failure, provides a correction over previous algorithms, and provides essential details that have been previously omitted, is presented in Section 3.5. Normal operation and checkpointing is detailed in Section 3.5.1 and the recovery algorithm in

Section 3.5.2. The proof of the algorithm's correctness is given in Section 3.6. Several important issues are further discussed in Section 3.7 and conclusions are drawn in Section 3.8.

Chapter 4 starts out describing previous work in coordinated checkpointing techniques. A simple algorithm to record a global state, which is a reworded version of the snapshot algorithm in[Chan85] is briefly discussed in Section 4.1.1. The global checkpointing algorithm presented in[Tami84], which is a complete error recovery technique that records a consistent global state on stable storage, is discussed in Section 4.1.2. In Section 4.1.4 the global state recording algorithm is extended to record the states of a subset of the system (an interacting set of processes) as a way of *maintaining* a consistent global state on stable storage. Sections 4.1.4 and 4.1.5 prove the correctness of checkpointing and recovering (respectively) an interacting set of process to and from stable storage.

Section 4.2 describes that basic structure of an error recovery scheme based on the previous, proven, interacting set algorithms. Section 4.2.1 provides pseudo-code and some important high-level details on the coordinating of the checkpoint and recovery algorithms. Sections 4.2.1.1 and 4.2.1.2 show that non-interfering checkpoint/recovery sessions and coordinator conflicts in interfering checkpoint/recovery sessions are correctly handled, respectively. In Section 4.3 the complete synchronous coordinated checkpointing algorithm is presented including discussion of support for both the dynamic virtual circuit and packet-switching communication models. Section 4.4 provides details on recovering from communication errors (Section 4.4.1) and node failures (Section 4.4.2). Handling failures during checkpoint sessions is shown to not be a problem in Section 4.4.3.

A new asynchronous coordinated checkpointing algorithm is presented in Section 4.5. For ease of explanation the correctness of this algorithm is first described for the case where there is one active coordinator (process) and one active checkpointing session in the system (Section 4.5.1). This case is then extended to *n* coordinators and 1 checkpointing session (Section 4.5.2), and finally, *n* coordinators and *n* checkpointing sessions (Section 4.5.3). Techniques for minimizing data movement during checkpointing by using an enhanced virtual memory system are outlined in Section 4.6.

In Chapter 5 we present DERT (Distributed Error Recovery Testbed) a testbed for execution-driven simulation of error recovery techniques. The motivation behind the testbed design is given in Section 5.2. Section 5.3 describes the testbed and Section 5.4 the recovery schemes simulated on the testbed. Experiences getting applications to run on the testbed are highlighted in Section 5.5 followed by experiences simulating the recovery schemes in Section 5.6. Emphasis is placed on problems encountered and examples of the simulator's flexibility and realism. Section 5.7 discusses simulator performance. Previous work is briefly described in Section 5.8 and Section 5.9 concludes the chapter.

Our performance evaluation of application-transparent distributed error recovery techniques is the subject of Chapter 6. In Section 6.1 we first characterize the overhead incurred when *any* checkpointing/rollback scheme is used to provide fault tolerance for a scalable multicomputer system (Section 6.1.1) and then characterize and compare the overhead incurred by several different message logging (Section 6.1.2) and coordinated checkpointing techniques (Section 6.1.3). In Section 6.2 we use DERT to evaluate the performance of asynchronous message

logging and some optimizations (Section 6.2.1), and then to evaluate synchronous and asynchronous coordinated checkpointing and an ‘‘optimistic’’ independent checkpointing algorithm (Section 6.2.2). The primary focus of this evaluation is on the impact of the checkpointing algorithms on application execution time relative to the execution time of the application without any checkpointing, under error-free operation. Chapter 7 concludes the dissertation.

Chapter Two
System Model

The target system for which error recovery is required is a multicomputer consisting of hundreds or thousands of VLSI nodes which communicate via messages over point-to-point links. In the first section of this chapter we describe the computational model of the target system. This includes several basic ideas and key definitions which are used throughout the thesis and which serve as a foundation for proving the correctness of the presented recovery algorithms. In Section 2.2 the characteristics of the multicomputer are described and some assumptions are made. Section 2.3 describes the interprocess communication model used by the multicomputer. The final section details the error detection techniques that are used to detect node failures and communication failures; all message logging algorithms assume that communication is reliable while the presented coordinated checkpointing algorithms use a less restrictive model where the detection of communication failures can be incorporated into the error recovery algorithms — resulting in a potentially significant reduction in overhead.

2.1. Computational Model

In this thesis we use a combination of the distributed system models presented in [Chan85] and [Stro85].

2.1.1. Fault-Free Operation

As in [Chan85] a distributed system consists of a finite set of n processes and a finite set of channels. A process is defined by a set of states, including an initial state, and a set of events. The state of a process consists of all of the storage elements associated with than process. Processes communicate via directed channels which deliver error-free messages in FIFO order and with arbitrary finite delay. The state of a channel is determined by the sequence of messages sent along it excluding the messages received along it. An *event* is the cause of a state change and is either: 1) an internal process event, 2) the receipt of a message, or 3) the sending of a message. If a send event for a message m has occurred in some process then a corresponding receive event must eventually occur in some other process.

A *global state* of a distributed system is a set of n component process and channel states [Chan85]. The *initial global state* is one in which the state of each process is its initial state and the state of each channel is empty.

The *system history* is the set of executed events. The *depends-on*[Stro85] relation (*happens before*[Lamp78]) describes the relationship between two events a and b. An event b depends on event a $(a \rightarrow b)$ if and only if a and b are events in the same process and a occurs before b, or a is the sending of a message m by a process and b is the receipt of that message by another process. If b depends on a and c depends on b, then c depends on a (transitive closure), while if a does not depend on b and b does not depend on a, then the two events are said to be *concurrent*. A *valid* system history is a system history for which the depends-on relation is an irreflexive partial ordering[Schn85].

2.1.2. Operation with Failures and Recovery

When a failure occurs the system must be restored to a global state that is a possible intermediate state in fault-free execution. Such an intermediate state (a *consistent global state*[Chan85]) is composed entirely of *consistent* process and channel states such that every pair of processes and the connecting channels ‘‘agree’’ on the number and content of messages which have been exchanged (see Figure 2.1). States can become inconsistent in two ways:

1) If a message is *lost* (sent but not received) correctness is violated since, if a message is sent it must eventually be received. Correctness is restored by reconstructing and (re)delivering the message or by undoing the send event. and,

2) If a message is *orphaned* (received but not sent) correctness is violated because the system history is no longer valid. Validity is restored by reconstructing the send event or undoing the receive event.

Checkpointing and rollback of process and channel states can be used to provide transparent error recovery for application processes. Recording a process state is not considered to be an event, therefore checkpointing does not alter the ordering of events in the system history. When a failure occurs the global system state may contain inconsistent process and channel states due to lost or orphaned messages.

Message logging techniques use checkpoints, message logs, and ordering information (the depends-on relation) to roll back failed process(es) and then *adjust* the state of that process to be consistent with the rest of the system state. Some non-failed process states may be *orphaned* (see Figure 2.1) and must also be rolled

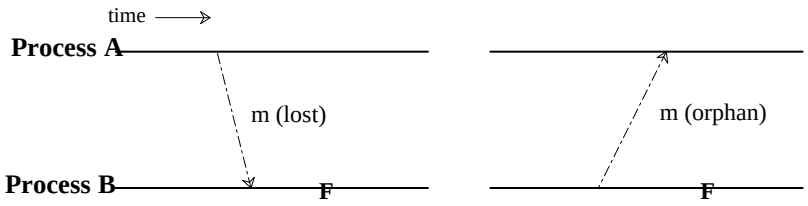


Figure 2.1: In both cases process B has failed and its state has been lost. In the first case A has sent a message for which there is no corresponding receive event (i.e. m has been *lost*). A consistent global state requires that the message not be lost (e.g. m was logged and will be received) or else that the sending of the message be undone (e.g. roll A back). In the second case A has received a message for which there has been no corresponding send event. A is an *orphaned computation*[Lisk87] (or *orphan*) that must be undone unless the send event is reconstructed.

back because inconsistent information (orphan message(s)) has been incorporated into their states.

Message logging techniques require applications to be *deterministic* in their actions: given a process state (checkpoint) and a sequence of inputs (message log), the process will generate the same outputs. When a failure occurs, the failed process(es) is rolled back and its message log is replayed. Ideally, all the messages that the process had received plus ordering information are retrievable from stable storage, in which case their subsequent replay results in the process recovering completely and identically to the point at which it previously failed. As long as duplicate outputs are detected and deleted, no other processes are affected by the rollback and subsequent replay. Synchronous message logging techniques achieve this by logging all input messages synchronously to stable storage and taking a potentially significant performance penalty during normal operation. However, if input messages are logged asynchronous of process execution, then recovery must

result in a consistent system state despite possible lost messages (not logged at the time of failure) and orphaned process states.

Coordinated checkpointing techniques maintain a *consistent global state* on stable storage. Then, when a hardware failure is detected all or part of the consistent global state is used to restore a valid system state from which computation can be resumed.

With the *global* coordinated checkpointing technique, a checkpoint session begins by flushing all messages in transit to their destinations, after which the entire system state is saved to stable storage. A recovery session simply flushes and discards any messages in transit and retrieves the saved global state.

Global checkpointing has been extended to perform checkpointing and recovery on smaller portions of the system — sets of interacting processes.

Definition 2.1:

An *interacting set of processes* is the set of processes which have communicated directly or indirectly since their last checkpoint such that,

If p is a member:

Any process q that has communicated with p since q's last checkpoint is also a member.

It follows from this definition that an interacting set of processes forms a strongly connected system graph where the processes in the interacting set are vertices and the communication(s) between them are arcs.

Thus, a distributed system is composed of a collection of *disjoint* interacting sets of processes. Since there has been no communication between processes in

different sets since their last checkpoint, a new checkpoint of a process in one interacting set is consistent with both the old and new checkpoint of another process which is in some other interacting set. Hence, different interacting sets may be checkpointed and recovered independently and a consistent global state (checkpoint) is always maintained.

These basic ideas are extended in the chapters on message logging and coordinated checkpointing techniques (Chapters 3 and 4, respectively) where we use this model to prove the correctness of our recovery algorithms.

2.2. Assumptions

The target system requiring fault tolerance is a multicomputer consisting of hundreds of nodes which communicate via messages over point-to-point links. Each node includes a processor, local memory, and a communication coprocessor. The nodes operate asynchronously and messages may have to pass through several intermediate nodes on their way to their destinations. The system is used for ‘‘general purpose’’ applications which have no hard real-time constraints. Errors can occur at any time as a result of *hardware* faults in the nodes or in the communication links.

The error recovery scheme is based on the existence of stable storage where checkpoints can be safely maintained. Such stable storage may be implemented as mirrored disks. Some of the nodes in the system, which we call *disk nodes*, are connected to such ‘‘reliable’’ disks. We assume that a failure of the disks themselves or of the disk nodes causes a *crash* (i.e., an unrecoverable error).

The connectivity of the system is high so that the probability of the system

partitioning due to the failure of a node(s) is low enough so that it is reasonable for partitioning to cause a *crash*.

The *state* of a process, which gets checkpointed periodically and recovered once an error is detected, is the contents of all of the memory and registers used by the process. This includes some system tables, such as the list of all the virtual circuits currently established to and from the process.

Since each node can be time-shared between multiple processes, it may have to participate in multiple simultaneous checkpointing and recovery sessions. Hence, it is not advisable to implement checkpointing and recovery as part of the kernel. Instead, whenever checkpointing or recovery of a particular process is initiated, the kernel spawns a special *handler* process that performs the necessary operations. The handler can suspend the process, manipulate its state, and allow it to resume normal operation. In the rest of the thesis we will often discuss the actions of participants in checkpointing and recovery sessions. These ‘‘participants’’ are really the handlers corresponding to the processes being checkpointed or recovered.

2.3. Interprocess Communication

A fault-tolerant multicomputer must be able to detect and recover from errors in interprocessor communication. The message delivery mechanism has a strong effect on the performance of checkpointing and recovery schemes. Many error recovery schemes [Stro85, Koo87] are based on the use of reliable FIFO communication channels between nodes. These channels are implemented by underlying communication protocols which typically involve sending redundant

bits for error detection, appending sequence numbers to each message, and the use of end-to-end acknowledgments [Tane81]. The message logging algorithm described in this thesis (Chapter 3) is based on this kind of reliable protocol.

For the coordinated checkpointing algorithms presented in this thesis (Chapter 4) two communication mechanisms (containing no facilities for error detection) are considered: virtual circuits and simple packet switching. In a system where processes communicate using *virtual channels* (or virtual circuits) [Bert87, Reed87], i.e., if processes on two nodes that are not immediate neighbors need to communicate, a logical circuit is set up from the source to the destination by placing appropriate entries in the routing tables of each intermediate node along the way. Once the path is set up, there is very little routing overhead for packets sent through the circuit and FIFO ordering of these messages is maintained. With message/packet switching no path is established in advance between the sender and the receiver. Every packet is routed independently at each hop and packets (messages) may arrive at their destination out of order. We assume that the operating system will reorder packets as necessary.

2.4. Error Detection

As previously mentioned, errors in the system may be a result of node failures or failures in the communication links. We assume that the nodes are self-checking and produce an error indication whenever their outputs are incorrect[Tami83].

In most systems, errors in message transmission are detected by including with each message *check bits*, which the receiver uses to determine whether the

contents of a message has been corrupted. Lost messages are detected by protocols that involve acknowledging each message as well as transmission of sequence numbers with each message[Tane81]. The disadvantage of these techniques is that they involve transmission of redundant bits and thus ‘‘waste’’ communication bandwidth. Since the probability of an error in transmission is low, it is wasteful to check the validity of each message or packet independently. For the coordinated checkpointing algorithms the following technique is used.

As proposed in[Tami84], each node has two special purpose registers for error detection associated with each of its ports. One of these registers contains the CRC (Cyclic Redundancy Check) check bits for all the packets that have been sent from the port. The other register contains the CRC check bits for all packets received. These special purpose registers are linear feedback shift registers (LFSRs) and their contents are updated in parallel with the transmission of each packet[Elki82].

In order to check the validity of all the packets transmitted through a particular link, each node sends to its neighbor the contents of the LFSR used for outgoing packets. The neighbor can then compare the value it receives with the value in its LFSR for *incoming* packets and signal an error if it finds a mismatch. In this scheme, if packet switching is used, *all* the links in the system must be checked in this way before committing to a new checkpoint. Otherwise, the state of a node corrupted by an erroneous message may be checkpointed and later used for recovery.

Alternatively, *virtual LFSRs* can be maintained. With virtual circuits, a virtual LFSR is used to accumulate the signature of the packets transmitted from the source process of a virtual circuit and another virtual LFSR is used to

accumulate the signature of packets received by the destination process of that virtual circuit. Communication between processes in the interacting set can then be checked without checking all the links in the system, by performing ‘‘end-to-end’’ checks on all the virtual circuits between processes in that set. Each node might, for example, maintain a cache of virtual LFSRs by those circuits which have either a source or destination process at that node. Hence, the LFSRs are not updated by packets being transmitted *through* a node but only for those whose source or destination is the current node. If packet switching is used the same technique can be employed despite the fact that messages may take ‘‘any’’ path to get to their destination.

The packets used to coordinate the creation of checkpoints and for error recovery must be verified before they are used. Hence, for these packets, an error detecting code is used and redundant bits are transmitted with the packet. Thus, there are two types of packets in the system: normal packets that do not include any information for error detection, and special control packets, called *fail-safe* packets, that are used only for transmitting information between handlers and which include a sufficient number of redundant bits to detect likely errors in transmission. The *fail-safe* packets are either error-free or the error is easily detectable by the receiving node.

When a node fails, recovery involves rolling back the union of the interacting sets of all the processes that were running on that node and requires eventual roll back of all interacting sets of processes which had any messages in transit on that node at the time of its failure. When an error is detected by a mismatch of the LFSRs on two ends of a physical link and *packet switching* is used, the entire

system must be rolled back since there is no way to determine which processes were affected by corrupt messages. If virtual circuits are used, and an error is detected by a mismatch of signatures on the two ends of a virtual circuit then only the interacting set which includes the two processes connected by that virtual circuit need to be rolled back. See Chapter 4, Section 4.4 for more details.

Chapter Three
Message Logging Techniques

Message logging techniques checkpoint process states and (log) interprocess messages onto stable storage. When a process fails and is rolled back, its *message log* is played back to it, so that when the message log has been depleted, the process is in a state consistent with the non-failed processes in the system. Message logging techniques require application processes to be *deterministic*: given a process state and a sequence of inputs (message log), the process will generate the same outputs. Processes can be checkpointed independently, thus minimizing disruptions to the entire system, and a single failed process can be recovered, potentially without interfering with the operation of other processes.

Many papers have been published which use the *message logging* technique as the basis for application-transparent distributed error recovery algorithms [Borg83, Powe83, Stro85, John87, Stro88, John90a, Sist89, Jalo89, Elno92a]. In this chapter a new asynchronous (optimistic) message logging algorithm based on [Stro85] is presented. The computational model described in the previous chapter provides the foundation for a complete proof of correctness of the presented algorithm. This model is also used to thoroughly explain the technical details of asynchronous message logging and to make corrections to previously published algorithms. The new algorithm uses both stable *and* volatile message logs to recover more of the failed process(es)' lost computation. The use of volatile message logs also reduces the number of non-failed processes (orphans) that are forced to roll back in order to recover a consistent system state.

Previous work on message logging algorithms is discussed in Section 3.1. Section 3.2 lists data structures required by asynchronous message logging. Section 3.3 illustrates the basic approach of asynchronous message logging and summarizes the major ideas presented in [Stro85]. In Section 3.4 we show how the presented correctness model is applied to asynchronous message logging. The new algorithm is presented in Section 3.5. The algorithm is discussed and its correctness proven and summarized in Section 3.6. Some relevant details are discussed in Section 3.7. Section 3.8 concludes this chapter. The performance analysis of the algorithm, and some optimizations, using a scalable, execution-driven simulator, running real parallel applications and assuming a failure-free system can be found in Chapter 6.

3.1. Previous Work

Previous work on distributed message logging algorithms can be divided into pessimistic (atomic [Borg83, Powe83], synchronous [John87, Jalo89]) and optimistic (asynchronous [Stro85, Stro88, Sist89]) protocols. In *pessimistic* message logging, the computation of an application and the logging of its messages to stable storage are synchronized. Algorithms relying on *atomic* logging use hardware (e.g. single bus) or software (e.g. reliable multicast) to ensure that both the receiver and stable storage receive the message. *Synchronous* algorithms do not require atomicity but block the sending process from sending any further messages until it has been confirmed that the message sent has been received and logged. *Optimistic* message logging algorithms asynchronously log messages to stable storage without delaying process execution. The principal advantage of

pessimistic schemes is a quicker recovery due to the avoidance of *orphans*. However, any synchronization can significantly degrade the performance of a distributed application.

The synchronous algorithm presented in [John87] logs messages in the volatile memory of the sender. Synchronization on every message is required. When a process sends some message *m* it is not allowed to send any further messages until the corresponding acknowledgment, containing sequencing information, has been received and the sequencing information has been logged (in volatile memory). Due to the use of *volatile message logs* this algorithm can only tolerate the failure of a single process.

The ‘‘pseudo’’-synchronous algorithm in [Jalo89] takes a message checkpoint (some time) *prior* to each send. This message checkpoint contains a copy of all messages received after the previous send and is sent to a backup process on another node in the system. If message delays can be arbitrarily long then a process may have to wait an arbitrarily long time before it can send its next message. In the worst case a receive always occurs just before a send and the full synchronization penalty is incurred. In the best case the ‘‘last’’ receive before the next send occurs far enough ahead of that send that no synchronization penalty is incurred. Thus, ‘‘minimizing’’ synchronization penalties requires that one either correctly guess which receive is the *last* receive or that one take a message checkpoint after *every* receive.

The asynchronous message logging algorithm presented in this thesis is based on the algorithm presented in[Stro85] but is extended to use volatile message logs in addition to stable logs in order to recover as much of the failed process(es)’ pre-

failure execution as possible. As discussed in Sections 3.3 and 3.6.3 there is a critical flaw (omission) in their presented algorithm.

In [John87] volatile logs are used to hold *all* messages since the target machine is assumed to be a collection of diskless workstations. As previously discussed this (exclusive) use of volatile message logs only tolerates the failure of a single process. In [Stro88] volatile logs are used to reduce the amount of messages that must be logged to stable storage. Messages are stored in volatile memory until they are removed or ‘‘spooled’’ to disk. The additional use of volatile logs in this way is orthogonal to the issues discussed in this chapter. Additionally, neither a complete algorithm nor a complete proof was included in [Stro88].

In [John87] the pessimistic algorithm was also extended to be an optimistic algorithm by requiring the sending of dependency information on each message[Stro85]. To be able to keep a single checkpoint per process an orphan-detection algorithm must be run prior to taking a new checkpoint to ensure that the old checkpoint is no longer needed[John87]. This implies that a new checkpoint would be taken only if no orphans were detected. In [John90a] this optimistic algorithm was further extended to keep multiple checkpoints per process. Although a complete optimistic algorithm was not provided in either paper, it appears that this extended algorithm is identical to Strom and Yemini’s optimistic algorithm except that messages are stored in the sender’s volatile memory instead of on stable storage.

In [Sist89] correctness proofs were given for a simplified asynchronous message logging algorithm which requires the rollback of all processes in the system, assumes the loss of all volatile memory, and assumes that at most one

failure occurs. This simplified algorithm does not exhibit the principal advantages of message logging which, in general, requires the checkpointing and rollback of only one process at a time. A second asynchronous algorithm is presented and proven in [Sist89] where the recovery procedure is *always* able to recover the maximal (most recent) global state, but still requires the rollback of the entire system. Unlike other message logging algorithms which require the transmission of $O(n)$ information with every message, this algorithm only requires $O(1)$ information. However, a process must have the ability to take a checkpoint of a *past* state in order to ensure that the maximal (consistent) global state exists on stable storage. An implementation of such a checkpointing scheme requires, for example, 1) the calculation of the consistent state to be checkpointed (requiring $O(n^3)$ messages [Sist89]), 2) the restoration of an older checkpoint, 3) the replay of the stable message log until the chosen state is reached, and 4) the checkpointing of that chosen state. This requires significant computing resources and is more complex and time-consuming than simply saving the *present* process state to stable storage.

A new message logging algorithm, called Manetho[Elno92a], addresses applications that require periodic, and timely, commitment of messages to the external world. In this algorithm dependency tracking involves the (more costly) manipulation and transmission of antecedence (dependency) graphs, instead of vectors, but an external (output) message can be committed after a single synchronous disk operation.

3.2. Data Structures for Asynchronous Message Logging

The following data structures are required to support asynchronous message logging and are maintained for each application process[Stro85]:

- *Incarnation Number*, Inc_s — incremented each time a process s is rolled back.
- *Send Sequence Number*, $SSN_s[r]$ — incremented when a message is sent from process s to some process r.
- *Expected Incarnation Number*, $eInc_r[s]$ — the incarnation number that a process r expects to receive from s.
- *Expected Send Sequence Number*, $eSSN_r[s]$ — the send sequence number that a process r expects to receive from s. This number is incremented after the arrival and acceptance of a message with the expected $SSN_s[r]$.

  Inc_s, $SSN_s[r]$, $eInc_r[s]$, and $eSSN_r[s]$ are used to ensure FIFO ordering of messages across process incarnations.
- *Receive Sequence Number*, RSN_r — incremented and then assigned to a new incoming message.

  The start of a new *state interval* in a process is indicated by the receipt of a message. A state interval is identified by the following triple: (r, Inc_r, RSN_r). RSN_r combined with Inc_r is used as an ‘‘ordering filter’’ that forces determinism upon the nondeterministic arrival of messages. *RSN order* refers to the lexicographical ordering of (Incarnation,RSN) pairs.
- *Dependency Vector*, DV_s — each entry $DV_s[n]$ contains an incarnation

number and RSN ($DV_s[n].Inc$, $DV_s[n].RSN$) which indicates the lexicographically latest state interval of each process n that s's current state interval *depends on* (as defined in Section 2.1), if any.

- *Incarnation Start Table*, IST_s — contains an entry, $IST_s(Inc_n, n) = firstRSN$ for each incarnation of process n as long as any trace of that incarnation remains in the system. Each entry indicates the RSN of the first message that will be (or has been) received from that incarnation.

  This table is used to determine if a process' current state is orphaned and if incoming messages are orphan messages.

- *Log Vector*, LV_s — each entry $LV_s[n]$ contains an incarnation number and RSN ($LV_s[n].Inc$, $LV_s[n].RSN$) which indicates how far process s has been informed that process n has logged.

  Every process periodically broadcasts its log vector entry which indicates the last message it has logged to stable storage.

3.3. Basic Approach of Message Logging

In [Stro85] Strom and Yemini proposed an *optimistic* message logging technique for non-shared-bus architectures where logging proceeds asynchronous of process computation. The recovery of a process' current state depends on its earliest checkpoint and on the disk's log of the messages which the process has received since that checkpoint. When a process fails some messages might not yet be logged to disk. To achieve a consistent system state additional processes (*orphans*) may have to be rolled back. Thus this approach is termed ''optimistic''

because it does not incur the overhead of synchronization during normal operation in order to avoid orphans.

In order to maintain correctness despite the loss of recovery information (messages) sending a message involves the following steps:

1) A message M from sender S to receiver R is sent along with S's current *dependency vector* and $SSN_s[r]$. S saves a copy of M until R notifies S that the message has been logged (step 4).

2) If the SSN of M matches the *expected* SSN ($eSSN_r[s]$ and $eInc_r[s]$) maintained by R, *and* the message is determined not to be an *orphan* then R updates its dependency vector with the dependency vector sent with M since R's state now depends on the state interval in which S sent M.

3) R sends a copy of the message to disk.

4) R ''eventually'' receives notice that the message has been logged and then ''eventually'' notifies S that the message has been logged.

The state of each process q is periodically checkpointed. A checkpoint C_q^i is reclaimed (discarded) when every message upon which the checkpointed state C_q^{i+1} depends has been logged to stable storage *and* when all messages that were sent by q prior to C_q^{i+1} have been logged to stable storage. When a checkpoint C_q^i is reclaimed the log of all messages received prior to and including the checkpoint state C_q^{i+1} can be discarded. In this chapter, *earliest checkpoint* refers to the earliest checkpoint that has not yet been reclaimed.

Messages can be committed to the outside world when *every* message upon

which they depend has been logged. This is determined by comparing the local log vector to the message's dependency vector.

When a process q fails the earliest checkpoint of that failed process is restored. The restored process then plays back its stable log until an orphan or the end of the log is reached. If the last message replayed from the log has RSN order (Inc_q, X), then a new incarnation entry, $IST(Inc_q+1, q) = X+1$, is added to IST_q and is broadcast to the rest of the system in a *Recovery* message. This informs the other processes that any state intervals greater than (Inc_q, X) and less than $(Inc_q+1, X+1)$ are lost. A process p receiving a *Recovery* message updates its IST and determines if it is an orphan by examining its dependency vector and IST to see if its current state depends on any unrecoverable state interval.

This summarizes the main ideas presented in[Stro85]. Although this algorithm detects orphans (processes and messages) and performs *state backout*, it does not always recover/replay all the messages necessary for a correct recovery. The basic flaw is that they do not properly deal with non-orphan messages which are not replayed during the ordered replay phase. Specifically, they do not *guarantee* that such messages will be delivered to the receiver following recovery and this can lead to a lost message inconsistency (the sender state indicates that the message has been sent while the intended receiver never receives the message).

Consider the following example (see Figure 3.1). A process Q fails and is rolled back. There are two messages in Q's stable log but the first one is determined to be an orphan (it depends on a state interval that is known to be lost). The second message is not explicitly replayed from the stable log. When P receives a *Recovery* message from Q, P determines that it is not an orphan (does

not depend on any state intervals known to be lost) and that it has no messages in volatile memory for *Q* (the message had been reclaimed when it was successfully logged to stable storage — steps 1) and 4) above). This is an inconsistent state due to a lost message.

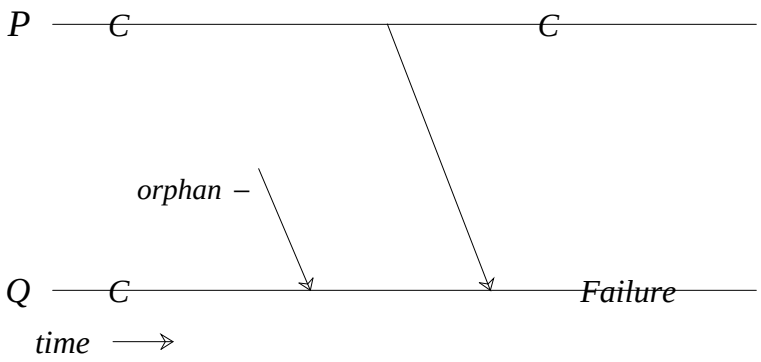


Figure 3.1: An example where a message needed for correct recovery is not retrieved.

It should be noted that this flaw causes inconsistencies only for those sessions where the receiver is the recovering process and where the sender does not send any additional messages to the receiver following recovery. If the sender does send a message to the receiver following recovery, the sender and receiver half-session protocols discussed in[Stro85] will (eventually) correctly deliver the messages to the receiver. Alternatives for solving this problem are briefly discussed in Section 3.6.3. This problem was also overlooked in[Sist89].

3.4. Recovering a Consistent Global State

As described in Section 2.1, a consistent global state is composed entirely of consistent process and channel states such that every pair of processes and the connecting channels ‘‘agree’’ on the number and content of messages which have been exchanged. Ordering information (the depends-on relation) is used to determine if two process states are inconsistent (due to lost or orphan messages) as per Figure 2.1. Since the checkpointing of processes does not affect the ordering of events in the system history, only recovery needs to be examined to show correctness.

Recovery consists of:

a) Restoring the earliest (unreclaimed) checkpoint of the failed process(es), and

b) *Reenacting* the failed process’ previous (pre-failure) behavior as thoroughly as possible. Ideally, enough information exists to reenact the process’ pre-failure execution up to the point at which it had previously failed. If so, no other processes are adversely affected by the failure since no messages have been lost or orphaned.

   If some recovery information has been lost then recovery must also include,

c) *Purging* the effects of any behavior that could not be successfully reenacted due to incomplete recovery information.

With message logging schemes there are no *lost* messages. A non-orphan input message can *always* be retrieved from either — the stable message log, the volatile message log of a non-failed sender, or from the replay of a failed sender

(case 1 in Figure 2.1) [Stro88]. An input message is an *orphan* if there exists a process in which a receive event has occurred for that message and no corresponding send event (exists or) can be reconstructed in some other process (case 2 in Figure 2.1). Orphaned (inconsistent) processes occur due to *incomplete recovery information* which manifests itself through:

a) *Unordered messages:* any message that does not have receive sequence information. This includes any message that has been (or will be) sent that 1) has not been logged to stable storage, *and* 2) has not had its receive sequence information logged in the sender's volatile memory. A message that is logged to stable storage can *become* unordered if, during replay, a logged message that precedes it in RSN order is determined to be an orphan (see b) below). *Ordered messages* are messages that have been successfully logged to stable storage *or* have been logged in the (non-failed) sender's volatile memory along with receive sequence information.

During recovery unordered messages occur when either 1) the sender and receiver fail and the message is not logged, 2) the message was never received in the pre-failure execution, or 3) the message was received but the sender didn't receive ordering information from the receiver. (Note: new messages sent in the post-failure execution which were not sent in the pre-failure execution fall under the definition of unordered messages). Since the ordering information of these messages has been ‘‘lost’’ (or never established) it can not be guaranteed that the post-failure execution (recovery) will continue to be identical to the pre-failure execution if these messages are replayed. In other words, the inputs may be processed in a different order and subsequent send events can not be guaranteed to

be reconstructed.

b) *Orphan messages:* a message that depends on some message that is 1) unordered (lost) as the result of a failure(s) or 2) an orphan message. The orphan message itself might be retrievable but since it transitively depends on some message whose ordering information could not be determined it can not be guaranteed that an identical message (or any message at all) will be resent. A process that has received an orphan message is an *orphan process*.

Logging the receive order at the sender *may* result in fewer unordered messages than if the receive order is not sent to the sender as in[Stro85]. This is especially the case if the sending of received messages with their RSNs to stable storage can be delayed at the receiver and lost when the receiver fails. This may be the case if the logging of messages to stable storage is a low-priority task or if messages are blocked together to be logged as one unit. Fewer unordered messages in turn result in fewer orphan messages and orphan processes.

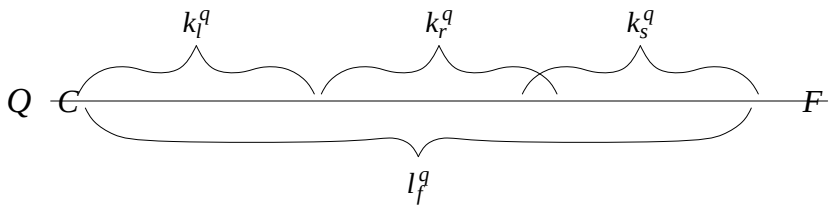


Figure 3.2: $k_f^q=\{k_l^q,k_r^q,k_s^q\}$.

At the time of failure of a process q, k_f^q are the messages that have been *sent to* the failed process q since its earliest checkpoint and l_f^q messages have been *sent from* q.

Messages in k_f^q can be broken down into disjoint groups (see Figure 3.2):

a) *Logged* messages (k_l^q): these messages have been received, assigned an

RSN, and logged to stable storage. As such, it is *guaranteed* that it is possible to obtain copies of the messages and their (pre-failure) RSNs.

b) *Received but not logged* messages (k_r^q): these messages have been received and it *might* be possible to obtain copies of the messages with their (pre-failure) RSNs from somewhere, but there is no guarantee that this is possible (they have not been logged to stable storage).

c) *Sent but not received* messages (k_s^q): these messages have been sent but either no RSNs have been assigned to them yet or, the RSN cannot be obtained from anywhere.

Note: any or all of these groups may be empty.

Once a message is received, an RSN is sent back to the sender. Once a message is sent to receiver q, it becomes a *potential* member (if q fails) of k_f^q and k_s^q. When the sender receives the RSN, if the stable log has not yet received the message with its RSN, the message becomes a potential member of k_r^q. When a copy of the message with its RSN arrives at the stable log, the message becomes a potential member of k_l^q.

If there are several senders, the order in which RSNs are received by the senders may be different from the order in which messages arrive at the receiver (due to concurrency). Hence it is possible for messages to become members of k_r^q while some messages with *lower* RSNs are still in k_s^q. Thus, there may be ‘‘gaps’’ in the RSNs of messages in k_r^q, forcing the end of the deterministic replay phase of recovery.

According to the correctness model presented in Section 2.1, for each

message $m \in k_f^q$ (that was sent at, prior, or even after, the failure of q) either:

1) m is not an orphan and its receipt must be reflected in the post-failure execution of q, or

2) m is an orphan and its sending must be undone (i.e. the sender must be rolled back to some state interval prior to the one in which the message was sent).

Also, messages must be received in the order they were sent, within each pair of processes in spite of the failure (to satisfy FIFO channel assumption).

For each message $m \in l_f^q$ (that was sent by q prior to its failure) either:

1) the message must be regenerated during q's replay. This is guaranteed by the deterministic assumption *as long as* q's inputs are identical to q's pre-failure behavior. These duplicates must not be processed a second time (i.e. must be detected and deleted). or,

2) each message that is not regenerated during q's replay must not be received. This occurs when the input stream diverges from the pre-failure behavior due to an unordered or orphan message. The remaining messages in l_f^q are *orphan messages* and their receipt must be undone.

In other words, every pair of processes must agree which messages have been sent and which have not (such that no messages are lost or duplicated) — thus resulting in a consistent post-failure global state. Ordering information (the depends-on relation) is used to determine if the process states are consistent.

3.5. Algorithms

There are three main differences between the algorithm described in[Stro85] and the new algorithm provided in this chapter. The differences are as follows:

1. As in [Stro85] receive sequence information is logged to stable storage with the message. Unlike[Stro85] this information is also sent back to the sender to be logged in its volatile memory.
2. During recovery, the *entire* stable log is replayed, except for any orphans that are detected. The stable log is then followed by messages logged at the senders and new messages. (In [Stro85] the stable log is explicitly replayed only up to the first orphan that is encountered, as discussed in Section 3.3.)
3. Unlike in[Stro85] steps are included to ensure that the incarnation start table (IST) is robustly maintained.

The first difference is an optimization to reduce the number of orphans (additional processes that will have to roll back). The second is a correction to previous asynchronous algorithms and the third has not been previously specified, but must be included in any asynchronous message logging algorithm.

3.5.1. Normal Operation and Checkpointing

Much of this part of the algorithm is simply reworded from[Stro85], using the data structures defined in Section 3.2. The major change is the sending of the receive sequence information back to the sender. Several minor modifications have been made to either clarify or expand the original description.

The Message Delivery Protocol:

1) A message M from sender S to receiver R is sent along with S's current *dependency vector* and $SSN_s[r]$. S saves a copy of M until R notifies S that the message has been logged (step 4).

2) At receiver R the message must pass through two filters: an *SSN protocol* filter to detect duplicate messages and an *orphan detection* filter to detect orphan messages. (These two filters are provided in the next subsection). If the message successfully passes through the filters then it can be delivered to the application. R updates its own dependency vector with the dependency vector sent along with the message since R's state now depends on the state interval in which S sent M. This is done as follows:

$$DV_r[r] \leftarrow (DV_r[r].Inc, DV_r[r].RSN+1), \text{ and}$$

$$DV_r[s] \leftarrow max((DV_r[s].Inc, DV_r[s].RSN),$$
$$(MsgDV[s].Inc, MsgDV[s].RSN)), \text{ for } s \neq r$$

where *max* is the lexicographically largest (incarnation,RSN) pair.

3) R sends the message to disk along with $SSN_s[r]$ and the message's dependency vector (including the receive order in R's $DV_r[r]$ entry and the sender's incarnation number in $DV_r[s]$). R also sends its $DV_r[r]$ entry back to S, so that the sender will temporarily have the message's ordering information.

4) R ''eventually'' receives notice that the message has been logged and then ''eventually'' notifies S that the message has been logged.

Each process q is periodically checkpointed along with its DV_q, SSN_q, $eSSN_q$, and $eInc_q$. Checkpoint C_q^i can be reclaimed when it meets both of the following conditions:

1) No longer needed for state backout:

   if $(LV_q[n].Inc, LV_q[n].RSN) >= (C_q^{i+1}.DV[n].RSN,\ C_q^{i+1}.DV[n].RSN)$ for all processes n.

2) No longer needed for for message recovery:

   all messages sent by q between C_q^i and C_q^{i+1} have been logged in stable storage by the receivers.

3.5.2. Recovery

The recovery of a failed process q consists of the following steps:

1. The earliest checkpoint of q is restored. This checkpoint includes DV_q, SSN_q, $eSSN_q$, $eInc_q$. The current IST_q is also retrieved from stable storage.
2. All ordered messages $m \in k_l^q$ and $m \in k_r^q$ are replayed in *RSNorder*. Playing back k_r^q in *RSNorder* is trivial since messages are logged in *RSNorder*. Playing back k_r^q in *RSNorder* involves the merging of logs obtained from all of the senders. This *ordered playback* ends when:
   a) an orphan message is discovered, or
   b) all messages in k_l^q (and k_r^q) have been replayed. or,
   c) a *gap* in the k_r^q messages occurs: caused by either 1) the ‘‘simultaneous’’ failure of a sender(s) 2) or by the overlap of the k_r^q and k_s^q groups.
3. At this point, since q’s input stream may diverge from its pre-failure execution q:
   a) begins a new incarnation ($DV_q[q].Inc \leftarrow DV_q[q].Inc + 1$),
   b) logs this fact in q’s IST_q ($IST_q(DV_q[q].Inc, q) = DV_q[q].RSN + 1$) on

stable storage and marks this new entry as ‘‘unconfirmed’’.

c) broadcasts *Recovery* messages, containing this new *IST* entry, along with any of q’s *other IST* entries which are (still) marked ‘‘unconfirmed’’, to all other processes p.

4. All remaining messages in k_f^q (which are all unordered messages) are then replayed and (re)logged.

5. Eventually q will receive acknowledgments from all processes p that they have received and logged q’s new entry. q then marks this entry in its IST_q on stable storage as ‘‘confirmed’’.

As discussed above, the second step includes an optimization to previous published techniques and may result in fewer orphan processes. The fourth step corrects the problem discussed in Section 3.3. The robust maintenance of the *IST* is also essential to restoring a valid system state — if the *IST* is missing any entries then a process may not determine that it is an orphan. This issue has not previously been described (see Section 3.7 for more discussion).

When a *Recovery* message is received by p, p:

1. Updates IST_p with the new entry from q. If any of p’s entries in IST_p are marked ‘‘unconfirmed’’ then these are resent to q.

2. Checks to see if its *current* state interval depends on some orphan message:
$\exists\, n \mid (Inc_n > DV_p[n].Inc) \wedge (IST(Inc_n, n) \leq DV_p[n].RSN)$, where $p \neq n$
This check must be performed for *each* one of n’s entries (Inc_n) in the IST. If the process p is determined to be an orphan process then it must be rolled back to its earliest checkpoint.

3. Sends an acknowledgment to q which indicates that p has received the *Recovery* message and has logged q's *IST* entry to stable storage in IST_p.

Every message received by q (and sent by p) is put through the following two filters as mentioned in the previous subsection.

SSN Protocol filter: checks for duplicate messages (re)generated by rolled back processes

a) $SSN_p[q] = eSSN_q[p]$ and $Inc_p = eInc_q[p]$

The message is the next expected message and is passed to the orphan detection filter.

b) $SSN_p[q] < eSSN_q[p]$

There are two possible cases:

1) $Inc_p \leq eInc_q[p]$

The message is a regenerated one and is deleted. or,

2) $Inc_p > eInc_q[p]$

The message is the first message of the new incarnation to be received by q. If it passes the orphan detection filter then q updates both $eInc_q[p]$ and $eSSN_q[p]$.

c) $SSN_p[q] > eSSN_q[p]$

The receiver p has failed and the message is part of q's volatile log k_s^q. The message is held for subsequent replay when $SSN_p[q] = eSSN_q[p]$ (case a).

Orphan detection filter: The message is an orphan and is deleted if the following is true:

$\exists\, p \mid (MsgDV[p].Inc < Inc_p) \wedge (IST(Inc_p, p) \leq MsgDV[p].RSN)$, where

$p \neq q$

This check must be performed for *each* one of p's entries (Inc_p) in the IST.

3.6. Proofs of Correctness

The detection of an orphan or unordered message during replay (step 2 of the recovery algorithm) indicates that complete recovery can not be achieved and further execution *may* diverge from the pre-failure execution of the failed process. All replay *prior* to the first orphan/unordered message has been identical to the pre-failure execution (since the application is assumed to be deterministic) and, hence, any messages sent so far are identical to and compose a prefix of the sequence of messages in l_f^q (See Figure 3.3). If all regenerated messages are either 1) deleted if previously received or 2) received if previously *not* received (due to the failure(s)) then reexecution *up to* the first orphan/unordered message is *correct*. All messages sent have been received with no duplicates or omissions.

All remaining messages in l_f^q *depend on* the orphan/unordered message and are now orphans. Being orphans, these messages have not been (re)sent in the post-failure execution and hence must not be received. Every process that depends on such a message must be rolled back to a state that does not depend on 1) a message that has been unordered as a result of the failure(s) or 2) an orphan message.

To prove the correctness of the above recovery algorithm the next three subsections show, in detail:

1) that regenerated messages (in l_f^q) are detected and deleted,

2) that orphans (in l_f^q) are detected by the receiving processes and their receipt is

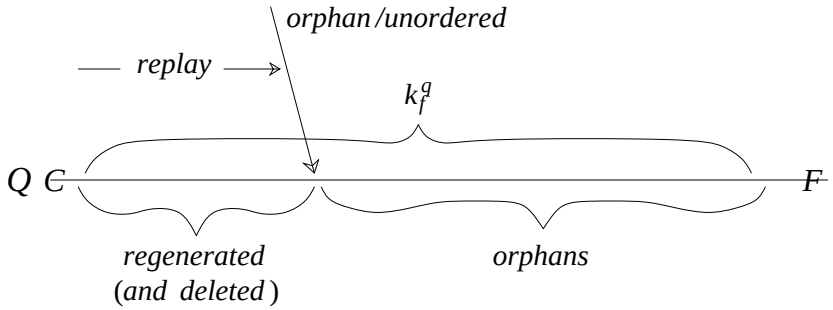


Figure 3.3: Recovery consists of identical replay of the pre-failure execution of failed process Q up to the first detected orphan/unordered message. Since the post-failure execution of Q may now diverge from its pre-failure execution, a new incarnation is begun and all outputs of Q ($m \in l_f^q$) which have not been regenerated during recovery are orphans.

undone — purging the system of all state intervals that depend on any state intervals (of each failed process Q) that could not be successfully reenacted, and

3) that the remaining messages in k_f^q (those that have not yet been replayed) are correctly handled — all non-orphan messages in k_f^q are received and all orphan messages are not sent (or received).

3.6.1. Deleting Regenerated Messages

Theorem 3.1: The SSN protocol filter is sufficient to correctly delete all regenerated messages and accept only non-regenerated messages.

Proof —

a) Assume a regenerated message m sent by p to q is not deleted but is instead accepted.

A regenerated message is, by definition of recovery, sent prior to a *Recovery*

message. In order for the regenerated message to be accepted the $(Inc_p, SSN_p[q])$ pair that arrived with it must have been (lexicographically) $\geq (eInc_q[p], eSSN_q[p])$. Also, for the message to be *erroneously* accepted q must not have failed (see below) and the message must have been previously received by q. Since p's post-failure behavior prior to sending a *Recovery* message is identical to its pre-failure behavior, then it must have sent the identical $Inc_p, SSN_p[q]$ on the pre-failure version of the message as it has just sent on the regenerated message. In order for q to accept the pre-failure message the following had to be true: $eSSN_q[p] = SSN_p[q]$. Since $eSSN_q[p]$ would have been subsequently incremented, and q has not failed, it is not less than the $SSN_p[q]$ of the regenerated message. (If q *had* failed and rolls back to a state *after* having received the original message, then q's restored $eSSN_q[p]$ must be $> SSN_p[q]$. If q had failed and rolled back to a state *prior* to receiving the message, then accepting the message would not be erroneous).

b) Assume a non-regenerated message m sent by p to q is deleted instead of accepted.

In order for the message to be deleted the following must be true: $(Inc_p, SSN_p[q]) < (eInc_{q[p]}, eSSN_q[p])$. Since non-regenerated messages are sent after *Recovery* messages (and message delivery is FIFO), all such messages must have the latest, and largest, Inc_p. As a result they are all lexicographically larger than $(eInc_{q[p]}, eSSN_q[p])$ since $Inc_p > eInc_q[p]$. The first such message updates the eSSN and eInc values maintained by q. If any subsequent message has a lexicographically smaller send sequence pair then it must be regenerated (case a).

$\therefore$ since regenerated messages can not be accepted instead of deleted and non-

regenerated messages can not be deleted, the above algorithm correctly handles all messages. □

In other words, the *SSN* protocol filter preserves the correct ordering of messages across incarnations.

It should be pointed out that when a duplicate message is detected *RSN info* is *not* sent back to the (rolled back) sender (step 3 of the Message Delivery Protocol). This is not important as far as correctness is concerned, since this simply results in an ‘‘extra’’ unordered message in the event of the failure of the receiver prior to the logging of the original (not duplicate) message to stable storage.

3.6.2. Handling Orphan Messages

When a process receives a message it checks whether or not the message is an orphan prior to processing it (e.g. see Lemma 3.1 and Figure 3.4). If the message *is* an orphan it is deleted since, by definition, it depends on some state interval that is no longer part of the system history. The dependency vector transmitted with the message is a recording of the *depends on* relation (Lemma 3.2). If the receiving process knows which state intervals are part of the system history and which are not, for all possible sending processes S, then the receiving process can determine if the incoming message is an orphan.

Lemma 3.1: Any message $m \in k_f^q$ may be an orphan.

Proof —

A way for a message to become an orphan is when, following recovery involving the message sender, the state interval during which the message was sent is no longer part of the live history[Stro85] of the system. This has nothing to do with

whether the message is unlogged (k_s), logged at the sender (k_r), or logged on stable storage (k_l). $\therefore$ any message $m \in k_f^q$ may be an orphan. □

For example, in Figure 3.4 the ordering information for the first message received by P has not been logged at the sender or on stable storage when P fails. Due to concurrency both R and Q may have successfully logged their input messages to stable storage. The messages received by R and Q are orphans because they depend on the first message received by P.

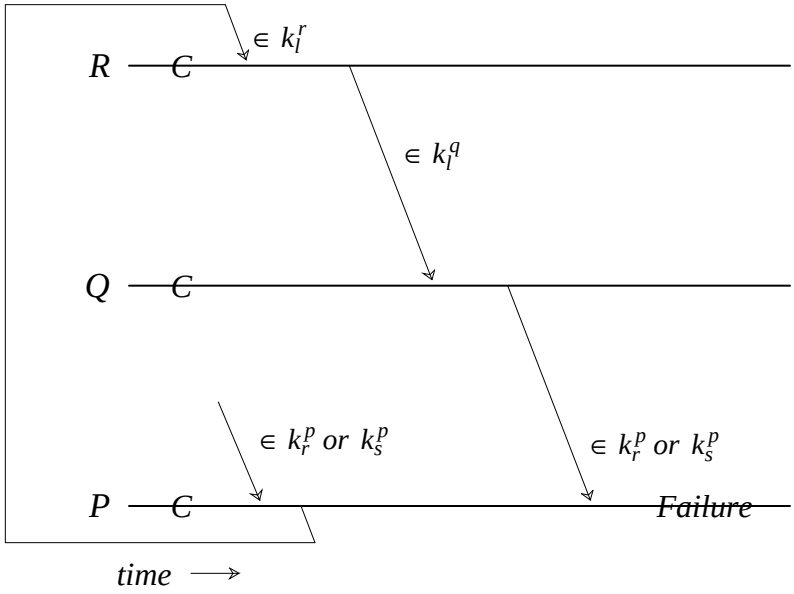


Figure 3.4: The first message received by P can be an unordered message. Hence the messages received by R and Q can be orphan messages.

Lemma 3.2: It is sufficient for a process q to know (record) only the lexicographically latest state interval of each process p upon which it depends.

Proof —

This would be insufficient only if the following were possible: Process q can depend on a state interval of some process p that is in the system history *and* depend on some earlier interval of the same process that is *not* in the system history. This is trivially disproved since a state interval of any process p depends on all previous state intervals of that process (definition of *depends on*).

$\therefore$ the dependency vector correctly and completely records the *depends on* relation. □

Each IST_q records the partial ordering of events for a process q. The system history is defined by the *IST*s of all processes in the system. The following example illustrates how the system history is recorded. Suppose the following messages (inputs) to process q reside on stable storage — listed in (Inc_q,RSN) pairs: (0,0)(0,1)..(0,6),(1,4)..(1,8),(2,9)..(2,13),(3,7)..(3,10). Process q has the following IST entries: $(0,q) = 0$; $(1,q) = 4$; $(2,q) = 9$; $(3,q) = 7$. This means that q received messages one through six in incarnation zero (any subsequent messages in incarnation zero were never logged). Upon rollback message (0,4) was determined to be an orphan and a new incarnation was begun during which messages (1,4)...(1,8) were logged before another rollback was required, and so on. (3,10) was the last message to be logged. The events q contributes to the system history are thus (0,0)..(0,3)(1,4)..(1,6)(3,7)..(3,10). All other events (messages) are ‘‘lost’’ (orphaned) and any state interval that *depends on* one or more of them is an orphan.

This dependency exists due to orphan messages which can manifest themselves in two ways:

a) An *arriving* message is an orphan (i.e. the IST has already been updated), or

b) A message that has already arrived is determined to be an orphan when the

Recovery message arrives (i.e. the IST had *not* been updated when the message arrived).

An *arriving* orphan message is detected by process q if the following is true:

$\exists\, p \mid (MsgDV[p].Inc < Inc_p) \wedge (IST(Inc_p, p) \leq MsgDV[p].RSN)$ [Stro85], where $p \neq q$

This check must be performed for *each* one of p's entries (Inc_p) in the IST. Since, as shown in Lemma 3.1, a message $m \in k_l^q$ can be an orphan message, dependency vectors *must* be logged with the message and its receive order information. This has been overlooked in previous papers and will be addressed again in Section 3.7 .

In addition to checking each message if it is an orphan, whenever a process q receives a *Recovery* message it checks to see if its *current* state interval depends on some unordered/orphan message:

$\exists\, p \mid (Inc_p > DV_q[p].Inc) \wedge (IST(Inc_p, p) \leq DV_q[p].RSN)$ [Stro85], where $p \neq q$

This check must be performed for *each* one of p's entries (Inc_p) in the IST. If the process q is determined to be an orphan process then it must be rolled back to its earliest unreclaimed checkpoint.

Theorem 3.2: Every process that is orphaned by the loss of pre-failure state intervals due to unordered message(s), and no processes that are not orphans, will be rolled back.

Proof —

The post-failure execution is identical to the prefailure execution until the first unordered/orphan message is encountered during the replay of a rolled-back

process. Then a new IST entry is logged to stable storage before any further (re)execution is performed. By definition of the IST, a new entry $IST(Inc_q, q) = RSN$ causes all state intervals greater than $IST(Inc_q - 1, q) = RSN - 1$ to become orphan intervals. The new IST entry is guaranteed to be received by every process because:

1) (*Recovery*) messages are reliably delivered without errors (assumption) and,

2) *Recovery* messages will be resent in the event of the failure of some process(es) p prior to the logging of q's new entry onto the stable storage copy of IST_p. (This is the purpose of ''confirmed'' flag associated with new IST entries on stable storage).

Since a process' dependency vector correctly records which state intervals it depends on (Lemma 3.2) and the ISTs record the system history, the presented recovery algorithm for detecting orphans detects all orphan processes and only orphan processes. □

3.6.3. Messages in k_f^q Following Ordered Playback

The detection of an orphan or unordered message during replay (step 2 of the recovery algorithm) indicates that complete recovery can not be achieved and further execution *may* diverge from the pre-failure execution of the failed process. All replay *prior* to the first orphan/unordered message has been identical to the pre-failure execution. This phase is called *ordered playback*. The remaining messages in k_f^q can be found in stable logs and in sender volatile logs. The issue that needs to be addressed is how to handle these messages. The obvious choices are to discard all of them, to replay all of them, or some combination of the two.

This part of the recovery algorithm has not been discussed in previous message logging papers.

Discarding a message in the log of a recovering process is equivalent to a lost message during normal operation. Hence, in order to ensure consistency, either the sender is rolled back to an interval prior to the sending of the discarded message, or a copy of the message has to be stored (logged) somewhere when it is originally sent so that it can be retransmitted (replayed) as part of recovery.

Consider the first alternative of discarding some of the messages remaining in k_f^q after the ordered playback and rolling back the sender(s) of those messages. There are two problems with this solution: A) how to identify the senders that need to be rolled back, and B) how to ensure that senders do not reclaim the checkpoints that they will need for this purpose. Identifying senders that need to be rolled back requires a way to map messages (those that were found in the logs and discarded) back to their senders. This information can be either maintained at the sender (a list of messages along with the receiver IDs, receiver incarnations and RSNs) or maintained with the message logs (e.g. a sender ID for each message). The second problem with rolling back the senders is that they may need to be rolled back to a checkpoint that has already been reclaimed, by following the checkpoint reclaiming rules described in[Stro85].

The checkpoint reclaiming rules in[Stro85] allow a checkpoint C^i to be reclaimed when 1) it is not needed for state backout (it does not depend on any state interval that has not been logged) and 2) all the messages sent prior to a subsequent checkpoint, C^{i+1}, have been acknowledged as logged to stable storage. However, as discussed in Section 3.3 these checkpoint reclaiming rules may

reclaim a checkpoint that is needed for rolling back a sender of a message discarded from the stable log. Hence, an additional requirement must be met before a checkpoint can be reclaimed. Specifically, C^i can be reclaimed only if all messages sent prior to C^{i+1} are somehow guaranteed not to become orphans. This requires that every message, upon which any one of these messages depends must be logged to stable storage.

The use of sender rollback to solve the lost message inconsistency problem results in additional process rollbacks and extra overhead. Hence, a solution based on replaying all the messages remaining after ordered rollback is preferable. Specifically, *all* non-orphan messages on the stable log are replayed, in RSN order, and then (re)logged to stable storage. In detail, every message $m \in k_f^q$ is either:

- *ordered* — all messages $m \in k_l^q$ and any messages $m \in k_r^q$ retrieved from non-failed senders. or,
- *unordered* — all messages $m \in k_s^q$ retrieved from non-failed senders, any messages $m \in k_l^q$ that follow (RSN order) an orphan message, and any newly sent messages that, therefore, were not in k_f^q. This category includes messages sent by senders that *did* fail.

Assumption: for ease of explanation we assume that either all messages in k_r^q and k_s^q for a particular sender are retrieved (sender hasn't failed) or none of them are (sender has failed). As long as the sender sends these messages in the original SSN order this restriction is easily lifted.

All non-orphan messages in k_f^q, ordered followed by unordered, will be received and only the senders of orphan messages will have to roll back. Thus case 2 requires no extra information to be logged and no additional processes to be

rolled back.

Theorem 3.3: All senders of orphan messages have rolled back or will roll back.

Proof —

By definition of orphan messages —

An orphan message transitively depends on some unordered message. Therefore, the sender of an orphan message must have either:

1) receiv*ed* an orphan message (and is therefore an orphan process which has or will roll back, Thm. 3.2), or
2) detected an unordered message *during recovery* (in which case the process has already rolled back).

Hence, all senders of orphan messages have rolled back or will roll back. □

Theorem 3.4: All non-orphan messages in k_f^q will be received.

Proof —

It follows from Thm. 3.2 that all *orphan* messages will not be received and from Thm. 3.3 that they will not be sent. Therefore we now look only at non-orphan messages.

- All messages in k_f^q are retrieved from the senders unless the sender(s) have failed (assumption), or
- For each sender process p that *has* failed the messages it contributed to k_f^q will either 1) be regenerated or 2) become orphans in the post-failure execution of p.
    1) If the message(s) is regenerated it will be received by q. The regenerated message might, however, be processed by q in a different

lexicographical position in the post-failure (Incarnation,RSN) order compared to its original position in k_f^q. However, this position can only differ with respect to concurrent events (messages from other senders) and not with respect to those messages sent from p (follows from Thm. 3.1 — messages can only be received in monotonically increasing send sequence order).

2) If the message(s) is an orphan then it is no longer in the system history and must not be sent or received. The *sender* has already rolled back — thereby undoing the send. Since the sender has failed:

if the message *is not logged* it can not be retrieved from the failed sender since messages that are not yet logged are stored in volatile memory and are lost during failure/rollback.

if the message *is logged*, the logged orphan can pass through the orphan detection filter only if the IST table entry indicating that it is an orphan has *not* been made (and, therefore, a *Recovery* message has not been received). If this occurs the receiving process will have to roll back (again) to undo the receipt of the logged orphan when the *Recovery* message is eventually received (which is guaranteed by Thm. 3.2). This time the orphan will not be received (the IST *must* have been updated).

$\therefore$ all messages in k_f^q are either retrieved from non-failed senders, are regenerated by failed/rolled back senders, or they become the orphans of the failed sender. Retrieved messages and regenerated messages are received and orphans are not (or, if they are received their receipt is later undone). Hence all non-orphan messages in k_f^q will be received by the post-failure/rollback execution of process q. □

It would appear that the problem of case 1a (reclaiming checkpoints) exists for case 2 as well since messages that are logged may become orphans. This is not the case.

Theorem 3.5: A checkpoint C_q^i can be reclaimed when it is:

1) No longer needed for state backout:

   $(LV_q[p].Inc, LV_q[p].RSN) >= (C_q^{i+1}.DV[p].Inc, C_q^{i+1}.DV[p].RSN)$ for all processes p.

   That is — every state interval upon which the checkpointed state interval C_q^{i+1} depends has been logged. *and*,

2) No longer needed for for message recovery:

   When all messages sent by q between C_q^i and C_q^{i+1} are logged by the receivers to stable storage.

Proof —

1) We have already shown that a dependency vector correctly records a state interval's dependencies (Lemma 3.2). Therefore, if messages are eventually logged and log vectors are periodically updated then C_q^i will eventually not be needed for state backout.

2) Once a message is logged it is always used during replay — unless it is determined to be an orphan (by the orphan detection filter) or a duplicate (by the SSN protocol filter). If the message sent by q is a duplicate then it is clearly not needed for recovery. If the message sent by q can become an orphan then C_q^i is needed for state backout and will not be reclaimed (part 1).

$\therefore$ checkpoint C_q^i can be reclaimed when all messages sent prior to the

checkpointed state have been logged to stable storage and when every state interval upon which C_q^{i+1} depends has been logged. □

3.6.4. Summary

In Section 2.1 we stated that recovery must restore a consistent global state — a system state composed entirely of consistent process (and channel) states such that every pair of processes and the connecting channels ‘‘agree’’ on the number and content of messages which have been exchanged. In the message logging algorithm, for a single process q that fails, the pre-failure messages sent to q (k_f^q) are broken down into three categories: logged (k_l^q), received but not logged (k_r^q), sent but not received (k_s^q). All of the messages in k_f^q are either retrieved from the stable message log (k_l^q), from non-failed senders (some/none/all of k_r^q and k_s^q), or are regenerated by failed senders (unless they are orphans). All *orphan* messages in k_f^q are neither sent (Thm. 3.3) nor received (Thm. 3.2) while all *non-orphan* messages in k_f^q are sent (retrieved from stable storage, from the volatile memory of non-failed senders or from the reexecution of failed senders) and received (Thm. 3.4). Each pre-failure message sent by q (l_f^q) is either regenerated and deleted (Thm. 3.1) or is orphaned and the receiver is rolled back (Thm. 3.2). Thus, all messages sent are received and all messages received are sent such that no messages are lost or duplicated.

3.7. Other Issues

In the presented recovery algorithm, the IST of a process is maintained on stable storage using an optimized distributed commit protocol. A confirmed/unconfirmed flag is added to q's own IST entries so that a failed process q can continue recovery (and normal operation) *without* waiting for the replies to the broadcasted *Recovery* message. Thus, the critical path during recovery is 1) retrieving the earliest checkpoint, data structures, and the current IST, 2) replaying ordered messages until an unordered or orphan message is encountered, 3) logging the new IST entry to stable storage and broadcasting the *Recovery* messages, and 4) finishing replay with unordered messages (from stable storage and the senders volatile logs). With this method every process is still guaranteed to eventually receive and log every (new) IST entry. Further optimizations are possible. In any case, robustly updating the IST is critical to any message logging algorithm (in which orphans can occur) since processes must be able to correctly determine whether or not their current state depends on some unrecoverable state interval.

In step 4 of the Recovery algorithm any remaining messages on the stable log are treated as unordered messages (which are not part of the system history) and may later be logged as ordered messages (which are part of the system history). Any logged unordered messages which are determined to be orphans will not be (re)logged. We have not provided any additional mechanism for reclaiming these unordered messages. These messages are reclaimed when, as stated in Section 3.3, the previous checkpoint has been reclaimed. The disadvantage of not reclaiming these unordered and orphan messages earlier is that they will be replayed during

every recovery until they have been reclaimed. These messages could be reclaimed sooner if, during replay:

1) The SSN, sender ID, and sender incarnation number of each message being (re)logged by q is compared to any unordered messages in q's stable log. A match indicates that the unordered message can be reclaimed. and,

2) When an unordered message is determined to be an orphan, a special system message can be sent to reclaim the orphan from the stable log.

We anticipate that very few such messages will exist and that the overhead on stable storage and during recovery will be minimal and therefore not warrant early reclamation.

The issue of committing outputs to the outside world has not been discussed in detail because it is identical to the problem of reclaiming checkpoints. Output commitment can be implemented in a variety of ways. For example, each node in the system can be held responsible for collecting information about the logging progress of every process in log vectors (Section 3.2). When *all* messages (state intervals) upon which the output message depends have been logged, the message may be released [Stro85]. This requires periodic broadcasting of logging progress by all processes. Alternatively ‘‘intelligent’’ disk nodes could keep track of any messages tagged ‘‘output’’ and then release them in a similar fashion. Other methods that attempt to commit the outputs more quickly, with some performance overhead, have been presented [Stro88, Elno92a] (see Chapter 6, Section 6.2.1.4 for a brief discussion).

3.8. Summary

A new asynchronous message logging algorithm for multicomputers has been presented. A correctness model based on[Lamp78, Chan85] was used as the foundation for a complete set of proofs. The presented algorithm is distributed, application-transparent, and n-fault-tolerant. Process checkpoints and interprocess messages are logged to stable storage asynchronous of process execution. Messages and receive sequencing information are also temporarily logged in the local volatile memory of the sender. This allows the algorithm to use volatile message logs in addition to stable message logs to recover as much of the failed process(es)' pre-failure execution as possible. The use of these volatile logs reduces the number of non-failed processes (orphans) which must roll back in order to achieve a consistent system state. In fact, an orphan process Q will occur only when 1) a message m received by P was not successfully logged to disk at the time of P's failure, 2) P sent a message to Q after receiving m and prior to failing, *and* 3) the sender of m does not have a copy of that message (with *RSN*) — i.e. the sender also failed.

One of the major contributions of this chapter is the presentation of a complete proof that this optimized algorithm is *correct* — in the presence of failures the algorithm can be used to recover one or more processes in a distributed application without violating the application's partial ordering of events and without duplicating or deleting any events. Previous papers have given partial proofs or have made strong assumptions that negate the advantages of using message logging as an error recovery technique.

The presented algorithm maintains and improves upon the principal

advantages of the message logging technique by checkpointing and rolling back individual processes whenever possible. Further, it employs no synchronization (pessimism) and is therefore more likely to be able to support applications with different computation-communication ratios. Since the algorithm is n-fault-tolerant and does not assume the existence of any local stable storage it is suitable for use on multicomputers which have multi-tasking nodes and remote storage devices — as long as sufficient extra network bandwidth, disk bandwidth, and disk storage is provided for the message logs and checkpoint states. Chapter 6 contains a performance analysis of this and other distributed application-transparent recovery algorithms appropriate for multicomputers.

Chapter Four
Coordinated Checkpointing Techniques

Coordinated Checkpointing techniques checkpoint a set of processes together (the entire system or a subset thereof) in such a way that each pair of process states on stable storage are consistent with one another. The recovery algorithm is guaranteed to be able to find a recovery line and hence be able to recover a valid system state. These algorithms allow the system to be n-fault-tolerant, keep only one generation of checkpoints on disk, and avoid the domino effect.

In this chapter we present two new low-overhead algorithms based on the coordinated checkpointing of processes. The first algorithm is synchronous coordinated checkpointing of interacting sets of processes, which extends the global coordinated checkpointing technique[Tami84] to perform checkpointing and recovery of sets of interacting processes rather than of the entire system. This extension reduces the disruption to normal operation since it usually involves only a subset of processes instead of the entire system. With coordinated checkpointing of interacting sets of processes, checkpointing involves saving a consistent snapshot of the states of an interacting set of processes in such a way that a valid global checkpoint of the system state is maintained on stable storage at all times. In the proposed schemes checkpointing and recovery are done at the level of processes with no system-wide central coordination. Multiple checkpointing and recovery sessions may be active simultaneously. Unrelated sessions do not interfere with one another, while the actions of related ones are properly coordinated. Processes which are not part of the interacting set need not participate

in checkpointing/recovery and continue to do useful work. However, synchronous coordinated checkpointing, like global checkpointing[Tami84] still involves synchronous checkpoint sessions, where all members of the interacting set being checkpointed suspend normal execution for the duration of the checkpoint session.

The second algorithm is asynchronous coordinated checkpointing which achieves the benefits of checkpointing processes independently (i.e. without coordination), but without suffering from the domino effect. The key feature of this scheme is the use of volatile checkpoints in combination with the coordinated checkpointing technique to minimize disruption to normal operation due to checkpointing. A volatile checkpoint is simply a copy of the process state in local volatile memory. Checkpointing begins by copying the changed state of a process to local volatile storage, after which the process may resume execution. The rest of the checkpointing session involves identification of the set of processes to be checkpointed and the transfer of the volatile checkpoints to stable storage. Thus, a process is suspended only a small fraction of the time it takes to complete a checkpointing session. This fraction of time is independent of network delays and the number of processes that constitute the application. The cost, relative to synchronous coordinated checkpointing, is an increase in the complexity of the checkpointing algorithm which must ensure that the volatile checkpoints are saved as part of a consistent global state even though the processes being checkpointed have resumed execution. However, volatile checkpoints allow much of this local copying to be avoided through the use of a slightly modified virtual memory system. This chapter provides detailed proofs of correctness for our algorithms, based on the model presented in Chapter 2.

In the next section we provide an brief overview of previous work on coordinated checkpointing algorithms. Then we describe a global state recording algorithm, from [Chan85]. In Section 4.1.2 we show how this compares to the complete error recovery technique, global checkpointing, described in[Tami84]. We then provide high-level pseudo-code to checkpoint and roll back the states of an interacting set of processes. Several simplifying assumptions are made so that we can prove the correctness of the basic approach of checkpointing and recovering interacting sets of processes free of low-level implementation issues. These simple, short proofs are applicable to any algorithm based on checkpointing interacting sets of processes. We first show that the pseudo-code can be used to *maintain* a consistent global state on stable storage and then show that, when a failure occurs, only the interacting set(s) of the failed process(es) needs to be rolled back.

In Section 4.2 we describe how the checkpointing or rollback of an interacting set is coordinated in a tree-structured manner with the coordinator being at the dynamically-chosen root. We describe how the algorithm is designed around a two-phase-commit protocol, how independent checkpoint and recovery sessions do not interfere with each other, and how coordinator conflicts are handled. Several proofs of the correctness of these procedures are provided.

The synchronous checkpointing algorithm is described in Section 4.3 and provides details on supporting communication environments that use either virtual circuits or packet-switching. The recovery algorithm, used with either the synchronous or asynchronous checkpointing algorithms, is described in Section 4.4. Recovery from communication errors is discussed first followed by

recovery from node failures. Section 4.4.3 specifically discusses how our checkpointing algorithms are robust to (non-crash) failures.

In Section 4.5 we present the complete asynchronous coordinated checkpointing algorithm in which we minimize the interruption of normal processing due to checkpointing. In Sections 4.5.1, 4.5.2, and 4.5.3 we detail the handling of multiple ‘‘competing’’ checkpoint coordinators and checkpoint sessions. These three sections are rather similar to the synchronous algorithm, so we provide a detailing of the actions without going through lengthy proofs. Techniques for minimizing data movement during checkpointing by using an enhanced virtual memory system are outlined in Section 4.6 and then the chapter is concluded in Section 4.7. In Chapter 6 a performance analysis of these algorithms, and others, is provided.

4.1. Previous Work

Barigazzi and Strigini [Bari83] proposed an error recovery procedure for multicomputers that involves the periodic saving of the state of each process by storing it both on the node where it is executing and on another backup node. The critical feature of this procedure is that all interacting processes are checkpointed together, such that their checkpointed states are always consistent with one another. Thus the *domino effect* cannot occur and it is sufficient to store only one ‘‘generation’’ of checkpoints. The schemes presented in this chapter use this idea of checkpointing and recovering dynamically changing sets of interacting processes.

With the recovery scheme described in[Bari83] a large percentage of the

memory is used for backups rather than for active processes. The resulting increased paging activity leads to increases in the average memory access time and the load on the communication links. This load is also increased by the required acknowledgment of each message and transmission of redundant bits for error detection. The communication protocols, which are used to assure that the message ‘‘send’’ and ‘‘receive’’ operations are atomic, require additional memory and processing resources for the kernel. Thus, performance is significantly reduced relative to an identical system where no error recovery is implemented. The schemes proposed in this chapter eliminate the requirements for atomic message transmission and provide the ability to save the checkpoints on disk, where they need not have a detrimental effect on system performance. They also incorporate the use of practical, low overhead error detection mechanisms, to reduce even further the overhead required on application messages.

The idea of checkpointing and recovering interacting sets of processes is extended in[Tami84] to checkpointing and recovering the entire system (global checkpoints). That scheme does not have the disadvantages discussed above of the scheme in[Bari83]. The problem with the global checkpointing technique is that checkpointing is expensive since it requires saving the state of the entire system. Thus, for performance reasons, the time between checkpoints is relatively long (possibly tens of minutes). Hence, the system can only be used for ‘‘batch applications,’’ such as large numerical computations, where the possibility of losing minutes of computation during recovery is an acceptable price for the resulting low overhead (a few percent [Tami84]).

The global checkpointing technique has been extended to perform

checkpointing and recovery of sets of interacting processes rather than of the entire system [Koo87, Tami89]. This extension reduces the disruption to normal operation since it usually involves only a subset of processes instead of the entire system.

Our synchronous algorithm[Tami89] (described in Section 4.3 and Section 4.4) differs from the one presented in[Koo87] in several ways. First, we do not assume that the network makes channels ‘‘lossless’’, thus the definition of an interacting set is more general (a superset). We have instead specifically provided for low overhead recovery from node failures and from communication failures when the system uses either a virtual circuits or packet-switching model for communication. Also, through the use of local dynamic communication vectors we do not require any message tags (labels) on either normal or system messages. By flushing and discarding messages during a recovery session we completely avoid the livelock problem they describe. Lastly, we differ in our methods of handling interfering checkpoint/recovery sessions (see Section 4.2.1.2) and do not unnecessarily abort checkpointing sessions.

However, as in[Tami84], all of these coordinated recovery schemes still involve *synchronous* checkpoint sessions, where all members of the interacting set being checkpointed suspend normal execution for the duration of the checkpoint session. In the worst case, all processes in the system may belong to a single interacting set and all normal computation in the system has to be suspended for the entire checkpoint session. Our second algorithm, asynchronous coordinated checkpointing, requires processes to be suspended only for the taking of a local volatile checkpoint — a fraction of time that is independent of network delays and

the number of processes that constitute the application. The taking of a volatile checkpoint is highly amenable to optimization. No asynchronous coordinated checkpointing algorithm has previously been proposed.

4.2. Basic Approach of Coordinated Checkpointing

Instead of checkpointing processes independently, coordinated checkpointing techniques ‘‘simultaneously’’ checkpoint multiple processes in order to maintain a consistent global state on stable storage. *Global* checkpointing techniques record the entire (consistent) system state in every checkpointing session. Alternatively, coordinated checkpointing of interacting sets of processes can be used to update a subset of the consistent global state at a time, while ensuring the existence of a consistent global state at all times. In this section we show the correctness of both of these approaches.

4.2.1. Recording a Consistent Global State

All coordinated checkpointing techniques have in common the characteristic of keeping a consistent global state on stable storage. When a hardware failure is detected all or part of the consistent global state is used to restore a valid system state from which computation can be resumed. These techniques all share the advantage that domino effect is entirely avoided by keeping only one generation of checkpoints in stable storage.

In[Chan85] a *consistent global state* is defined as follows:

- n is the number of messages sent by process p along channel c before p’s

state is recorded,

- n' is the number of messages sent along c before c's state is recorded,
- m is the number of messages received by process q before q's state is recorded,
- and m' is the number of messages received along c before c's state is recorded.

A *consistent global state* requires $n=n'$ and $m=m'$ for all channels c between processes p and q in the system[Chan85]. If q (the receiving process) records c's state and we consider the state of channel c to actually be part of process q's state then a consistent global state requires $n=m$, for all channels c between processes p and q. In other words, instead of recording channels as separate objects, all channels are recorded as empty ($n'=m'$) and q records as part of its state all messages which have been received along c such that no messages are lost and no messages are duplicated (see Section 2.1). The initial global state is consistent and $n = 0 = m$ for all channels c between each pair of processes p and q.

To record a consistent global state Chandy and Lamport presented the following algorithm which is initiated by some process p (reworded slightly from[Chan85]):

Global State Recording Algorithm

Marker-Sending Rule for a process p:

For each channel c, incident on, and directed away from p:

— p sends one marker along c after p records its state and before p sends

further messages along c.

Marker-Receiving Rule for a process q:

On receiving a marker along a channel c:

if q has not recorded its state **then**

begin

— q records its state;

— q records the state c as the empty sequence

— q sends one marker along each channel c, incident on, and directed away from q after q records its state and before q sends further messages along c;

end

else

— q records the state of c as the sequence of messages received along c after q's state was recorded and before q received the marker along c.

If one process p initiates the global-state recording algorithm then, in order for all processes in the system to record their state a path must exist between process p and all other processes q. In other words, the system graph, where each arc is a channel and each vertex a process, must be strongly connected. Then, if no marker remains forever in an incident input channel, the global-state recording algorithm will result in each process recording its state (and the states of all incoming channels) and will terminate in finite time[Chan85].

The above algorithm assumes that channels are static for the duration of a

distributed computation. In other words, all channels that will exist between processes exist at the beginning of the computation and continue to exist until the end of the computation. Processes must also have static life for the duration of the computation.

4.2.2. Global Checkpointing

Global Checkpointing techniques[Tami84] maintain a consistent global state in stable storage by checkpointing and recovering the entire system. The algorithm presented in[Tami84] to take a global checkpoint is fundamentally quite similar to the above Global State Recording Algorithm. However, the global checkpointing algorithm was specifically designed for use with large multicomputers where the *channels* defined above are actually (directed) high-speed dedicated links and *processes* are the nodes interconnected by them. Recording the state of a node includes all local memory, any messages *flushed* from incoming links by markers (system messages), and all appropriate system tables. The target multicomputer is assumed to be strongly connected. The global state is recorded on disks which are connected to some fraction of the nodes in the system. When an error is detected, due to a hardware failure in the links or in the nodes, the entire system is rolled back to the latest global checkpoint stored on the disks. A proof given in [Chan85] shows that a (static) distributed system can be rolled back to a global checkpoint and not violate the partial ordering of events in the system.

The problem with the global checkpointing technique is that checkpointing the entire system state is expensive. Thus, for performance reasons, the time between checkpoints is relatively long (on the order of thirty minutes). This means

that the system can only be used for batch applications, such as large numerical computations, where the possibility of losing thirty minutes of computation during recovery is an acceptable price for the resulting low overhead (a few percent [Tami84]).

4.2.3. Recording Interacting Sets of Processes

The error recovery schemes presented in this chapter maintain a consistent global state on stable storage by periodically recording the states of interacting sets of processes. In contrast to global checkpointing the entire system need not suspend normal operation during checkpointing — only the interacting set being checkpointed is suspended. Asynchronous checkpointing of interacting sets (see Section 4.5) further minimizes the time that processes must be suspended. During recovery only the interacting set(s) containing the process(es) that were running on the failed node(s) are rolled back.

4.2.4. Maintaining a Consistent Global State

In this section we will prove that a consistent global state can be maintained in stable storage by recording the checkpoints of dynamically changing interacting sets of processes. In the next section we will show that interacting sets can also be used to roll the system back to a consistent state when a failure has occurred. In both sections we make several simplifying assumptions which will be addressed as implementation issues in later sections. Thus, the proofs given in the following sections are generally applicable to all error recovery techniques based on checkpointing and rollback of interacting sets of processes.

First, we briefly consider a system in which processes are *statically* partitioned into tasks such that processes within a task do not communicate with those that are outside of the task. In such a system channels need not exist between tasks. Recording the global state of this system would then require each task to record its own ‘‘global’’ state with the above Global State Recording algorithm which we might now call a Task State Recording algorithm. It is easy to see that if there are N tasks in the system, each containing some number of processes, any grouping of N recorded task states (one for each task) will form a consistent global state simply because processes within one task can not affect processes within another. In other words, there is no ordering of events in different tasks, as specified by the *happens before* relation defined in Section 2.1, and hence such events are concurrent.

We now extend this simple principle to present and prove the correctness of *updating* a recorded consistent global state by checkpointing interacting sets of processes. In this global state updating algorithm (procedure *Chkp*) subsets of system processes are *dynamically* partitioned into interacting sets of processes. In order to be able to dynamically determine the members of an interacting set of processes, each process p maintains a *Buddies* list:

Def. 4.1: For a process p, let *Buddies (p)* be the set of processes such that $q \in Buddies(p)$ if, and only if, p has either sent a message to q or received a message from q since the last checkpoint of p.

Note that in general $q \in Buddies(p)$ does <u>not</u> necessarily imply that $p \in Buddies(q)$. For example, a message may have been sent by p but not yet received by q.

We can now define an interacting set in terms of the *Buddies* list:

Def. 4.2: The *interacting set* with respect to process p_1, $Inter(p_1)$, is the set of processes containing all processes q such that either (1) $q \in Buddies(p_1)$, (2) $\exists$ a process p_2, such that $p_2 \in Inter(p_1)$ and $q \in Buddies(p_2)$, or (3) $q = p_1$.

Buddies and *Inter* can be viewed as binary relations: e.g. $q \in Buddies(p)$ as $(p,q) \in Buddies$ and $q \in Inter(p)$ as $(p,q) \in Inter$. Def. 4.2 states that *Inter* is the reflexive transitive closure of *Buddies*.

$Inter(p_1)$ is the set of processes that may be affected (directly or indirectly) by messages sent by p_1 since the last checkpoint of p_1 *or* which have produced messages which may have affected p_1 directly or indirectly. Note that, in general, $p_2 \in Inter(p_1)$ does <u>not</u> necessarily imply that $p_1 \in Inter(p_2)$.

When a process p is (periodically) checkpointed, the following procedure is performed:

Procedure Chkp (the Global State Updating Algorithm):

1) When a process p is checkpointed, all processes in $Inter(p)$ (Def. 4.2) are also marked for checkpointing.
2) All messages sent by processes in $Inter(p)$ are flushed to their final destinations.
3) The actual checkpointing is then performed.

These interacting sets are checkpointed independently to stable storage, like the above tasks, and a consistent global state (checkpoint) is always maintained. Note that if the interacting sets were static instead of dynamic the global-state updating

algorithm would be equivalent to the above task recording algorithm and, starting with an initial global state, a consistent global state would always be maintained.

Def. 4.3: Two processes, p and q, where $q \in Buddies(p)$, are said to be *buddy-consistent* if, and only if, their states reflect the same number and content of **direct** message exchanges between p and q.

Note that if $q \in Buddies(p)$ and p and q are *buddy-consistent*, then it must be the case that $p \in Buddies(q)$. This definition implies that no messages between p and q are either lost or duplicated.

Def. 4.4: A set of processes, CPS, is *consistent*, if, and only if, the following two conditions hold: (1) if $p \in CPS$ then $q \in Inter(p)$ implies $q \in CPS$, (2) for every pair of processes in CPS, $p \in CPS$ and $q \in CPS$, if $p \in Buddies(q)$ or $q \in Buddies(p)$, then p and q are *buddy-consistent*.

If the system is fault free and all messages in transit are flushed to their final destinations, all the processes in the system form a consistent process set (CPS). Note that a single process, p, for which $\{p\} = Inter(p)$, is a consistent process set.

Def. 4.5: Two processes are said to be *consistent* if, and only if, there exists a consistent process set of which both are members.

Def. 4.6: A set of processes, $DCPS$, is *dynamically consistent* if, and only if, the following holds: if the execution of all processes in $DCPS$ is blocked and all messages are flushed to their final destinations, all the members of $DCPS$ form a consistent process set.

Def. 4.7: Two processes are said to be *dynamically consistent* if, and only if, there exists a dynamically consistent process set of which both are members.

At this time we assume that the checkpoints of all processes within an interacting set are committed onto stable storage at the same ‘‘instantaneous’’ moment. Since we plan to use the consistent global state for recovery and need to recover from hardware failures which can occur at any time a consistent global state must *always* be available for recovery. In other words, the saved global state can not be allowed to be inconsistent even for a short period of time. If this requirement is imposed on Chandy and Lamport's global state recording algorithm then, when a new global state is being recorded we can not write over the previous state until the new global state has been completely saved to stable store. For example, one process p might *coordinate* the global state recording algorithm and, after all processes have written their states to stable store, issue a *commit* command which would erase the old global state[Tami84]. This same idea can be used to commit an interacting set of checkpoints. How this is achieved will be discussed in Section 4.3.

We will now prove that the above global state updating algorithm can be used to maintain a consistent global state. First we reiterate our assumptions and then introduce and prove Lemma 4.1, which holds that, when an interacting set is being checkpointed, all of the processes which are in the interacting set (and no processes not in the interacting set) record their states. Lemma 4.2 shows that when an interacting set is checkpointed, the checkpoints of every pair of processes within the interacting set are consistent (i.e. they form a *consistent process set*). Then Lemma 4.3, which follows from Lemma 4.2, shows that the checkpoints of every pair of processes within the system are always consistent (i.e. that the entire system forms a *consistent process set*).

Assumptions:

— Only one checkpoint session is ‘‘active’’ in the system at a time.

— One process p initiates the checkpoint session.

— Channels are directed, FIFO, error-free and transmit messages with arbitrary finite delay.

— Channels are static: if a distributed computation requires two processes p and q to communicate with each other then two directed channels (channel c from p to q and channel c' from q to p) exist between those processes for the duration of the computation.

— Processes are static: all processes that can fail exist at the beginning of a distributed computation and continue to exist for the duration of the computation.

— All the checkpoints of an interacting set are committed to stable storage atomically and at the same time.

— The distributed computation starts with an initial, consistent recorded global state.

— The system is fault-free.

Lemma 4.1: If $p_3 \in Inter(p_2)$ and $p_2 \in Inter(p_1)$, then $p_3 \in Inter(p_1)$.

Proof: Follows directly from the fact that *Inter* is a reflexive transitive closure of *Buddies* (see comment following Def. 4.2). □

Lemma 4.2: Assuming that the system is fault-free, the set of processes $P = Inter(p)$, which is checkpointed as described by Procedure *Chkp*, forms a *consistent process set*.

Proof: Consider $q_1 \in P$ and $q_2 \in Inter(q_1)$. By Lemma 4.1, this implies $q_2 \in Inter(p)$ so condition (1) in Def. 4.4 holds.

Assume $p_1 \in P$ and $p_2 \in P$. There are three cases to be considered: (1) $p_2 \in Buddies(p_1)$, (2) $p_1 \in Buddies(p_2)$, or (3) $p_2 \notin Buddies(p_1)$ and $p_1 \notin Buddies(p_2)$. Case (3) obviously satisfies condition (2) of Def. 4.4. The case where (1) or (2) hold is covered by the discussion of either one.

If $p_2 \in Buddies(p_1)$, since the system is fault-free and messages from p_1 and p_2 have been flushed to their final destinations, it must be the case that the state of both processes reflect the same message exchanges between them and thus they are *buddy-consistent* (Def. 4.3). Similarly, condition 2 in Def. 4.4 holds starting with $p_1 \in Buddies(p_2)$. □

Lemma 4.3: If every time a process p is checkpointed, its entire interacting set $P = Inter(p)$ is also checkpointed as described in Procedure *Chkp*, then the set of all checkpointed processes (all the processes in the system), W, forms a consistent process set.

Proof: Since W includes all the processes in the system, condition (1) in Def. 4.4 is satisfied. Consider two processes in the system checkpoint p_1 and p_2. If both processes were checkpointed together, i.e., $p_1 \in P$ and $p_2 \in P$, then, by Lemma 4.2, they satisfy condition (2) in Def. 4.4.

Assume $p_1 \in P$ and $p_2 \notin P$. If $p_2 \in Inter(p_1)$, then, based on Lemma 4.1, $p_2 \in Inter(p)$. Since this contradicts $p_2 \notin P$, it must be the case that

$p_2 \notin Inter(p_1)$. Hence, $p_2 \notin Buddies(p_1)$.

Consider the possibility that $p_1 \in Inter(p_2)$. By assumption, the p_2 checkpoint was stored prior to the p_1 checkpoint. If at the time p_2 was checkpointed it was the case that $p_1 \in Inter(p_2)$, then, following Procedure *Chkp*, p_1 was also checkpointed. Since p_2 is a checkpointed state and not the dynamic state of the process, it is guaranteed that p_2 has not sent *any* message to any other process since it was checkpointed. Hence $Buddies(p_2) = \phi$. Thus, $p_1 \notin Buddies(p_2)$.

Assume $p_1 \notin P$ and $p_2 \notin P$. Hence, both p_1 and p_2 are old checkpointed states. Thus, $Buddies(p_1) = \phi$ and $Buddies(p_2) = \phi$. Therefore, $p_1 \notin Buddies(p_2)$ and $p_2 \notin Buddies(p_1)$.

Based on the above, condition (2) of Def. 4.4 holds in all cases. □

4.2.5. Recovery From Consistent Global States

Based on Lemma 4.3, when an error occurs, we can restore a consistent system state by discarding all messages in transit and restoring all processes to their checkpointed states [Tami84]. However, in order to reduce the impact of recovery on system operation, it is desirable to reduce the number of processes that have to be rolled back [Koo87, Tami89]. When an error is detected, all the processes that *could have been* affected by the error are identified. The sets of processes that have interacted with the affected processes since their last checkpoint are determined, and the states of all these processes are rolled back to that last checkpoint. It is assumed that the entire state of a processor where an error is detected is lost. Specifically, if a processor running process p fails, we cannot obtain $Buddies(p)$ from the failed processor. The rest of this section shows

that if this procedure is followed, then, following recovery, all the processes in the system form a *dynamically consistent process set*.

The recovery operation proceeds as follows:

Procedure Rback:

1) A process p is marked for rollback.
2) Any process q, such that $p \in Buddies(q)$ is marked for rollback.
3) For each q, all processes $x \in Inter(q)$ are marked for rollback.
4) All messages in transit sent from processes marked for rollback are discarded from the network.
5) All processes marked for rollback are restored to their last checkpoint and normal operation is resumed.

Def. 4.8: If process p is marked for rollback, we denote by $Recov(p)$ the entire set of processes which must be rolled back, as determined by Procedure *Rback*.

Lemma 4.4: If the most recent checkpoints of all processes in the system form a consistent process set and Procedure *Rback* is performed for a process p, then all processes *running* on the system, R, (as opposed to their checkpointed states) form a *dynamically consistent process set*.

Proof: Since R includes all processes in the system, condition (1) in Def. 4.4 is satisfied. Consider two processes p_1 and p_2. If both processes are in $Recov(p)$, they are recovered together from their last checkpoint. Hence, by assumption, both processes are in the same consistent process set. Thus, condition (2) in Def. 4.4 holds.

Consider the case when $p_1 \notin Recov(p)$ and $p_2 \notin Recov(p)$. Assuming that the system was fault-free prior to the rollback, all processes in the system were in a dynamically consistent process set. Hence, if all messages were flushed to their destinations, condition (2) of Def. 4.4 must have held for p_1 and p_2. Since the rollback did not involve p_1 and p_2, condition (2) of Def. 4.4 must still hold.

Assume $p_1 \in Recov(p)$ and $p_2 \notin Recov(p)$.

Consider the possibility that $p_2 \in Inter(p_1)$. If p_1 was added to $Recov(p)$ in Step (2) of Procedure *Rback*, $p_2 \in Inter(p_1)$ implies $p_2 \in Recov(p)$ — a contradiction. If p_1 was added to $Recov(p)$ in Step (1) or (3) of Procedure *Rback*, *exist* $q \in Recov(p)$ such that $p_1 \in Inter(q)$. In this case $p_2 \in Inter(p_1)$ implies $p_2 \in Inter(q)$ which implies $p_2 \in Recov(p)$ — a contradiction. Hence, it must be the case that $p_2 \notin Inter(p_1)$. Note that this implies $p_2 \notin Buddies(p_1)$, therefore, if $p_1 \notin Buddies(p_2)$, then condition (2) in Def. 4.4 holds, and the proof is complete.

It remains to consider the case $p_1 \in Buddies(p_2)$. Since $p_2 \notin Recov(p)$, it must be the case that $p_2 \notin Buddies(p_1)$. Hence, the only interaction between p_2 and p_1 is a message(s) in transit from p_2 to p_1. If all messages are flushed to their destinations, the message(s) from p_2 will arrive at p_1 and $\{p_2, p_1\}$ will become *buddy-consistent*. At this point, condition (2) in Def. 4.4 will hold. □

Theorem 4.1: If Procedure *Chkp* is used for checkpointing and Procedure *Rback* is used for recovery, all processes in the system will form a *dynamically consistent process set* following recovery.

Proof: Directly from Lemma 4.3 and Lemma 4.4. □

The rest of this chapter can be viewed as describing techniques (synchronous and asynchronous) for efficient implementation of the procedures described above.

4.3. Coordinating Interacting Sets as Trees

In the previous sections we have shown how checkpointing interacting sets of processes can be used to maintain a consistent global state on stable storage and how interacting sets of processes can be recovered from that consistent global state in the event of a hardware failure. Further, in both cases the system is left in a (dynamically) consistent global state. However, we have made many unreasonable assumptions that have greatly simplified the error recovery problem. In the next three subsections we will eliminate or relax many of these assumptions. Finally, in Section 4.3 (and Section 4.4) we present a complete error recovery scheme — synchronous coordinated checkpointing.

A key problem with implementing the *Chkp* procedure is that, in order to commit the checkpoints of an interacting set atomically and at the same time, we must know when the interacting set has been found and when the states of all the processes within the interacting set (including the states of appropriate incoming channels) have been recorded to stable storage (and are therefore ready to be committed). Hence the checkpoint and recovery sessions require coordination.

To this end the process which initiates a session is chosen to coordinate that session. In reality, a handler process performs all checkpointing/recovery duties as discussed in Section 2.2. Checkpoint sessions are initiated by ‘‘timers’’ associated with each process[Bari83]. These timers may, for example, count machine instructions executed by the process. Thus, the frequency of checkpointing can be tuned to the specific needs of a task: a higher frequency of checkpointing results in higher overhead but in less work being lost when recovery is necessary. When the

processes in the interacting set begin execution again, after the checkpoint session has completed, their timers are reinitialized. Note that such timers are not an accurate method of determining when a checkpoint takes place: a process will checkpoint *at least* as frequently as is intended by the initial setting of its timer. When a checkpoint timer ‘‘goes off’’ the execution of the corresponding process, with process ID *pid*, is suspended and a *checkpoint coordinator*, with ID CC_{pid}, is started up. This checkpoint coordinator is then responsible for coordinating the checkpoint session, including committing the interacting set to disk.

When a node failure is detected a *recovery coordinator*, with ID RC_{pid} where *pid* is the process ID of a failed process, is started up in place of each failed process. Every node is assumed to maintain a list of the processes which are running on each of their neighboring node(s). Therefore, when a node fails its neighbor(s) will initiate recovery for the processes which were running on that node. In the event of multiple, simultaneous node failures a list of failed processes can always be gotten from stable storage if necessary, since every process (which might fail) must have an initial state stored on stable storage before that process begins execution. Each recovery coordinator coordinates the rollback of the interacting set to which its failed process belongs.

When an error in transmission is detected, via a signature mismatch during a checkpoint session (Section 2.4), the checkpoint handler that discovers the mismatch starts up a recovery coordinator and then aborts. The recovery coordinator is responsible for rolling back the interacting set that was in the midst of checkpointing.

In the rest of this section we address the issue of coordinating non-interfering

checkpoint/recovery sessions and then address the problems that arise when multiple interfering checkpoint/recovery sessions are allowed.

4.3.1. Coordinating the Checkpoint/Recovery Sessions

The *coordinator* handler is dynamically determined as part of each session. Using the methods described above, it is possible for more than one process within the same interacting set to initiate a checkpoint and/or recovery session. In the extreme case an interacting set with n processes may have n coordinators: i.e. if all n checkpoint timers go off ‘‘simultaneously’’, if all n processes fail, or some mixture of the two cases. In this section we assume that there is only one coordinator per session and only one session checkpointing/recovering at a time (in the system). In the next two sections we will address the issue of concurrent checkpoint/recovery sessions and multiple coordinators within a single session.

An interacting set of processes forms a *communication graph* where there is a vertex for each process and each arc indicates that communication has taken place between the two processes it connects. The communication graph can be transformed into a *communication tree* by designating one of the vertices as the ‘‘root process’’ or *coordinator*. This coordinator is the handler of the process which initiated the checkpoint/recovery session. All vertices which have arcs connected to the root (‘‘children’’ of the root) are called *first-level processes*. Processes/vertices which have no children are called *leaves*. The communication tree is the fundamental unit around which our algorithms are structured.

When checkpointing or recovery is initiated, the kernel spawns a *handler* process that performs the necessary operations. A handler initiated as a direct

result of a ‘‘checkpointing timer’’ triggering or an error being detected begins its operation assuming that it will be the coordinator of a checkpointing or recovery session. The handler forms a communication tree using the dynamic communication vector, the list of processes with whom there has been *direct* communication since the last checkpoint session (*Buddies list*), which is maintained for each process[Bari83]. This list is also called a *first-level list* since, if the process becomes a coordinator (and root of a communication tree), the processes on this list are the *first-level processes* mentioned above.

The coordinator initiates formation of a communication tree by sending *CHECKPOINT* or *ROLLBACK* messages to all the processes on its first-level list. These processes are then placed in either a ‘‘checkpointing’’ or ‘‘recovering’’ state, handler processes are spawned for them, the handlers send *CHECKPOINT/ROLLBACK* messages to all their first-level processes (except for the parent process), and so on. A process that is already part of the tree (has received a *CHECKPOINT/ROLLBACK* message) informs the sender that it will not be its child. A process is a *leaf process* if it has communicated only with processes that are already part of the communication tree. Each leaf process informs its parent that it is its child and that it is a leaf. Each non-leaf process waits for confirmations/denials from the roots of all its subtrees and then sends a confirmation acknowledgment to its parent. This level-by-level process continues back up to the root process. When the final acknowledgment is received by the root process, the communication tree is complete — the interacting set has been found.

The corresponding synchronous algorithms to establish a communication tree,

for either a checkpoint or recovery session, are as follows (*note*: these algorithms do not implement a complete error recovery scheme):

Algorithm to coordinate a recovery/checkpoint session:

A *COORDINATOR*, with ID CC_p or RC_p, is started up for process p which initiated the checkpoint/recovery session (as described above).

- If *COORDINATOR* is a checkpoint coordinator then p's state is recorded.
- Process p does not execute (sends no messages) until the session is complete.
  - *CHECKPOINT/ROLLBACK* messages are sent to each process r where $r \in Buddies(p)$.
  - The *COORDINATOR* then waits for acknowledgments (*CH_ACK/RE_ACK* messages) from all *Buddies*.
    - If the *COORDINATOR* is a checkpoint coordinator then all messages received from each *Buddy* r after p's state was recorded and before the corresponding *CH_ACK* (child or not_child) message was received from r are recorded.
    - If the *COORDINATOR* is a recovery coordinator then the sequence of messages which have been received from each *Buddy* r before the corresponding *RE_ACK* (child or not_child) message from r was received are discarded.
  - If any *CHECKPOINT/ROLLBACK* messages are received from some process q then
    - If ''checkpointing'' then record all messages received from q after p's state was recorded and before the *CHECKPOINT* message from q was received.
    - If ''recovering'' then discard any messages received from q prior to receiving the *ROLLBACK* message from q.
    - Send a *CH_ACK*(not_child)/*RE_ACK*(not_child) message to process q.
- Once *all CH_ACK/RE_ACK* messages have been received, then the communication tree has been formed and the interacting set has been found. The *COORDINATOR* sends *CH_FOUND/RE_FOUND* messages to all processes from whom a *CH_ACK*(child)/*RE_ACK*(child) message was received to inform them that the interacting set has been found.

Algorithm to participate in a recovery/checkpoint session:

A *HANDLER*, with ID CH_q or RH_q, is started up for process q which receives a *CHECKPOINT/ROLLBACK* message from some process p.

- If *HANDLER* is a checkpoint handler then q's state is recorded.
- Process q does not execute until the session is complete.
  - *CHECKPOINT/ROLLBACK* messages are sent to each process r where $r \in Buddies(p)$, except $r = p$.
  - The *HANDLER* waits for acknowledgments (*CH_ACK/RE_ACK* messages) from all *Buddies* (except $r = p$).
    - If the *HANDLER* is a checkpoint handler then all messages received from each

Buddy r after q's state was recorded and before the corresponding *CH_ACK* (child or not_child) message was received from r are recorded.

- If the *HANDLER* is a recovery handler then the sequence of messages which have been received from each *Buddy* r before the corresponding *RE_ACK* (child or not_child) message from r was received are discarded.

— If any *CHECKPOINT/ROLLBACK* messages are received from some process s then

- If ''checkpointing'' then record all messages received from s after q's state was recorded and before the *CHECKPOINT* message from s was received.
- If ''recovering'' then discard any messages received from s prior to receiving the *ROLLBACK* message from s.
- Send a *CH_ACK*(not_child)/*RE_ACK*(not_child) message to process s.

— Send a *CH_ACK*(child)/*RE_ACK*(child) message to process p and wait for a *CH_FOUND/RE_FOUND* message from p.

If we carefully examine procedure *Chkp* and the above algorithms we see that, for a checkpoint session, the above *COORDINATOR/HANDLER* algorithms clearly implement the *Chkp* procedure. These algorithms are similar to the original Global State Recording Algorithm, where *CHECKPOINT* and *CH_ACK* messages serve the role of markers. The *COORDINATOR* algorithm includes both the marker-sending and marker-receiving rules while the *HANDLER* algorithm implements just the marker-receiving rule.

4.3.1.1. Multiple Independent Checkpoint/Recovery Sessions

The previous sections did not handle the existence of more than one checkpointing or recovery coordinator running at the same time and/or more than one active checkpointing or recovery session. However, in our system model, any process can initiate a checkpoint/recovery session (and attempt to be that session's coordinator) at any point in time. Checkpoint/recovery messages have the identifier of the coordinator associated with them. This identifier consists of the coordinator's node ID and process ID (for the entire system the (node ID, process

ID) pairs are totally ordered). When a process receives a checkpoint/recovery message, it becomes a member of the interacting set identified by that coordinator ID (a member of $Inter(z)$ for coordinator z). In the following two sections, we present strategies for handling the situation of a process receiving additional checkpoint/recovery messages with different coordinator IDs after having already joined an interacting set.

After receiving the first *CHECKPOINT* or *ROLLBACK* message, specifying CC_y or RC_y, respectively, as the coordinator, a process p may receive from some process q a second *CHECKPOINT* or *ROLLBACK* message specifying a coordinator CC_z or RC_z, where $z \neq y$. There are two possible cases to be considered:

1) Following the procedures for forming the communication tree described earlier, the tree coordinated by y will be formed (y will receive all the *CH_ACK/RE_ACK* messages) *without* including q.

2) Following the procedures for forming the communication tree, the tree can not be formed without including q.

The first case means that once the tree is formed $q \notin Inter(y)$. The second case means that once the tree is formed $q \in Inter(y)$. For the rest of this section we will consider the first case; we will address the correct handling of competing coordinators (case 2) in the next section.

Lemma 4.5: Consider a process p, which is already participating in a checkpoint/recovery session after receiving a *CHECKPOINT* or *ROLLBACK* message specifying the coordinator to be CC_y or RC_y, respectively. If p receives from a process q a *CHECKPOINT/ROLLBACK* message specifying

a different coordinator, CC_z or RC_z, then $q \notin Buddies(p)$ and $p \in Buddies(q)$.

Proof —

The procedures for checkpointing and recovery specify that the only way p can receive a *CHECKPOINT/ROLLBACK* message from q is if $p \in Buddies(q)$. Since the first *CHECKPOINT/ROLLBACK* message p receives specifies y as the coordinator, it must be the case that $p \in Inter(y)$. By the assumption for this section, when p received this *CHECKPOINT/ROLLBACK* message and until the time that the tree formation completed (y received all the *CH_ACK/RE_ACK* messages) it was the case that $q \notin Inter(y)$. If, at any time prior to or during the formation of the tree it was the case that $q \in Buddies(p)$ then, by Def. 4.2, it would also be the case that $q \in Inter(y)$, violating the assumption for this section. Thus the only way it would be possible for q to be a member of $Buddies(p)$ is if p sent or received a normal message from q *after* y completed forming its communication tree. However as discussed in Section 4.2.1, once the communication tree is formed, participants in the session do not send or receive *any* normal message. Hence it is not possible for q to become a member of $Buddies(p)$ until the checkpoint/recovery session coordinated by y is completed. Thus $q \notin Buddies(p)$. □

In other words, this case occurs only when a process q sends a message to p which arrives during p's checkpoint/recovery session (*after* the session has begun) and *before* q joins/starts a checkpoint/recovery session. Another way of stating the above is that q is not in p's interacting set but p is in q's interacting set, from the perspective of p and q, respectively. In the next section we deal with the case

where q might actually be in p's interacting set (via *indirect* communication with p) but *looks* like it is in a different interacting set because it is sending a *CHECKPOINT/ROLLBACK* message with a different coordinator ID.

Now, there are four permutations that need to be addressed: when a process p is in the midst of a checkpoint or recovery session a *CHECKPOINT* or *ROLLBACK* message, *with a different coordinator ID*, may arrive from some process q. We now elaborate on how these cases are handled and motivate their correctness by showing that $n = m$ is preserved (no messages are lost or duplicated) for the checkpoints, or recovered states, of p and q in each of the four cases (Theorem 4.2): We define the sequence of events that comprise a failed process q to be: $e_0^q \leq e_{c-1}^q \leq e_c^q \leq e_f^q$, where

(1) e_0^q is the first event which occurs while q is in its initial state,

(2) e_{c-1}^q is the last event to occur in q before q took its (last) checkpoint,

(3) e_c^q is the first event that occurred in q after q took its checkpoint, and

(4) e_f^q is the last event that occurred in q before q failed.

The events of a non-failed process p can be defined similarly simply without state e_f^p.

case 1: If p is involved in a checkpoint session and receives a *ROLLBACK* message from q then we could:

— abort the current checkpoint session and roll back p's interacting set (as well as q's interacting set) or

— complete p's checkpoint session and discard all messages which have arrived from q prior to receiving the *ROLLBACK* message.

Clearly the latter method is preferred, so as to minimize work that is lost during recovery.

Proof —

Recall, that for this section, and thus all four cases discussed here, a different coordinator ID on a *CHECKPOINT* or *ROLLBACK* message indicates that the sending and receiving processes are not part of the same interacting set.

Since p is in the midst of a checkpoint session (and is therefore suspended from executing), there is no event $e_j^p, e_c^p \leq e_j^p$ which is causally dependent on some event $e_i^q, e_c^q \leq e_i^q \leq e_f^q$. Further, there is no event $e_i^q, e_c^q \leq e_i^q \leq e_f^q$ which is causally dependent on some event $e_j^p, e_j^p \leq e_c^p$ (or upon which e_j^p is causally dependent) (From Lemma 4.5). By discarding the message(s) m that q sent to p (which resulted in $p \in Buddies(q)$, but which have not yet been processed by p) we remove any possible dependency between p and q. Therefore p does not have to roll back and $n = m$ between the checkpoints of p and q when p's checkpoint session completes (From Lemma 4.3). □

case 2: If p is involved in a recovery session and receives a *ROLLBACK* message from q then all messages that arrived from q prior to receiving the *ROLLBACK* message must be discarded and p simply continues its own recovery sessions and sends a *RE_ACK(not child)* message to q.

Proof —

Since the checkpoints of processes p and q are consistent (Lemma 4.3) then when p and q are rolled back, and all channels between them are cleared of messages, then $n = m$ (Follows from Lemma 4.4). □

case 3: If p is involved in a recovery session and receives a *CHECKPOINT*

message from q then p must either:

— cause q to switch from a checkpoint session to a recovery session, or

— join q's checkpoint session *after* p's recovery session has completed.

The latter method is preferred in order to minimize work that is lost when a failure occurs.

Proof —

The ''old'' checkpoint of q and p's checkpoint are consistent (Lemma 4.3). There exists no event $e_j^p, e_c^p \leq e_j^p$ that causally depends on any event $e_i^q, e_c^q \leq e_i^q$ (and vice versa), (From Lemma 4.5). Therefore p can roll back to its checkpoint and q does *not* have to roll back (extends from from Lemma 4.4). Process q can not be committed to its current checkpoint until after p replies (sends a *CH_ACK*) to q (from the algorithms above). Process p does not reply to q until its checkpoint session has been committed (also from algorithm). However, p's checkpoint will not be consistent with q's new checkpoint unless p checkpoints as part of q's interacting set. Since $p \in Buddies(q)$ p receives a *CHECKPOINT* message from q and, as a result, joins q's checkpoint session. At the end of the checkpoint session the checkpoints of p and q will be consistent (Lemma 4.2). Therefore, given the assumption that the checkpoints of an interacting set are committed atomically and at the same time, then at no time are the checkpoints of p and q not consistent. □

case 4: if p is involved in a checkpoint session and receives a *CHECKPOINT* message from q then p could either:

— join q's checkpoint session *after* completing its own checkpoint session, or

— abort q's checkpoint session — i.e. it may be desirable to adopt a system policy that prevents any process from participating in more than one checkpoint session at a time.

Proof —

First we examine the approach where p joins q's checkpoint session after completing its own checkpoint session.

At the end of p's checkpoint session p's new checkpoint will be consistent with q's old checkpoint (Lemma 4.3). Process q will not be committed to its new checkpoint until p replies (sends a *CH_ACK*) to q (from the algorithms above). Process p does not reply to q until its checkpoint session has been committed (from algorithms). To be consistent with q's new checkpoint p must checkpoint as part of q's interacting set, which it will do since $p \in Buddies(q)$. At the end of q's checkpoint the checkpoints of p and q will be consistent (Lemma 4.3). Therefore, given the assumption that the checkpoints of an interacting set are committed atomically and at the same time, then at no time are the checkpoints of p and q inconsistent. □

Second, p aborts q's checkpoint session.

In order to abort a checkpoint session we need to introduce a new system message *ABORT* which p would send to q. Process q would then forward this message to all processes to whom it previously sent a *CHECKPOINT* message *or* received a *CHECKPOINT* message from (and so on). Then q would cease all checkpoint duties and destroy the new checkpoint. Eventually all processes in q's interacting set would receive the *ABORT* message (proof follows trivially from Lemma 4.1). Since q's checkpoint can not be committed until p replies to q with a *CH_ACK*

message (from the algorithms), then q's checkpoint will clearly never be committed if p replies with an *ABORT* message. Process q's ''old'' checkpoint (only checkpoint after the new one is destroyed) is consistent with p's new checkpoint (from Lemma 4.3). Further, if p's checkpoint is also aborted (i.e. by some process in q's interacting set), the ''old'' checkpoint of p is consistent with the ''old'' checkpoint of q by the same argument. □

For the remainder of this thesis we will assume that the first option is chosen: in other words we will not allow a process which belongs to one checkpoint session to abort another checkpoint session, since, in the worst case ''deadlock'' could occur where processes p and q never checkpoint. We have now shown that, for all four cases, $n = m$ is preserved for the checkpoints, and recovered states, for two processes p and q where p and q belong to different interacting sets (and therefore different checkpoint/recovery sessions). This completes our informal proof.

4.3.1.2. Multiple Coordinators

In the previous section we assumed that two system messages that contain different coordinator IDs were sent by processes belonging to two different interacting sets. Therefore, when a process p involved in a session coordinated by a process x receives a *ROLLBACK/CHECKPOINT* message from some process q with coordinator y, p and q are guaranteed to be in distinct interacting sets. In this section we relax this assumption completely. Since it is possible for several processes *within a single interacting set* to initiate checkpointing and/or recovery sessions ''simultaneously'', processes p and q *might* actually be in the *same* interacting set, in which both x and y initiated a checkpoint/recovery session. In

order to be able to commit the checkpoints of an interacting set atomically and at the same time we must have a single coordinator for each session. To this end we *deterministically* choose a checkpoint/recovery coordinator from amongst those processes that initiated a checkpoint/recovery session.

One simple protocol is to always choose recovery over checkpointing, when a *ROLLBACK* message is received by a checkpointing process, and to always choose a ‘‘larger’’ coordinator ID over a ‘‘smaller’’ coordinator ID, when a *ROLLBACK* message (with a different coordinator ID) is received by a recovering process or a *CHECKPOINT* message (with a different coordinator ID) is received by a checkpointing process[Tami89]. Due to the stepwise confirmation/denial process of the algorithms it is possible to create a correct a single communication tree with one coordinator by ‘‘disassembling’’ all but one of the subtrees and incorporating their members in the single ‘‘winning’’ tree. If a checkpointing process receives a *ROLLBACK* message, the *ROLLBACK* message ‘‘wins’’ regardless of the coordinator ID. If a checkpointing (or recovering) process receives another *CHECKPOINT* (or *ROLLBACK*) message with a larger ID, the new message ‘‘wins’’. In any case the process propagates the winning *CHECKPOINT* or *ROLLBACK* messages to all its first-level processes, regardless of whether or not it previously propagated any losing *CHECKPOINT* or *ROLLBACK* messages.

Eventually, the step-by-step propagation of the winning session ‘‘flushes out’’ all remnants of the losing session and a consistent communication tree for checkpointing or recovery is established. The proof of this follows easily from Lemma 4.1. Since the decision is deterministic all handlers will make the same decision and all processes in the interacting set will receive a message propagating

the new decision.

While this method is fairly simple, we actually roll back more processes than is absolutely necessary — as will shortly become apparent. To get a better protocol we turn back to the proof from the previous section (Theorem 4.2). Now that more than one process can initiate a session we still have the same four cases as before, except that we can no longer assume that process q belongs to a different interacting set than p.

We examine the four cases again, where multiple checkpoint/recovery sessions may coexist in the system at the same time *and* any process has the ability to initiate a checkpoint/recovery session at any time.

Rules to handle conflicting coordinator IDs and/or sessions:

case 1: A checkpointing process p with coordinator CC_x (where p may equal x) receives a *ROLLBACK* message from a process q with:

a) a different coordinator ID CC_y, or *b)* the same coordinator ID.

case a: a different coordinator ID, and

— $q \in Buddies(p)$

Process p switches to a recovery session with coordinator CC_y and will send a *RE_ACK(child)* message to q. Switching to a recovery session requires propagating the decision (sending *ROLLBACK* messages with proper *CCID*) to all processes to whom p sent previously sent *CHECKPOINT* messages, and to the (parent) process from whom p received a *CHECKPOINT* message if $p \neq x$.

— $q \notin Buddies(p)$

Process p continues the checkpoint session.

If CC_x receives all expected *CH_ACK* messages then p will receive a *CH_FOUND* message. Process p will then send a *RE_ACK(not child)* message to q.

Otherwise, if CC_x receives a *ROLLBACK* message from a process $r \in Buddies(p)$ (case a) then p will switch to a recovery session and send a *RE_ACK(child)* message to q.

case b: with the same coordinator ID,

— Process p switches to a recovery session (with coordinator CC_x), propagates the decision, and sends a *RE_ACK(child)* message to q.

case 2: A recovering process p with coordinator CC_x receives a *ROLLBACK* message from a process q with:

a) a different coordinator ID CC_y, or *b)* the same coordinator ID.

case a: with a different coordinator ID, and

— $q \in Buddies(p)$

Process p chooses the larger coordinator ID, x or y. If CC_y ''wins'' then the switch from CC_x to CC_y is propagated to all of p's *Buddies* (except for q) via *ROLLBACK* messages tagged with CC_y. A *RE_ACK(child)* message is sent to q. If CC_x ''wins'' ID then q will initiate the coordinator switch when it receives the *ROLLBACK* message p sent.

— $q \notin Buddies(p)$

Process p simply continues with its own recovery session.

Process p will later send a *RE_ACK(not child)* message to q when it receives a *RE_FOUND* message, or, if p later receives a *ROLLBACK* message with coordinator CC_y (and $y > x$) from one of p's *Buddies* then p will switch to coordinator CC_y and send a *RE_ACK(not child)* message to q.

case b: with the same coordinator ID,

Process p simply discards all messages received from q prior to receiving the *ROLLBACK* message, sends a *RE_ACK(not child)* to q, and continues with its recovery.

case 3: A recovering process p with coordinator CC_x receives a *CHECKPOINT* message from a process q with:

a) a different coordinator ID CC_y, or *b)* the same coordinator ID.

case a: with a different coordinator ID, and

— $q \in Buddies(p)$

Process p ignores the *CHECKPOINT* message.

— $q \notin Buddies(p)$

Process p will join the checkpoint session after the recovery session is done, unless q later sends a *ROLLBACK* message — in which case, if p is still recovering then case 2, otherwise if p has resumed executing then p will rollback with q.

case b: with the same coordinator ID,

Process p ignores the *CHECKPOINT* message.

case 4: A checkpointing process p with coordinator CC_x receives a *CHECKPOINT* message from a process q with:

a) a different coordinator ID CC_y, or *b)* the same coordinator ID.

case a: with a different coordinator ID, and

— $q \in Buddies(p)$

Process p chooses the larger coordinator ID, x or y. If CC_y ''wins'' then the switch from CC_x to CC_y is propagated to all of p's *Buddies* (except for q) via *CHECKPOINT* messages tagged with CC_y. A *CH_ACK(child)* message is sent to q. If CC_x ''wins'' ID then q will initiate the coordinator switch when it receives the *CHECKPOINT* message p sent.

— $q \notin Buddies(p)$

Process p simply continues with its own checkpoint session.

Process p will later join q's session as q's child after p finishes its current checkpoint session unless q and p's interacting sets later merge (e.g. if p later receives a *CHECKPOINT* message with coordinator CC_y (and $y > x$) from a process $r \in Buddies(p)$ then p will switch to coordinator CC_y and send a *CH_ACK(not child)* message to q).

case b: with the same coordinator ID,

Process p records all messages received from q prior to receiving the *CHECKPOINT* message, sends a *CH_ACK(not child)* to q, and

continues with its checkpoint session.

In general, if processes p and q belong to sessions with the *same* coordinator ID, then p and q belong in the same interacting set. In such a case, a checkpointing process p receiving a *ROLLBACK* message from q (with the same coordinator ID) will switch over to a recovery session; a recovering process p receiving a *ROLLBACK* message from q will discard all messages previously received from q and send a *RE_ACK(not child)* message to q; a recovering process p receiving a *CHECKPOINT* message ignores that message (since q will switch over to a recovery session); and a checkpointing process p receiving a *CHECKPOINT* message saves all messages previously received from q and sends a *CH_ACK(not child)* message to q.

If processes p and q belong to sessions with *different* coordinator IDs, then it must be determined whether or not p and q belong to the same interacting set. If process p receives a *ROLLBACK/CHECKPOINT* message from q and q is in p's view of the interacting set ($q \in Buddies(p)$) then p and q clearly belong to the same interacting set. Thus, if p and q belong to the same type of session, then the larger coordinator ID is chosen and p or q (whichever one belonged to the losing coordinator's session) takes steps to flush out the smaller coordinator ID by resending the *ROLLBACK/CHECKPOINT* messages with the winning coordinator ID. If p and q are in different types of sessions then the recovery session with ''win'' out over the checkpoint session.

Otherwise, if q is *not* in p's view of the interacting set ($q \notin Buddies(p)$) then p continues with its checkpoint/recovery session. If some other process r which is in q's interacting set (where r could be q) and some process s which is in p's

interacting set (where s could be p, but *NOT* ($s=p$ *AND* $r=q$)) are in each others view of the interacting set then processes s and r will cause the two interacting sets to merge into one. However, if there are no such processes s and r then p will eventually receive the *CH_FOUND/RE_FOUND* message that it expects and, knowing that p and q are in separate sessions, will send a not child acknowledgment to q (or vice versa) and simply continues with the session. Note: ''will eventually receive the *CH_FOUND/RE_FOUND*'' will happen only if deadlock does not occur. This deadlock problem (and our solution) is discussed in Section 4.5.3.

This protocol may be somewhat difficult to comprehend because of the subtle difference between one process being in another process' *view* of the interacting set and two processes actually being in the same interacting set. The complexity stems from the existence of the following situation: a process p is in process q's view of its (q's) interacting set ($p \in Buddies(q)$) while q is not in p's view of its (p's) interacting set ($q \notin Buddies(p)$) and we may later decide that p and q are *not* in the same interacting set. The important point to note is that we *cannot* simplify the problem by saying ''if one process is in the view of another then the two processes are in the same interacting set''.

It is easiest to illustrate this through an example. Assume we have the aforementioned situation and that p and q are both in the midst of checkpointing. Further there exists no processes r and s (from p and q's interacting sets, respectively) which are both in each others view of the interacting set. In our protocol the interacting sets never merge — p's checkpoint session completes and then p joins q's checkpoint session. Why can't the checkpoint sessions of p and q

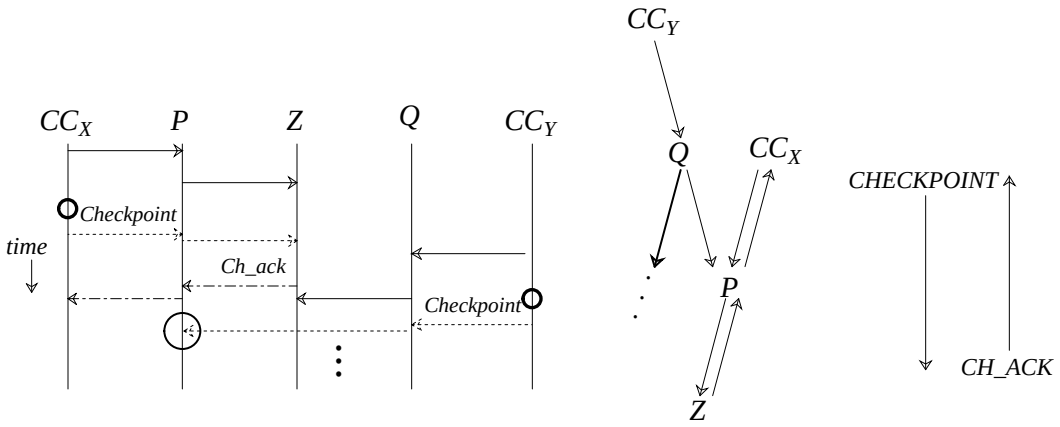


Figure 4.1: Should process p join q's checkpoint session, after its own session has finished, or should the two sessions merge into one? This figure is an example of why two sessions should not always be merged. If the sessions do merge then CC_x has no way of knowing when the interacting set is complete.

simply always be merged instead of p later joining q's session? The reason is as follows — in Figure 4.1 p has received *CH_ACK* messages from its only child z and has sent a *CH_ACK* message to its parent (and coordinator) x. Recall that q is not in process p's view of the interacting set, hence p did not send a *CHECKPOINT* message to q and does not wait for a *CH_ACK* message from q. At the time that x receives the *CH_ACK* message from its only child it expects that the interacting set is now complete. If p were to allow q's interacting set to merge with its own at this time then p has expanded its interacting set from processes p, x, and z to include q and all the processes with which q has communicated. The coordinator, x, must be informed of this expansion because the expanded interacting set is not complete since q has not finished receiving *CH_ACK* messages from its children. Hence x does not know when the interacting set is finally complete and therefore checkpointing will not complete. To this end, the *CH_ACK* message must be a guarantee that no more *new* processes will join the

interacting set hence the problem is solved by not allowing q to join p's checkpoint session - instead p will later join q's session.

4.4. Synchronous Coordinated Checkpointing

In the previous section we have shown how checkpointing and recovery of interacting sets of processes can be coordinated in a distributed manner, where any process can coordinate (or attempt to coordinate) a checkpoint/recovery session at any time, and a checkpoint/recovery session can be initiated at any time. With the technique presented, checkpoint/recovery sessions will be merged into a single session, with all but one coordinator eliminated. Further, any number of checkpoint and recovery sessions can be active in the system at a time.

In this section the complete synchronous checkpointing scheme is presented, including a detailed communication model whose use relaxes the assumption made in section 4.1.4 that channels are directed, FIFO, error-free, static, and transmit messages with arbitrary finite delay. In this section and the section on recovery (Section 4.4) the technique by which the checkpoints (of an interacting set of processes) are atomically committed to stable storage will be detailed. The resulting scheme is then robust to the ‘‘simultaneous’’ failure of an arbitrary number of processes.

4.4.1. Supporting Dynamic Virtual Channels and Processes

In this section a simple model for communication between processes using virtual channels is presented. This model employs the error detection methods discussed in section 2.4, and is, as a result, able to provide directed, FIFO, error-free channels, which can be dynamically created and destroyed, at very low overhead. The communication between processes using virtual channels can be modeled as follows:

- Each process p has some number of *open* unidirectional virtual channels of which p is either source or destination. We assume that source and destination addresses include the unique address of the node on which the process currently resides.
- Each end of a virtual channel has a FIFO message queue and a *signature* associated with it. This signature can be stored in a linear feedback shift register as discussed in Section 2.4. We denote all outgoing queues as $SendQ_p[q]$ with signature $Ssig_p[q]$, where p and q are the source and destination processes of the channel, respectively. All incoming queues are denoted $RecvQ_p[q]$ with signature $Rsig_p[q]$, where p and q are the destination and source processes of the channel, respectively.
- A *dirty bit* is associated with each end of the virtual channel. The dirty bit is reset at the end of a checkpoint/recovery session and set when a message is sent/received on the channel.
- When the channel is first opened, via a special ‘‘create circuit’’ message routed hop-by-hop from the source process to the destination process, the two signatures at the ends of the channel must be initialized to the same value, the

channel must be empty, and the *SendQ* and *RecvQ* associated with the channel must also be empty.

- When a channel is closed a special ‘‘destroy circuit’’ message is sent down the channel (from source to destination) which flushes any messages in transit on the channel to the destination process. This message carries the source signature of the channel which is compared to the destination signature of the channel. If a mismatch is found then recovery is initiated (see Section 4.4). Otherwise, if the channel has been used since the last checkpoint session involving the source and destination processes (i.e. the dirty bits are set), then the channel is put on a list of *closed* channels at both the source and destination processes. Thus the *dynamic communication information* (*Buddies* list) of a process is determined from all the dirty open and closed channels it has. The list of closed channels associated with a process is destroyed at the end of that process' next checkpoint.
- When a process p executes a $SEND(M, dst)$ the message M is appended to $SendQ_p[dst]$. When a process p executes a $RECEIVE(M, dst)$ the message M is removed from the head of $RecvQ_p[src]$. A process can receive a message from a specific destination or on a first-come-first-served basis and may send a specific message to more than one process.
- A MsgHandler, one per node, is responsible for handling the actual sending and receiving of messages at a node. The MsgHandler periodically sends messages in some ‘‘fair’’ way by removing a message M from the head of some $SendQ_{src}[dst]$, updating the corresponding $Ssig_{src}[dst]$, and sending the message out on the link. When a message M arrives at a node MsgHandler

appends M to the appropriate $RecvQ_{dst}[src]$ and updates $Rsig_{dst}[src]$. The use of LFSRs (linear feedback shift registers) allows the updating of *Rsig* and *Ssig* to take place in parallel with message receipt and transmission. If the arriving message is a *fail-safe* message (i.e. a recovery or checkpointing message, see Section 2.4), then, after it is determined to be free of errors, it is delivered to the appropriate checkpoint or recovery handler.

- The checkpoint or recovery handler of some process p sends its fail-safe messages to the handler of some process q by appending them to $SendQ_p[q]$. Any fail-safe messages, when sent (or received) off-node by MsgHandler, are not used to update the signature(s) associated with that channel. Instead MsgHandler either examines or appends error correcting codes onto the fail-safe messages before delivering or sending them, respectively.

In Section 4.1 we assumed that all communication took place over static channels that guaranteed delivery of error-free messages in FIFO order with arbitrary finite delay. The model we have now adopted is more realistic. Channels between processes are created and destroyed dynamically and at any time. However, when a channel is destroyed sufficient information is kept around until the next checkpoint session involving the processes on either end of the channel. We have presented a low overhead error detection mechanism which can be used for communication between application processes. Lost messages, including those which were in transit through a node when that node fails, are detected during the next checkpoint session involving the sender and intended receiver processes. Recovery is then initiated at that time (see Section 4.4).

The same model can be used in a system employing *packet-switching*, instead

of virtual channels, with a couple changes. Application messages are routed hop-by-hop from source to destination, hence no ‘‘create/destroy circuit’’ messages are used. Also, instead of using dirty bits, a *list* of processes with whom direct communication has taken place must be maintained between checkpoints.

As previously discussed, every process that can fail must have an initial state stored on stable storage before execution can begin. Furthermore, every process must take a checkpoint upon termination. This is necessary to ensure that the process has executed correctly, since immediate error detection is not performed. It is also necessary in order to ensure that other processes, which have communicated with the terminating processes since its last checkpoint (i.e. processes in the terminating processes interacting set), will have a checkpoint to which they can correctly roll back after that process has terminated. Once this checkpoint has taken place, then all traces of the process including the checkpoint just taken can be erased. As an optimization the checkpoint of a terminating process need not actually be physically sent to disk. Instead a ‘‘dummy entry’’ can be maintained by the disk server for the new checkpoint. When the dummy entry becomes ‘‘valid’’ all entries relating to the corresponding process are removed.

4.4.2. Synchronous Checkpointing of Interacting Sets

Checkpointing is triggered by a ‘‘checkpointing timer,’’ which causes the kernel to spawn a *handler* process. This handler begins its operation assuming that it will *coordinate* a checkpointing session for the interacting set of the process with which the timer is associated. In synchronous checkpointing, once a process becomes involved in a checkpoint session it does not execute again until its

checkpoint has been committed to disk. Checkpointing each process involves the following basic steps:

1. The process is halted. A *handler* process, that will participate in the checkpointing session on behalf of the halted process, is initiated.
2. The portion of the process state that is not in memory (e.g. contents of registers) is stored in a dedicated buffer area in local memory.
3. The process state in memory (or, as discussed in Section 4.6, just the modified pages) is sent to stable storage.
4. The interacting set/checkpoint tree is found. The communication channels between processes in the interacting set are *flushed* and any packets flushed from incoming channels are sent to stable storage as part of the checkpoint state.
5. Once all the checkpoint states, including flushed packets, of all the processes in the interacting set have been written to stable store, those states are committed and the handlers terminate. Processes then resume normal operation.

If the checkpointing session is triggered by the timer of process X, the *checkpoint coordinator*, CC_X, is responsible for sending process X's state to disk, finding the current interacting set (*checkpoint tree*) by sending *CHECKPOINT* messages (markers with error detections bits) to all *first-level processes* Y (as defined in Section 4.2.1), and checking for communication errors on all channels from processes Y to process X (as described in Sections 2.4 and 4.3.1).

Any process, Y, receiving a *CHECKPOINT* message will be suspended and a

checkpoint handler (CH_Y) will be started up in its place. This handler is responsible for copying Y's state and sending it to disk; sending *CHECKPOINT* messages to all first-level processes *except* for the coordinator/parent; and checking for any errors on all of Y's incoming channels (virtual circuits). This continues until *leaf processes* are found — processes which have communicated only with processes already a part of the tree. The checkpoint handler for each leaf process then sends a *CH_ACK* message (marker with error detections bits) to its ''parent''. Any handler receiving a second, or more, *CHECKPOINT* message (i.e. already has a ''parent'' in the checkpoint tree) receives and sends a marker and checks for communication errors but denies child status of that handler.

Once a parent of a leaf process receives *CH_ACK* messages from all of it's children, a *CH_ACK* message is sent to its parent, and so on. When CC_X receives *CH_ACK* messages from all processes to which it sent *CHECKPOINT* messages, the entire interacting set has been found. Thus, *CHECKPOINT* and *CH_ACK* messages serve the purpose of *flushing* messages to their destinations, carrying error detection bits from one end of a channel to the other for comparison, and finding the interacting set.

Once CC_X receives *CH_ACK* messages from all its first-level processes, it sends a *CH_FOUND* message down the tree, thus notifying the handlers that all messages have been flushed to their destinations and that message queues can now be sent to disk. Once these queues have been saved, *CH_DONE* messages are sent from the leaf processes to their parents, and so on. When CC_X receives *CH_DONE* messages from all its children, it ''knows'' that the checkpoints of all processes in the tree have been written to disk.

CC_X commits all processes to the new checkpoint by directing its disk node to commit to the new checkpoint of X and to destroy X's previous checkpoint. When the disk node acknowledges this operation, CC_X sends *CH_COMMIT* messages down the tree. *CH_RESUME* messages are then sent up the tree after each handler commits its process' state to disk. Each handler which sends a *CH_RESUME* flags its process as ''runnable'', clears the *first-level list* for its process (which contains the dynamic communication information — the *Buddies* list) and terminates. CC_X terminates last. Processes cannot participate in or initiate new checkpointing sessions until they receive the *CH_RESUME* message.

4.4.3. Concurrent Invocations of the Algorithms

As discussed in Section 4.2, it is possible for several processes within an interacting set to initiate checkpointing and/or recovery sessions ''simultaneously''. To solve this problem a single coordinator is deterministically chosen to coordinate the session. To this end, *CHECKPOINT* and *ROLLBACK* messages contain checkpoint/recovery coordinator IDs and all handlers locally store their current coordinator ID. Since there is a total ordering of node and process identifiers, a process receiving *CHECKPOINT* or *ROLLBACK* messages originating from different coordinators can pick the coordinator with the ''largest'' ID.

As discussed in Section 4.2.1.2 a simple protocol would be to always choose recovery over checkpointing: i.e. *ROLLBACK* messages always win over *CHECKPOINT* messages regardless of the coordinator ID. Specifically, a *ROLLBACK* message will cause a checkpointing handler to immediately stop all

activities related to checkpointing and join the recovery session. Since the first-level list is not cleared until the process has completed all phases of a checkpointing session, the recovery session is guaranteed to include all the handlers in the ongoing checkpointing session that may otherwise be waiting indefinitely for checkpointing-related responses. If a change in coordinator or session type is made then the handler propagates this decision to its first-level processes by resending *CHECKPOINT/ROLLBACK* messages with the new coordinator ID. Eventually the winning coordinator will ‘‘flush out’’ all remnants of the losing session.

However, a better protocol, discussed at length in Section 4.2.1.2, switches from checkpointing to recovery only when the process which has sent the *ROLLBACK* message has been positively determined to be part of the receiving process’ interacting set. This may involve postponing the response to the *ROLLBACK* message until more information is gathered (see Section 4.2.1.2). While this protocol is more complex, only the processes which *must* be rolled back will be rolled back.

4.4.4. Checkpointing with Packet-Switching

In a packet-switching environment messages are routed on a hop-by-hop basis over virtually any path in the system. Flushing of messages in transit during checkpointing is more complex since the entire system must be flushed. This requires temporarily stopping all the processes in the system from generating new messages while existing messages in transit are flushed to their final destinations and checks are performed on *every* communication link. As described by Tamir

and Gafni[Tami87], this is a classic distributed termination problem and the solution we use here is the same one used in [Tami87], which is derived from [Shav86, Dijk80].

4.5. Recovery

A recovery sessions is initiated when an error is detected. If there is a mismatch between the two signatures at the ends of a virtual circuit, a single interacting set needs to be rolled back: the set containing the two processes on either end of the virtual circuit. When an error in a node's outputs is detected by a neighboring node [Tami83], recovery may involve multiple interacting sets since all processes executing on the node and their interacting sets must be rolled back. Any messages lost due to the node's failure will be detected later when a signature mismatch occurs.

4.5.1. Recovery From Communication Errors

The algorithm to recover from a communication error is similar in structure to the *synchronous* checkpointing algorithm, where recovery coordinators and handlers replace their checkpointing counterparts. When a signature mismatch occurs, the kernel initiates rollback by spawning a *recovery coordinator* process. A *recovery tree* is created by propagating ROLLBACK messages which are acknowledged by RE_ACK(CHILD/NOT_CHILD) messages just as CHECKPOINT and CH_ACK messages are used. Unlike checkpointing, no signature comparisons are made, and all messages which are flushed to their destinations are discarded (this takes care of the recovery *livelock* problem

discussed in [Koo87]). For each process Y in the interacting set, the associated recovery handler RH_Y requests the process' state from the appropriate disk node. When a handler receives RE_DONE messages from all its children and its entire process state and associated message queues, it sends a RE_DONE message to its parent. When the coordinator receives its checkpointed state and RE_DONE messages from all its children, it sends RE_RESUME messages to it's children, marks its associated process ''runnable'', and terminates. Upon receipt of a RE_RESUME, RH_Y forwards the message to its children, terminates, and process Y resumes normal processing. No ''commit'' or ''resume acknowledgment'' phases are needed.

4.5.2. Recovery From Node Failures

When a neighbor detects that a node is faulty, *node-level recovery* is initiated. We assume that there is a reconfiguration algorithm that supplies the recovery algorithm with destination node(s) to which processes are to be restored. Since no information is available from the failed node(s), recovery trees must be constructed based on information available outside the failed node(s). An additional phase is added to the general structure of the algorithms to collect the missing information — namely, the list of first-level processes (*Buddies list*, see Procedure *Rback* in Section 4.1.5). In this phase, each neighbor which has detected the node failure starts up a Recovery Initiator Process (RIP_Y where Y is the *node* ID on which RIP_Y is running). The tasks performed by this process are: determining which processes were on the failed node(s), determining the first-level processes for each failed process, and starting Recovery Coordinators (phase two) for *each failed process*.

Phase two is identical to recovery from communication errors.

In order to determine which processes were on the failed node we require that the immediate neighbors of each node keep track (a list) of all processes running on the node. It should be noted that the disks must have lists of all processes in the system (the union of all the disk server process tables, see Figure 4.2 in the next subsection). Hence, if multiple nodes should fail simultaneously we can determine which processes are still in existence on the system and hence, which processes have failed. After determining which processes have failed, RIP_Y needs to get information equivalent to first-level process lists, for *each failed process*, before tree construction can begin. This is done by broadcasting RECOVERY messages (containing a list of failed processes) to all working nodes in the system. Each node returns a ‘‘FIRST-LEVEL message’’ which contains a list of processes on that node who have communicated directly with processes on the failed node (*Buddies list*). With this information all the required recovery trees can be constructed and recovery can proceed as in the previous subsection.

4.5.3. Handling Failures during Checkpoint Sessions

As discussed in Section 4.3.3, a recovery session has precedence over a checkpointing session. Specifically, if a recovery session has been chosen to supersede checkpointing, the checkpoint handlers stop all checkpointing activity and join the recovery session. For example, if the checkpointing coordinator receives a ROLLBACK message while waiting for CH_DONE messages from its children, it never sends a CH_COMMIT but instead immediately forwards the ROLLBACK message to all its children.

Each disk node has a disk server process that saves and restores checkpoints from the disk. The server process maintains a table that contains the status of the checkpoint of each process whose state is stored on the disk (Figure 4.2). Normally, a process has one entry, or checkpoint. During a checkpoint session, a second entry is made as the new checkpoint arrives at the disk node. This entry is invalid until the last state packet is received.

Disk Server Process Table

	PID	*Valid*	*New*	*Current CCid*	*Disk Address*	*Pointer*
A)	*F*	1	0	*B*		
B)	*F*	1	0	*B*		
		0	1	*A*		
C)	*F*	1	0	*B*		
		1	1	*A*		
D)	*F*	1	0	*A*		

Figure 4.2: The table maintained by the disk-server process on the disk node. A) shows an entry for process F between checkpoint sessions. Process B was F's previous checkpoint coordinator. The disk address field points to the location on disk of F's checkpoint. B) shows process F in the midst of a checkpoint session coordinated by process A. In C) F has two valid checkpoints and, in D), the newest state has been committed.

Each process has a *version* variable stored on the node where it is executing[Tami84]. When an error is detected this variable is used to determine

which version of the process checkpoint on disk should be used. The version variable has three possible values: *known, old,* and *unknown.* During normal operation, the version is always ‘‘known’’, meaning that there is only one valid checkpoint on disk. When the handler begins sending a new checkpoint state of a process to disk, the version variable changes to ‘‘old’’. When message queues are sent to disk, the handler changes the version to ‘‘unknown’’ and waits for a CH_COMMIT message. When the CH_COMMIT is received the handler sends a COMMIT message to the disk node. When the disk’s acknowledgment is received, the version is changed back to ‘‘known’’.

The version variable associated with the checkpoint *coordinator* never changes since the version is always ‘‘known’’, i.e., there is never more than one *valid* checkpoint state on disk. This means that the checkpoint coordinator never passes through step C in Figure 3, but moves directly from B to D — effectively committing the entire interacting set to the new checkpoint. This makes the checkpoint algorithm robust to failures. If any or even all processes in the interacting set fail, recovery coordinators look up the disk entry for the process being recovered — if there are two valid states for that process then the coordinator’s disk entry is used to determine which state to roll back to. If the coordinator’s entry has two states on disk, one valid and one not, then the process rolls back to its older checkpoint, while if there is only one valid entry the the process rolls back to its newer checkpoint.

4.6. Asynchronous Coordinated Checkpointing

Synchronous checkpointing of interacting sets requires that all processes (in the interacting set) be suspended for the duration of the checkpoint session. In the worst case the entire system may have to be suspended as in Global Checkpointing[Tami84]. The key feature of *asynchronous* checkpointing is that interruption of normal processing is minimized: the normal execution of a participating process is suspended only as long as it takes to copy its state in local memory (see Section 4.6 for further optimizations). Each process in the interacting set is ‘‘checkpointed’’ in *local volatile* memory and then resumes normal operation while the volatile checkpoints are copied to stable storage. Previous work has focused on optimizing the two-phase commit protocol in order to possibly reduce the length of time the process has to suspended[Kim93]. We eliminate the concern of overhead incurred by the two-phase commit structure of the checkpointing algorithm by making checkpointing activities asynchronous of process execution.

Checkpointing each process involves the following basic steps:

1. The process is halted. A *handler* process, that will participate in the checkpointing session on behalf of the halted process, is initiated.
2. The portion of the process state that is not in memory (e.g. contents of registers) is stored in a dedicated buffer area in local memory.
3. The process state in memory (or, as discussed in Section 4.6, just the modified pages) is copied to a dedicated buffer area for later transfer to stable storage.

4. *CHECKPOINT* messages are queued up for transmission to all neighbors (*Buddies*). (Any subsequent incoming messages will be delivered to the process *and*, possibly, saved as part of the checkpoint state.)

5. The process resumes normal operation.

6 The *handler* continues with the checkpointing session. Any outstanding packets in the communication channels to/from neighbors are flushed, the interacting set is identified, consistency of checkpoint and current states is maintained, and the coordinated checkpoint is committed to stable storage.

As in the synchronous checkpointing algorithm, checkpointing is triggered by a ‘‘checkpointing timer’’. The checkpoint handler assumes that it will coordinate the checkpoint session for the interacting set of processes to which it belongs. Processes that are not part of the interacting set do not participate in the checkpoint. The length of time that processes in the interacting set are suspended is *independent* of network delays *and* the number of processes that constitute the application. The cost is increased complexity in the initial phases of the checkpointing algorithm — until the interacting set has been found.

The checkpointing algorithm, with coordinator x, from the perspective of process p (p and x may be the same process), is as follows:

when either —

a) the checkpoint timer for process p expires ($x = p$) *or*

b) p has received its *first* $CHECKPOINT^x$ message from x or from $y \in Inter(x)$.

the following steps are taken:

i) p takes a checkpoint (C_p) (steps 2 and 3 above).

ii) The *Buddies*(p) list is formed: $q \in$ *Buddies*(p) if, and only if, process p has either sent a message to q or received a message from q since the last checkpoint of p.

iii) $CHECKPOINT^x$ messages are sent (queued) to all $q \in$ *Buddies*(p), except p's parent (if $p \neq x$). If $p \neq x$ then a CH_ACK^x message is sent (queued) to p's parent.

When any *marker* (*CHECKPOINT*, *CH_ACK*, or *SUBTREE–NC* message) is received by p from z, $MARKER_p[z]$=*TRUE*.

Until the interacting set has been found (step vii), if a *marker* has *NOT*:

a) been received from process z, then any application messages delivered to p (into $RecvQ_p[z]$, as discussed in Section 4.3.1) must also be saved in $MQ_p[z]$, or

b) been sent to process z, then p may queue messages to any z ($\notin$ *Buddies*(p)) but they must not be transmitted until a marker is sent or step vii is reached, whichever occurs first.

This step is critical to ensuring that the checkpoints of p and z will be consistent.

iv) ** p resumes normal execution.

v) p's handler now waits for CH_ACK^x and $SUBTREE^x$ messages from all $q \in$ *Buddies*(p), except for p's parent. *SUBTREE* messages are *SUBTREE-NC* (not-child) or *SUBTREE-C* (child).

vi) Once the *SUBTREE* messages have all arrived, $p \neq x$ sends a $SUBTREE-C^x$ message to its parent. If $p = x$ then the interacting set has been found and all in-transit messages have been flushed. Now, for each z from which a marker was received, $p = x$ saves $MQ_p[z]$ with C_p, and sends CH_FOUND^x messages to its children.

vii) Upon receipt of CH_FOUND^x from the parent, $MQ_p[z]$, for all z from which a marker has been received, must be saved with C_p. The CH_FOUND^x is forwarded to the children and p's handler waits for a) an acknowledgment from disk that the new checkpoint has been saved and b) CH_DONE^x messages from the children — indicating that *their* checkpoints have also been saved.

viii) Once the *CH_DONE* messages have all arrived *and* C_p is acknowledged to be stored on stable storage, p's handler sends a CH_DONE^x to its parent. If $p = x$ p directs its disk node to *commit* the most recent checkpoint and then sends CH_COMMIT^x messages to its children.

ix) Upon receipt of CH_COMMIT^x message from the parent, the handler directs the disk to commit the current checkpoint and CH_COMMIT^x is forwarded to the children. p's handler now waits for CH_RESUME^x messages from the children — indicating completion of their checkpoint handlers. x terminates once it receives the last CH_RESUME^x.

The *CHECKPOINT* and *CH_ACK* messages are used to locate the members of $Inter(x)$ and to ensure that the checkpoints of each pair of processes (p,q), where $p,q \in Inter(x)$, will be buddy-consistent. *SUBTREE* messages (child and not-child) are primarily used to complete the communication tree. *CH_FOUND*

messages, which travel down the tree, indicate that the tree is complete, the interacting set has been found, and that all messages have been flushed to their destinations (i.e. all the checkpoint states, *with MQ*, are buddy-consistent, and therefore form a consistent process set). Once stable storage has acknowledged receipt of each of the new checkpoints *CH_DONE* messages travel up the tree. The coordinator instructs the disk to commit to the new checkpoint (erasing the old) and then sends *CH_COMMIT* messages down the tree so that the rest of the interacting set will commit to their checkpoints. The checkpoint handlers terminate as *CH_RESUME* messages flow up the tree.

In synchronous coordinated checkpointing[Tami89, Koo87] processes do not resume normal operation and thus can not interfere with the *CHECKPOINT*, *CH_ACK*, and *SUBTREE* phases of the checkpoint session. In the proposed asynchronous coordinated checkpointing algorithm processes resume normal operation *prior* to these phases — increasing their complexity significantly. Important algorithmic details will be addressed in the following subsections: first we will look in more detail at the *CHECKPOINT*, *CH_ACK*, and *SUBTREE* phases of asynchronous checkpointing when 1) only one process (x) in the interacting set initiates (and coordinates) the checkpoint session and 2) only one checkpoint session is active in the system. In subsequent sections we will relax constraint 1) and then 2). At the end of the *CHECKPOINT*, *CH_ACK*, and *SUBTREE* phases the checkpoints of the interacting set form a consistent process set and the rest of the checkpoint session is simply the (robust) transfer of the checkpoint states to stable storage as discussed in Section 4.4.

4.6.1. Asynchronous Checkpointing: 1 Coordinator, 1 Session

In this section there is only one active checkpoint session with one coordinator, x. From the perspective of a process $p \in Inter(x)$ (where p and x may be the same process), every process q falls into one of the following three cases:

A) $q \in Buddies(p)$ and $q \in Inter(x)$:

Since q is a buddy of p, p has sent a marker, $CHECKPOINT^x$, to q. Step iii above and reliable transmission of recovery messages ensure that p will receive a marker from q.

When p receives either a 1) $CHECKPOINT^x$ or 2) CH_ACK^x then $MARKER_p[q]=TRUE$ and $MQ_p[q]$ is saved with C_p. In the first case p and q have ''simultaneously'' sent *CHECKPOINT* messages to one another. In order to create the interacting set tree, p declines child status with q by sending a $SUBTREE-NC^x$; q will do the same. In the second case q is a child of p thus p will receive a $SUBTREE-C^x$ message from q in the future.

B) $q \notin Buddies(p)$ and $q \in Inter(x)$:

This case is only interesting if q sends an application message to p before taking C_q (as part of $Inter(x)$) and that message is received by p after C_p is taken. Any such message(s) are saved in $MQ_p[q]$ (step iii-a above). When q takes C_q q sends $CHECKPOINT^x$ to p. Upon receipt of the $CHECKPOINT^x$ message p sets $MARKER_p[q]=TRUE$ and saves $MQ_p[q]$ with C_p. Note that since $q \notin Buddies(p)$, any messages sent by p to q have been blocked from leaving the node (step iii-b above). $SUBTREE-NC^x$ is sent to q and *then* any blocked messages to q are released.

C) $q \notin Buddies(p)$ and $q \notin Inter(x)$:

When p receives CH_FOUND^x from its parent any messages in $MQ_p[q]$ are discarded and any blocked messages to q are released.

This section solves the basic problems associated with ensuring that the checkpoint states of q and p are consistent. First, MQ is used to capture messages in transit ‘‘between checkpoints’’ so that, in the consistent global checkpoint state, all messages sent have been received (see example message $m1$ in Figure 4.3). Second, p has to be prevented from sending messages to any $q \notin Buddies(p)$ until either a) markers have been exchanged between p and q or b) p receives CH_FOUND^x from its parent. If q is allowed to receive messages from p where $q \notin Buddies(p)$ then p and q could end up with inconsistent checkpoint states as depicted by example message $m2$ in Figure 4.3. We show in Section 6.2 that very few messages needed to be blocked in our simulation runs.

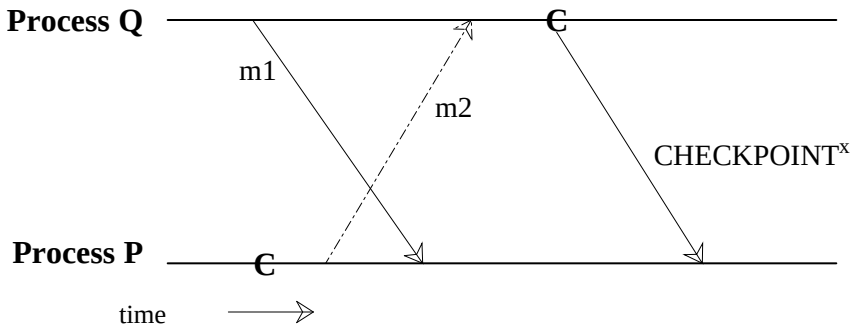


Figure 4.3: Inconsistent checkpoints. No communication between p and q occurred before p’s checkpoint, thus q joined the checkpoint session as the buddy of some other process $r \in Inter(x)$. If q processes $m2$ before taking its checkpoint then q’s checkpointed state has received a message which p’s checkpoint never sent. This case is prevented from occurring by delaying transmission of $m2$ to q. Note that message $m1$ will correctly be saved in $MQ_p[q]$.

4.6.2. Asynchronous Checkpointing: N Coordinators, 1 Session

In this case multiple checkpoint handlers compete to coordinate the same checkpoint session. Again every process q falls into one of the following three cases:

A) $q \in Buddies(p)$ and $q \in Inter(x)$:

Besides 1) and 2) from the previous subsection, p could receive a 3) $CHECKPOINT^y$ from q. First, p sets $MARKER_p[q]$=$TRUE$ and saves $MQ_p[q]$ with C_p. If $y > x$ then q becomes p's new parent and p sends $CHECKPOINT^y$ to all $Buddies(p)$, except q, thereby propagating the change in coordinator. If $y < x$ then p does nothing. (Since $q \in Buddies(p)$, q will switch to coordinator x when it receives the $CHECKPOINT^x$ sent by p.)

B) $q \notin Buddies(p)$ and $q \in Inter(x)$:

In addition to 1) from the previous subsection, p might receive a 2) $CHECKPOINT^y$ from q. This marker can not be ''rejected'' because $q \in Inter(x)$ and therefore p and q must checkpoint together to ensure that the global checkpoint state remains consistent. Since, in this section, there is only one active checkpoint session p knows that q must be a member of $Inter(x)$. p responds by setting $MARKER_p[q]$=$TRUE$ and saving $MQ_p[q]$ with C_p. If $y > x$ then q becomes p's new parent and p then sends $CHECKPOINT^y$ to all $Buddies(p)$, except q. If $y < x$ p sends a $CHECKPOINT^x$ to q — forcing q to change coordinators.

C) $q \notin Buddies(p)$ and $q \notin Inter(x)$:

Same as in the previous subsection. Since we have only one checkpoint session active in the system (and q is not part of it since $q \notin Inter(x)$) then p can not

receive a $CHECKPOINT^{y}$ from q.

This section shows the basic mechanism for handling multiple processes attempting to coordinate *the same* checkpoint session. By imposing a total ordering amongst all process IDs we can guarantee that only one coordinator will ‘‘win’’ and that the checkpoint session will proceed.

Since asynchronous coordinated checkpointing allows processes to resume execution at the beginning of the checkpoint session it is essential that the very first *CHECKPOINT* message received from any process q, regardless of coordinator ID, result in *MARKER* being set to true. This first marker indicates that any previously-received messages were sent before q took its checkpoint and that any messages subsequently received were sent after q’s checkpoint. Once the marker has been received then any subsequent messages sent by q do not need to be saved in $MQ[q]$.

4.6.3. Asynchronous Checkpointing: N Coordinators, N Sessions

In this section we relax all constraints and allow multiple checkpoint handlers to compete to coordinate multiple checkpoint sessions. Every process q falls into one of the following three cases:

A) $q \in Buddies(p)$ and $q \in Inter(x)$: Same as in ‘‘N’’ Coordinators, 1 Session.

B) $q \notin Buddies(p)$ and $q \in Inter(x)$ and

C) $q \notin Buddies(p)$ and $q \notin Inter(x)$:

Now we can have the case that $q \in Inter(y)$ and $y \notin Inter(x)$. p can not tell the difference between B) and C) with its local $Buddies(y)$ information. After

receiving an application message from q p might receive either a 1) $CHECKPOINT^x$ or 2) $CHECKPOINT^y$ from q. If a $CHECKPOINT^x$ message is received then q is clearly in $Inter(x)$ and p responds as in 1 Coordinator, 1 Session, part B. If a $CHECKPOINT^y$ message is received p still does not know if $q \in Inter(x)$ or if $q \notin Inter(x)$. As discussed in the previous section p can not ''reject'' (negatively respond to) the $CHECKPOINT^y$. [Note: rejection is the approach taken in the synchronous algorithm presented in Koo and Toueg[Koo87] which must also deal with conflicting checkpoint sessions. We found, in our simulations, that this can result in many checkpoints never completing — creating too much ''useless'' overhead.]

Also, *unlike* the previous section p can not just positively respond to q's $CHECKPOINT^y$ either. The reason is as follows: Since $q \notin Buddies(p)$, p is not ''expecting'' a marker from process q. Therefore p can enter step vi of the algorithm and send a $SUBTREE-C$ to the parent. In effect p is telling its parent that ''the interacting set (tree) is complete from my perspective''. But, p might now receive a $CHECKPOINT^y$ from q. If p responds positively to this message then, if $q \notin Inter(x)$ p is *expanding* the interacting set to include q, thus ''violating'' its $SUBTREE-C$ message. With this approach p's coordinator has no way of knowing when the interacting set is complete. [Alternatively, a termination detection algorithm could be used to determine when the interacting set has been found. We decided this was too costly in terms of the time required to run the algorithm and the number of messages required.]

The last choice is for p to defer handling of $CHECKPOINT^y$. Several things can happen:

i) q changes to coordinator x and follows the $CHECKPOINT^y$ message with a $CHECKPOINT^x$. p then responds as in 1 coordinator, 1 session, part A.

ii) p changes to coordinator y and then responds to q as in 1 coordinator, 1 session, part A.

In i) and ii) q is clearly a member of $Inter(x)$.

iii) If $q \notin Inter(x)$ then p receives a CH_FOUND^x from its parent. At this time p finally knows that $q \notin Inter(x)$: p completes its current checkpoint and afterwards joins q's checkpoint as q's child.

iv) If $q \in Inter(x)$ deadlock can occur, as shown in Figure 4.4. To prevent deadlock we add the following to our algorithm. When p, checkpointing with coordinator x, decides to defer handling of a $CHECKPOINT^y$ message, p sends two messages — one to each coordinator — relaying the fact that p is not responding to a $CHECKPOINT^y$ message. If, for example (see Figure 4.4), coordinator x receives *two* messages: one from a process $r \in Inter(y)$ indicating that r can not respond to a $CHECKPOINT^x$ message and one from a process $p \in Inter(x)$ indicating that p can not respond to a $CHECKPOINT^y$ message, then x knows that $Inter(x)=Inter(y)$. If $y>x$ then x changes coordinators, making y its parent and sending $CHECKPOINT^y$ messages to all $Buddies(x)$. If $y<x$ then x does nothing — knowing that y will switch coordinators to x and become x's child.

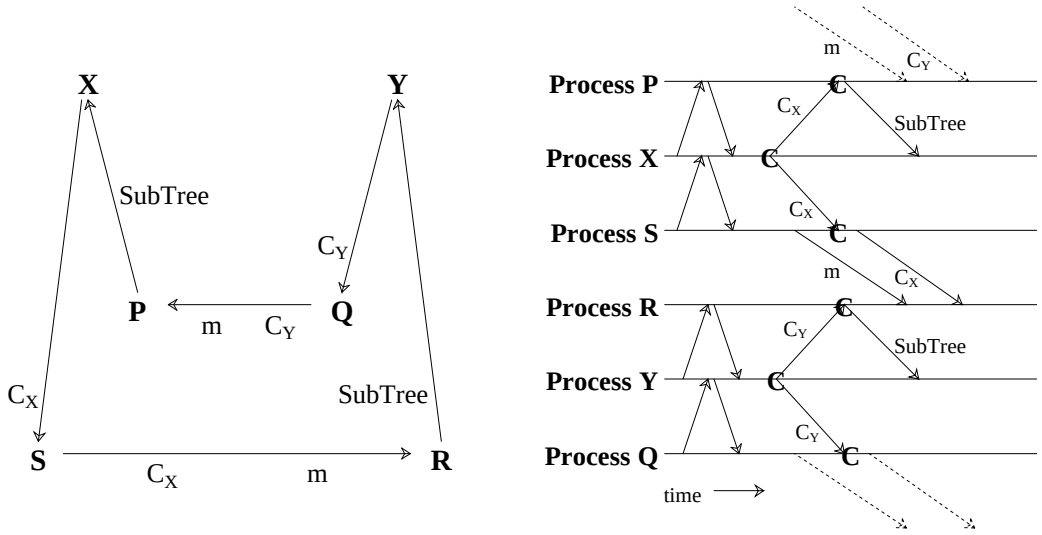


Figure 4.4: Deadlocked checkpoint session. The cycle is depicted on the left and the details of the example on the right. The problem is that process Q can not send a $SUBTREE-C$ message to Y without receiving a $SUBTREE$ from P. The same situation exists for S. P (and R) can not respond because they have already informed their respective coordinators that their portion of the interacting set (tree) is complete. Thus the algorithm is deadlocked and can not proceed.

4.6.4. Summary

We have now finished detailing the checkpointing algorithm for the general case. At the end of the *CHECKPOINT*, *CH_ACK*, and *SUBTREE* phases the checkpoints of the interacting set form a consistent process set. The robust transfer of these states to stable storage was addressed in Section 4.4. In coordinated checkpointing algorithms processes not part of the interacting set do not participate in the checkpoint sessions. In the asynchronous algorithm the length of time that a process is suspended is independent of network delays and the number of processes in the interacting set. This is similar to algorithms that checkpoint processes independently except that processes in our algorithm will, for a given checkpoint

interval, checkpoint more often as they participate in checkpoint sessions that they did not initiate. This can be tuned by increasing the checkpoint interval so that, on average the *actual* checkpoint interval is the desired checkpoint interval. Section 6.2 contains a performance comparison of independent and coordinated checkpointing, in a failure-free system.

The cost of asynchronous coordinated checkpointing compared to synchronous is the need to (possibly) delay some messages from being transmitted and an increase in the complexity of the *CHECKPOINT*, *CH_ACK*, and *SUBTREE* phases of the checkpoint algorithm. We have proven the correctness of synchronous coordinated checkpointing and have focused our efforts on implementing the asynchronous algorithm on a parallel simulator.

4.7. Minimizing the Cost of Checkpointing Using the Virtual Memory System

Asynchronous coordinated checkpointing is amenable to several optimizations that can dramatically decrease the amount of data that must be moved during checkpointing as well as the local memory space needed for the volatile checkpoints. The key to these optimization is to use an enhanced virtual memory system. This system will help the scheme in two ways: 1) it will identify the pages that have been modified since the last checkpoint and thus eliminate the need to checkpoint pages that have not changed, and 2) it will allow the volatile checkpoint to be taken by simply copying page table entries without actually moving data in local memory unless it becomes necessary.

The first optimization requires adding a special ‘‘dirty bit’’ to each entry in the page table. This dirty bit is cleared when a process is checkpointed and is set

whenever there is a *write* to the page. When a page is first allocated, is is marked as ‘‘dirty.’’ During checkpointing, only dirty pages that are in local memory need to be physically copied to stable storage for checkpointing. Pages that are ‘‘clean’’ according to this special dirty bit have not changed since the last checkpoint and are included in the new checkpoint in stable storage using simple pointer manipulation (there is no need to move the page from the physical location in stable storage where it was stored for the previous checkpoint).

In standard virtual memory systems, page table entries include control bits that mark the page as *read-only*, *read-write*, *execute-only*, etc. The time to take the volatile checkpoint as well as the space required for it in local memory can be cut dramatically by adding another type of page: *copy-on-write* [Bobr72]. Taking a volatile checkpoint involves copying the page table entries of resident dirty pages and marking the corresponding entries in the page table of the process being checkpointed as *copy-on-write*. As long as the process does not actually attempt to write into the page, the page will *not* be copied to the volatile checkpoint area in local memory. Instead, as long as the process is only reading the page, the checkpointing handler, that is copying the page to stable storage, and the process using the page for normal processing will share the page. If the process tries to write to the page, a special page fault will be triggered and the page will be copied to a different location in memory. On the other hand, if the checkpointing session completes before the process attempts to write to the page, the page table entry is restored to its previous state and any writes by the process will *not* trigger a page fault. It should be noted that this very promising optimization can *only* be used with asynchronous checkpointing. Previous checkpointing techniques block

process execution until checkpointing is complete so that ‘‘sharing’’ of pages between the process and the checkpointing handler is meaningless.

In a typical system pages containing code are usually read-only, loads are more frequent than stores, and a checkpointing session is expected to take significantly less time than the interval between checkpoints. Based on these facts we expect the optimizations described above to significantly reduce both the time to take a volatile checkpoint and the time to commit the checkpoint to stable storage.

4.8. Conclusions

We have presented two new distributed error recovery schemes for multicomputers based on coordinated checkpointing of interacting sets that minimize disruptions to normal processing due to checkpointing. The domino effect can not occur and only one generation of checkpoints needs to be maintained on stable storage. The algorithms have been designed to handle both communication errors and node failures.

In the synchronous checkpointing scheme an interacting set of processes is suspended from normal computation until all of the process states have been committed to stable storage. The asynchronous scheme is based on (coordinated) volatile checkpointing of interacting sets of processes followed by copying of the checkpoints to stable storage ‘‘in the background’’ involving minimal interference with normal computation. The schemes are integrated with an efficient error detection mechanism that avoids the need for message acknowledgments, transmission of message sequence numbers, and transmission of check bits with

each message. Both schemes can recover from multiple node failures, lost messages, and corrupt messages. Multiple checkpointing and recovery sessions may be active in the system simultaneously operating independently if possible and merging correctly when necessary.

Additionally, as discussed in Section 4.6, the asynchronous algorithm is amenable to highly-effective optimizations that can be supported with relatively small enhancements to a conventional virtual memory system. With these optimizations, a volatile checkpoint can be taken by only a few dozen instructions without physically copying any pages in local memory.

Using the proposed techniques it is possible to implement a highly reliable, general-purpose, large multicomputer system in which the fault tolerance characteristics involve minimal performance overhead and are completely transparent to the user.

Chapter Five

Execution-Driven Simulation of Error Recovery Techniques

DERT (Distributed Error Recovery Testbed) is a testbed for simulating and evaluating the performance of several different classes of application-transparent distributed error recovery schemes. DERT is built on top of an event-driven, message-passing, object-oriented, multithreaded simulation kernel. Above the kernel is a simulated hypercube multicomputer. Actual compiled distributed applications are executed on the simulated multicomputer. Instrumentation of the application assembly code is used to account for application execution time and to allow accurate simulation of periodic application-transparent checkpointing. Checkpointing is implemented in full detail, including associated overhead per message, additional messages, and changes to the memory system. The testbed is fully functional — it can use one of several recovery schemes to restore one or more processes when a simulated node failure(s) occurs.

The overhead of distributed error recovery schemes is caused by a complex combination of additional message traffic, additional processing of sends and receives, as well as the checkpointing of process states. Realistic performance comparisons can only be done with actual applications and the ability to measure the impact of all of these factors. DERT provides for execution-driven simulation and permits easy modification of a wide variety of system parameters, thus offering a level of flexibility not easily achieved by experimentation on a particular real machine. This chapter describes the design, functionality, and use of DERT,

focusing on problems encountered and on examples of its flexibility and realism.

5.1. Introduction

The reliability requirements of distributed systems and of scalable multicomputers can only be met using fault tolerance techniques. The mechanism for recovering a valid system state following the detection of an error is key to implementing fault tolerance. The main desirable features of the error recovery mechanism are: 1) low overhead during normal operation, even for communication-intensive applications, and 2) application transparency so as not to increase the complexity of the application software. There are two main classes of distributed error recovery schemes that are application transparent and prevent unconstrained rollback propagation: *message logging* [Powe83, Borg83, Stro85, John87, Stro88, Sist89, Jalo89, Wang92, Elno92a] (a kind of ‘‘independent’’, or uncoordinated, checkpointing [Rand78, Russ80, Wood81, Bhar88]) and *coordinated checkpointing* [Bari83, Tami84, Koo87, Tami89]. Our research has involved developing new recovery algorithms that fall into both classes (see Chapter 3 for our message logging algorithm, and Chapter 4 for our coordinated checkpointing algorithms). An essential facet of this work has been the performance analysis of these and other distributed recovery schemes.

Realistic performance analysis of distributed checkpointing and rollback techniques is difficult due to the numerous factors that must be taken into account (e.g. parameterization of the target system and of the applications). Thus, most performance analyses of distributed recovery techniques have been limited to greatly simplified analytical models. Realistic experimental performance

evaluations have appeared only recently [John90b, Bhar90, Wang92, Elno92b, Kim93], but with almost no analysis in the context of multicomputer systems — where normal communication latency is small (microseconds) and many applications are communication-intensive.

Our main goal in developing DERT was to measure the impact of error recovery techniques on the execution time of distributed applications on realistic multicomputers. We wanted to be able to see how the overhead changes as various system parameters change and as applications are scaled to additional nodes. Thus, accuracy (realism) and flexibility were the primary factors influencing the choice of an evaluation methodology. To facilitate research on error recovery schemes it was particularly important to be able to evaluate and optimize a variety of low-level architectural and operating system features.

With DERT, we were successful in satisfying the above goals for a distributed error recovery testbed. DERT is built on top of a multi-threaded simulation kernel, called *Simon* [Swop86]. It runs actual distributed applications whose assembly code has been instrumented to 1) count the number of assembly instructions executed and 2) take checkpoints at precise intervals, saving only the modified application state. The impact of several different recovery schemes and optimizations, on the execution time of both coarse- and fine-grained applications during error-free operation has been measured. It is the intention of this chapter to relay our experiences in developing and using this simulator. We have found it to be a powerful and flexible tool for evaluating the overhead and scalability of recovery schemes in the absence of a large multicomputer system. There are disadvantages with the choice, however, and these will be discussed.

The motivation behind the testbed design is given in Section 5.2. Section 5.3 describes the testbed and Section 5.4 the recovery schemes simulated on the testbed. Experiences getting applications to run on the testbed are highlighted in Section 5.5 followed by experiences simulating the recovery schemes in Section 5.6. Emphasis is placed on problems encountered and examples of the simulator's flexibility and realism. Section 5.7 discusses simulator performance. Previous work is briefly described in Section 5.8 and Section 5.9 concludes the chapter.

5.2. Testbed Design Considerations

The testbed is designed to evaluate the performance of application-transparent error recovery techniques running on a scalable multicomputer. We focus on the overhead incurred by these schemes during error-free operation, since failures are assumed to be relatively infrequent in the target system. Our goal is to explore recovery scheme overhead while varying a number of system characteristics: stable storage and network parameters, cost of operating system functions, etc. We are also interested in exploring a range of application characteristics such as communication frequency and the degree to which an application is scaled. Finally, we want to be able to compare message logging to coordinated checkpointing — schemes with unrelated overhead factors. This pointed to execution-driven simulation where system parameters can be controlled and the time to *execute* applications under various schemes can be measured, with detailed measurement of the overhead incurred by each error recovery technique.

We need to simulate application execution and operating system functions at

a very low level to build confidence in the results. The primary metric we want is the application's execution time with a particular checkpointing scheme relative to the application's execution time without any checkpointing. For message logging techniques this means incorporating into the application's execution time the cost of sending and updating dependency vectors with each application message, logging application messages to stable storage, maintaining some additional bookkeeping information, and periodically saving the state of each process. Coordinated checkpointing techniques require that the periodic saving of the states of interacting processes be properly coordinated and that a list of process interactions since the previous checkpoint be kept, and that the application's execution time include the time to perform these functions. To further increase the confidence in our results we made the testbed fully functional; even though our testbed, at this time, is not being used to measure recovery performance, the recovery algorithms can be used to successfully recover from simulated node failures. The next section describes the testbed in detail.

5.3. The Testbed

The testbed is a variable-sized hypercube built on top of the multi-threaded simulator *Simon* [Swop86]. *Simon* is an event-driven simulation tool which provides routines for creating and managing objects, time-multiplexing the execution of object procedures, sending messages between objects, and debugging. Messages are delivered between the objects (implemented as Modula-2 coroutines) through output ports, input ports, and the ‘‘links’’ that connect them. Using these facilities we have constructed a simulator of a multicomputer. This testbed runs

distributed applications whose code has been instrumented at the assembly level to 1) count the number of assembly instructions executed by the application and 2) take checkpoints at precise intervals. Thus we can evaluate the performance overhead of error recovery schemes when used by a variety of applications employing coarse-, medium- or fine-grained parallelism.

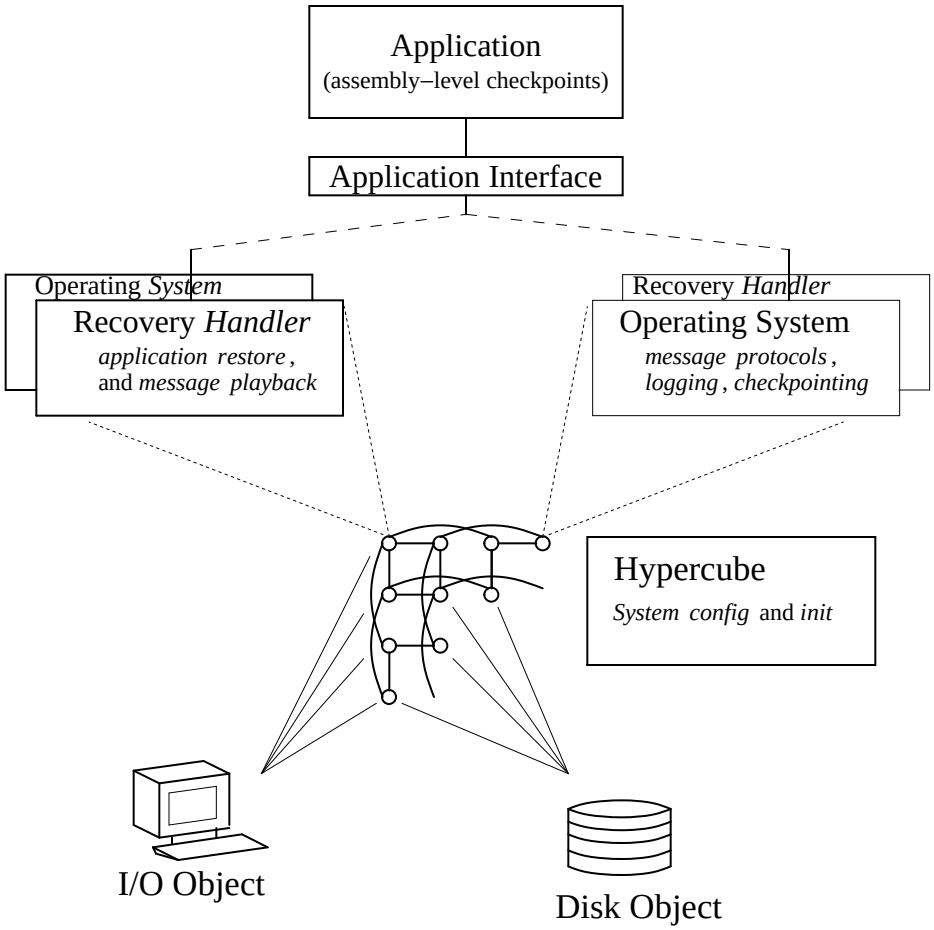


Figure 5.1: Structure of the simulation testbed.

The simulator is written in Modula-2, with some C and assembly routines, and runs on SPARC workstations. It runs distributed applications written in C or Modula-2. Two major error recovery techniques, and several optimizations, have

been implemented: asynchronous message logging and asynchronous coordinated checkpointing. Independent checkpointing has also been coded. Nearly 20K lines of code have been written, with the two main algorithms exhibiting similar complexity. For the message logging algorithm, most of the code implements message handling, logging, dependency tracking, and recovery actions, while with the coordinated checkpointing algorithm most of the complexity rests on determining which action to perform in which phase of the algorithm.

There are four different types of objects (threads) in the testbed, described in the sections below: the application object (one per simulated node, up to 256 nodes),the operating system object (one per simulated node, up to 256 nodes), the I/O object (or ‘‘host processor’’, one object), and the disk object (one object).

5.3.1. The Interconnection Network

The interconnection network is arranged in a hypercube topology. This topology was chosen mainly due to the availability of hypercube applications. All of the distributed error recovery schemes discussed in this thesis are independent of any particular multicomputer topology. The size of the hypercube must be specified by the application itself (up to 256 nodes). Each of the nodes in the hypercube is actually simulated as two objects: the application process and the operating system. Thus the nodes being simulated are not multitasking and there is no context switching between the application and the operating system. Instead, events in the operating system and the application process are simulated as executing in parallel; this greatly simplified the design of the simulator. (Mayfly[Davi92], designed at HP Laboratories actually implements this model).

The network operates with an inexpensive store-and-forward datagram protocol using *e*-cube routing[Sull77, Lang82]. When link conflicts are encountered, messages are stored at that node until the link is available. Up to thirty in-transit messages can be stored at each input port. The simulated cost of sending a message, forwarding a message (through an intermediate node), or receiving a message is ten assembly instructions. Link service time depends on the length of the message — a byte is transmitted each 50 *nsecs* (corresponding to a link speed of 20 *Mbytes/sec*).

Flow control and deadlock handling are incorporated into the testbed. The costs of flow control and deadlock handling are not simulated and the availability of global information in the testbed is exploited to keep this code simple. When a message needs to be sent out of an output port, the sending operating system object checks the availability of buffer space, in the input port at the other end of the link. If there is space in the input buffer for the message then the message is sent, otherwise the sending operating system object will suspend itself until buffer space is available (unless doing so makes it possible for deadlock to occur). The intended receiving operating system object eventually ‘‘wakes up’’ the sending operating system object when space becomes available.

To prevent deadlocks from occurring an operating system object will check, before suspending its execution to wait for buffer space, if such a suspension will result in a cycle of suspended objects. For example, operating system object 6 needs to send a message to operating system object 2 which is waiting on operating system object 3 which is waiting on the disk object which is waiting on operating system object 6. If operating system object 6 suspends its execution to wait for

operating system object 2 to free up buffer space then a cycle of suspended objects exists and it becomes possible for deadlock to occur. If such a cycle would occur then the operating system object will *not* suspend itself, but will instead go ahead and send the message even though the input port has no (simulated) room.

5.3.2. Simulating the Execution of Application Processes

Simulating application execution at the clock cycle level meant performing substantial instrumentation of assembly code. This instrumentation served two purposes: to accumulate the number of assembly instructions being executed and to determine if a checkpoint should be taken. The application is compiled down to assembly instructions and then passed through a parser that inserts assembly code, at the end of each basic block, to count the number of executed instructions. Each instruction is assumed to execute in $100\,nsec$, with most floating point instructions taking $300\,nsec$ and multiply and divide instructions taking $900\,nsec$. This dynamic instruction count accumulation is straightforward and similar to tools like PIXIE [Syst89]. The count is kept in a ‘‘stopwatch’’ variable which is used to periodically synchronize the application process with the global simulation clock.

At the *beginning* of each basic block it is determined if it is time to take a checkpoint, by checking the accumulated running time since the previous checkpoint (of this process). The checkpoint interval is a parameter to the simulator and is specified in clock ticks. If the running time is greater than or equal to the checkpointing interval, the application process 1) sends a checkpoint message to the operating system and 2) executes a blocking receive to wait for a message from the operating system indicating that the checkpoint is sufficiently

complete for the application to resume execution. When implementing this part of the instrumentation we encountered several SPARC architecture- and SunOS-specific problems. For example, performing a procedure call as part of the instrumentation meant that we first had to save (and later restore) register O7, of the current register window. This is due to the fact that the SPARC architecture uses the O7 register of each register window to store the return address for called procedures, which see the return address register as I7. Between procedure calls, however, register O7 is used to store temporary values. If code to save and restore O7 is not placed before and after an inserted call instruction, a ‘‘live’’ value in O7 may be lost.

In addition to this problem we also had to write an assembly routine to save (and later restore) the floating point registers in the floating point coprocessor. This is because TRANSFER(), the system call used by Modula-2 to context switch between coroutines, does not save or restore these registers. Erroneous behavior can occur if, when an application process is in the midst of a series of floating point calculations, a decision to checkpoint is made, and the subsequent transfer to another object happens without saving the floating point registers.

When engineering a major software project, it is desirable to avoid assembly-language programming and implementing architecture-specific routines. Tracking down low-level bugs which this approach generates within a complex simulator is not easy. Further, the portability of the testbed is severely limited. However, to achieve the highest degree of accuracy in our simulations, we had to run applications whose code had been instrumented at this level. This code is completely separate from the rest of the testbed, and once debugged has run

without problem for many months.

Another problem we encountered could have significantly affected the accuracy of our simulations. Processes normally synchronize with the global simulation clock before a message is sent or received and before a checkpoint is taken. In some applications a process(es) may execute for an extended period of time without sending or receiving a message. Thus processes can get ‘‘out of sync’’ with one another and the global simulation clock. The problem this can cause can be illustrated as follows: a process *x* executes for a long time and then starts a lengthy delay to account for the time just executed. This process now receives an ‘‘unexpected’’ message from process *y* (e.g. ‘‘take a checkpoint’’). The problem is that *x* has executed far ahead of the point where it would have received the message if *x* and *y* were actually running in parallel (and not simulated as running in parallel). Such a simulation is correct — it corresponds to a *possible* execution, but probably not a likely execution. This problem exists whenever unanticipated messages or events can occur (e.g. interrupt handling) and it affects the number of checkpoints taken, the size of the checkpoints, and the application execution time. Our solution was simple: when we check if the checkpoint timer has expired (at the start of each basic block) we also check how much time has passed since the process was last suspended. If it is greater than some *TIME_SLICE* amount the process is forced to give up the processor and synchronize with the global simulation clock (see simulation measurements in Section 5.5).

5.3.3. The Operating System

Each application process (object) is bound to a separate operating system object, which can be called only through the three routines that make up the *application interface*: Send, Receive, and Checkpoint. The operating system handles the network interface, checkpointing and recovery duties, and the maintenance of bookkeeping information. All operating system routines — send, receive, memory-to-memory copies, table lookup, etc — have costs associated with them in terms of the number of assembly instructions required to perform the function; these costs are incorporated into the simulator. Thus all functions in our testbed are simulated at the clock cycle level for greatest accuracy.

Each application process is checkpointed prior to execution. After this initial checkpoint, only the modified state of the process is checkpointed. The checkpoint-detection code in each basic block checks the checkpoint timer. If it has expired, a message is sent to the operating system. Upon receipt of this message the operating system enters checkpoint mode and initiates a checkpoint session on behalf of the application process. The operating system 1) takes a volatile checkpoint of the process in local memory 2) wakes up the process as soon as possible considering the size of the checkpoint and the specific checkpointing algorithm and 3) sends the volatile checkpoint to stable storage. With message logging, for example, the application resumes execution immediately after the volatile checkpoint has completed. The checkpoint state consists of: the stack, statically allocated memory containing the process' global variables, and heap from which memory is allocated on demand. Some bookkeeping information is also checkpointed.

The size of the checkpoint state significantly impacts the overhead of the checkpointing algorithms. Thus, for all techniques, only the pages modified since the previous checkpoint are copied during a volatile checkpoint. In order to determine which pages have been modified, the operating system keeps a checksum for each page from the previous checkpoint of the process. A new checksum is calculated at the beginning of the current checkpoint and the two are compared; a difference indicates that the page has been modified and thus needs to be checkpointed †. The new checksum is then stored. The taking of a volatile checkpoint can be optimized further (see Sections 5.6.3 and 4.6).

Recovery sessions are initiated by a simulation message whose receipt signifies that a hardware (node) failure has occurred. The application state and all volatile memory is assumed to be lost and the operating system enters recovery mode to initiate a recovery session and restore the application's state. At some point during the recovery session the operating system wakes up the application process to complete the recovery or to begin re-execution from its consistent restored state.

5.3.4. The Host Processor (I/O Object) and Stable Storage (Disk Object)

Applications must be structured as an I/O object and some number of application processes. The I/O object is responsible for initializing the hypercube and calling the testbed procedures that create all the required objects. The I/O object then acts as the ‘‘outside world’’ (host processor) by generating all inputs

† With 32-bit checksums, on average one of every 2^{32} modified pages is not detected using this method.

and collecting all outputs. There is no restriction as to how this is done.

A single fully-connected disk object is used to simulate m disks; the disk nodes are assumed to have unlimited buffering capability so conflicts do not exist. By default there is one disk per application process. The disk object's main function is to log messages and checkpoints in 100 byte blocks at a rate of three megabytes per second per disk; m disks are simulated by multiplying the speed of the disk by m.

5.3.5. Summary

Like the target multicomputer, the testbed can be scaled to hundreds of processes (nodes) which communicate solely via messages over point-to-point links. Hardware faults are simulated in the nodes, but not currently in the links. Reliable disk nodes are simulated by a single disk object with one three Mbyte/sec disk per application process; disk traffic does not traverse the network. All application and system operations are simulated at the assembly level. To reduce complexity, multitasking and context switching are not simulated; the operating system and application execute concurrently on a node.

5.4. The Recovery Schemes Simulated

Several different error recovery schemes have been incorporated into the testbed: asynchronous message logging (and optimizations), synchronous (fully and partially) and asynchronous coordinated checkpointing, as well as an ''optimal'' independent checkpointing scheme. Checkpointing and recovery are implemented in full detail for the message logging and coordinated checkpointing

algorithms. The independent checkpointing algorithm simulates the checkpointing of application process states, but is ‘‘optimal’’ because it does not simulate *any* additional overhead, such as message ordering, maintenance of bookkeeping information, etc.

The performance overhead of the error recovery schemes can be broken into three parts: 1) overhead incurred during normal operation; 2) the cost of checkpointing process states; and 3) the cost of recovery when failures occur. In the testbed we focus on (1) and (2) since this is the overhead that impacts the system the majority of the time. Only the functionality of recovery, not the performance, has been tested. What follows is a brief itemization of the basic functions required by the message logging and coordinated checkpointing algorithms during normal operation and checkpointing that had to be incorporated into the testbed. A more detailed overhead itemization for these and other error recovery techniques can be found in Section 6.1.

(Asynchronous) Message Logging —

During *normal operation* the following is required:

- Extra processing by the send and receive primitives (to track dependencies).
- Periodic broadcast of logging status.
- Periodic reclamation of old checkpoints and message logs.
- Local memory for bookkeeping data structures and volatile message logs.
- Stable storage for the message log including the dependency

vectors.

- Network bandwidth for dependency vectors and for copies of messages (and their dependency vectors) being sent to remote stable storage.
- Disk bandwidth for messages and their dependency vectors.
- Delay in committing outputs to the external world.

This overhead is likely to have a more significant performance impact on applications with fine-grained parallelism than on applications with coarse-grained parallelism.

During *checkpointing*, network and disk bandwidth are needed to transmit local bookkeeping data structures and the process' current state, and stable storage is required for the checkpoint state and data structures. This overhead is dependent on the checkpoint interval and the size of the process state.

(Asynchronous) Coordinated Checkpointing —

During *normal operation* the overhead incurred is:

- Local memory for the *Buddies* list and processing to update it.
- Delay in committing outputs to the external world.

Also, compared to a system that has no process recovery but uses check bits to send reliable messages — *less* operating system processing and network bandwidth are required to transmit messages, if message transmission error checking is incorporated into coordinated checkpointing as described in Section 4.3.1 and [Tami91].

During *checkpointing* the following is required:

- The process is suspended the length of time it takes to copy the current state into local volatile memory and to queue outgoing *Checkpoint* messages.
- Significant (‘‘background’’) processing by the checkpoint handlers.
- Network and disk bandwidth to transmit checkpoints and message queues; additional network bandwidth for recovery scheme messages.
- Stable storage for the checkpoint: up to two copies per process.
- Coordinated checkpointing results in ‘‘bursty’’ network and disk traffic.

Message logging techniques incur more overhead during normal operation than do coordinated checkpointing techniques and less during checkpointing sessions. Thus, detailed characterization of distributed applications is required to accurately compare the two techniques. All overhead itemized in this section is simulated by the testbed at the level of assembly instructions.

5.5. Experiences with the Applications

Six applications have been incorporated into the simulator: *fast fourier transform*, *Router*, *Placement*, *Matrix QR factorization*, *Olfactory* and *Extract* [Hsu90, Bowe88]. Some of the characteristics of these applications, determined by examining simulation output, are listed in Table 5.1. Fast fourier transform (*fft*) is run over 2^{12} points. *Matrix QR* is run for four successive matrices (64x64, 96x96, 128x128, 160x160). *Router* (with input of 72 nets) and *Placement* (with 183 cells) are both VLSI CAD tools that use simulated annealing algorithms.

Extract is a VLSI CAD tool that performs circuit extraction (on a CIF file with 66K rectangles). *Olfactory* is a hypercube implementation of an anatomically realistic model of a real neural network (the piriform (Olfactory) cortex). All of the applications, with the exception of *QR*, become more communication-intensive as they scale (with constant problem size). *Extract* has the largest state (e.g. 1.4 to 2.9 Mbytes of modified state checkpointed per process) and is the least communication-intensive (sending a message every .5 to 1.7 million instructions). This is in contrast to *Router* where, with 16 processes, each process sends a message, on average, every 492 instructions. We did not know, prior to executing the applications on the testbed what sort of execution profile they would exhibit. Thus, the simulator is useful for obtaining such information (e.g. for use in an analytical model).

In developing the testbed a significant problem was encountered within the applications themselves in the form of intraprocess and interprocess nondeterminism. Message logging algorithms require applications to be deterministic in their actions — therefore, all *intra*process nondeterminism had to be removed. For example, random number seeds had to be saved and restored with the checkpoint.

More difficult however, was a specific case of *inter*process nondeterminism caused by the interaction of 1) the use (by *Placement*) of a ring broadcast to update data on the processors and 2) the use of periodic checkpointing. The simulated annealing algorithm used by *Placement* chooses master-slave pairs in every annealing iteration. Without checkpointing, an application, using the ring broadcast, can be run multiple times, with the same inputs, and it will always give

No. Nodes	4	8	16	32	64	128	256
	Total number of assembly instrs executed by the application						
Extract:	2219M	3001M	3957M	-	-	-	-
OLF:	-	359M	486M	737M	1245M	2259M	4303M
FFT:	-	-	2017M	2081M	2160M	2260M	2422M
QR:	819M	819M	820M	821M	-	-	-
Place:	349M	468M	692M	-	-	-	-
Router:	254M	299M	351M	-	-	-	-
	Total number of messages sent and total message volume (in MBytes)						
Extract: # msgs	1299	2969	7941	-	-	-	-
volume	12.7	25.3	63.5	-	-	-	-
OLF: # msgs	-	2807	7483	18.7K	44.9K	105K	239K
volume	-	11.8	25.3	52.2	106.1	214.3	432.1
FFT: # msgs	-	-	32.0K	80.1K	192K	448K	1.02M
volume	-	-	65.6	82.0	98.4	114.8	131.1
QR: # msgs	55.3K	55.3K	55.4K	55.4K	-	-	-
volume	58.6	58.6	58.6	58.6	-	-	-
Place: # msgs	735	16.5K	131K	-	-	-	-
volume	1.8	3.4	7.9	-	-	-	-
Router: # msgs	127K	326K	715K	-	-	-	-
volume	7.2	16.3	15.5	-	-	-	-
	Avg. No. of pages modified after 146 million instructions (for Extract) and 46 million instructions (for all others).						
Extract:	1.85K	1.62K	1.42K	-	-	-	-
OLF:	-	16.34	19.70	22.14	24.41	25.68	26.54
FFT:	-	-	2.31	2.56	2.97	3.55	4.23
QR:	1.15	1.30	1.58	2.07	-	-	-
Place:	58.63	59.18	57.40	-	-	-	-
Router:	30.00	37.45	48.99	-	-	-	-

Table 5.1: Application statistics: communication frequency (avg. number of assembly instructions executed between sends), number of messages sent, message volume, and typical checkpoint state size (the number of 1Kbyte *modified* pages checkpointed by an independent checkpointing algorithm using a particular checkpoint interval).

the same outputs; the same master-slave pairs will be chosen, in the same order. With checkpointing, however, it is possible for a *different* process to exit a particular ring broadcast first and then decide to become a master. Thus the chosen master-slave pairs are different than in the simulation without checkpointing — altering the rest of the application's execution and resulting in significant changes to application execution time and application results. This made it nearly impossible to compare the execution times of the application with and without checkpointing and say anything meaningful about the overhead incurred by checkpointing.

There were two possible solutions. We could run multiple simulations with checkpointing, measure confidence intervals, and determine an average checkpointing overhead. Instead of this we chose the simpler solution of replacing the ring broadcast with a broadcast that guarantees the order in which processes will exit the broadcast. It should be noted that this problem is not peculiar to the simulator; the same problem would exist if the applications were running directly on a multicomputer.

Out of the large number of simulations that we have run, a very small number had unexpected results in terms the application's execution time, number of checkpoints taken, etc. This turned out to be due to the ''out of sync'' problem described in Section 5.3, which was solved by time-slicing the execution of the application processes. After this problem was solved, we ran additional simulations to determine a reasonable value for the TIME_SLICE variable and to see if our results are sensitive to the value of this variable (see Table 5.2). In general, varying the TIME_SLICE (specified in assembly instructions) does not

appreciably alter application execution time, but was successful in eliminating our rare, aberrant results.

TIME_SLICE	# processes							
	2	4	8	16	32	64	128	256
	Extract							
450	1.1584	1.1739	1.1384	1.1215	-	-	-	-
4.5K	1.1582	1.1740	1.1383	1.1215	-	-	-	-
450K	1.1548	1.1722	1.1391	1.1143	-	-	-	-
45M	1.1557	1.1753	1.1417	1.1156	-	-	-	-
	Olfactory							
450	-	-	1.0030	1.0034	1.0031	1.0036	1.0034	1.0078
4.5K	-	-	1.0030	1.0033	1.0031	1.0036	1.0034	1.0078
450K	-	-	1.0031	1.0033	1.0031	1.0036	1.0036	1.0078
45M	-	-	1.0031	1.0033	1.0031	1.0036	1.0036	1.0078

Table 5.2: Execution time with asynchronous coordinated checkpointing relative to execution time without any checkpointing for several values of TIME_SLICE (the number of assembly instructions executed during a single timeslice).

5.6. Experiences with the Recovery Schemes

The design of the simulator was driven by the following goals: to examine the impact of multiple error recovery schemes on application execution time (without failures), to use a range of communication- and computation-intensive applications, and to examine the overhead of the schemes as the applications are scaled (without failures). Our simulator is fully functional; using one of the recovery algorithms, when a simulated failure occurs, one or more processes can be restored from checkpointed states and the application will correctly (re)execute from that point. However, we are predominately interested in the overhead incurred during normal operation and checkpointing. Therefore, for the simulations which do not measure

the overhead of failure recovery, much of the code that performs the actual checkpointing and rollback functions is ‘‘turned-off’’ while still incorporating the overhead incurred by these functions into the simulations. This conserves disk space, memory, and swap space, increasing the degree to which the applications can be scaled. The main things we did were:

With all schemes:

— Account for the time the simulated disk(s) take to checkpoint process states, but do not checkpoint the states onto the workstation’s disk.

— Do not send checkpoints to the (simulated) disk, but simulate the sending of a message of the correct size and include the size information in the message.

With message logging:

— Do not log messages (or dependency vectors) onto disk or in the sender’s memory; simulate the size of the message but do not actually send/log it.

This greatly reduced the memory usage of the simulator such that it is dominated by the memory needs of the application, O(n) arrays used to store recovery information and collect statistics, and simulator information (object information, etc). With these optimizations *Olfactory* and *fft* could be scaled to 256 nodes when executing on a Sun-4 with 32 *Mbytes* of physical memory and 100 *Mbytes* of swap space. The other applications had been hard-coded to scale to only 16 or 32 nodes.

5.6.1. Message Logging

The flexibility of the testbed allowed us to examine recovery scheme overhead at a very low level. By measuring the processing overhead of individual recovery components, we can 1) determine the performance impact of each component, 2) devise how to improve the performance of the recovery scheme, and 3) measure the effectiveness of the resulting optimizations. This capability is briefly illustrated below, using the asynchronous message logging technique.

We broke the overhead of asynchronous message logging (*AML*) into its major component parts — *Msg_Ovrhd* (overhead caused by dependency tracking, maintenance of bookkeeping information, etc) and *Chkpt_Ovrhd* (overhead caused by the checkpointing of process states, also called *Optimal*)). The functions of these major components are all simulated at the assembly level. After measuring the performance impact of each component, we then devised and simulated optimizations to the kernel code (*Orph_roll* and *DV_diffs*) and hardware support (*Optimal*) to reduce the total overhead, focusing on the message handling component (*Msg_Ovrhd*).

The assembly-level operations (overhead), required by the message handling routines of the message logging algorithm (*AML*) and its optimizations (*Orph_roll*, *DV_diffs*, and *Optimal*), are listed in Table 6.1. The algorithms investigated were:

- *AML* is the basic asynchronous message logging technique (Chapter 3 and [Stro85]).
- *No_Chkpt* is the application without any recovery scheme and with an inexpensive send and receive protocol. In Figure 5.2 *No_Chkpt* is always 1.0. All schemes are graphed relative to *No_Chkpt*.

- *Msg_Ovrhd* is the application without any recovery scheme but with expensive send and receive primitives equivalent to the computation time performed by the operating system in *AML*. This isolates the operating system processing overhead (by the sender and receiver) in the *AML* scheme.
- *Chkpt_Ovrhd* assumes that the message handling operations, simulated in the *Msg_Ovrhd* simulations, are performed in parallel with normal processing by special-purpose hardware (hence *Optimal*). This isolates the checkpointing overhead in the *AML* scheme.
- *Orph_roll* is the *AML* scheme with two optimizations we devised: 1) Messages are *not* examined to determine if they are orphans — orphan detection is delayed until a *Recovery* message arrives. The cost is the potential rollback of additional processes that could have been avoided by detecting the orphan message earlier. 2) The message is delivered to the application as soon as it passes the SSN protocol. The dependency vector is updated *after* the message is delivered; the operating system must complete this update before sending any new messages.
- *DV_diffs* is an optimization, previously proposed by others, that sends only the difference between the dependency vector previously sent by S to R and S's current dependency vector (i.e. X entries instead of N). These simulations *include* the *Orph_roll* optimizations.

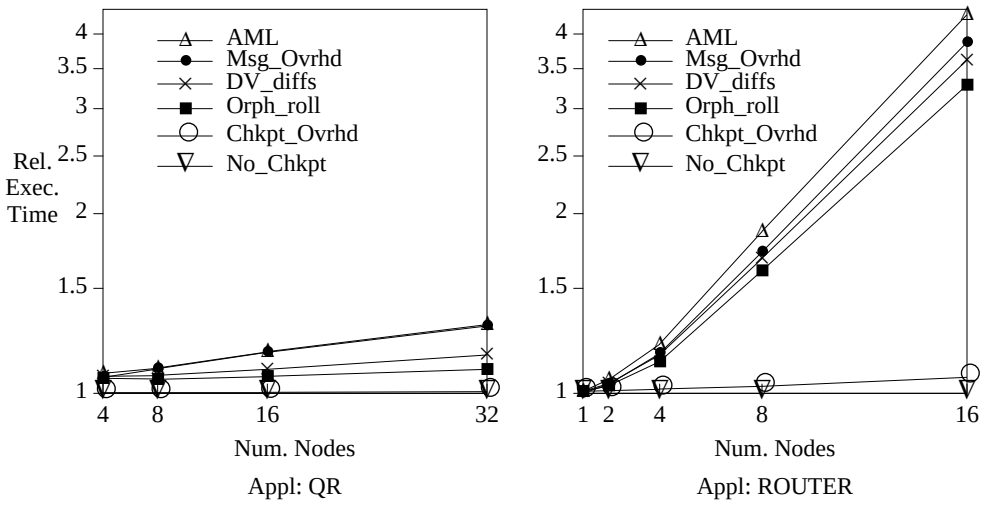


Figure 5.2: Application execution time with message logging relative to execution time with no recovery scheme. The checkpoint interval is 4600000 instructions.

Figure 5.2 shows typical results from the simulator running these recovery algorithms for two of the applications (*fft* performed much worse, *Placement* a little better than *QR*, *Olfactory* a little better than *Router*, and *Extract's* overhead came only from checkpointing). A detailed performance analysis is contained in Chapter 6. Using the asynchronous message logging technique we have shown that our testbed can be used to examine the impact of recovery techniques on actual distributed applications in a high-performance multicomputer environment and that the maximum flexibility afforded by such a simulator allows low-level investigation of architectural tradeoffs and optimizations.

5.6.2. Coordinated Checkpointing

Running simulations using coordinated checkpointing and independent checkpointing illustrates the importance of measuring recovery algorithm overhead on applications as they are executing. It is generally expected that independent checkpointing is less disruptive to system operation (e.g. [Bhar90, Li90]) because only a single process is involved in a checkpoint session; coordinated checkpointing requires the ‘‘simultaneous’’ checkpointing of multiple processes. This intuition is compounded by the fact that, with coordinated checkpointing, processes will checkpoint more often than is dictated by the checkpoint interval. Indeed, in the simulations most interacting sets contained *all* of the application processes and processes checkpointed measurably more often than with independent checkpointing.

We coded several coordinated checkpointing algorithms: *F_Synch*, (fully-)synchronous coordinated checkpointing where processes do not resume execution until the checkpoint has completed, *P_Synch*, (partially-)synchronous coordinated checkpointing where processes resume execution after the interacting set has been found, and *A_Synch*, our asynchronous coordinated checkpointing algorithm. The coded *independent* checkpointing algorithm is, in a sense, ‘‘optimal’’ because only the overhead of checkpointing is simulated — any costs for dependency tracking and/or message ordering are not simulated. Figure 5.3 shows some results of simulations using the *Placement* and *Olfactory* applications. In *Olfactory*, independent checkpointing performed significantly worse than all three coordinated checkpointing algorithms, and in *Placement*, outperformed only the fully synchronous coordinated checkpointing algorithm.

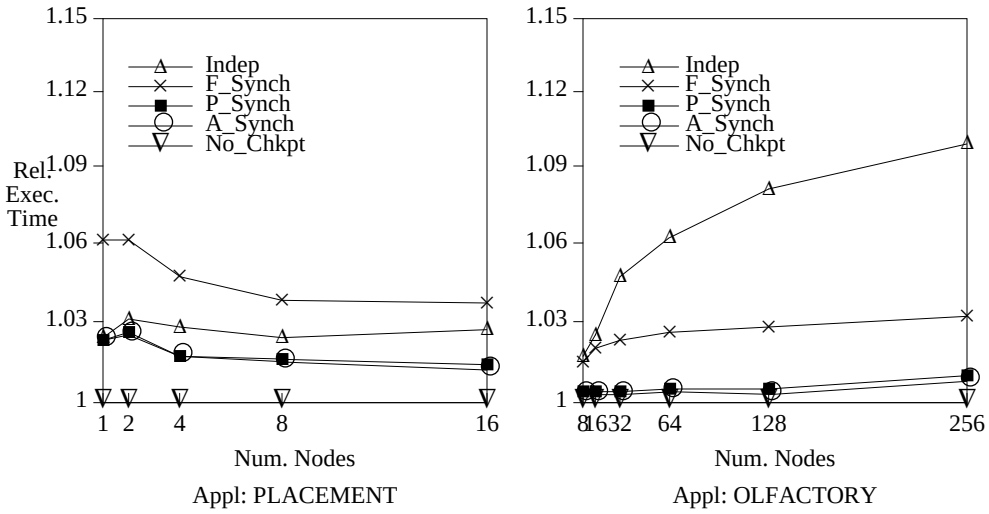


Figure 5.3: Application execution time with coordinated and independent checkpointing relative to execution time without checkpointing. The checkpoint interval is 4600000 instructions.

It appears that the coordination of checkpointing sessions can be beneficial. With independent checkpointing there is a higher probability that process y is checkpointing when process x is waiting to receive a message from y. With coordinated checkpointing x and y are likely to be checkpointing simultaneously, reducing CPU idle time. This confirms results presented in [Elno92b]. A detailed performance analysis of coordinated checkpointing can be found in Chapter 6. It is fairly easy to see the difficulty in comparing message logging and coordinated checkpointing techniques analytically; the results of our examination of coordinated and independent checkpointing, without message logging, further confirm the significance of using execution-driven performance analysis.

5.6.3. Optimizations

The flexibility of the testbed can be exploited to get an approximate measure of the effectiveness of the copy-on-write technique [Bobr72]. With the copy-on-write technique all modified pages are flagged for checkpointing (marked as copy-on-write) at the beginning of each checkpoint. The application process then resumes execution. The first time it attempts to write to a page marked copy-on-write then, if the page has not yet been written to stable storage, the process is suspended while the page is copied in volatile memory. After the copy-on-write has completed, the process continues execution. A reduction in overhead occurs when copy-on-write is performed for less than the total number of modified pages. Incorporating ‘‘true’’ copy-on-write into the simulator requires each memory access (during a checkpoint) to determine whether the page being written to is marked copy-on-write. Indirect and relative addressing modes made this low-level check all but impossible, so an approximation of copy-on-write was pursued. In [Elno92b], true copy-on-write was analyzed by altering the V-System kernel.

We had already implemented an efficient mechanism for tracking which pages have been written since the previous checkpoint (Section 5.3.3). We added a copy-on-write approximation to the simulator by computing the checksum for each (previously modified) page at fixed-length intervals after a checkpoint until all the pages have been written to stable storage. In our approximation we do *not* suspend the process when a new page has been written to and thus overestimate the number of pages that need to be copied in volatile memory. To correct for this we ‘‘speed up’’ writing pages to disk by an amount equivalent to the time that the process would have been delayed. (Thus we do not need to run separate simulations to get

Placement				Extract			
# Procs	CoW Pgs	Dirty Pgs	$\frac{\text{CoWPgs}}{\text{DirtyPgs}}$	# Procs	CoW Pgs	Dirty Pgs	$\frac{\text{CoWPgs}}{\text{DirtyPgs}}$
6.0 MB/s disks (one per process)							
1	1580	17456	0.09	1	55614	223903	0.25
2	3914	37837	0.10	2	41380	268921	0.15
4	2039	24918	0.08	4	36721	247437	0.15
8	3909	49031	0.08	8	33715	310941	0.11
16	4615	80685	0.06	16	46094	418667	0.11
3.0 MB/s disks (one per process)							
1	2781	17456	0.16	1	81521	223903	0.36
2	6478	37837	0.17	2	53902	268921	0.20
4	3460	24918	0.14	4	49413	247437	0.20
8	6492	49031	0.13	8	44444	320941	0.14
16	7887	80685	0.10	16	62255	418667	0.15
0.75 MB/s disks (one per process)							
1	7201	17456	0.41	1	126259	223903	0.56
2	14280	37837	0.38	2	81964	268921	0.30
4	7098	24918	0.28	4	81360	247437	0.33
8	12037	49031	0.25	8	64774	320941	0.20
16	13521	80685	0.17	16	100398	418667	0.24

Table 5.3: This table shows, for the applications *Placement* and *Extract*, 1) the estimated aggregate number of pages copied by the copy-on-write protocol, 2) the actual aggregate number of dirty pages, and 3) the fraction of dirty pages which need to be copied. 1) and 3) are calculated assuming that the disk server is writing dirty pages to stable storage at 6.0, 3.0, and 0.75 Mb/s per process.

our results). The length of the interval between checksum calculations determines the accuracy of the approximation; an interval of 2.25 msec appears to be accurate since, for the majority of intervals (e.g. 92 - 95 percent), *no* new pages are written.

We ran simulations of the copy-on-write approximation using *Placement* and *Extract*. Checkpoints were taken every 46 million instructions (*Placement*) and 146 million instructions (*Extract*). Table 5.3 shows results for disk throughputs of

.75, 3.0, and 6.0 Mbytes/sec. Even for slower disks, copy-on-write can be effective at reducing the number of pages which need to be copied in local memory as part of a volatile checkpoint.

5.7. Simulator Performance

No. Nodes	1	2	4	8	16	32	64	128	256
	Simulation time (nsec) per application assembly instruction								
Extract:	2281	2305	2235	2210	2231	-	-	-	-
(w/ indep)	2694	2734	2677	2648	2600	-	-	-	-
OLF:	-	-	-	349	459	558	646	700	745
(w/ indep)	-	-	-	440	539	649	730	794	836
FFT:	-	-	-	-	946	1053	1153	1384	1832
(w/ indep)	-	-	-	-	1084	1138	1261	1486	1979
QR:	-	-	564	575	573	578	-	-	-
(w/ indep)	-	-	599	621	628	631	-	-	-
Place:	672	721	776	802	1052	-	-	-	-
(w/ indep)	1018	1068	1122	1169	1439	-	-	-	-
Router:	2028	2344	2680	3462	4877	-	-	-	-
(w/ indep)	2711	2995	3281	4090	5341	-	-	-	-

Table 5.4: Simulation time per application assembly instruction; numbers are given for the application without checkpointing and with independent checkpointing.

Finally, we comment on the efficiency of the simulator as it runs the applications. Since processes run sequentially on a uniprocessor, the simulator will be at least a factor of n slower than an n-node multicomputer running the application. We examine the overall efficiency of the simulator by measuring simulation time per application assembly instruction (Table 5.4). As can be seen from the table, the simulator execution rate ranges from a low of .187 MIPS (*Router*) to 2.71 MIPS (*Olfactory*), when run on a host whose rate is 25-35 MIPS.

It appears that, for *Router*, a communication-intensive application, the overhead of sending messages in the simulator shows up as the application is scaled. For *Extract* and *QR*, the simulator becomes relatively more efficient as the application is scaled because the application is becoming less efficient (e.g spending more time waiting for messages). For the remaining applications the simulator becomes less efficient as it spends more time switching processes contexts.

5.8. Previous Work

In [John90b], sender-based message logging was implemented on an ethernet-based network of Sun workstations (V-system). Applications composed of 8 processes were run to obtain experimental results of checkpointing, logging, and recovery overhead. In [Bhar90], synchronous coordinated checkpointing and independent checkpointing were implemented on an ethernet-based network of Sun-3/50 workstations. Input parameters were varied using ‘‘dummy’’ processes which ‘‘communicated’’ through Sun Unix message queues. Many simulation results were reported. In [Elno92b], coordinated checkpointing and independent checkpointing [Bhar88] were implemented on an ethernet-based network of Sun workstations (V-system). Applications composed of 16 processes were run to show that the coordination overhead of coordinated checkpointing can be quite small. In [Wang92], communication trace-driven simulation of 8 process applications from an Intel iPSC/2 hypercube were used to determine rollback distance, percentage of messages logged, and the number of checkpoints reclaimed. Finally, in [Kim93], analytical simulations, using fixed parameters (e.g. message sending intervals of 500 to 7500 seconds), were used to obtain some specific,

limited, results.

5.9. Summary and Conclusions

We have developed a testbed for analyzing the performance of distributed application-transparent error recovery techniques. It simulates a scalable multicomputer (up to 256 nodes) and is currently able to run six applications and a number of recovery schemes. The recovery schemes are fully functional; an application will execute correctly in the presence of simulated node failures. We have described, in detail, the functionality of the testbed, the difficulties we encountered during development, and some of our experiences simulating applications and recovery schemes. The actual performance evaluations of the recovery techniques are described in Chapter 6.

The development of the testbed was motivated by the desire to be able to compare the performance of message logging and coordinated checkpointing techniques in a scalable multicomputer environment. The differing sources of overhead for message logging and coordinated checkpointing techniques precluded the use of analytical models in order to achieve accurate results. Our comparison of coordinated and independent checkpointing (without message logging) has further indicated the significance of using execution-driven simulations (or experiments) instead of those driven by communication traces or employing fixed parameters. We found that the coordination of checkpointing sessions led, in general, to *less* overhead than did independent checkpointing, given our target system assumptions and the applications used.

We have striven to develop a testbed that is realistic and accurate —

application and operating system execution is simulated at the assembly instruction level through low-level timing and assembly code instrumentation. The flexibility of the testbed has enables us to: directly compare different error recovery schemes, devise and measure optimizations to the message handling routines of asynchronous message logging schemes, measure the effectiveness of the copy-on-write technique, simulate an ‘‘optimal’’ message logging scheme, fine-tune a large number of parameters, readily switch from one recovery scheme to another, etc. Few resources are required, the testbed runs on any one of our SPARC 2 or SPARC IPC workstation (25 or 40 MHz, with 24 to 64 Mbytes of physical memory).

This testbed does have some drawbacks. The most serious is the restriction on the size of the application; the degree to which the application can scale is limited by the virtual and physical memory of a single workstation. An obvious solution to this would be to distribute the simulator to run on multiple workstations. Significant changes to the simulator would be required: for example, the deadlock avoidance code would have to be rewritten, as it currently utilizes global variables. Another disadvantage is that the simulation of a multicomputer, running actual distributed applications and recovery schemes, results in a large, complex testbed. Nonetheless, we have found this testbed to be a powerful, flexible tool for analyzing application-transparent error recovery techniques.

Chapter Six
Performance Evaluation

In this chapter we characterize the overhead incurred when *any* checkpointing/rollback scheme is used to provide fault tolerance for a scalable multicomputer system similar to that which we describe in Chapter 2. We then characterize and compare the overhead incurred by several different message logging and coordinated checkpointing techniques. In Sections 6.2.1 and 6.2.2 we evaluate the performance of several specific techniques using DERT — our simulation testbed described in Chapter 5.

6.1. Detailed Itemization of Recovery Scheme Overhead

In a system which supports only non-interacting processes, checkpointing involves periodically suspending each process, copying the process' state to stable storage, and then allowing the process to resume normal computation. When a node fails, all information on the node is assumed to be lost. The checkpointed states of all the processes that were on the failed node are restored from stable storage to working nodes. The processes may then resume computation from these older states[Rand78]. In order to be able to tolerate node failures that occur during checkpointing, the old checkpoint of a process is erased only after the new checkpoint has been completely written to stable storage.

When processes interact with one another, additional costs are incurred by the checkpointing and rollback algorithms in order to ensure that a consistent system state can be recovered whenever a failure(s) occurs.

The overhead of a fault tolerance scheme is determined by the frequency of checkpointing, the ‘‘cost’’ of saving each checkpoint, extra operations required during normal computation for error detection and maintenance of bookkeeping information, and the ‘‘cost’’ of recovery. The rest of this section details the overhead of the checkpoint and recovery algorithms used by a ‘‘generic’’ algorithm which checkpoints processes independent (for non-interacting and interacting processes), the independent checkpointing algorithm presented in [Bhar88], optimistic message logging[Stro85], two other major message logging techniques [Elno92a, Wang92], and synchronous and asynchronous coordinated checkpointing of interacting sets of processes.

6.1.1. Overhead Of Checkpointing/Recovery of Independent Processes

Certain facets of overhead are shared by all checkpointing and rollback algorithms. What follows is an itemization of a ‘‘generic’’ recovery scheme that checkpoints and rolls back non-interacting processes without any coordination between checkpointing and recovery sessions. We also examine the impact that communication between processes can have on this overhead itemization.

a) *computation time lost during checkpointing and normal operation*: Without the use of redundant hardware, a fraction of the processing cycles on each node must be devoted to checkpointing. Specifically, each node that has processes that are participating in a checkpoint session, must suspend normal operation in order to run the handlers that perform the various functions necessary for checkpointing. If the communication coprocessor can operate independently of the application processor, normal execution of *other* processes on the node may resume before the

checkpoint session is complete. Depending upon whether the checkpointing scheme is synchronous or asynchronous the process being checkpointed may need to be suspended until its checkpoint session is complete and its state is stored on stable storage. With synchronous checkpointing a process does not resume normal execution until it has been confirmed that its checkpoint is committed to stable storage. With asynchronous checkpointing a volatile checkpoint is taken and then the process being checkpointed can resume normal operation. Taking a volatile checkpoint involves copying the entire process state, or just the modified state, into local volatile memory. The copy-on-write technique can be employed to reduce the amount of local copying required, as discussed in Section 4.6.

If processes interact with one another then checkpointing results in the indirect loss of additional computation time. This happens when an application process has to wait (longer) to receive a message because the sending process is currently suspended from execution due to its participation in a checkpointing session. The use of an underlying reliable communication protocol, required by the recovery algorithm, may also cause some loss of computation time as can the maintenance of local or remote bookkeeping information. Lost computation time directly impacts the time it takes to execute an application.

b) *load on the communication network*: In the ‘‘worst case’’ the entire process state has to be sent to disk. If virtual memory is employed, part of the state is already on disk and the size of the state in local memory, which has to be sent to disk, is related to the process’ *working set*. With appropriate hardware support for identifying modified pages (Section 4.6), only the process state which has been changed since the last checkpoint is sent to disk.

If processes interact then a checkpoint session may involve some coordination between checkpoint handlers thus increasing load on the communication network. Extra bookkeeping information and message logs or queues may need to be transmitted and saved to ensure that a consistent system state can be recovered in the event of a hardware failure. Again, if an underlying reliable communication protocol is required by the recovery algorithms then this will also cause some increased load on the communication network.

c) *disk bandwidth required*: A significant fraction of the total disk bandwidth available on the system may be needed for saving checkpoints, thus limiting the performance of I/O-intensive applications.

Additional bandwidth may be needed for bookkeeping information and message logs or queues.

d) *disk storage required*: The minimum storage requirement is one complete checkpoint state per process plus additional space for storing a second copy of the checkpoint of processes during checkpointing sessions.

If processes interact then multiple checkpoints may have to be stored (due to domino effect) as well as bookkeeping information and message logs or queues.

e) *storage required in local memory*: Part of local memory is needed for storing program code for handlers and for volatile checkpoints.

If processes interact then bookkeeping information (e.g. a list of processes that have communicated with local processes) and message logs or queues may also need to be stored. Such use of local memory can increase paging activity, load on communication links and the average memory access time.

f) *special processing during recovery*: Each node participating in a recovery session must suspend normal operation so that it can run the recovery handlers. If processes interact then some coordination between recovery handlers will also be required.

g) *computation time lost due to rollback*: When a process is rolled back, the computations it performed since its last checkpoint are lost and have to be repeated. This overhead is inversely proportional to the frequency of checkpointing.

If processes interact then the computation time lost can be more significant. It is still possible that only the failed process(es) will have to roll back (e.g. if synchronous message logging with independent checkpointing is employed). However, some techniques that checkpoint processes independently may need to roll back a failed process to a checkpoint that is not the most recent (due to domino effect), and some additional non-failed (orphaned) processes may also have to roll back. If processes checkpoint in a coordinated fashion then a group of processes will need to be rolled back together to yield a consistent system state. An increase in network load may also result from the replaying of computations.

h) *delay in committing outputs to the outside world*: Outputs can not be committed to the outside world until it can be guaranteed that they will not be rolled back in the event of a failure(s). When processes do not interact outputs produced by a particular process can be committed to the outside world after the next checkpoint of that process has completed.

When processes interact output commitment delays vary widely from algorithm to algorithm. For many algorithms outputs can not be committed until

the next recovery line is established on stable storage (via coordinated checkpointing or independent checkpointing). Output commitment is identical to the issue of checkpoint reclamation. If an output can be committed then all previous checkpoints (and message logs) can be reclaimed.

Bhargava and Lian have proposed an algorithm based on the independent checkpointing of process states[Bhar88]. With this algorithm, the use of a reliable end-to-end communication protocol is assumed and processes can be nondeterministic in their execution. The overhead incurred by this algorithm can be broken down as follows:

a) *computation time lost during checkpointing and normal operation*: The overhead lost during checkpointing is ‘‘minimized’’ — checkpointing is asynchronous of process execution (uses volatile checkpoints) and involves only one process at a time. Each process maintains an input information table (IIT) which is updated on stable storage every checkpointing session. This two-dimensional table is used to keep track of message exchanges between the various checkpoint intervals of all of the processes. During normal operation, computation time is lost due to the processing needed to append a checkpoint interval number and sessions number onto each application message and also to check the sequencing of those numbers upon receipt of the message. The checkpoint interval number must be inserted into the current (local) entry of the IIT after a message has been received.

b) *load on the communication network*: Extra load on the communication network is incurred when a process state and the corresponding IIT are sent to disk. Addition load is required for the reliable communication protocol and to send the

checkpoint interval number and sessions number with each message.

c) *disk bandwidth required*: For the checkpointing of process states and the updating of the IITs.

d) *disk storage required*: For each process one (possibly large) IIT is maintained and one or more checkpoint states. In the worst case, all checkpoint states may have to be retained until the application completes.

e) *storage required in local memory*: For local volatile checkpoints (if used), the current entry of the IIT, and local bookkeeping information (small).

f) *special processing during recovery*: All communicating processes participate in the recovery session. Significant processing is required to use the IITs to calculate the most recent recovery line on stable storage.

g) *computation time lost due to rollback*: A group of communicating processes is rolled back to the recovery line which has been calculated by the recovery algorithm. This recovery line can include the current states of non-failed processes. The amount of computation time actually lost is highly dependent on the communication behavior and checkpoint frequency of the application. Nonetheless, this algorithm does not guarantee any progress of the recovery line, thus processes may have to roll back to their initial states. Non-failed processes continue with normal operation until it has been determined that they must be rolled back. Thus the recovery handlers run, at least in part, asynchronous of process execution.

h) *delay in committing outputs to the outside world*: Outputs can be committed to the outside world after the next system-wide recovery line has been established. In the worst case this may not occur until the application has completed. Reclamation

of checkpoint states is the same.

This algorithm is tolerant of *n* process failures and the worst case scenarios can be avoided if it is possible to periodically invoke a coordinated checkpointing algorithm to establish a recovery line on stable storage.

6.1.2. Overhead of Asynchronous Message Logging

The overhead incurred in a system using optimistic message logging (Chapter 3[Stro85]), can be characterized as follows:

a) *computation time lost during checkpointing and normal operation*: The time lost during checkpointing is ‘‘minimized’’ — checkpointing is asynchronous of process execution (uses volatile checkpoints) and involves only one process at a time. Throughout normal operation processing is required by the sender to append the dependency vector and (updated) send sequence number onto each application message. The receiver must check the send sequencing, determine if the message is an orphan (compare the dependency vector to the IST), update the receiver’s dependency vector, and update the receive sequence number. Each message must also be both acknowledged and logged to stable storage. All communication must be through a reliable protocol. Logging status is periodically broadcast and checkpoints and message logs are periodically reclaimed (a coordinated effort).

b) *load on the communication network*: Includes checkpoints (plus current local bookkeeping information), the logging of all messages sent between processes, the overhead of the reliable communication protocol (including message acknowledgments), and the transmission and logging of bookkeeping information (e.g. dependency vectors). This overhead is further examined in the next

subsection.

c) *disk bandwidth required*: Similar to (b).

d) *disk storage required*: Generally, the storage requirement might be one checkpoint state per process plus the stable message logs. Older checkpoints are discarded as soon as it can be determined that they will not be needed for either *state backout* (undoing the effects of messages) or for resending messages lost by a failed process. Earlier message logs can also be committed/reclaimed at this time (as soon as all the state intervals upon which a message depends have been logged). This is dependent on the communication behavior of the applications, the rate at which messages are logged, and the frequency of checkpointing. If some processes are slow to log messages and/or checkpoint then all processes that (transitively) interact with these processes will not be able to reclaim storage as quickly as they otherwise would.

e) *storage required in local memory*: Local storage requirements include volatile checkpoints, messages being held until they have been logged, dependency vectors, SSNs, RSNs, INCs, log vectors, etc.

f) *Special processing during recovery*: Collect any volatile logs from the senders, and the stable log from the disk. The message log must then be played back to the recovering process(es) until it is brought up-to-date. The IST is updated and *Recovery* messages are broadcast.

g) *lost computation time due to rollback*: All processes *directly* affected by a hardware failure must be rolled back, along with orphans, if any. Processes may have to roll back multiple times (to earlier checkpoints) unless they are, by default, restored to the earliest non-reclaimed checkpoint. There is no need to wait for

messages to be sent ‘‘on-demand’’ by non-failed (non-rolled back) processes because these messages are already present in the message log.

h) *delay in committing outputs to the outside world*: A message can not be committed to the outside world until every state interval upon which it depends has been logged. This is determined by comparing the message’s dependency vector to a log vector (which records the logging progress of every process).

Optimistic recovery is a distributed, n-fault-tolerant, application-transparent scheme requiring an underlying reliable communication protocol. The only restriction made on application processes is that they be deterministic.

6.1.2.1. The Cost of Logging Messages to Disk

Some application programs for multicomputers may require each process to send a short message (say, eight bytes) every 100 instructions [Jage87, Dall87]. Such communication-intensive applications may be incompatible with message logging techniques since very large amounts of data may have to be logged. For example, consider a multicomputer with 512 20 MIPS processors running such an application. In this system a total of approximately **820 Mbytes** are sent as messages every second. Logging this much information is a major problem. Specifically, high-performance disk drives have a maximum bandwidth of 3 Mbytes per second. Thus, if all messages are to be logged to disk, as required by Strom and Yemini’s original scheme[Stro85], more than *200* disk drives will have to be dedicated to logging messages.

Message logging techniques also require bookkeeping information (dependency vectors, sequence numbers, etc) to be transmitted with each message.

As a result, the load on the communication network and the complexity of send and receive operations, which have to manipulate this bookkeeping information, is significantly increased. Some of this information (e.g. dependency vectors) must also be logged to disk, further increasing the number of disks required to support message logging techniques.

6.1.2.2. Other Message Logging Techniques

In sender-based message logging[John87] messages are logged in local, volatile store and correct operation can not be assured if more than one process can fail at a time. In n-fault-tolerant logging[Stro88] messages are logged in volatile store until it is determined that they can be discarded or spooled (logged to disk). Using the above example, volatilely logging messages in local memory involves storing an average of 1.6 Mbytes/sec (391 4K pages every sec) on every node. In both cases performance will be affected by having to copy messages to memory and because some fraction of main memory is needed to maintain message logs.

To reduce the local memory usage with sender-based message logging [John87], processes may have to checkpoint frequently in order to limit the size of the message logs. In the case of n-fault-tolerant logging [Stro88], the amount of information to be logged to disk is reduced by discarding some messages and predicting the arrival order of messages (using a deterministic heuristic). Performance will depend on how well the heuristic performs, on how many messages can be discarded, and on how much compute time and how many extra messages are required to run the heuristic. We do not itemize the overhead of sender-based message logging because we are restricting our attention to n-fault-

tolerant algorithms. We do not itemize the overhead of n-fault-tolerant logging because a complete algorithm was not given (many details were omitted including where and how dependency vectors are maintained, where and how $<SSN, RSN>$ pairs are logged, etc). Many of these omitted details are essential to both the performance and correctness of the approach.

Two more algorithms demand closer examination: Manetho[Elno92a] and Wang and Fuchs' optimistic message logging[Wang92], which is not really a ''true'' message logging technique. In Manetho messages are logged in the local volatile memory of the sender and antecedence graphs are used in place of dependency vectors. An antecedence graph is simply an extension of the dependency vector in which *all* the events (state intervals) upon which a state interval (or message) depends are recorded. This eliminates orphan processes and orphan messages since all messages sent by non-failed processes have (in the sender's local memory) all of the ordering information which is needed to regenerate the state interval during which that message was sent. Furthermore, a message may be committed to the outside world as soon as its antecedence graph has been (synchronously) logged to stable storage.

The cost is that an antecedence graph can grow well beyond the size of a dependency vector and it must be transmitted with every application message just as a dependency vector is. The size of a process' antecedence graph is reduced by periodically logging it to stable storage and then transmitting and updating only the unlogged (''current'') portion. Updating the receiver's antecedence graph is a more complex operation than updating a dependency vector and each node in the graph has five fields instead of three. Additionally, it must be possible to be able to

record, as part of the antecedence graph, all the information necessary to regenerate an identical internal process event during a recovery session. Other message logging techniques do not have this restriction. The overhead breakdown of this technique is as follows:

a) *computation time lost during checkpointing and normal operation*: Processes checkpoint independently and asynchronously. In addition to the process' state, the volatile message log and current antecedence graph are also saved to stable storage during the checkpoint session. Throughout normal operation processing is required by the sender to append the antecedence graph onto each application message and to save a copy of the message and graph in local volatile memory. The receiver must check the sequencing of the message and update its local antecedence graph. All communication must be through a reliable protocol. The antecedence graph may be periodically logged to stable storage during normal operation.

b) *load on the communication network*: Includes checkpoints (plus current antecedence graph and volatile message logs), the overhead of the reliable communication protocol, and the transmission and logging of bookkeeping information (e.g. antecedence graph).

c) *disk bandwidth required*: Similar to (b). All volatile message logs are written to stable storage (with antecedence graphs) so bandwidth needs may be substantial as discussed above.

d) *disk storage required*: At most two checkpoint states per process plus the volatile message logs (saved to stable storage during each checkpoint) and the antecedence graphs. Generally, the storage requirement might be one checkpoint

state per process plus the stable message logs. If desired, processes can be forced to checkpoint so that older messages logs and antecedence graph nodes can be reclaimed. In general, garbage collection should be straightforward and storage needs should not be great (despite the fact that bandwidth requirements may be quite significant).

e) *storage required in local memory*: Local storage requirements include volatile checkpoints, the volatile message logs including antecedence graphs, the local current antecedence graph and a small amount of additional bookkeeping information. Since all messages are logged in the local memory of the sender, local memory requirements may be substantial, as discussed in Section 6.1.2.1, and adversely impact performance.

f) *special processing during recovery*: All interacting processes (senders) participate in recovery. (More accurately, remote procedure calls are invoked, by the recovering process, on the node of each process which had sent message(s) to it prior to the failure). Each of these processes (synchronously) log their antecedence graphs to stable storage and then search their volatile message logs for any messages needed by the failed process(es) to restore a state consistent with the non-failed processes in the system.

g) *computation time lost due to rollback*: Only the failed processes need to roll back. Significant processing time is needed to retrieve the volatile message logs, log all the antecedence graphs to stable storage, and to replay the log.

h) *delay in committing outputs to the outside world*: A message can be committed to the outside world after a single synchronous disk operation which logs the message's antecedence graph to stable storage.

This algorithm is tolerant of n process failures, requires communication to be reliable, and assumes that processes are deterministic.

Wang and Fuchs proposed an optimistic message logging algorithm which is not really a ‘‘true’’ message logging technique. Unlike other message logging algorithms processes can be nondeterministic. Like coordinated checkpointing techniques this algorithm records (logs) the messages sent along ‘‘channels’’ between processes, but checkpointing is independent, not coordinated, so this information can be significant. One of the main assumptions of this approach is that there is no access to the communication facilities in the multicomputer and thus that it can not be determined, by the application and checkpointing routines, if a particular message has arrived (correctly or not).

The basic approach is to log all messages in local volatile memory at the receivers and then determine, during the next checkpointing session, which of these messages actually belong to ‘‘channels’’ (as in coordinated checkpointing techniques; here they are called ‘‘state messages’’). These messages are then logged to stable storage. Since checkpointing is not synchronized it is possible that a large number of messages will be ‘‘state messages’’ and thus have to be logged to stable storage. To determine which messages are state messages, each application message must be tagged with the process’ current checkpoint interval (incremented at the beginning of a checkpoint session), a processor (process) identifier, and a send sequence number. Each processor (process) also maintains a communication information table (on stable storage) which records the dependencies between the checkpoint intervals of interacting processes. During recovery these tables can be used to determine a recovery line. Interacting

processes will thus be rolled back to this recovery line (somewhat similar to coordinated checkpointing techniques). Checkpoints are reclaimed in two ways: 1) checkpoints taken earlier than the most recent global recovery line can be discarded and 2) a reclamation algorithm can be run to determine which checkpoints can never belong to a recovery line and thus can be discarded.

In general this algorithm, similar to [Bhar88], does not guarantee that the recovery line will progress. Depending on the communication behavior of the application and the likelihood of ‘‘accidentally’’ saving a recovery line, some number of checkpointed states may actually never become become part of any recovery line and thus are ‘‘wasted overhead’’.

a) *computation time lost during checkpointing and normal operation*: Processes checkpoint asynchronously and independently, requiring no coordination overhead. Processing overhead is required to append a checkpoint interval number, send sequence number, and processor identifier onto each message. The receiving process logs all incoming messages into its local volatile memory. The communication information table is updated during each checkpointing session and all state messages are logged to stable storage. Additional processing overhead is incurred to determine which messages are state messages.

b) *load on the communication network*: Additional load on the communication network is required for the message tags, volatile checkpoints, transmission of state messages to stable storage, and to update the communication information table.

c) *disk bandwidth required*: Similar to (b).

d) *disk storage required*: Each process has, on stable storage, one communication information table, one or more checkpoint states, and the corresponding state

messages. In the worst case, all (‘‘non-obsolete’’) checkpoint states may have to be retained until the application completes. A reclamation algorithm can be used to determine which checkpoint states can never belong to a recovery line (and thus may be reclaimed).

e) *storage required in local memory*: Local memory is needed for volatile checkpoints, a small amount of bookkeeping information, and the volatile message log. All messages are logged in local volatile memory until the next checkpoint session. During the checkpoint session some of these messages are discarded and the rest are logged to stable storage. As discussed in Section 6.1.2.2, logging a large number of messages in local memory can adversely impact application performance.

f) *special processing during recovery*: During recovery significant processing is required; all non-failed processes participate in the determination of the recovery line by using communication information tables to construct an extended checkpoint graph.

g) *computation time lost due to rollback*: The recovery line can include the current states of non-failed processes but multiple non-failed processes may still need to be rolled back. The processes which do roll back may have to roll back to states older than their most recent checkpoint. In fact, the algorithm does not guarantee any progress of the recovery line, thus processes may have to roll back to their initial states.

h) *delay in committing outputs to the outside world*: Messages may be committed to the outside world only after the establishment of the next recovery line (which may not occur until the application completes). Checkpoints taken prior to the

recovery line may be discarded and a special checkpoint reclamation algorithm can be used to garbage collect checkpoint states that can *never* be part of a recovery line.

This algorithm is tolerant of n process failures and, as in[Bhar88], the worst case scenario could be avoided if it is possible to periodically invoke a coordinated checkpointing algorithm to establish a recovery line on stable storage (e.g. using the packet-switching approach, Section 4.3.4, and a high-level approach to reliable communication since access to low-level communication facilities is assumed to be impossible).

6.1.3. Overhead Of Coordinated Checkpointing of Interacting Sets

Unlike most other error recovery schemes, coordinated checkpointing of interacting sets of processes does not assume that the message delivery system is made reliable via sliding windows (or two-phase commit protocols) and error detection bits for two reasons. First, the use of sliding windows has the stringent requirement that message transmission delays must be bounded while the use of two-phase commit protocols imposes a high level of overhead on every message. Secondly, communication errors occur very infrequently relative to the frequency at which messages are sent. Hence, it is not desirable to penalize the system continuously by checking for errors on every message. Instead, detection of communication errors (and recovery from them) is incorporated into the checkpointing/recovery algorithms[Tami84, Tami89] (Chapter 4).

In[Tami89, Tami91] ‘‘normal’’ interprocess messages have no error detection bits, no bookkeeping information, and require no acknowledgments. Instead

signatures (accumulated by Linear Feedback Shift Registers, see Section 2.4), which are updated in parallel with message transmission, are maintained for each pair of communicating processes. For example, if virtual circuits are used for interprocess communication then signatures are accumulated for each incoming and outgoing virtual circuit. Communication between processes in the interacting set can be checked during a checkpoint session by performing ‘‘end-to-end’’ checks on all the virtual circuits between processes in that set. If a ‘‘mismatch’’ is detected then the interacting set containing the processes on both ends of the virtual circuit are rolled back. Messages which are used to coordinate the creation of checkpoints and for error recovery must be verified before they are used and hence carry error detection bits. This reduces the total overhead incurred by the error recovery scheme during normal operation.

The overhead of *synchronous* coordinated checkpointing can be generally characterized as follows:

a) *computation time lost during checkpointing and normal operation*: The interacting set of processes is suspended until the checkpointing session has terminated. The interacting set may contain, in the worst case, all of the processes that constitute the application. If the nodes are time-shared, they may continue to execute other processes. Executing the checkpoint handlers and coordinating their activities also results in lost computation time. During normal operation, updating the *Buddies* list may require some very small processing overhead.

b) *load on the communication network*: Increased load on the communication network results from sending checkpoints and the current contents of the message queues to disk. This traffic may be ‘‘bursty’’ because multiple processes are

checkpointed ‘‘simultaneously’’. Additional overhead is minimized since i) no additional check bits are added to normal messages, and ii) there is no need to log messages to stable storage. During checkpointing there is increased load on the communication network due to coordination messages.

c) *disk bandwidth required*: Only process checkpoints are sent to disk. This bandwidth is thus entirely dependent on application size and checkpoint frequency. Timers can be used to fine tune this amount of checkpoint state on a ‘‘per application’’ basis (true for any checkpointing technique).

d) *disk storage required*: One checkpoint per process (two during checkpointing). Only the most recent committed recovery line needs to be retained.

e) *storage required in local memory*: ‘‘Minimal’’: only dynamic communication information (*Buddies* list) needs to be stored.

f) *special processing during recovery*: Recovery of a failed process requires the coordination of the interacting set of processes containing the failed process. Messages may need to be flushed and discarded and the checkpoint states need to be restored.

g) *computations lost due to recovery*: All processes directly affected by an error *and* their interacting set(s) must be rolled back. In the worst case the interacting set(s) contains all of the processes in the application which must then be rolled back to their most recent checkpoints.

h) *delay in committing outputs to the outside world*: Messages can not be committed until the next checkpointing session completes.

The overhead of *asynchronous* coordinated checkpointing can be generally

characterized as follows:

a) *computation time lost during checkpointing and normal operation*: A process is suspended until the volatile checkpoint has been taken and *Checkpoint* messages have been queued to all processes with whom direct communication has taken place since the last checkpoint (the *Buddies* list). Processes may have to be suspended again during the checkpoint session if a synchronous send is executed and the target process might be in *or* outside of the interacting set. As soon as it has been determined whether or not the target process is part of the interacting set the suspended process may resume executing. Asynchronous sends never *directly* result in the process being suspended, although the message will not actually be sent until it can be determined whether the receiving process is part of the interacting set of the sender. Thus, the sender may be suspended if it expects an immediate response to the message. Executing the checkpoint handlers and coordinating their activities also results in lost computation time. During normal operation updating the *Buddies* list may require some very small processing overhead.

b) *load on the communication network*: Same as for synchronous coordinated checkpointing.

c) *disk bandwidth required*: Same as for synchronous coordinated checkpointing.

d) *disk storage required*: Same as for synchronous coordinated checkpointing.

e) *storage required in local memory*: ‘‘Minimal’’: dynamic communication information needs to be stored during normal computation as well as storage for volatile checkpoints and holding buffers during checkpoint sessions.

f) *special processing during recovery*: Same as for synchronous coordinated

checkpointing.

g) *computations lost due to recovery*: Same as for synchronous coordinated checkpointing.

The proposed synchronous and asynchronous coordinated checkpointing schemes are n-fault-tolerant, do not require communication to be reliable, and make no restrictions on application behavior.

6.2. Performance Evaluation Using DERT

The performance overhead of error recovery techniques can be broken into three parts: 1) overhead incurred during normal operation (not including the cost of checkpointing process states); 2) the cost of checkpointing process states; and 3) the cost of recovery when failures occur.

Since failures in the target system are assumed to be infrequent, the performance analysis in this dissertation is limited almost exclusively to examining the overhead that error recovery techniques impose on the system during normal operation (with checkpointing) relative to a system that has no logging or checkpointing. We examine asynchronous message logging in detail (as presented in Section 3.5 and[Stro85]), and then asynchronous coordinated checkpointing (as presented in Section 4.5) along with an ‘‘optimal’’ independent checkpointing algorithm.

We focus on the impact of the checkpointing algorithms on application execution time relative to the execution time of the application without any checkpointing. Thus we isolate (a) *computation time lost during checkpointing and normal operation* and make sure that the (simulated) (b) *network bandwidth*

required and (c) *disk bandwidth required* are more than sufficient to support the error recovery schemes and the applications. These execution time measurements also ignore the (d) *disk storage required* and (e) *local memory storage required* by the recovery schemes, as was discussed in the previous section. We will examine (b) and (c) in the case of asynchronous message logging and asynchronous coordinated checkpointing and briefly address the issue of (h) *delays in committing outputs*. Simulations were run to test the correctness, but not the performance, of the recovery algorithms (f and g).

If the communication network and/or disk bandwidth can not support the increased load caused by the use of a particular error recovery scheme, then the execution time of the application may be directly or indirectly affected. We ignore any additional overhead that might result from limited available network and disk bandwidths by ensuring that both are more than adequate. For the applications we used and the recovery schemes we examined this was easily satisfied by the default 20 Mbyte/sec links in our testbed and by simulating a dedicated 3 Mbyte/sec disk at each node. This dedicated disk is used solely for the checkpointing of process states and logging of application messages. In most cases much less bandwidth is actually needed to support these activities.

The amount of local memory storage required by an error recovery scheme can also impact the execution time of the applications. This overhead is ignored by our testbed; no paging activity is simulated (i.e. all applications are completely resident within a node's local memory) and the available local memory is assumed to be ''infinite''. Available disk storage is also assumed to be infinite.

6.2.1. Performance of Asynchronous Message Logging

Our research addresses the design and performance of error recovery schemes on massively parallel systems. Thus we have focused our attention on scalability issues and on examining applications that exhibit different communication/computation ratios. The trend in multicomputers is toward systems that have communication protocols with very small overhead (e.g. Transputer, Paragon, CM-5[Sper93], J-machine[Nuth92, Noak93], AP1000[Hori93], active messages[Eick92], etc). Previous performance analyses of message logging techniques have looked at small-scale multiprocessors (16 nodes or less) where communication is relatively expensive[John90b, Wang92]. It is not surprising that the additional overhead incurred by message logging, in terms of operating system processing during normal computation and per-message overhead, may not be significant in such systems (especially for ethernet-based multiprocessors).

We limit our evaluation here to the asynchronous message logging algorithm presented in this dissertation (based on[Stro85]) and a few simple optimizations. In all of the simulations processes are checkpointed independently. Checkpointing a process involves 1) copying *only the modified 1K byte pages* of the process state in local dedicated buffer space, then 2) waking up the application, and 3) transferring the process state, in the background, to stable storage. This overhead is isolated in the graphs (see *Chkpt_Ovrhd* below).

The rest of the overhead stemming from the message logging scheme is incurred during normal operation. A small amount of overhead results from the periodic reclamation of disk storage (checkpoints and message logs) and the

periodic broadcast of each process' logging status. The remaining overhead is determined by the amount of per-application-message processing that must be performed by the operating system (and/or hardware). Thus, this potion of the total overhead is highly dependent on the communication behavior of a particular application. Table 6.1 itemizes the per-application-message processing, in terms of the number of assembly instructions required to perform each function. We assume that the base communication protocol used by the target multicomputer is quite inexpensive (ten assembly instructions to perform either a send, a ''forward'' through an intermediate node, or a receive).

AML is the asynchronous message logging algorithm presented in Chapter 3. The processing performed by the operating system, in order to *send* an application message, entails: copying the O(n) dependency vector onto the message, sending the message from the node and then saving a copy of the message and its dependency vector in local volatile memory. The operating system processing required to *receive* an application message entails: checking the send sequence number of the message, determining if the message is an orphan, updating the receiver's dependency vector, delivering the message to the application, and then sending an acknowledgment to the sender and logging the message to disk. There are additional costs associated with 1) each link transfer, 2) each intermediate node, 3) the processing, by the sender, of the acknowledgment from the receiver and 4) the processing, by the receiver, of the acknowledgment from the disk. These costs, listed in Table 6.1, are all incorporated into the simulator.

Looking at the first half of Table 6.2 we can see how the operating system processing overhead, associated with each application message, increases as the

	AML	Orph_roll	DV_diffs	Optimal	No_Chkpt
Sender:					
DV diff	-	-	$N*33$	-	-
Tag copy	$N*1.5$	$N*1.5$	$X*1.5$	-	-
SEND	10	10	10	10	10
Save DV	-	-	$N*1.5$	-	-
Setup to -	129	129	129	-	-
Save msg	$(size+N)*1.5$	$(size+N)*1.5$	$(size+X)*1.5$	-	-
Link Xfer:	$\frac{size+N+4}{2}$	$\frac{size+N+4}{2}$	$\frac{size+X+4}{2}$	$\frac{size+N+4}{2}$	$\frac{size}{2}$
Intermediate Node:	10	10	10	10	10
Receiver:					
RECV	10	10	10	10	10
SSN check	43	43	43	-	-
Orphan check	$6+N*33$	-	-	-	-
Update DV	$6+N*48$	-	-	-	-
DELIVER	0	0	0	0	0
Send Ack	145	145	145	-	-
Send Log	129	129	129	-	-
Update DV	-	$6+N*48$	$6+X*48$	-	-
Ack_Msg	304	304	304	-	-
Disk_Ack	274	274	274	-	-

Table 6.1: Breakdown of operating system processing overhead incurred per application message in terms of assembly instructions executed. N is the number of processes and $X <= N$.
A one word checksum is sent with each message (and the message is saved at the senders) to satisfy the reliable message transmission requirement. Check bit processing is assumed to be free. Each dependency vector entry (incarnation number and *RSN*) is (optimistically) assumed to be contained in a single byte. Additionally, there is no cost to send the send sequence number (just to check the sequencing). There is additional overhead in the form of a periodic broadcast of logging status and a periodic (coordinated) attempt to reclaim old checkpoints.

No. Nodes	4	8	16	32	64	128	256	dynamic
O/S overhead per application message (in assembly instructions executed)								
AML:	1392	1728	2400	3744	6432	11808	22560	(+*size** 1.5)
Orph_roll:	1254	1458	1866	2682	4314	7578	14106	(+*size** 1.5)
DV_diffs:	1188	1326	1602	2154	3258	5466	9882	(+*size** 1.5 +*X** 51)
No_Chkpt:	20	20	20	20	20	20	20	20
Optimal:	20	20	20	20	20	20	20	20
Avg. No. of assembly instrs executed by each application process between sends								
Extract:	1.71M	1.01M	.498M	-	-	-	-	-
OLF:	-	128K	64.9K	39.4K	27.7K	21.5K	18.0K	-
FFT:	-	-	63.0K	26.0K	11.2K	5.04K	2.37K	-
QR:	14.8K	14.8K	14.8K	14.8K	-	-	-	-
Place:	475K	28.4K	5.28K	-	-	-	-	-
Router:	2000	917	492	-	-	-	-	-

Table 6.2: Total assembly instruction counts; for each of the five schemes (*AML*, *Orph_roll*, *DV_diffs*, *No_Chkpt*, and *Optimal*) the total number of assembly instructions executed by the operating system *per application message* is divided into a static part (dependent on the number of nodes/processes) and a dynamic part. The dynamic part must be added to *each* of the static totals. *size* represents the size of the application message, in bytes. The second half of the table shows, for each of the six applications, the average number of assembly instructions executed between sends (from the same process). These numbers are readily calculated from the information given in Table 5.1.

number of application processes is scaled. In addition to the presented *AML* algorithm we also looked at several optimized versions of the algorithm which attempt to reduce the amount of per-application-message processing required: *Orph_roll*, *DV_diffs*, and *Optimal*. The different recovery schemes simulated were:

- *AML* is the presented asynchronous message logging scheme.
- *No_Chkpt* is the application without any message logging or checkpointing

and with an inexpensive send and receive protocol (see Table 6.1). In Figures 6.1 and 6.2 *No_Chkpt* is always 1.0. All other schemes are graphed relative to *No_Chkpt*.

- *Msg_Ovrhd* is the application without checkpointing or message logging but with expensive send and receive primitives equivalent to the computation time performed by the operating system of the *AML* scheme. This isolates the contribution of operating system processing, by the sender and receiver, to the overall overhead for the *AML* scheme.
- *Chkpt_Ovrhd* assumes that the operations listed in Table 6.1 are performed ‘‘for free’’ in parallel with normal processing by special-purpose hardware. This isolates the overhead contributed by checkpointing process states plus a small periodic cost for broadcasting log vectors and reclaiming checkpoints. Thus, $Msg_Ovrhd + Chkpt_Ovrhd = AML$.
- *Orph_roll* is the presented asynchronous message logging scheme with the following two optimizations: the orphan detection filter is eliminated and the update of the dependency vector is performed *after* the message is delivered to the application process (see Table 6.1). Since the process’ dependency vector *is* updated with the dependency vector on each application message then, when a *Recovery* message is received the recovery handler can still determine that the application’s state has been orphaned. The cost is the potential rollback of an additional process(es) that could have been avoided by detecting the orphan message earlier. However, if failures are infrequent relative to the frequency of communication then this cost may be justified. The second optimization delivers the message to the application as soon as it

passes the SSN protocol. The dependency vector can be updated *after* the message is delivered to the application as long as the operating system can not be interrupted to deliver a new message from the application *before* completing the update.

- *DV_diffs* is an optimization, previously proposed by others, that computes the difference between the dependency vector previously sent from a process S to a process R and S's current dependency vector - which is then sent with the application message (i.e. X entries instead of N). These simulations *include* the *Orph_roll* optimizations.

Table 6.1 shows the breakdown of operating system processing overhead required, per application message, for each of these simulated algorithms. Table 6.2 shows the change in the total number of assembly instructions executed by the operating system, per application message, (from Table 6.1) as the hypercube and application are scaled. The second half of Table 6.2 indicates, for each application, the frequency with which processes communicate with one another (given as the average number of assembly instructions an application process executes between sends).

Figures 6.1 and 6.2 show the overhead of the different recovery schemes in terms of increased application execution time for the applications *fft*, *Router*, *Placement*, *Matrix QR*, *Olfactory*, and *Extract*. The checkpoint interval is just under 500 msec for *fft*, *Router*, *Placement*, matrix *QR*, *Olfactory* and just under 1500 msec for *Extract*. The default disk bandwidth of 3Mbytes/sec was insufficient to support the checkpointing of *Extract*'s large process state every 500 msec. The y axis in these graphs is the application's execution time with a

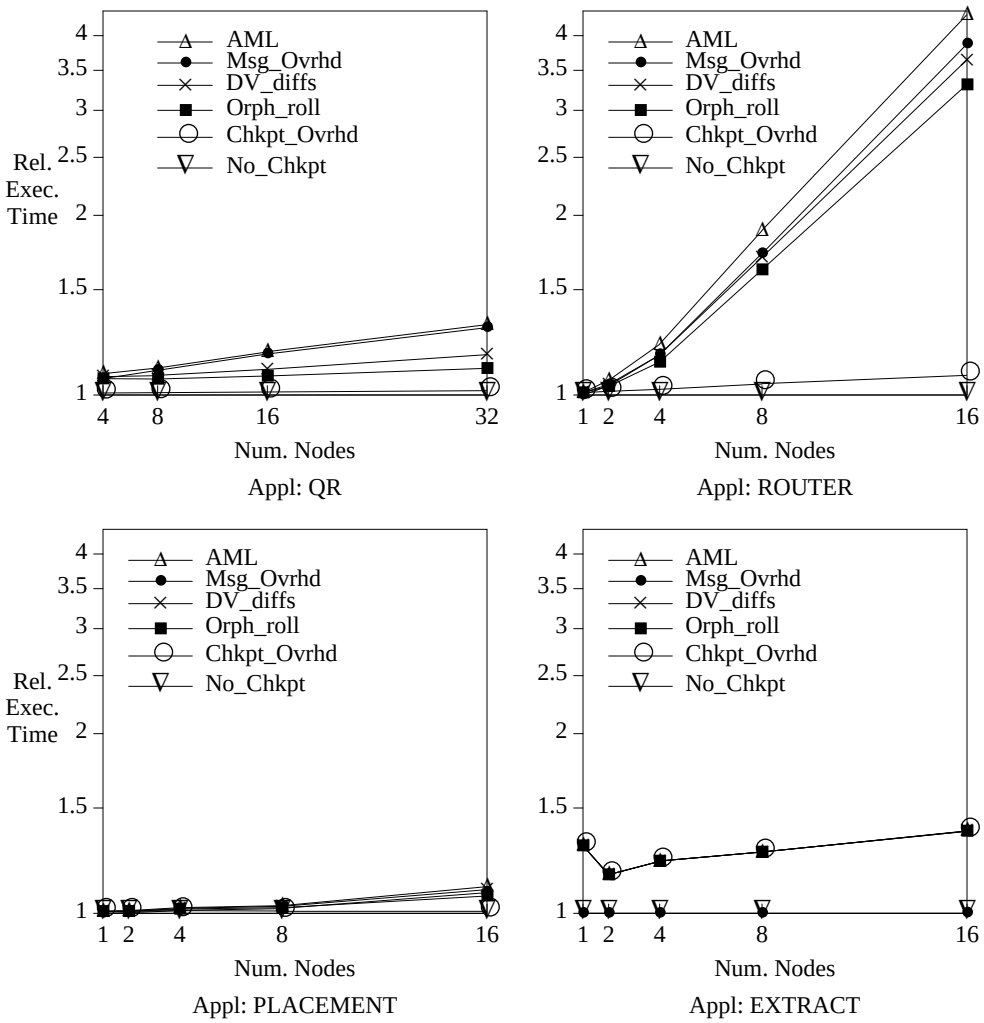


Figure 6.1: Application execution time with message logging relative to execution time without any recovery scheme. Checkpoints are taken every 4600000 instructions (14600000 instructions for *Extract*).

particular checkpointing algorithm relative to the application's execution time without any checkpointing. This axis is given in logarithmic scale so that all of the graphs would be both readable and use the same scale. The *x* axis indicates the number of processes that constitute the application and ranges from 1 to 256 nodes (depending upon the application). In all cases the problem size is held constant

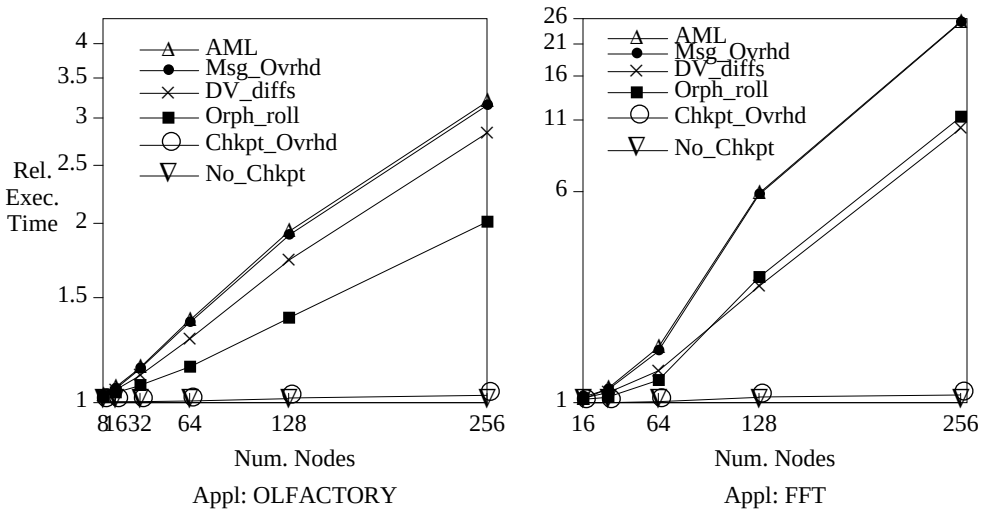


Figure 6.2: Continuation of Figure 6.1. Application execution time with message logging relative to execution time without any recovery scheme. Checkpoints are taken every 4600000 instructions. (Note change in Y axis for the *fft* graph).

while the application is scaled.

fft has very little process state to checkpoint (two to four 1Kbyte pages, see Table 5.1 in Section 5.5) therefore almost all of the recovery scheme overhead is due to processing by the operating system, which is confirmed by the *Msg_Ovrhd* simulations. This overhead scales quite poorly — overhead is only 3 percent and 13 percent when *fft* is comprised of 16 and 32 processes, but climbs to 60 percent, 592 percent and 2535 percent when the application is scaled to 64, 128, and 256 processes. It is worth noting that, in our simulator, the application and operating system do not need to be context switched since they are simulated as executing in parallel. Having to perform these context switches would further increase the total overhead incurred by operating system processing. Table 6.2 helps to clarify the poor scalability of the recovery algorithm. This table shows that the amount of

processing performed by the application between message sends decreases as the application scales while, at the same time, the amount of operating system processing required per message increases, due to the increasing size of the dependency vector.

The optimizations *Orph_roll* and *DV_diffs* offer significant improvement but still scale poorly (reaching overhead levels of 1026 and 1130 percent, respectively, when *fft* is scaled to 256 processes). Extensive hardware support, if feasible, could reduce or eliminate this problem as evidenced by the *Chkpt_Ovrhd* simulations. All overhead above the *Chkpt_Ovrhd* curve is *completely independent of the checkpoint frequency* (for these simulations where operating system does not need to block waiting for disk accesses to complete) and is dependent on the communication frequency of the application and the size of the messages sent.

Like *fft*, *QR* also has a small process state (one to two pages) and the majority of its overhead comes from operating system processing. With 32 processes, 95 percent of the 31.2 percent total overhead can be attributed to the per-application-message operating system processing. In general, we found that the *Orph_roll* optimizations perform well (with 32 processes, the total overhead reaches only 10.8 percent for *QR*). Adding the optimization of sending only the changes to the dependency vector, in addition to the *Orph_roll* optimizations, actually increases the overhead for *QR*, *Router*, *Placement*, and *Olfactory* applications. This is because the dependency vector difference must be computed on-the-fly as part of the send primitive. Thus, this optimization does not appear to be worthwhile for these applications and our assumptions.

Router is the most communication-intensive application (see Table 6.2) and

the operating system processing overhead incurred by message logging again scales very poorly. The optimized algorithms show significant improvement but hardware assistance is needed to eliminate the trend. *Placement* exhibits the same trend as it is scaled but to a significantly reduced degree (overhead is less than ten percent for all simulations). The overhead of the computation-intensive *Extract* application is fairly ‘‘steady’’ (between 16 and 37 percent) and is *completely* dependent on the checkpoint frequency. This is not surprising since messages are sent only at a rate of one per 0.5 to 1.7 *million* instructions executed. Lastly, *Olfactory* simulations show results similar to *fft* and *Router*.

These simulations have led us to conclude that for applications, like those used, that run in a multicomputer environment and are medium- or fine-grained, the operating system message handling overhead required by asynchronous message logging algorithms may be prohibitive. This problem worsens considerably as the applications are scaled. *Chkpt_Ovrhd* simulations suggest that hardware support, if feasible, can have a dramatic impact on overhead.

6.2.1.1. Overhead Due to Network Bandwidth

In our simulations the message logging algorithm’s network bandwidth requirements did not contribute to increased application execution time. This is an optimistic result for two main reasons —

1) applications are assumed to be completely resident in the node’s local memory and thus no paging activity occurs when the application is not being checkpointed, and
2) every node has a dedicated link to a dedicated disk so that paging activity due

No. Nodes	Total Messages	Percent Recovery	Percent Application	No. Nodes	Total Messages	Percent Recovery	Percent Application
Router				*FFT*			
1	372	93.3	6.7	16	155012	79.3	20.7
2	269378	75.0	25.0	32	340132	76.4	23.6
4	508067	75.0	25.0	64	971460	80.2	19.8
8	1312325	75.1	24.9	128	826383	87.3	12.7
16	2868440	75.1	24.9	256	2003983	88.1	11.9
Placement				*Olfactory*			
1	2659	65.5	34.5	8	13544	79.2	20.8
2	62774	74.7	25.3	16	35477	78.9	21.1
4	66492	74.8	25.2	32	82096	77.2	22.8
8	151004	76.1	23.9	64	272913	83.6	16.4
16	530161	75.2	24.7	128	826383	87.3	12.7
				256	2003983	88.1	11.9
Matrix QR				*Extract*			
4	223493	75.2	24.8	1	943	91.3	8.7
8	223012	75.2	24.8	2	2460	86.6	13.4
16	229751	75.9	24.1	4	7841	83.4	16.6
32	277501	80.0	20.0	8	16805	82.3	17.7
				16	50881	84.4	15.6

Table 6.3: Total messages sent in each simulation. Also given is the percentage of this total which is attributable to the error recovery algorithm (*AML*) and the percentage attributable to the application.

to checkpointing does not require any network bandwidth either.

Thus only application messages and messages required by the recovery algorithm (acknowledgment and log vector broadcasts) require network bandwidth. Further, only one application runs on our testbed at a time, thus the network is hardly stressed to begin with.

In our system model (Chapter 2) we assumed that the disk nodes would be remotely located, not locally, and therefore, when examining the total number of messages required by the recovery algorithm we included the additional messages

required to log application messages to disk along with the disk acknowledgment messages. In our simulator both of these types of messages actually use only the dedicated link to disk and thus have no impact on network traffic. The total number of messages sent in each simulation are listed in Table 6.3 and are ‘‘worst-case’’ in the sense that acknowledgment messages and log messages are not piggybacked onto application messages, but are transmitted separately. From this table we can see that from 65.5 to 93.3 percent of the total messages sent are due to the asynchronous message logging scheme. The information in these tables was taken from the same simulations that produced the graphs in Figures 6.1 and 6.2.

6.2.1.2. Overhead Due to Disk Bandwidth

The disk bandwidth requirements of asynchronous message logging were not a significant problem for several reasons:

1) The process states were small — ranging from 80 Kbytes (*fft*) to approximately 3 Mbytes (*Extract*).
2) Only the portion of the process state modified since the previous checkpoint was checkpointed.
3) Each node was simulated as having its own disk connected by a dedicated link.
4) The applications were not very communication-intensive (except for *Router*).

Table 6.4 indicates the percentage of application execution time that each disk was kept busy logging messages and checkpoints. It also shows the breakdown (by percentage) of time spent on each activity. Table 6.5 shows the *total* disk

No. Nodes	Percent Exec. Time	Relative Checkpoint	Percent Logging	No. Nodes	Percent Exec. Time	Relative Checkpoint	Percent Logging
Router				*FFT*			
1	1.8	(99.5	00.5)	16	10.8	(5.2	94.8)
2	4.7	(22.3	77.7)	32	12.9	(4.2	95.8)
4	7.6	(9.2	90.8)	64	13.3	(2.7	97.3)
8	10.9	(3.8	96.2)	128	10.3	(1.1	98.9)
16	9.1	(2.5	97.5)	256	8.0	(0.5	99.5)
Placement				*Olfactory*			
1	5.1	(99.7	0.3)	8	11.1	(6.8	93.2)
2	5.5	(83.4	16.6)	16	16.4	(4.3	95.7)
4	3.5	(72.5	27.6)	32	20.5	(2.8	97.2)
8	2.8	(58.6	41.4)	64	21.8	(2.6	97.4)
16	3.4	(42.7	57.3)	128	19.5	(1.7	98.3)
				256	16.3	(1.3	98.7)
QR Matrix				*Extract*			
4	19.5	(0.3	99.7)	1	5.1	(99.6	0.4)
8	18.9	(0.3	99.7)	2	3.1	(98.9	1.1)
16	17.7	(0.3	99.7)	4	3.2	(95.5	4.5)
32	14.9	(0.3	99.7)	8	2.6	(92.4	7.6)
				16	2.0	(85.4	14.6)

Table 6.4: Percentage of application execution time that the disk(s) was (were) busy (on average).
This percentage is then broken into time spent logging messages and time spent checkpointing process states.

bandwidth required, for each application, in order to support the checkpointing of process states and the logging of messages. The information in these tables was taken from the same simulations that produced the graphs in Figures 6.1 and 6.2.

Total Disk Bandwidth Required (Mbytes/sec)			
# Nodes		# Nodes	
	Router		*FFT*
1	4.205	16	69.113
2	6.059	32	91.532
4	9.384	64	156.171
8	24.783	128	400.399
16	50.725	256	1408.379
	Placement		*Olfactory*
1	17.773	8	12.656
2	41.448	16	26.606
4	30.530	32	55.556
8	56.717	64	121.335
16	114.864	128	280.713
	Matrix QR		*Extract*
4	56.224	1	323.120
8	58.167	2	272.108
16	61.680	4	282.487
32	64.375	8	331.927

Table 6.5: The total disk bandwidth required to support asynchronous message logging (in Mbytes/sec).

6.2.1.3. Reducing the Number of Orphan Processes

In the proposed message logging algorithm (Section 3.5) we introduced the optimization of logging the *RSN* at the sender in order to reduce the number of orphan processes that result from a failure. This could be useful in a system that requires a probabilistic bound on the time it takes for the system to recover from a node failure (e.g. soft real-time). Any improvement in the number of orphan processes by this optimization relies on the assumption that the logging of the *RSN* at the sender proceeds faster and/or more robustly than the logging of messages

(and *RSNs*) to stable storage. This may be a reasonable assumption since logging the *RSN* at the sender requires the transmission of a very small message that can be sent immediately after the *RSN* is calculated. In contrast, logging to stable storage requires a copy of the entire application message and its dependency vector to be sent to stable storage.

Further, logging the *RSN* at the sender is a ‘‘free’’ operation if an acknowledgment is already needed by the reliable message delivery protocol. Ideally, if the *RSN* could be guaranteed to reach the sender then orphans would never occur as the result of a single process failure †. Also, reducing the probability that *any* orphan will occur is important because orphan messages can continue to propagate, creating more orphan processes, up until the time that recovery information (the new IST entry) has been received throughout the system (see Table 6.6).

Table 6.6 shows that, in our simulations, orphans are, in general, not likely to occur. This is because ample disk bandwidth, with no network congestion, is available to log messages as fast as they arrive. However, in the unlikely event that a single orphan does occur, *many* orphans can occur. Thus, in some systems, logging the *RSN* temporarily at the sender, in addition to logging it on stable storage, may be beneficial to avoiding an extended recovery session involving the rollback of multiple (orphaned) processes.

In *non*-soft-real-time systems this optimization can have a significant

---

† As in *synchronous* message logging algorithms which, by not allowing process execution to proceed until the *RSN* has been logged, guarantee that orphans will not occur in the event of a single process failure.

beneficial affect if any of the following conditions are true:

1) Techniques for reducing the number of messages logged to disk are used[Stro88, Wang92]. These techniques log *all* messages in local memory and delay the logging of (a smaller number of) messages to disk.

2) If it is desirable to log messages in groups as a single unit (as mentioned in[Stro85, Wang92]) instead of individually as we did.

3) There is periodic congestion at the disk or logged messages must cross the network and there is periodic network congestion.

In all of these cases the logging of messages to stable storage is delayed, further increasing the chances that one orphan (and thus many orphans) may occur as the result of a failure.

	No. Simulations	No. Orphan Processes		No. Simulations	No. Orphan Processes
	Placement			*Olfactory*	
16 nodes	136	none	64 nodes	138	none
	14	15		1	1,2,4,5,18,32,63
				2	16
				3	8
	Router			*FFT*	
8 nodes	136	none	64 nodes	141	none
	2	1		1	9,13,16,21
	12	7		2	8
16 nodes	136	none		3	7
	1	1,13,14			
	13	15			

Table 6.6: The number of orphans that result when a node, running a single process, fails.

6.2.1.4. Committing Outputs to the External World

Using asynchronous message logging techniques, a message can not be committed to the outside world until it can be guaranteed that the message will not be ‘‘undone’’ if a failure occurs. In other words, the message can be released once it can no longer be orphaned — the state intervals upon which it depends have all been logged to stable storage. Two optimizations have been proposed[Stro88, Elno92a] to reduce the time it takes to commit messages to the outside world.

With our assumptions and the applications we used the optimizations presented in[Stro88] would *not* commit outputs in a more timely fashion than our algorithm. In[Stro88] all messages are logged in local volatile memory. By associating ‘‘NeedCounts’’ with each message it can be determined that some (possibly most) messages will not actually need to be logged to stable storage. This requires some cooperative processing by the operating systems on different nodes. A second optimization uses a heuristic to predict the arrival of messages. If the heuristic is correct then ordering information also does not need be logged to stable storage. As they state, this does not change the basic idea that a message can not be committed to the outside world until it is guaranteed to be reproducible. The argument in [Stro88] is that, by reducing the number of messages logged to stable storage, messages are logged more quickly and thus the latency of committing external messages would be reduced. This is a reasonable assumption when the disk lags behind the application when all messages are being logged. In our simulations disk bandwidth was not a bottleneck and thus these optimizations would not improve the commit time of external outputs. Also, it should be noted that even if it takes some (unacceptable amount of) time to log messages to stable

storage, null logging is still not necessarily beneficial under our assumptions. This is because coordinated, and potentially significant, extra operating system processing is required for maintaining NeedCounts.

In[Elno92a] antecedence graphs instead of dependency vectors are maintained for each process. This allows each process to know the ordering of *all* events upon which it depends. In contrast, dependency vectors only record the most recent event in every other process upon which a process depends. Thus, if antecedence graphs are used, committing outputs to the external world requires only a single synchronous disk operation to record the graph of the process sending the output message and no multi-node coordination. For systems that require the timely (and periodic) commitment of output messages this may be an attractive technique. However, the additional processing required to update the graph and to send the incremental portion of the graph on each message is likely to be significantly larger than the equivalent activities for a dependency vector. Thus it is a reasonable expectation that, for systems that fit our assumptions and for the applications we used, operating system processing overhead would be *significantly* higher than for the basic asynchronous message logging algorithm we simulated. In this case it is quite possible that outputs will not be committed any faster (wall clock time) due to the overall increase in application latency. Both techniques require further examination.

6.2.2. Performance of Coordinated Checkpointing

In this section we evaluate the overhead during normal operation (with checkpointing) that synchronous and asynchronous coordinated checkpointing techniques impose on the system relative to a system without any checkpointing. We also compare the coordinated checkpointing approach to a somewhat ‘‘optimistic’’ implementation of independent checkpointing.

In all of our simulations checkpointing the process state involves copying the *modified* 1K byte pages of the process state into local dedicated buffer space. This state is then transferred to non-local stable storage (*incremental checkpointing*). In coordinated checkpointing techniques all messages in transit between members of the interacting set are flushed to their destinations and then written to stable storage as part of the destination’s checkpoint. The performance, during normal operation, of coordinated checkpointing and independent checkpointing (without message logging) techniques is highly dependent on the cost of saving process states to stable storage. Asynchronous coordinated checkpointing and all independent checkpointing techniques (including message logging) can benefit from the additional use of the copy-on-write optimization (see Section 4.6). A comparison between incremental checkpointing (which checkpoints only the modified process state) and copy-on-write checkpointing was made in [Elno92b]. The copy-on-write optimization was also examined, using our simulation testbed, in Section 5.6.3. These results are orthogonal to the performance comparison made in this section and can be used to reduce the performance overhead incurred by any one of a number of asynchronous recovery schemes.

During normal operation the coordinated checkpointing techniques presented

in this dissertation incur only a very small amount of additional overhead in order to maintain the *Buddies* list on behalf of each application process. Independent checkpointing techniques generally incur some amount of per-application-message operating system processing overhead[Bhar88, Wang92]. In the ‘‘optimistic’’ independent checkpointing algorithm, we do not simulate *any* additional overhead during normal operation †. Also, even though the detection of message transmission failures has been incorporated into our coordinated checkpointing algorithms (a technique which can not be used by independent checkpointing algorithms), we have not incorporated this optimization into the coordinated checkpointing algorithms simulated on the testbed. Instead, coordinated and independent checkpointing are assumed to use the same reliable communication protocol. Both of these points favor independent checkpointing in our performance comparison.

The simulated error recovery schemes are:

- *No_Chkpt* is the application without any checkpointing scheme. In the figures *No_Chkpt* is 1.0 and all other schemes are graphed relative to *No_Chkpt*.
- *A_Synch* is the presented asynchronous coordinated checkpointing algorithm (see Section 4.5).
- *F_Synch* is (fully-)synchronous coordinated checkpointing in which processes do not resume execution until the new checkpoint states

† This algorithm is an incomplete recovery scheme; in the event of a failure, processes would always have to roll back to their initial states because this algorithm does not provide any support to make it possible to determine if some other recovery line exists on stable.

have been committed to stable storage and the checkpoint session has completed (see Section 4.3).

- *P_Synch* is (partially-)synchronous coordinated checkpointing where processes are allowed to resume execution as soon as the interacting set has been found.
- *Indep* is an ‘‘optimistic’’ version of independent checkpointing in which processes are checkpointed once per checkpoint interval, but no additional overhead (e.g. message sequence numbers and message logging) is simulated.

In all of the simulations for each of the recovery schemes processes start out with ‘‘staggered’’ checkpoint intervals to eliminate any ‘‘cold start’’ effect where processes run nearly synchronous with one another despite the lack of any external (hardware or software) synchronization.

Figures 6.3 and 6.4 show the overhead of several recovery schemes in terms of increased application execution time for the applications *fft*, *Router*, *Placement*, *Matrix QR*, *Extract*, *Olfactory*. The checkpoint interval is 14600000 instructions for *Extract*, and 4600000 instructions for all other applications. The x axis indicates the number of processes that constitute the application and ranges from 1 to 256 nodes (depending upon the application). The problem size is held constant while the applications are scaled. Unlike Figures 6.1 and 6.2 the y axis (the application’s execution time with a particular checkpointing algorithm relative to the application’s execution time without any checkpointing) is not given in logarithmic scale.

Tables 5.1 and 6.2 show statistics for each application in terms of

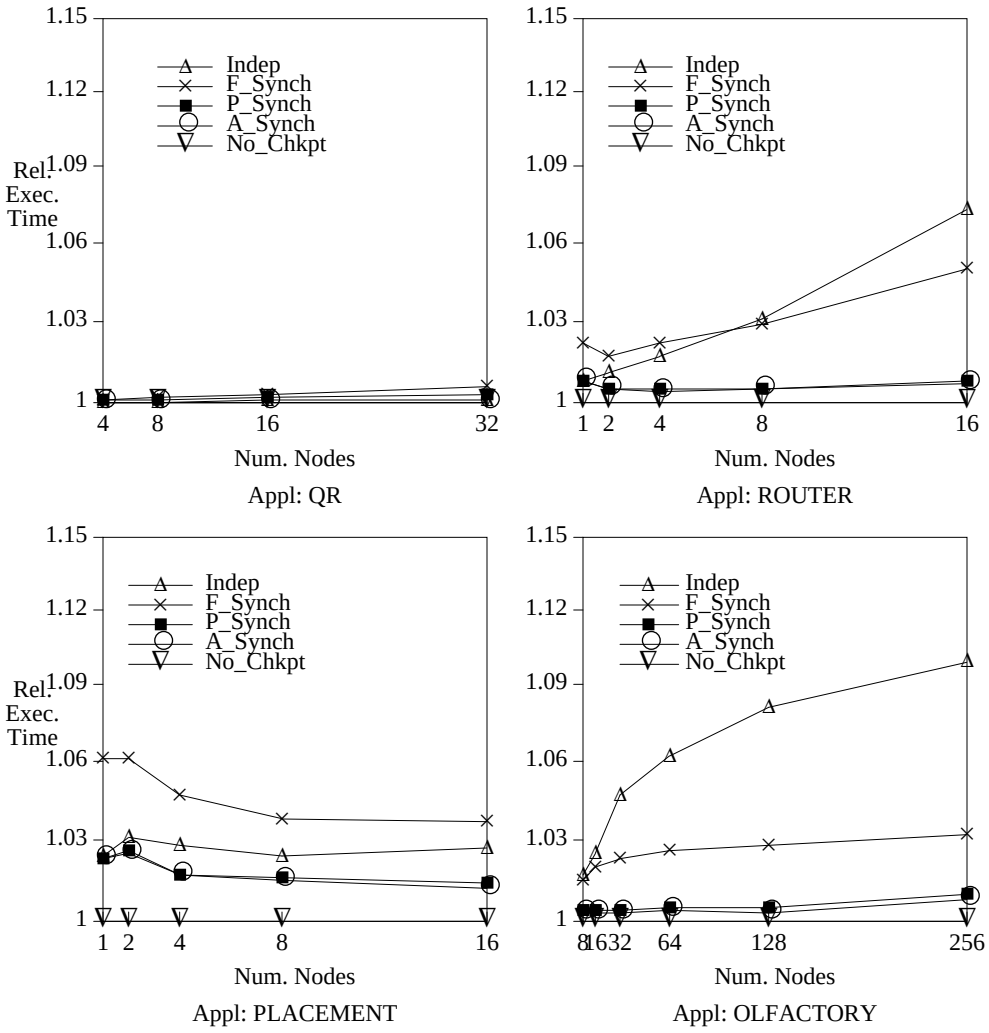


Figure 6.3: Application execution time with coordinated and independent checkpointing relative to application execution time without any recovery scheme. Checkpoints taken every 4600000 instructions.

communication frequency, total number of application messages sent and message volume. For all of the graphs in Figures 6.3 and 6.4 the disk and network bandwidths are (more than) sufficient to support both the applications and the simulated error recovery schemes, such that they do not contribute to any increase in the application's execution time. Sections 6.2.2.1 and 6.2.2.2, below, briefly

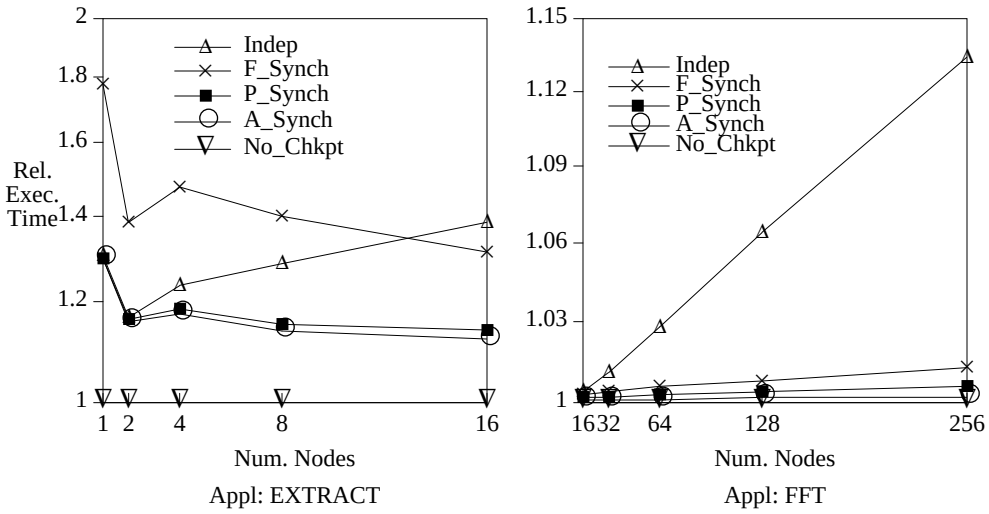


Figure 6.4: Continuation of Figure 6.3 (Note: with Extract checkpoints are taken every 14600000 instructions, instead of 4600000 instructions, and the y axis is different).

examine the network and disk bandwidth requirements of asynchronous coordinated checkpointing.

Table 6.7 indicates, for a given checkpoint frequency, the number of checkpoints taken by coordinated checkpointing schemes relative to *Indep*. By having processes checkpoint in groups, a checkpoint session is initiated by the first process that completes its checkpoint interval — resulting in checkpoints being taken, on average, more frequently than is dictated by the checkpoint interval. This table shows that, as the application is scaled, processes protected by coordinated checkpointing checkpoint more and more often compared to those protected by independent checkpointing. If the checkpoint interval can be tuned on a per application basis then this affect can be reduced. Table 6.7 also indicates the average size of the modified process state checkpointed by *A_Synch* and *Indep*; the checkpoint states are smaller for *A_Synch* because processes checkpoint more

	Checkpointing Statistics								
	Number of hypercube nodes								
	1	2	4	8	16	32	64	128	256
	Number of *A_Synch* Checkpoints / Number of *Indep* Checkpoints								
Extract†:	1.00	1.01	1.05	1.08	1.18	-	-	-	
OLF:	-	-	-	1.02	1.05	1.00	1.15	1.04	1.11
FFT:	-	-	-	-	1.05	1.12	1.18	1.20	1.48
QR:	-	-	1.01	1.03	1.16	1.21	-	-	-
Place:	1.14	1.13	1.24	1.74	1.81	-	-	-	-
Router:	1.00	1.03	1.10	1.11	1.18	-	-	-	-
	Avg. Size of Checkpoint State (1K Byte Pages) — *A_Synch* includes msg Qs								
Extract† *(Asy)*:	2.90K	1.60K	1.83K	1.55K	1.24K	-	-	-	-
(Ind):	2.86K	1.57K	1.85K	1.62K	1.42K	-	-	-	-
OLF *(Asy)*:	-	-	-	15.86	18.70	21.86	21.69	24.58	24.23
(Ind):	-	-	-	16.34	19.70	22.14	24.41	25.68	26.54
FFT *(Asy)*:	-	-	-	-	2.26	2.45	2.71	3.15	3.33
(Ind):	-	-	-	-	2.31	2.56	2.97	3.55	4.23
QR *(Asy)*:	-	-	1.16	1.31	1.52	1.88	-	-	-
(Ind):	-	-	1.15	1.30	1.58	2.07	-	-	-
Place *(Asy)*:	69.71	75.05	55.20	43.45	41.39	-	-	-	-
(Ind):	69.25	76.78	58.63	59.18	57.40	-	-	-	-
Router *(Asy)*:	25.67	22.88	27.47	33.66	42.08	-	-	-	-
(Ind):	25.53	23.52	30.00	37.45	48.99	-	-	-	-
	Avg. (Weighted) Size of Interacting Set (*A_Synch*)								
Extract† 2:	0.89	0.11	x	x	x	x	x	x	
4:	0.54	-	0.46	x	x	x	x	x	x
8:	0.38	-	-	0.62	x	x	x	x	x
16:	0.20	-	-	-	0.80	x	x	x	x
OLF 8-128:	size of interacting set = number of nodes								
256:	0.00	-	-	-	-	0.03	-	-	0.97
FFT 16:	0.34	-	-	-	0.66	x	x	x	x
32:	0.06	-	-	-	-	0.94	x	x	x
64:	0.10	-	-	-	-	-	0.90	x	x
128:	0.17	-	-	-	-	-	-	0.83	x
256:	0.20	-	-	-	-	-	-	-	0.80
QR,Place,Router:	size of interacting set = number of nodes (2, 4, 8, 16, or 32)								

Table 6.7: Checkpoint statistics for applications *Extract†* (checkpoint interval of 14600000 instructions) and *Olfactory*, *fft*, *QR*, *Placement*, *Router* (checkpoint interval of 4600000 instructions). Section one shows the number of checkpoints taken under *A_Synch* relative to the number of checkpoints taken with *Indep*. Section two gives the size of the checkpoint state for both schemes and Section three gives the average size of the interacting set (weighted by the total number of nodes) under *A_Synch*.

often. This mitigates, somewhat, the fact that processes checkpoint more often with coordinated checkpointing techniques. Lastly, the table shows the average weighted size of the interacting set in a checkpoint session. From this information we can see that, for the given checkpoint frequency, interacting sets tend to contain *all* of the processes in the application or, less frequently, just one of the processes.

Looking at Figure 6.3, none of the simulated error recovery schemes exhibit much overhead with QR. As can be seen in Table 6.7, QR has a very small working set: only one to two pages (per process) are checkpointed to stable storage and thus overhead is extremely low (less than one percent).

Router has significantly more state than *QR*, but overhead remains below ten percent for all simulated recovery schemes. A surprising result is that *Indep*, for 8 and 16 processes, performs worse than even *F_Synch*, the ‘‘fully synchronous’’ coordinated checkpointing scheme. It has generally been believed that independent checkpointing is a desirable feature, since each process need only be suspended the length of time it takes to copy its state — requiring no coordination with other processes. However, it is precisely this lack of coordination that results in greater overhead. As *Router* scales it becomes more communication intensive (see Table 6.2). As a result there is a greater probability that a process *P*, expecting a message from a process *Q*, will have to wait longer for that message because *Q* is in the midst of a checkpoint. With coordinated checkpointing *P* and *Q* have a high probability of checkpointing together (see Table 6.7) and thus overlapping much of their checkpoint overhead.

The impact of coordination upon the total overhead is quite significant since in *F_Synch* processes are suspended until all checkpoints in the interacting set have

been written and committed to disk, while with *Indep* each process is only suspended the length of time it takes to copy a single volatile checkpoint in local memory. In fact, in the ‘‘best case’’ our optimistic implementation of independent checkpointing would always perform at least as well as *A_Synch*, if we just use process suspension time as a predictor of checkpoint overhead. However our simulations show that this is erroneous and that it is in fact desirable for processes to coordinate their checkpointing. It is interesting to note that performance is good despite the fact that, with coordinated checkpointing, processes checkpoint more often than the checkpoint interval dictates. Further, table 6.7 shows that this shortening of the *actual* checkpoint interval gets worse as the application scales; the first process to reach the end of the checkpoint interval that initiates a checkpoint session — forcing other processes in the interacting set to checkpoint early. Nonetheless *A_Synch* overhead grows only slightly as the application scales and, if one can fine tune the checkpoint interval, this overhead can be reduced.

With *Placement* checkpointing overhead peaks at six percent for *F_Synch* and decreases as the application scales. All other schemes exhibit less overhead and show a similar trend. The reason for this is that the average number of dirty pages checkpointed to stable storage decreases as the application scales, which results in a stable or reduced level of overhead. This is also true for *Extract*, whose graph is similar to *Placement*. *Placement* is not communication-intensive compared to *Router* (see Table 6.2) and subsequently *Indep* performs better than *F_Synch* but still not as well as *P_Synch* and *A_Synch*.

Overhead with *Olfactory* is less than ten percent for all recovery schemes. *A_Synch* and *P_Synch* have very low overhead, and *Olfactory*’s rather small state

results in low overhead for *F_Synch*. Despite the fact that *Olfactory* is not nearly as communication-intensive as *Router*, *Indep* overhead goes up as the application scales and is always worse than even *F_Synch*. The overhead levels with *fft* are comparable. Overhead for the coordinated checkpointing schemes is very small due to *fft*'s tiny working set (see Table 6.7).

Extract's working set is 100 times the size of the other applications and this results in overhead of at least twelve percent and almost eighty percent in the worst case (*F_Synch* with one process). The checkpoint interval for *Extract* is 14600000 instructions and is limited by our assumption of 3 Mbytes/sec disk bandwidth per node; The disk can not handle a smaller interval. *Extract* is the least communication-intensive application but its large working set still results in *Indep* performing worse than *F_Synch* when run with 16 processes.

In all of the these simulations *A_Synch* and *P_Synch* performed almost the same. The main difference between the two algorithms is that with *P_Synch* processes are suspended until the interacting set is located. Thus process suspension time is dependent on network delays and the speed with which the remote operating system can respond to checkpoint requests. *A_Synch* overhead is independent of non-local activities. There are several reasons why this difference does not show up in our simulations. First, the operating system is modeled as a separate node *and* performs no duties aside from message routing and checkpointing — resulting in a quick response to checkpoint requests. Also, the network is very lightly loaded since only one application is running and there is no paging — resulting in fast message transmission. Thus the interacting set can be located quickly — amounting to less than one percent of total application runtime

in our simulations. However, we believe that independence from network delays and remote node computation is desirable. *A_Synch* will always perform at least as well as *P_Synch* with the only cost being an increase in algorithmic complexity during the checkpointing phase that locates the interacting set. Also *A_Synch* should scale well, while if network delays can be arbitrarily long *P_Synch* is not likely to scale as well. However, we do note that, if the network is congested, the application will take longer to execute and thus the *percentage* of overhead contributed by the error recovery scheme might *not* actually increase. We examine the performance of these recovery schemes under increased checkpoint frequency in Section 6.2.2.3.

6.2.2.1. Overhead Due to Network Bandwidth

Coordinated checkpointing techniques, as proposed in this dissertation, do not require additional network bandwidth during normal operation. In fact, if the detection of message transmissions errors is incorporated into the checkpointing algorithms as we have proposed, then less network bandwidth is needed compared to many alternative methods of implementing a reliable message delivery protocol (see Section 2.4). However, coordinated checkpointing does requiring the sending of many recovery scheme messages during checkpointing (and recovery) sessions. Table 6.8 indicates the total number of messages sent in each of the asynchronous coordinated checkpointing simulations. These totals are broken down into the percentage of application and recovery scheme messages. Recovery scheme messages make up anywhere from 0.5 to 23.4 percent of the total messages sent for *Router*, *Placement*, *fft*, *QR*, and *Olfactory* (excluding the single process case for

No. Nodes	Total Messages	Percent Recovery	Percent Application	No. Nodes	Total Messages	Percent Recovery	Percent Application
Router				*FFT*			
1	176	85.8	14.2	16	38235	16.1	83.9
2	67664	0.5	99.5	32	88088	9.0	91.0
4	127464	0.4	99.6	64	202268	4.9	95.1
8	327825	0.4	99.6	128	461165	2.7	97.3
16	718965	0.5	99.5	256	1047493	2.1	97.9
Placement				*Olfactory*			
1	873	0.0	1.0	8	3688	23.4	76.6
2	16549	4.0	96.0	16	8994	16.6	83.4
4	17544	4.6	95.4	32	21251	11.9	88.1
8	39567	8.8	91.2	64	50641	11.3	88.7
16	142084	7.7	92.3	128	115362	9.2	90.8
				256	262696	8.9	91.1
Matrix QR				*Extract*			
4	56757	2.5	97.5	1	193	57.5	42.5
8	56901	2.8	97.2	2	576	42.9	57.1
16	57197	3.2	96.8	4	1940	32.8	67.2
32	57445	3.5	96.5	8	4469	33.4	66.6
				16	11590	31.3	68.6

Table 6.8: Total messages sent in each simulation. Also given is the percentage of this total which is attributable to the error recovery algorithm (asynchronous coordinated checkpointing) and the percentage attributable to the application.

Router). The percentages are higher for *Extract* where the total number of messages sent is very low since *Extract* is very communication-unintensive. In general, the percentage of recovery scheme messages tends to decrease as the applications are scaled. No attempt to decrease or optimize the number of messages sent during checkpoint or recovery sessions has been made since these statistics have not proved to have any impact upon application or system performance under our assumptions.

Total Disk Bandwidth Required (Mbytes/sec)			
# Nodes		# Nodes	
	Router		*FFT*
1	4.434	16	1.142
2	1.846	32	1.415
4	1.893	64	1.919
8	3.003	128	2.838
16	5.498	256	5.166
	Placement		*Olfactory*
1	9.702	8	1.425
2	13.055	16	2.444
4	5.805	32	4.290
8	9.206	64	8.515
16	14.222	128	16.087
	Matrix QR		*Extract*
4	0.214	1	326.904
8	0.252	2	283.597
16	0.343	4	298.140
32	0.487	8	355.830

Table 6.9: The *total* disk bandwidth required to support asynchronous coordinated checkpointing (in Mbytes/sec).

6.2.2.2. Overhead Due to Disk Bandwidth

Since most of the applications have small process checkpoint states (see Table 5.1) the disk bandwidth requirements of asynchronous coordinated checkpointing are very low (see Table 6.9). *Extract* is the only exception (where each process checkpoint state is roughly 2.7 Mbytes). For *Extract* the disk bandwidth requirements of asynchronous coordinated checkpointing are quite similar to asynchronous message logging; for the rest of the applications

coordinated checkpointing requires *significantly* less disk bandwidth than message logging.

6.2.2.3. Simulations with Increased Checkpoint Frequency

To look more closely at the potential difference between the partially-synchronous (*P_Synch*) and asynchronous (*A_Synch*) coordinated checkpointing algorithms, within the framework of our simulator, we shortened the checkpoint interval to 50K instructions and then 5K instructions for the following four applications: *Matrix QR*, *Router*, *fft*, and *Olfactory*. This increases the total overhead of checkpointing and shows the contribution of interacting set determination time to this overhead. These simulation results are shown in Figures 6.5 and 6.6.

QR (with 50K instructions checkpoint interval) still has quite low overhead — no more than 4.5 percent (*F_Synch* with 16 processes) — due to a small process state. *Indep* and *A_Synch* have identical overhead at .4 percent for all *QR* simulations. *P_Synch* overhead increases somewhat, with the contribution of interacting set determination being .2, .7, 1.0, and 2.0 percent as the application scales through 4, 8, 16 and 32 nodes. When the checkpoint interval decreases to 5K instructions *Indep* stays at a constant 3.3 percent while *A_Synch* overhead starts at 3.9 percent for 4 nodes and goes up to 5.0 for 32 nodes. This is because 1) the small checkpoint interval results in smaller interacting sets (54 to 91 percent have only two processes) and thus there is less overlap of checkpointing overhead and more interference with normal processing and 2) the fact that processes checkpoint more often than the checkpoint interval dictates. *If* the checkpoint interval can be

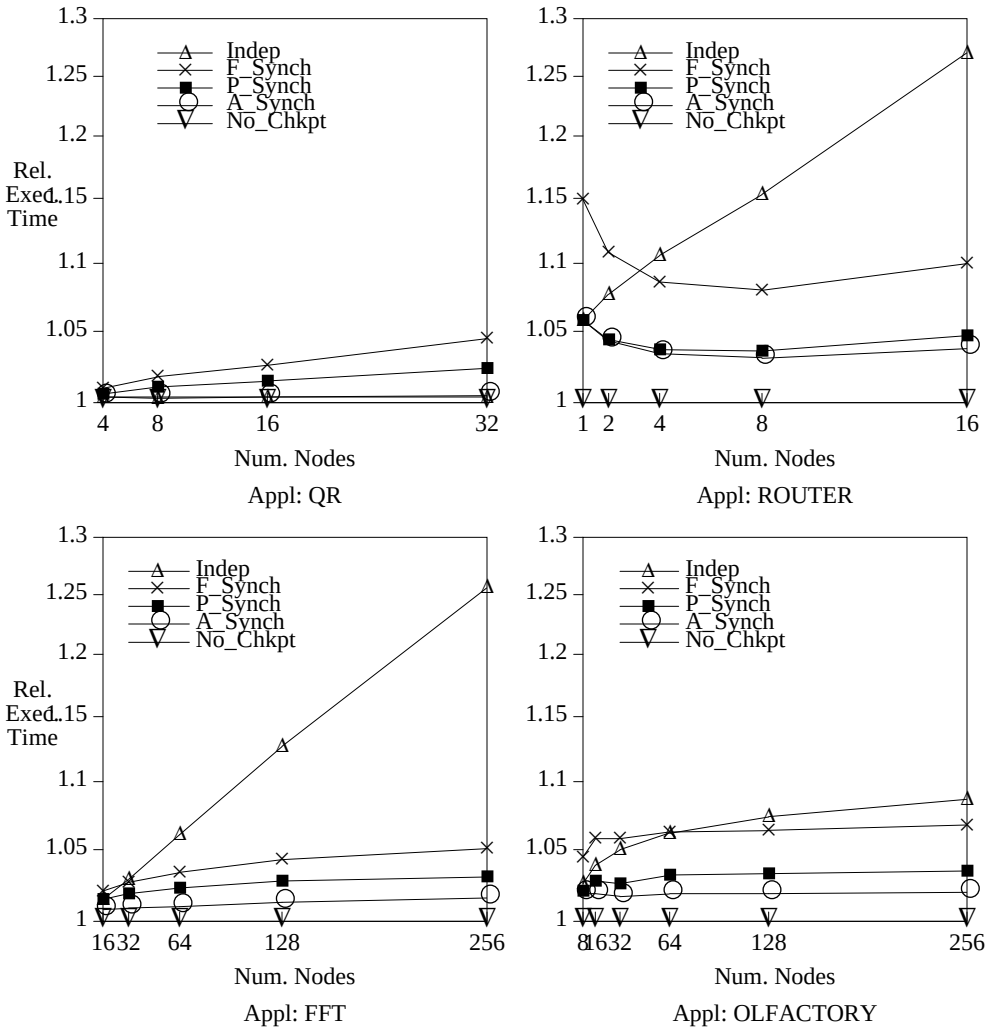


Figure 6.5: Application execution time with coordinated and independent checkpointing relative to application execution time without any recovery scheme. Checkpoints are taken every 50000 instructions.

tuned then, in the ‘‘worst’’ case, as the checkpoint frequency is increased, asynchronous coordinated checkpointing will ‘‘become’’ independent checkpointing (with interacting sets of only one process) and the overhead of the two algorithms will be comparable for this application.

As the checkpoint frequency increases *Router* results remain similar to our

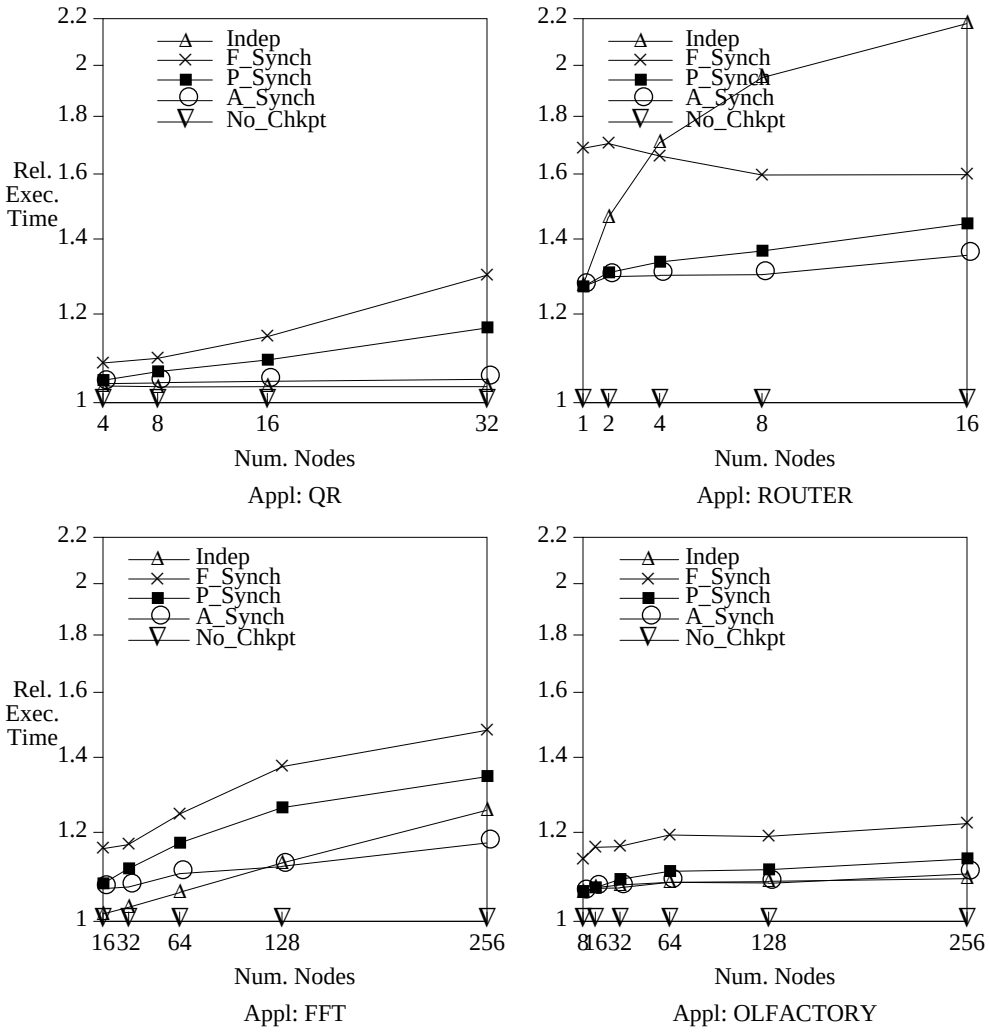


Figure 6.6: Application execution time with coordinated and independent checkpointing relative to application execution time without any recovery scheme. Checkpoints are taken every 5000 instructions.

original simulation results. Even with a 5K instruction checkpoint interval the interacting set contains all processes 87 to 100 percent of the time due to the high communication frequency of the *Router* application. Hence *A_Synch* performs best in all cases and *Indep* has the highest level of overhead. The overhead contribution of interacting set determination is at most 0.7 percent (50K interval with 16

	Checkpointing Statistics (Checkpoint interval of 5000 instructions)								
	Number of hypercube nodes								
	1	2	4	8	16	32	64	128	256
	Number of *A_Synch* Checkpoints / Number of *Indep* Checkpoints								
OLF:	-	-	-	1.00	1.01	1.00	1.03	1.02	1.11
FFT:	-	-	-	-	1.05	1.12	1.18	1.20	1.48
QR:	-	-	1.01	1.03	1.16	1.21	-	-	-
Router:	1.00	1.03	1.10	1.11	1.18	-	-	-	-
	Avg. Size of Checkpoint State (1K Byte Pages) — *A_Synch* includes msg Qs								
OLF *(Asy):*	-	-	-	2.00	2.01	2.01	2.02	2.04	
(Ind):	-	-	-	2.00	2.01	2.01	2.02	2.04	2.53
FFT *(Asy):*	-	-	-	-	2.26	2.45	2.71	3.15	3.33
(Ind):	-	-	-	-	2.31	2.56	2.97	3.55	4.23
QR *(Asy):*	-	-	1.16	1.31	1.52	1.88	-	-	-
(Ind):	-	-	1.15	1.30	1.58	2.07	-	-	-
Router *(Asy):*	25.67	22.88	27.47	33.66	42.08	-	-	-	-
(Ind):	25.53	23.52	30.00	37.45	48.99	-	-	-	-
	Avg. (Weighted) Size of Interacting Set (*A_Synch*)								
OLF 8:	0.71	0.19	0.09	0.01	x	x	x	x	x
16:	0.54	0.24	0.14	0.06	0.02	x	x	x	x
32:	0.40	0.30	0.09	0.10	0.09	0.02	x	x	x
64:	0.24	0.34	0.14	0.10	0.12	0.05	0.01	x	x
128:	0.17	0.33	0.15	0.09	0.09	0.09	0.06	0.02	x
256:	0.11	0.39	0.12	0.07	0.06	0.06	0.14	0.04	0.01
FFT 16:	0.53	0.17	0.26	0.04	-	x	x	x	x
32:	0.28	0.17	0.17	0.17	0.17	0.04	x	x	x
64:	-	-	-	0.31	0.18	0.17	0.34	x	x
128:	-	-	-	-	-	-	-	1.00	x
256:	0.01	-	-	-	-	-	-	-	0.99
QR 4:	0.89	0.11	x	x	x	x	x	x	
8:	0.54	-	0.46	x	x	x	x	x	x
16:	0.38	-	-	0.62	x	x	x	x	x
32:	0.20	-	-	-	0.80	x	x	x	x
ROUTER: 2:	-	-	x	x	x	x	x	x	x
4:	-	-	-	x	x	x	x	x	x
8:	-	-	-	-	x	x	x	x	x
16:	-	-	-	-	-	x	x	x	x

Table 6.10: Checkpoint statistics for *Olfactory*, *fft*, *QR*, *Router* (checkpoint interval of 5000 instructions). Section one shows the number of checkpoints taken under *A_Synch* relative to the number of checkpoints taken with *Indep*. Section two gives the size of the checkpoint state for both schemes and Section three gives the average size of the interacting set (weighted by the total number of nodes) under *A_Synch*.

processes) and 7.2 percent (5K interval with 16 processes) which is 15 and 17 percent of the total *P_Synch* overhead, respectively.

With *FFT* approximately half of *P_Synch*'s overhead (1 percent with the 50K checkpoint interval and 1 to 11 percent with the 5K interval) comes from the time it takes to determine the interacting set. The large relative percentage is due to *FFT*'s small process state. With a 5K checkpoint interval *A_Synch* has smaller interacting set sizes - especially for 16, 32 and 64 nodes, but *A_Synch* exhibits overhead comparable to *Indep* with 128 nodes and less with 256 nodes. *Indep*'s steeper curve is probably due to the fact that *FFT* becomes increasingly communication intensive as the application scales.

Olfactory is not very communication-intensive compared to *Router* and *FFT*. So while *Indep* does not perform as well as the coordinated checkpointing schemes with a checkpoint interval of 50K instructions, the breakdown of the interacting set size and larger number of checkpoints experienced by *A_Synch* evens up the overhead with *Indep* using a 5K checkpoint interval. Locating the interacting set contributes .5 to 2.5 percent to *P_Synch* overhead for 50K interval and .5 to 3.4 percent for 5K interval.

Table 6.10 is the same as Table 6.7 but for a checkpoint interval of 5000 instructions. This table indicates the number of checkpoints taken by coordinated checkpointing schemes relative to *Indep*; the average size of the modified process state checkpointed by *A_Synch* and *Indep*; and the average weighted size of the interacting set in a checkpoint session (which has broken down considerably compared to the checkpoint interval of 4600000 instructions).

While *Indep* exhibits overhead comparable to *A_Synch* for applications *QR*

and for the 5K checkpoint interval simulations using *Olfactory*, *A_Synch* has a more consistent level of overhead that appears to scale well even as the checkpoint interval is significantly decreased. This indicates that it may be useful in soft real-time systems due to rather predictable overhead levels. While *P_Synch* and *A_Synch* perform nearly the same in our original (46M instructions checkpoint interval) simulations, decreasing the interval shows that finding the interacting set contributes anywhere from a negligible amount to 11 percent of overhead on its own. We believe that *A_Synch* is a ‘‘robust’’ implementation of coordinated checkpointing whose performance will not degrade much with network congestion or scaling. If the checkpoint interval is short enough that most interacting sets are small relative to the number of processes in the application, then coordinated checkpointing is likely to perform similar to *Indep*.

As mentioned previously, with *A_Synch* the sending of a message may have to be temporarily delayed if the (checkpointing) process is attempting to send the message outside of the (locally known) interacting set, before that interacting set has been found (see Section 4.5.1). In our simulations very few application messages, generally less than 0.1 percent, 0.3 in the worst case, needed to be temporarily blocked. Since all of the six applications used nonblocking sends, this delay only affected how soon the message could be received by the destination process. This appears to have had an insignificant impact on the total overhead of asynchronous coordinated checkpointing.

Chapter Seven
Conclusions

We have described several application-transparent error-recovery techniques which can be used to provide fault tolerance for large general-purpose multicomputers. Message logging techniques have the advantage of fast checkpoint and recovery sessions but may be inappropriate for high-performance multicomputer systems running communication-intensive applications due to the overhead of 1) operating system processing required to maintain and transmit bookkeeping information with each application message and 2) logging messages and bookkeeping information to disk (or local memory). Furthermore, message logging can only be used to provide fault tolerance for processes which are deterministic in their actions. Global checkpointing techniques are likely to incur the lowest level of overhead during normal operation but are only appropriate for use with applications that can tolerate the possible loss of many minutes of computation during recovery (e.g. batch applications). Coordinated checkpointing of interacting sets is a ‘‘compromise’’ between message logging and global checkpointing and therefore shows the most promise in providing error recovery for high-performance multicomputers that run both communication- and computation-intensive applications. Coordinated checkpointing techniques minimize the overhead incurred during normal operation at the cost of a potentially expensive recovery. In the worst case, the entire application may need to be rolled back as a result of the failure of one node. However, the overhead incurred during checkpointing is dependent almost entirely on the checkpoint frequency which can

be tuned to the specific needs of each application. Further, any application process which must be rolled back during a recovery session is always rolled back to its most recent (committed) checkpoint.

A correctness model was presented and used as the foundation for a complete set of proofs for the new asynchronous message logging algorithm and the new coordinated checkpointing algorithms.

The presented message logging algorithm is distributed, application-transparent, and n-fault-tolerant. Process checkpoints and interprocess messages are logged to stable storage asynchronous of process execution. Messages are also temporarily logged in the local volatile memory of the sender. This allows the algorithm to use volatile message logs in addition to stable message logs in order to recover as much of the failed process(es)' pre-failure execution as possible. The use of these volatile message logs reduces the number of non-failed processes (orphans) which must roll back in order to achieve a consistent system state. We were motivated to develop this algorithm when a previously published message logging algorithm failed to perform correctly when implemented on our simulation testbed. One of the major contributions of this chapter was the presentation of a complete proof that this optimized algorithm is *correct* - in the presence of failures the algorithm can be used to recover one or more processes in a distributed application without violating the application's partial ordering of events and without duplicating or deleting any events.

We have also presented two new distributed error recovery schemes, based on coordinated checkpointing of interacting sets of processes, that minimize disruptions to normal processing due to checkpointing. In the synchronous

coordinated checkpointing scheme an interacting set of processes is suspended from normal computation until all of the process states have been committed to stable storage. The asynchronous coordinated checkpointing scheme is based on the (coordinated) volatile checkpointing of interacting sets of processes followed by copying of these checkpoints to stable storage ‘‘in the background’’ involving minimal interference with normal computation. The asynchronous algorithm is amenable to highly-effective optimizations that can be supported with relatively small enhancements to a conventional virtual memory system. With these optimizations, a volatile checkpoint can be taken by only a few dozen instructions without physically copying any pages in local memory. The cost of asynchronous coordinated checkpointing, compared to the synchronous scheme, is an increase in the complexity of the checkpointing algorithm — which must ensure that the volatile checkpoints are saved as part of a consistent global state even though the processes being checkpointed have resumed execution. Proofs of correctness were provided for both coordinated checkpointing schemes.

Both coordinated checkpointing schemes are integrated with an efficient error detection mechanism that avoids the need for message acknowledgments, transmission of message sequence numbers, and transmission of check bits with each message. Both schemes can recover from multiple node failures, lost messages, and corrupted messages; the domino effect can not occur and only one generation of checkpoints needs to be maintained on stable storage. Multiple checkpointing and recovery sessions may be active in the system simultaneously, operating independently if possible and merging correctly when necessary.

Realistic performance analysis of distributed checkpointing and rollback

techniques is very difficult due to all of the factors that must be taken into account (e.g. parameterization of the target system and of distributed applications). Thus, most performance analyses of distributed recovery techniques have been limited to greatly simplified analytical models. Realistic performance evaluations have appeared only recently, with almost no analysis in the context of multicomputer systems — where low overhead and scalability are highly desirable features in an error recovery scheme. Our goal was to measure the impact of error recovery techniques on the execution time of actual distributed applications as the applications are scaled. We built a Distributed Error Recovery Testbed (DERT) on top of an event-driven, message-passing, object-oriented, multithreaded simulation kernel in order to achieve accurate results, reasonable efficiency, and maximum flexibility, without consuming large amounts of resources. This testbed simulates a scalable multicomputer (up to 256 nodes) and runs on SPARC workstations; it runs actual distributed applications whose assembly code has been instrumented to 1) count the number of assembly instructions executed and 2) take checkpoints at precise intervals, saving only the modified application state. Several recovery schemes can be simulated on the testbed. These recovery schemes are fully functional; an application will execute correctly in the presence of simulated node failures. We have described, in detail, the functionality of the testbed, the difficulties we encountered during development, and our experiences simulating applications and recovery schemes.

DERT has been used to measure the impact of several recovery schemes and optimizations, on the execution time of both coarse- and fine-grained applications during error-free operation. Using this execution-driven simulator we have shown

that the message handling overhead required by asynchronous message logging algorithms can have a substantial impact on application execution time. This overhead increases dramatically for more communication-intensive applications as they are scaled (up to 16, 32, or 256 processes, with constant problem size). Two optimizations we proposed significantly reduced recovery overhead, while a third optimization, previously suggested by others, often increased overhead. However, asynchronous message logging, even with the optimizations, still scaled very poorly for three of six applications. Additional simulations showed that hardware support may be required to eliminate this trend.

Three coordinated checkpointing algorithms were also coded into the simulator: fully-synchronous, partially-synchronous, and asynchronous. These algorithms were compared to one another and to independent checkpointing, without message logging, by measuring their impact on application execution time in a system without failures. Both the partially-synchronous and asynchronous coordinated checkpointing algorithms add less than six percent overhead to the application's execution time for five of the six applications (for a fixed checkpoint interval); independent checkpointing added less than fourteen percent. The sixth application's higher level of overhead (less than 20 percent for partially-synchronous and asynchronous coordinated checkpointing, less than 40 percent for independent checkpointing) is due to that application's larger working set. This overhead could be significantly reduced, for the asynchronous coordinated or independent checkpointing algorithms, if the copy-on-write optimization is available. In all cases, the percentage of overhead incurred by asynchronous coordinated checkpointing is nearly independent of the size of the multicomputer

(with constant problem size). Asynchronous coordinated checkpointing also exhibits less increase in overhead, compared with the other algorithms, as checkpoint frequency is increased.

In many simulations, the coordinated checkpointing of multiple processes resulted in significantly *less* overhead than independent checkpointing of processes. Independent checkpointing is usually assumed to have lower overhead because no coordination of checkpointing sessions is required and because coordinated checkpointing results in processes being checkpointed more often than is dictated by the checkpoint interval. This assumed lower overhead makes independent checkpointing attractive even though it may suffer from the domino effect during recovery. However, we found that coordinated checkpointing actually performed better than independent checkpointing in many cases because processes are more likely to be checkpointing at the same time, overlapping much of their checkpoint overhead, and doing useful work at the same time — actually resulting in *less* total overhead.

In conclusion, using the proposed techniques it is possible to use, in particular, asynchronous coordinated checkpointing to implement a highly reliable, general-purpose, large multicomputer system in which the fault tolerance characteristics involve minimal performance overhead and are completely transparent to the user.

Bibliography

[Aham89] M. Ahamad and L. Lin, ‘‘Using Checkpoints to Localize the Effects of Faults in Distributed Systems,’’ *8th Symposium on Reliable Distributed Systems,* pp. 2-11 (October 1989).

[Aviz85] A. Avizienis, ‘‘The N-Version Approach to Fault-Tolerant Software,’’ *IEEE Transactions on Software Engineering* **SE-11**(12) pp. 629-638 (December 1985).

[Bari83] G. Barigazzi and L. Strigini, ‘‘Application-Transparent Setting of Recovery Points,’’ *Digest FTCS-13,* pp. 48-55 (June 1983).

[Bert87] D. Bertsekas and R. Gallager, *Data Networks,* Prentice Hall (1987).

[Bhar88] B. Bhargava and S.-R. Lian, ‘‘Independent Checkpointing and Concurrent Rollback for Recovery in Distributed Systems - An Optimistic Approach,’’ *7th Symposium on Reliable Distributed Systems,* pp. 3-12 (October 1988).

[Bhar90] B. Bhargava, S.-R. Lian, and P.-J. Leu, ‘‘Experimental evaluation of concurrent checkpointing and rollback-recovery algorithms,’’ *Proceedings of the International Conference on Data Engineering*, pp. 182-189 (March, 1990).

[Bobr72] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, ‘‘TENEX, a Paged Time Sharing System for the PDP-10,’’ *Communications of the ACM* **15**(3) pp. 135-143 (March 1972).

[Borg83] A. Borg, J. Baumbach, and S. Glazer, ‘‘A Message System

Supporting Fault Tolerance,'' *Proceedings 9th Symposium on Operating System Principles*, pp. 90-99 (October 1983).

[Bowe88] J. Bower, M. Nelson, M. Wilson, G. Fox, and W. Furmanski, ''Piriform (Olfactory) Cortex Model on the Hypercube,'' *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pp. 977-999 (January 1988).

[Chan75] K. M. Chandy, ''A Survey of Analytic Models of Rollback and Recovery Strategies,'' *IEEE Computer* **8**(5) pp. 40-47 (May 1975).

[Chan85] K. M. Chandy and L. Lamport, ''Distributed Snapshots: Determining Global States of Distributed Systems,'' *ACM Transactions on Computer Systems* **3**(1) pp. 63-75 (February 1985).

[Chin83] F. Chin and K. V. S. Ramarao, ''Optimal Termination Protocols for Network Partitioning,'' *ACM Symposium on Principles of Distributed Databases*, pp. 25-35 (1983).

[Dall87] W. J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills, ''Architecture of a Message-Driven Processor,'' *14th Annual Symposium on Computer Architecture*, pp. 189-196 (June 1987).

[Dall88] W. J. Dally, ''Fine-Grain Message Passing Concurrent Computers,'' *Proceedings of the Third Conference on Hypercube Concurrent Computers* **1** pp. 2-12 (January 1988).

[Davi92] A. Davis, ''Mayfly: A Scaleable, Parallel Processing System,'' *List and Symbolic Computation* **5**(1/2) pp. 7-47 (May 1992).

[Dijk80] E. W. Dijkstra and C. S. Scholten, ‘‘Termination Detection for Diffusing Computations,’’ *Information Processing Letters* **11**(1) pp. 1-4 (August 1980).

[Eick92] T. v. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, ‘‘Active Messages: A Mechanism for Integrated Communication and Computation,’’ *19th Annual International Symposium on Computer Architecture*, pp. 256-266 (May 1992).

[Elki82] S. A. Elkind, ‘‘Reliability and Availability Techniques,’’ pp. 63-181 in *The Theory and Practice of Reliable System Design*, ed. D. P. Siewiorek and R. S. Swarz, Digital Press (1982).

[Elno92a] E. Elnozahy and W. Zwaenepoel, ‘‘Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit,’’ *IEEE Transactions on Computers* **41**(5) pp. 526-531 (May 1992).

[Elno92b] E. Elnozahy, D. Johnson, and W. Zwaenepoel, ‘‘The Performance of Consistent Checkpointing,’’ *11th Symposium on Reliable Distributed Systems*, pp. 39-47 (October 1992).

[Fraz89] T. M. Frazier and Y. Tamir, ‘‘Application-Transparent Error-Recovery Techniques for Multicomputers,’’ *The Fourth Conference on Hypercube Concurrent Computers and Applications*, (March 1989).

[Fuji92] R. M. Fujimoto, J.-J. Tsai, and G. C. Gopalakrishnan, ‘‘Design and Evaluation of the Rollback Chip: Special Purpose Hardware for Time Warp,’’ *IEEE Transactions on Computers* **41**(1) pp. 68-82

(January 1992).

[Gray81] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger, ‘‘The recovery manager of the System R Database Manager,’’ *Computing Surveys* **13**(2) pp. 223-242 (June 1981).

[Hori93] T. Horie, K. Hayashi, T. Shimizu, and H. Ishihata, ‘‘Improving AP1000 Parallel Computer Performance with Message Communication,’’ *20th Annual International Symposium on Computer Architecture*, pp. 314-325 (May 1993).

[Hoss83] S. H. Hosseini, J. G. Kuhl, and S. M. Reddy, ‘‘An Integrated Approach to Error Recovery in Distributed Computing Systems,’’ *Digest FTCS-13*, pp. 56-63 (June 1983).

[Hsu90] J.-M. Hsu and P. Banerjee, ‘‘Performance Measurement and Trace Driven Simulation of Parallel CAD and Numeric Applications on a Hypercube Multicomputer,’’ *17th Annual International Symposium on Computer Architecture*, pp. 260-269 (May 1990).

[Jage87] W. J. Jager and W. M. Loucks, ‘‘The P-MACHine: A Hardware Message Accelerator for a Multiprocessor System,’’ *1987 International Conference on Parallel Processing*, pp. 600-609 (August 1987).

[Jalo89] P. Jalote, ‘‘Fault Tolerant Processes,’’ *Distributed Computing* **3**(4) pp. 187-195 (Sept. 1989).

[John87] D. B. Johnson and W. Zwaenepoel, ‘‘Sender-Based Message

Logging,'' *Digest FTCS-17*, pp. 14-19 (July 1987).

[John88] D. B. Johnson and W. Zwaenepoel, ''Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing,'' *7th Annual ACM Symposium on Principles of Distributed Computing*, pp. 171-181 (August 1988).

[John90a] D. B. Johnson and W. Zwaenepoel, ''Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing,'' *Journal of Algorithms* **11**(3) pp. 462-491 (September 1990).

[John90b] D. B. Johnson and W. Zwaenepoel, ''Distributed System Fault Tolerance Using Sender-Based Message Logging,'' COMP TR90-119, Rice University, Houston, TX (May 1990).

[Kim93] J. L. Kim and T. Park, ''An Efficient Protocol for Checkpointing Recovery in Distributed Systems,'' *IEEE Transactions on Parallel and Distributed Systems* **4**(8) pp. 955-960 (August 1993).

[Kim84] K. H. Kim, ''Distributed Execution of Recovery Blocks: An Approach to Uniform Treatment of Hardware and Software Faults,'' *Proceedings 4th International Conference on Distributed Computing Systems*, pp. 526-532 (May 1984).

[Kim86] K. H. Kim, J. H. You, and A. Abouelnaga, ''A Scheme for Coordinated Execution of Independently Designed Recoverable Distributed Processes,'' *Digest FTCS-16*, pp. 130-135 (June 1986).

[Koo87] R. Koo and S. Toueg, ''Checkpointing and Rollback-Recovery for Distributed Systems,'' *IEEE Transactions on Software Engineering*

SE-13(1) pp. 23-31 (January 1987).

[Kubi82] C. Kubiak, J. P. Andre, B. Grandjean, D. Mathieu, and J. Rolland, ‘‘Penelope: A Recovery Mechanism for Transient Hardware Failures and Software,’’ *Digest FTCS-12*, pp. 127-133 (June 1982).

[Lamp78] L. Lamport, ‘‘Time, Clocks, and the Ordering of Events in a Distributed System,’’ *Communications of the ACM* **21**(7) pp. 558-565 (July 1978).

[Lang82] C. R. Lang, ‘‘The Extension of Object-Oriented Languages to a Homogeneous Concurrent Architecture,’’ Ph.D. Dissertation, California Ins. of Tech., 5014:TR:82, California Institute of Technology (1982).

[Lee84] Y.-H. Lee and K. G. Shin, ‘‘Design and Evaluation of a Fault-Tolerant Multiprocessor Using Hardware Recovery Blocks,’’ *IEEE Transactions on Computers* **c-33**(2) pp. 113-124 (February 1984).

[Li90] K. Li, J. F. Naughton, and J. S. Plank, ‘‘Real-Time, Concurrent Checkpointing for Parallel Programs,’’ *Symposium on Principles and Practice of Parallel Programming*, pp. 79-88 (March 1990).

[Lisk87] B. N. Liskov, R. Scheifler, E. Walker, and W. Weihl, ‘‘Orphan Detection,’’ *Digest FTCS-17*, pp. 2-7 (July 1987).

[Nett86] E. Nett, R. Kroger, and J. Kaiser, ‘‘Implementing a General Error Recovery Mechanism in a Distributed Operating System,’’ *Digest FTCS-16*, pp. 124-129 (July 1986).

[Noak93] M. D. Noakes, D. A. Wallach, and W. J. Dally, ‘‘The J-Machine

Multicomputer: An Architectural Evaluation,'' *20th Annual International Symposium on Computer Architecture*, pp. 224-235 (May 16-19, 1993).

[Nuth92] P. R. Nuth and W. J. Dally, ''The J-machine Network,'' *International Conference on Computer Design*, pp. 420-423 (October 1992).

[Peat81] G. Peattie, ''Quality Control for ICs,'' *IEEE Spectrum* **18**(10) pp. 93-97 (October 1981).

[Powe83] M. L. Powell and D. L. Presotto, ''Publishing: A Reliable Broadcast Communication Mechanism,'' *Proceedings 9th Symposium on Operating Systems Principles*, pp. 100-109 (October 1983).

[Rand75] B. Randell, ''System Structure for Software Fault Tolerance,'' *IEEE Transactions on Software Engineering* **SE-1**(2) pp. 220-232 (June 1975).

[Rand78] B. Randell, P. A. Lee, and P. C. Treleaven, ''Reliability Issues in Computing System Design,'' *Computing Surveys* **10**(2) pp. 123-165 (June 1978).

[Reed87] D. A. Reed and R. M. Fujimoto, *Multicomputer Networks: Message-Based Parallel Processing*, The MIT Press (1987).

[Russ80] D. L. Russell, ''State Restoration in Systems of Communicating Processes,'' *IEEE Transactions on Software Engineering* **SE-6**(2) pp. 183-194 (March 1980).

[Schn85] F. B. Schneider and L. Lamport, ''Paradigms for Distributed

Programs,'' pp. 431-480 in *Lecture Notes in Computer Science 190*, ed. M. Paul and H. J. Seigert, Springer-Verlag, Berlin (1985).

[Seit85] C. L. Seitz, ''The Cosmic Cube,'' *Communications of the ACM* **28**(1) pp. 22-33 (January 1985).

[Shav86] N. Shavit and N. Francez, ''A New Approach to Selection of Locally Indicative Stability,'' *13th ICALP (Lecture Notes in Computer Science 226)*, pp. 344-358 Springer-Verlag, (1986).

[Sist89] A. P. Sistla and J. L. Welch, ''Efficient Distributed Recovery Using Message Logging,'' *PODC'89*, pp. 223-238 (August 1989).

[Skee81] D. Skeen, ''Nonblocking Commit Protocols,'' *Proceedings ACM-SIGMOD Conference on Management of Data*, pp. 133-147 (1981).

[Sper93] E. Spertus, S. C. Goldstein, K. E. Schauser, T. V. Eicken, and Et.al., ''Evaluation of Mechanisms for Fine-grained Parallel Programs in the J-Machine and the CM-5,'' *20th Annual International Symposium on Computer Architecture*, pp. 302-313 (May 1993).

[Stro85] R. E. Strom and S. Yemini, ''Optimistic Recovery in Distributed Systems,'' *ACM Transactions on Computer Systems* **3**(3) pp. 204-226 (August 1985).

[Stro88] R. E. Strom, D. F. Bacon, and S. Yemini, ''Volatile Logging in n-Fault-Tolerant Distributed Systems,'' *Digest FTCS-18*, pp. 44-49 (June 1988).

[Sull77] H. Sullivan and T. R. Brashkow, ''A Large Scale Homogeneous

Machine,'' *4th Annual International Symposium on Computer Architecture*, pp. 105-124 (1977).

[Swop86] S. M. Swope and R. M. Fujimoto, ''Simon II Kernel Reference Manual,'' Technical Report UUCS 86-001, University of Utah, Salt Lake City, UT (March 1986).

[Syst89] M. C. Systems, *RISCompiler and C programmer's Guide*. 1989.

[Tami83] Y. Tamir and C. H. Séquin, ''Self-Checking VLSI Building Blocks for Fault-Tolerant Multicomputers,'' *International Conference on Computer Design*, pp. 561-564 (November 1983).

[Tami84] Y. Tamir and C. H. Séquin, ''Error Recovery in Multicomputers Using Global Checkpoints,'' *13th International Conference on Parallel Processing*, pp. 32-41 (August 1984).

[Tami87] Y. Tamir and E. Gafni, ''A Software-Based Hardware Fault Tolerance Scheme for Multicomputers,'' *1987 International Conference on Parallel Processing*, pp. 117-120 (August 1987).

[Tami89] Y. Tamir and T. Frazier, ''Application-Transparent Process-Level Error Recovery for Multicomputers,'' *Hawaii International Conference on System Sciences-22*, (January 1989).

[Tami91] Y. Tamir and T. M. Frazier, ''Error-Recovery in Multicomputers Using Asynchronous Coordinated Checkpointing,'' Computer Science Department Technical Report CSD-910066, University of California, Los Angeles, CA (September 1991).

[Tane81] A. S. Tanenbaum, *Computer Networks*, Prentice Hall (1981).

[Tong92] Z. Tong, R. Kain, and W. Tsai, ‘‘Rollback Recovery in Distributed Systems Using Loosely Synchronized Clocks,’’ *IEEE Transactions on Parallel and Distributed Systems* **3**(2) pp. 246-251 (March 1992).

[Wang92] Y.-M. Wang and W. K. Fuchs, ‘‘Optimistic Message Logging for Independent Checkpointing in Message-Passing Systems,’’ *11th Symposium on Reliable Distributed Systems*, (October 1992).

[Whit85] C. Whitby-Strevens, ‘‘The Transputer,’’ *12th Annual Symposium on Computer Architecture*, pp. 292-300 (June 1985).

[Wood86] M. H. Woods, ‘‘MOS VLSI Reliability and Yield Trends,’’ *Proceedings of the IEEE* **74**(12) pp. 1715-1729 (December 1986).

[Wood81] W. G. Wood, ‘‘A Decentralised Recovery Control Protocol,’’ *Digest FTCS-11*, pp. 159-164 (June 1981).