Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

OPTIMISTIC ALGORITHMS FOR DISTRIBUTED TRANPARENT PROCESS REPLICATION

Arthur Paul Goldberg

July 1991
CSD-910050

UNIVERSITY OF CALIFORNIA

Los Angeles

Optimistic Algorithms for Distributed Transparent Process Replication

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

Arthur Paul Goldberg

1991

The dissertation of Arthur Paul Goldberg is approved.

Nathaniel Grossman

Nathaniel Grossman

[illegible]

Kirby Baker

Jack Carlyle

Jack Carlyle

[illegible]

Mario Gerla

[illegible]

David R. Jefferson, Committee Chair

University of California, Los Angeles

1991

I dedicate my dissertation to my parents, Alan and Barbara, and to my two loves, Meira Weinzweig and La Meira.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

David R. Jefferson has been my advisor for the last 7 years at UCLA, guiding me patiently forward. I thank David for giving his time generously and teaching me how to do research.

I thank the other members of my committee for their time. D. Stott Parker, Algirdas Avizienis and Rosser Nelson were on the qualifying oral committee. I thank Mario Gerla, Jack Carlyle and Nathaniel Grossman for attending my final defense on short notice. And I thank Kirby Baker for remembering me from one presentation to the next!

Shaula Yemini's pestering helped propel this dissertation towards completion. I appreciate her decision to have IBM pay my wages during the final stages of this work.

My physicians at the UCLA IBD clinic, especially Dr. Stephan Targan, have played a major role in helping me improve my health so I could finish this work. Dr. Peter Wolfe's advise has also been helpful.

Several friends and colleagues, including David Bacon, Peter Reiher, Kong Li and Sung Hyun Cho have read parts of this dissertation and provided helpful comments. Thomas Marlowe read text and provided comments more quickly than a faculty member is allowed by law.

I thank Jaime Moreno, Frank Schaffa and Miquel Huguet for responding to my impatient questions about LaTeX typesetting.

Rob Strom and David Bacon provided helpful comments in discussions about the dependency-tracked replication in Chapter 8.

Richard and Irina Sher have generously allowed me to camp on their sofa while I presented my final oral and filed this dissertation. I wish them many healthy and happy children. Thank you, Malcolm Gordon, for the sleeping bag.

David Rapkin deserves profound thanks for helping me through the dark times, and endlessly reminding me that there was light to be found if I looked hard enough.

Verra Morgan brought the UCLA bureaucracy to its knees so that I could graduate. She's been a source of administrative help, wise advice and enriching friendship.

My family and friends have always stood by my side.

I thank the Torfs. Andy and Mike would have been thrilled that I'm finally no longer a student. Ady's excitement at my accomplishments always enhances my pleasure in them. I thank Lois for having so much faith that I would finish my degree that she presented me a graduation gift in advance.

I thank my brother Bennett and sisters Anne and Marcia for their support. Anne, my California sibling, has often provided shelter from the storm.

For over 3 decades my parents have encouraged me to do what I want, but do it as well as I can. Without their loving encouragement I wouldn't be finishing my degree today.

And I thank Meira for loving me, and being uncontrollably thrilled at my achievement. *Tu eres mi alma*, too.

VITA

August 17, 1955	Born, Boston, Massachusetts
1977	A.B. Harvard College, Magna cum laude, Astrophysics
1984	MS, UCLA, Computer Science
1986 - 1989	IBM Graduate Student Fellowship
1989 - 1991	RSM, IBM Watson Research Lab

PUBLICATIONS

Arthur P. Goldberg. An Object-Oriented Simulation of Pool Ball Motion. UCLA Masters Thesis, 1984.

- -, Ajei Gopal, Kong Li, Robert E. Strom, and David F. Bacon. Transparent recovery of mach applications. In *First USENIX Mach Workshop*, Burlington, VT, October 1990.

- -, Ajei Gopal, Andy Lowry, and Rob Strom. Restoring consistent global states of distributed computations. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, Santa Cruz, CA, May 1991.

- - and David Jefferson. Transparent process cloning: A tool for load management of distributed systems. In *Proceedings of 1987 International Conference on Parallel Processing*, pages 728 - - 734, August 1987.

- -, Steve Lavenberg, and Gerald J. Popek. A Verified Distributed System Performance Model of Locus. In *Proceedings of the 1983 Sigmetrics Conference of Evaluation of Computer System Performance*, Seattle Wash., 1983.

Andy Lowry, James R. Russell, and Arthur P. Goldberg. Optimistic failure recovery for very large networks. In *Proceedings of the Symposium on Reliable Distributed Systems*, Pisa, Italy, September 1991.

Robert E. Strom, David F. Bacon, Arthur P. Goldberg, Andy Lowry, Daniel Yellin, and Shaula Alexander Yemini. *Hermes: A Language for Distributed Computing.* Prentice Hall, January 1991.

ABSTRACT OF THE DISSERTATION

Optimistic Algorithms for Distributed Transparent Process Replication

by

Arthur Paul Goldberg

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1991

Professor David R. Jefferson, Chair

Process replication is an operating system function that can represent any sequential process in a distributed application as a set of concurrently executing instances, called replicas. The purpose of process replication is to speed up the execution of distributed applications. Suppose an application contains a *bottleneck* process—a process whose speedup would speedup the entire application. If most messages executed by a bottleneck process do not modify its local state, then executing multiple replicas of the bottleneck can speedup an application's execution in two ways. First, communication delays can be reduced by locating the replicas near processes communicating with the bottleneck. Second, parallelism can be increased by executing the replicas concurrently on multiple processors.

To be convenient process replication must be *transparent*—execution of a message that modifies a replicated process's state must *appear* to the application to simultaneously modify the states of all the process's replicas. Implementing replication is challenging because it is difficult to achieve both transparency and good performance.

We present several *optimistic* algorithms for transparent process replication. The algorithms are optimistic in that they "guess" that a message whose execution modifies a replica's state can be executed before applying the modification to the process' other replicas, without having the application observe the delayed consistency. If the guess is wrong then execution of the message may have to be undone. However, if the probability the guess is correct is sufficiently high then the advantages of executing parts of a computation earlier will outweigh the cost of support for undo plus the cost of undoing incorrect executions. We present two optimistic replication algorithms. The first modifies the Time Warp optimistic distributed simulation system into a replication mechanism. The second implements a dependency-tracked mechanism based on the Optimistic Recovery fault-tolerance algorithm. We present designs for these algorithms, and discussions of their performance.

CHAPTER 1

Introduction

Communications networks connecting distributed systems, such as Ethernet, AppleTalk, DECnet, NetLan, SNA, IBM's token ring, the internet and NSFNet, have become widely used in the last decade. Distributed applications, such as network routing protocols, electronic mail, banking systems, network management tools, electronic bulletin boards, distributed simulation, software development environments, and health-care systems, have taken advantage of the increased communication.

To provide good service to interactive users these applications must perform well. One characteristic shared by these applications is frequent and distributed use of important data and services, such as routing tables in routing protocols, distribution lists in electronic mail systems, environmental state in distributed simulations, and documentation and programs in software development environments. To improve performance these applications often *replicate* the files storing important data, distributing copies of the files to many of the computers running parts of the application. For example, routing tables, documentation and programs are usually widely replicated. Without replication, access to the files could become a *bottleneck* and slow down the computation. The frequency of access would overburden the computer storing the file, and the widely distributed sources of the accesses would cause excessive communication delays.

In this dissertation we extend this concept of replication in several directions:

- We replicate *processes*, rather than files, so that distributed applications composed of communicating processes can obtain the performance benefits of replication.

- We embed the functionality of replication in the distributed operating system, so that it need not be reimplemented in each application.
- We design algorithms that manage a replicated process so that the application cannot detect the replication and can be written as if the process were not replicated.

1.1 Purpose and goals of process replication

We assume that the operating system implements an interface which executes application programs composed of sets of communicating processes. From the programmer's point of view each process consists of a set of variables accessible only to the process (the process's *state*) and a sequential *program* which describes the process's behavior. Processes communicate with one another and external systems only by sending and receiving messages. The operating system reliably transmits a message from the process sending it to the receiving process designated by the sender. See [SY83, PS83, BS89, Hoa81, Par72] for a more thorough discussion of these ideas.

This dissertation proposes that process replication be used to improve the performance of distributed application programs written in this process-model abstraction. We assume these applications are long-lived programs composed of possibly large numbers of communicating processes. These application programs run on distributed computer systems composed of a potentially large number of processors interconnected by communications networks.

This dissertation presents algorithms for transparent process replication, a new technique for speeding up the execution of distributed application programs. A distributed multi-processor operating system can *replicate* the bottleneck processes of a multi-process application, and position the replicas on processors to reduce the application program's communication delays and increase its parallelism. The replication algorithms we describe are *transparent* to any application whose processes are replicated; the operating system makes the collective behavior of the replicas of a replicated process logically identical to the behavior of a single instance of the process, so that no executing process or replica can detect the replication of any process.

Replicating a bottleneck process can improve the performance of a distributed application if a sufficient fraction of messages received by the process do not modify its state. The simplest example is a *read-only* server process, one that accepts request messages, performs some computation and/or data retrieval, and sends a reply, all with no net change in the server process's internal state.

Replicating such a process can decrease the communications distances between the process and its communicants. For example, consider an application with 4 processes and its communications patterns as shown schematically in Figure 1.1. If this application runs on a system that limits two processes to a processor then

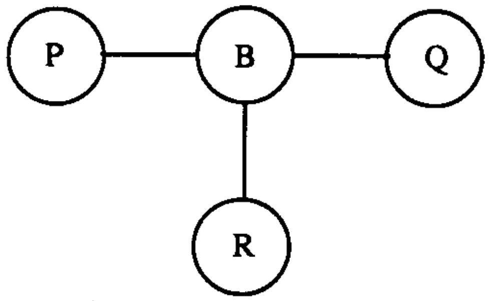

Figure 1.1: A four process application

B must communicate remotely with at least 2 processes. However, if B executes as 3 replicas, as in Figure 1.2, then it can communicate locally with each of the

B_1 Q	B_2 R	B_3 S

Figure 1.2: Replicas placed two to a processor

other three processes.

In addition, if a process like B becomes a bottleneck in a computation because it does not have enough computing resources to execute the messages sent to it then performance can be improved by representing the process by multiple replicas so that several requests may be served in parallel by replicas on separate processors. Figure 1.3 shows a tiny execution history of the application shown in Figure 1.1

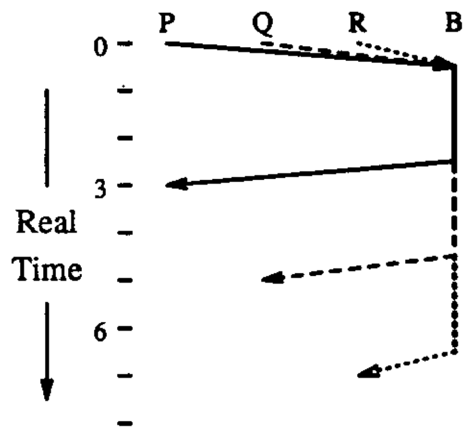

Figure 1.3: A small execution of a four process application

in which the communication delay between processes is fixed at 0.5 time units and the execution time of a message is 2.0 time units. B is a central server for the other processes, which repeatedly send it requests. (The solid line indicates a message sent from P to B; the execution of the message by B; and the reply message from B back to P. The dashed line indicates the same communication with Q, and the dotted line the communication with R.) B executes the messages from P, Q and R sequentially, so it takes 7.0 time units for the computation to complete. If B is a read-only process then a replicated execution with the same performance parameters as Figure 1.3 could complete the same execution in only 3 time units, as shown in Figure 1.4.

Not only can replicating a read-only process speed up an application; it often may pay to replicate processes that are not read-only as long as the cost of keeping the replicas consistent is offset by increased parallelism and shorter communica-

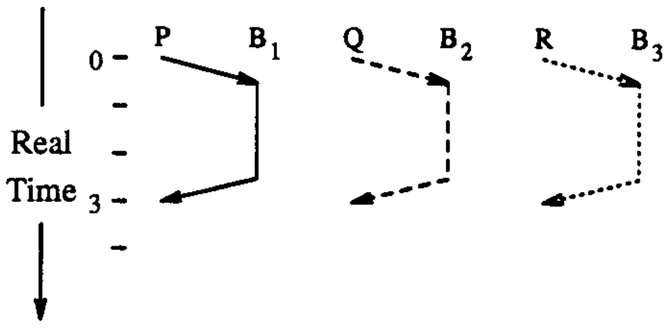

Figure 1.4: A small execution with a replicated bottleneck process

tion delays. For example, Figure 1.5 shows another replicated execution of the unreplicated program in Figure 1.3 in which the message from Q to B causes a change of state at B. Therefore the message is executed at all three replicas of B to cause the same state change everywhere. Figure 1.3 shows the message being executed last at each replica. This execution completes in 4.5 time units, which is

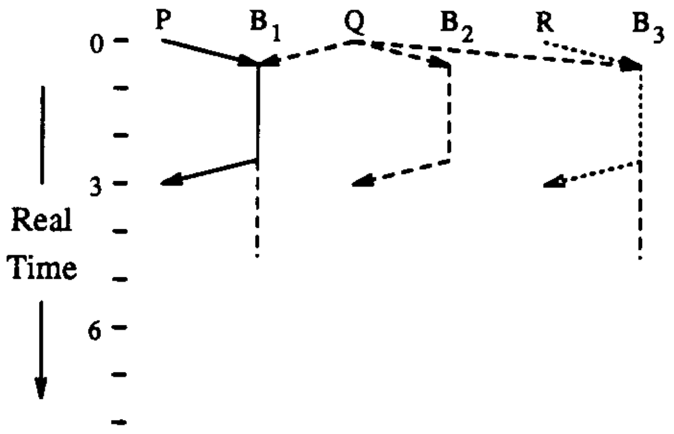

Figure 1.5: A small execution with a write message

still faster than the 7.0 time units execution of the unreplicated computation.

The fundamental challenge to implementing process replication in an operating system is controlling the replicas so that the application program cannot observe any replication. In effect, the operating system manages replicas to create the *illusion* that all replicas of a process have the *same state at all times*, so communication with one replica of a process is equivalent to communication with any other. Since replicas execute on different processors, the operating system must create this illusion without actually requiring synchronous replica execution.

The replication mechanism must classify messages into two kinds—READ *messages* and WRITE *messages*. An input message received by a replica of a process is a READ message if executing the message does not cause a net change in the replica's state; otherwise it is a WRITE message. An input message can be a READ message even if its receiver sends messages in response, and even if the receiver's state temporarily changes—as long as any changes are reversed before the next message arrives. Note that being a READ message is not a property of the message alone, but of its interaction with its receiver. Whether an input message is a READ message depends on the state of the receiver when the message arrives; the same message received by the same process in a different state may not be a READ message.

Unless messages are labeled as READ or WRITE messages by the application, it is not generally possible to determine whether a message is a READ message until after it has been executed; our replication mechanism classifies messages by observing their executions (see Chapter 9).

Severe bottleneck processes will be represented by more replicas than other processes, while processes that are not bottlenecks will not be replicated. The *load management policy* system component decides the number of replicas of a process and the placement of those replicas on processors. In this work we concentrate on the algorithms for a process replication system; we ignore the policy issues about when and where to place replicas.

1.2 Optimism

A new style of distributed algorithm has emerged recently [Jef85, SY85, SY87]. *Optimistic* protocols are a general approach to speeding up distributed algorithms. An opportunity for optimistic execution arises whenever an algorithm must decide how to proceed, but the information on which to base the decision is not locally available. The usual—*pessimistic*—approach is to block, waiting until the information is collected. The optimistic approach *guesses* the information and proceeds with the computation.

For example, consider a login program that verifies a password and then starts program X. A pessimistic implementation will wait until the password is verified before starting X. An optimistic implementation would guess that the password will be correct and start X in parallel with verification of the password.

If the guess turns out to have been correct then the program will have executed more quickly. If the guess was incorrect then the computation that depended on the guess will have to be *undone*. Optimistic algorithms must provide general undo and commit support so that appropriate action can be taken when the correctness of a guess is determined. An optimistic algorithm can run faster than a pessimistic algorithm that solves the same problem if guesses are correct with high enough probability that the advantages of executing parts of a computation earlier outweigh the cost of support for undo plus the cost of undoing incorrect executions.

We present optimistic protocols for process replication. Optimism is a good strategy to use for process replication because a guess can be made which will often be correct and will avoid considerable blocking. Consider a WRITE message m that arrives at a replica of process P. A pessimistic replication algorithm must obtain a consensus among all the replicas of P before executing m. Our optimistic algorithms guess that the consensus can be obtained, and execute m before communicating with the other replicas.

Two styles of optimistic computations have been extensively explored. Time Warp [JBW+87] optimistically executes *Virtual Time* [Jef85] programs like distributed discrete event simulations. In the Virtual Time model an application indicates the relative order in which a message should be executed by attaching an *event time* timestamp to each message. The semantics of Virtual Time executes the messages in a computation in the order of their event times. Time Warp executed Virtual Time programs optimistically, achieving Virtual Time semantics without actually executing messages in event time order. Time Warp implementations have demonstrated impressive speedups relative to sequential simulation [WHF+89, Fuj88a] through the parallel execution of Virtual Time programs.

The family of optimistic algorithms presented by Strom and his co-workers [SY85, YSB87, SY87, ABG+91] track a partial order in the execution history of a computation. The partial order is determined by the computation's communication pattern. Consider events A and B in the computation. Event A follows event B in the partial order if and only if data created at A can be communicated to B by being stored in processes' states and transmitted in messages. This is called *dependency tracking.* These algorithms have been applied to the problems of fault-tolerance in network based distributed systems [SY85, SBY87, Bac90, LRG91, GGL+90, SYB88], automatic parallelization of sequential programs [BS91], and design of distributed recovery and concurrency control protocols [YSB87, SY87].

We present optimistic process replication algorithms in both the Virtual Time model and the dependency tracking model.

1.3 Objectives of this dissertation

The primary objective of this dissertation is explore optimistic process replication. In pursuing this objective we achieve the following accomplishments.

- We extend the common notion of *file* replication to the new and more general concept of transparent *process* replication.

- We enumerate the ways a process replication system could erroneously allow an application to detect that its processes were replicated.
- We introduce use of optimistic protocols in the management of replicated processes.
- We design and present two novel *optimistic* replication algorithms, one based on Time Warp, and the other based on the dependency-tracked style of optimism. We compare the expected performance of these two optimistic replication algorithms.
- We present mechanisms that automatically classify messages as READ *messages* and WRITE *messages* and incorporate these mechanisms into the replication algorithms.

1.4 Differences from related work

There has been considerable work on replication in distributed systems, most of which addresses the problem of replicating data, such as records or relations in a database, or files in a file system [Her86, JB86, BDS84, DS83, WB84, All83, GM79, FM82, Lei84]. The major distinguishing features of our work are that:

- we advocate replicating executing processes, not just data or abstract data types [Her86];
- we recommend replication for load management, instead of fault-tolerance [PGPH90, GP91, PGPH91];
- we present mechanisms for maintaining replica consistency even though a message cannot be identified as a READ or a WRITE message until after it is executed; and
- we analyze some of the performance improvements possible with replication, and present the conditions under which to expect them.

There are several issues related to process replication that we do not discuss in this dissertation. First, we do not describe any load management policies for deciding which processes to replicate, when to increase or decrease the number of replicas, or where to place the replicas. Second, we discuss mechanisms for maintaining the consistency of processes already replicated, but do not discuss mechanisms for creating or destroying replicas. Finally, we hide certain low-level protocol issues such as flow control. These are all important issues that must be dealt with before replication can become a practical and routine tool for load management.

CHAPTER 2

Related Work on Replication

2.1 Introduction

This chapter reviews related work on replication. This work falls into two categories: replication to improve performance, and replication for fault-tolerance. While our work falls into the first category, we also review the second because it is so closely related and such an active field of research.

It is fundamentally important that while replication for performance improvement and replication for fault-tolerance may be achieved by logically similar software architectures, they tend to use significantly different algorithms. This is because performance improvement algorithm design seeks to reduce the length of communications channels and the number of messages, whereas fault-tolerance algorithm design forces an increase the number of messages and the distances they travel in order to achieve redundancy and increase the likelihood that the failures of replicas are independent.

2.2 Replication to improve performance

Chu [Chu69] was apparently the first researcher to observe that replicating a file could improve a distributed system's performance. He introduced the following optimization problem. Suppose multiple computers in a distributed system access a file. Which computers should store copies of the file to minimize the total cost of using it? Chu modelled the cost of using a file as a sum of file storage and transmission costs. The model constrains a solution to limit the mean delay to access each file at each processor and limit the file storage used at each processor.

The transmission costs includes the expense of sending data to all copies of a file for an update.

The model has several drawbacks. The cost for synchronization of concurrent updates is ignored. The model does not vary the number of copies of a file—it fixes the number and then optimizes their locations. And the expression for the constraints is extremely complex. Chu formulated the optimization problem as a 0-1 linear programming problem (by using an innovative method that transforms a polynomial problem to a linear problem) and presented a sample solution.

Chu's work initiated much further research. Casey [Cas72] reexamined the optimization problem and devised a simple linear cost function. The cost function sums the cost of communication for updates and reads, and the cost of file storage. Casey's model still ignores synchronization cost, but the number of copies of a file is a free variable. Like Chu, Casey ignores the design of algorithms to synchronize updates to a replicated file, and their influence on the cost function. The cost function is shown to be equivalent to a model for determining the economically optimal locations of manufacturing plants and warehouses. Methods for obtaining optimal solutions to the cost function are exponential in the number of processors in the network, so Casey just derives bounds on the optimal number of copies of the file.

In Casey's model R and U represent the number of read and update operations per unit time, respectively, emanating from each node. He shows that any allocation of r copies of the file for which $r > 1 + R/U$ is more costly than the optimal one copy assignment, independently of communications costs. He proves this by comparing the cost of the optimal one copy assignment with the cost of a multiple copy assignment and showing that when the above inequality holds the increased cost of updating multiple files exceeds the decreased cost of reading more nearby files. This implies that if each node generates at least half of its traffic in the form of updates then the optimal assignment allocates only one copy of the file.

These conclusions do not apply directly to process replication. First, the cost function ignores the increased performance parallel access to multiple file copies offers. For example, if file reads can keep several file servers fully busy, it may pay to replicate the file even if replication does not decrease communications cost. Second, since the cost function assumes a linear cost for all communications it assigns the same cost to the critical path in a computation as to updates of files that could be done in parallel using optimistic computation.

Casey presents a heuristic that can reduce the effort to find the optimal allocation of multiple copies of a file. He considers a graph, G, in which each vertex corresponds to an allocation of the copies of a file to the computers in a network. If adding or deleting one copy of the file changes one allocation into another then an edge in G connects the two corresponding vertices. He shows that all paths from the vertex which allocates zero copies of the file to the optimal allocation vertex have monotonically decreasing cost functions. His algorithm to find the optimal assignment explores only nodes on these paths.

Morgan and Levin [ML77] extended this work by more realistically modeling file access. They represent a file access as a user communication to a program followed by the program's communication to a file. The user, the program, and the file may all reside on different nodes in the network. Thus, minimizing a cost function involves finding the optimal location of both programs and file copies. Despite the revised model, Morgan uses the same methods as Casey to represent and solve the optimization problem. Beyond extending the modelling, the work offers no new insight.

Ruan and Tichy [RT87] also studied file replication. They developed a Markov chain that models the number of file transfers between sites. The model can characterize the interdependence of file accesses: the tendency of one site to be the main user of a file, and the tendency of a site to re-access a file it just used. They assume that a file access reads or updates the entire file. The performance of three file access patterns are examined. They assume some synchronization protocol will keep the files consistent. *Remote access* is their no replication baseline case—an access simply reads or updates the single file copy. In the *Pre-replication* protocol a read accesses a local file if it exists or else transfers one over the net, thereby creating a new file copy, and a write access updates all file copies. The *Demand access* protocol updates a file copy only when necessary. A read accesses the local copy if it is current, otherwise it copies the primary copy. A write updates only the primary copy.

Their cost function is equal to the total number of file transfers in a computation. Under the assumption of a fully connected network the analysis shows that *demand access* incurs the fewest file transfers.

Like earlier work, Ruan and Tichy ignore concurrency control costs, assuming that quantity of data transferred is the main performance predictor and that files are much larger than control messages. In addition, their model ignores performance timing.

Whether this assumption holds depends on the application. If response time is critical, then delay must be minimized even if bandwidth is expensive. However, if bandwidth is cheap, as we expect, then *delay* should be avoided. Delays due to synchronization algorithms and contention for file servers must then be modelled.

2.3 Replication to increase availability

Considerable work has focussed on the use of replication for fault-tolerant availability [AD76, Tho78, Gif79, PPR$^+$83, BBG83, DS83, Gun83, Her84, BDS84, Bir86, Her86, PGPH91, GL91, PGPH90, GP91]. Replication can make files more accessible when machines or communications links fail. A key issue in the design of fault-tolerant replication algorithms is whether to allow concurrent update of multiple copies when there is no communication between them, that is, when a *partition* occurs.

One approach prevents application programs from detecting that files are replicated by keeping file copies *consistent*. This approach does not allow concurrent updates on different sides of a partition. This work prefents replicas from becoming inconsistent by many different methods. Primary copy [AD76], majority consensus [Tho78], weighted voting [Gif79] (which has many variants) and quorum consensus [Her86] are all varieties of the same strategy: allow an update to occur in at most one connected subnetwork. By comparison with our work these approaches are 'pessimistic'—they do not allow a file to execute a write request until some protocol has assured that the write will not make the file inconsistent.

An example of this approach applied to processes instead of files is Cooper's work [Coo85], which proposes that a process execute as a "troupe"—a set of replicas executing on machines that have independent failure modes. Processes are assumed to communicate via remote procedure call. The operating system transforms a remote procedure call into a fault-tolerant communication between two troupes that is called a "replicated procedure call". The replicated procedure call sends a message from each member of the calling troupe to each member of the receiving troupe.

A member of the receiving troupe collects the messages from each member of the sending troupe, selects the most common message, executes it and returns a reply to each member of the sending troupe. If the messages differ too much then the receiver aborts. A member of the sending troupe collects the reply messages and returns the most common reply to the application. This algorithm does not scale with the degree of replication because the message cost is quadratic in the number of replicas. However, for fault-tolerance the cost would be acceptable.

The other approach to dealing with failures allows concurrent updates to copies of a replicated file on opposite sides of a communication partition [PPR+83, FM82, PGPH91, PGPH90, GP91]. The objective is to make makes files more available, so that if any copy can be accessed it can be written and read. The cost is that files can become inconsistent and application programs may need to reconcile the inconsistencies. This strategy is based on the belief that actual conflicting updates in a general purpose filing system are infrequent. This work differs from our work in that we do not allow the inconsistent states arising from optimistic execution to be seen by an application.

The *Ficus* file system [PGPH90, GP91] allows updates on two sides of a partition and then uses *version vectors* [PPR$^+$83] to detect inconsistent updates. Essentially, the version vector counts the updates to each copy of a file.[1] If file X is replicated twice and copies X_1 and X_2 get updated concurrently in two different partitions then when communication is re-established between the partitions the conflicting update will be detected.

However, consistency is defined only with respect to a single file—programs that interact with several files will not be notified of inconsistent updates. For example, consider two processes, P and Q, and two files, X and Y, each replicated twice. Suppose a failure separates the system into a partition with process P, file copy X_1 and file copy Y_1 and another partition with Q, X_2 and Y_2. Let process P's program be: write X; read Y. Let Q's program be: write Y; read X. When communication is re-established between the partitions no conflict is found because each file was updated in only one partition. However, a conflict may in fact exist because if the files had not been replicated the reads could not have preceded the writes at both files.

There is too much work on using replication to achieve fault-tolerance to fairly survey it here. See [Sch90], which reviews a general method for implementing a fault-tolerant service by replicating servers, and the references therein.

[1]A count is needed, rather than a modification bit. Partitions can be created and reconnected arbitrarily. Suppose there are three replicas of a file, all members of one partition. Suppose copy 1 gets partitioned from the other two, and then copies 2 and 3 get partitioned from each other. Then copy 2 gets updated. Copies 2 and 3 rejoin, and 3 is updated to the content of 2. If the version vector stored only a modification bit then 2 and 3 would have to have the bit reset to unmodified, in case they partitioned again. However, this would lose a record of their update with respect to copy 1.

CHAPTER 3

Our Model of Distributed Computation

To provide a context for the discussion of process replication we present the following model of a distributed application program. This is the model that the programmer has in mind when writing an application, and the semantics we describe will be preserved by our process replication mechanisms. Although we will present process replication mechanisms in the context of this model, they are in fact more generally applicable because this model can emulate most other distributed computing paradigms.

We assume that an application is composed of concurrently executing processes that communicate only via asynchronous messages and do not share memory. Each application process is a sequential program structured like the program in Figure 3.1. We choose this particular structure because it is pedagogically convenient for describing process replication.

Each process can access a set of state variables (`State_Vars` in the template) that exist for its entire lifetime. The values and sizes of the state variables may change, but their names remain. The body of a process is a `while`-loop. At the start of each `while`-loop iteration the process receives one message (called `Input_Message` in the template) sent by some (most likely another) process. Depending on the value of `State_Vars` and `Input_Message`, the process may then modify its state variables and/or send messages. We encapsulate the details of creating an output message in the function `Output_Message`. Since a process can send a message only after receiving one, we refer to output messages as *responses* to the input message received at the start of the outer loop. Observe that the `State_Vars` are the only variables in the program's scope when it receives a mes-

```
begin
   declare State_Vars := Initial_Values1;
   while Condition1( State_Vars ) do
      Receive( Input_Message );
      begin
         declare Temp_Vars := Initial_Values2;
         if Condition2( State_Vars, Temp_Vars, Input_Message)
            then Modify( State_Vars, Temp_Vars);
         end if
         while Condition3( State_Vars, Temp_Vars, Input_Message)
         do
            Send( Output_Message( State_Vars, Input_Message,
               Temp_Vars) );
         end while
      end
   end while
end
```

Figure 3.1: Template for an application process's code

sage. As shown, the program may declare additional temporary variables and use them in modifying the state variables or in generating responses to the input message.

A process can receive a mixture of read and write messages, but the application code need not be concerned with determining a message's classification. The process replication layer, with hardware assistance if possible, will determine during a `Receive()` call (or at the termination of a process) the classification of the message passed to the process during the *previous* `Receive()` call. This is discussed in Chapter 9.

In this model, application messages are addressed directly to application processes by name. The call `Send(Output_Message())` does not block—it returns as soon as the operating system has buffered the output message, without waiting for the message to arrive at the receiver. We assume messages are reliably delivered, but with arbitrary delay.

The application programmer should expect all messages addressed to a process to be funneled into a single input queue. When the process executes the `Receive(Input_Message)` system call, if the input queue holds any messages the operating system returns the next one; otherwise the call blocks until one arrives. Messages are not necessarily passed to a process in the same order that they were sent, even if they were sent by the same process.

A computation is initiated by the arrival of one or more messages from outside the system. Communication outside the system (input and output) passes through special input and output mechanisms provided by the process replication layer.

To simplify our presentation we assume that application processes are neither created nor destroyed dynamically, although this restriction could be lifted.

We require the object (run-time) code for each application process to be deterministic. The sequence of states it passes through and the sequence of messages it sends depend only on the process's initial state and the sequence of messages it receives. Therefore, any non-deterministic event, such as reading a real-time clock or executing a non-deterministic programming construct, must be implemented as an input message to the process. We require deterministic object code so that the replicas of a process can be kept consistent by executing the same messages, and so that optimistic algorithms can roll back a replica and replay its execution.

Although each process in this model is deterministic, the behavior of a collection of communicating processes is generally non-deterministic because messages concurrently transmitted to a process may arrive in any order. In our model, all non-determinism in a distributed program's behavior results from this cause.

This model can emulate many other distributed computing and synchronization paradigms. With suitable choices for the state variables, message formats, and conditional logic in the processes, it can express any computable, deterministic sequence of computation and communication events. For example, such a program can behave as though it communicates by asynchronous FIFO channels, or by rendezvous, or by remote procedure calls. FIFO channels between application processes can be emulated by inserting sender names and sequence numbers into each message and endowing each receiving process with enough buffer space and logic to execute the messages from each sending process in increasing sequence number order. (One paradigm that cannot be directly emulated because of its non-deterministic control structures, however, is Hoare's CSP [Hoa85], although any version of it with only deterministic control structures could be.)

In our presentation of the process replication mechanisms we will consistently use the communication terms shown in Figure 3.2. An executing process that

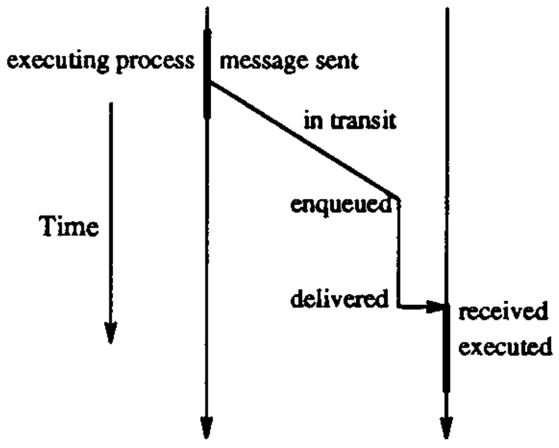

Figure 3.2: Communication terms

wants to communicate *sends* a message (in its inner loop). The message stays *in transit* for an unspecified period of time, after which it *arrives* at its destination and is immediately *enqueued* by the operating system in the receiving process's input queue. At some later time the message is *delivered* by the operating system to the receiving process, which *receives* and then *executes* it. Similar terminology will be employed when the communicants are replicas.

3.1 Assumptions about the environment

We assume that the replication layer's knowledge of the application running above it is limited to the interaction across the application program interface—sending WRITE and READ messages, and receiving messages. The replication layer has no knowledge of the application program semantics. This enables a replication mechanism to execute any message passing application which satisfies this simple model, and saves the application programmer the trouble of imparting other complicated semantic information.

We also assume that the replication mechanism, the subject of this thesis, interacts with a replication *policy* system over a well defined interface. The policy system tells the mechanism when and where to create and destroy replicas. These operations can be invoked at any time and requested of any part of the mechanism. Therefore, the mechanism must assume that many processes may be replicated.

CHAPTER 4

Architecture of Process Replication Mechanisms

4.1 System architecture

A system using process replication consists of 3 run-time software layers, the *Application program* layer, the *Replication mechanism* layer, and the *Distributed operating system* layer.

The application is a distributed program composed of communicating processes. It was written to run in a distributed computing environment with the properties and application programming interface described in Chapter 3.

The replication mechanism is one part of a complete load management system. The system consists of a load management policy component which collects performance measurement data and makes decisions about the *number* of replicas of each process and the *placement* of replicas on processors.

This chapter sketches the architecture of the design of the replication mechanism layer. An application process is represented by a non-empty set of executing replicas. (A singly-replicated process is simply a boundary case.) The replicas of a process P replicated k times are identified P_1, P_2, ..., P_k. In our later chapters we present the algorithms which control the execution of running replicas, message communication between the replicas, and synchronization and consistency maintenance among replicas. Since the replicas communicate only via messages, the replication algorithms are primarily contained in the implementation of the `Send()` and `Receive()` message communication calls implemented by the replication layer. Replica dynamics—the creation and destruction of replicas—would also be implemented in this layer, but we do not discuss these issues in detail, except to make certain that the create and destroy system calls would not be excessively difficult to implement.

The bottom layer, the *Distributed operating system* layer, provides the basic message communication services used by the replication algorithms. From the point of view of the distributed operating system, each replica is a separate process. Thus, sending a message to a distributed operating system process will reliably deliver it to one executing replica. The replication layer will also use services that manage its virtual memory, timeshare a processor, etc. We assume that a function to checkpoint a process's state can be implemented in terms of the the distributed OS functionality.

Several widely used operating systems could provide these services, such as Mach [ABB+86, BBB+90, WT89], 4.3 BSD Unix [LMKQ89], and others discussed in [Bal89]. A small amount of our work would need to be incorporated in this layer—some of the message classification mechanism discussed in Chapter 9.

In a real system the replication code would be integrated into the distributed operating system, primarily to realize performance optimizations the integration would allow. We discuss the replication code separately so the algorithms can be presented in isolation.

4.2 The basic structure of the replication mechanism

The replication layer *represents* an application process by a non-empty set of replicas. The replicas of a process can execute on any processor in the system. All replicas of an application process execute the same code. The replication code implements `Send()` and `Receive()` system calls, and controls the scheduling of the multiple replicas of a process so that they cooperate to mimic the behavior of one un-replicated process.

4.3 READ and WRITE messages

We assume that after a replica receives a message M it executes the message for a while and then receives another message or terminates. If the value of the replica's state, including all variables it can access, is the same after executing M as it was before executing M, then we say that M is a READ message. If the value of the replica's state changed, then we say that M is a WRITE message.

4.3.1 Distinguishing READ messages from WRITE messages

To simplify presentation of the replication algorithms we assume, for much of the dissertation, that the application labels a message as a WRITE or a READ. Unfortunately, obtaining this information from the application may require the application programmer to modify the program to make it able to run on top of process replication. Another drawback of having the application label messages as WRITE or READ is that the label must be pessimistic since the application cannot anticipate the receiver's state at the time the message will be executed. In addition, changing a receiving process's code may require changing the message label given by the sender, since whether the message is a READ or WRITE may also have changed. This lack of modularity would be a great inconvenience.

Therefore, after presenting the replication algorithms, in Chapter 9 we present designs for replication-level code that observes message execution so it can distinguish between WRITE and READ messages transparently. Following that, in Section 9.5, we show how to integrate the replication algorithms with the code that distinguishes between kinds of messages.

Just as an application process is represented by a set of replicas, a message sent by the application layer is represented by a *set* of messages between replicas. Because we are replicating processes to improve performance (assuming that fault-tolerance is provided by other software) and we assume that almost all messages received by a replicated process are READ messages, we optimize the performance of READ messages. Therefore, a READ is routed to only one replica, while a WRITE message must update all replicas.

The replication mechanism keeps the replicas of an application process consistent by using one of a class of protocols illustrated in Figure 4.1. Suppose that Q

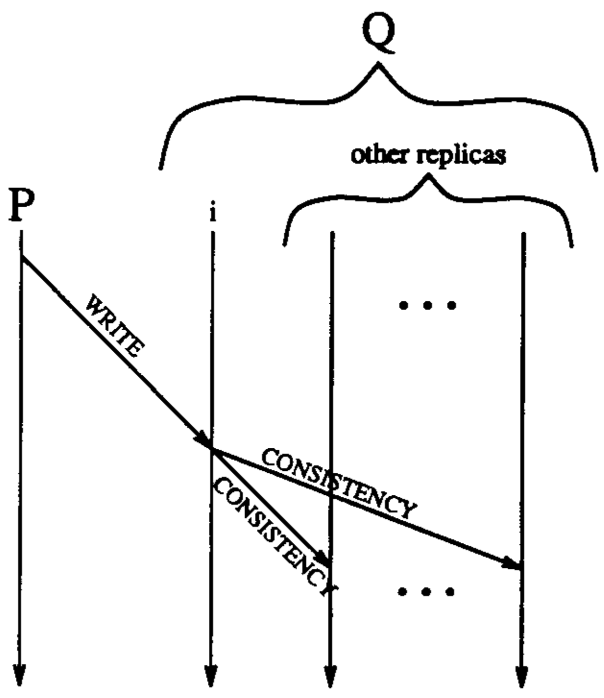

Figure 4.1: A set of messages update a replicated process

is represented by n replicas. An application's WRITE message sent to Q is represented by a collection of n messages, one called the *original* WRITE message, and $n-1$ others, called CONSISTENCY messages. The replication code routes the messages so that exactly one of the replicas, Q_i, executes the original WRITE message, and each other replica receives a CONSISTENCY message. On the other hand, a READ is represented by only the *original* READ message, executed by just one of the replicas.

Figure 4.2 shows the contents of a replication message. Application code identifies a message's `Receiving_process` and creates its `Data`. The message also contains the identity of the sending process. And, as indicated above, we assume in the first part of the dissertation that the application indicates whether the message is a READ or a WRITE message.

Field	Data Type	Used by Layers
Sending_process	process_name	Application Layer and layers below
Receiving_process	process_name	
Data	anything	
Read_or_write	boolean	Replication Mechanism Layer and layers below
Sending_replica	replica_name	
Receiving_replica	replica_name	
Original_or_consistency	boolean	
Control_information	unspecified	

Figure 4.2: Replication message fields

A replication message contains the names of the sending and receiving replicas, and a field that distinguishes original from consistency messages. The field Control_information indicates information used by the replica synchronization algorithms we present below.

When we say "a replica received a CONSISTENCY message" we mean that the replication code received a CONSISTENCY message addressed to the replica and passed its application-visible portion to the replica. Application code sees only message contents written by a sending replica and cannot distinguish between receiving a WRITE message and a CONSISTENCY message.

Whenever a replica executes a message it may generate one or more application messages in response, as indicated by the Send() call inside the **while** loop in Figure 3.1. However, the replication code transmits only output messages generated in response to an original READ or WRITE message and discards all those generated in response to a CONSISTENCY message.

An original message transmitted by the replication code is routed to one of the replicas of its receiver, preferably one nearby the sender.

4.3.2 Correctness of replication algorithms

In this section we briefly discuss the correctness of replication algorithms. Our replication algorithms make replication *transparent* to the application.

In the previous chapter we defined our programming model. An application is composed of communicating processes. A *process* consists of *state variables*, which encode its state, and a *sequential program*, which determines what the process does when it receives a message. While a process executes a message (computes from the instant it receives the message until it receives the next message) it can modify its state variables and/or send messages.

A process's program is *deterministic*—all state variables changes and messages sends during the execution of message m are completely determined by the program, the content of m, and the value of the state variables at the time the process receives m. Messages are executed by a process one at a time; that is, one message execution is *atomic* with respect to all other message executions.

In an unreplicated execution messages are processed in an order that is consistent with potential causality [Lam78]. Therefore, an application process sending a message can make the following assumption about the order in which messages are executed:

- **Causality.** If the fact that message m was sent to process P by process Q could have caused a message m' to be sent by Q' to P, then P executes m before m'.

As stated in [Sch90], a replication algorithm is correct if it ensures the following:

- **Replica Coordination.** For each process, all replicas of the process receive and execute the same sequence of messages.

This requirement can be decomposed into two separate requirements concerning the dissemination of messages to the replicas of a process:

- **Agreement.** Every replica of a process receives each message sent to the process.
- **Order.** Every replica of a process executes the messages it receives in the same relative order.

We relax Agreement in two ways. The first relaxation is possible because we identify messages that do not change a process's state. A READ message r, whose execution does not modify the state variables of the replica that receives it, need only be sent to a single replica. This is because the response from this replica will be correct, and because the state of the replica that executes r will remain identical to the states of replicas that do not.

Second, we can relax agreement because we assume the system is failure-free. Only one replica of a process need send the messages produced in response to an input message. Therefore, if executing a message w changes the state of a replica of process P, rather than send the same message w to each replica of P we send w to one replica of P and a CONSISTENCY message c to each other replica. When a replica executes c its state undergoes the same change as it would if it had executed w, but none of the messages the replica sends are transmitted.

More formal theories for constructing replication algorithms that do not satisfy Replica Coordination are proposed in Aizokowitz [Aiz89] and Mancini [MP88]. Both theories are based on proving that an ensemble of replicas implements the same specification as a single replica. Aizokowitz uses temporal logic descriptions of state sequences, whereas Mancini and Pappalardo use an algebra of action sequences.

4.3.3 Replication errors

We now exhibit three classes of replication errors, or three different ways that a replicated execution could be incorrect. Error classification gives insight into the challenges of designing replication mechanisms.

- In a *write-write* (W-W) error two replicas of a process are updated in different orders, as shown in Figure 4.3. If a W-W error occurred then the Order condition would be violated.

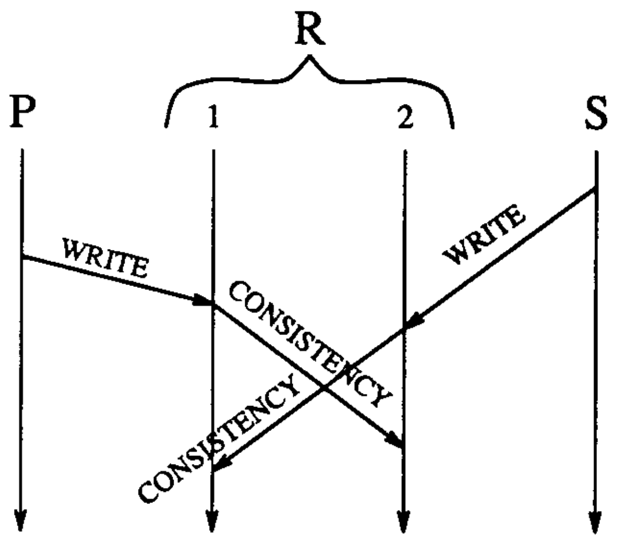

Figure 4.3: Write-write (W-W) error

- In a *read-write* (R-W) error a message that depends on an update to a one replica of a process accesses a different replica of the process before the update has been applied there, as shown in Figure 4.4. This violates Causality. In the example, the fact that the WRITE was sent to R could have caused the READ to be sent to R, which means that the READ should be executed after the CONSISTENCY message at R_1.

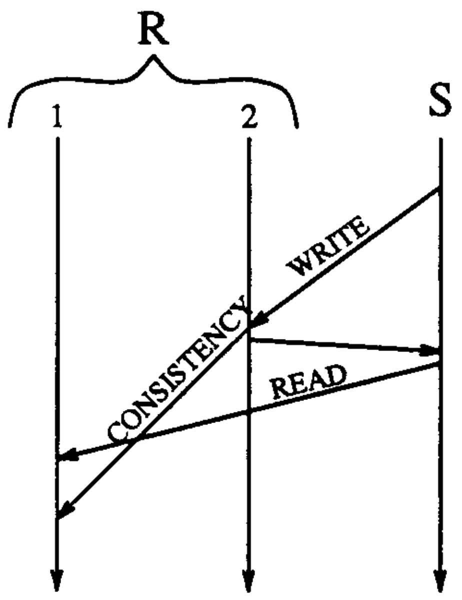

Figure 4.4: Write-read (W-W) error

- In a *circular-dependency* (C-D) error multiple replicated processes would communicate and violate Causality, as shown in Figure 4.6. Figure 4.5 shows an *unreplicated* computation involving 4 processes. Figure 4.6 shows the same computation with two processes replicated and incorrectly managed so that a C-D error occurs. Why does this execution violate Causality? Both processes Q and R execute a READ message and a WRITE message. Based on the execution of the CONSISTENCY messages, the READ is executed before the WRITE at each process. However, this is impossible because the READ message sent to R is a response to the WRITE sent to Q and the READ message sent to Q is a response to the WRITE sent to R. Thus the execution has a circular causality, as shown in Figure 4.7.

4.3.4 Replication synchronization strategies

This dissertation considers two dramatically different strategies for replication synchronization, *pessimistic* replication and *optimistic* replication. Figure 4.8

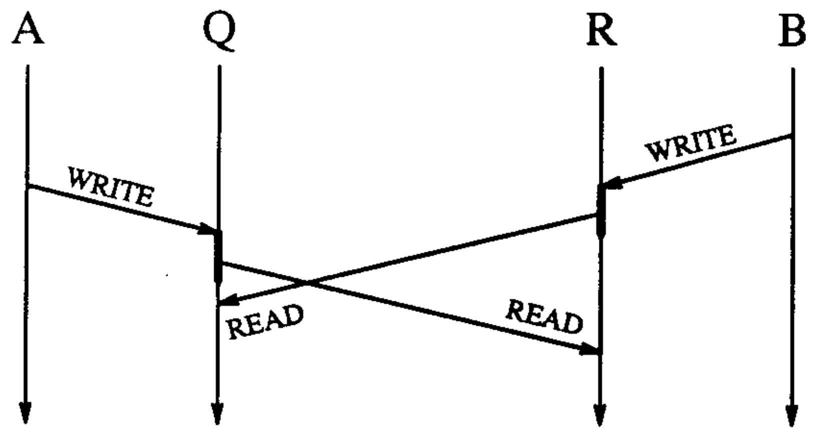

Figure 4.5: Application level of example computation with C-D error

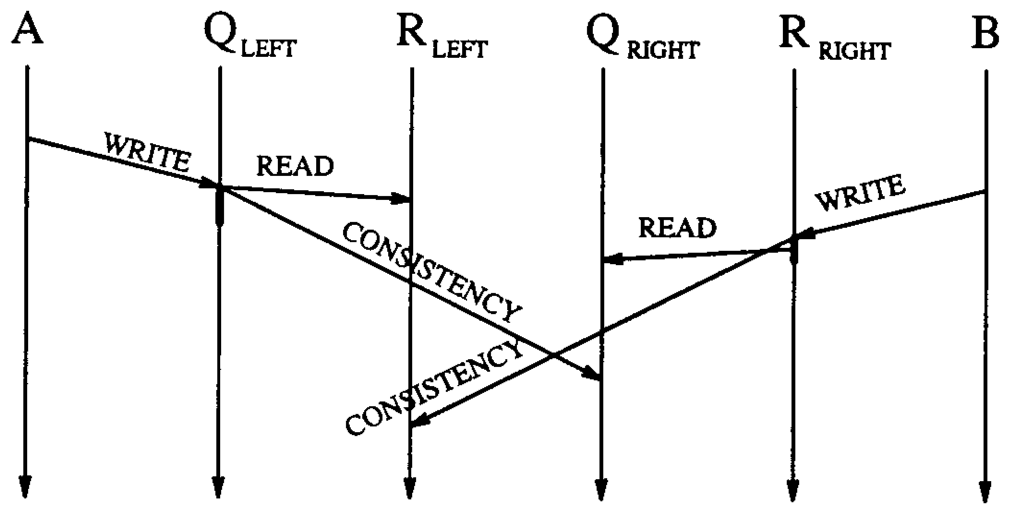

Figure 4.6: C-D error

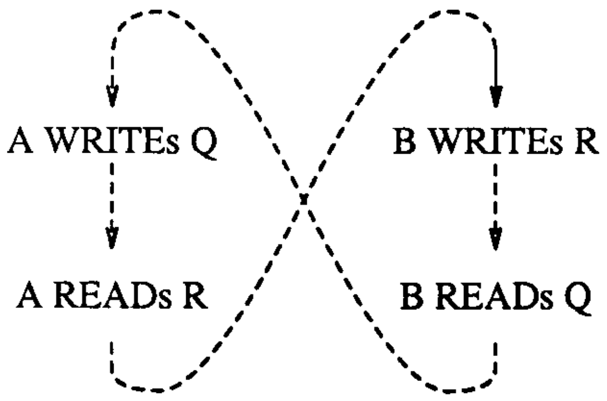

Figure 4.7: Loop in C-D error

schematically compares the message execution timing of these two strategies.

A pessimistic mechanism blocks execution of a message until it knows the execution could not cause a replication error. To establish this knowledge a distributed consensus must be reached before an arriving WRITE message can be executed. Figure 4.8 shows a pessimistic mechanism—the WRITE is executed only after completion of a distributed protocol involving the CONSISTENCY messages. We discuss a pessimistic replication mechanism to present the usual approach, and to illustrate its performance drawbacks.

An optimistic mechanism can execute messages earlier than the pessimistic mechanism. In particular, the optimistic mechanisms allows a replica to execute a WRITE *in parallel* with the distributed consensus protocol that determines whether the execution could cause a replication error, as illustrated in the bottom of Figure 4.8. The consensus protocol will eventually decide whether the WRITE execution causes a replication error. The mechanisms are 'optimistic' in that they optimistically assume that the message execution will not cause a replication error. If it does not, which for good performance must be the usual outcome, then the execution *commits*; if it does cause replication errors then optimistic mechanisms must undo, or *roll back*, the message execution. If, in the example of Figure 4.8,

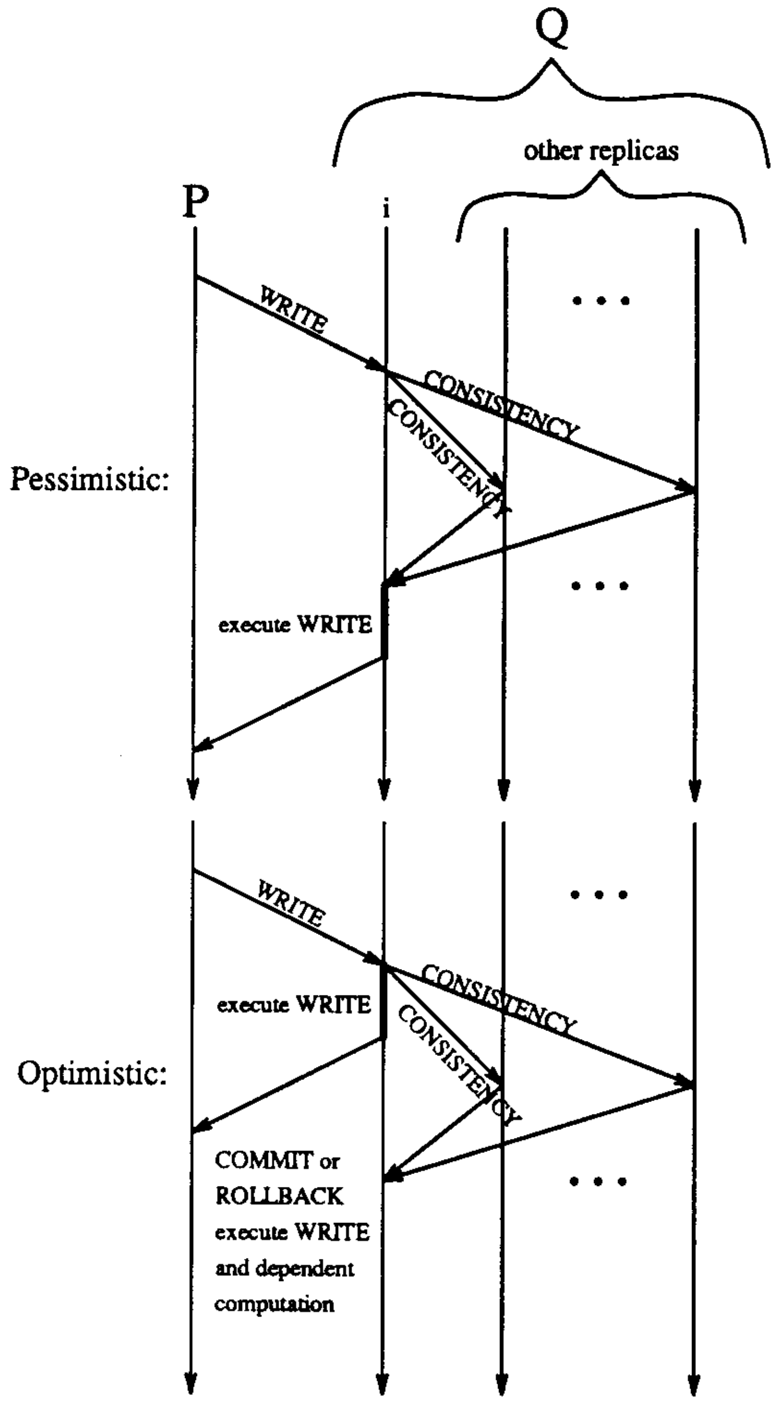

Figure 4.8: Message execution timings of optimistic and pessimistic mechanisms

the consensus protocol determines that the execution does not cause an error then the WRITE execution and the message sent from Q_i to P commit; otherwise the execution and the transmission must be undone.

We have designed two different optimistic replication mechanisms. The first (described in Chapter 7) use Time Warp [Jef85, JBW+87], a mechanism originally intended to execute distributed discrete-event simulation (DES) programs. This replication mechanism transforms an application program of replicated processes into a DES and then runs the program on Time Warp.

The second optimistic approach (described in Chapter 8) uses techniques similar to the algorithms in [SY85, BS91, YSB87, SY87, LRG91, GGL+90]. It looks for replication errors by tracking dependencies on updates to processes. A potential replication error is detected by observing a loop in the dependency graph. Some replication errors can be averted; others must be undone by rolling back the replicas involved.

4.4 Process to replica mapping

A *replica map* in the replication layer contains the mappings between application processes and executing replicas. The replica map provides several functions which map an application process identifier into the identity of one or more replicas of the process:

- The *best replica* function (`replica_map( 'best_replica', receiver )`) identifies the replica of the process `receiver` which the load management software predicts will most quickly execute a message. Note that the best replica is not necessarily the closest replica. Instead, it may be a nearby replica running on the least-loaded processor, or running on the physically fastest processor, or some combination thereof.

  Load management policy algorithms control a process's degree of replication and the locations of its replicas. These algorithms change the contents of the replica maps as they adjust the configuration of replicated processes to changing performance conditions. To give the load management software as much freedom as possible we assume that the best replica of a process will vary arbitrarily from processor to processor at a given time, and from time to time on a given processor. The difficult problem of dynamically determining the best replica is beyond the scope of this thesis.

A READ message is always routed to the best replica of its receiver.

- The *all replicas* function (`replica_map( 'all_replicas', receiver )`) returns a list of all replicas of process `receiver`. It is used to find replicas other than the best replica.

Some of our algorithms distinguish between a process's 'primary' replica and its 'secondary' replicas.

- The *primary replica* function, (`replica_map( 'primary_replica', receiver )`), maps an application process name into the process's unique primary replica. In some algorithms, WRITE messages are directly routed to the primary replica to co-ordinate their synchronization.

We refer to the replica map and use the `replica_map( )` functions elsewhere. The function call data type `set_of_replica_map_functions` will refer to these replica map functions.

CHAPTER 5

Pessimistic Process Replication

We introduce process replication by first presenting a distributed pessimistic mechanism. This pessimistic mechanism illustrates the fundamental architecture of a system whose performance will be improved by being implemented optimistically in the following two chapters.

Figure 5.1 shows the architecture of the process replication system. The three

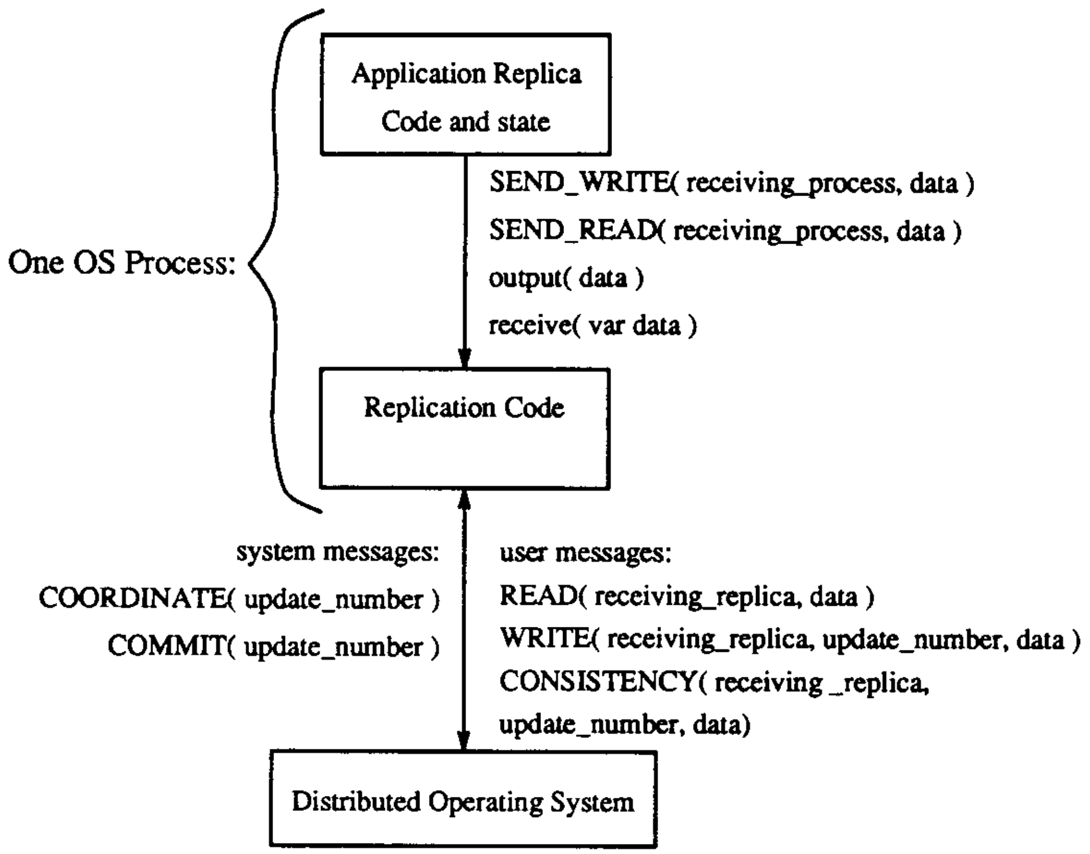
One OS Process:
Application Replica Code and state
SEND_WRITE(receiving_process, data)
SEND_READ(receiving_process, data)
output(data)
receive(var data)
Replication Code
system messages:
COORDINATE(update_number)
COMMIT(update_number)
user messages:
READ(receiving_replica, data)
WRITE(receiving_replica, update_number, data)
CONSISTENCY(receiving _replica, update_number, data)
Distributed Operating System

Figure 5.1: Pessimistic process replication architecture

boxes represent the three software layers enumerated at the beginning of Chapter 4. The application code in the *application program* layer makes the four calls indicated. These calls are supported by the *replication mechanism* layer, which implements the calls made by the application code. We assume that these two layers execute *on top of* some standard distributed operating system. Most likely the application process code is linked to the replication mechanism code, and together they run as a single OS process.

We assume that the application does not circumvent the replication mechanism by communicating directly with the distributed OS, nor does it modify variables used by the replication code.

As Figure 5.1 shows, we assume an application makes the following calls on the replication code:

- *SEND_WRITE(receiving_process, data)* sends a WRITE message containing *data* to the application process *receiving_process*.
- *SEND_READ(receiving_process, data)* sends a READ message containing *data* to the application process *receiving_process*.
- *output(data)* outputs *data*.[1]
- *receive(* var *data)* which receives the content of the next input message into the variable *data*.

The replication code implements the application's communications calls. To implement the replication algorithm one instance of the replication code communicates with other instances of the replication code via the messages indicated in Figure 5.1. The messages are categorized into two kinds, *user* messages (shown on the right side of the Figure), which contain data for replicas to receive, and *system* messages (shown on the left side), which contain control information for

[1]Input to a replicated process must be disseminated to the replicas. This can be handled by an unreplicated process that reads the input and sends it in a message to the replicated process.

the replication mechanism. WRITE and CONSISTENCY messages are collectively called UPDATE messages because they update the state variables of the replica that executes them.

When a replica sends a message it is routed to the 'best' replica of its receiving process. As stated in Section 4.4 the identify of the 'best' replica of a process varies with processor and time as the system's performance changes. Therefore, a message can be routed to any replica of its receiving process. The message will be executed by the replica to which it was routed. If the message is a WRITE message then the replication mechanism will create and transmit CONSISTENCY message copies of the WRITE to each other replica of its receiving process. A CONSISTENCY message is identical to the WRITE message, except for the field identifying it as a CONSISTENCY message.

Messages are buffered in an input message queue (IMQ) at each replica. Messages are executed by the replica in the order they are dequeued from the front of the IMQ. A message's execution must be delayed until it can be certain that the correctness requirements given in Section 4.3.2 will be satisfied.

The Order requirement can be satisfied by assigning unique numerical identifiers to the messages sent to a process and having all replicas of the process execute the messages according to the order of the identifiers.

A message m is defined to be *committed*[2] at a replica r once no message with an identifier less than or equal to the identifier on m can be subsequently delivered to r. Then, the Order requirement is implemented by having a replica (1) execute only committed messages and (2) always execute the committed message with the smallest unique identifier.

A WRITE message is stamped with a unique numerical identifier called an *update number* (un) by the replication code running its receiving replica. The CONSISTENCY message copies of a WRITE have the same un as the WRITE. The

[2]Schneider [Sch90] uses the term *stable* to describe the same situation. We use the term *commit* because it is more widely used in optimistic computation, as in [SY85, JBW+87].

UPDATE messages received by a replica are executed in *un* order. The *un* values created by the replication code running the replicas of one process are all distinct.[3] A message M with *un* u is denoted M(u), as in WRITE(u), CONSISTENCY(u), UPDATE(u), etc.

The replication mechanism maintains the following invariant.

- **Increasing *un* values (IUV).** After the replication code receives or stamps a message with *un* value u, the code will only stamp messages with *un* values x such that $x > u$.

For this algorithm, we assume that the messages sent by the replication code running one replica to the replication code running another replica arrive in the order they are sent. If If this functionality is not provided by the distributed OS then it can be easily implemented by assigning sequence numbers to the messages sent over the channel and delivering messages in sequence number order.

The key issue in this algorithm is determining when a message becomes committed. When an UPDATE(u) message arrives it is *uncommitted*, because another replica of the process might send a CONSISTENCY(x) message with $x < u$. By invariant IUV an UPDATE(u) message enqueued in the IMQ at R_i becomes committed when all other replicas of R have received an UPDATE(u) message.

Every WRITE message receipt initiates a three-phase protocol we call the *update* protocol, shown in schematically Figure 5.2.[4] The replica that will execute the WRITE is called the protocol *controller*. After assigning *un* u to WRITE(u) the replication code at the controller does the following:

1. create and transmit a CONSISTENCY(u) messages to each other replica of the process,

2. receive a COORDINATE(u) message from each other replica, and

[3]There is no relation between the *un* values of messages received at different processes. The *un* values are sequence numbers for the receiving process and *not* globally comparable timestamps.

[4]This protocol, and the rest of the pessimistic replication algorithm, depends on the assumption that the number of replicas are static and known to the controlling replica.

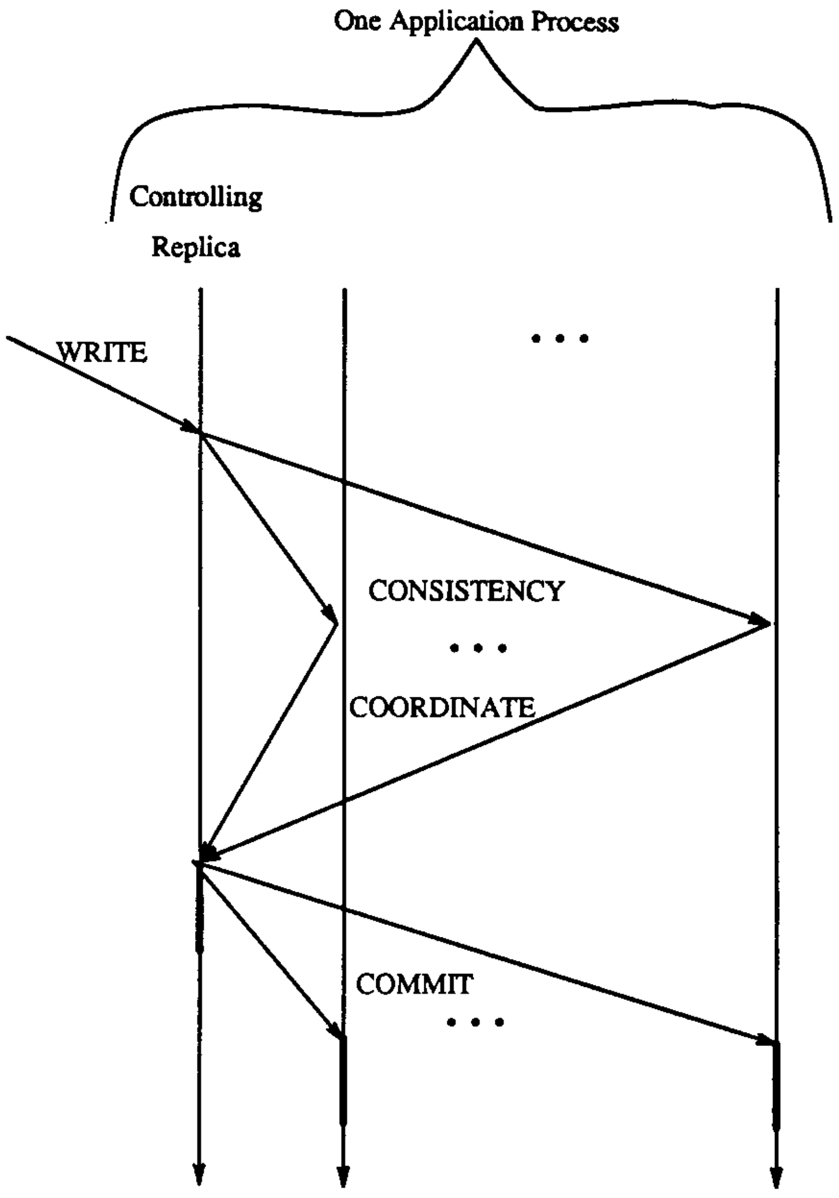

Figure 5.2: Update protocol

3. transmit a COMMIT(u) message to each other replica.

When a CONSISTENCY(u) message arrives the replication code enqueues it in the IMQ and sends an COORDINATE(u) message back to the controller. When a COMMIT(u) message arrives the UPDATE(u) message in the IMQ is marked *committed.* An UPDATE(u) message must be in the queue because the replica has already sent a COORDINATE(u) message.

By invariant IUV and the FIFO transmission of messages, when the controller has received an COORDINATE(u) message from each other replica WRITE(u) is *committed.* Since we assume message deliveries are reliable and are no failures, this will eventually happen. All of the UPDATE(u) message can then be executed, so the controller sends a COMMIT(u) message to each replica. Figure 5.2 indicates the time at which a WRITE message or CONSISTENCY message can be executed by a thick vertical line.

A READ is not stamped with an *un*. If it were then there would need to be some coordinating communication between the replica that receives the READ and the other replicas of the process before the READ could be executed. We want to avoid this coordinating communication to optimize performance for READ execution.

The UPDATE messages in a replica's IMQ are ordered by their *un* values. When a READ arrives at a replica the message is enqueued at the end of the IMQ, so it will be executed after all messages already in the queue. This queuing discipline and the update protocol above ensure that this algorithm satisfies the Causality requirement given in Section 4.3.2, which we now show.[5]

Consider message m sent to process P. Either m updates the state of P or it does not. If m updates the state of P then m is represented by a WRITE executed at one replica of P and CONSISTENCY messages executed at all other replicas of P. These messages cannot cause some message m' to be sent to P until they are executed, but they are not executed until they are committed. By IUV no WRITE

[5]The Causality requirement says that if the fact that message m was sent to process P by process Q could have caused a message m' to be sent by Q' to P, then P executes m before m'.

message m' that arrives after m is committed can be executed before m. By the queuing disciplíne, no READ message m' that arrives after m is committed can be executed before m. Therefore, m' must be executed after m at all replicas of P.

If m does not update the state of P then m is represented by a READ executed by one replica of P. Since m is represented by only one message, any message m' caused by the execution of m must be executed at P after m.

5.1 Pseudocode implementation of pessimistic replication mechanism

We now present a pseudocode implementation of the pessimistic replication mechanism. Data types for this pessimistic replication mechanism are defined in Figure 5.3.

```
replica_name :  record (
  process_name :  charstring,
  replica_index :  integer);
message_kinds :  enumeration ( 'WRITE', 'READ', 'CONSISTENCY',
  'COORDINATE', 'COMMIT');
msg :  record (
  sender, receiver :  replica_name,
  data :  charstring,
  kind :  message_kinds,
  un :  real,
  committed :  boolean);
message_queue :  queue of msg;
uncommitted_q_entry :  record (
  msg :  msg,
  coordinates_needed :  integer,
  reads :  message_queue )
```

Figure 5.3: Data types for process replication mechanisms

We assume that a queue, such as message_queue, supports the following operations.

- enqueue(imq : message_queue, m : msg) enqueues m as the last message in imq.
- dequeue(imq : message_queue, var m : msg) is an operation that dequeues the first message in imq and stores it in m.
- empty(imq : message_queue) returns 'true' if imq is empty, 'false' otherwise.

We assume that a queue can be shared by different parts of the replication code, and that a dequeue operation on an empty queue will block. The variables and constants used by the replication code are declared in Figure 5.4. The shared variable and queues are marked as SHARED in comments following their declarations.

```
declare:
  highest_un :  real  -- SHARED
  committed_q :  message_queue -- committed message queue, SHARED
  unique_id :  real  -- unique value for this replica, in (0, 1)
  uncommitted_q :  queue of uncommitted_q_entry indexed by
     *.msg.un  -- SHARED
  this_replica :  replica_name   -- initialized constant
  num_replicas :  integer  -- initialized constant
  executing_consistency :  boolean
  replica_map :  set_of_replica_map_functions
```

Figure 5.4: Replication code state

The code splits a replica's IMQ into two parts, the committed_q and the uncommitted_q. The committed_q contains messages that can be executed immediately, whereas the uncommitted_q contains messages that cannot yet be executed.

To make the code shorter, we assume the uncommitted_q is stored in an associative queue. A **queue of** t **indexed by** *.f is a queue containing entries with data type t which is ordered and indexed by the field f of each entry.[6] The data type of f must have a total ordering.

Each field in the uncommitted_q contains one UPDATE message and a set of READ messages. uncommitted_q is ordered and indexed by the msg.un fields of its entries. For example, uncommitted_q[r].msg is the message msg in the entry indexed by r.

Given the declarations

```
q :  queue of t indexed by *.f
m :  t
```

and assuming r has the same type as m.f, we assume the indexed queue q supports the following operations:

- q[r] := m stores m into the entry indexed by r
- delete(q, r) deletes the entry indexed by r from q
- n := next_entry(q, r), as used in the processing of a COMMIT message in Figure 5.7, returns the index of the next entry in q following r in n, or infty if there are no more entries.
- s := smallest_entry(q) returns the index of the entry with the smallest index in s (or infty if there are no entries).

Code for this pessimistic replication protocol is outlined in Figures 5.5, 5.6 and 5.7. Figure 5.5 shows the code which interfaces with application replicas. The other code handles messages as they arrive at the replication code. The code in Figure 5.6 handles user messages; the code in Figure 5.7 handles system messages.

The replica interface in Figure 5.5 is simple. To shorten the code we have incorporated SEND_WRITE and SEND_READ into a single send call with the kind of the message as an argument. We assume that the operating system call transmit(

[6]This is a modification of the *table* data type supported by the *Hermes* language, as described in [SBG+91].

m) transmits message m to the replica indicated in m.receiver, and that print() produces output.

```
------------------
receive( var data )
  dequeue( committed_q, m )
  executing_consistency := (m.kind = 'CONSISTENCY')
  data := m.data
------------------
output( data )
  if (not executing_consistency) then
    print( data )
  end if
------------------
send( receiver, data, kind )
  if (not executing_consistency)
  then
    m.sender := this_replica
    m.data := data
    m.kind := kind
    m.receiver := replica_map( 'best_replica', receiver )
    transmit( m )
  end if
```

Figure 5.5: Replica interface calls

receive dequeues messages in order from the committed_q message queue. It will block if the queue is empty.

No output is produced, or messages are sent, while a replica is executing a CONSISTENCY message. send routes a message to the 'best' replica of a receiving process. (The replica_map functions are described in Section 4.4.)

Code executed on arrival of a user message m is shown in Figure 5.6. The highest un of any message ever received at a replica is stored in highest_un. When a CONSISTENCY message arrives highest_un is set to the maximum of itself and the message's un. When a WRITE message arrives it is assigned a un greater than highest_un; then highest_un is set to the un.

```
-------------------
'WRITE':
 m.un := ceiling( highest_un ) + unique_id
 highest_un := m.un
 m.committed := 'false'
 uncommitted_q[m.un].msg := m
 uncommitted_q[m.un].coordinates_needed := num_replicas - 1
 -- disseminate CONSISTENCY messages
 m.kind := 'CONSISTENCY'
 m.sender := this_replica
 for replica in
   replica_map( 'all_replicas', this_replica.process_name) do
   if not( ( replica = this_replica ) ) then
     m.receiver := replica; transmit( m )
   end if
 end for
-------------------
'CONSISTENCY':
 highest_un := max( m.un, highest_un )
 uncommitted_q[m.un].msg := m
 m.kind := 'COORDINATE'
 m.receiver := m.sender
 m.sender := this_replica; transmit( m )
-------------------
'READ':
 if empty( uncommitted_q ) then
   enqueue( committed_q, m )
 else
   enqueue( uncommitted_q[ highest_un ].reads, m )
 end if
```

Figure 5.6: Code executed on arrival of a user message m

The unique_id is a real in $(0,1)$ that is distinct for each replica of a process. When a WRITE message arrives it is assigned a un that is

- unique with respect to any un given to any WRITE by any replica of this process, and
- larger than the un of any UPDATE yet seen at this replica.

The WRITE is marked uncommitted and inserted in the uncommitted_q in a position indexed by its un. CONSISTENCY message copies of the WRITE are disseminated to all other replicas of the process.

When a CONSISTENCY message arrives it is inserted in the uncommitted_q and a COORDINATE message is sent back to the replica that received the WRITE.

When a READ message arrives, if the replica is not waiting for an UPDATE to commit then the READ message can be executed without waiting for a COMMIT message, so it is enqueued in the committed_q; otherwise it is enqueued in the uncommitted_q behind the last UPDATE.

Figure 5.7 shows the code executed on arrival of a system message m. If an arriving COORDINATE(u) message is the last COORDINATE(u) reply to CONSISTENCY(u) from another replica, then UPDATE(u) messages have committed, so a COMMIT(u) is sent to every replica.

When a COMMIT(u) message arrives UPDATE(u) is marked *committed.* Then all messages in the uncommitted_q *earlier* than the first uncommitted message are transferred from the uncommitted_q to the committed_q. Since messages are received by the replica from the committed_q in order, this ensures that (1) only committed messages are executed, and that (2) committed messages are executed in un order.

5.1.1 Liveness

This protocol is live. Assuming reliable message delivery and that the replication system works fast enough to handle incoming message traffic, each UPDATE(u)

```
-------------------
'COORDINATE':
  decrement( uncommitted_q[m.un].coordinates_needed )
  if ( uncommitted_q[m.un].coordinates_needed = 0 )
  then
    m.un := m.un
    m.sender := this_replica
    m.kind := 'COMMIT'
    for replica in
      replica_map( 'all_replicas', this_replica.process_name) do
        m.receiver := replica; transmit( m )
    end for
  end if
-------------------
'COMMIT':
  uncommitted_q[m.un].msg.committed := 'true'
  -- transfer committed messages from uncommitted_q to committed_q
  i := smallest_entry(uncommitted_q)
  while ( i <= highest_un ) do
    if ( uncommitted_q[i].msg.committed ) then
      enqueue( committed_q, uncommitted_q[i].msg)
      delete( uncommitted_q, i )
      while not(empty(uncommitted_q[i].reads)) do
        dequeue( uncommitted_q[i].reads, m )
        enqueue( committed_q, m )
      end while
      i := next_entry( uncommitted_q, i )
    else
      exit while
    end if
  end while
```

Figure 5.7: Code executed on arrival of a system message `m`

will eventually commit. Therefore all messages in IMQs will eventually be received. Thus, messages sent by an executing replica of an application process will be received and executed by a replica of its receiving process.

5.2 Performance

The performance drawback of this pessimistic mechanism is that the replicas of a process cannot receive any newly arrived messages during the update protocol. In particular, if UPDATE message m is enqueued in the `uncommitted_q` IMQ of replica R at real-time t, then a READ message that arrives at R after t cannot be executed until after m is committed and executed. This interval must be at least as long as the round trip delay to the controlling replica.

The drawback of the *pessimistic* strategy of this protocol is that if WRITE messages are rare, as we assume, then in most cases waiting for an UPDATE to commit is an unnecessary delay. Most UPDATE messages were committed before the protocol was able to determine so, and could have been executed earlier than they were.

Our *optimistic* replication algorithms, presented in the next three chapters, take advantage of this observation.

CHAPTER 6

Replicating the Processes of Virtual Time Applications

6.1 Virtual Time

Virtual Time (VT) [Jef85] is a fundamental paradigm for synchronizing the interactions of processes in a concurrent program. The processes in a VT program communicate solely by scheduling *events* for each other. A process schedules an event by sending a message which specifies the event's *receiving process* and *receive virtual time* (RVT). The RVT is an element of a totally ordered type (such as the reals) with a positive infinite value (∞).

An event is the processing of a message by its *receiving process*; i.e. all execution by the *receiving process* from receiving the message to the next receive. A VT message contains the following components:

- *sending process*
- *send virtual time* (SVT)
- *receiving process*
- *receive virtual time* (RVT)
- *event data*

All fields are written by the *sending process* and read by the *receiving process*.

A process's *local virtual time* (LVT) is defined to be either the RVT of the message the process is currently executing, or ∞ if the process is not executing a message. A message's *send virtual time* (SVT) equals the LVT of the process that sends it.

Virtual time enforces a natural notion of causality—the arrow of causality points in the direction of increasing virtual time. This is enforced by the two fundamental rules of VT semantics [Jef85]:

1. The events at a process are executed in increasing RVT order.

2. The SVT of a message must be less than its RVT.

VT is an extremely powerful synchronization model. For example, one process can make a pair of events at two different processes appear to execute atomically by scheduling the two events to happen at the same RVT.[1]

We can think of a VT program as the evaluation of a function defined on a space-time plane, where the virtual space position corresponds to process identity. (The function is defined sparsely—only at the space-time points of events.) Figure 6.1 shows one message on this plane. The input to the function is the set of processes and their initial states, and the set of input messages.

6.2 Replicating VT processes

We now show how to transform a VT application program into a VT program whose processes can be replicated. Because VT requires application processes to state the virtual time of every interaction it is easy to replicate the processes of a VT application.

To implement the transformation we place a layer of *replication code* below the application (see Figure 6.2). The replication code intercepts all application communication calls. The combined application and process replication code is still a VT program. A *Virtual Time Machine* is any system that executes VT programs, such as a sequential discrete-event simulation (DES) engine or the Time Warp [JS82] distributed simulation engine.

[1]We assume that two event messages received by a process cannot have the same RVT. Alternatively, events with the same RVT can be ordered by the remaining content of the message, as is done in the Time Warp Operating System [JBH$^+$85].

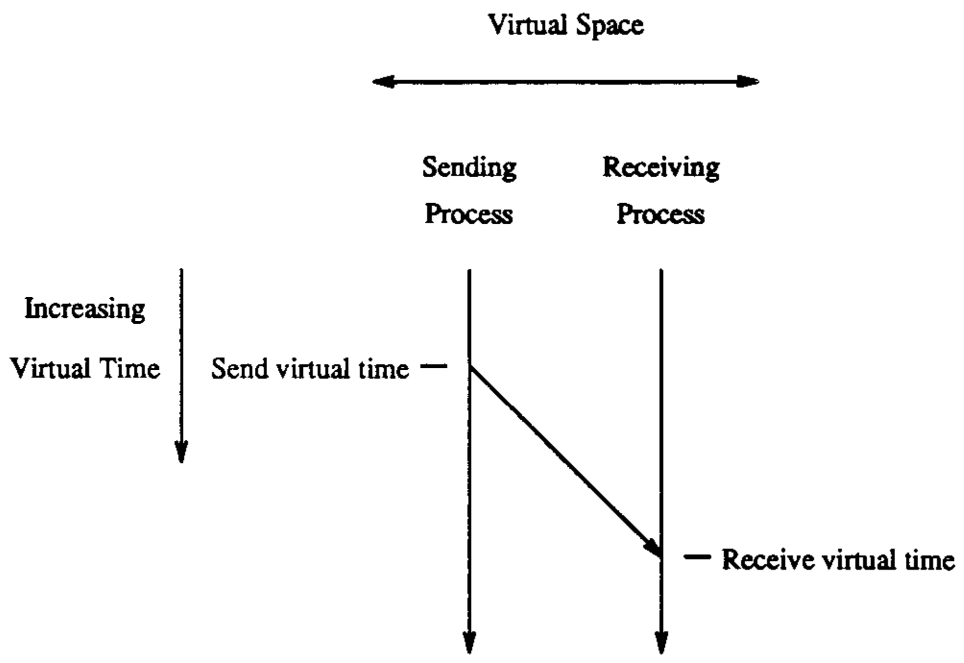

Figure 6.1: A virtual time event message

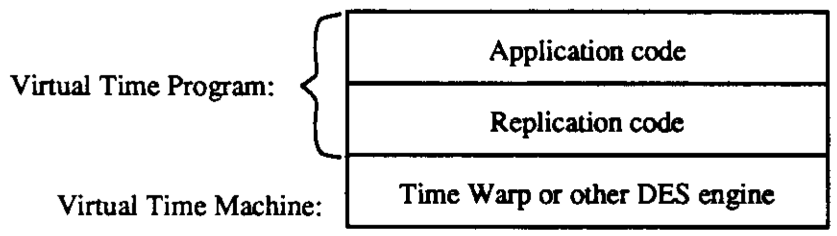

Figure 6.2: VT process replication architecture

We abstract the interface provided by a Virtual Time Machine into four essential functions:

- `vt_send( data, receiving_process, receive_time )` is a command that sends an event message which will be executed by the *receiving process* `receiving_process` at *RVT* `receive_time`. (The message's *sending process* and *SVT* are filled in by the operating system.)

- `vt_receive( data )` receives the next message into the variable `data`.

- `vt_output( data, receive_time )` outputs `data`. Output from a process is displayed in `receive_time` order; output with the same `receive_time` must be displayed in the order it was created.

- `vt_virtual_time( )` returns the process's LVT.

As stated in Chapter 4, we assume that the application identifies each message it sends as either a READ message or a WRITE message. The replication code implements almost the same set of functions as VT except that it distinguishes between WRITE and READ messages.

- `READ( data, receiving_process, receive_time )` sends a READ message to the process `receiving_process`.

- `WRITE( data, receiving_process, receive_time )` sends a WRITE message.

- `receive( data )` receives the next message.

- `output( data, receive_time )` outputs `data`.

A replicated process R is represented by a set of replicas, each one of which is a separate VT process, as illustrated in Figure 6.3. Although the application code and the replication code may occupy the same address space, we assume the application does not modify the replication code's state. We also assume that the application code does not circumvent the replication code by interacting directly with the Virtual Time Machine.

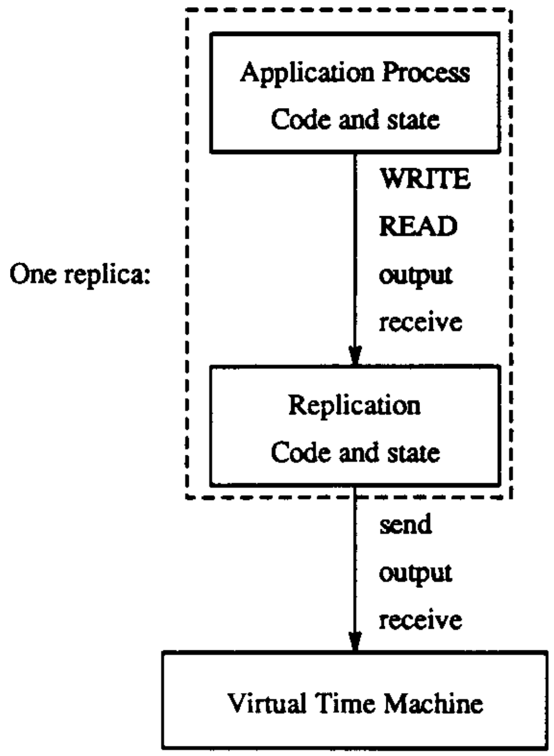

Figure 6.3: One replica

We assume that the replication code can access replica map functions (described in Section 4.4) which identify the replicas of a process. The replication code routes a READ message sent to process R to the 'best' replica of process R (Figure 6.4).

The replication code keeps the replicas of a replicated process consistent by transforming a WRITE message sent to process R into a set of messages with the same RVT—one WRITE routed to one of R's replicas and a set of CONSISTENCY messages routed to R's other replicas (Figure 6.5). Because the VT paradigm guarantees that all processes execute messages in RVT order, executing the WRITE message and the set of CONSISTENCY messages becomes an atomic update to the

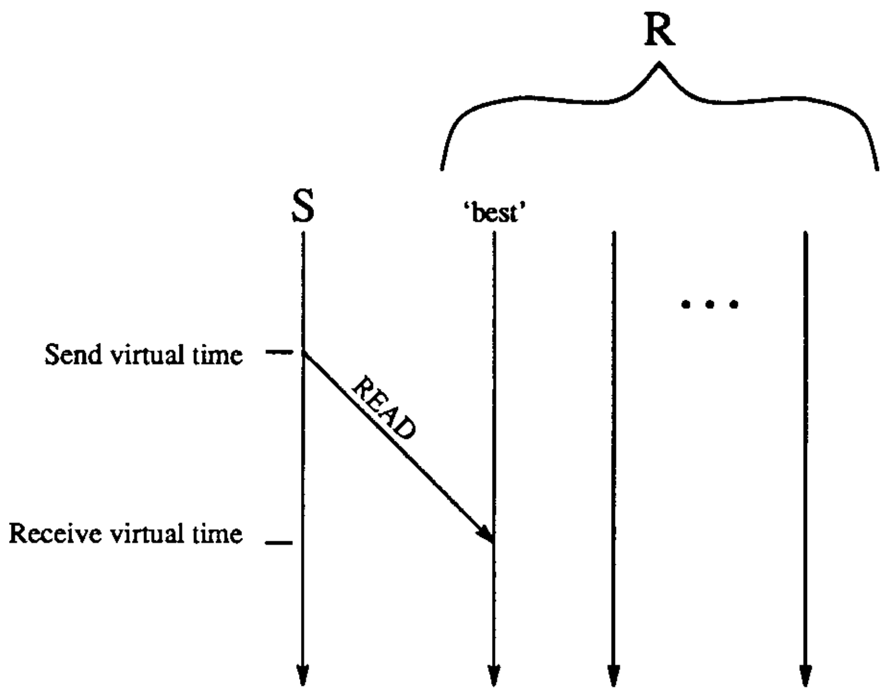

Figure 6.4: A READ message sent to a replicated VT process

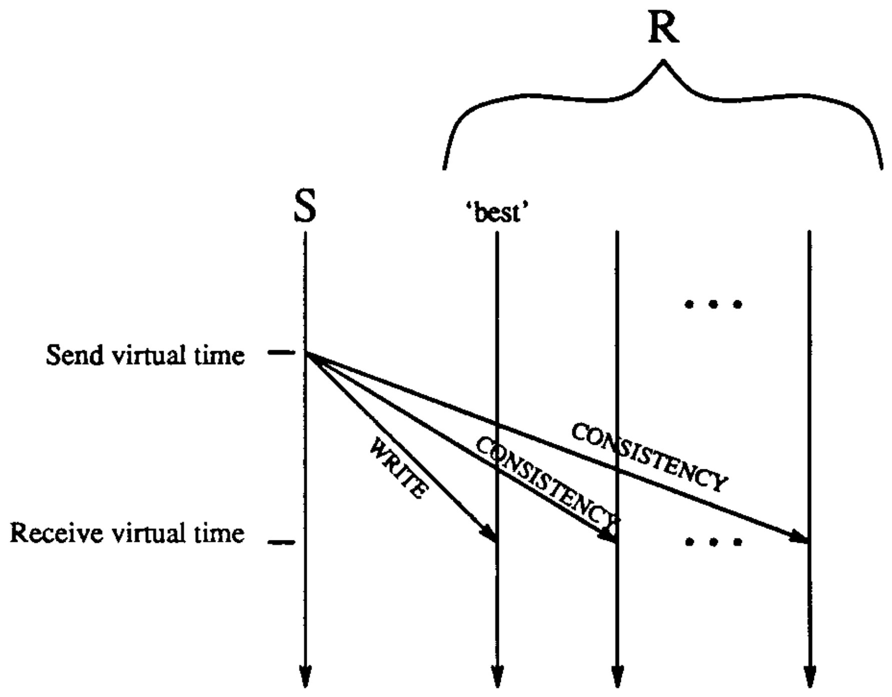

Figure 6.5: A WRITE message is sent to a replicated VT process

message's *receiving process.*

The replication code is outlined in Figure 6.6. The state has several variables. `replica_map` is a set of capabilities to the process to replica mapping functions defined in Section 4.4. `executing_consistency` is true when the replica is executing a CONSISTENCY message. `this_replica` is a constant that gives the name of the local replica. If the simulation is executing a CONSISTENCY message then sending a message or producing output is a no-op. Otherwise a READ operation sends a READ to the 'best' replica. A WRITE operation sends a WRITE message to the 'best' replica and a CONSISTENCY message to each other replica.

A `receive` operation receives a message from the VT machine, records whether the message is a CONSISTENCY and then passes the message's data to the simulation process.

6.3 Time Warp

Time Warp is an operating system that executes process-oriented discrete-event simulation programs in parallel. That is, Time Warp executes VT programs. Experimental and analytic performance analysis indicates that Time Warp is currently the best system for executing VT programs in parallel [Fuj87, Fuj88a, Fuj88c, Fuj88b, RMM88, RM88].

Time Warp's innovative feature is *optimistic*, parallel, execution of VT programs. At an instant of real time Time Warp executes processes at different virtual times on many processors. In a correct computation the messages sent to a process must appear to be executed in RVT order. Of course, only a process's currently available input messages can be executed. Time Warp *optimistically* executes the available messages in RVT order.

If the optimism proves unfounded, then a process's execution must be undone. For example, consider a process R that has already executed message M. Suppose a message N such that $N.RVT \leq M.RVT$ arrives for process R. N is called

```
declare
  executing_consistency : boolean
  replica_map : application_proc_to_replica_function
  m : application_message
  replica, this_replica : replica_identifier
------------------------
READ( data, receiver, receive_time )
  if (not executing_consistency) then
    m.data := data; m.kind := 'READ'
    vt_send( m, replica_map( 'best_replica', receiver ),
      receive_time)
  end if
------------------------
WRITE( data, receiver, receive_time )
  if (not executing_consistency) then
    m.data := data; m.kind := 'WRITE'
    vt_send( m, replica_map( 'best_replica', receiver ),
      receive_time )
    m.kind := 'CONSISTENCY'
    for replica in replica_map( 'all_replicas', receiver ) do
      if not( ( replica = this_replica ) then
        vt_send( m, replica, receive_time )
      end if
    end for
  end if
------------------------
output( data )
  if (not executing_consistency) then
    vt_output( data )
  end if
------------------------
receive( data )
  vt_receive( m )
  executing_consistency := ( m.kind = 'CONSISTENCY')
  data := m.data
```

Figure 6.6: Pseudocode for the replication layer

a *straggler* message, because it has arrived late. R's state must be restored to the content it would have had just before executing message N. In addition, all consequences of events with virtual times greater then or equal to $N.RVT$ must be undone. That is, R must be *rolled back* to before $N.RVT$.

Time Warp implements these requirements with a general distributed rollback and commitment mechanism. Time Warp's design and implementation have been fully described elsewhere [Jef85, JM84], so we do not do so here. Experimental results [WHF+89, JBW+87] have demonstrated that Time Warp can efficiently use many processors to execute a single parallel VT (simulation) program.

Since its invention at Rand in 1982 [JS82, JS85, Jef89], Time Warp has been extensively investigated by Jefferson and others. The investigations include experimental prototypes developed at JPL [JBH+85, WHF+89, JBW+87, HBL+89, RFBJ90, ELP+89, PEWJ89], The University of Utah [Fuj88a, Fuj88c, Fuj87, Fuj88b, Fuj89], and elsewhere [TK88]; theoretical research at UCLA [Jef85, Sam84] and USC [JM84, Gaf88, Gaf85, BJ85, Ber86]; design of specialized hardware by Fujimoto [FTJJG88, Fuj88c]; and performance analysis [RMM88, RM88, LL89a, LL90, LL89b, LL89c, Nic89, LMS83]. Research by the Jade project at the University of Calgary [LU85, UDCB86, LCWU88, Lom88, Wes88, BCLU89] has led to a commercial Time Warp system. For a survey of distributed simulation work see [Kau87, Mis86].

CHAPTER 7

Virtual Time Synchronization of Executions of Distributed Applications with Replicated Processes

7.1 Introduction

In Chapter 6 we showed how to replicate the processes of a Virtual Time (VT) application program. In this chapter we show how to replicate the processes of a distributed application that has been programmed according to the model of Chapter 3. Thus, this chapter describes the first of our two optimistic algorithms for process replication. We transform the distributed application into a VT program, thereby enabling us to reuse some techniques from Chapter 6.

Since Time Warp is the best known general engine for distributed execution of VT programs we first consider the performance of the transformed application program *on top of* Time Warp. We show that the transformed computation will cause unnecessary rollbacks and excessively delay commitment. We fix this problem by incorporating replication *into* Time Warp, thereby creating a new operating system, *Replication Time Warp* (RTW). RTW runs applications with replicated processes with few rollbacks—WRITE and READ messages will not cause rollbacks when they arrive, although CONSISTENCY messages may.

7.2 Sender assignment of event times to messages

The central issue in this chapter is the selection of the virtual times imposed on the distributed application. The replication code selects the RVT of every message sent by the application. The RVT values must be selected so the computation is correct. In addition, the choice of RVT values makes a big impact on the resulting system's performance.

7.3 Architecture

In this section we insert a layer of replication code between the application code and the Virtual Time Machine, which we assume is Time Warp. This layer intercepts the communication primitives calls made by the application's replicas. It assigns RVT values, or *event* times, to the messages sent by the application code. We assume, for the present, that the replication code runs *on top* of TW. The code run by each Time Warp process consists of the replica's application code linked together with the replication code we will present. We assume that the application code 'behaves'. In particular, it does not

- circumvent the replication code by communicating with TW directly, or
- modify the variables used by the replication code, even though it may be able to do so because they share an address space.

This system's architecture is illustrated in Figure 7.1.

To produce a program that satisfies Virtual Time semantics the replication code must assign RVT values so that each message's RVT is greater than its SVT. This requirement is easily satisfied by simply setting message m's RVT to be $m.SVT + 1$.

Figure 7.3 shows an implementation of the replication code. The data types used are defined in Figure 7.2.

The variables the replication code uses are declared above the code. Each of the three interface calls, `receive`, `send`[1] and `output` are shown. The replica is sequential, so at most one piece of replication code is active at a time.

We assume the following services are provided by the TW virtual time machine:

- `vt_receive( m )` receives the next message, storing it into `m`.

[1]We have combined the `READ` and `WRITE` calls into a single send call to shorten the code.

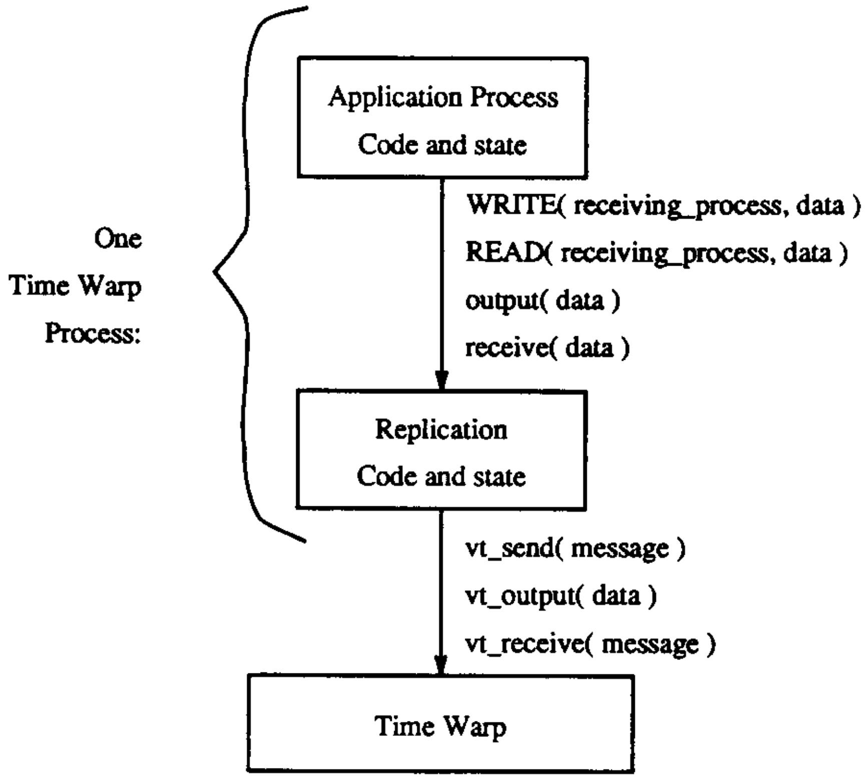

Figure 7.1: Architecture of a replicated application on top of Time Warp

```
replica_name :  record (
 process_name :  charstring,
 replica_index :  integer);
virtual_time :  real;
message_kinds :  enumeration( 'WRITE', 'READ', 'CONSISTENCY');
message :  record (
 SVT, RVT : virtual_time,
 sender, receiver :  replica_name,
 data :  charstring,
 kind :  message_kinds);
```

Figure 7.2: Data type definitions

```
declare
  executing_consistency :  boolean
  local_virtual_time :  virtual_time
  replica_map :  set_of_replica_map_functions
  this_replica :  replica_name  -- init to this replica's name
  m :  message
-----------------------
receive( var data )
  vt_receive( m )
  local_virtual_time := m.RVT
  executing_consistency := ( m.kind = 'CONSISTENCY')
  data := m.data
-----------------------
send( receiver, data, kind )
  if (not executing_consistency) then
    m.data := data
    m.kind := kind
    m.receiver := replica_map( 'best_replica', receiver )
    m.RVT := local_virtual_time + 1
    vt_send( m )
    if ( kind = 'WRITE' ) then
      m.kind := 'CONSISTENCY'
        for replica in replica_map( 'all_replicas', receiver ) do
          if not( ( replica = this_replica ) ) then
            m.receiver := replica
            vt_send( m )
          end if
        end for
      end if
  end if
-----------------------
output( data )
  if (not executing_consistency) then
    vt_output( data )
  end if
```

Figure 7.3: Pseudocode for replication on top of Time Warp

- `vt_send( m )` sends the message `m` to the destination specified in the message field `m.receiver`.

- `vt_output( data )` outputs the data given at the current virtual time.

The *RVT* of the message currently being executed, called the *local virtual time* (LVT) [Jef85], is stored into `local_virtual_time`. The local virtual time is always copied into the *SVT* of a message that is sent, so the statement `m.RVT := local_virtual_time + 1` sets the message's *RVT* to be one greater than its *SVT*.

The replicas of a replicated process are kept consistent by assigning the same *RVT* to all CONSISTENCY message copies of a WRITE message, and disseminating the CONSISTENCY messages to each replica of the process.

The result is a completely deterministic execution of the application program. All events which are n message transmissions away from the beginning of the computation execute at the same virtual time, n. This deterministic execution may perform extremely poorly, because the replicas' LVTs can drift apart. In fact, the difference between the LVT of two replicas at a given real time is unbounded. For example, consider the computation shown in Figure 7.4, which labels each message with its *RVT*. Virtual time advances quickly at processes P and S because they communicate among themselves frequently, whereas virtual time advances slowly at process T. When message M sent by T arrives at S it causes S to roll back two message executions, since two messages with earlier *RVT* values have already been executed.

Another consequence of the fact that the LVTs may differ so greatly is that parts of a computation can wait a long time to commit. In the example, the message with *RVT* of 6 executed at P will not commit until replica T executes a message with *SVT* greater than 6.

These problems arise because the *RVT* values assigned to messages bear no relation to real time. In the next section we show how the *RVT* of a message can approximate the real time as which it is received, so virtual time advances more

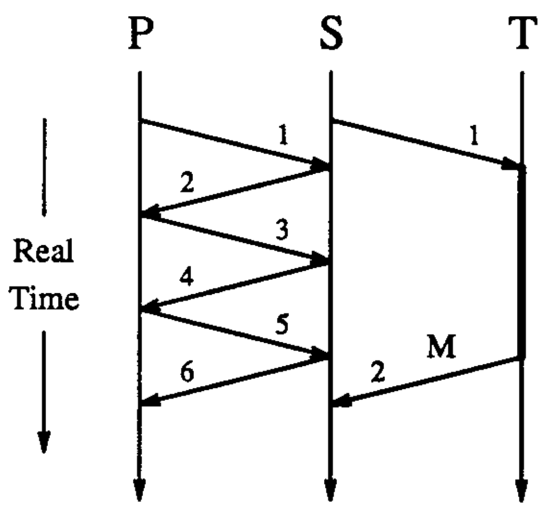

Figure 7.4: Message M causes rollback at process S

smoothly.

7.4 Incorporating replication into Time Warp

Imagine that all replicas can read accurate real-time clocks and that all communication delays are precisely known. Then a message sender could precisely predict the real-time of the message's arrival. Figure 7.5 shows a computation in which the sending replication code sets each WRITE and READ message's *RVT* to precisely the real-time of the message's arrival at its receiving replica.

We assume the application runs on the network system in the top of Figure 7.5, which shows the placement of replicas on processors and the deterministic delay of 1 time unit between adjacent processors. Every READ and WRITE message arrives at a replica when the local real-time equals the message's *RVT*. These messages cannot cause a rollback because no message with a larger *RVT* could have arrived earlier. A CONSISTENCY message can cause a rollback, however, because it must have the same *RVT* as a WRITE, but is delivered to a different processor at a later time. Although the first WRITE sent from *S* to *R* in Figure 7.5 updates both

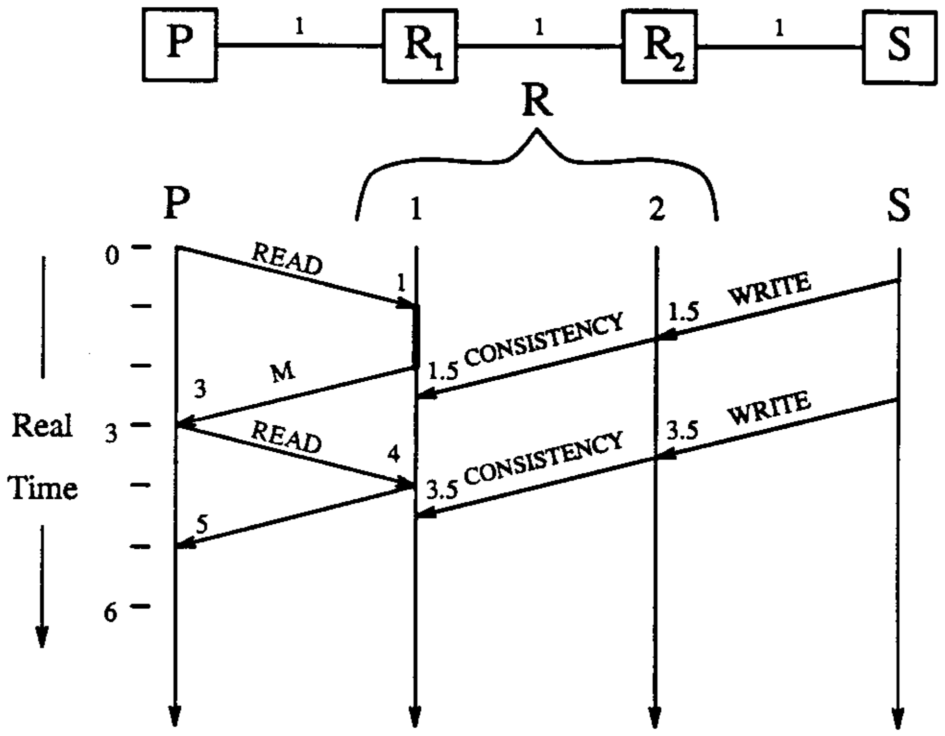

Figure 7.5: Message RVT equals real-time of message arrival

replicas of R without a rollback, the second one rolls back R_1. The execution of the READ sent by P at real-time 3 will be rolled back.

This design is idealized, because the assumptions that clocks are perfectly synchronized and message communication delay is precisely predictable are both unrealistic. However, we have designed a new operating system which has the same performance results by incorporating replication *into* Time Warp, thereby creating *Replication Time Warp* (RTW). RTW is a modified version of Time Warp that runs applications with replicated processes.

With replication incorporated into the OS there is greater freedom to manipulation virtual times. RTW sets the RVT of a READ or WRITE message *after* the message arrives at its receiving replica, so that enqueuing the message cannot cause a rollback. In addition, RTW sets the RVT to be as large as the local real-time, so that virtual time advances at approximately the rate of real-time. RTW is a combination of all the functionality of Time Warp to execute VT programs with the functionality of the replication code to manipulate virtual times and keep the replicas of a process consistent.

Application replicas run directly on RTW, as shown in Figure 7.6. Each appli-

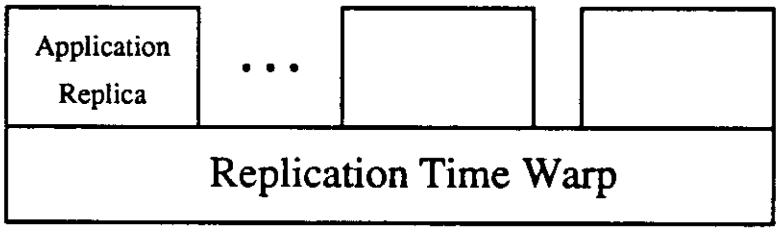

Figure 7.6: Replication time warp (RTW) architecture

cation replica is a single RTW process. Whereas Time Warp runs Virtual Time programs, RTW runs applications with replicated processes.

Unlike a Time Warp process, an RTW application replica never reads or writes virtual time values. While executing a message a replica *cannot* read its LVT.

When a replica sends a message the application code does *not* assign the message's *RVT*.

Whereas every Time Warp message always contains an initialized *RVT*, in RTW the *RVT* of a READ or WRITE message is still uninitialized while the message is in transit from its sender to its receiver. Instead, RTW assigns the *RVT* of a READ or WRITE message at the last possible moment—just before the message is enqueued in its receiver's IMQ. By postponing assignment of the *RVT*, RTW can guarantee that enqueuing the message will not cause a rollback.

RTW maintains a logical clock C [Lam78] for each replica. C always satisfies an important invariant:

- Invariant LC (logical clock): The value of C maintained for a replica is always greater than the largest *RVT* of any message that has been executed by the replica.

This invariant is maintained by ensuring that the value of C maintained for a replica exceeds the *RVT* of each message that arrives at the replica before the message is enqueued in the replica's IMQ.[2]

We also assume that RTW can read a local real-time clock T. T need not be globally synchronized, nor does it need to be highly accurate. The algorithm will work correctly even if T stops advancing entirely. We do assume that failure does not set T backwards.

The purpose of T is to provides a source for *RVT* timestamps. The *RVT* assigned an original message is always at least as large as the local clock T. If the T clocks at different replicas are synchronized then the *RVT* assigned a message at globally synchronized real-time *gt* will be *gt*. In this situation, GVT will advance to *gt* when all messages whose *RVT* values were assigned by real-time *gt* are executed or unsent by rollback. This avoids the slow commit problem described in Section 7.3.

[2]C is *not* part of a replica's state, so it does not get restored when the replica is rolled back.

When a READ or WRITE message m arrives at a replica, RTW stores the maximum of C, T and m.SVT + 1 into the message's *RVT*. Invariant LC guarantees that enqueuing m will not cause a rollback. When a CONSISTENCY message arrives at a replica, its *RVT* is already initialized. Enqueuing it may cause a rollback.

7.5 Pseudocode that incorporates replication into Time Warp

This section shows the replication code that will be combined with Time Warp to make RTW. Almost all of RTW is implemented by adding two pieces of code to Time Warp (see Figure 7.7).[3] The first piece (shown in Figure 7.9) handles calls

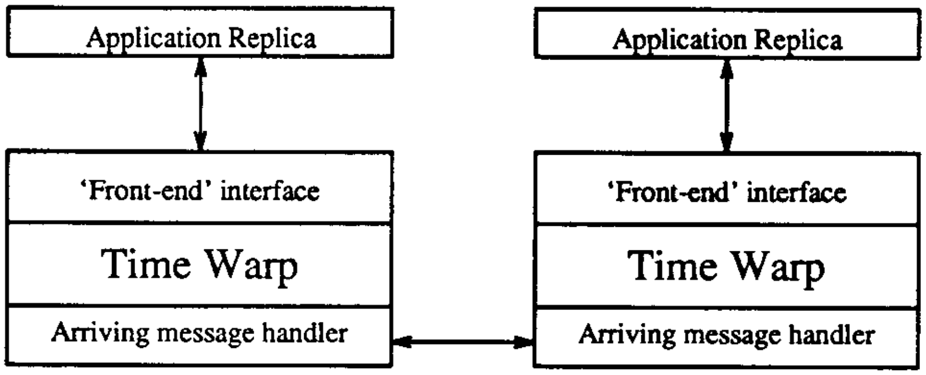

Figure 7.7: Replication time warp (RTW) architecture (detail)

made by an application replica, and can be thought of as a 'front-end' for Time Warp. The second piece (shown in Figure 7.10) of code handles user messages as they arrive at an RTW operating system. It maintains C and assigns *RVT* values to WRITE and READ messages.

Figure 7.9 shows the RTW front-end for Time Warp.[4] Most data types used in this code were shown earlier, in Figure 7.2; the new ones are shown in Figure 7.8. We assume that the type input_message_queue is the type of the TW IMQ. The

[3]Some other small changes to TW are discussed in Sections 7.5.2 and 7.5.3

[4]We have combined the READ and WRITE calls into a single send call to shorten the code.

IMQ is shared by several pieces of code, so we assume the shared use is controlled so each access is atomic.

```
rtw_message :  record (
  SVT, RVT : virtual_time,
  sender, receiver :  replica_name,
  data :  charstring,
  kind :  message_kinds,
  uid :  unique_identifier);
input_message_queue :  -- data type declared in TW
```

Figure 7.8: RTW data types

The `receive` call obtains the replica's next input message. We have assumed that `tw_next_message` is the internal Time Warp call which advances the IMQ pointer to the next message, and returns a copy of the message. (An efficient implementation would access the message directly, rather than copy it.)

`receive` records whether the message the replica receives is a CONSISTENCY message. If so, then send and output calls are no-ops.

`send` builds a message. It simply records whether the message is a READ or a WRITE in `m.kind` and transmits `m` to the 'best' replica of the receiving process.[5]

We assume that `tw_transmit( m )` is the internal Time Warp call which transmits message `m` to the destination indicated in `m.receiver`, and does related work, such as enqueuing a copy of `m`'s antimessage in the sender's OMQ. We have explicitly shown the assignment of a message's SVT to show that all fields of a message but the RVT have been initialized before it is transmitted. The function `replica_map` was defined in Section 4.4.

Figure 7.10 shows the code executed when a user message `m` arrives at RTW. This code assigns an RVT to each READ or WRITE message so that a message's RVT exceeds its SVT, and maintains invariant LC. It then enqueue the message

[5]We ignore error recovery code, which would recover from an incorrect `kind` or a non-existent `receiver`, etc.

```
declare
  m :  rtw_message
  executing_consistency :  boolean
  IMQ : input_message_queue
  replica_map :  set_of_replica_map_functions
  this_replica :  replica_name  -- init to this replica's name
------------------------
receive( var data )
  -- advance IMQ pointer and return next message
  m := tw_next_message( IMQ )
  data := m.data
  executing_consistency := ( m.kind = 'CONSISTENCY')
------------------------
send( receiver, data, kind )
  if (not executing_consistency) then
    m.data := data
    m.SVT := LVT()
    m.kind := kind
    m.sender := this_replica
    m.receiver := replica_map( 'best_replica', receiver ) )
    m.uid := create_unique_id()
    tw_transmit( m )
  end if
------------------------
output( data )
  if (not executing_consistency) then
    tw_output( data )
  end if
```

Figure 7.9: Pseudocode for RTW 'front end' for Time Warp

in the replica's IMQ; we have assumed that `tw_enqueue( m, IMQ )` is the TW procedure that enqueues an input message `m` in the queue `IMQ`.

```
declare
  C : virtual_time
  IMQ : input_message_queue
  replica_map :  set_of_replica_map_functions
  this_replica :  replica_name  -- init to this replica's name
------------------------
  'CONSISTENCY':
    C := max( C, m.RVT + 1 )
    tw_enqueue( m, IMQ )
------------------------
  'WRITE' OR 'READ':
    m.RVT := max( T, C, m.SVT + 1 )
    C := m.RVT + 1
    tw_enqueue( m, IMQ )
    if ( m.kind = 'WRITE' ) then
      m.kind := 'CONSISTENCY'
      for replica in
          replica_map( 'all_replicas', m.receiver.process ) do
        if not( ( replica = this_replica ) ) then
          m.receiver := replica
          tw_low_level_transmit( m )
        end if
      end for
    end if
```

Figure 7.10: Pseudocode for RTW arriving message handler

If the arriving message `m` is a CONSISTENCY message then the code ensures that invariant LC will hold after the message is enqueued by doing:

`C := max( C, m.RVT + 1 )`

If the arriving message `m` is a WRITE or a READ message then `m.RVT` is uninitialized. The statement `m.RVT := max( T, C, m.SVT + 1 )` does several things.

- First, `C` is set to be as large as the real-time clock `T`.

- Second, because invariant LC holds and the statement makes `m.RVT >= C`, this statement ensures that enqueuing `m` cannot cause a rollback.
- Third, this statement ensures that the *RVT* of message `m` will be greater than `m.`*SVT*.

The next statement, `C := m.RVT + 1`, ensures that LC will hold after `m` has been enqueued in the IMQ.

To keep replicas consistent, when a WRITE message arrives at a replica, RTW creates and transmits CONSISTENCY messages to all other replicas of the process. The CONSISTENCY messages have the same *RVT* as the WRITE message.

7.5.1 Distinct event times at a replica

An important improvement on this *RVT* selection method can guarantee that no replica *ever* receives multiple messages with the same *RVT*. This avoids the need to sort the messages received by one replica at a single virtual time by their content field, as must be done to make the execution of a VT program deterministic by Time Warp [JBH+85].

We assume each replica of a process has a unique identifier, `replica_unique_id`, which is a real in $(0, 1)$.[6] The statement in the processing of WRITE and READ messages

```
m.RVT := max( T, C, m.SVT + 1 )
```

is replaced by the statement

```
m.RVT := ceiling( max( T, C, m.SVT + 1 ) ) + replica_unique_id
```

The ceiling function makes the result of the maximum operation integer, so that `replica_unique_id` determines the *RVT* value's fractional part.

A replica never receives multiple messages with the same *RVT* for the following reasons. The RTW running replica R_i determines the *RVT* values of all the WRITE and READ messages R_i receives. By the statement above that assigns `m.RVT`,

[6]Replicas of *different* processes can have the same identifier.

the RVT values assigned at one replica are an increasing sequence, so they all differ. The RVT values of CONSISTENCY messages executed at R_i are assigned at replicas $R_{j,j\neq i}$. CONSISTENCY message RVT values assigned at one replica are an increasing sequence, so they all differ. CONSISTENCY message RVT values assigned at different replicas differ because, by the uniqueness of `replica_unique_id`, they all have a different fractional part. Therefore, each message executed by a replica has a different RVT.

7.5.2 Antimessages and annihilation

Time Warp undoes the sending of a message by transmitting the message's *antimessage*. A Time Warp message has a *sign* field, which can store either + or −. When a TW process sends a message m, one copy with a + in the sign field, denoted $+m$, is transmitted to m's receiver; another copy of m with a − sign field, $-m$, is saved in the sender's OMQ. When the sending of m must be undone because of a rollback, $-m$ is removed from the sender's OMQ, transmitted to m's receiver and enqueued in the receiver's IMQ, where $+m$ and $-m$ *annihilate*[7]—both messages are discarded. If the $+m$ message that had been in the IMQ had already been executed, then the process is rolled back to before that execution.

As stated in [Jef85], *To "unsend" a message it suffices simply to transmit its antimessage.* Thus, Time Warp's annihilation rule is that

- Two messages annihilate if they have the same content (*sending process*, SVT, *receiving process*, RVT and *data*) and different signs.

Because RTW assigns a READ or WRITE message's RVT after the message arrives at its receiver, the $-m$ copy in the OMQ does not store the RVT. Therefore, RTW cannot use TW's annihilation rule—the messages would not match. To solve this problem, RTW creates a unique identifier (uid) each time a message is sent by an application replica. Thus, Figure 7.9 contains the statement

[7]Like subatomic particles and anti-particles annihilate, hence the name.

```
m.uid := create_unique_id()
```

which assigns a unique identifier to m before it is sent. The same uid is written on both the $+m$ transmitted to m's receiver and the $-m$ saved in the sender's OMQ. In RTW, two messages annihilate if and only if they have opposite signs and the same uid.

This approach loses the elegance of matching and annihilating messages solely on the basis of their content. (Other TW implementations have modified the mechanism that matches a message with its antimessage. For example, Fujimoto's [Fuj88a] shared memory TW stores a pointer to a transmitted message in the OMQ. To send an antimessage the algorithm follows the pointer and annihilates the message it finds.)

How is the sending of a CONSISTENCY message undone? Suppose replica R_i sent WRITE message w to replica P_j. If R_i is rolled back and the sending of w is undone, then sending the CONSISTENCY message copies of w must also be undone. Therefore, if $-w$ is transmitted from R_i to P_j, RTW also transmits an antimessage $-w$ to each other replica of process P.[8] (This part of the code is not shown since it is buried in the rollback implementation.) $-w$ has the same `uid` as the CONSISTENCY messages because the code (in Figure 7.10) that creates the CONSISTENCY messages copies the `uid` already in the WRITE message. Therefore, when $-w$ arrives at the other replica it will annihilate the CONSISTENCY message.

As part of this mechanism, when a CONSISTENCY message is created no negative signed message is saved in the OMQ. We assume that the TW function `tw_low_level_transmit( m )` used in Figure 7.10 to send CONSISTENCY messages simply transmits a message without enqueuing a negative message in the OMQ.

7.5.3 Commitment and termination

Time Warp defines GVT at real time r is defined as the minimum of (1) all virtual times in all virtual clocks at time r, and of (2) the RVT of all messages

[8]This assumes that the RTW running R_i statically knows the set of all replicas of R.

that have been sent but have not yet been processed at time r.

Because the C clocks do not advance when a replica does not receive messages, for RTW we modify the definition of GVT as follows: GVT at real time r is the minimum of (1) all clocks T at time r, and of (2) the *RVT* of all messages that have been sent but have not yet been processed at time r. (This assumes that T will not be set backward for any reason.)

We change the definition of GVT because a READ or WRITE message in transit in an RTW system will have an unknown *RVT*. GVT is at least as small as the global minimum of all T clocks because the smallest *RVT* that can be assigned at real-time r or later is at least the minimum of all clocks T at time r.

Unlike Time Warp, RTW will never advance GVT to ∞, so the TW termination criteria of GVT $= \infty$ would never be satisfied. Instead, an RTW computation terminates when there are no messages which have been sent but have not been completely executed.

7.5.4 Deterministic replay

It would be a performance drawback to require that optimistic systems like Time Warp save a process's state before each message execution. Therefore, Time Warp checkpoints a process's state only intermittently. It recreates an intermediate state needed by a rollback by restoring the process to an earlier state and then re-executing messages from the process's IMQ until the process achieves the desired state. It is essential that the replay be *deterministic*, that is, that re-executing messages precisely recreate the process's state.

Clearly, reading a real-time clock is not a deterministic action. Therefore, when RTW rolls back a replica and re-executes WRITE and READ messages from the replica's IMQ, RTW does *not* re-assign new *RVT* values to the messages. An *RVT* value is only assigned to a WRITE or READ message once—when the message arrives at processor running the replica.

Thus, replay of an RTW replica is deterministic.

7.6 Conclusions

We have presented *Replication Time Warp* (RTW), a new operating system that executes applications with replicated processes. This is the first of our two optimistic replication algorithms. RTW uses the synchronization power of VT to update the replicas of a process atomically.

RTW is a small extension to the Time Warp distributed simulation system. It incorporates all the mechanisms of Time Warp. In addition, RTW internally determines the *RVT* of each message. *RVT* values are chosen so that a READ message or a WRITE message cannot cause a rollback when it is enqueued. CONSISTENCY message copies of a WRITE are created with the same *RVT* as the WRITE message so that the replicas of a process are kept consistent. A CONSISTENCY message can cause a rollback.

CHAPTER 8

A Dependency-Tracked Mechanism for Optimistic Process Replication

8.1 Motivation

Pessimistic process replication suffers from a major performance drawback: the update protocol prevents each replica from receiving new messages for a long interval. The interval is at least as long as the replica's round-trip communications delay to the primary replica. The interval begins when the receive service enqueues a CONSISTENCY(i) message in the IMQ and does not finish until the service receives the COMMIT(i) message, as shown in Figure 5.2. Thus, a single WRITE ties up all the replicas of a replicated process for an extended period.

This poor performance frustrates us as designers, because the pessimistic replication mechanism slows execution to prevent replication errors that we believe would almost never occur. For example, suppose a READ arrives at a replica of process R while the replica cannot receive the READ because it has enqueued an uncommitted CONSISTENCY. If the READ does not depend on the update by the uncommitted CONSISTENCY message or any other uncommitted UPDATE, then the READ could be received by R's replica without causing a replication error. Communication paths make it unlikely that the READ depends on the update by the uncommitted CONSISTENCY; the CONSISTENCY message traveled directly from the primary replica, whereas the READ traveled from another process which had earlier received a reply from a replica of R. We see these differences in speed in the figures showing prototypical replication errors, like Figure 4.4. Of course, the replication mechanism must prevent these errors. Because we assume message delivery can take arbitrarily long, the mechanism must protect against the worst cases.

Since actual replication errors will be extremely infrequent the pessimistic mechanism needlessly delays execution. This *optimistic* mechanism takes a different strategy. It *tracks dependency* in the computation so it knows which state changes must have taken place before each point in a computation. The dependency tracking enables the optimistic mechanism to *prevent* W-W and R-W errors and detect and undo C-D errors. If C-D errors occur extremely rarely, as we believe, then undoing them will be an insignificant performance cost.

8.2 Principles of optimism

Optimistic concurrent computation can perform better than an equivalent pessimistic computation by executing input that may not be 'correct'; the optimistic computation performs better because it is active while the pessimistic computation would be idle waiting for the input to be proven 'correct'. The optimistic computation makes a 'guess' that the input is correct. The term 'correct' describes the state of the input with respect to some condition of the distributed algorithm. For example, in process replication an input message is 'correct' if executing it cannot cause replication errors.

The performance of an optimistic computation should be evaluated by comparing it with the performance of the equivalent pessimistic computation. An optimistic computation's performance will depend on the trade-offs between the benefits of executing inputs earlier and the costs of undoing the execution of 'incorrect' inputs plus the costs of saving the history that make undoing possible.

An optimistic computation is a complicated algorithm because it must have the capability to undo the distributed effects of executing 'incorrect' input. An optimistic computation tracks the effects of one part of a computation on another. This tracking, called *dependency tracking*, supports the propagation of *commitment* and *rollback*:

Commit Part of a computation *commits* when it is correct and all parts of the computation it depends on have committed.

Rollback Part of a computation must be *rolled back* if it depends on input that will never commit.

An optimistic computation postpones actions that *cannot* be undone, such as producing output, until the action depends only on committed input.

The optimistic process replication system architecture (see Figure 8.1) is centered around the *replica history manager*. The replica history manager provides the same application program interface as the pessimistic and Time Warp based optimistic replication mechanisms discussed earlier. It stores the data and implements the algorithms that support commitment and rollback.

The history manager's actions center around two data structures:

- an *input message queue* (IMQ) which stores an ordered sequence of the replica's recently executed and as yet unexecuted input messages[1], and
- a *state queue* (SQ) which stores copies of the replica's recent states.

These will be discussed more thoroughly in Section 8.5. In the next few sections we will discuss how optimistic process replication tracks dependencies, how it commits part of a computation and how it rolls back part of a computation.

8.3 Tracking dependencies

Tracking dependencies serves two purposes. First, dependency information is used to prevent W-W and R-W errors and detect C-D errors. Second, dependency information helps propagate commitment and rollback.

The *dependencies* the system tracks are updates of replica states. A dependency is denoted P_i, which indicates the ith update of process P's state.

Since execution of a READ does not change its recipient's state it does not create a dependency to track. Any later message that accesses the replica cannot

[1]A note on wording: Given an IMQ $\cdots, M, \cdots, N, \cdots$ whose messages will be executed from left to right, M before N, we say message M comes before N, or is earlier than N, or is older than N; symmetrically N follows M, or is younger than M, or comes after M.

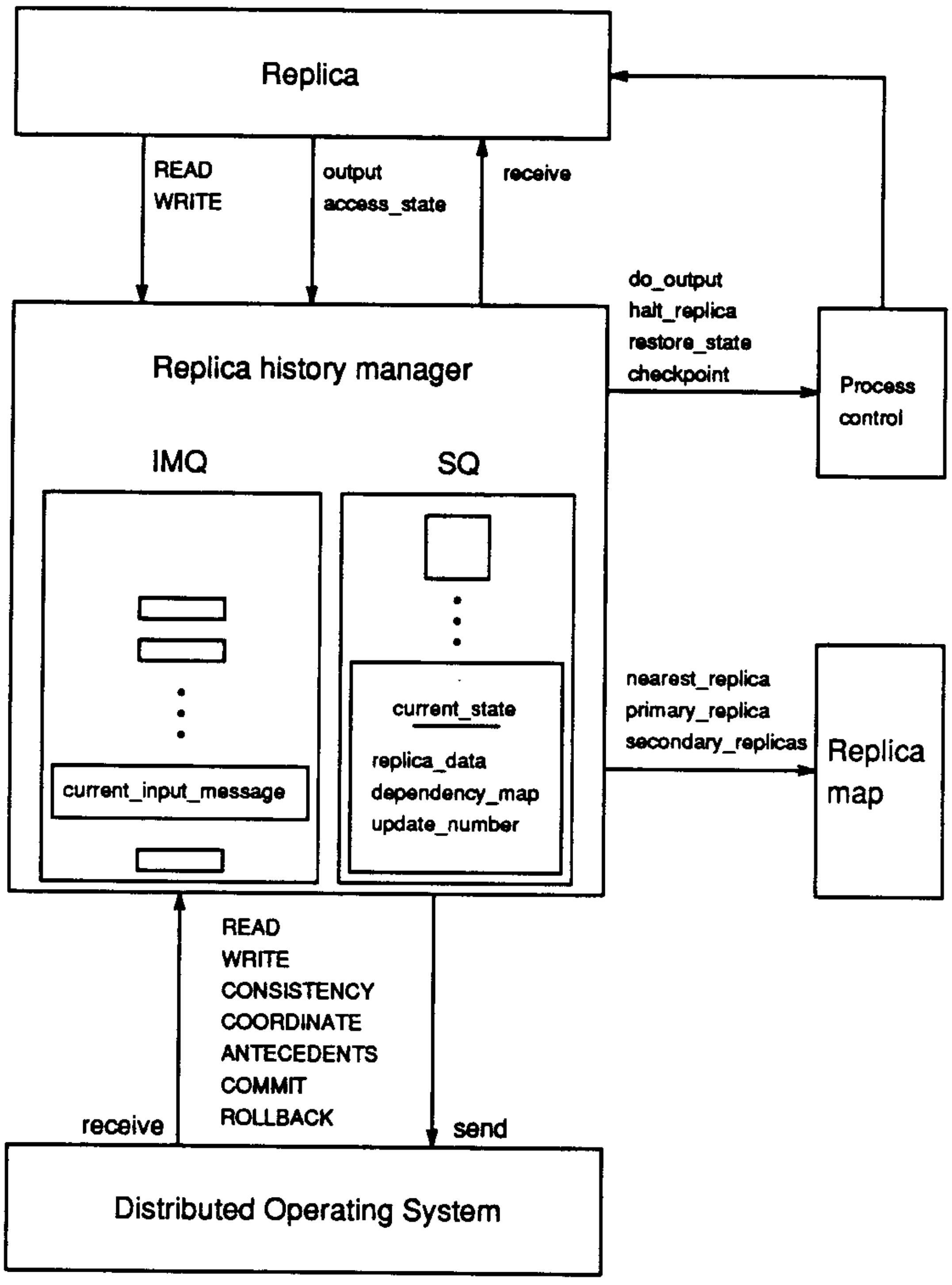

Figure 8.1: Optimistic process replication system architecture

determine whether the READ has been executed, so there is no need to require that the READ was executed earlier.

The state updates that "happen before" [Lam78] a particular event in a computation are those that can be traced forward to the event through communication in messages and states. The dependencies of a message or state are stored in its *dependency map* (like an Optimistic Recovery dependency vector, as in [SY85], which is denoted by a set in comparison signs: $< P_i, Q_j, R_k \cdots >$. Since P_i precedes P_{i+1} by definition, a dependency map contains at most one entry for each process—the most recent previous update at that process.

Dependencies are tracked by including a dependency map in every message. When a replica sends a message, the replication mechanism appends a copy of the dependency map of the replica's current state to the message. When a replica receives a message, the message's dependency map is *merged* into the dependency map of the replica's current state. The merged dependency map contains a union of the dependencies in the message and state dependency maps. If both the message and state maps depend on the same process, then the merged map shows a dependency on the latest of the two dependencies.

When a replica receives an UPDATE message, the update's dependency is specially merged into the replica's state. Thus, when a replica of process Q executes WRITE(i) or CONSISTENCY(i) it adds Q_i to its dependency map. Figure 8.2 illustrates dependency tracking in a small computation. The dependency map carried by a message is shown as a <> enclosed list following the message. The dependency map of a replica's state is shown beside the state line.

The update protocol attempts to *commit* an update to a process's state. Unless it has been committed, we say that the update is *uncommitted.*

An important performance optimization comes from the recognition that committed dependencies do not need to be tracked because executing a message that depends on only committed updates cannot cause a replication error. If a dependency map does not depend on any uncommitted update of process P then the

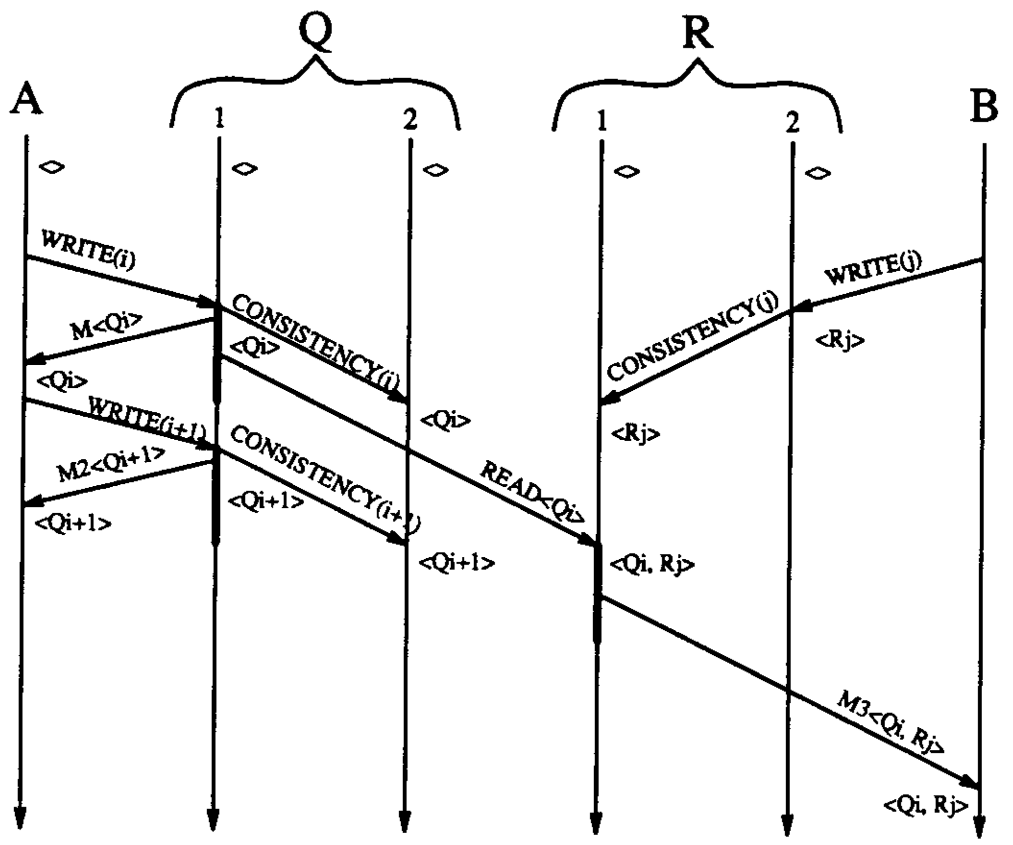

Figure 8.2: Dependency tracking example

map need not store any entry for P.

8.4 Update protocol

A WRITE message is routed to its receiving process's *primary* replica, whose replica history manager assigns the next un to the WRITE. (We initially discuss a protocol coordinated by a primary replica because we feel it is easier to reason about than a fully distributed protocol. Later, in Section 8.9.2, we describe a distributed update number server that can decentralize the functions of the primary replica without changing the design of the update protocol.) For *un* i a CONSISTENCY(i) message is sent to each of the process's secondary replicas, and the WRITE(i) message is enqueued in the primary replica's IMQ. A CONSISTENCY(i) message has the same user message content and the same dependency map as WRITE(i).

8.4.1 An example

Figure 8.3 shows a sample execution of the optimistic update protocol in which the update commits without a rollback. A WRITE is sent by process A to replicated process R. R's primary replica[2] assigns *un* i to the WRITE and begins the update protocol. WRITE(i) is delivered to the primary replica, which begins executing it immediately, as shown by the thick vertical line. Process R sends a message called M back to process A. R sends M after receiving WRITE(i) so M's dependency map contains R_i. While process A executes M it produces some output; since the output depends on the uncommitted update R_i, the replica history manager for A delays the output.

Meanwhile, process R performs the update protocol. CONSISTENCY(i) is sent to the other replica, which sends back a COORDINATE(i) message. A special dependency map (shown in parentheses) is carried in the COORDINATE(i) message

[2]When we say "R's primary replica" we mean "the replica history manager managing R's primary replica".

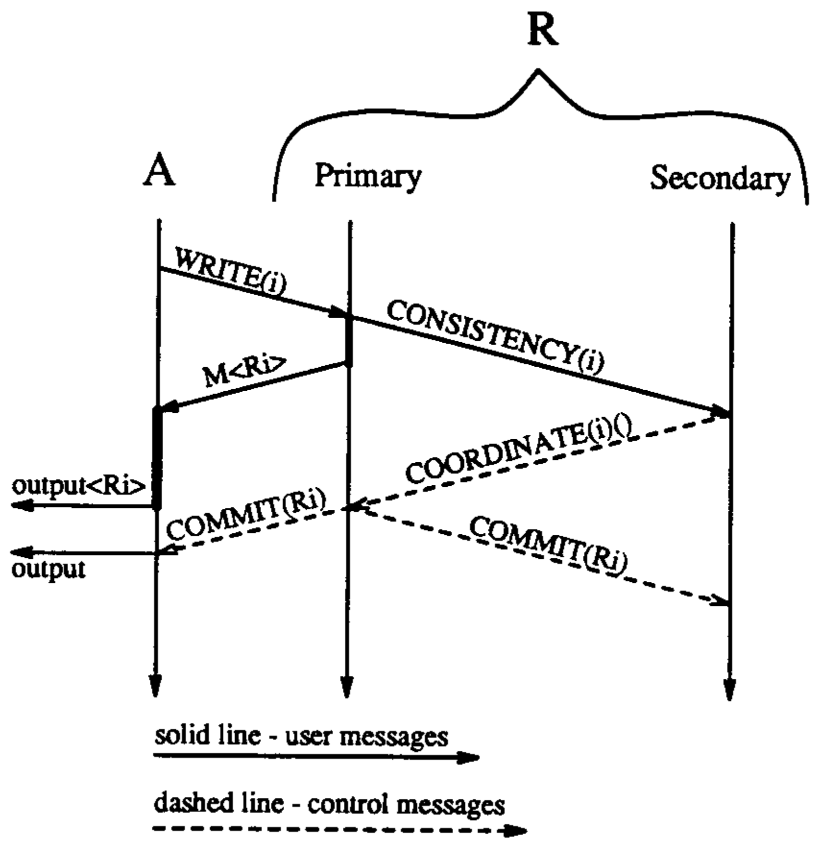

Figure 8.3: Optimistic protocol

to indicate all *new* additional dependencies picked up at the secondary replica. In the example, the COORDINATE(i) message's dependency map is empty because update R_i did not acquire any additional dependencies.

When COORDINATE(i) arrives, R's primary examines the status of update i. First, COORDINATE(i) messages have arrived from all secondary replicas. Second, update i does not depend on any uncommitted dependencies. Satisfying these two conditions commits update R_i, so R broadcasts COMMIT(R_i). When A's replica history manager receives COMMIT(R_i) everything the output depends on has committed, so the output can actually be written.

This example illustrates the performance benefits of optimism. Optimism has enabled R's primary replica to execute WRITE(i) earlier than the pessimistic protocol would have allowed. As a result M was returned to process A earlier, thereby allowing A to execute it in parallel with the update protocol. When the update committed, A had already completed the execution of M. The output has been produced earlier. It was sped up by an amount of time slightly less than the sum of the execution times of the WRITE message and M.

8.4.2 Committing an update

An update commits if executing its UPDATE messages cannot cause replication errors. To commit an update, the following conditions must hold:

C1 All replicas of the process have been told about the update.

C2 All other updates earlier than the update have already been committed.

If these conditions have been satisfied for update i at process R then executing its messages cannot cause replication errors. We show this in Section 8.8.3.

The primary replica determines that these conditions hold for update i when it finds that:

1. All other replicas have replied to the CONSISTENCY(i) message by sending a COORDINATE(i) message.

2. All updates that the update depends on, including the dependencies communicated in COORDINATE(i) messages, have committed.

The protocol implements this as follows. A WRITE in the IMQ has fields indicating the message's *un* (update_number) and the remaining number of acknowledgements from other replicas it needs (coordinates_needed). (See Figure 8.8). When a COORDINATE(i) message arrives the coordinates_needed count in WRITE(i) is decremented. R's primary replica commits update i when the coordinates_needed count in WRITE(i) goes to 0 and the dependency map in WRITE(i) becomes empty. Unless the update is in a "dependency loop", this will eventually happen. Other some replica will need to be rolled back.

8.4.3 Rollback

If the timing of message arrivals never causes a potential C-D error (like the one in Figure 4.6) then all updates will eventually commit.

Potential W-W errors and potential R-W errors are resolved by using local information to order messages correctly. However, if a dependency loop occurs then at least one of the updates in the loop must be rolled back to enable commitment to continue.

If a dependency loop occurs (see Figure 8.4) then the updates involved cannot commit because condition **C2** is not satisfied. In Figure 8.4 both primary replicas have received COORDINATE messages from their secondary replicas. However, Q cannot commit update i because it depends on uncommitted dependency R_j. Let us use $x \rightarrow y$ to indicate y depends on x. In this example Q's primary replica observes that $R_j \rightarrow Q_i$. Symmetrically, R's primary replica observes that $Q_i \rightarrow R_j$. This is a *dependency loop.*

Neither primary replica can commit the update. (Note that Figure 8.4 shows an C-D error involving the two processes with two replicas each. This is the minimum number that can participate in a dependency loop. Other loops involving more

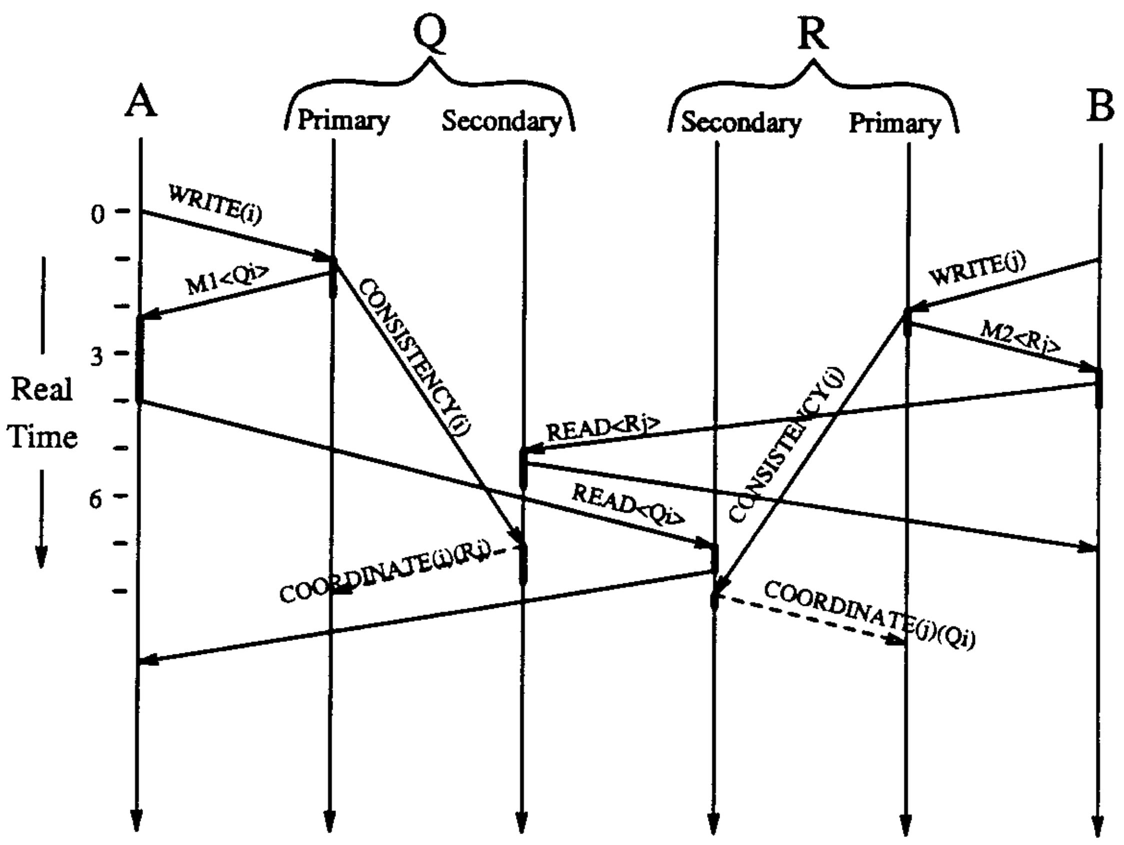

Figure 8.4: A dependency loop in the optimistic protocol

processes and/or replicas can occur.)

This dependency loop must be broken so the computation can progress. Since both Q and R's secondary replicas have executed the READ at least one of them will have to be rolled back.

To make sure the computation makes progress, one of the updates in the loop is not rolled back. Otherwise the dependency loop could form repeatedly.

To ensure that an update does not get starved—that is, to prevent an update from repeatedly becoming part of a dependency loop and then being rolled back—we always do not rollback the update with the earliest *real time* timestamp. The timestamp is assigned when a WRITE arrives at a primary replica. These timestamps can be picked from the physical clocks described in Section 7.4, which only need to be logical clocks for correctness.

A primary replica detects that it must roll back an update when it determines that the update participates in a dependency loop and that the update is not the earliest update in the loop.

When the `coordinates_needed` count in WRITE(i) goes to 0, if the message's dependency map still contains dependencies other than R_i then the primary replica sends a DEPENDENCIES message to the primary replica of each process in the dependency map. The DEPENDENCIES message indicates an edge in a potential dependency loop and the timestamp of the update. For example, in the execution of Figure 8.4 Q's primary will send a DEPENDENCIES $(Q_i, 1.0)(R_j)$ message to R's primary replica, as shown in Figure 8.5. The message indicates that Q's primary knows $R_j \rightarrow Q_i$, and that the timestamp of Q_i is 1.0.

Process R's primary detects the loop shown in Figure 8.6 and rolls back R_j because its timestamp, 2.0, is greater than the timestamp of Q_i, 1.0. R broadcasts ROLLBACK(R_j) messages.

When a replica's history manager receives a ROLLBACK(R_j) message it undoes all computation that depends on dependency R_j and removes dependency R_j from all dependency sets. In this example, the rollback will undo message $M2$, the READ

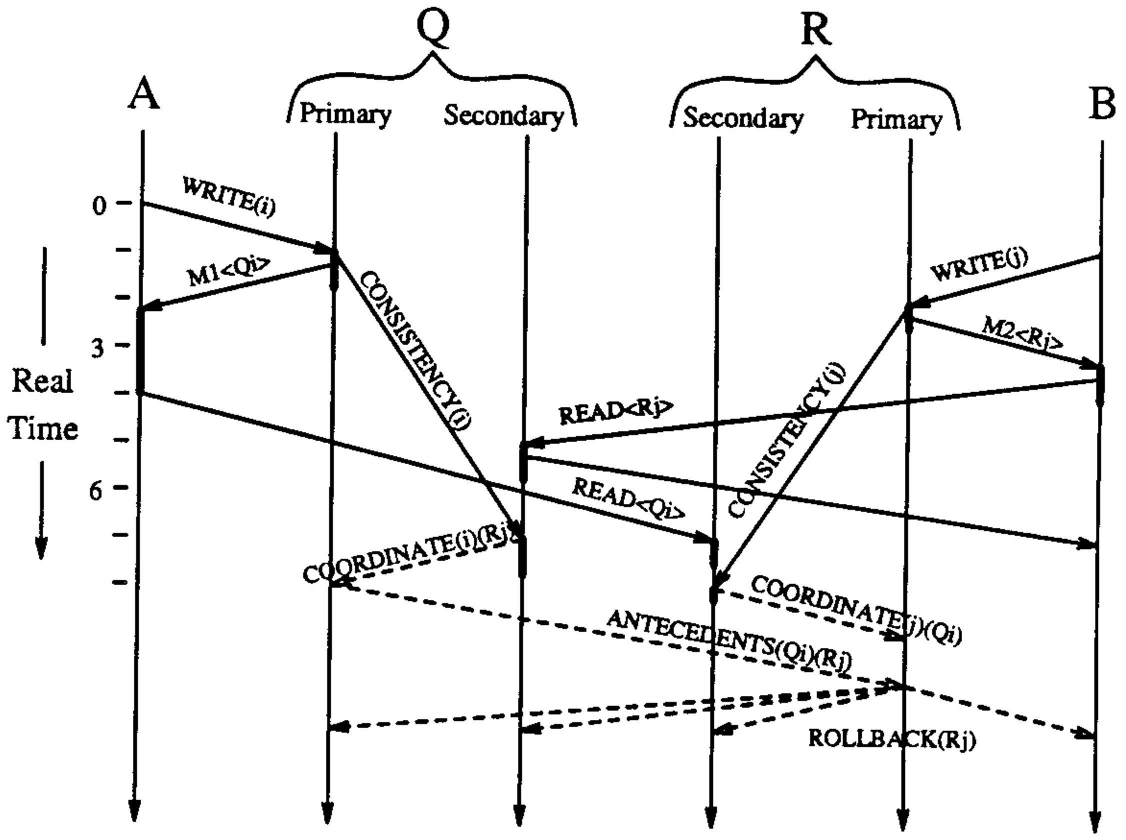

Figure 8.5: Dependency loop detected

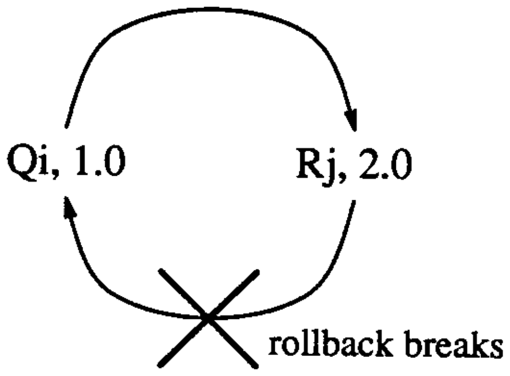

Figure 8.6: Dependency loop

$M2$'s execution sent, and CONSISTENCY(j).

After the rollback, Q's primary will be able to commit update i because it will no longer depend on uncommitted update R_j. R will resubmit WRITE(i) and it will commit after R_j. The final outcome is shown in Figure 8.7.

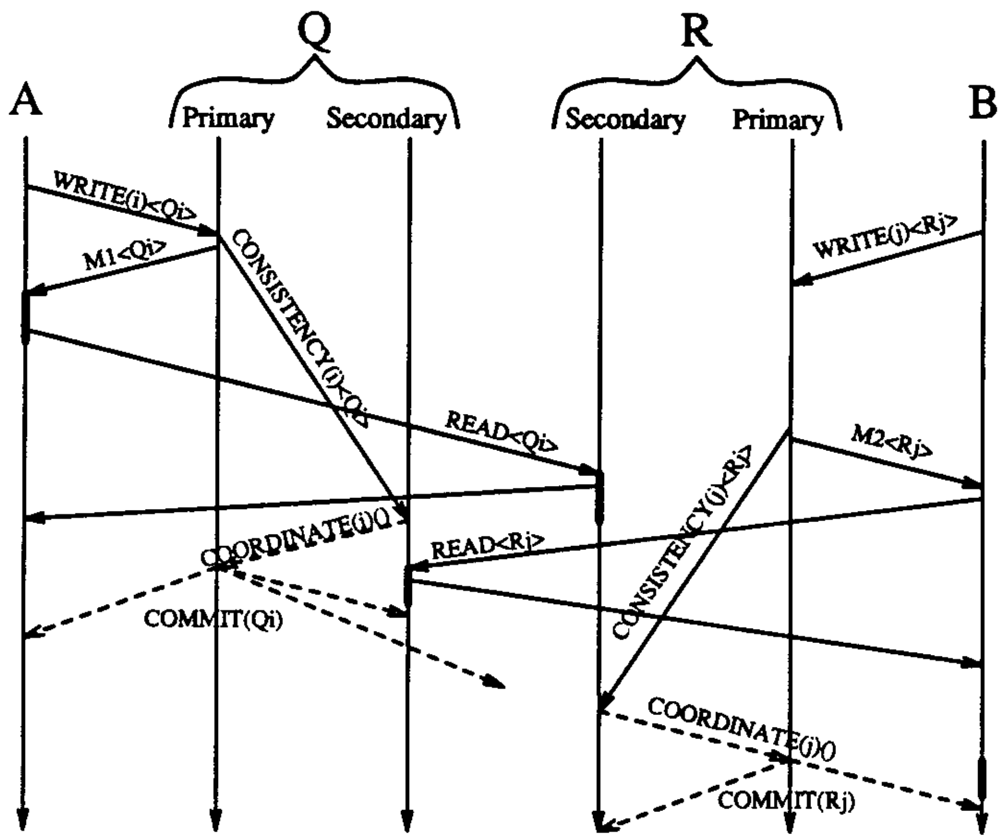

Figure 8.7: Commit after rollback

A complete discussion of rollback and detecting and breaking dependency loops is presented after a discussion of the structure of the replica history manager.

8.5 The replica history manager

A replica history manager controls one replica of a replicated process (see Figure 8.1). The manager makes the replica's execution consistent with its local view of causality. This section discusses the detailed design of an optimistic replica history manager.

The history manager controls a replica's execution by handling its message sends and receives, and its output operations. An optimistic manager also needs the capabilities to interrupt the replica's execution, save its state and restart it from a saved state. We assume these services are provided by the local operating system or can be written in terms of its function.

Recall that the primary data structures in a replica history manager are the *input message queue* (IMQ) and the *state queue* (SQ).

An entry in the SQ stores a snapshot of the replica's *state*, and some control information. The snapshot contains a copy of all the data accessible to the replica's code, such as the stack, the heap, etc. The snapshot, or checkpoint, is taken while a replica is suspended in between message executions. Thus, a state in the SQ fits into a particular slot in the sequence of messages in the IMQ. There can be several WRITE message executions in between checkpoints.

Performance considerations determine the best times to take checkpoints; the storage and computation cost of taking checkpoints must be traded-off against the benefit of short rollbacks. Evaluating these trade-offs quantitatively is beyond the scope of this dissertation.

The history manager stores a replica's recent states and input messages to give it the ability to undo uncommitted message executions. To support rollback, the SQ must store one state older than the earliest uncommitted input message. This is a committed state. All input messages younger than the youngest committed state must be retained in the IMQ. As older input messages get committed, unnecessary input messages and states should be discarded to reclaim storage.

A message in the IMQ stores the contents of the message the sending process sent and control data used by the replication algorithm. Some of the contents of an input message are described by the Hermes [SBG+91][3] data type definitions

[3]A record type is declared by `type_name : record ( field_name : field_type , ...`

in Figure 8.8. The message_content field stores the message contents created by the sending process. kind identifies whether a message is a WRITE, a READ or a CONSISTENCY. dependencies stores the message's dependency map, as discussed above. As COMMIT messages convey the information that dependencies have been committed, the dependencies are removed. Conversely, as ROLLBACK messages convey the information that updates are being undone, input messages containing these dependencies are discarded. Eventually, an input message's dependency map must become empty. If an input message's execution might be undone, then output produced while executing the message is buffered until the rollback cannot occur. output_messages stores the buffered output. Figure 8.8 assumes output is a charstring, although any type of data could be output, of course.

end_of_replay supports replay after a rollback. A rollback sets it to 'true' in the message whose execution marks the beginning of normal execution after the replay.

The last four fields are not used in READ messages. Only UPDATE messages maintain update_number. It is the update number of this state update, as we have discussed. Only WRITE messages maintain timestamp, which is the real time the WRITE arrived at the primary replica, coordinates_needed, which counts the remaining COORDINATE messages expected from secondary replicas, and descendants, which stores the updates that must follow this update, as indicated in DEPENDENCIES messages sent by other replicas.

The message in the IMQ most recently or currently being executed is called the *current input message*. Many operations on the IMQ refer to it.

A state in the SQ contains the replica's checkpoint and some control information. The dependencies records the state's dependencies. The last state in the

). An enumeration is a type with a finite set of values. A polymorph is a type that can hold any type. A table stores a set of elements. Its type is declared by table_type_name : [ordered] table of [keys(field_list)] { typestate } . ordered indicates that the table has an ordered sequence of elements. keys(field_list) indicates that the value in the fields in field_list must be unique in each elements. The typestate can be ignored.

```
enqueued_message :  record (
  message_content :  message_content,
  kind :  message_kinds,
  committed :  boolean,
  dependencies :  dependencies,
  output_messages :  output_messages,
  end_of_replay :  boolean,
  update_number :  update_number,
  timestamp :  real_time,
  descendants :  dependencies,
  coordinates_needed :  integer);
enqueued_state :  record (
  checkpoint :  checkpoint,
  dependencies :  dependencies,
  update_number :  integer,
  in_replay :  boolean );
message_kinds :  enumeration( 'WRITE', 'READ', 'CONSISTENCY');
message_content :  polymorph;
checkpoint :  polymorph;
dependency :  record
  ( process :  charstring;
  update_number :  integer;
  timestamp :  real_time );
dependencies :  table of dependency keys(process) {full};
real_time :  real;
output_message :  charstring;
output_messages :  ordered table of output_message {full};
```

Figure 8.8: Contents of messages in the IMQ and states in the SQ

SQ is the *current_state*, the state the replica references when it accesses data. The `in_replay` field is `'true'` when a replica is replaying its input after a rollback.

The next two sections describe the replica history manager's algorithms: the services it provides to a replica, and the response it makes to messages from other replica managers.

8.6 Application interface services

The history manager supports *send*, *output* and *receive* operations by the application process.

8.6.1 Send and output

Both a message send and output do nothing if the message being executed is a CONSISTENCY message or the replica is in replay, because in both cases the execution is simply recreating the replica's state.

This chapter assumes that the application indicates whether the message is a READ or a WRITE. As discussed earlier (page 34) the `replica_map` provides a `nearest_replica` destination to route READ messages and a `primary_replica` destination to route WRITE messages. When a message is sent the dependency map of the *current_state* is copied into the message's dependency map.

Output should be immediately sent to the outside world if the replica execution has committed up through the current input message, as would be indicated by an empty dependency map for the current state. Otherwise output must be buffered in the input message's `output_messages` field.

8.6.2 Receive and delivering messages to a replica

A history manager's performance objective is to execute its replica's input messages as fast as possible.

A replica requests another input message by issuing a *receive* call. The history manager delivers the next input message from the IMQ to the replica. The history manager will delay delivering the next message if it depends on 'missing' UPDATE messages. Essentially, this delay is an optimization to prevent future rollbacks. For example, if the next message in the IMQ of a replica of process P has P_8 in its dependency map and UPDATE(8) has not already been executed, then the history manager waits until UPDATE(8) arrives before executing the message that depends on it. This delay cannot deadlock because it always waits for a message that must already be in transit.

8.7 Replication system interface services

This section discusses the actions of a history manager processing a message from another history manager.

A user message (WRITE, READ or CONSISTENCY) is the IMQ enqueued after messages that have been executed. In addition, it is enqueued so that the sequence of messages in the queue is consistent with dependency. A message must be enqueued ahead of messages that depend on it:

- An UPDATE(i) is enqueued ahead of an UPDATE(j) message for all $j > i$.
- At a replica of process P an UPDATE(i) must be enqueued ahead of all other messages that depend on P_i.

For example, if an IMQ at a replica of process P contains the messages:
CONSISTENCY(9), READ $< P_{10} >$, CONSISTENCY(11)
then CONSISTENCY(10) must be inserted between CONSISTENCY(9) and the READ.

Another degree of freedom is available in manipulating the IMQ. The history manager can temporarily delay enqueuing a CONSISTENCY message. If the replica is not ready to execute them this could be used to give priority to the execution of READ messages. A READ would be enqueued before CONSISTENCY messages so that it could be executed earlier and depend on fewer uncommitted updates.

However, inserting the READ earlier in the queue would slow commitment of the updates brought by the CONSISTENCY message.

8.7.1 Handling READ, WRITE and CONSISTENCY messages

A READ is enqueued in the replica's IMQ.

A WRITE is assigned the replica's next update number and a real-time timestamp. The coordinates_needed field is initialized to the number of replicas of the process minus 1. The committed field is marked false. CONSISTENCY message copies of the WRITE are made and sent to each other replica. The WRITE is enqueued in the primary replica's IMQ.

When a CONSISTENCY is enqueued in the IMQ The uncommitted dependencies in messages ahead of the CONSISTENCY in the IMQ are merged together and stored in a COORDINATE message, which is sent back to the primary replica.

8.7.2 Handling COORDINATE and DEPENDENCIES messages

When a COORDINATE(i) message arrives the dependencies in the COORDINATE message are merged into the dependencies in WRITE(i). The history manager checks whether update i participates in a loop, as discussed in Section 8.7.5. The coordinates_needed count in WRITE(i) is decremented. If the count drops to 0 the primary replica examines the WRITE message's dependencies. If the set is empty then update i commits. The primary replica processes the newly committed input message, as described in Section 8.7.3.

If the dependencies set is not empty then a dependency loop *may* exist. Suppose P's primary replica is considering update i has received the last COORDINATE(i) message. For each dependency R_j in the dependencies set, the replication code managing P sends a DEPENDENCIES $(P_i)(R_j)$ message to R's primary replica.

When a DEPENDENCIES $(P_i)(R_j)$ message arrives, R's primary replica inserts P_i into the descendants set of WRITE(i). R's manager checks whether update i

participates in a loop, as discussed in Section 8.7.5. Note that the descendants set must be stored because either a COORDINATE or a DEPENDENCIES message can arrive and detect a loop.

8.7.3 Handling COMMIT messages

When update i of process P commits, the replication code running P's primary replica broadcasts a COMMIT(P_i) message. (Section 8.9.1 describes an optimization that may decrease message traffic by replacing a broadcast with point-to-point message transmission to the replicas that need the COMMIT message.)

When a history manager receives COMMIT(P_i) it deletes P_i from the dependency maps and descendants maps of all messages in the IMQ and all states in the SQ.

Removing these dependencies will commit all READ and CONSISTENCY messages that depend only on uncommitted dependency P_i, and all WRITE messages that depend only on uncommitted dependency P_i and have received all COORDINATE messages. These are newly committed messages, and their actions can be released to the outside world.

For all newly committed input messages the manager does several things:

- If the input message is WRITE(j) message update j has committed. A COMMIT(j) message is broadcast.
- Output produced while executing the input message and buffered in its `output_messages` filed is transmitted to the outside world.

Some irreversible actions are done by the operating system, such as reporting a replica's 'divide by 0' or 'memory access violation' error. These irreversible actions must be delayed until they commit. Thus, an error that would cause a replica to fail and be killed must delay killing the replica. The replica may roll back and not die.

Data in the IMQ and SQ that are no longer needed can be discarded to reclaim storage. A state in the SQ commits if all earlier messages in the IMQ have committed. States in the SQ earlier than the latest committed state can be discarded. All messages that have already been executed and are earlier in the IMQ than the earliest remaining state can also be discarded.

COMMIT(P_i) and ROLLBACK(P_i) broadcasts must ferret out all dependencies on P_i, including those in messages in transit.

This problem arises because a user message which depends on P_i could still be in transit to replica R when COMMIT(P_i) arrives at R's history manager.[4]

For this kind of network, each history manager must store a list of the committed dependencies, and use the list to delete the committed dependencies from arriving user messages. Each history manager must also store a list of the rolled back dependencies, and use the list to discard arriving user messages that depend on rolled back dependencies.

Unnecessary entries can be discarded from these lists as the computation progresses. If P_i has been committed, then all P_j for $j < i$ has been committed. In principle, an entry in the rolled back dependencies list can never be discarded, since a message that depends on the dependency could arrive after an indefinite delay. However, other techniques, such as message aging, can be used to eliminate such messages.

8.7.4 Handling ROLLBACK messages

On receiving a ROLLBACK(P_i) message the history manager discards all messages in the IMQ and states in the SQ that depend on P_i. Let M be the earliest message that depended on P_i. If the replica had executed M then the manager must *roll back* the replica. Conceptually, the replica's state is restored to the value it had just *before* the replica executed M and the replica begins executing the remaining messages in the IMQ. To restore the replica to that state, its state

[4]If the network is order-preserving and the history manager processes messages in order, then a broadcast of COMMIT(P_i) can spreads out as a flood from P's primary replica and 'captures' and eliminates all the dependencies on P_i in its wake. Such a network would not need the mechanism below, but we think these assumptions are unreasonable.

is restored from the snapshot in the previous state in the SQ, and messages in the IMQ between the previous state and M are executed in *replay*. Replay only recreates the replica's state. Side-effects, such as message sends and output, are ignored.

Replay must exactly recreate the replica's state. Therefore, the state's value must have been completely determined by the replica's execution of the messages in the IMQ. Deterministic replay can be difficult to achieve, as we discuss more fully in Section 8.8.2.

If a replica that needs to be rolled back is actively executing a message, then it must be interrupted and halted, as discussed by Jefferson in [JBW+87]. The replica's execution must be interrupted because it

The message execution must be interrupted because the replica may be in an infinite loop because it is executing a message out of order. If the execution were not interrupted then the rollback would never happen. Such an infinite loop would occur, for example, if the application code implemented "if I detect that I'm replicated then enter an infinite loop".

8.7.5 Detecting dependency loops

A History manager looks for a dependency loop whenever a COORDINATE or a DEPENDENCIES message arrives. A WRITE message's `dependencies` field contains the updates which precede it, while the `descendants` field contains updates which follow it. A dependency loop exists if an update is in both fields. Thus, a loop always involves a pair of updates. The update with the later timestamp is rolled back. For example, if P's primary replica detects a loop involving WRITE(i) and the timestamp of P_i is later than the timestamp of the other update in the loop, then it will broadcast ROLLBACK(P_i) messages.

All the updates that precede a WRITE will have been stored in its `dependencies` field when a COORDINATE message has arrived from every secondary replica. The `descendants` field will eventually contain all updates which follow the WRITE because dependencies fields become complete and when one does the history manager sends DEPENDENCIES messages to the primary replica of each update in the dependencies fields. Thus, all loops will eventually be detected.

8.8 Correctness issues

8.8.1 Distinguishing among the multiple update protocols of a WRITE

After an update is rolled back it is re-submitted, that is, the update protocol is done again. The dependency labels of these two updates must differ, so that the rollback of the first does not affect any of the second. To implement this each primary replica maintains an *attempts* table mapping each update number to the number of its attempts. The attempt number distinguishes different attempts to do the update, so a dependency identifier becomes a four-tuple: (process, update number, timestamp, attempt).

When a primary replica conducts its first update i protocol it inserts (i, 1) into the attempts table. Each subsequent attempt to do update i increments the attempt count for i. When update i commits, its entry is removed from the attempts table.

These four-tuple dependencies carry implicit rollback information. If a message with dependency (P, i, t, x) arrives then computation that depends on (P, i, t, y) is rolled back if $y < x$. Similarly, the merge operation considers a dependency larger if it has a larger attempt field. All other operations with dependencies remain the same.

8.8.2 Replay and determinism

For best performance, a replica's state should not be checkpointed too frequently. Therefore we place no restrictions on the spacing between checkpoints. As we said in Section 8.7.4, to restore a replica to a state that was not checkpointed the replica's state is restored to an earlier checkpoint and the desired checkpoint is recreated by *replaying* the replica's execution of messages from the IMQ.

The replay must be *deterministic*—that is, it must recreate the exact state that the replica had before it executed the particular message. This requires that all messages and other input which determine the replica's state must be retained in the IMQ. The replay must be deterministic because if it were not the replica would have followed a new and *different* execution path during the replay than it did when previously executing the input messages. The messages the replica sent during the previous execution were already sent to other replicas. Their content will indicate that the replica followed the previous path. Clearly, these two different execution paths are inconsistent.

Describing the determinism requirement abstractly takes no effort; however real systems are messy, and recording all the input which determines a process's behavior can be challenging. Other work done to implement another optimistic algorithm, Optimistic Recovery, on top of the Mach operating system describes some of these problems [GGL+90]. Note that the determinism requirement can be circumvented, if necessary, by checkpointing a replica's state in between each execution of an UPDATE message.

8.8.3 Correctness proof

An update commits if executing its UPDATE messages cannot cause replication errors. It can commit if the following conditions hold:

C1 All replicas of the process have been told about the update and,

C2 All other updates earlier than it have already been committed.

If these conditions have been satisfied for update i at process R then the update cannot cause replication errors.

It cannot cause a W-W error because condition C2 forces all updates j for $j < i$ to have already committed. C1 applied to update j implies that all UPDATE(j)

messages will have already been enqueued. Thus, no UPDATE(i) message could be executed earlier than an UPDATE(j) message.

Recall that a CONSISTENCY(i) message is enqueued ahead of messages which logically follow it:

- An UPDATE(i) is enqueued ahead of any UPDATE(j) messages for all $j > i$.
- At a replica of process P an UPDATE(i) must be enqueued ahead of all other messages that have P_i among their dependencies.

Update i cannot cause a R-W error because any message M that depends on R_i cannot have its results committed until update i commits. At all replicas of R, UPDATE(i) messages will be executed before message M.

Update i cannot participate in an C-D error because condition C1 implies that each other replica of R has replied to CONSISTENCY(i) with a COORDINATE(i) message and C2 implies that none of those replicas can depend on an uncommitted update.

8.9 Important optimizations

To simplify this exposition we have left out several important optimizations that we now describe.

8.9.1 Targeted routing of ROLLBACK and COMMIT messages

To reduce message traffic ROLLBACK and COMMIT messages should be sent to only the replicas that need them, instead of being broadcast. This approach can significantly reduce message traffic if broadcasts are expensive, or uncommitted messages are rare. To support this, each replica history manager maintains a map from dependency to the set of replicas that have been sent messages containing the dependency. When a replica receives or generates a ROLLBACK or COMMIT message it forwards the message to all replicas which have been sent the dependency. The dependency can then deleted from the map, because a ROLLBACK or COMMIT message will not need to be forwarded for it again.

8.9.2 Distribution of primary replica functions

Using a primary replica to coordinate a replicated process presents several performance disadvantages. First, relative to secondary replicas the primary replica and the network near it must handle a disproportionately large amount of message traffic. Second, because a WRITE message must be routed to a primary replica rather than a secondary replica that may be more nearby, an uncommitted message sent in reply to the WRITE incurs an unnecessary communications delay.

To resolve these performance problems we eliminate the distinction between primary and secondary replicas. In the resulting "fully" distributed replication algorithm a WRITE is routed to any replica. The replica executes the message and runs its update protocol.

The *un* sequence, which had been coordinated by the primary replica, is created distributively. Essentially, creating the *un* sequence is necessary to prevent W-W errors. In keeping with the spirit of optimistic computation, we propose that the update number sequence be created *optimistically*.

First, we outline the structure of fully distributed process replication:

- The replica *guesses* that the WRITE will sequentially follow the last update the replica knows about. It assigns the WRITE message the next *un*, and does the update protocol. In parallel the replica attempts to commit its *un* guess.

- If only one replica used the *un* then the guess will commit. However, if multiple replicas used the same *un* then but one of them will roll back.

The sequence number protocol creates its own set of dependencies. These dependencies must be propagated, and eventually committed or rolled back. This does not require a separate history manager. Rather, the sequence number generator uses the dependency tracking, commitment and rollback functionality of the optimistic replica history manager: Dependency tracking tracks both sets of dependencies; a point in the computation does not commit until all its dependencies commit; and a point in the computation rolls back if any of its dependencies roll back. The optimistic sequence number generator uses the IMQ and SQ data structures maintained by the optimistic replica history manager.

This illustrates an important general principle of optimistic computation which we also saw in Chapter 6 which discusses replication of Time Warp processes: once a mechanism provides the basic foundation of optimistic computation–dependency tracking, history recording, rollback and commit—other optimistic protocols can readily use the foundation. It pays to think optimistically.

8.9.3 Distributed sequence number creation

This section focuses on the problem of optimistically creating sequence numbers. This algorithm can be composed with the with the process replication mechanism above to make fully distributed replication.

We abstract the problem as follows. Consider a distributed team of clerks serving a set of customers. The team's job is to give customers a dense sequence of integers. (This abstraction maps simply to the replication problem. Each clerk in the team corresponds to one replica of a replicated process.)

The customers are impatient, so they demand that a clerk to respond immediately to a request for a sequence number, even though another clerk, who is far away, may use the same number. Thus, the clerks must *guess* sequence numbers.

In exchange for their impatience, the customers are prepared to undo any work dependent on a guessed sequence number. The customer expects to eventually be told by the clerk whether the number guessed was correct or incorrect. If the guess was incorrect then the clerk will guess another number, eventually obtaining a correct guess.

Essentially, the team of clerks are a replicated process that stores a single integer and provides the service "get value and increment". Process replication cannot be used to implement this replicated process, because it will be used to implement process replication.

Customers interact with the clerks via four messages. (See Figure 8.9.)

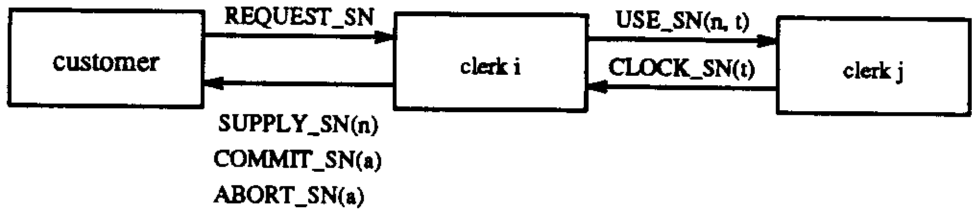

Figure 8.9: Sequence number protocol messages

- A customer sends a REQUEST_SN message to a clerk to get a sequence number.
- SUPPLY_SN(n) is sent by a clerk to give a customer sequence number n.
- A clerk sends COMMIT_SN(a) to tell a customer that the sequence number guess a committed.
- A clerk sends ROLLBACK_SN(a) to tell a customer that the sequence number guess a was uncorrect.

Each clerk maintains a clock which must keep logical time [Lam78] and hopefully keeps good real time. We discussed this kind of clock in Chapter 7. We assume every time value is unique, which can be implemented by using a process's unique identifier as a fractional part of the time values it generates.

When a clerk receives a REQUEST_SN message, it timestamps the request with its local clock value. The committed sequence of sequence numbers given out by the clerks will be in timestamp order. Clerks keep a queue of all sequence number uses, called the sequence number queue (SNQ). The fields of an SNQ entry are

```
sn_use :  record (
  timestamp :  real_time  -- real-time when the request arrives
  sequence_number :  integer -- the sequence number
  request_or_use :  boolean  -- was request received here?
  customer :  process_name   -- name of the customer
  unique_id :  real          -- identifier for this SN guess
  dependencies :  dependencies -- other dependencies of the request
);
```

Figure 8.10: SNQ entry

listed in Figure8.10. The SNQ is ordered by request timestamp. A clerk responds to a REQUEST_SN by sending next sequence number, one greater than the largest one in the SNQ, to the customer in a SUPPLY_SN(n) message.

Because the clerks keep logical time, the local time is later than the reported time of any other clerk. Therefore, the timestamp of the new request is larger than the timestamp of any other request in the SNQ. The request is recorded by appending an entry to the end of the SNQ.

A clerk informs other clerks that it used a sequence number by sending them a USE_SN(n, t) message, which indicates the sequence number and its timestamp, as shown in Figure 8.11. When a clerk receives a USE_SN(n, t) message it inserts the message in timestamp order in its SNQ. Hopefully, the message will be inserted after all requests. Otherwise, the clerk rolls back and recalculates the sequence numbers of the remaining SNQ entries. If a request in the SNQ obtains a new sequence number then the clerk does a ROLLBACK for the request's current unique id, guesses the new sequence number, creates a new unique id for the new guess, and sends the customer a new SUPPLY_SN(n) message with the new sequence number.

When a clerk receives a ROLLBACK unique id A message, it deletes all messages containing A from the SNQ and rolls back as when it obtains a USE_SN(n, t) message.

A clerk responds to a USE_SN(n, t) message by sending its new local time in a

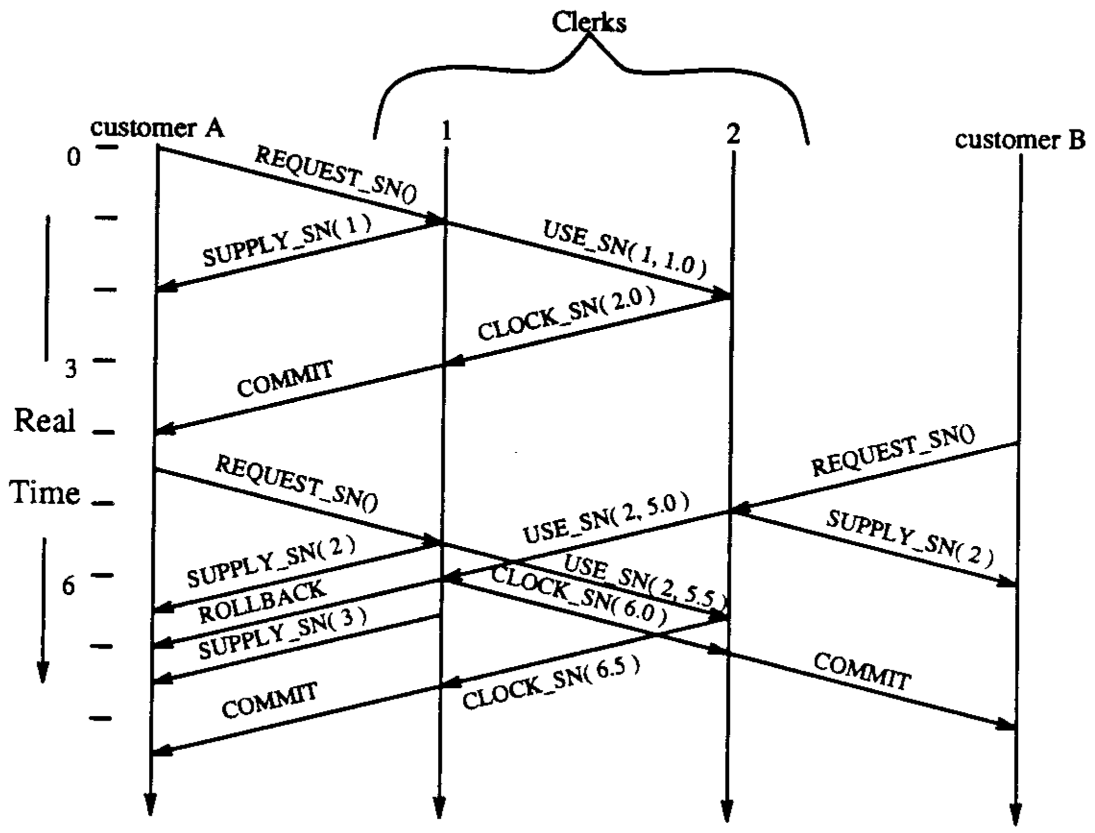

Figure 8.11: A sequence number protocol execution history

CLOCK_SN(t) message to the clerk that used the sequence number.

This sequence number algorithm must be viewed as part of a larger optimistic computation, so dependencies are tracked, committed and rolled back. A clerk creates a unique id for each sequence number guess. The unique id is recorded in the request's entry in the SNQ. The SUPPLY_SN(n) message and guesses with larger timestamps depend on this guess.

How is a sequence number guess committed? We define the *global time* (GT) (similar to Virtual Time's *Global Virtual Time*, Jefferson [Jef85]) as the earliest time of all the clerks' clocks. To obtain a lower bound on GT each clerk tracks the progress of all clocks. When clerk i receives a message with timestamp t from clerk j, clerk i knows that j's clock is at least at time t. We assume the messages sent between a pair of clerks are delivered in order. When GT passes the timestamp of a sequence number request, the request's unique id commits, because all requests with earlier timestamps must already be in the SNQ.

8.9.4 Pseudocode for distributed sequence number creation

This section lists pseudocode for this protocol.

```
state:
  SNQ : table of sn_use     -- queue of sequence number use
  clerk_id :  integer        -- identifier of this clerk
  clocks :  table of clock  -- array of clerks's times
------------------------------
-- customer requests a sequence number
requestSN(customer, dependencies)
  timestamp the request with current clock
  create unique id for request
  s := last sequence number used in SNQ
  increment s
  request.sequence_number := s
  insert request at end of SNQ
  send supplySN( s ) back to customer
  send useSN( s, timestamp ) to each other clerk
------------------------------
```

```
-- clerk indicates it used sequence number at timestamp
useSN(clerk, timestamp)
  insert useSN in SNQ in timestamp order
  call clock(clerk, timestamp)
  for M in SNQ following useSN do
    increment M.sequence_number
    if ( M.request_or_use ) then
      -- new guess
      rollback( M.unique_id )
      M.unique_id := create_unique_id()
      send supplySN(M.sequence_number) to M.customer
    end if
  end for
------------------------------
-- clerk's clock is at least time
clockSN(clerk, time)
  clocks[clerk] := max( clocks[clerk], time+1 )
  -- commit as far as clocks allow
  global_time := min( clocks )
  for M in SNQ do
    if (M.request_or_use and M.timestamp <= global_time ) then
      commit( M.unique_id )
    end if
  end for
------------------------------
-- rollback all computation dependent on unique_id
rollback(unique_id)
  remove entries in SNQ with unique_id in dependencies
  -- recalculate sequence numbers
  for M in SNQ do
    if (M get a new sequence_number and M.request_or_use ) then
      -- new guess
      rollback( M.unique_id)
      M.unique_id := create_unique_id()
      send supplySN(M.sequence_number) to M.customer
    end if
  end for
```

8.9.5 Correctness

The protocol clerks must eventually satisfy every REQUEST_SN message, that is, it must not deadlock or starve a request.

Assuming all other dependencies eventually commit or rollback, a sequence number request of a clerk will eventually commit. The request is assigned a permanent timestamp. Because sequence numbers are allocated in timestamp order, once all other requests with earlier timestamps are known, the request commits.

Messages are reliably delivered, so eventually each other clerk will reply to a USE_SN(n, t) message with a clock message that says its clock has past the time of the request's timestamp. Messages are delivered in order, so once another clerks' clock is known to have passed some time, no other requests with timestamps less than the time will arrive. Thus, the request will commit.

8.9.6 Complexity

Consider a system of c clerks. We calculate the message count bounds for generating sequence numbers, under the assumption that the rest of the replication system does not introduce rollbacks, which could undo an arbitrary amount of this algorithm and increase the message count per committed sequence number.

The message count lower bound for generating one sequence number is $2c$. One request generates $c - 1$ USE_SN(n, t) messages and each USE_SN(n, t) message produces one CLOCK_SN(t) message. The clerk sends a SUPPLY_SN(n) message a COMMIT_SN(a) message to the customer.

The message count upper bound for generating one sequence number is $O(c)$. Consider a set of n concurrent requests. Each request generates $c - 1$ USE_SN(n, t) messages plus $c - n$ CLOCK_SN(t) messages. Possibly all of the clerks will send their customer a ROLLBACK_SN(a) message, 2 SUPPLY_SN(n) messages, and a COMMIT_SN(a) message. Thus, the total message count will be $2c + 2$.

8.10 Performance characteristics

What will be the 'typical' response times of a message sent to a replicated process? How long will it take to commit? These questions cannot be answered with hard bounds, because every message execution might have to wait to commit. However, we can give some insight into performance by considering the most likely scenarios.

We assume a well 'tuned' replication system in which WRITE messages are infrequently received by replicated processes; in all situations they should be fewer than half the messages, in most situations they should be fewer than one-tenth the messages. Since WRITE messages are infrequently sent to replicated processes, very few replicas will depend on uncommitted dependencies. Therefore, most messages—READ messages executed by a replicated process and most messages of any kind executed by an unreplicated process—will be committed before they get executed. Most output will be generated instantaneously. Most replicas will store only the current state in the SQ.

A process sending a WRITE message to a replicated process will receive a reply in the time it takes to execute the message (plus other queuing delays, of course). The reply will carry the dependency for committing the WRITE message; a COMMIT message for this dependency will arrive later, delayed by the maximum round-trip delay between the primary replica and the other replicas of the process. Figure 8.3 showed this execution example.

We believe that the primary cost of replication will be checkpointing a second copy of a replica's state before executing a WRITE. If possible, this checkpoint should be taken earlier during a period when the replica is idle, so the additional cost does not be on the critical path.

Statistically, we believe rollbacks will rarely occur, so they should not be an important consideration in the system's performance. Our future work will simulate optimistic replication to evaluate its performance quantitatively.

CHAPTER 9

Transparent Read and Write Message Classification

A highly desirable property of any operating system change is that all programs that ran on the original operating system should run on the changed operating system. The change may require that programs be recompiled, but the source code should not need modification. A change that satisfies this property is called *transparent*, because an application program cannot 'see' the change. A change that is not transparent is undesirable because it forces programmers to 'fix' programs 'broken' by the change. We have designed process replication to be a transparent improvement to existing operating systems. An important requirement of this design is that the operating system classify each user message execution as a READ or a WRITE message execution, without help from the application.

By definition, executing a WRITE message makes a net change to a replica's state, whereas executing a READ message makes no net change. In brief, to classify a message the operating system observes its execution. If the replica's state changes then the message is a WRITE, otherwise it is a READ. This chapter presents several classification mechanisms and compares their performance characteristics.

The following criteria should be used to evaluate the quality of a state modification detection mechanism:

1. No WRITE message is classified as a READ message. Otherwise the mechanism incorrectly classifies a WRITE message and the replicas of a process may become inconsistent.

2. Few READ messages are classified as WRITE messages. Incorrectly classifying a READ message generates unnecessary CONSISTENCY messages, thereby slowing down the computation and counteracting the purpose of replication.

3. The state modification detection should not take 'too much' computing time or consume 'too much' memory. In the final analysis the detection method must use few enough resources so that process replication can speed up the application, which can only be determined by actual experience.

9.1 Other uses of state change detection

Determining whether a process's state changes is an old problem. A similar issue arises in virtual memory systems. For example, in a demand paged system the primary copies of a running process's data pages are stored on a secondary storage paging device. A subset of these pages also reside in memory. When the memory that a page occupies (called a page frame) needs to be used for another purpose the page is removed from memory. If the page has been modified while occupying the page frame then its contents must replace the primary copy on the paging device [Den70]. Otherwise, the page can simply be thrown away, without incurring the expense of writing it to the paging device. Determining whether a page has been modified is an important enough performance optimization that virtual memory hardware almost always associates a *modification bit* with each page in the page table. The hardware sets the bit whenever an instruction writes a memory location in the page. When the virtual memory software loads a page into memory it clears the modification bit; when the page is discarded the modification bit indicates whether it must be written to disk. One of the message classification mechanisms we propose uses virtual memory modification bits.

This virtual memory example is one instance of a general principle; many systems use knowledge that some data has not changed to avoid the effort to write a redundant copy of the data. Both optimistic replication mechanisms occasionally save a replica's state. Clearly, a state should be saved only if it differs from the previously saved state. Implementing this optimization requires determining whether executing a message modifies a process's state.

9.2 What is a state?

We begin with an informal definition of a process' *state*. Intuitively, a process's state is the minimum set of data that, together with its future input messages, will completely determine its future behavior. (Recall that we assume deterministic processes.) We must be careful not to take this definition too literally. Some of a process' future behavior may depend on external errors, such as hardware failures, communications deadlocks, or insufficient memory problems. The state cannot include data on external conditions like these.

For example, the state may be a process' entire address space and machine register state. Virtual memory saves this data when it swaps a process out of memory. This set of data certainly determines a process' future behavior.

However, we do not want to include the entire address space in a state because some of it may be inaccessible. For example, stacks and heaps can contain inaccessible memory. A process cannot reliably use data below the bottom of a stack that grows downward or unreferenced heap memory. Including inaccessible memory in the state will harm performance by causing READ messages to be incorrectly classified as WRITE messages. For example, suppose a process receives a message and uses temporary stack space to compose a reply. The contents of the stack may change but the process will still have the same state when the message execution finishes.

Garbage collection of heap storage can also cause spurious WRITE message classification, because it will change the content of heap pages. If possible, it should not be done during a message execution.

9.3 State change detection mechanisms

We now present three mechanisms that can detect state modifications; the first simply compares states, the second relys on virtual memory hardware page modification bits, and the third recompiles a user program so its execution records state modifications in a special state variable.

Perhaps the most obvious method to detect a state modification copies the process' state before it executes the message, and then compares the copy with the state after the execution completes. This method uses time and space linear in the size of the state. It might take too long, so we present two other methods to detect state modification.

9.3.1 Hardware assisted state change detection

We now proceed in two phases; first we describe the ideal hardware for state change detection, and then we describe how it can be emulated using current virtual memory hardware. Ideally, the hardware would provide a `StateChange` bit for each process (stored in the process control block) and an `In_State` bit for each page (stored in the page table). When the state change detection code creates a process it sets the `In_State` bit for each page that stores StateVariables. Before a process begins executing a message the operating system clears the `StateChange` bit. When a message execution completes the value of `StateChange` indicates whether the message was a WRITE.

The hardware works as follows. While a user process runs, any instruction that writes into a page with `In_State` true makes the process' `StateChange` bit true. The `StateChange` bit can be set in parallel with the instruction execution, in the same way that a page modification bit is set in parallel, so it should not slow down a process' execution.

This hardware does not exist, but state change detection can be implemented on existing hardware that provides virtual memory page modification bits. (In fact it can be implemented so easily that special purpose hardware should not be built.) The implementation involves two pieces of system software—the state change detection code and the virtual memory code.

In our design the hardware `Page_Mod` bit will be used both to detect page modification for virtual memory and to detect state changes. The bit is reset whenever a new page is loaded or a new message execution begins. But before it is reset the value is saved.

The system maintains a table with two new boolean variables for each page:

- State_Mod indicates, along with the hardware, whether a page has been modified during a message execution.
- Saved_Page_Mod indicates, along with the hardware, whether a page has been modified while resident.

These variables should be viewed as additional fields in the page table, which are stored elsewhere in memory because the page table contents are determined by the machine architecture.

These booleans are manipulated when pages are loaded and unloaded and message executions start and finish. Virtual memory software is changed to do the following. Before a page is removed from memory (unloaded) the page's State_Mod bit is set to true if its hardware Page_Mod is set. The page has been modified while resident if its hardware Page_Mod bit OR'ed with its Saved_Page_Mod is true. As in existing virtual memory code, when a page is loaded its hardware Page_Mod is cleared.

Before a message execution starts all the receiving process's State_Mod bits are cleared. For each page in the process's state, if the hardware Page_Mod bit is true then Saved_Page_Mod is set true. Then all Page_Mod bits are cleared. When a message execution finishes the receiving process's state has been modified if and only if any page in the state has State_Mod or Page_Mod set.

Pseudocode for state change detection is presented in Figure 9.1. It is implemented in the process replication receive() system call and some small revisions to the virtual memory software.

This state change detection mechanism costs little. The time and storage costs to detect a state change are linear in the number of pages in the replica's state. A constant number of operations are added to each virtual memory page swap.

```
----------------------------------------------------------
Incorporate into receive():
-- ++++++++++++++++
  -- at finish of receive():
  -- Did the message execution just completed change the state?
  State_Change := false
  for page in state do
  begin
    State_Change := State_Change or
      Page_Mod[page] or State_Mod[page]
  end for
  -- State_Change indicates whether the state changed
-- ++++++++++++++++
  -- at start of receive():
  -- Prepare for next message execution
  for page in state do
  begin
    Saved_Page_Mod := Saved_Page_Mod or Page_Mod
    Page_Mod := false
    State_Mod := false
  end for
----------------------------------------------------------

Incorporate in the virtual memory code:
-- ++++++++++++++++
  -- when unloading a page:
  if Page_Mod
    then State_Mod := true end if
-- ++++++++++++++++
  -- replace line in virtual memory code which reads Page_Mod
  -- to determine whether a page is dirty:
  page_dirty := Saved_Page_Mod or Page_Mod

-- ++++++++++++++++
  -- add to end of virtual memory code which loads a page:
  Saved_Page_Mod := false
```

Figure 9.1: Hardware assisted state change detection code fragments

If two conditions exist then using hardware support is the best way to detect a state change. First, the machine must have hardware page modification support for virtual memory. And second, the virtual memory code must be available for modification. If these conditions do not hold then another technique to detect state changes may be implemented for programs written in languages with appropriate semantics.

9.3.2 Compilation to code that automatically detects state changes

Applications written in some programming languages could be compiled by an enhanced compiler that produces code that automatically detects state changes. This is possible, of course, only when source code for the application program is accessible, and the compiler is accessible and can be modified.

In addition to generating object code the enhanced compiler defines a run-time boolean variable called `State_Change` which records whether executing a message modifies a process' state. Each straight line section of code containing operations that modify the state is followed by an operation that sets `State_Change`. Compiled user operations never access `State_Change`.

At run-time, the operating system sets `State_Change` to false just before beginning a message execution. If `State_Change` is true after the replica executes the message then it changed the process' state.

The challenge is to find the state altering operations statically. Clearly, if `x` is a state variable, then adding `10` to `x` alters the state. Modifying a temporary variable inside the scope of the block containing the `receive()` call does not modify the state. Pointers and aliasing make finding state altering operations more difficult. Modifying the variable pointed to by p may not change the state, but this cannot be determined at compile time. In this case the compiler has to be pessimistic, and assume that the state is modified. Languages that do not have pointers or aliasing, like Hermes [SBG$^+$91], eliminate this problem.

How will this state change detection mechanism perform? In the worst case code produced by the enhanced compiler runs only twice as slow as normal code. Each operation that sets `State_Change` accompanies at least one operation that modifies a state variable. Thus, the number of operations doubles at worst. Since code ordinarily includes many instructions per straight line segment it would be reasonable to expect execution to slow by much less than a factor of 2.

The computational cost of this method also compares favorably with that of brute force state comparison. Assume that comparing one StateVariable takes one instruction. Only if the state is completely modified more than once will this method add more steps to a message execution than a comparison of complete states. That is, the number of set `State_Change` true instructions during the execution of a message is less than the number of instructions used to compare states. It is unlikely that a program would change its state so extensively, because only the last value of a variable remains.

9.4 Comparison of state detection mechanisms

We now compare the complexity of these state modification detection mechanisms; the brute force state comparison, the hardware assisted method, and the enhanced compiler method. The user must decide at compilation time whether to use the enhanced compiler method. If the process is written in a language that can determine StateVariable accesses at compile time then a recompilation based method would be best since it bounds the mechanism's time to less than 50 percent of the message execution time and does not depend on the availability of page modification bits. Among the other two methods, the hardware assisted method is preferable to brute force state comparison in both time and space by a large constant factor—the page size.

So far we have discussed how to detect state modification of processes that adhere to the rigid computing model presented in Chapter 3. The state of a process described by arbitrarily structured code is considerably more complex. The operating system observes the process' state when it calls `receive`. If a process calls `receive` at multiple places in its code then its state includes the process's program counter, all variables in its scope when it makes the call, and its run-time stack. Determining that a process with this structure receives a READ message is nearly impossible, since all of the process' address space must be considered as its state, and any execution will change some of the contents of the address space. Thus, only processes that adhere to the computing model above are appropriate to test for state modification.

9.5 Replication with message classification

To have real transparency the process replication mechanism must use state change detection to distinguish between READ and WRITE messages. Any of the message classification mechanisms of the previous sections can be integrated with a replication mechanism.

In a replication system with transparent message classification the kind of a message whose execution has not completed is unknown, because the message's kind depends on the state of its receiving replica at the completion of its execution. Therefore, we add another member to the set of message kinds, UNKNOWN, which is the kind of an original message whose execution has not finished. When the execution of an UNKNOWN message completes, its kind is changed to READ or WRITE.

We now discuss integration of transparent message classification with our three replication mechanisms: pessimistic, RTW optimistic, and dependency-tracked optimistic. It is easier to integrate message classification into the optimistic replication algorithms because they can easily run the update protocol after executing an UNKNOWN message to determine its kind, so we discuss them first.

Transparent message classification delays the replication mechanism actions that depend on knowing whether a message is a READ or a WRITE. When a

UNKNOWN message is executed the replication mechanism must be prepared—in case the message is determined to be a WRITE—to appropriately synchronize the WRITE message's execution.

9.5.1 Message classification integrated into RTW

When an UNKNOWN message m arrives at a replica R, RTW assigns m an RVT as described in Section 7.4.

If message classification determines that m's kind is READ then RTW does nothing when m's execution completes. If message classification determines that m's kind is WRITE then RTW creates and transmits CONSISTENCY message copies of m. This mechanism has the same logic as the mechanism described in Section 7.4. However, it may perform worse, because the CONSISTENCY messages are sent later than they would have been had m been labeled by its sender, so they are more likely to cause a rollback.

If m is determined to be a WRITE then sending its CONSISTENCY message copies is 'part of' m's execution. In particular, GVT must not be allowed to progress past $m.RVT$ until the CONSISTENCY messages have all been executed.

9.5.1.1 Rollback

Although m's kind may be determined to be READ after one execution, its kind may not stay READ. If R receives a straggler CONSISTENCY message with an RVT less than $m.RVT$, then RTW will roll back R to before m's execution. If m's kind had been determined to be READ then the rollback must reset m's kind to UNKNOWN. This is necessary because the state of R when m is re-executed may differ from the state of R when m was previously executed. The state can be different because the straggler may have altered it. As a result of the different state, m's re-execution may behave differently and modify R's state.[1]

[1]The kind of a WRITE could also be reset to UNKNOWN, but doing so would accomplish nothing unless the CONSISTENCY messages sent to other replicas were annihilated, which might be more

Incorporating message classification into RTW requires these changes to RTW. In the next section we discuss incorporation of message classification into dependency-tracked optimistic replication.

9.5.2 Message classification integrated into dependency tracked replication

Incorporating message classification into dependency-tracked replication is similar to incorporating message classification into RTW. When the execution of an UNKNOWN message completes, the message's kind might be determined to be WRITE, so the mechanism must be prepared for this possibility.

Suppose an UNKNOWN message m is executed by replica P_i. In case m becomes a WRITE the replication mechanism assigns m update number Pj. Messages sent during m's execution depend on Pj. If m is determined to be a WRITE then the replication mechanism uses Pj as m's update number as if m had been classified as a WRITE by its sender. If m is determined to be a READ then Pj is 'thrown away'—dependency on Pj stops being an obstacle to commitment.

For example, Figure 9.2 shows the execution of a message m with no dependencies by a replica P_i with no dependencies. Update number Pj is assigned to m, so message $m2$ sent during m's execution carries a dependency on Pj. If m is determined to be a READ, as on the left of Figure 9.2, then update number Pj commits immediately, so COMMIT(Pj) is sent to the receiver of $m2$. If m is determined to be a WRITE, as on the right of Figure 9.2, then the replication mechanism begins the update protocol immediately, sending CONSISTENCY(i) copies of m to the other replicas of P.

The example in Figure 9.2 is simplified, because neither m nor P_i have uncommitted dependencies when m is executed. Actually they both may have dependencies. Suppose that m carries dependencies $m.d$ and P_i's state has dependencies $s.d$ when P_i executes m. If m is determined to be a READ, then update num-

work than leaving m classified as a WRITE.

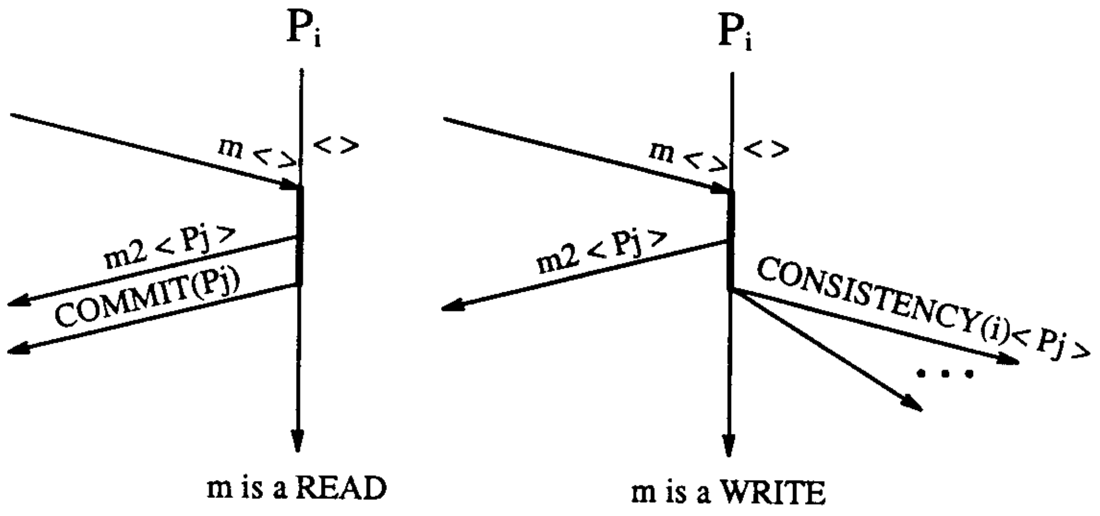

Figure 9.2: Dependency tracked replication with message classification

ber Pj commits automatically if $m.d$ and $s.d$ commit. If m is determined to be a WRITE then the standard dependency-tracked update protocol determines whether Pj commits.

Like RTW with message classification, when dependency-tracked replication with message classification rolls back a replica it may need to undo earlier READ classifications. In particular, if a READ message can be re-executed by a replica in a different state (than the previous time the message was executed) then the message must be reclassified as an UNKNOWN message

9.5.3 Message classification integrated into pessimistic replication

Incorporating message classification into pessimistic replication is extremely difficult because the replication mechanism normally does the update protocol for a WRITE *before* the message is executed. However, with message classification the update protocol cannot be done until the message's execution determines that it is a WRITE.

Therefore, the pessimistic replication mechanism must do a 'test execution' of an UNKNOWN message to determine whether the message is a READ or a WRITE as shown in Figure 9.3. Before doing the 'test execution' the mechanism saves the replica's state. During the 'test execution' messages output by the replica are buffered.

If the message is determined to be a READ, then the buffered output can be sent.

If the message is determined to be a WRITE, then the buffered output must be discarded and the replica's state is restored to the checkpoint saved before the 'test execution'. Then the update protocol is done for the message, and the message is executed again for its 'real execution'.

9.6 Conclusions

An important advantage of transparent process replication is that by using message classification the kinds of messages can be determined at run-time by the replication mechanism. Incorporating message classification into the replication mechanisms is much easier for the optimistic mechanisms than for the pessimistic mechanisms.

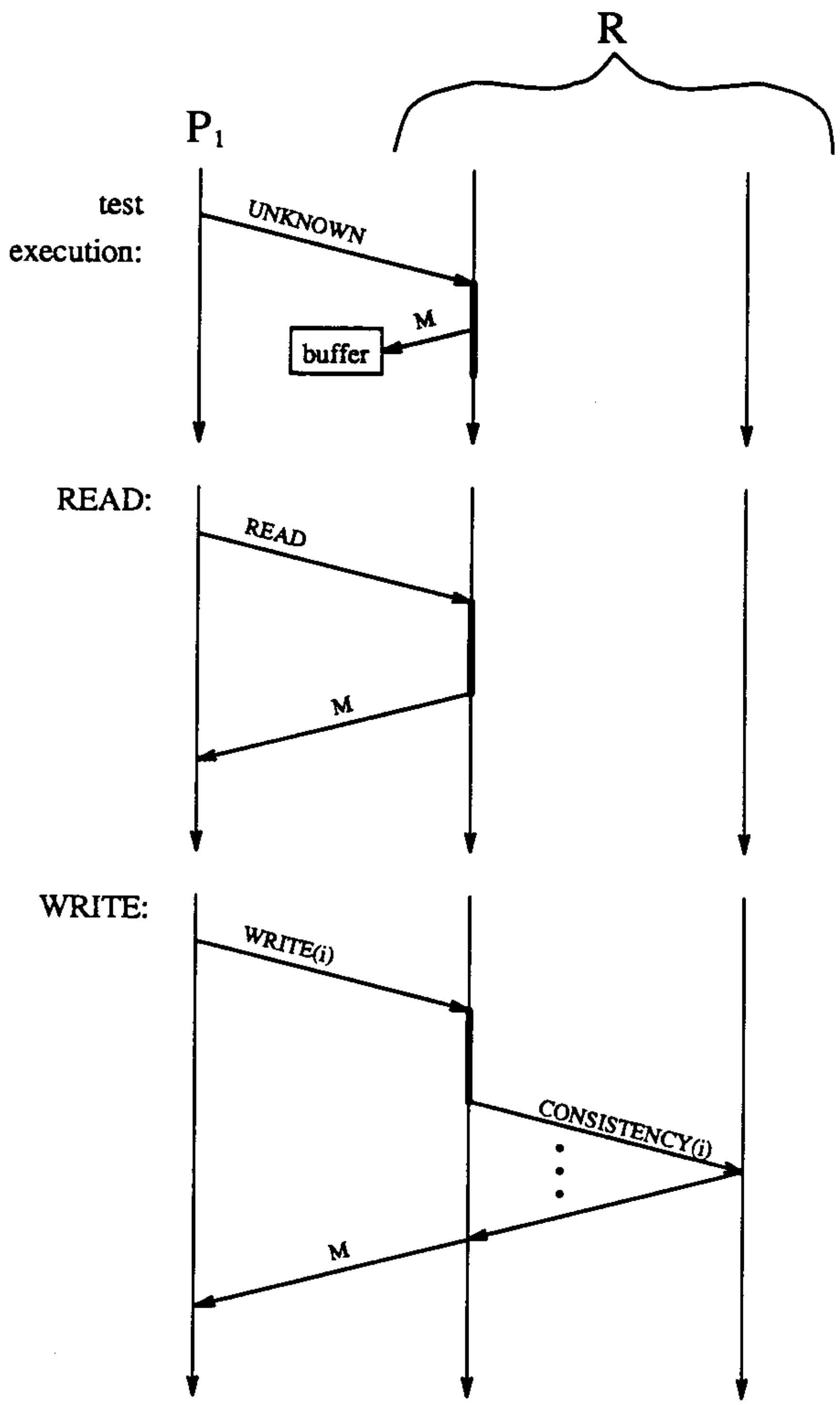

Figure 9.3: Pessimistic replication with message classification

CHAPTER 10

Performance of Process Replication

In this chapter we consider the performance of process replication. First, we consider the question "What is the maximum speedup replication can achieve?" by simulating the performance of two small applications with their bottleneck processes replicated. Our simulation results show that replicating a bottleneck process can substantially speed up a distributed computation, provided that only a small fraction of the messages the process receives are WRITE messages. Next, we qualitatively compare the performance of pessimistic replication with the performance of optimistic replication. Last, we consider the tradeoffs between throughput and response time for our two optimistic replication mechanisms: RTW, and dependency-tracked replication.

10.1 Speedup available from replication

To estimate the maximum speedup achievable from process replication we have developed two simple simulation programs. We recognize that performance measures would ideally be obtained by observing an implementation of process replication running on an actual distributed system, but building such a system was beyond the scope of this research. Two simple application programs were modeled:

tandem application Several processes communicated in tandem, like a Unix pipeline of processes. One process was a bottleneck.

server application Multiple client processes accessed a single server process. The server was overloaded by requests from the clients.

The simulations modeled the execution of each of these application on several multiprocessor architectures. We concentrated on throughput performance, so the performance measure we present is the speedup achieved by a program's replicated execution, s, which we define as the ratio of the program's execution time from start to finish without replication to the program's execution time with replication. We assume that enough processors are available to run each replica on its own processor, so a given application used more processors when it was executed with a replicated process than when it was executed without replication.[1]

10.1.1 Tandem application

We considered a tandem application of 3 processes, a first process (F), a bottleneck process (B), and a last process (L), as shown in Figure 10.1. The mean

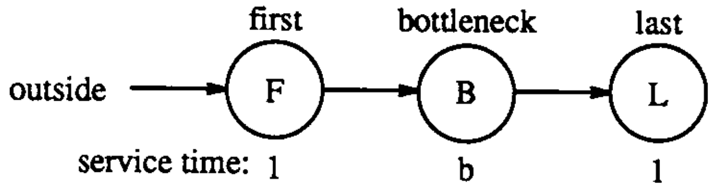

Figure 10.1: Tandem application

execution time of messages at F and L was 1 time unit, while at the bottleneck B it was b, with $b >$ 1time unit. We replicated process B. We assumed that each time a process received a message it sent a message, and that process F received N messages from an outside source, with the messages separated by mean interarrival times of 1 time unit.

A tandem program is appropriate to study for several reasons. First, a larger, arbitrarily shaped, computation may contain a subset of processes connected in tandem. If that subset is a bottleneck in the large computation then speeding it

[1]An alternative speedup measure would compare the times of two executions running on the same hardware. This would not take advantage of the full potential of replication, and might yield a lower speedup. However, it would model situations with fixed resources more fairly.

up can speed up the entire computation. Second, a three process application in which only one process is replicated is easy to study.

First, we consider the mean maximum speedup, s_{max}, that could be achieved by replicating B. Let the following parameters describe the system.

b - the mean service time of a message at B

n - the number of replicas of B

w - the probability that a message sent to B is a WRITE

The speedup is bounded from above in two ways. First, s cannot exceed b because the fastest rate at which processes F and L (which, we assume, are not replicated) can execute messages is 1 per time unit. Second, s cannot exceed the number of replicas of B times the proportion of time they spend executing READ and WRITE messages when they are fully utilized. Thus, we have

$$s_{max} \leq \min(b, n/(wn + 1 - w))$$

We simulated the execution of this tandem application on a pipeline connected multiprocessor. The single link communication delay was 1 time unit. We did not model contention for link access, so the communications delay between any two replicas was the number of links on the shortest path between the processors running them. The message execution times were sampled from exponential distributions with the means discussed above.

The replicas were placed on processors as shown in Figure 10.2. We assumed that the multiprocessor had enough idle processors so that each replica of the bottleneck could run on its own processor. The simulation assumes that process F labels a message sent to process B as a READ or a WRITE. Because the simulation did not implement any replication synchronization mechanism, it provides an upper bound on the speedup that replication can provide in the system we assumed. The destination of a message sent by process F was selected from the replicas of

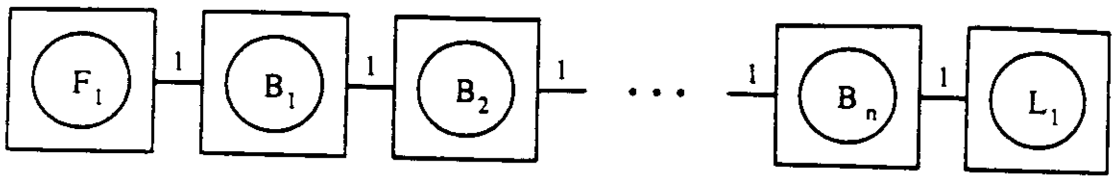

Figure 10.2: Placement of tandem replicas on processors

process B in a round-robin pattern: the i^{th} message sent by F was routed to replica $B_{(i \bmod n)+1}$. When process F sent a WRITE message, a CONSISTENCY message was routed to each other replica of B.

We wrote a discrete-event simulation of this system, and then varied the input parameters b, n and w. Speedup curves are presented in Figure 10.3. In the left graph in Figure 10.3, $b = 2$; in the right graph $b = 8$. The simulations achieve

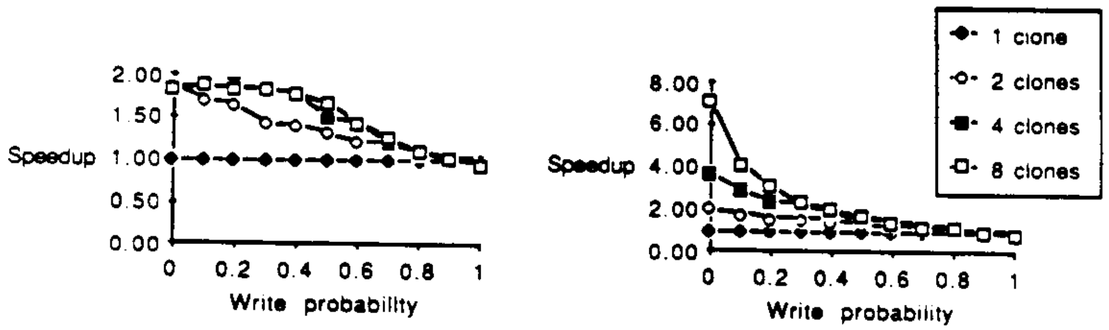

Figure 10.3: Speedup of replicated tandem application

speedups near s_{max} for all ranges of the input parameters.

These results illustrate several points. First, s decreases monotonically with increasing probability that B receives a WRITE message, dropping to $s = 1$ at $w = 1$. Second, as the upper bound model predicts, increasing the number of replicas may not increase s. When b is only 2, increasing n from 4 to 8 generates no more speedup, because 4 replicas are enough to relieve the bottleneck at B.

10.1.2 Client-server application

We simulated the performance of a client-server application composed of a single server process S and a set of client processes. Each client process repeatedly sent a message to S, waited until it received a reply, and then used its cpu to execute the reply. S repeatedly waited for a request from some client, executed the request as soon as it finished executing earlier requests, and sent a reply back to the client. A message sent to S was identified as either a WRITE message or a READ message.

We wrote a DES of a system that executes this application with S as a replicated process. We selected performance parameters that make S a bottleneck process. Several parameters described the system:

c - number of client processes

n - number of replicas of S

r - probability a message sent to S is a READ

Message service times at the processes were determined as follows:

- message execution time at any replica: exponentially distributed, mean = 1 time unit
- communications link delay: constant value, mean = 1 time unit

We simulated the client-server application running with a replicated server on two different communications topologies, a ring network and a fully-connected network. The model ignored contention for communications links. On the ring the communications delay between two replicas equaled the number of hops between their processors. On the fully-connected network the communications delay equaled the single hop delay. We parameterized the performance of the network by varying the ratio of the communications delay over one network link to the mean execution time of a message, which we call d.

We did not implement a process replication mechanism. Instead we simulated the performance of a hybrid dependency-tracked mechanism. The hybrid is similar to a primary replica mechanism in that all WRITE messages are sent to the primary replica. The hybrid is similar to a distributed sequence number selection mechanism in that the primary replica attached an update number to a WRITE message. The primary replica then forwarded the WRITE to the replica of S nearest the client, where the message was executed.

The simulation did not model rollback. This simplification was not, in fact, an inaccuracy in modeling an execution of the client-server application running with a replicated server. In a primary replica dependency-tracked replication with only one replicated process no rollbacks will occur. The model also ignored the costs of dependency tracking and state saving.

The client processes, represented by single replicas, and the replicas of S were symmetrically located on the processors of the multiprocessor. Suppose the system is represented by 2 replicas of S (S_1 and S_2) and 4 client processes (client-A, client-B, client-C and client-D). Figure 10.4 shows the replicas located on the 6 processors of a ring, and Figure 10.5 shows the replicas located on a network of fully connected

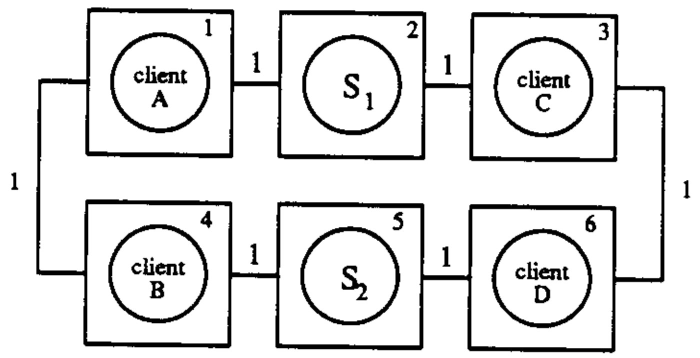

Figure 10.4: Replicas of a server and its clients on a ring

processors.

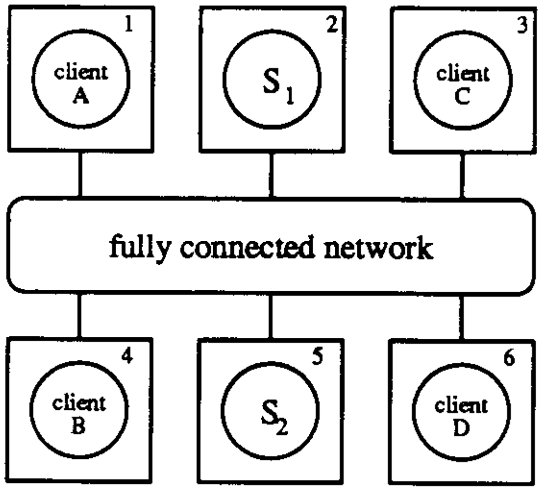

Figure 10.5: Replicas of a server and its clients on a fully connected network

Representative simulation results for a client-server application with 8 clients are shown in Figure 10.6. Each curve gives the speedup as a function of the topology of the underlying network, d, r and n. Simulation of ring network performance is shown in the left column; the right column shows the performance on the fully-connected network. The rows show results from different values of d, 0.01, 1 and 100, as indicated on the right-hand side of Figure 10.6. At $d = 0.01$ communication is essentially instantaneous; at $d = 100$ computation is essentially instantaneous. The simulations were run with $r = 0.0, 0.3, 0.5, 0.7, 0.8, 0.9, 0.95$ and 1.0.

A set of simulations was run with different random number seeds—the vertical bars about a data point indicates one standard deviation about the mean.

What do these simulation results say about the performance of the client-server application with a replicated server? As expected, the speedup, s, increases monotonically with the READ probability r.

When $d = 0.01$, communication is essentially instantaneous so only replication's increased parallelism can improve performance. Thus, in the top row the

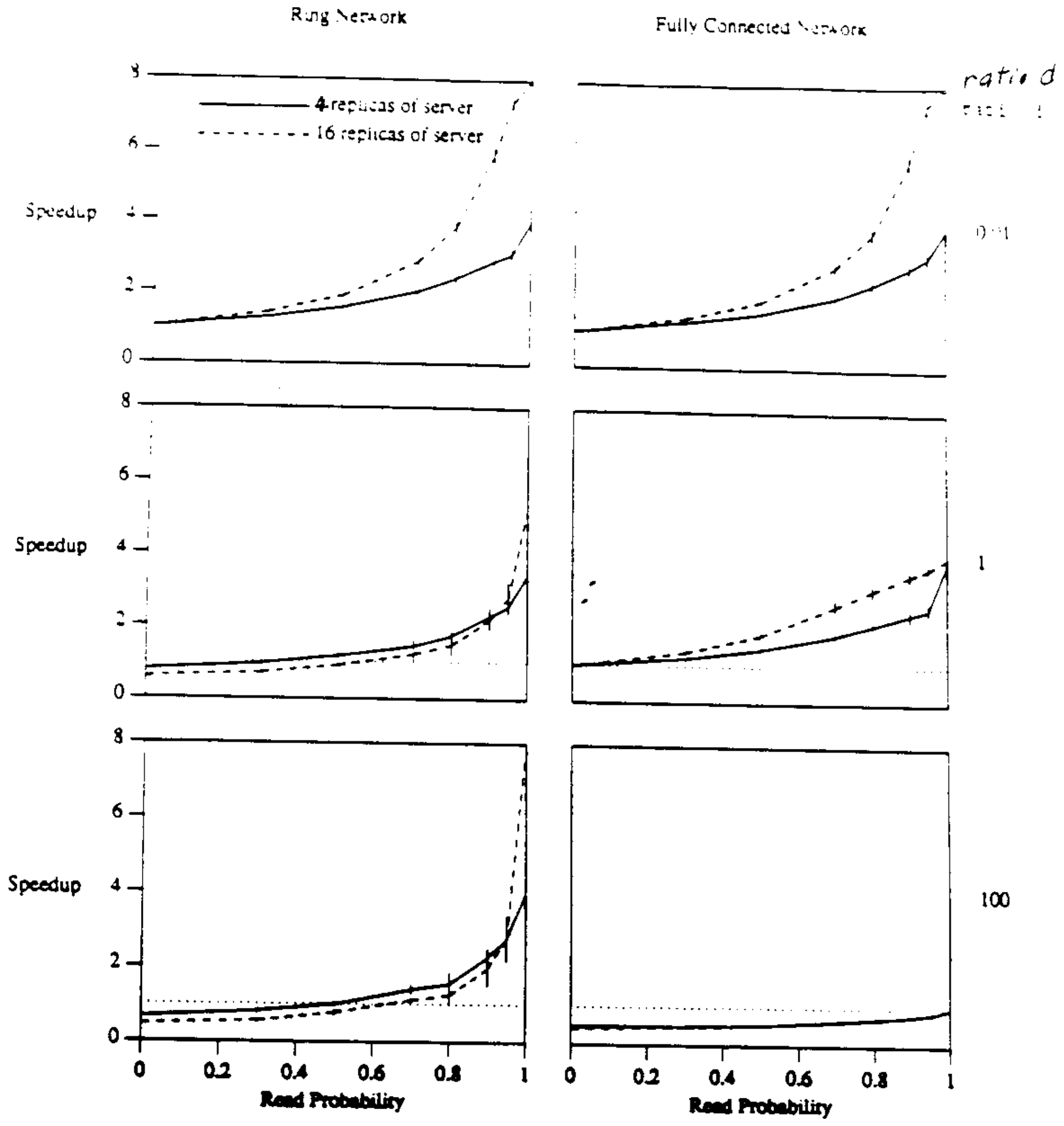

Figure 10.6: Speedup of replicated client-server application

performance of both networks is almost exactly the same.

When $d = 100$, computation is essentially instantaneous so replication can only improve performance by decreasing communications distances. Thus, in the bottom row of data, the ring network shows some speedup, but the full-connected network shows no speedup. Replication can obtain speedup on the ring network by decreasing communications distances; by contrast, on the fully connected network all processes are equally distant whether or not they are replicated, so replication can improve performance only by increased parallelism.

When communication is instantaneous ($d = 0.01$) speedup is independent of the topology of the network. At $r = 1$ the speedup is 4 with 4 replicas of S because the replicas are fully occupied with useful work; 16 replicas only achieves a speedup of 8 because there are only 8 clients to keep the replicas of S busy. At $r = 0$ there is no speedup because the replicas of S are busy executing CONSISTENCY messages.

When computation is essentially instantaneous ($d = 100$) speedup depends greatly on the topology of the network. On a fully connected network with $r < 1$ replication *slows* down the execution because a WRITE message has a three hop path—from client to primary replica to replica nearest client to client—rather than the two hop path—client to only replica to client—without replication. A full implementation of optimistic sequence numbers should eliminate this slowdown. On the ring network this slowdown effect trades off against the speedup obtained by placing replicas of S nearer the clients. The crossover point is about at $r = 0.5$. s at $n = 16$ is slightly worse than s at $n = 4$ because at $n = 16$ the ring has more processors (24 versus 12) so the communications delay around the ring increases and the computation slows down.

When computation and communication are equally expensive we see the effects of both situations in which they are instantaneous. These performance parameters are the most realistic.

These simulation results show that replicating a bottleneck server process of a client-server application can substantially speed up a distributed computation, provided that only a small fraction of the messages received by the server are WRITE messages.

10.2 Comparative performance of replication mechanisms

In this section we compare the performance characteristics of our replication mechanisms. We consider the storage and computation costs of replication because they affect the speedup replication can provide. We also examine the *response time* of an algorithm running on top of a replication mechanism, which is the time the system takes to respond to an input from outside the system.

First we examine the performance of pessimistic replication. Then we compare the performance of our two optimistic replication mechanisms: RTW replication and dependency-tracked replication.

10.2.1 Comparison of pessimistic and optimistic

To execute a WRITE message pessimistic replication must block all replicas for at least one maximum round-trip delay among them. Optimistic replication will let the computation continue in parallel with the protocol to update all secondary replicas.

10.2.2 Comparison of RTW and dependency-tracked process replication

We now compare the performance of RTW, the Time Warp synchronized replication algorithm, with the performance of the dependency-tracked replication mechanism. We assume that the real-time clocks RTW uses are synchronized, and that the dependency-tracked mechanism is fully distributed.

10.2.2.1 Costs

The two optimistic replication mechanisms have some costs that are similar and some costs that differ. Both mechanisms must save enough of a replica's history to support rollback. They must occasionally save a replica's state, and they must save input messages until the messages are no longer needed to support rollback.

We now consider the differences between the two mechanisms. Let P be the number of replicated processes, and R be the maximum number of replicas of a process. To support rollback and commit both RTW and dependency-tracked replication must track the causality between parts of the computation. A piece of causality data is stored in each message and state. To track causality RTW keeps a size $O(1)$ virtual time value in each state and message. Dependency-tracked replication maintains a size $O(P)$ dependency set per message and state.

Therefore, RTW has a smaller cost for causality tracking than dependency tracked replication. However, heuristic optimizations may reduce the dependency-tracked information per message to less than $O(P)$. Suppose the messages communicated between two replicas are received in FIFO order. Then, when S sends message m to R, the only dependencies that m must carry are the *new* dependencies that S has acquired since it last sent a message to R. This can be done extremely efficiently [Gol91]. We speculate that when a distributed computation exhibits "locality of reference" in its communications patterns—most messages go between replicas that frequently communicate—then most messages will carry few new dependencies.

These two mechanisms have very different approaches to *commitment.* Consider the cost when there are not rollbacks. Time Warp runs a GVT [Jef85, Sam84] algorithm periodically. The cost of the algorithm is $O(N)$ messages, where N is the number of processors in the system. Since GVT is calculated periodically, an arbitrary number of update messages can be committed per GVT calculation, so the commitment cost per WRITE can be arbitrarily low. However, the response time grows as the delay between GVT calculations grow.

The message cost of committing an dependency-tracked update which doesn't roll back is $O(R)$, because there are $R-1$ COORDINATE messages sent.

With respect to state saving costs, the two optimistic replication mechanisms have similar costs. To minimize rollback both mechanisms should save state after each WRITE, although deterministic replay enables them to save state less frequently.

10.2.2.2 Rollback costs

Rollbacks are an important cost in optimistic computations. In general, the amount of work undone by a rollback is unbounded. However, we can obtain a little intuition by counting the *initial* rollbacks of the replicas of a process receiving a WRITE, where an initial rollback is the first rollback in a (potential) cascade of rollbacks. Figure 10.7 shows some initial rollbacks in a Time Warp synchronized system.

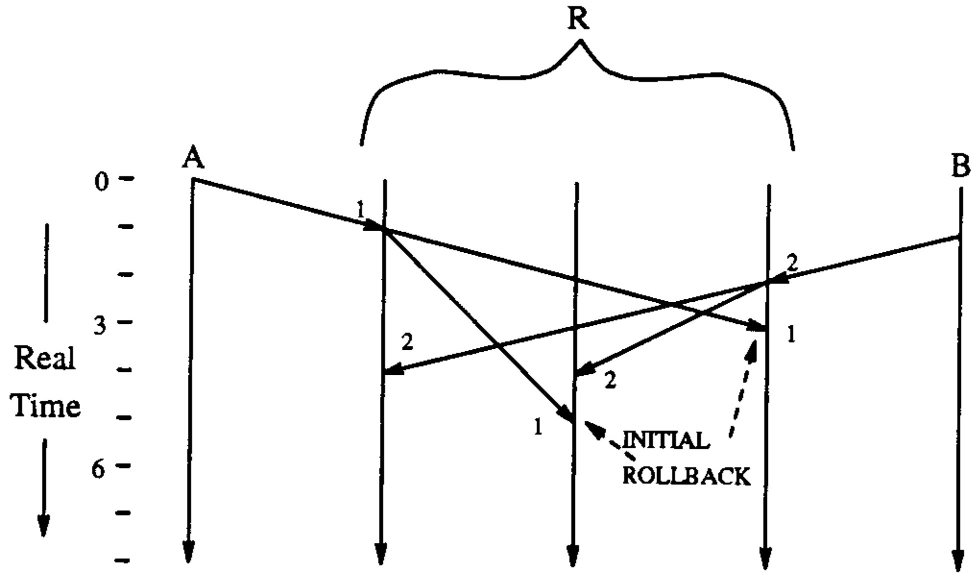

Figure 10.7: Time Warp initial rollbacks

In an execution with only one WRITE there will be no rollbacks, so we consider executions with two WRITE messages.

Suppose the two WRITE messages go to the same process. Because Time Warp based replication uses RVT to determine dependency, a CONSISTENCY message can cause rollbacks that are not logically required by process replication. For example, see Figure 10.8. We say that the READ from process P to replica R_1

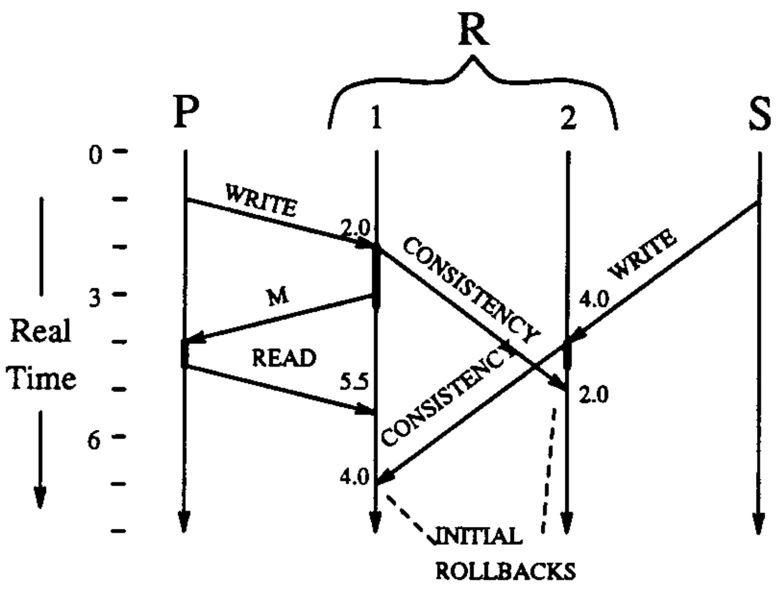

Figure 10.8: False dependency

has a *false dependency* on the CONSISTENCY message from R_2 to R_1, because the CONSISTENCY message rolls back the READ, although the READ message could correctly execute first.

Two WRITE messages can cause as many as $2(R-1)$ initial rollbacks, because every CONSISTENCY message can arrive at a replica out of order. There cannot be any more initial rollbacks because READ and WRITE messages get their RVT values assigned when they arrive and cannot cause a rollback.

A dependency-tracked replication system (with distributed sequence number selection) has a smaller upper bound on initial rollbacks because does not have false dependencies. Figure 10.9 shows the same computation as in Figure 10.8, but synchronized by dependency-tracked replication. Assume that the USE_SN(n,t)

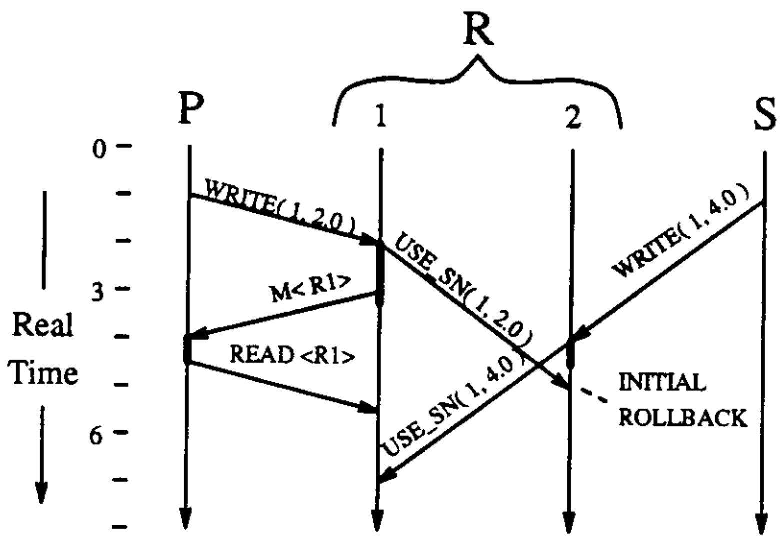

Figure 10.9: Dependency-tracked initial rollbacks

and CONSISTENCY messages are piggy-backed together. The USE_SN(n,t) message from R_1 to R_2 rolls back the WRITE execution at R_2 because the timestamp of sequence number 1 at R_1 is smaller, but the USE_SN(n,t) message from R_2 to R_1 cannot cause a rollback, because nothing can have executed at R_1 that depends on WRITE(1, 2.0). In general, two WRITE messages sent to one process can cause at most $R-1$ initial rollbacks.

Continuing the comparison, we now consider 2 WRITE messages sent to 2 different processes. Suppose each process is replicated R times. A Time Warp based replication can have $2(R-1)$ initial rollbacks, because, as discussed above, each CONSISTENCY message can cause a rollback. In addition, commitment requires a GVT computation.

The dependency-tracked replication may rollback to avoid an C-D error. This will cause an initial rollback for each CONSISTENCY message of the rolled back update, or $R-1$ initial rollbacks. Committing the 2 WRITE messages requires

$3(R-1)$ CONSISTENCY messages and 2 DEPENDENCIES messages, in the worst case—the WRITE that is not rolled back sends one set of CONSISTENCY messages, and the message that is rolled back sends two sets, one that is rolled back and one that is not.[2]

10.2.2.3 Response time performance

The *response time* performance of the two optimistic mechanisms differs greatly. Response time is the time it takes a system to respond to an input—in an optimistic system this is the time interval between sending an input to the system and receiving a *committed* response to the input. We consider *lower* bounds to the response time.

The dependency-tracked mechanism can sometimes respond to an input message in the time it takes to execute the input. When a replica that does not depend on uncommitted dependencies executes a READ input message the execution commits immediately. Also, an *unreplicated* process that does not depend on uncommitted dependencies can commit the execution of a WRITE immediately.

Define $D(Q)$ as the longest communications delay between replicas of process Q. The time to execute a WRITE input message at Q is bounded from below by $2D(Q)$ because each other replica must reply to a CONSISTENCY messages.

Because it tracks dependencies less precisely, Time Warp synchronized replication has a longer response time. The response time is bounded by the time to do a GVT calculation, which is at least $2D$, where D is the longest communications delay between *any* two replicas.

10.3 Conclusions

Our two optimistic replication mechanisms trade off space against time. Time Warp minimizes the storage and communication bandwidth costs by summariz-

[2]The second set of CONSISTENCY messages contains the same data as the first, so the data does not need to be resent.

ing causality information in a single VT value. As a consequence, it risks 'false dependencies' which cause unnecessary rollbacks.

Dependency-tracked replication stores $O(P)$ dependency information per message and state, where P is the number of replicated processes. This additional information reduces the likelihood of a 'false dependency', thereby reducing the number of rollbacks.

Because Time Warp tracks less dependency information, commitment using GVT requires communication with all the replicas in a computation. Dependency-tracked replication can commit updates by just communicating with the replicas of a replicated process, which can take place more locally and more quickly.

On the other hand, since Time Warp tracks less dependency information, it can postpone commitment and commit many updates with a single GVT. Dependency-tracked replication must use control messages (COORDINATE and DEPENDENCIES) to detect C-D errors. The number of control messages is in proportion to the number of updates.

CHAPTER 11

Conclusions

We shine a spotlight on optimistic transparent process replication. We introduce the concept of *process* replication. Small examples and simulations illustrate the ability of process replication to speed up the execution of distributed applications. Because traditional, pessimistic algorithms for managing replicated objects block access to the object when it is being modified, we propose optimistic algorithms for process replication.

We present optimistic replication algorithms in each of the two major styles of optimistic protocols, Time Warp distributed simulation and dependency-tracked optimism. Both algorithms implement process replication, but their approaches are dramatically different. Comparing the performance of these two styles indicates that the dependency-tracked replication algorithm may have faster response time than the Time Warp based algorithm, but at the expense of increased bandwidth and storage for tracking dependencies.

To make process replication *transparent* we explore mechanisms to distinguish READ messages from WRITE messages. These mechanisms are integrated with the process replication algorithms.

11.1 Future work

- To better understand their performance we intend to implement and/or simulate both optimistic process replication mechanisms.
- The load management policy algorithms that determine the number and location of process replicas need to be designed.

- For load management, process replication *dynamics* must be supported. This includes mechanisms to create, kill and move a replica, and routing algorithms to deliver messages to dynamically changing sets of replicas.

Bibliography

[ABB+86] Mike Acetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer Usenix Conference*, July 1986.

[ABG+91] J. S. Auerbach, D. F. Bacon, A. P. Goldberg, G. S. Goldszmidt, M. T. Kennedy, A. R. Lowry, J. R. Russell, W. Silverman, R. E. Strom, D. M. Yellin, and S. A. Yemini. High-level language support for programming reliable distributed systems. Technical Report RC16441, IBM T. J. Watson Research Center, January 1991.

[AD76] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the Second International Conference on Software Engineering*, October 1976.

[Aiz89] J. Aizikowitz. *Designing distributed services using refinement mappings.* PhD thesis, Cornell, 1989. Available as TR 89-1040.

[All83] J. E. Allchin. A suite of robust algorithms for maintaining replicated data using weak consistency conditions. In *Proceedings of the Third Symposium on Reliability in Distributed Software and Database Systems*, October 1983.

[Bac90] David F. Bacon. How to log all filesystem operations (while only writing a few to disk). Technical Report RC, IBM T.J. Watson Research Center, 1990.

[Bal89] Henri E. Bal. *The Shared Data-Object Model as a Paradigm of Programming Distributed Systems.* PhD thesis, Vrije Universiteit te Amsterdam, 1989.

[BBB+90] Robert V. Baron, David Black, Willism Bolosky, Jonathan Chew, Richard P. Draves, David B. Golub, Richard F. Rashid, Jr. Avadis Tevanian, and Michael Wayne Young. Mach kernel interface manual. Technical report, CS Department, CMU, April 1990.

[BBG83] Anita Borg, Jim Baumbach, and Sam Glazer. A message system supporting fault tolerance. In *9th ACM Symposium on Operating Systems Principles*, October 1983.

[BCLU89] D. Baezner, Cleary, Lomow, and Unger. Algorithmic optimizations of simulations on Time Warp. In *Proc. 1989 SCS Multiconference on Distributed Simulation*, pages 73–78, March 1989.

[BDS84] Joshua Bloch, Dean S. Daniels, and Alfred Z. Spector. Weighted voting for directories: A comprehensive study. Technical Report CMU-CS-84-114, CMU, 1984.

[Ber86] Orna Berry. *Performance Evaluation of the Time Warp Distributed Simulation Mechanism.* PhD thesis, University of Southern California, May 1986.

[Bir86] Ken Birman. Low cost management of replicated date in fault-tolerant distributed systems. *ACM Transactions on Computer Systems*, 4(1):54–70, Feb. 1986.

[BJ85] Orna Berry and David Jefferson. Critical path analysis of distributed simulation. *Proc. 1985 SCS Multiconference on Distributed Simulation*, pages 57–60, 1985.

[BS89] David F. Bacon and Robert E. Strom. Implementing the Hermes process model. Technical Report RC 14518, IBM T.J. Watson Research Center, 1989.

[BS91] David F. Bacon and Robert E. Strom. Optimistic parallelization of communicating sequential processes. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, April 1991.

[Cas72] R. G. Casey. Allocation of copies of a file in an information network. *Proc AFIPS SJCC*, 40:617–625, 1972.

[Chu69] Wesley W. Chu. Optimal file allocation in a multiple computer system. *IEEE Transactions on Computers*, c-18(10):885–889, Oct. 1969.

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

[Den70] Peter Denning. Virtual memory. *Computing Surveys*, 2(3), September 1970.

[DS83] Dean Daniels and Alfred Z. Spector. An algorithm for replicated directories. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, August 1983. Montreal, Canada.

[ELP+89] M. Ebling, M. Loreto, M. Presley, Fred Wieland, and David Jefferson. An ant foraging model implemented on the Time Warp operating system. *Proc. 1989 SCS Multiconference on Distributed Simulation*, pages 21–26, 1989.

[FM82] Michael Fischer and Alan Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *Proceedings of the ACM Symposium on Principles of Database Systems*, March 1982. Los Angeles, California.

[FTJJG88] Richard M. Fujimoto, Tsai, J.-J., and G. Gopalakrishnan. Design and performance of special purpose hardware for Time Warp. *Proc. 15th Symp. on Computer Architecture*, 1988.

[Fuj87] Richard M. Fujimoto. Performance measurements of distributed simulation strategies. Technical Report UUCS-87-026a, Department of Computer Science, University of Utah, November 1987.

[Fuj88a] Richard Fujimoto. Time Warp on a shared memory multiprocessor. Technical Report UUCS-88-021, University of Utah, December 1988.

[Fuj88b] Richard M. Fujimoto. Lookahead in parallel discrete event simulation. *International Conference on Parallel Processing*, August 1988.

[Fuj88c] Richard M. Fujimoto. Performance measurements of distributed simulation strategies. *Proc. 1988 SCS Multiconference on Distributed Simulation*, pages 14–20, 1988.

[Fuj89] Richard M. Fujimoto. Time Warp on a shared memory multiprocessor. Technical Report UUCS-88-021a, Computer Science Department, University of Utah, January 1989.

[Gaf85] Anat Gafni. *Space Management and Cancellation Mechanisms for Time Warp*. PhD thesis, University of Southern California, 1985. Also available as TR-85-341.

[Gaf88] Anat Gafni. Rollback mechanisms for optimistic distributed simulation. *Proc. 1988 SCS Multiconference on Distributed Simulation*, 1988.

[GGL+90] Arthur P. Goldberg, Ajei Gopal, Kong Li, Robert E. Strom, and David F. Bacon. Transparent recovery of mach applications. In *First USENIX Mach Workshop*, Burlington, VT, October 1990.

[Gif79] David K Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating Systems Principles*, December 1979.

[GL91] Richard Golding and Darrell D. E. Long. Accessing replicated data in a large-scale distributed system. Technical Report 91-01, Concurrent Systems Laboratory, UC Santa Cruz, 1991.

[GM79] Hector Garcia-Molina. *Performance of Update Algorithms for Replicated Data in Distributed Databases.* PhD thesis, Stanford University, 1979.

[Gol91] Arthur P. Goldberg. Efficient dependency tracking in distributed computations. Technical report, IBM research, 1991. In preparation.

[GP91] Richard G. Guy and Gerald J. Popek. Algorithms for consistency in optimistically replicated file systems. Technical Report CSD-910006, UCLA Computer Science Department, 1991.

[Gun83] Per Gunningberg. *Fault-Tolerance Implemented by Voting Protocols in Distributed Systems.* PhD thesis, Uppsala University, Sweden, 1983.

[HBL+89] Phil Hontalas, Brian Beckman, Mike Di Loreto, Leo Blume, Peter Reiher, K. Sturdevant, L. Warren, J. Wedel, Fred Wieland, and David Jefferson. Performance of the colliding pucks simulation on the Time Warp operating systems (Part 1: Asynchronous behavior & sectoring). *Proc. 1989 SCS Multiconference on Distributed Simulation*, pages 3–7, 1989.

[Her84] Mauricy Herlihy. *Replication methods for abstract data types.* PhD thesis, MIT, 1984.

[Her86] Maurice Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, February 1986.

[Hoa81] C. A. R. Hoare. The emperor's old clothes. *Communications of the ACM*, 24(2):75–83, Feb. 1981.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International Series in Computer Science, 1985.

[JB86] Thomas A. Joseph and Kenneth P. Birman. Low cost management of replicated data in fault-tolerant distributed systems. *ACM Transactions on Computer Systems*, 4(1), February 1986.

[JBH+85] David Jefferson, Brian Beckman, Steve Hughes, Eric Levy, Todd Litwin, John Spagnuolo, Jon Vavrus, Fred Wieland, and Barbara Zimmerman. Implementation of Time Warp on the caltech hypercube. In *Proc. 1985 SCS Multiconference on Distributed Simulation*, January 1985.

[JBW+87] David Jefferson, Brian Beckman, Fred Wieland, Leo Blume, Mike Di Loreto, Phil Hontalas, Pierre Laroche, Kathy Sturdevant, Jack Tupman, Van Warren, John Wedel, Herb Younger, and Stebe Bellenot. Distributed simulation and the Time Warp operating system. In *Proc. 11th ACM Symposium on Operating Systems Principles*, pages 77–93, August 1987. Also available as UCLA Technical Report 870042.

[Jef85] David Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404, July 1985.

[Jef89] David Jefferson. Virtual time II: The cancelback protocol for storage management in Time Warp. In *Principles of Distributed Computing*, August 1989. Quebec City.

[JM84] David R. Jefferson and Ami Motro. The Time Warp mechanism for distributed database concurrency control. Technical Report TR-84-302, University of Southern California, January 1984.

[JS82] David Jefferson and Henry Sowizral. Fast concurrent simulation using the Time Warp mechanism, part I: Local control. Technical Report N-1906-AF, Rand, December 1982.

[JS85] David Jefferson and Henry Sowizral. Fast concurrent simulation using the Time Warp mechanism. *Proc. 1985 SCS Multiconference on Distributed Simulation*, pages 63–69, January 1985.

[Kau87] F. J. Kaudel. A literature survey on distributed discrete event simulation. *Simuletter*, 18(2):11–21, June 1987.

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

[LCWU88] Lomow, Cleary, West, and Unger. A performance study of Time Warp. In *Proc. 1988 SCS Multiconference on Distributed Simulation*, volume 19, pages 50–55, February 1988.

[Lei84] Gerald Leitner. Stylized interprocess communication - a kernel primitive for reliable distributed communication. In *Fourth Symposium on Reliability in Distributed Software and Database Systems*, October 1984.

[LL89a] Y.-B. Lin and E.D. Lazowska. Exploiting lookahead in parallel simulation. Technical Report 89-10-06, Department of Computer Science and Engineering, University of Washington, 1989.

[LL89b] Y.-B. Lin and E.D. Lazowska. The optimal checkpoint interval in Time Warp parallel simulation. Technical Report 89-09-04, Department of Computer Science and Engineering, University of Washington, 1989.

[LL89c] Y.-B. Lin and E.D. Lazowska. A study of Time Warp rollback mechanisms. Technical Report 89-09-07, Department of Computer Science and Engineering, University of Washington, 1989.

[LL90] Y.-B. Lin and E.D. Lazowska. Optimality considerations for "Time Warp" parallel simulation. *Proc. 1990 SCS Multiconference on Distributed Simulation*, 1990.

[LMKQ89] Samuel J. Leffler, Marshall K. McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley Publishing Company, 1989.

[LMS83] Steve Lavenberg, Richard Muntz, and Behrokh Samadi. Performance analysis of a rollback method for distributed simulation. In A.K. Agrawala and S.K. Tripathi, editors, *PERFORMANCE '83*, pages 117–132. North-Holland Publishing Company, 1983.

[Lom88] Greg Lomow. *The Process View of Distributed Simulation.* PhD thesis, University of Calgary, 1988.

[LRG91] Andy Lowry, James R. Russell, and Arthur P. Goldberg. Optimistic failure recovery for very large networks. In *Proceedings of the Symposium on Reliable Distributed Systems*, Pisa, Italy, September 1991.

[LU85] G.A. Lomow and B.W. Unger. Distributed software prototyping and simulation in jade. *Canadian INFOR*, 23(1):69–89, February 1985.

[Mis86] J. Misra. Distributed discrete-event simulation. *Computing Surveys*, 18(1):39–65, March 1986.

[ML77] Howard L. Morgan and K. Dan Levin. Optimal program and data locations in computer networks. *Communications of the Association for Computing Machinery*, 20(5):315–322, May 1977.

[MP88] Luigi Mancini and Guiseppe Pappalardo. Towards a theory of replicated processing. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 175–192. Springer-Verlag Lecture Notes in Computer Science, 331, September 1988.

[Nic89] D.M. Nicol. The cost of conservative synchronization in parallel discrete event simulations. Technical report, Department of Computer Science, College of William and Mary, 1989.

[Par72] David I. Parnas. On the criteria for decomposing systems into models. *CACM*, 15(12):1053–1058, December 1972.

[PEWJ89] M. Presley, M. Ebling, Fred Wieland, and David Jefferson. Benchmarking the Time Warp operating system with a computer network simulation. *Proc. 1989 SCS Multiconference on Distributed Simulation*, pages 8–13, 1989.

[PGPH90] Thomas W. Page, Jr., Richard G. Guy, Gerald J. Popek, and John S. Heidemann. Architecture of the Ficus scalable replicated file system. Technical report, UCLA Computer Science Department, 1990.

[PGPH91] Gerald J. Popek, Richard G. Guy, Thomas W. Page, Jr., and John S. Heidemann. Replication in Ficus distributed file systems. Technical Report CSD-910006, UCLA Computer Science Department, 1991.

[PPR+83] D. Stott Parker, Jr., Gerald J. Popek, Gerald Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, SE-9(3), May 1983.

[PS83] Francis N. Parr and Robert E. Strom. NIL: A high-level language for distributed systems programming. *IBM Systems Journal*, 22(1-2):111–127, 1983.

[RFBJ90] Peter Reiher, Richard M. Fujimoto, S. Bellenot, and David Jefferson. Cancellation strategies in optimistic execution systems. *Proc. 1990 SCS Multiconference on Distributed Simulation*, 1990.

[RM88] D.A. Reed and A. Malony. Parallel discrete event simulation: The chandy-misra approach. *Proc. 1988 SCS Multiconference on Distributed Simulation*, pages 8–13, 1988.

[RMM88] D.A. Reed, A.D. Malony, and B.D. McCredie. Parallel discrete event simulation using shared memory. *IEEE Transactions on Computers*, 14(4):541–553, April 1988.

[RT87] Zuwang Ruan and Walter F. Tichy. Performance analysis of file replication schemes in distributed systems. In *Proceedings of 1987 Sigmetrics Conference*, 1987.

[Sam84] Behrokh Samadi. *Distributed Simulation: Algorithms and Performance Analysis*. PhD thesis, University of California at Los Angeles, 1984.

[SBG+91] Robert E. Strom, David F. Bacon, Arthur Goldberg, Andy Lowry, Daniel Yellin, and Shaula Alexander Yemini. *Hermes: A Language for Distributed Computing.* Prentice Hall, January 1991.

[SBY87] Robert E. Strom, David F. Bacon, and Shaula Alexander Yemini. Volatile logging in n-fault-tolerant distributed systems. Technical Report RC 13373, IBM T.J. Watson Research Center, 1987.

[Sch90] Fred Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *Computing Surveys*, 22(4):299–320, December 1990.

[SY83] Robert E. Strom and Shaula Alexander Yemini. NIL: An integrated language and system for distributed programming. In *SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, June 1983.

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

[SY87] Robert E. Strom and Shaula Alexander Yemini. Synthesizing distributed and parallel programs through optimistic transformations. In Yechiam Yemini, editor, *Current Advances in Distributed Computing and Communications*, pages 234–256. Computer Science Press, Rockville, MD, 1987.

[SYB88] Robert E. Strom, Shaula Alexander Yemini, and David F. Bacon. A recoverable object store. In *Hawaii International Conference on System Sciences*, volume II, pages 215–221, Kailua-Kona, HI, January 1988.

[Tho78] R. H. Thomas. A solution to the concurrency control problem for multiple copy databases. In *Proceedings of the 16th IEEE Computer Society International Conference*, Spring 1978.

[TK88] Pete Tinker and Murry Katz. Parallel execution of sequential Scheme with ParaTran. In *Lisp and Functional Programming Conference*, pages 28–39, 1988.

[UDCB86] Unger, Dewar, Cleary, and Birtwistle. A distributed software prototyping and simulation environment: Jade. In *Proc. 1986 SCS Multiconference on Distributed Simulation*, volume 17, pages 63–71, January 1986.

[WB84] Gene T.J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the Third*

Annual ACM Symposium on Principles of Distributed Computing, August 1984. Vancouver, Canada.

[Wes88] Darrin West. Optimising Time Warp: Lazy rollback and lazy reevaluation. Master's thesis, University of Calgary, January 1988.

[WHF+89] Fred Wieland, L. Hawley, A. Feinberg, M. Loreto, Peter Reiher Leo Blume, Brian Beckman, Phil Hontalas, S. Bellenot, and David Jefferson. Distributed combat simulation and Time Warp: The model and its performance. *Proc. 1989 SCS Multiconference on Distributed Simulation*, pages 14–20, 1989.

[WT89] Linda R. Walmer and Mary R. Thompson. A programmer's guide to the mach system calls. Technical report, CS Department, CMU, December 1989.

[YSB87] Shaula Alexander Yemini, Robert E. Strom, and David F. Bacon. Improving distributed protocols by decoupling recovery from concurrency control. Technical Report RC 13326, IBM T.J. Watson Research Center, 1987.