Techniques for Simplifying the Programming of Distributed Systems

Richard Koo

Ph.D. Thesis

87-858
August 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

TECHNIQUES FOR SIMPLIFYING THE PROGRAMMING OF DISTRIBUTED SYSTEMS

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Richard Kwok Yan Koo

August, 1987

Techniques for Simplifying the
Programming of Distributed Systems

Richard Kwok Yan Koo, Ph.D.
Cornell University 1987

It is difficult to design and verify distributed programs that execute correctly despite transient processor failures, or despite variable and unpredictable processor speeds and message transmission times. In this thesis, we describe a checkpointing/rollback mechanism that allows programmers to write distributed programs with the simplifying assumption that processors do not fail, and then run these programs correctly on systems with transient processor failures. We also describe a translation mechanism that can be used to write programs with the simplifying assumptions that processors execute in synchronized steps and messages take exactly one step to arrive, and then run these programs correctly on systems that violate these assumptions. Both mechanisms are transparent to the programmer, and they can be applied to solve a large class of problems.

Biographical Sketch

Richard Kwok Yan Koo was born in the British colony of Hong Kong on August 26, 1960. Presumably, he began his schooling at the age of three (though he had no such recollection). Notwithstanding such an early start, he did not finish his secondary education until 1978. He entered Colorado College as a freshman in the fall of 1978, and graduated with a Bachelor of Arts degree eight semesters later. During his time at Colorado College, he became convinced that he enjoyed mathematics more than economics or physics. He was also convinced that he would rather go to graduate school than get a job. Thus, in the fall of 1982, he enrolled in the Ph.D. program in computer science at Cornell University. On Valentine's Day 1985, he received a M.S. degree in computer science.

To my parents

Acknowledgements

I am indebted to professor Sam Toueg, my thesis advisor, for his guidance and advice at every step of my thesis research. His insistence on precise and clear exposition will serve me well in the future. I would also like to thank professors John Gilbert and Narahari Prabhu for serving on my thesis committee, and professor Kenneth Birman for providing generous financial support.

Tracing my intellectual roots, I would like to express my appreciation of the mathematics faculty of the Colorado College. In particular, I thank professor Steven Janke for introducing me to the field of computer science, professor Paul Perlmutter for convincing me to switch my major from economics to mathematics, and professors John Watkins and Frederick Tinsley for encouraging me to pursue graduate studies.

Many people in the computer science department of Cornell have helped me in many ways over the years and I would like to thank them. They are: Jacob Aizikowitz, Özalp Babaoğlu, Kenneth Birman, Rance Cleaveland, Ajei Gopal, Jim Hook, Roger Hoover, Tommy Joseph, Anne Neirynck, Fred Schneider, Frank Schmuck, Amitabh Shah, T.K. Srikanth, Patrick Stephenson, Jennifer Widom, and last but certainly not the least, the coterie of Sam Toueg's students consisting of Gil Neiger, Amr El Abbadi, and Micah Beck.

Support of this work was provided by the Defense Advanced Research Projects Agency, Department of Defense, under ARPA order 5378, Contracts MDA903–85–C–0124 and N00140–87–C–8904, and by the National Science Foundation under grants DCR–8412582 and DCR–8601864.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

A distributed system consists of a number of processors connected by a communication network. Each processor has its own storage and computing units, which are not shared with other hosts. Different processors communicate by sending messages to each other via the communication network.

A processor can coordinate its actions with other processors by executing a *distributed* program, or can eschew coordination with other processors by executing a local program. One advantage that distributed programs have over local programs is *concurrency*. By concurrency, we mean that the different parts of a distributed program can be executed simultaneously by different processors.

Researchers have identified various kinds of behaviors that processors can display when they fail. For example, processors can fail by halting [LF82], by omitting to send messages [Had84], by omitting to send and receive messages [PT86], or by performing arbitrary actions [PSL80]. In this thesis, however, we assume that processors fail only by halting, and that processor failures are transient; i.e., processors eventually resume execution.

The storage unit of each processor is divided into two parts: volatile and non-volatile. When a processor fails, the data in its volatile storage are lost forever, whereas the data in its non-volatile storage are retained. A processor *recovers* from a failure by resuming execution from data in its non-volatile storage. Due to the loss of data in volatile storage, the recovery of a processor may create an error in the execution of a distributed program. This problem is illustrated by the following example.

Consider an execution in which processor i sends a message m to grant processor j permission to use resource x. Suppose that i stores this information only in its volatile storage. Moreover, suppose that processor i fails, loses this information, and then recovers. The resulting execution is inconsistent because, while processor i has no information that it has granted j permission to use x, processor j is certain that i has. The inconsistency may lead to incorrect usage of resource x. Consequently, one desirable property of distributed programs is that they execute correctly despite transient processor failures.

The work-load of processors and message traffic in the communication network typically vary over time. As the work-load of a processor increases, the speed at which it executes a program typically decreases. The opposite is true when the work-load decreases. Similarly, as the message traffic in the communication network increases/decreases, the time the network takes to deliver a message from its sender to its receiver typically increases/decreases. Another desirable property of distributed programs is that they should execute correctly even if processor speeds and message transmission times vary over time.

Unfortunately, it is difficult to design and verify programs that execute correctly despite transient processor failures, or despite variable and unpredictable processor speeds and message transmission times. In this thesis, we develop two mechanisms that simplify the design of such programs. Specifically, in Chapter 2, we develop a distributed checkpointing/rollback mechanism that allows a programmer to design and verify distributed programs with the simplifying assumption that processors do not fail. We can then use this checkpointing/rollback mechanism to run these programs correctly on systems in which processors experience transient failures. Then, in Chapters 3 and 4, we develop a translation mechanism for designing and verifying programs with the simplifying assumption that processors execute in synchronized steps and all messages take exactly one step to arrive. We can then use this translation mechanism to run these programs correctly on systems that violate this simplifying assumption; i.e., systems where processors have variable and unpredictable execution speeds and messages transmission times. Both mechanisms are transparent to the program designer, and they can be applied to solve a large class of problems.

Chapter 2

Checkpointing and Rollback-Recovery*

2.1 Introduction

Checkpointing and rollback-recovery are well-known techniques that allow processors to make progress in spite of failures [RLT78]. The failures under consideration are transient problems such as hardware errors and transaction aborts; i.e., those that are unlikely to recur when a processor restarts. With this scheme, a processor takes a checkpoint from time to time by saving its state on stable storage [LS79]. When a failure occurs, the processor rolls back to its most recent checkpoint, assumes the state saved in that checkpoint, and resumes execution.

We first identify consistency problems that arise in applying this technique to a distributed system. We then propose a checkpoint algorithm and a rollback-recovery algorithm to restart the system from a consistent state when failures occur. Our algorithms prevent the well-known "domino effect" as well as livelock problems associated with rollback-recovery. In contrast to previous algorithms, ours are fault-tolerant and involve a minimal set of processors. With our approach, each processor stores at most two checkpoints in stable storage. This storage requirement is shown to be minimal under general assumptions.

The organization of this chapter is as follows: We discuss the notion of consistency in a distributed system in Section 2.2, and describe our system model in Section 2.3. In Section 2.4 we identify the problems to be solved. Sections 2.5 and 2.6 contain the checkpoint and rollback-recovery algorithms respectively. The algorithms are extended for concurrent executions in Section 2.7. In Section 2.8 we consider optimizations. Section 2.9 contains our conclusion.

2.2 Consistent Global States in Distributed Systems

The notion of a consistent global state is central to reasoning about distributed systems. It was considered in [Ran75,Rus77,Pre83] and formalized by Chandy and Lamport [CL85]. In this section, we summarise their ideas.

In a distributed computation, an *event* can be a spontaneous state transition by a processor, or the sending or receipt of a message. Event *a directly happens before* event *b* [Lam78] if and only if

1. *a* and *b* are events in the same processor, and *a* occurs before *b*; or

2. *a* is the sending of a message *m* by a processor and *b* is the receiving of *m* by another processor.

The transitive closure of the directly happens before relation is the *happens before* relation. If event *a* happens before event *b*, *b happens after a*. (We abbreviate happens before, "before" and happens after, "after".)

A *local state* of a processor *p* is defined by *p*'s initial state and the sequence of events that occurred at *p*. A *global state* of a system is a set of local states, one from each processor. The *state of the channels* corresponding to a global state *s* is the set of messages sent but not yet received in *s*. We can depict the occurrences of events over time with a time diagram, in which horizontal lines are time axes of processors, points are events, and arrows represent messages from the sending processor to the receiving processor. In this representation, a global state is a cut dividing the time diagram into two halves. The state of the channels comprises those arrows (messages) that cross the cut. Figure 2.1 is a time diagram for a system of four processors.

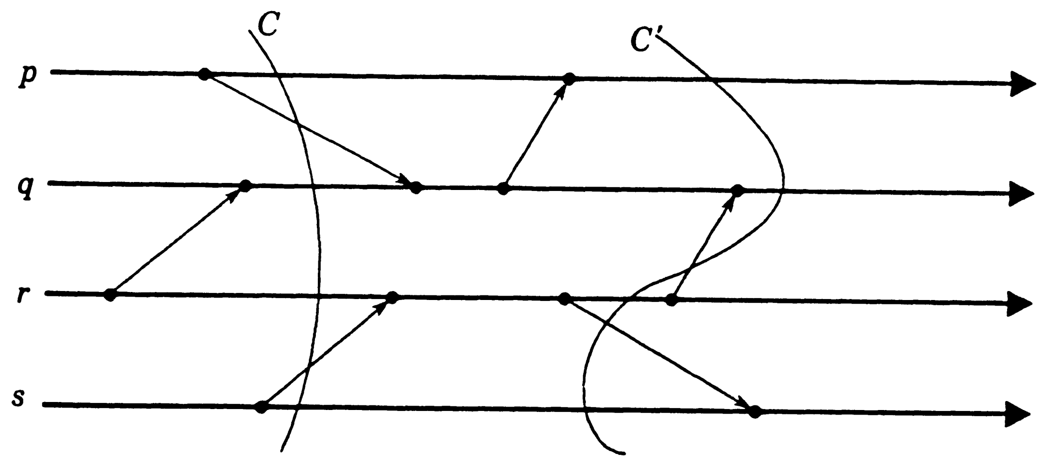

Figure 2.1: Consistent and inconsistent cuts.

Informally, a cut (global state) in the time diagram is *consistent* if no arrow starts on the right hand side and ends on the left hand side of it. This notion of consistency fits the observation that a message cannot be received before it is sent in any temporal frame of reference. For example, the cuts c and c' in Figure 2.1 are consistent and inconsistent cuts, respectively. The state of the channels corresponding to cut c consists of one message from p to q, and another message from s to r. Readers are referred to [CL85] for a formal discussion of consistent global states.

2.3 System Model

The distributed system considered in this chapter has the following characteristics:

1. Processors do not share memory and communicate via messages sent through channels.
2. Channels can lose messages. However, they are made (virtually) lossless and first-in-first-out by some end-to-end transmission protocol

(such as the standard sliding window protocol [Tan81]).

3. Processors can fail by stopping, and whenever a processor fails, all other processors are informed of the failure in finite time.

We assume that the communication network is never partitioned by failures of processors.

We want to develop our algorithms under a weak set of assumptions. In particular, we do not assume that the underlying system is a database transaction system [FGL82,JB86]. This special case admits simpler solutions: the mechanisms that ensure atomicity of transactions can hide inconsistencies introduced by the failure of a processor. Furthermore, we do not assume that processors are deterministic: this simplifying assumption is made in previous results (e.g., [SY85,JB86]).

2.4 Identification of Problems

A *checkpoint* is a saved local state of a processor. A set of checkpoints, one per processor in the system, is *consistent* if the saved states form a consistent global state. Restarting a system from a set of inconsistent checkpoints may cause problems as illustrated below.

Processor p takes a checkpoint at time X and then sends a message to processor q (Figure 2.2). After receiving this message, q takes a checkpoint at time Y. Subsequently, p fails and restarts from the checkpoint taken at X. The global state at p's restart is inconsistent because p's local state shows that no message has been sent to q, while q's local state shows that a message from p has been received. If p and q are processors supervising a customer's account at different banks, and the message transfers funds from p to q, the customer will have the funds at *both* banks when p restarts. This inconsistency persists even if q is forced to roll back and restart from its checkpoint taken at Y. Consistent checkpoints prevent such a problem. Hence, our goal is to derive an algorithm for creating consistent sets of checkpoints, and a rollback-recovery algorithm to restart the system from these consistent checkpoints.

Rollback-recovery from consistent checkpoints may cause message losses, as shown in Figure 2.3. Processor q sends a message m to processor p, p receives m and fails at F, and then p and q roll back and recover from X

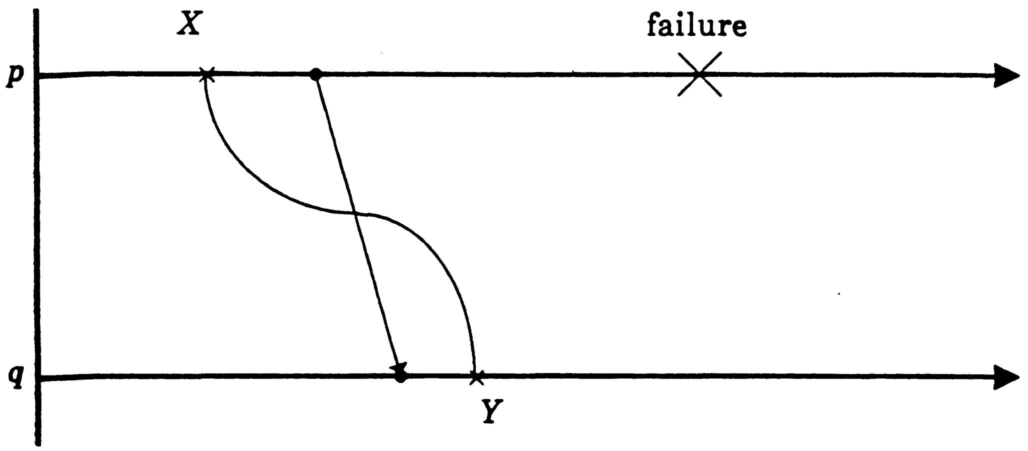

Figure 2.2: Inconsistent checkpoints.

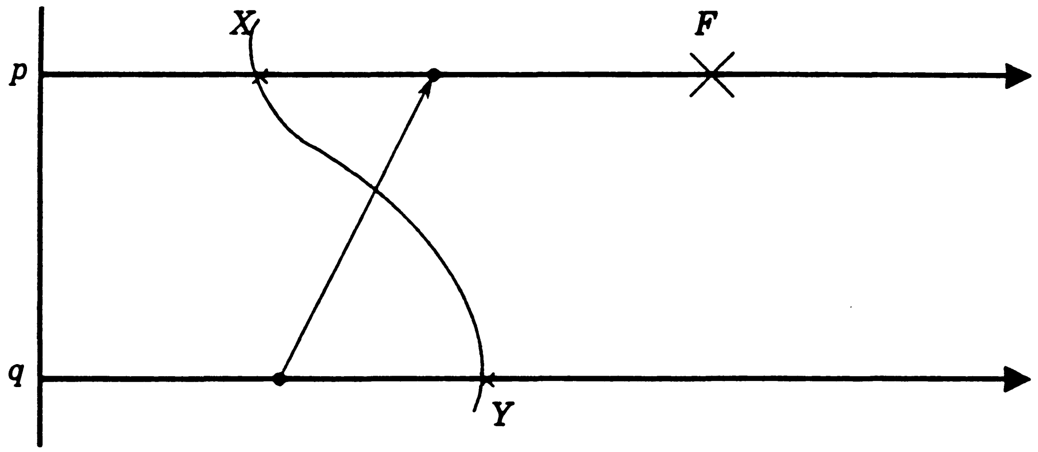

Figure 2.3: Message loss due to rollback recovery.

and Y, respectively. At this point, q is in a state in which it has already sent m, and p is in a state in which m has not been received. Furthermore, the channel from q to p is empty. Hence, the system recovers from the consistent state $\{X, Y\}$ in which the message m is lost. This state can also be reached in an execution that had no rollback-recovery: q sends m and reaches Y, p reaches X, and the channel loses m. These two executions are indistinguishable to p and q. In both cases, the loss of m is masked by the end-to-end transmission protocol that we have assumed for the channels. Hence, standard end-to-end protocols can handle message losses that are due to channels, as well as losses that are due to site failures and rollback-recovery.

The problem of ensuring that the system recovers to a consistent global state after transient failures has two components: checkpoint creation and rollback-recovery; we examine each one in turn.

2.4.1 Checkpoint Creation

There are two approaches to creating checkpoints. With the first approach, processors take checkpoints independently and save all checkpoints on stable storage. Upon a failure, processors must find a consistent set of checkpoints among the saved ones. The system is then rolled back to and restarted from this set of checkpoints [ALS79,Rus80,Woo81,Had82].

With the second approach, processors coordinate their checkpointing actions such that each processor saves only its most recent checkpoint, and the set of checkpoints in the system is guaranteed to be consistent. When a failure occurs, the system restarts from these checkpoints [TS84].

The main disadvantage of the first approach is the "domino effect" as illustrated in Figure 2.4 [Ran75,Pre83]. In this example, processors p and q have independently taken a sequence of checkpoints. The interleaving of messages and checkpoints leaves no consistent set of checkpoints for p and q, except the initial one at $\{X_0, Y_0\}$. Consequently, after p fails, both p and q must roll back to the beginning of the computation. For time-critical applications that require a guaranteed rate of progress, such as real time process control, this behavior results in unacceptable delays. An additional disadvantage of independent checkpoints is the large amount of stable storage required to save all checkpoints.

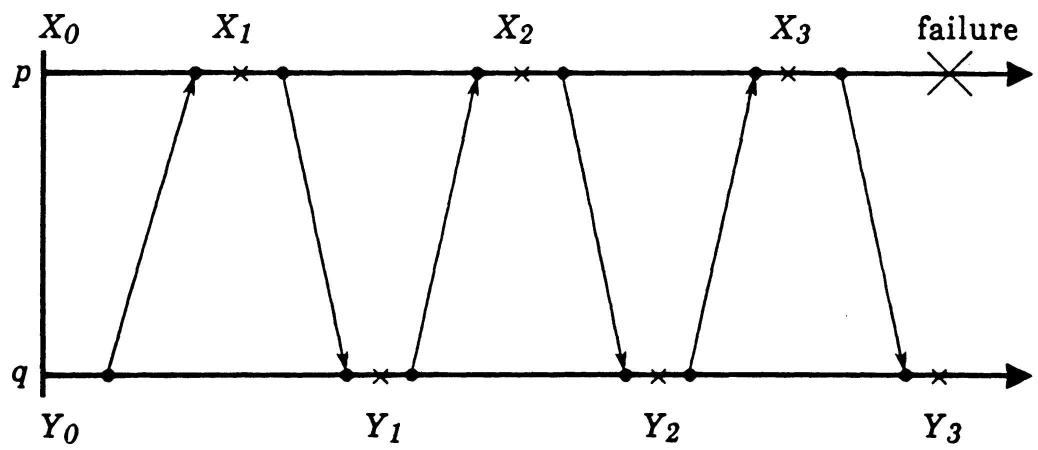

Figure 2.4: "Domino effect" following a failure.

To avoid these disadvantages, we pursue the second approach. In contrast to [TS84], our method ensures that when a processor takes a checkpoint, a minimal set of additional processors are forced to take checkpoints.

2.4.2 Rollback-Recovery

Rollback-recovery from a consistent set of checkpoints appears deceptively simple. The following scheme seems to work: Whenever a processor rolls back to its checkpoint, it notifies all other processors to also roll back to their respective checkpoints. It then installs its checkpointed state and resumes execution. Unfortunately, this simple recovery method has a major flaw. In the absence of synchronization, processors cannot all recover (from their respective checkpoints) simultaneously. Recovering processors asynchronously can introduce ***repeated rollbacks***; i.e., situations in which a single failure can cause an infinite number of rollbacks. Such a situation is illustrated below.

Figure 2.5 illustrates the histories of two processors, p and q, up to p's failure. Processor p fails before receiving the message n_1, rolls back

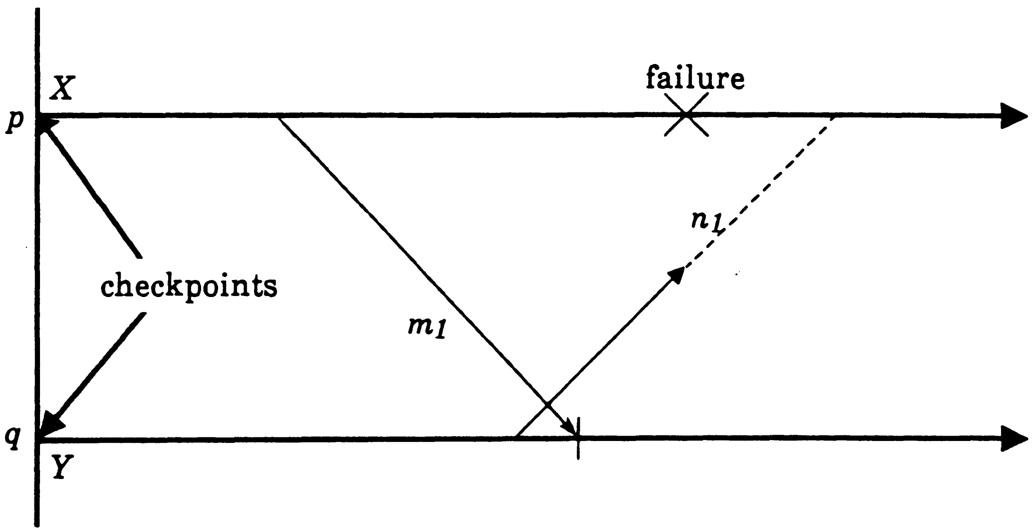

Figure 2.5: Histories of p and q up to p's failure.

to its checkpoint, and notifies q. Then p recovers, sends m_2, and receives n_1. After p's recovery, p has no record of sending m_1, whereas q has a record of its receipt. Therefore, the global state is inconsistent. To restore consistency, q must also roll back (to "forget" the receipt of m_1). After q rolls back, it has no record of sending n_1 whereas p has a recording of receiving n_1. Hence, p must roll back the second time to restore consistency (Figure 2.6). Furthermore, q sends n_2 and receives m_2, after it recovers. Message n_2 is received by p after it rolls back. However, as a result of this second rollback, p "forgets" the sending of m_2. Therefore, q must roll back the second time to restore consistency. And this second rollback of q will cause the third rollback of p because p receives the message n_2. It is now clear that p and q can be forced to roll back forever, even though no additional failures occur.

Our rollback-recovery algorithm solves this livelock problem. It tolerates failures that occur during its execution, and forces a minimal set of processors to roll back after a failure; whereas in [TS84], a single failure forces the system to roll back as a whole and the system crashes (and does not recover) if a failure occurs while it is rolling back.

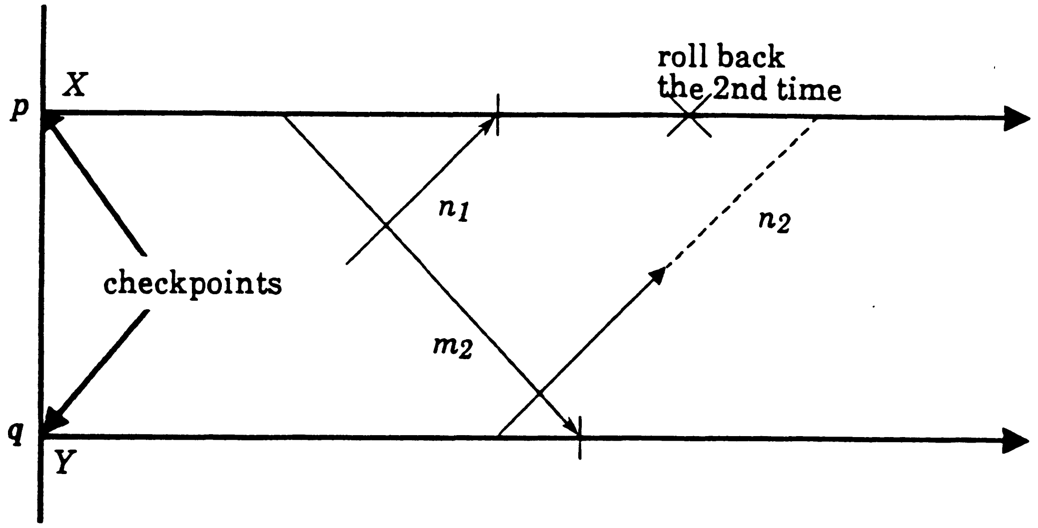

Figure 2.6: Histories of p and q up to p's second rollback.

2.5 Checkpoint Creation

2.5.1 Naive Algorithms

From Figure 2.2, it is obvious that if every processor takes a checkpoint after every sending of a message, and these two actions are done atomically, the set of the most recent checkpoints is always consistent. But creating a checkpoint after every send may be expensive. We may naively reduce the cost of the above method with a strategy such as "every processor takes a checkpoint after every k sends, $k > 1$" or "every processor takes a checkpoint on the hour". However, the former can be shown to suffer domino effects by a construction similar to the one in Figure 2.4, whereas the latter is meaningless for a system that lacks perfectly synchronized clocks.

2.5.2 Classes of Checkpoints

Our algorithm saves two kinds of checkpoints on stable storage: permanent and tentative. A permanent checkpoint cannot be undone. It guarantees that the computation needed to reach the checkpointed state will not be repeated. A tentative checkpoint, however, can be undone or changed to be a permanent checkpoint. When the context is clear, we call permanent checkpoints "checkpoints".

Consider a system with a consistent set of permanent checkpoints. A checkpoint algorithm is *resilient* to failures if the set of permanent checkpoints is still consistent after the algorithm terminates, even if some processors fail during its execution. To exclude the impractical "naive" algorithm (in last section) from our consideration, henceforth, we consider only those systems where processors either cannot afford to take a checkpoint after every send, or cannot combine the sending of a message and the taking of a checkpoint into one atomic operation. The following theorem shows that checkpoint algorithms for these systems must store at least two kinds of checkpoints on stable storage to be resilient to failures.

Theorem 2.5.1 *No resilient checkpoint algorithms that take only permanent checkpoints exist.*

Proof: By contradiction. Suppose that such an algorithm A exists. Consider the following scenario: p and q are processors in the system. Suppose that at some time $t > 0$, processor p invokes A to take a checkpoint, and that by time t, p has received a message m_q from q, and q a message m_p from p. Furthermore, suppose that A terminates by time t', and that p takes a permanent checkpoint C_{p,t_p} at time t_p where $t < t_p \leq t'$. Since A is resilient, the set of checkpoints at the termination of A must be consistent. Therefore, processor q must also have taken a permanent checkpoint C_{q,t_q} at time t_q, $t < t_q \leq t'$. Let d be the minimum time required for the failure of a processor to be detected. Depending on whether $t_p \leq t_q$ or $t_p > t_q$, we now construct another execution of A showing that A is not resilient to failure.

Case 1: $t_p \leq t_q$. Let q fail in the time interval $(\max(t, t_q - d), t_q)$. Processor p discovers the failure after t_q, hence after t_p. (See Figure 2.7.) Consequently, C_{p,t_p} is taken although C_{q,t_q} is not. Since C_{p,t_p} is a permanent checkpoint that cannot be undone, and q fails before making a permanent

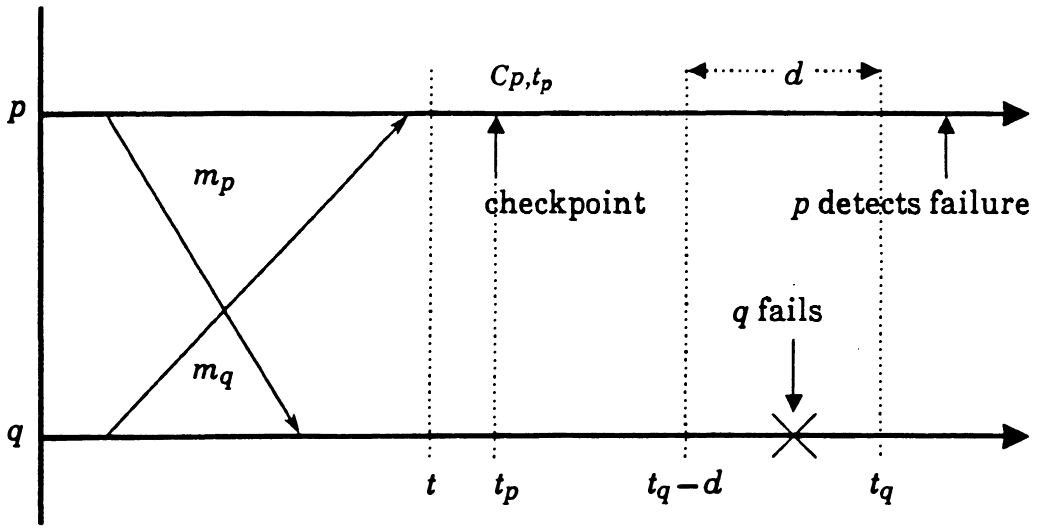

Figure 2.7: The scenario when $t_p \leq t_q$ and q fails.

checkpoint, the sending of m_q is "forgotten" forever whereas the receipt of m_q is "remembered" always, no matter what $\mathcal{A}$ does after p detects the failure. Hence, contrary to our assumption, Algorithm $\mathcal{A}$ is not resilient.

Case 2: $t_p > t_q$. Let p fail in the time interval $(\max(t, t_p - d), t_p)$. The rest of the proof is analogous to Case 1. □

Theorem 2.5.1 shows that in those systems we consider, any resilient checkpoint algorithm must store at least two kinds of checkpoints.

2.5.3 Our Checkpoint Algorithm

We first assume that a *single* processor invokes the algorithm to take a permanent checkpoint. In Section 2.7, we will extend the algorithm for concurrent invocations. We also assume that no site fails during the execution of the algorithm. In Section 2.5.3.4, we extend the algorithm to handle such failures. The algorithm sends its messages over (virtually) lossless and FIFO channels.

2.5.3.1 Motivation

The algorithm is patterned on two-phase-commit protocols. In the first phase, the initiator q takes a tentative checkpoint and requests all processors to take tentative checkpoints. If q learns that all processors have taken tentative checkpoints, q decides all tentative checkpoints should be made permanent; otherwise, q decides tentative checkpoints should be discarded. In the second phase, q's decision is propagated and carried out by all processors. Since all or none of the processors take permanent checkpoints, the most recent set of checkpoints is always consistent.

However, our goal is to force a minimal set of processors to take checkpoints. The above algorithm is modified as follows: a processor p takes a tentative checkpoint after it receives a request from q *only if* q's tentative checkpoint records the receipt of a message from p, and p's latest permanent checkpoint does not record the sending of that message. (Note that the definition of consistency requires only that every message recorded as "received" in a checkpoint should also be recorded as "sent" in another checkpoint; and not vice versa.) Processor p determines whether this condition is true using the label appended to q's request. This labeling scheme is described below.

Messages that are not sent by the checkpoint or rollback-recovery algorithms are *system* messages. Every system message m contains a field, which is a label denoted by $m.l$. Each processor uses monotonically increasing labels in its outgoing system messages. We define $\perp$ and $\top$ to be the smallest and largest labels, respectively. For any processors q and p, let m be the last message that q received from p after q took its last permanent or tentative checkpoint. Define:

$$last\_rmsg_q(p) = \begin{cases} m.l & \text{if } m \text{ exists} \\ \perp & \text{otherwise} \end{cases} .$$

Also, let m be the first message that q sent to processor p after q took its last permanent or tentative checkpoint. Define:

$$first\_smsg_q(p) = \begin{cases} m.l & \text{if } m \text{ exists} \\ \perp & \text{otherwise} \end{cases} .$$

When q requests p to take a tentative checkpoint, it appends $last\_rmsg_q(p)$ to its request; p takes the checkpoint only if $last\_rmsg_q(p) \geq first\_smsg_p(q) > \perp$.

2.5.3.2 Informal Description

Processor p is a *ckpt_cohort* of q if q has taken a tentative checkpoint, and $last\_rmsg_q(p) > \perp$ before the tentative checkpoint is taken. The set of ckpt_cohorts of q is denoted $ckpt\_cohort_q$. Every processor p keeps a variable $willing\_to\_ckpt_p$ to denote its willingness to take checkpoints. Whenever p cannot take a checkpoint (for any reason), $willing\_to\_ckpt_p$ is "no". The initiator q starts the checkpoint algorithm by making a tentative checkpoint and sending a request "take a tentative checkpoint and $last\_rmsg_q(p)$" to all $p \in ckpt\_cohort_q$. A processor p *inherits* this request if $willing\_to\_ckpt_p$ is "yes" and $last\_rmsg_q(p) \geq first\_smsg_p(q) > \perp$. If p inherits a request, it takes a tentative checkpoint and sends "take a tentative checkpoint and $last\_rmsg_p(r)$" requests to all $r \in ckpt\_cohort_p$. If p receives but does not inherit a request from q, p replies $willing\_to\_ckpt_p$ to q.

After p sends out its requests, it waits for replies that can be either "yes" or "no", indicating a ckpt_cohort's acceptance or rejection of p's request. If any reply is "no", $willing\_to\_ckpt_p$ becomes "no"; otherwise $willing\_to\_ckpt_p$ is unchanged. Processor p then sends $willing\_to\_ckpt_p$ to the processor whose request p has inherited. From the time p takes a tentative checkpoint to the time it receives the decision from the initiator, p does not send any system messages.

If all the replies from its ckpt_cohorts arrive and are all "yes", the initiator decides to make all tentative checkpoints permanent. Otherwise the decision is to undo all tentative checkpoints. This decision is propagated in the same fashion as the request "take a tentative checkpoint" is delivered. A processor discards its previous checkpoints after it takes a new permanent checkpoint.

The algorithm is presented in Figure 2.8. For simplicity, we create a fictitious processor called *daemon* to assume the initiation and decision tasks of the initiator. In practice, daemon is a part of the initiator.

2.5.3.3 Proofs of Correctness

We consider a single invocation of the algorithm, and we assume no processor fails in the system.

Lemma 2.5.1 *Every processor inherits at most one request to take a tentative checkpoint.*

Daemon processor:

send(*initiator*, "take a tentative checkpoint and $\top$");

await(*initiator*, $willing\_to\_ckpt_{initiator}$);

if $willing\_to\_ckpt_{initiator}$ = "yes" $\longrightarrow$

send(*initiator*, "make tentative checkpoint permanent");

$\square$ $willing\_to\_ckpt_{initiator}$ = "no" $\longrightarrow$

send(*initiator*, "undo tentative checkpoint");

fi.

All processors p:

- Initial State:

$first\_smsg_p(daemon) = \top$;

$$willing\_to\_ckpt_p = \begin{cases} \text{"yes"} & \text{if } p \text{ is willing to take a checkpoint} \\ \text{"no"} & \text{otherwise} \end{cases} ;$$

{ to be continued on the next page }

Figure 2.8: Algorithm C1: the Checkpoint Algorithm.

Figure 2.8 (continued)

- Upon receipt of "take a tentative checkpoint and $last\_rmsg_q(p)$" from q **do**

  if $willing\_to\_ckpt_p$ and $last\_rmsg_q(p) \geq first\_smsg_p(q) > \perp \longrightarrow$

  take a tentative checkpoint;

  for all $r \in ckpt\_cohort_p$,

  send(r, "take a tentative checkpoint and $last\_rmsg_p(r)$");

  for all $r \in ckpt\_cohort_p$, await(r, $willing\_to\_ckpt_r$);

  if $\exists\ r \in ckpt\_cohort_p$, $willing\_to\_ckpt_r$ = "no" $\longrightarrow$

  $willing\_to\_ckpt_p$:= "no" **fi**;

  fi;

  send(q, $willing\_to\_ckpt_p$);

  od.

- Upon receipt of m="make tentative checkpoint permanent" or

  m="undo tentative checkpoint" **do**

  if m="make tentative checkpoint permanent" $\longrightarrow$

  make tentative checkpoint permanent;

  $\square$ m="undo tentative checkpoint" $\longrightarrow$

  undo tentative checkpoint;

  fi;

  for all $r \in ckpt\_cohort_p$, send(r, m);

  od.

Proof: Immediately after a processor p inherits a request it takes a tentative checkpoint. From the time p takes this checkpoint to the time it receives the initiator's decision, p does not send any system messages. Therefore, during this interval of time $first\_smsg_p(q) = \perp$ for all q, and p cannot inherit additional requests. □

Lemma 2.5.2 *Every processor terminates its execution of Algorithm C1.*

Proof: Any processor that executes C1 without taking a tentative checkpoint clearly terminates. Let p be a processor that takes a tentative checkpoint. By Lemma 2.5.1, p takes a tentative checkpoint exactly once. Consequently, to prove that C1 terminates at p, it suffices to prove that after p takes a tentative checkpoint, it does not wait forever for either the "yes" or "no" from its ckpt_cohorts, or the initiator's decision.

Let q be a ckpt_cohort of p. If q inherits p's request to take a tentative checkpoint, it sends $willing\_to\_ckpt_q$ to p before waiting for the initiator's decision. On the other hand, if q does not inherit p's request, it sends $willing\_to\_ckpt_q$ to p immediately after receiving p's request. Therefore, there can be no deadlock involving p waiting for q's reply and q waiting for the initiator's decision.

Neither can processor p be in a deadlock waiting for replies from its ckpt_cohorts. To show this, note that if q inherits a checkpoint request from p, p inherits a request before q does. The inherit relation cannot be circular, and hence no deadlock can arise. Therefore, p will receive replies from all its ckpt_cohorts.

After the initiator receives replies from all its ckpt_cohorts, it decides whether to make tentative checkpoints permanent or not. This decision is guaranteed to reach all processors that have taken tentative checkpoints since all processors forward the decision, and message channels are reliable. Thus processor p does not wait forever for replies from its ckpt_cohorts, or for the initiator's decision. □

The next lemma shows that C1 takes a consistent set of checkpoints.

Lemma 2.5.3 *If the set of checkpoints in the system is consistent before the execution of Algorithm C1, the set of checkpoints in the system is consistent after the termination of C1.*

Proof: Without loss of generality, assume new checkpoints are taken in C1. The proof is by contradiction. Suppose the set of checkpoints after C1 terminates is not consistent. Then there are two processors p and q such that p sent q a message m after making its permanent checkpoint, and q received m before making its permanent checkpoint. Since all checkpoints are consistent before the execution of C1, q must have taken its permanent checkpoint during this execution. Before q took a tentative checkpoint in C1, $\mathit{last\_rmsg}_q(p) \geq m.l$; hence, p was in $\mathit{ckpt\_cohort}_q$ and received a request to take a tentative checkpoint from q. When p received the request, $\mathit{willing\_to\_ckpt}_p$ had to be "yes" because q could not have made its tentative checkpoint permanent otherwise. Furthermore, either p had already taken a tentative checkpoint after sending m, or $\mathit{last\_rmsg}_q(p) \geq m.l \geq \mathit{first\_smsg}_p(q) > \perp$. In both cases, p took a tentative checkpoint after sending m. However, p makes its tentative checkpoint permanent if q makes its permanent. Consequently, p took a permanent checkpoint after sending m, which is a contradiction. □

We now show the set of processors that take new permanent checkpoints during the execution of Algorithm C1 is minimal. Let the set of processors that take new permanent checkpoints in C1 be $P = \{p_0, p_1, \cdots, p_k\}$, where p_0 is the initiator. Let $C(P) = \{c(p_0), c(p_1), \cdots, c(p_k)\}$ be the new permanent checkpoints taken by processors in P. Define an alternate set of checkpoints: $C'(P) = \{c'(p_0), c'(p_1), \cdots, c'(p_k)\}$ where $c'(p_0) = c(p_0)$ and for all $1 \leq i \leq k$, $c'(p_i)$ is either $c(p_i)$ or the checkpoint p_i had before executing C1.

Theorem 2.5.2 *$C'(P)$ is consistent if and only if $C'(P) = C(P)$.*

Proof: The *if* part is by Lemma 2.5.3. We now prove the *only if* part. The execution of C1 imposes a "p inherits a request from q" relation on the set of processors. Since this relation is noncircular and there is only one initiator, it can be represented as a tree T: the root of T is the initiator, and p is a child of q if and only if p inherits a request from q. If $p \in T$, it must make a new permanent checkpoint during the execution of C1; hence $p \in P$. If $p \in P$, either p is the initiator or it inherits a request; hence $p \in T$. Therefore, $p \in T$ if and only if $p \in P$.

Our proof is by contradiction. Suppose that $C'(P) \neq C(P)$ and $C'(P)$ is consistent. Let $r \in P$ such that $c'(r) \neq c(r)$. Note that $r \neq p_0$, and that there exists a path from r to p_0 in T. Since $c'(p_0) = c(p_0)$, there is an edge (p, q) on this path such that $c'(p) \neq c(p)$, and $c'(q) = c(q)$. When p inherits q's request, $last\_rmsg_q(p) \geq first\_smsg_p(q) > \perp$. Let m be the message that q receives from p such that $last\_rmsg_q(p) = m.l$. Since $m.l \geq first\_smsg_p(q)$, the sending of m is not recorded in $c'(p)$. But the receipt of m is recorded in $c'(q)$. Thus, $C'(P)$ is not a consistent set of checkpoints, a contradiction. □

Theorem 2.5.2 shows that if p_0 takes a checkpoint, then all processors in P must take a checkpoint to ensure consistency.

2.5.3.4 Coping with Failures

We now extend Algorithm C1 to handle processors' failures. We first consider the effects of failures on nonfaulty processors. When failures occur, a nonfaulty processor may not receive some of the following messages:

1. "yes" or "no" from ckpt_cohorts,
2. "make tentative checkpoint permanent" or "undo tentative checkpoint" from the initiator.

Suppose that processor p fails before replying "yes" or "no" to processor q's request. By the assumption of Section 2.3, q will know of p's failure. After q knows that p has failed, it sets $willing\_to\_ckpt_q$ to "no" and stops waiting for p's reply. Therefore, to take care of a missing "yes" or "no", it suffices to change the statement in C1 from

$$\textbf{if } \exists\, r \in ckpt\_cohort_p,\ willing\_to\_ckpt_r = \text{"no"} \longrightarrow$$
$$willing\_to\_ckpt_p := \text{"no"}\ \textbf{fi}$$

to

$$\textbf{if } \exists\, r \in ckpt\_cohort_p,\ willing\_to\_ckpt_r = \text{"no" or } r \text{ has failed} \longrightarrow$$
$$willing\_to\_ckpt_p := \text{"no"}\ \textbf{fi}.$$

Suppose that processor p does not receive the decision regarding its tentative checkpoint. If p undoes its tentative checkpoint or makes it permanent, it risks contradicting the initiator. The two-phase structure of C1 requires p to block until it discovers the initiator's decision [Ske82]. We will discuss ways to prevent blocking in Section 2.8.

We now consider the recovery of faulty processors. When a processor restarts after a failure, its latest checkpoint on stable storage may be tentative or permanent. If this checkpoint is tentative, the restarting processor must decide whether to discard it or to make it permanent. The decision is made as follows:

Suppose that the restarting processor is the initiator. The initiator knows that every processor that has taken a tentative checkpoint is still blocked waiting for its decision. Hence, it is safe for the initiator to decide to undo all tentative checkpoints and send this decision to its ckpt_cohorts. If the restarting processor is not the initiator, it must discover the initiator's decision regarding tentative checkpoints. It may contact either the initiator or those processors of which it is a ckpt_cohort; it follows the decision accordingly to terminate C1.

The restarting processor is now left with one permanent checkpoint on stable storage. It can recover from this checkpoint by invoking the rollback-recovery algorithm of Section 2.6.

Let C2 be the Algorithm C1 as modified above. C2 terminates if all processors that fail during the execution of C2 recover. At termination, the set of checkpoints in the system is consistent, and the set of processors that took new permanent checkpoints is minimal. The proofs for these properties are similar to those of C1 and are omitted.

2.6 Rollback-Recovery

We first assume that a *single* processor invokes the algorithm to roll back and recover (henceforth denoted *restart*). We also assume that the checkpoint algorithm and the rollback-recovery algorithm are not invoked concurrently. In Section 2.7, we will describe concurrent invocations of these algorithms. The algorithm sends its messages over (virtually) lossless and FIFO channels.

2.6.1 Motivation

The rollback-recovery algorithm is patterned on two-phase-commit protocols. In the first phase, the initiator q requests all processors to restart from their checkpoints. Processor q decides to restart all the processors if and only if they are all willing to restart. In the second phase, q's decision is propagated and carried out by all processors. We will prove that the two-phase structure of this algorithm prevents repeated rollbacks as discussed in Section 2.4.2. Since all processors follow the initiator's decision, the global state is consistent when the rollback-recovery algorithm terminates.

However, our goal is to force a minimal set of processors to roll back. If a processor p rolls back to a state saved before an event e occurred, we say that e is *undone* by p. The above algorithm is modified as follows: the rollback of a processor q forces another processor p to roll back *only if* q's rollback undoes the sending of a message to p. Processor p determines if it must restart using the label appended to q's "prepare to roll back" request. This label is described below.

For any processors q and p, let m be the last message that q sent to p before q took its latest permanent checkpoint. Define

$$last\_smsg_q(p) = \begin{cases} m.l & \text{if } m \text{ exists} \\ \top & \text{otherwise} \end{cases} .$$

When q requests p to restart, it appends $last\_smsg_q(p)$ to its request. Processor p restarts from its permanent checkpoint only if $last\_rmsg_p(q) > last\_smsg_q(p)$.

2.6.2 Informal Description

Processor p is a *roll_cohort* of q, if q can send messages to it. The set of roll_cohorts of q is $roll\_cohort_q$. To denote its willingness to roll back, every processor p keeps a variable $willing\_to\_roll_p$. Whenever p cannot roll back (for any reason), $willing\_to\_roll_p$ is "no". The initiator q starts the rollback-recovery algorithm by sending a request "prepare to roll back and $last_smsg_q(p)$" to all $p \in roll\_cohort_q$. A processor p *inherits* this request if $willing\_to\_roll_p$ is "yes", $last\_rmsg_p(q) > last\_smsg_q(p)$, and p has not already inherited another request to roll back. After p inherits the request, it sends "prepare to roll back and $last\_smsg_p(r)$" to all $r \in roll\_cohort_p$; otherwise, it replies $willing\_to\_roll_p$ to q.

After p sends out its requests, it waits for replies from each processor in $roll\_cohort_p$. The reply can be an explicit "yes" or "no" message, or an implicit "no" when p discovers that r has failed. If any reply is "no",

---

Daemon processor:

send(*initiator*, "prepare to roll back and $\perp$");

await(*initiator*, $willing\_to\_roll_{initiator}$);

if $willing\_to\_roll_{initiator}$ = "yes" $\longrightarrow$ send(*initiator*, "roll back");

$\square$ $willing\_to\_roll_{initiator}$ = "no" $\longrightarrow$ send(*initiator*, "do not roll back");

fi.

All processors p:

- Initial State:

  $ready\_to\_roll_p$ = true;

  $last\_rmsg_p(daemon) = \top$;

  $$willing\_to\_roll_p = \begin{cases} \text{"yes"} & \text{if } p \text{ is willing to roll back} \\ \text{"no"} & \text{otherwise} \end{cases} ;$$

{ to be continued on the next page }

Figure 2.9: Algorithm R: the Rollback Algorithm.

$willing\_to\_roll_p$ becomes "no", otherwise $willing\_to\_roll_p$ is unchanged. Processor p then sends $willing\_to\_roll_p$ to the processor whose request p inherits. From the time p inherits the rollback request to the time it receives the decision from the initiator, p does not send any system messages.

If all the replies from its roll_cohorts arrive and are all "yes", the initiator decides the rollbacks will proceed; otherwise it decides no processor will roll back. This decision is propagated to all processors in the same fashion as the request "prepare to roll back" is delivered. If failures prevent the decision from reaching a processor p, p must block until it discovers the initiator's decision. We discuss nonblocking algorithms in Section 2.8.

The rollback-recovery algorithm is presented in Figure 2.9. As in the

Figure 2.9 (continued)

- Upon receipt of "prepare to roll back and $last\_smsg_q(p)$" from q **do**

  if $willing\_to\_roll_p$ and $last\_rmsg_p(q) \geq last\_smsg_q(p)$ and $ready\_to\_roll_p \longrightarrow$

  $ready\_to\_roll_p :=$ false;

  for all $r \in roll\_cohort_p$, send(r, "prepare to roll back and $last\_smsg_p(r)$");

  for all $r \in roll\_cohort_p$, await(r, $willing\_to\_roll_r$);

  if $\exists\ r \in roll\_cohort_p$, $willing\_to\_roll_r$ = "no" or r has failed $\longrightarrow$

  $willing\_to\_roll_p :=$ "no" **fi**;

  fi;

  send(q, $willing\_to\_roll_p$);

  od.

- Upon receipt of m = "roll back" or m = "do not roll back"

  and $ready\_to\_roll_p$ = false **do**

  if m = "roll back" $\longrightarrow$ restart from p's permanent checkpoint;

  $\square$ m = "do not roll back" $\longrightarrow$ resume execution;

  fi;

  for all $r \in roll\_cohort_p$, send(r, m);

  od.

---

presentation of Algorithm C1, we introduce a fictitious processor called *daemon* to perform functions that are unique to the initiator of the algorithm.

2.6.3 Proofs of Correctness

We consider a single invocation of the rollback-recovery algorithm. The variable $ready\_to\_roll_p$ ensures that a processor p inherits at most one request to roll back. As a result, the variable also ensures that a processor rolls back at most once. To prove the termination of Algorithm R, it suffices to show that Algorithm R is free of deadlocks.

Lemma 2.6.1 *Algorithm R is deadlock-free.*

Proof: Similar to the proof of Lemma 2.5.2. □

We now show that the global state of the system is consistent after the termination of R.

Lemma 2.6.2 *If the system is consistent before the execution of Algorithm R, the system is consistent after the termination of Algorithm R.*

Proof: The proof is by contradiction. Suppose that after Algorithm R terminates at every process, the global state of the system is inconsistent. There must be a message m sent by a processor q to p such that during the execution of R, q undid the sending of m while p did not undo the receipt of m. We first show that p inherited a request to roll back. After q inherited a request to roll back, it stopped sending system messages. Hence, it must have sent a request to roll back to p *after* sending m. Moreover, since q undid the sending of m, $m.l > last\_smsg_q(p)$. On the other hand, processor p could not have taken a permanent checkpoint after receiving m and before receiving q's request: the creation of this checkpoint and the fact that q did not take a permanent checkpoint would contradict Lemma 2.5.3. Consequently, $last\_rmsg_p(q) \geq m.l > last\_smsg_q(p)$. In addition, the variable $willing\_to\_roll_p$ must have been "yes", for the initiator cannot have decided to roll back. Therefore, when q's request reached p, either p had already inherited a rollback request or it inherited q's request.

Since p and q received the same decision, p rolled back. Next we show that p's rollback undid the receipt of m. There are two cases to consider:

Case 1: m reached p after p inherited a rollback request. Since message channels are FIFO, m reached p before q's request did. The initiator made its decision after p replied to q's request. Therefore, p rolled back after receiving m.

Case 2: m reached p before p inherited a rollback request. We have shown that p did not take a permanent checkpoint after receiving m. Hence, the rollback of p undid the receipt of m. □

Lemma 2.6.2 ensures that a single execution of Algorithm R brings the system to a consistent state after a failure; since processors roll back at most once in any execution of R, R prevents repeated rollbacks.

Many existing rollback algorithms have the following undesirable property. If the initiator rolls back, it forces an additional set of processors P to roll back with it, even though the system will be consistent if some of the processors in P omit to roll back. For example, the algorithm in [TS84] requires all processors to roll back every time any processor wants to roll back. However, in some cases, the initiator could roll back alone and the system would still be consistent. With our algorithm, the set of processors that are forced to roll back with the initiator is minimal.

Theorem 2.6.1 *Let E be an execution of R in which the initiator, p_0, and an additional set of processors P roll back. Consider an execution E', identical to E except that a non-empty subset of processors in P omit to roll back upon receipt of the "roll back" decision. The execution E' leaves the system in an inconsistent state.*

Proof: The execution of R imposes a "p inherits a 'prepare to roll back' request from q" relation on the set of processors. Since this relation is noncircular and there is only one initiator, it can be represented as a tree T: the root of T is the initiator, p_0, and p is a child of q if and only if p inherits a request from q. If $p \in T$, it rolls back during the execution of R; hence $p \in P \cup \{p_0\}$. If $p \in P \cup \{p_0\}$, either p is the initiator or it inherits a request; hence $p \in T$. Therefore, $p \in T$ if and only if $p \in P \cup \{p_0\}$.

Our proof is by contradiction. Suppose $P' \subseteq P$ is the set of processors that omit to roll back in the execution E', and the system is consistent at the end of E'. Let $r \in P'$. There exists a path from r to p_0 in T. Since r omits to roll back and p_0 rolls back, there is an edge (p, q) on this path, such that p omits to roll back and q rolls back. When p inherits the "prepare to roll back" request from q, $last\_rmsg_p(q) > last\_smsg_q(p)$. Let m be the message that q sent to p such that $last\_rmsg_p(q) = m.l$. When q rolls back it undoes the sending of m. But since p omits to roll back, it does not undo the receipt of m. Thus, at the end of E', the global state of the system is inconsistent, a contradiction. □

2.7 Interference

In this section, we consider concurrent invocations of the checkpoint and rollback-recovery algorithms. An execution of these algorithms by processor p is *interfered* with if any of the following events occur:

1. Processor p receives a rollback request from another processor q while executing the checkpoint algorithm.

2. Processor p receives a checkpoint request from q while executing the rollback-recovery algorithm.

3. Processor p, while executing the checkpoint algorithm for initiator i, receives a checkpoint request from q, but q's request originates from a different initiator than i.

4. Processor p, while executing the rollback-recovery algorithm for initiator i, receives a rollback request from q, but q's request originates from a different initiator than i.

One single rule handles the four cases of interference: once p starts the execution of a checkpoint (rollback) algorithm, p is unwilling to take a tentative checkpoint (rollback) for another initiator, or to roll back (take a tentative checkpoint). As a result, in all four cases, p replies "no" to q. We can show this rule is sufficient to guarantee that all previous lemmas and theorems hold despite concurrent invocations of the algorithms. This rule can, however, be modified to permit more concurrency in the system. The modification is that in case (1), instead of sending "no" to q, p can begin executing the rollback-recovery algorithm after it finishes the checkpoint algorithm. We cannot allow a similar modification in case (2) lest deadlocks occur.

2.8 Optimization

When the initiator of the checkpoint or of the rollback-recovery algorithm fails before propagating its decision to its cohorts, it is desirable for processors not to block for its recovery. To prevent processors from blocking, we can modify our algorithms by replacing the underlying two-phase commit protocol with a nonblocking three-phase commit protocol [Ske82]. However, nonblocking protocols are inherently more expensive than blocking ones [DS83].

We next address the following problem: after a ckpt_cohort q of a processor p fails, p cannot take a permanent checkpoint until q restarts (p cannot know if the latest checkpoint of q records the sendings of all messages it received from q). To avoid waiting for q's restart, p can remove q from $ckpt\_cohort_p$ by restarting from its checkpoint (using the rollback-recovery algorithm). After its restart, processor p can take new checkpoints.

2.9 Conclusion

We have presented a checkpoint algorithm and a rollback-recovery algorithm to solve the problem of bringing a distributed system to a consistent state after transient failures. In contrast to previous algorithms, they tolerate failures that occur during their executions. Furthermore, when a processor takes a checkpoint, a minimal set of additional processors are forced to take checkpoints. Similarly, a minimal set of additional processors are forced to restart when a processor restarts after a failure. We also show that the stable storage requirement of our algorithms is minimal.

Chapter 3

Translation of Ideal Distributed Systems

3.1 Introduction

In many real distributed systems, processor speeds and message transmission times vary unpredictably over time. These unknown variations make coordination of processors' actions difficult. Hence, they complicate the design and verification of distributed programs that are to run on these real systems.

In this chapter, we develop a mechanism to simplify the programming for a large class of problems in real distributed systems. In particular, we develop a translation mechanism for designing programs and proving them correct with the simplifying assumption that the distributed system is *ideal*; i.e., processors execute in synchronized steps and messages take exactly one step to arrive. We can then use this translation mechanism to run these programs correctly on systems that violate this simplifying assumption; i.e., systems that are not ideal. This mechanism is transparent to programmers.

Our translation mechanism is based on a combination of checkpointing/rollback and message buffering. Given a program designed for the ideal system, the cost of running this program in a real system with our mechanism can be higher than the cost of running it in the ideal system. In fact, this additional cost is proportional to the deviation of this real system from the ideal system. The closer the performance of this system approaches that of the ideal system, the lower this cost.

The organization of this chapter is as follows. In Section 3.2, we formalize our notion of ideal and real distributed systems. Then we define translation in Section 3.3. Section 3.4 describes the translation mechanism, which is proved correct in Sections 3.5 and 3.6. The chapter ends with a discussion in Section 3.7.

3.2 Model of Distributed Systems

A distributed system consists of n processors that do not share memory and are completely connected by FIFO communication channels. We denote these n processors by the natural numbers $1, 2, \cdots, n$. Each processor is a deterministic state machine. In each state, a processor can execute zero or more atomic actions. Two particular actions are **send** and **receive**. A processor can send a message to (or receive messages from) processors by executing **send** (or **receive**). Communication channels deliver messages to a processor i by placing the messages in a buffer of i, Q_i. Messages are removed from Q_i when they are received by i. Buffer Q_i is not included in i's processor state.

A processor changes its state after executing an atomic action. The new state is a function of the old state, the action executed, and if the action is **receive**, the messages that are received.[1] Formally, let S_i be the set of states of processor i, A_i be the set of atomic actions i can execute, M be the set of messages, and $\mathcal{P}(M)$ be the set of subsets of M that contain at most one message from each processor. State transitions at processor i are determined by the function δ_i mapping from $S_i \times A_i \times \mathcal{P}(M)$ to S_i.

Every processor i has a *local* clock, c_i, that measures time discretely in ticks and is incremented by one tick for each execution of an atomic action by i. All local clocks are initially 0. When i's execution of an atomic action a advances c_i from tick c to tick $c+1$, a is said to occur at i at time $c+1$. Moreover, the state resulting from the execution of a is i's state at time $c+1$ (i's state at time 0 is its initial state). Clock c_i is not included in i's processor state.

The speeds of the processors are measured by a *global* clock. Note that this clock does not belong to any of the distributed systems we discuss; it is

[1] This is equivalent to stating that the new state is a function of the old state, the action executed, and the sequence of messages the processor has received.

merely a expositional device. This global clock measures time discretely in ticks, and is initially 0. We call a tick of the global clock a *global* tick. At every global tick, every processor takes either one *active* step by executing an atomic action, or one *inactive* step by not executing any atomic action. (The intervening inactive steps between any two successive active steps p and q can represent the additional time needed to finish executing p.) The number of active steps processor i has taken by a global tick u is equal to i's local clock at u. The speed of a processor is measured by the number of global ticks that elapse between its successive active steps: the smaller the number of global ticks elapsing between successive active steps, the greater the processor speed. Processor speeds may not be constant.

3.2.1 Histories

Let (a_c, c) denote the c^{th} active step of processor i, where a_c is the atomic action executed, and c is the local time at which the execution of a_c occurs. A *local history* of i is a pair $h_i = (s_i, \tau_i)$ such that

1. s_i is i's initial state;
2. τ_i is an infinite sequence of steps that i takes at successive global ticks.

Definition 3.2.1 *A global history H is a n-tuple $\{h_1, \cdots, h_n\}$ of local histories such that for all $1 \leq i \leq n$,*

1. *h_i is a local history of processor i; and*
2. *Processor j sends a message to processor i in the c^{th} step of h_j, if and only if for some $d > c$, i receives the message in the d^{th} step of h_i.*

When it is clear from context, "global history" is referred usually to as "history".

Let Φ and Δ be real numbers greater than or equal to 0. History H has a (Φ, Δ) *synchronicity*, if

1. For all $1 \leq i \leq n$, processor i takes at least one active step every Φ global ticks in H; and

2. For all $1 \leq i, j \leq n$, in H, processor j sends a message to processor i at global tick u, if and only if i receives the message by global tick $u + \Delta$.

Condition (1) means that processor speeds in H are bounded by Φ, while condition (2) means that message transmission times in H are bounded by Δ. The synchronicity (Φ, Δ) is bounded, if both Φ and Δ are bounded; it is unbounded otherwise.

A *distributed system* is a set (infinite) of global histories.

3.2.2 Model of Ideal Distributed System

A distributed system is *ideal*, if all its histories have $(1, 1)$ synchronicity; i.e., every processor takes one active step at every global tick, and every message sent takes exactly one global tick to go from its sender to its receiver. We call these processors *lock-step synchronized*. (Our definition of lock-step synchronized processors is similar to that of Dwork et al. [DDS87].) Since processors take one active step at every global tick, they do not take any inactive steps in the ideal system. Hence, their local clocks are synchronized perfectly; i.e., for all global tick $u > 0$, for all $1 \leq i, j \leq n$, $c_i = c$ at u if and only if $c_j = c$ at u. Moreover, processor j sends a message to processor i at local time $c_j = c$, if and only if i receives the message at local time $c_i = c + 1$.

3.2.3 Model of Real Distributed Systems

A distributed system is *real*, if each of its histories has bounded synchronicity; i.e., the processor speeds and message transmission times in every history are bounded. Note that it is possible that no one bounded synchronicity holds for all the histories of a real system. This fits our observation that while the processor speeds and message transmission times are bounded during any execution of a real system, none of the bounds that hold for particular executions may hold for *all* executions.

Local clocks in real systems are usually not synchronized perfectly. Moreover, since message transmission times may be greater than 1, even if j sends a message to i when $c_j = c_i = c$, i may not receive the message until, say, $c_i = c + 99$.

When designing a program for the ideal system, a programmer can take advantage of the fact that processors are lock-step synchronized and message transmission times are exactly 1; i.e., the correctness of the program may *depend* on these two properties. For example, a program can be designed to require a processor j to send a message m to processor i at $c_j = c$, and processor i to send a message m' to processor k at $c_i = c + 2$. The correctness of this program may depend on the fact that on an ideal system, i always receives m at $c_i = c + 1$; and thus, i always receives m before sending m'. However, running this program on a real system may result in message m arriving after i has already sent m'. Therefore, programs designed and proved correct for the ideal system may not be correct when they are run in real systems. They will be correct, however, if run with our translation mechanism.

3.3 Translation: a formal definition

We first define formally the executions of a distributed program in a distributed system (ideal or real) before defining translation.

3.3.1 Distributed Programs

A *local program* P_i of processor i is a total function that maps the processor states of i and i's local clock to i's atomic actions; i.e., P_i maps from $S_i \times c_i$ to A_i. (This is equivalent to stating that P_i is deterministic. It is easy to generalize to allow P_i be non-deterministic and non-total.) A *distributed program* is a n-tuple $\{P_1, P_2, \cdots, P_n\}$ such that for all $1 \leq i \leq n$, P_i is a local program of processor i.

Informally, a history $\{h_1, \cdots, h_n\}$ is an execution of a distributed program (or *satisfies*) $\{P_1, \cdots, P_n\}$, if for all $1 \leq i \leq n$, the sequence of atomic actions in h_i corresponds to an execution of local program P_i beginning from i's initial state in h_i. The formal conditions that must be met are as follows. For all $1 \leq i \leq n$ and $l > 0$, let

$s_i[0]$ be i's initial state in h_i;
$\sigma_i[0]$ be an empty set;
$a_i[l]$ be the action executed in the l^{th} active step of h_i;
$s_i[l]$ be i's state after $a_i[l]$ is executed; and

$\sigma_i[l]$ be the set of messages received by i in $a_i[l]$.

Formally, history $\{h_1, \cdots, h_n\}$ is *an execution of program* $P = \{P_1, \cdots, P_n\}$, if for all $1 \leq i \leq n$ and $l > 0$,

1. $a_i[l] = P_i(s_i[l-1], l-1)$, and

2. $s_i[l] = \delta_i(s_i[l-1], a_i[l], \sigma_i[l])$.

The *executions of a distributed program P in a system S* are the executions of P that are also histories of S.

3.3.2 Problem Specifications

A problem *specification* is a predicate on histories [NT86]. A program P *meets* a specification L in a system, if every execution of P in the system satisfies L. However, not all problem specifications can be met in both ideal and real distributed systems. For example, a specification that requires all processors to perform a particular action at the same time on the global clock cannot be met in systems in which processors are not lock-step synchronized [HM85].

Neiger and Toueg [NT86] introduced the notion of *logical specifications*. Informally, these are problem specifications that do not refer to the global times at which atomic actions are executed; they may only refer to local times at which atomic actions are executed. For example, "all transactions executed in a database are serializable" is a logical specification. Many problems in distributed systems have logical specifications: e.g., determination of global consistent states, leader election, resource allocation, concurrency control, transaction commit. To define logical specifications, we define first logically equivalent histories.

Definition 3.3.1 *Two histories $H = \{h_1, \cdots, h_n\}$ and $G = \{g_1, \cdots, g_n\}$ are* logically equivalent, *if for each $1 \leq i \leq n$,*

1. *processor i's initial states in h_i and g_i are the same; and*

2. *the sequences of active steps i takes in h_i and g_i are the same.*

Definition 3.3.2 *A specification L is* logical, *if for any two logically equivalent histories H and G, H satisfies L if and only if G satisfies L.*

Given a logical specification L that is met by some distributed program P in the ideal distributed system, *translation* is the transformation of P into another distributed program P' such that P' meets L in any real distributed system. In the next section, we will describe our mechanism to achieve translation.

3.4 Implementation of Translation

Fix a distributed program $P = \{P_1, \cdots, P_n\}$ that meets a logical specification L in the ideal system. We now describe the translation mechanism that runs P correctly in a real distributed system. The mechanism is based on the model of execution in the ideal system described in Figure 3.1. One iteration of the loop in the figure corresponds to one active step of processor i in the ideal system.

In the ideal system, since processors are lock-step synchronized and message transmission times are 1, every message that is sent at local time c (on the sender's clock) is received at local time $c+1$ (on the receiver's clock). In contrast, in real systems processors may not be lock-step synchronized and message transmission times may not be 1. For example, a message that is sent at time c may be received, say at time $c-2$ or at time $c+10$. When a message that is sent at time c is received before time $c+1$, we say that the message is received *early*; and when the message is received after $c+1$, we say that the message is received *late*. To guarantee that P will meet specification L in a real system, no messages should be received either early or late.

3.4.1 Handling Early Messages

It is easy to ensure that no messages is received early. First, whenever a processor sends a message m, it appends the time of the send (according to its local clock) to m as a timestamp (which we denote by c). Message m arrives too early if c is greater than the receiver's local clock. The receiver simply saves the early message in some buffer and receives it when its local clock reaches $c+1$. Thus, no messages is received early.

If c equals the receiver's local clock, then m is neither early nor late; it arrives *on-time*. A processor receives on-time messages in its first active

```
Begin
    { c_i is the local clock }
    { σ_i is the set of messages just received }
    { Q_i is buffer of arrived but not received messages }
    { s_i is the processor state }
    c_i, σ_i, Q_i := 0, ∅, ∅;
    s_i := initial state;
    while true do
        a := P_i(s_i, c_i);
        c_i := c_i + 1;
        if  a = receive ⟶ σ_i, Q_i := Q_i, ∅;
        [] a = "send m to all processors in J" ⟶ send(J, m);
        [] a ≠ receive and a ≠ send ⟶ skip;
        fi;
        s_i := δ_i(s_i, a, σ_i);
    od;
End.
```

Figure 3.1: Model of execution of program P_i in the ideal distributed system.

step after their arrivals.

If c is less than the receiver's local clock, m arrives late.

3.4.2 Handling Late Messages

Suppose that a message with timestamp c arrives at processor i late. To receive this message on-time, i has to first set back its clock to c, undo all its active steps taken after c, and restore the state it had at c. This series of operations is called a *rollback to time c*. After the rollback, processor i receives the message at time $c + 1$. This way, messages that arrive late are received on-time.

A processor that rolls back to receive a late message is called an *initiator*. As we have explained in Section 2.4, when a processor rolls back, other processors may have to also roll back to keep the system consistent. (These additional rollbacks are usually effected by *rollback messages* sent by initiators.) Furthermore, processors have to take checkpoints of their states to roll back.

Consequently, our translation needs a checkpointing/rollback mechanism. This mechanism should be free of the four potential problems that are identified in the previous chapter: domino effect, repeated rollbacks, message loss, and interference among concurrent initiations of checkpointing and rollbacks. Moreover, the checkpointing/rollback mechanism used in our translation should have the following property: that when the distributed program $P = \{P_1, \cdots, P_n\}$ is run by our translation in a real system, the processors are guaranteed to make "progress" (which is defined formally in Section 3.5.)

Unfortunately, if we use the checkpointing/rollback mechanism from the previous chapter, the resulting translation will not have the above progress property. The reason is as follows. Recall from Section 2.4 that rolling back to a consistent set of checkpoints may cause message losses. In the previous checkpointing/rollback mechanism, our technique to solve the message loss problem is based on the retransmission of lost messages. However, in the context of translation, retransmitted messages can arrive late and cause an unbounded number of rollbacks, as the following example illustrates.

Suppose that at local time $c_i = c + 1$, processor i receives a message m with timestamp c from processor j. Subsequently, both i and j roll back to time c (say because they receive rollback messages from a third processor).

While i's rollback undoes the receipt of m, j's rollback does not undo the sending of m. Thus, message m is lost. To overcome this loss, after j rolls back, it retransmits m again to i with timestamp c. Suppose that m arrives when $c_i = c + 2$. Processor i will have to roll back to c once more. Thus, j needs to retransmit m to i a third time. But again, m can arrive late at i, causing i to roll back to c. It is clear, then, that if every retransmitted m arrives late, processor i will have to roll back an infinite number of times.

Consequently, we have to use a different technique to solve the message loss problem. The resulting checkpointing/rollback mechanism is different from the one we described previously. Moreover, it will have the properties listed above: namely, that it is free of domino effect, repeated rollbacks, message loss, and interference among concurrent initiations of checkpointing and rollbacks; and that when program P is run by our translation in a real system, the processors are guaranteed to make progress.

Two initiations of rollbacks are said to be *concurrent*, if neither initiator receives a rollback message from the other initiator before initiating its rollback. We now describe the basic version of the new checkpointing/rollback mechanism. This version is based on the assumption that no initiations of rollbacks are concurrent. In the next chapter, we will describe the complete version, which does not depend on this simplifying assumption.

3.4.2.1 Checkpointing

To avoid domino effects, every processor in real systems makes a checkpoint after executing every atomic action prescribed by the distributed program P. (An optimization to reduce the number of checkpoints saved by processors is discussed in next chapter.) Consequently, when a processor rolls back to time c, no other processor needs to roll back to time before c; hence, there will be no domino effects. Furthermore, when a processor makes a checkpoint, it does not force other processors to also make a checkpoint. Hence, processors make their checkpoints *independently* of each other; and there will be no interference between processors' checkpointings, and between one processor's checkpointing and another processor's rollback.

3.4.2.2 Rollback

A processor i initiates a rollback to time c by sending a rollback message to all processors. Upon receiving the rollback message, a processor rolls back to c. (Although not all processors need to roll back, we omit this optimization for simplicity.) If after rolling back, a processor receives a message whose send is undone by i's initiation, it has to roll back again. This we call the repeated rollbacks problem. (A detailed example of repeated rollbacks is in Section 2.4.2.) Messages that cause repeated rollbacks are *orphans*.

Definition 3.4.1 *Let m be a message with timestamp c that processor i receives from processor j. Message m is an* orphan *to i at global tick u, if before u, some processor initiates a rollback to time $d < c$ such that as a result of this initiation,*

1. *i rolls back to time d before u; and*
2. *j rolls back to d after sending m.*

Note that the above definition does not depend on whether processor j rolls back to d before or after u. We introduce *incarnations* to help identify orphans.

By our assumption that there are no concurrent initiations of rollbacks, initiations of rollbacks are serially ordered. Therefore, for all $k > 0$, the k^{th} initiation of rollback and the k^{th} initiator are well-defined. A processor begins incarnation k after receiving the rollback message sent by the k^{th} initiator. Incarnation k is said to begin at time c, if the rollback of the k^{th} initiator is to c. Every processor i keeps the beginning times of incarnations in an array Beg_i. Let c_k be the beginning time of incarnation k.

$$\forall\, 1 \leq i \leq n,\ k > 0,\ Beg_i\,[k] = \begin{cases} c_k & \text{if } i \text{ has begun incarnation } k \\ \infty & \text{otherwise} \end{cases} .$$

Since the k^{th} initiator is well-defined, all processors have the same beginning times for their incarnation k; i.e., $\forall\, 1 \leq i, j \leq n$ and $k > 0$,

$$\text{if } Beg_i\,[k] \neq \infty \text{ and } Beg_j\,[k] \neq \infty \text{ then } Beg_i\,[k] = Beg_j\,[k].$$

Each processor i keeps the number of its current incarnation in a variable $incar_i$. The value of $incar_i$ is tagged to i's outgoing messages. The incarnation number and timestamp tagged to messages are used to identify orphans.

Lemma 3.4.1 *Let $[m, k, c]$ denote a message m tagged with incarnation number k and timestamp c that is sent to processor i. $[m, k, c]$ is an orphan to i upon its arrival, if and only if $\exists\, l > k,\ Beg_i\,[l] < c$.*

Proof: The if part: Suppose that there exists $l > k$, $Beg_i[l] < c$. Since the sender of $[m, k, c]$ sends the message at local time c during incarnation k, it will undo the sending of $[m, k, c]$ when it begins incarnation l at $Beg_i[l]$. Moreover, i has rolled back to $Beg_i[l]$ before m arrives. Thus, m is an orphan to i upon its arrival.

The only if part: Suppose that $[m, k, c]$ is an orphan upon its arrival. Let u be the arrival time of $[m, k, c]$ according to the global clock. By the definition of orphans, some processor initiates a rollback to $d < c$ before u such that (1) i rolls back to d before u, and (2) the sender of m rolls back to d after sending m. Let this initiation be the l^{th} initiation in the system. By (2), the sender of m begins incarnation l after sending m; hence, $l > k$. Moreover, by (1), i begins incarnation l at d before u. Thus, at u, $Beg_i[l] = d < c$. □

This lemma provides a means to identify orphans. Once a message is identified as an orphan, it is discarded. Therefore, there will be no repeated rollbacks.

3.4.2.3 Message Loss

To overcome message loss caused by rollbacks, processors save all the messages that they have received in a special buffer. (An optimization to reduce the number of messages saved is discussed in next chapter.) This special buffer is not affected by rollbacks. Therefore, even after rolling back, a processor can receive messages whose receipts it has undone.

After a processor i rolls back to time c, the special buffer contains two types of messages: those with timestamps less than c, and those with timestamps greater than or equal to c. The first type of messages are those whose receipts are not undone by i's rollback. Naturally, i should not receive any of these messages again after its rollback.

The second type of messages are those whose receipts are undone by i's rollback. Since other processors will also roll back to c, some of these messages become orphans after i's rollback. These orphans should be discarded. In contrast, the remaining non-orphans should be received again after i's rollback. This is because these non-orphans are exactly those messages that would be lost in the rollbacks, if they have not been saved in the special buffer.

Let $MsgQ_i$ denote processor i's buffer into which all messages newly arrived at i are placed. Since processor i already discards orphans in $MsgQ_i$, it is convenient to use $MsgQ_i$ as the special buffer. This means that messages are not removed from $MsgQ_i$ even after they have been received. Every message in $MsgQ_i$ is *unmarked* when it first arrives. An unmarked message becomes *marked* after it is received by i. When i rolls back to time c, marked messages with timestamp greater than or equal to c become unmarked. This way, they may be either received again or discarded as orphans, after the rollback.

3.4.3 Summary

We have described our techniques to ensure that messages are received on-time in real systems. We now show how to integrate these techniques into one translation mechanism that runs program P in real systems.

Two kinds of messages are used in translation: simulated and rollback messages. A simulated message has the form $[m, k, c]$ with k and c being the respective incarnation number and timestamp tags of m. Simulated messages are sent by processors when they execute program $P = \{P_1, \cdots, P_n\}$. A rollback message has the form $[\text{rollback}, k, c]$, which indicates that the k^{th} initiator is rolling back to time c.

Each processor i keeps a simulated clock c_i, a set of simulated messages that are just received σ_i, a buffer of arrived but not received simulated messages Q_i, and a simulated state s_i. They correspond to the local clock, the sequence of received messages just received, the buffer of arrived but not received messages, and the processor state, of processor i in the ideal system, respectively. Just like their counterparts in the ideal system, initially, $c_i = 0$, and $\sigma_i = Q_i = \emptyset$, and s_i is an initial state.

Processor i's current simulated state s_i, its current simulated clock c_i, and the local program P_i determine the next action a that i will simulate; specifically, $a = P_i(s_i, c_i)$. The simulation of a proceeds as follows. First, i makes a checkpoint of s_i. Then c_i is incremented by 1. Next, if a is **receive**, then all messages are removed from Q_i and placed into σ_i; if a is "**send** m to all processors in J", then i sends a simulated message $[m, incar_i, c_i]$ to each processor in J, with $incar_i$ being the number of i's current incarnation and c_i being the current value of i's simulated clock. Then i changes to a new simulated state, which is determined by $\delta_i(s_i, a, \sigma_i)$.

To ensure that simulated messages are always received on-time, when

Table 3.1: Summary of variables in translation.

Name	Brief Description
c_i	simulated local clock
σ_i	sequence of simulated messages just received
Q_i	buffer of arrived but not received simulated messages
s_i	simulated state
$incar_i$	current incarnation number
$Beg_i[k]$	beginning time of incarnation k
$MsgQ_i$	buffer of arrived messages
$SavedState_i[c]$	simulated state saved when $c_i = c$

a simulated message first arrives at i, it is not placed into Q_i; it is placed into $MsgQ_i$ instead. On-time simulated messages are copied from $MsgQ_i$ to Q_i, when processor i calls procedure PrepareQ$_i$. Newly arrived rollback messages are placed in $MsgQ_i$ and processed in PrepareQ$_i$ also. Procedure PrepareQ$_i$ is called just before processor i decides the action that i will simulate next.

Procedure PrepareQ$_i$ first processes all the rollback messages and then all the unmarked simulated messages, in $MsgQ_i$. Rollback messages are processed in ascending order of their incarnation numbers; i.e., if [rollback, k, c] and [rollback, k', c'] are both in $MsgQ_i$, and $k < k'$, then [rollback, k, c] is processed before [rollback, k', c']. A rollback message [rollback, k, c] is processed as follows. If $k = incar_i + 1$ and $c_i \geq c$, then i rolls back to c and begins incarnation k. The message [rollback, k, c] is removed from $MsgQ_i$ and discarded. On the other hand, if $k > incar_i + 1$ or $c_i < c$, then i just leaves the message in $MsgQ_i$ for processing in subsequent calls to PrepareQ$_i$. This is because if $k > incar_i + 1$, then i has not yet received the rollback message from the $(incar_i + 1)^{st}$ initiator. And if $c_i < c$, then a rollback to c is not yet possible for i. After all the rollback messages in $MsgQ_i$ have been processed, i begins the processing of unmarked simulated messages.

An unmarked simulated message, $[m, k, c]$, is processed in procedure PrepareQ$_i$ as follows. It is considered an orphan if $\exists\ l > k,\ Beg_i[l] < c$. Orphans are removed from $MsgQ_i$ and discarded. If $[m, k, c]$ is not an orphan and $c < c_i$, then it is late and i will have to roll back. However, since i may yet process more late messages, i does not decide on the time of rollback until it has processed all the unmarked simulated messages in $MsgQ_i$. Therefore, the only action i takes when processing a late message is to save in the variable $RoTime$ the smallest timestamp of all late messages (that it has processed during this execution of procedure PrepareQ$_i$). Note that $[m, k, c]$ remains unmarked in $MsgQ_i$. If $[m, k, c]$ is not an orphan, is not late, and $c > c_i$, then it is early and is left in $MsgQ_i$ unmarked. Finally, if $[m, k, c]$ is not an orphan, not late, and not early, then it is on-time. An on-time $[m, k, c]$ becomes marked in $MsgQ_i$ and m is added to Q_i. At the end of processing all the unmarked simulated messages, if i has processed at least one late message, i initiates a rollback to time $RoTime$.

See Table 3.1 for a summary of the variables used in our translation. Note the correspondences between c_i, δ_i, Q_i, and s_i in this table and those in Figure 3.1. See also Figures 3.2–3.3 for an algorithmic description of the translation mechanism. The $*$'s in Figure 3.2 mark those statements that do not appear in the execution model of Figure 3.1. Also, one iteration of the while-loop in Figure 3.2 is called "one iteration of our translation mechanism".

3.4.4 Bounding the length of each iteration

As described so far, the number of active steps taken by a processor in one iteration of our translation mechanism may grow without bound. This is due to the possibility that a fast processor j can send messages to a slow processor i faster than i can process them. Thus, the number of messages i has to process in one iteration of the translation mechanism can grow without bound, which results in the number of active steps taken by i in one iteration growing without bound also.

A simple solution for the above problem is to use end-to-end flow-control [Tan81] to prevent fast processors from deluging slow processors with messages. This means that a fast processor will stop sending messages to a slow processor i, once the number of messages that i has to process reaches a threshold. Thus, we can keep the number of messages that a processor processes in one iteration bounded. This implies that the number of active steps a processor executes in one iteration is also bounded. This property will be useful in proving that the translation mechanism is guaranteed to

---

Begin

$c_i, \sigma_i, Q_i := 0, \emptyset, \emptyset;$

$s_i :=$ initial state;

* $incar_i$, $MsgQ_i := 0, \emptyset;$

* $\forall\ 1 \leq j < \infty$, $Beg_i[j] := \infty;$

while *true* **do**

* $SavedState_i[c_i] := s_i;$ { checkpoint old state }

* PrepareQ$_i$();

$a := P_i(s_i, c_i);$

$c_i := c_i + 1;$

{ simulation of action a }

if a = receive $\longrightarrow$ $\sigma_i, Q_i := Q_i, \emptyset;$

$\square$ a = "send m to all processors in J" $\longrightarrow$ send(J, $[m, incar_i, c_i]$);

$\square$ $a \neq$ receive and $a \neq$ send $\longrightarrow$ skip;

fi;

$s_i := \delta_i(s_i, a, \sigma_i);$

od;

End.

Figure 3.2: Processor i's translation of program P_i.

Procedure PrepareQ$_i$()

Begin

{ process rollback messages in $MsgQ_i$ }

for each rollback message [rollback, k, c] in $MsgQ_i$ **do**

if { ready to begin incarnation k }

$k = incar_i + 1$ and $c_i \geq c \longrightarrow$

remove [rollback, k, c] from $MsgQ_i$;

{ begin incarnation k }

$incar_i$, $Beg_i[k] := k$, c;

{ roll back }

$c_i, s_i := c, SavedState_i[c]$;

unmark simulated messages in $MsgQ_i$ with timestamps $\geq c$;

$\square$ { too soon to begin incarnation k }

$k > incar_i + 1$ or $c_i < c \longrightarrow$ skip;

fi;

od;

{ to be continued on the next page }

Figure 3.3: Procedure PrepareQ$_i$.

Figure 3.3 (continued)

{ process simulated messages in $MsgQ_i$ }

$RoTime := \infty$;

for each unmarked simulated message $[m, k, c]$ in $MsgQ_i$ **do**

if { $[m, k, c]$ is orphan; discard it }

$\exists\ l > k,\ Beg_i[l] < c \longrightarrow$ remove $[m, k, c]$ from $MsgQ_i$;

$\Box$ { $[m, k, c]$ is late; keeps track of smallest rollback time }

$\forall\ l > k,\ Beg_i[l] \geq c$ and $c < c_i \longrightarrow RoTime := \min(RoTime, c)$;

$\Box$ { $[m, k, c]$ is early; leave it to be received later }

$\forall\ l > k,\ Beg_i[l] \geq c$ and $c > c_i \longrightarrow$ skip;

$\Box$ { $[m, k, c]$ is on-time; receive it }

$\forall\ l > k,\ Beg_i[l] \geq c$ and $c = c_i \longrightarrow$ add m to Q_i;

mark $[m, k, c]$;

fi;

od;

{ initiate rollback to $RoTime$ }

if $RoTime \neq \infty \longrightarrow \forall\ 1 \leq j \leq n$, send($j$, [rollback, $incar_i$+1, $RoTime$]) **fi**;

End of PrepareQ$_i$.

---

make progress.

3.5 Proof of Progress

We now prove that when our translation mechanism runs the distributed program $P = \{P_1, \cdots, P_n\}$ in a real distributed system, the processors are guaranteed to make progress. The proof is developed as follows. We first define the *simulation histories* of P. Then we define what it means for processors to make progress in a simulation history of P. Finally, we show that processors make progress in every simulation history of P.

3.5.1 Simulation Histories

Let $H = \{h_1, \cdots, h_n\}$ be a history of an execution of the program P using our translation mechanism; and let Φ and Δ be the bounds on processor speeds and message transmission times in H, respectively. Note that for all $1 \leq i \leq n$, the execution in the local history h_i is a sequence of iterations of the translation mechanism. These iterations are called *simulated* steps. In one simulated step, exactly one simulated action is executed, and the simulated clock is incremented by 1 exactly once. Furthermore, as we have discussed in Section 3.4.4, there is a bound on the number of active steps executed in one simulated step. Let this bound be C. Thus, in H, every processor takes at least one simulated step every $\Phi^* = C \cdot \Phi$ global ticks.

To make our analysis simpler, we use an abstraction of local history h_i called *local simulation history* sh_i. For all simulated steps p in h_i, if processor i takes u ($1 \leq u \leq \Phi^*$) global ticks to execute p, then we assume that in sh_i, i takes only one global tick to execute p, and does nothing for the next $u-1$ global ticks; i.e., in sh_i, the simulated step p is followed by $u-1$ inactive steps. The relationship between h_i and sh_i is illustrated in Figure 3.4. The n local simulation histories form the *simulation history* $SH = \{sh_1, \cdots, sh_n\}$.

Let p be a simulated step of a processor i. We use the notations *p.action* to denote the simulated action executed by i in p, *p.incar* and *p.clock* to denote the incarnation number and the value of simulated clock of i at the end of p, respectively, and *p.msgq* to denote the buffer $MsgQ_i$ at the beginning of p.

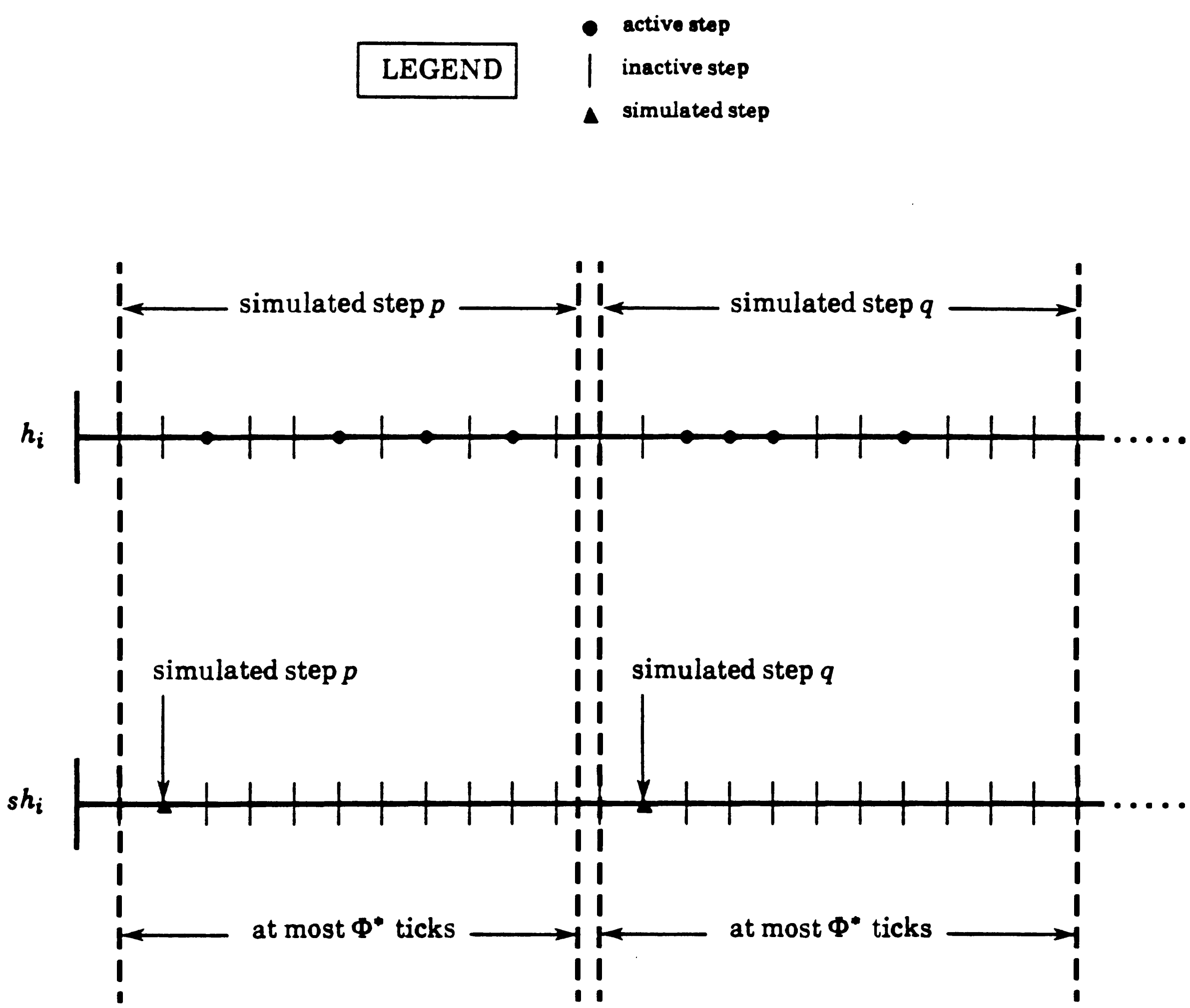

Figure 3.4: History h_i and its corresponding simulation history sh_i.

A processor i may roll back more than once during one simulated step. However, all these rollbacks are executed (in procedure PrepareQ$_i$) before i increments its simulated clock. Thus, if i executes one or more rollbacks during a simulated step p, then the last of these rollbacks must begin incarnation $p.incar$ at time $p.clock - 1$. This fact will be used often in our proofs.

A simulated step q *succeeds* another simulated step p in sh_i (or p *precedes* q), if q is after p, and there are no simulated steps between p and q.

Informally, a processor makes progress, if despite the processor's rollbacks, the number of simulated steps that it never undoes increases over time.

Definition 3.5.1 *Processor i is said to* make progress in *SH, if in SH, the number of simulated steps that i never undoes is infinite.*

3.5.2 Simulation Boundaries

Let SH be a simulation history in which processors take at least one simulated step every Φ^* global ticks, and the message transmission times are at most Δ global ticks. Our strategy to show that processors make progress in SH is roughly as follows. We first define the *simulation boundary* at global tick u, $B(u)$. We then show that at global tick u, every processor has already executed $B(u)$ simulated steps that will never be undone. Lastly, we show that $B(u)$ is an unbounded function of u. This will allow us to conclude that processors make progress in SH.

For all $u > 0$, let $p_i(u)$ be the last simulated step processor i takes by global tick u in SH.

Definition 3.5.2 *For all $u \geq 2(\Phi^* + \Delta)$, the* simulation boundary *at global tick u, $B(u)$, in SH is the largest integer T such that*

1. *For all $1 \leq i \leq n$, $T \leq p_i(u).clock$; and*
2. *The timestamps of all the simulated message sent in the global tick interval $[u + 1 - 2(\Phi^* + \Delta), u]$ are greater than or equal to T.*

If $u < 2(\Phi^* + \Delta)$, $B(u)$ is defined to be 1.

Let q and p be two simulated steps of a processor i such that q precedes p. If $q.clock \geq p.clock$ then i is said to *genuinely* roll back to $p.clock - 1$ in p.

Every rollback is caused directly or indirectly by the arrival of some late simulated message at the initiator of the rollback. More specifically, a rollback to time c is due to the arrival of a late simulated message with timestamp c. The following lemma shows that if a genuine rollback occurs at global tick u, the late simulated message that caused this rollback was sent during the global tick interval $[u + 1 - 2(\Phi^* + \Delta), u - 2]$.

Lemma 3.5.1 *For all $1 \leq i \leq n$ and $u \geq 2(\Phi^* + \Delta)$, let p be the simulated step that processor i takes at global tick u. If i genuinely rolls back to $p.clock - 1$ in p, then the simulated message that causes the rollback is sent in the global tick interval $[u + 1 - 2(\Phi^* + \Delta), u - 2]$.*

Proof: Let q be the simulated step preceding p in SH, and t be the global tick when i takes the step q. Since i takes at least one simulated step every Φ^* global ticks, $u - t \leq \Phi^*$.

Since i genuinely rolls back to $p.clock - 1$ in p, $p.clock \leq q.clock$. Let $[m, k, p.clock - 1]$ be the simulated message that causes i to roll back to time $p.clock - 1$. Also, let j be the initiator of this rollback. Suppose that $[m, k, p.clock - 1]$ is sent at global tick v. If $v \geq u - 1$, then the earliest global tick when $[m, k, p.clock - 1]$ would arrive at j is u, and the earliest global tick when i would receive j's rollback message is $u + 1$. This is too late for i to roll back at global tick u to $p.clock - 1$; hence, $v < u - 1$. Our remaining task is to show that $v \geq u + 1 - 2(\Phi^* + \Delta)$.

The message $[m, k, p.clock - 1]$ takes at most Δ global ticks to arrive at j. Moreover, j takes at least one simulated step in Φ^* global ticks. Therefore, j must initiate the rollback by global tick $v + \Phi^* + \Delta$; i.e., j sends the rollback message $[\text{rollback}, p.incar, p.clock - 1]$ to i by global tick $v + \Phi^* + \Delta$. By global tick $v + \Phi^* + 2\Delta$, $[\text{rollback}, p.incar, p.clock - 1]$ must arrive at i. There are two cases to consider, depending whether $[\text{rollback}, p.incar, p.clock - 1]$ arrives at i by step q or not.

Case 1: $[\text{rollback}, p.incar, p.clock - 1]$ arrives at i after step q. Then, $v + \Phi^* + 2\Delta > t$. Since, $u \leq t + \Phi^*$, $u < v + 2(\Phi^* + \Delta)$.

Case 2: $[\text{rollback}, p.incar, p.clock - 1]$ arrives at i by step q. Since i genuinely rolls back in p, $q.clock \geq p.clock$. Thus, the only reason why $[\text{rollback}, p.incar, p.clock - 1]$ did not cause i to roll back to time $p.clokc - 1$ during step q must be that when i processes $[\text{rollback}, p.incar, p.clock - 1]$ during q, it has not yet begun some incarnation $k < p_i.incar$. Let m_k

be the rollback message that causes i to begin incarnation k. Clearly, m_k did not arrive at i by step q; otherwise, i would process m_k before [rollback, $p.incar, p.clock - 1$] during q (recall that rollback messages are processed in ascending order of their incarnation numbers). However, since all initiations of rollbacks are serialized, m_k must be sent to i before [rollback, $p_i.incar, p.clock - 1$] is sent to i (on the global clock). Moreover, since it takes m_k at most Δ global ticks to arrive at i, and it did not arrive by step q, [rollback, $p_i.incar, p.clock - 1$] is sent at less than Δ global ticks before q. Therefore, $\Delta > t - (v + \Phi^* + \Delta)$. Since $u \leq t + \Phi^*$, $u < v + 2(\Phi^* + \Delta)$. □

The next lemma shows that if processor i increments its simulated clock at global tick $u + 1$, then after the increment, the simulated clock must be greater than the simulation boundary at global tick u.

Lemma 3.5.2 *For all $1 \leq i \leq n$ and $u \geq 2(\Phi^* + \Delta)$, if processor i takes $p_i(u + 1)$ at global tick $u + 1$, $p_i(u + 1).clock > B(u)$.*

Proof: If i does not rollback in $p_i(u + 1)$, then clearly, $p_i(u + 1).clock = p_i(u).clock + 1$. Hence, $p_i(u + 1).clock > B(u)$. If i genuinely rolls back to $p_i(u + 1).clock - 1$ in $p_i(u + 1)$, then by Lemma 3.5.1, $p_i(u + 1).clock - 1$ is greater than or equal to $B(u)$. Thus, $p_i(u + 1).clock > B(u)$. Finally, if i rolls back in $p_i(u + 1)$ but not genuinely, then $p_i(u + 1).clock > p_i(u).clock$. Thus, $p_i(u + 1).clock > B(u)$. □

Corollary 3.5.1 *For all $1 \leq i \leq n$ and $u \geq 2(\Phi^* + \Delta)$, $p_i(u + 1).clock \geq B(u)$.*

Proof: If $p_i(u + 1) = p_i(u)$, the lemma is obviously true. However, if $p_i(u + 1) \neq p_i(u)$, then $p_i(u + 1)$ must be taken by processor i at global tick $u + 1$. By Lemma 3.5.2, $p_i(u + 1).clock > B(u)$. □

Now we can show that $B(u)$ is a monotonic and non-decreasing function of u.

Lemma 3.5.3 *For all $u \geq 2(\Phi^* + \Delta)$, $B(u + 1) \geq B(u)$.*

Proof: To show that $B(u+1) \geq B(u)$, we must verify that (1) for all $1 \leq i \leq n$, $p_i(u+1).clock \geq B(u)$; and (2) the timestamp of every simulated message sent in the global tick interval $[u+2-2(\Phi^*+\Delta), u+1]$ is greater than or equal to $B(u)$. Condition (1) is satisfied by Corollary 3.5.1. Now consider a simulation message $[m, k, c]$ that is sent in $[u+2-2(\Phi^*+\Delta), u+1]$. There are two cases to consider.

Case (1): $[m, k, c]$ is sent in $[u+2-2(\Phi^*+\Delta), u]$. Then $[m, k, c]$ is sent in $[u+1-2(\Phi^*+\Delta), u]$. Hence, $c \geq B(u)$.

Case (2): $[m, k, c]$ is sent at global tick $u+1$. By Lemma 3.5.2, $c > B(u)$.
□

Next, we show that processors' simulated clocks will never be smaller than $B(u)$ after global tick u.

Corollary 3.5.2 *For all* $1 \leq i \leq n$ *and* $v > u \geq 2(\Phi^* + \Delta)$, $p_i(v).clock \geq B(u)$.

Proof: By Lemma 3.5.3, for all $v > u$, $B(v) \geq B(u)$. Since $p_i(v).clock \geq B(v)$, $p_i(v).clock \geq B(u)$. □

By the definition of $B(u)$, processors' simulated clocks at global tick u are greater than or equal to $B(u)$. Hence, by global tick u, all processors have executed at least $B(u)$ simulated steps; i.e., by global tick u, every processor has executed one simulated step at simulated time 1, another step at simulated time 2, $\cdots$, and another step at simulated time $B(u)$. By Corollary 3.5.2, after global tick u, the simulated clocks are never smaller than $B(u)$. Thus, at global tick u, all the processors have executed $B(u)$ simulated steps that will never be undone.

Next, we show that $B(u)$ is an unbounded function of u. This is the last step in our proof that processors make progress in the simulation history SH.

Lemma 3.5.4 *For all* $u \geq 2(\Phi^* + \Delta)$, $B(u + 2(\Phi^* + \Delta)) > B(u)$.

Proof: To show that $B(u+2(\Phi^*+\Delta)) > B(u)$, we need to verify that (1) for all $1 \leq i \leq n$, $p_i(u+2(\Phi^*+\Delta)).clock > B(u)$; and (2) the timestamp of every simulated message sent in global tick interval $[u+1, u+2(\Phi^*+\Delta)]$ is greater than $B(u)$. Since every processor takes at least one simulated step in Φ^* global ticks, every processor has taken at least one simulated step in $[u+1, u+2(\Phi^*+\Delta)]$. Both (1) and (2) will be satisfied if we can show that for all $1 \leq i \leq n$ and for all the simulated steps q_i that processor i takes in $[u+1, u+2(\Phi^*+\Delta)]$, $q_i.clock > B(u)$. Let q_i be a step that i takes at a global tick v in the interval $[u+1, u+2(\Phi^*+\Delta)]$. By Lemma 3.5.3, $B(v-1) \geq B(u)$. Since $q_i = p_i(v)$, by Lemma 3.5.2, $q_i.clock > B(v-1)$, Therefore, $q_i.clock > B(u)$. □

Theorem 3.5.1 *For all* $1 \leq i \leq n$*, processor* i *makes progress in the simulation history* SH*.*

Proof: We need to show that for all $1 \leq i \leq n$, in SH, the number of simulated steps that processor i never undoes in SH is infinite. According to Corollary 3.5.2, by global tick u, i has executed at least $B(u)$ simulated steps that are never undone in SH. By Lemma 3.5.4, $B(u)$ grows without bound as u goes to ∞. Therefore, in SH, the number of simulated steps that i never undoes is infinite. □

Note that the minimum simulated time and the minimum timestamp of a simulated message are both 1. Hence, $B(2(\Phi^*+\Delta)) \geq 1$. From Lemma 3.5.4, we can derive that

$$\begin{gathered} B(2(\Phi^*+\Delta)) \geq 1, \\ B(4(\Phi^*+\Delta)) \geq 2, \\ \vdots \\ B(2l(\Phi^*+\Delta)) \geq l, \\ \vdots \\ \text{etc.} \end{gathered}$$

Combining this with Lemma 3.5.3 yields:

$$\forall\, l \geq 1 \text{ and } 2l(\Phi^*+\Delta) \leq u < 2(l+1)(\Phi^*+\Delta),\ B(u) \geq l.$$

Consequently, every $2(\Phi^*+\Delta)$ global ticks in the simulation history SH, each processor takes at least one simulated step that is never undone . Since a processor requires only 1 global tick to take the corresponding active step in the ideal system, the execution of the program P by our translation mechanism is up to $2(\Phi^* + \Delta)$ times slower in SH. However, since some processor in SH takes only one active step every Φ global ticks, a factor of Φ slowdown is inherent. The factor of slowdown that is due to our translation is actually $2(\Phi^* + \Delta)/\Phi$.

Recall that $\Phi^* = C \cdot \Phi$ where C is a constant. Therefore, as Φ and Δ decrease, the slowdown factor $2(\Phi^* + \Delta)$ becomes smaller. Consequently, the cost of using our translation to run programs in a real distributed system as measured by the slowdown factor becomes cheaper, as the performance of the real system approaches that of the ideal system.

3.6 Proof of Correctness

Let $P = \{P_1, \cdots, P_n\}$ be a program designed to meet a logical specification L in the ideal system. We now prove that our translation mechanism can run P correctly in any real distributed system; i.e., that with the translation mechanism, P can still meet L in any real system. The proof is developed as follows. First, we define the *normalized simulation histories* of P. Then we show that every normalized simulation history is logically equivalent to an execution of P in the ideal system. Thus, the normalized simulation histories meet the same logical specification L that is met by the executions of P in the ideal system.

Let $SH = \{sh_1, \cdots, sh_n\}$ be a simulation history. A simulated step p in sh_i is *orphaned*, if it is before another simulated step q in sh_i such that $p.clock \geq q.clock$. The actions simulated in orphaned steps are undone subsequently by rollbacks. A simulation history is *normalized* by removing its orphaned and inactive steps.

Let $SH = \{sh_1, \cdots, sh_n\}$ be the resulting simulation history when we use our translation mechanism to run program P in a real distributed system. Moreover, let its corresponding normalized history be $N(SH) = \{N(sh_1), \cdots, N(sh_n)\}$. $N(SH)$ consists of all the simulated steps of SH that are never undone. We want to show that $N(SH)$ is "logical equivalent" to a history G that is an execution of program P in the ideal system. The notion of logical equivalency between normalized simulation histories and executions is a simple extension of the notion of logical equivalency between histories that is defined in Section 3.3.2. $N(SH)$ is *logically equivalent* to an execution G of P, if for all $1 \leq i \leq n$, (1) processor i's initial simulated state in $N(SH)$ and its initial state in G are the same; and (2) the sequence of actions simulated in $N(SH)$ is the same as the sequence of actions executed in G.

Our proof strategy is as follow. First, we show that a processor i's simulated clock increases by 1 in every simulated step in $N(SH)$. Next, we show that in $N(SH)$, a message is sent at simulated time c if and only if it is received at simulated time $c+1$. Then, we show that an execution of P in the ideal system, G, can be constructed from $N(SH)$; and lastly, we argue that G is logically equivalent to $N(SH)$.

Lemma 3.6.1 *For all $1 \leq i \leq n$ and $l \geq 1$, the simulated clock at the end of the l^{th} step of $N(sh_i)$ is equal to l.*

Proof: By contradiction. Let p be the first step in $N(sh_i)$ such that p is the l^{th} step of $N(sh_i)$, and $p.clock \neq l$. Since no steps in $N(sh_i)$ are orphaned, $p.clock > l$, and for all steps q after p in $N(sh_i)$, $q.clock > p.clock > l$. Thus, for all steps q in $N(sh_i)$, $q.clock \neq l$.

By the results established in Section 3.5, processor i's simulated clock is guaranteed to increase unboundedly in sh_i. Also, i increments its simulated clock by at most 1 in one simulated step of sh_i. Thus, there exists at least one simulated step q in sh_i such that $q.clock = l$. However, since q cannot be in $N(sh_i)$, q is orphaned in sh_i.

Let q' be a simulated step after q in sh_i such that $q'.clock \leq q.clock$. Since i's simulated clock increases unboundedly in sh_i, we can assume that $q'.clock = q.clock = l$. However, q' cannot be in $N(sh_i)$; it is orphaned in sh_i.

Therefore, by induction, there are infinite number of orphaned simulated steps q in sh_i such that $q.clock = l$. This contradicts our results that simulated clocks increase unboundedly in sh_i. $\Box$

Thus, for all $1 \leq i \leq n$, processor i's simulated clock is incremented by exactly 1 in every step of $N(sh_i)$.

In the next two lemmas, we show the following property: in $N(SH)$, processor j sends a simulated message to processor i in the c^{th} step of $N(sh_j)$, if and only if i receives the message in the $(c+1)^{st}$ step of $N(sh_i)$. It follows from Lemma 3.6.1 that this property corresponds to that in the ideal system, a message sent at local time c (on the sender's clock) is received at local time $c+1$ (on the receiver's clock).

Lemma 3.6.2 *For all $1 \leq i \neq j \leq n$ and $c \geq 1$, if processor j sends a simulated message to processor i in the c^{th} step of $N(sh_j)$, i receives the message in the $(c+1)^{st}$ step of $N(sh_i)$.*

Proof: By contradiction. Let p_j be the c^{th} step of $N(sh_j)$, and let $[m,k,c]$ be the simulated message that processor j sends to processor i in p_j. Let p_i be the $(c+1)^{st}$ step of $N(sh_i)$. By Lemma 3.6.1, $p_j.clock = c$ and $p_i.clock = c+1$. Suppose that i does not receive $[m,k,c]$ in p_i.

Since the timestamp of $[m,k,c]$ is c, $[m,k,c]$ can only be received in a step q such that $q.clock = c+1$. Thus, if $[m,k,c]$ were received in $N(sh_i)$, it would be received in step p_i, which is impossible. Thus, $[m,k,c]$ is not received in $N(sh_i)$.

We claim that throughout sh_i, i will not consider $[m,k,c]$ an orphan. This is because if $[m,k,c]$ were an orphan to i in some step of sh_i, then j would roll back to before time c after sending $[m,k,c]$ to i in step p_j. Thus, p_j would be orphaned, contradicting the fact that p_j is in $N(sh_j)$.

Let p'_i be a step in sh_i such that $p'_i.msgq$ contains $[m,k,c]$. Without loss of generality, assume that for all steps q before p'_i in sh_i, $[m,k,c] \notin q.msgq$; i.e., $[m,k,c]$ arrives at i at the beginning of step p'_i. Note that by the above claim, for all steps q after p'_i in sh_i, $[m,k,c] \in q.msgq$. There are three cases to consider, depending on whether p_i is before p'_i, $p_i = p'_i$, or p_i is after p'_i.

Case 1: p_i is before p'_i. Since p_i is not orphaned in sh_i, $p'_i.clock > p_i.clock = c+1$. Thus, when $[m,k,c]$ arrives at i at the beginning of p'_i, it will be a late message and causes i to roll back to c. As a result, p_i is orphaned: a contradiction.

Case 2: $p_i = p'_i$. Since $[m,k,c]$ arrives at i at the beginning of p_i, it is an unmarked simulated message in $p_i.msgq$. Thus, i will process it in p_i. By the above claim, $[m,k,c]$ is not an orphan in p_i; hence, $[m,k,c]$ must be either late or early, or on-time in p_i. However, $p_i.clock$ has to be greater than $c+1$ for $[m,k,c]$ to be late, less than $c+1$ for $[m,k,c]$ to be early, and equal to $c+1$ for $[m,k,c]$ to be on-time. Since $p_i.clock = c+1$, $[m,k,c]$ is on-time and is received in p_i: a contradiction.

Case 3: p_i is after p'_i. In case 2, we prove that if $[m,k,c]$ is unmarked in $p_i.msgq$, it is received in p_i. Therefore, assume that $[m,k,c]$ is marked in $p_i.msgq$. Let p''_i be the simulated step preceding p_i in sh_i. Since $[m,k,c]$ is marked in $p_i.msgq$, $[m,k,c]$ is marked at the beginning of p_i. Thus, $[m,k,c]$ is marked by the end of p''_i. This means that there exists some step q_i before

p_i such that (1) $[m, k, c]$ is received in q_i (i.e., $q_i.clock = c + 1$), and (2) i does not roll back to undo q_i after q_i and before p_i. Hence, for all steps q after q_i and before p_i, $q.clock \geq c + 1$. In particular, $p_i''.clock \geq c + 1$.

Since $p_i.clock = c+1$, $p_i''.clock \geq p_i.clock$. Thus, i must roll back to time c in p_i. This rollback will make $[m, k, c]$ an unmarked message in $p_i.msgq$. Hence, i receives $[m, k, c]$ in p_i: a contradiction. □

Lemma 3.6.3 *For all $1 \leq i \neq j \leq n$ and $c \geq 1$, if processor i receives a simulated message from processor j in the $(c+1)^{st}$ step of $N(sh_i)$, then j sends the message in the c^{th} step of $N(sh_j)$.*

Proof: By contradiction. Let p_i be the $(c+1)^{st}$ step of $N(sh_i)$, and let $[m, k, c]$ be the simulated message that processor i receives from processor j in p_i. By Lemma 3.6.1, $p_i.clock = c + 1$. Suppose that j does not send $[m, k, c]$ in the c^{th} step of $N(sh_j)$.

Let p_j be the step in sh_j in which j sends $[m, k, c]$. Note that $p_j.incar = k$ and $p_j.clock = c$. Step p_j cannot be in $N(sh_j)$ because if p_j were in $N(sh_j)$, then by Lemma 3.6.1, it would have to be the c^{th} step of $N(sh_j)$. Thus, p_j is orphaned in sh_j.

Let p_i be the v^{th} step in sh_i. Since $\{sh_1, \cdots, sh_n\}$ is a simulation history, p_j is the u^{th} step of sh_j for some $u < v$. Since p_j is orphaned, there exists a simulated step p_j' after p_j in sh_j such that $p_j'.clock \leq p_j.clock$. Without loss of generality, assume that p_j' is the first step after p_j to make p_j an orphan. This means that in p_j, processor j rolls back and begins incarnation $p_j'.incar$ at $p_j'.clock - 1$. And since p_j' is after p_j, $p_j'.incar > p_j.incar$. There are two cases to consider, depending on whether $p_j'.incar \leq p_i.incar$ or not.

Case 1: $p_j'.incar \leq p_i.incar$. Since the beginning times of incarnation $p_j'.incar$ at processors i and j are the same, i must also begin incarnation $p_j'.incar$ at $p_j'.clock - 1$. However, since $p_j'.clock < p_j.clock = c < p_i.clock = c + 1$, i must begin incarnation $p_j'.incar$ before step p_i; otherwise, p_i would be orphaned. This means that at the beginning of p_i, $Beg_i\left[p_j'.incar\right] = p_j'.clock - 1$. Since $p_j'.incar > p_j.incar = k$ and $Beg_i\left[p_j'.incar\right] = p_j'.clock - 1 < c$, $[m, k, c]$ is an orphan to i in p_i. This contradicts that i receives $[m, k, c]$ in p_i.

Case 2: $p_j'.incar > p_i.incar$. Then i begins incarnation $p_j'.incar$ after p_i. Since $p_j'.clock < p_i.clock$, p_i is orphaned in sh_i: a contradiction. □

For all $1 \leq i \leq n$, a sequence of active steps τ_i is constructed from $N(sh_i)$ as follows: for all $c > 1$, (a_c, c) is the c^{th} step of τ_i where a_c is the simulated action executed in the c^{th} step of $N(sh_i)$. Let $s_i[0]$ be the initial simulated state in $N(sh_i)$, and g_i denote the pair $(s_i[0], \tau_i)$. Moreover, let G denote the n-tuple $\{g_1, g_2, \cdots, g_n\}$.

Theorem 3.6.1 *G is an execution of the distributed program P in the ideal system.*

Proof: The proof is in three parts. First, we show that G is a history. Second, we show that G satisfies program P; and third, we show that G is a history of the ideal system.

Part 1: we show that G is a history. By Lemmas 3.6.2 and 3.6.3, a processor j sends a message to processor i in the c^{th} step of g_j, if and only if for some $d > c$, i receives the message in the d^{th} step of g_i. Thus, it suffices to show that for all $1 \leq i \leq n$, g_i is a local history of processor i. Let $g_i = (s_i[0], \tau_i)$. Since $s_i[0]$ is an initial simulation state of i in $N(sh_i)$, it is an initial state of processor i. Furthermore, by Lemma 3.6.1, for all $c > 0$, the action a_c in the c^{th} step of $N(sh_i)$ occurs at simulated time c. Thus, $(a_1, 1)(a_2, 2)(a_3, 3)\cdots$ is a possible sequence of active steps that processor i takes at global ticks $1, 2, 3, \cdots$. Therefore, g_i is a local history of i.

Part 2: we show that G satisfies program P. For all $1 \leq i \leq n$ and $l > 0$, let

- $s_i[0]$ be i's initial state in g_i;
- $\sigma_i[0]$ be an empty set;
- $a_i[l]$ be the action executed in l^{th} step of g_i;
- $s_i[l]$ be i's state after $a_i[l]$ is executed; and
- $\sigma_i[l]$ be the set of messages received by i in $a_i[l]$.

Note that $s_i[0]$ corresponds to i's initial simulated state in $N(sh_i)$, and for all $l > 0$, $s_i[l]$ corresponds to i's simulated state after executing $a_i[l]$ in the l^{th} step of $N(sh_i)$. To show that G satisfies $P = \{P_1, \cdots, P_n\}$, we have to show for all $1 \leq i \leq n$ and $l > 0$, $a_i[l] = P_i(s_i[l-1], l-1)$, and $s_i[l] = \delta_i(s_i[l-1], a_i[l], \sigma_i[l])$. We proceed by induction on l.

Fix a $l > 0$. Assume that for all $0 < c \leq l$, $a_i[c] = P_i(s_i[c-1], c-1)$ and $s_i[c] = \delta_i(s_i[c-1], a_i[c], \sigma_i[c])$. We have to show that

(1) $a_i[l+1] = P_i(s_i[l], l)$, and

(2) $s_i[l+1] = \delta_i(s_i[l], a_i[l+1], \sigma_i[l+1])$.

Let p and q denote the l^{th} and $(l+1)^{st}$ steps of $N(sh_i)$, respectively. By Lemma 3.6.1, $q.clock = l+1$, and by the construction of g_i, $a_i[l] = p.action$ and $a_i[l+1] = q.action$, respectively. From Figure 3.2, it is clear than $a_i[l+1] = P_i(s_i', c')$, where s_i' and c' are i's simulated state and simulated clock, respectively, immediately after i exits from the procedure PrepareQ_i. Since $q.clock = c' + 1$, $c' = l$. To establish condition (1), we now consider two cases.

First case: p precedes q in the simulation history sh_i. Then s_i' is also i's simulated state after executing $p.action$. Hence, $s_i' = s_i[l]$ and $a_i[l+1] = P_i(s_i[l], l)$.

Second case: there are simulated steps between p and q in sh_i. Since these steps are not between p and q in $N(sh_i)$, they are orphaned in sh_i. Therefore, i must roll back to time $q.clock - 1$ in step q. Since q is the $(l+1)^{st}$ step of $N(sh_i)$, by Lemma 3.6.1, $q.clock - 1 = l$. Thus, i rolls back to time l in q; and in this rollback, i restores its simulated state back to $s_i[l]$. Hence, $s_i' = s_i[l]$ and $a_i[l+1] = P_i(s_i[l], l)$.

It is clear from our translation mechanism that i's simulated state after executing $a_i[l+1]$ in q is $\delta_i(s_i[l], a_i[l+1], \sigma_i[l+1])$. Hence, $s_i[l+1] = \delta_i(s_i[l], a_i[l+1], \sigma_i[l+1])$. Condition (2) is also established.

Part 3: we show that G is a history of the ideal system. By its construction, G contains no inactive steps. Moreover, by Lemmas 3.6.2 and 3.6.3, for all messages m that are sent in G, if processor j sends m to processor i in the c^{th} step of g_j, then i must receive m by the $(c+1)^{th}$ step of g_i. Therefore, the synchronicity of G is $(1,1)$ and G is a history of the ideal system. □

By the construction of G, it is obvious that G and $N(SH)$ are logically equivalent. Since G is an execution of program P in the ideal system, G meets the logical specification L. Therefore, the normalized simulation history $N(SH)$ also meets L.

3.7 Discussion

We have proposed a new approach to solving the class of problems with logical specifications in real distributed systems. With this approach, to solve a problem with logical specification, a programmer only needs to design a solution for the ideal distributed system. Thereafter, this solution can be run using our translation mechanism to solve the given problem in any real system. This approach simplifies the solving of problems with logical specifications because it is easier to coordinate the actions of cooperating processors in the ideal system, where processors execute in synchronized steps and all message take exactly one step to arrive, than in real systems, where processor speeds and message transmission times vary over time.

Several previous results are related to our work. Lundelius [Lun86] and Neiger and Toueg [NT86] independently proposed message buffering mechanisms that can be used to achieve the following result: Let P be a program that meets a logical specification L in a distributed system in which processors execute in synchronized steps. With their mechanisms, P will still meet L when run in a distributed system where processors do not execute in synchronized steps. This result can be achieved also using our translation mechanism; but the message buffering mechanisms of Lundelius and Neiger et al cannot be used to achieve the result of this chapter. However, Lundelius also considered how to handle processor failures with her mechanism; whereas we did not consider processor failures.

Like our translation mechanism, Awerbuch's *synchronizers* [Awe85] allow programs to be first designed for the ideal system, and then run correctly in real systems. All synchronizers employ the following basic strategy. Let P be a program that is designed for the ideal system. Processors execute P in rounds. At the beginning of a round, processors send messages to other processors (according to their states and P). Then they wait to receive all the messages that are sent to them at the beginning of the round. Finally, they change their states and proceed to the next round. Since processor speeds and message transmission times in a real system are not known a priori, a processor cannot use time-outs to determine whether another processor has sent it messages at the beginning of a particular round. Thus, with synchronizers, even when a processor i is not required by the program P to send messages in a round, it still has to send messages to other processors in that round so that they do not wait for i's messages indefinitely. This means that with synchronizers, the cost of running P (in terms of time and messages) in a real system is always higher than that of running P in the ideal system, even when the real system performs like the ideal system.

Given a program P that is designed for the ideal system, the choice of using synchronizers or our translation mechanism to run P in a real system depends on P and the real system. If program P requires infrequent message exchanges between processors, then using a synchronizer to run P will be expensive because the synchronizer will still require processors to send messages in every round. Hence, our translation will be more suitable to run P. However, if P requires frequent message exchanges between processors and it is run in a real system in which some processors are much faster than others, then it is pointless for the fast processors to proceed without waiting for messages from the slow processors. In this case, synchronizers are more suitable to run P. In general, if using our translation to run a program in a system results in many rollbacks, a synchronizer will be more suitable.

Jefferson [Jef85] suggested the following approach for distributed programming. Given a logical specification L, a programmer designs a program and assigns to each event of the program a *virtual time*. Moreover, the programmer has to ensure that the assignment of virtual times satisfies Lamport's clock conditions [Lam78]; i.e., if event a can causally affect event b, then the virtual time of a must be less than the virtual time of b. Then, if the program is guaranteed to meet the specification L when its events are executed in the order of their virtual times, then Jefferson's *time warp* mechanism guarantees that the same program will meet the same specification in any distributed system. In contrast, with our approach for distributed programming, one has to only write a program to meet L in the ideal system; one does not need to assign virtual times to events, and ensure that such an assignment satisfies Lamport's clock conditions.

Jefferson's time warp mechanism was based on a combination of checkpointing/rollback and message buffering. However, repeated rollbacks are possible with his checkpointing/rollback scheme.

Chapter 4

Optimization and Extension of Translation

4.1 Introduction

We now discuss two problems that were raised but not solved when we described our translation mechanism in the previous chapter. The first problem is how processors can avoid saving all the checkpoints that they make and all the simulated messages that they receive, for the entire course of translation. The second problem is how to extend our translation mechanism to handle concurrent initiations of rollbacks. These two problems are solved in Section 4.2 and Section 4.3, respectively.

4.2 Discarding Checkpoints and Messages

Processors do not have to keep all the checkpoints that they make and all the messages that they receive, for the entire course of translation. This optimization is possible because the simulation boundary has been proved to be monotonic and non-decreasing function of the global clock in Section 3.5. For instance, if the simulation boundary at global tick u is c, then, after u, processors will never roll back to simulated time less than c; hence, they can discard checkpoints that were made and messages that were received before simulated time c. Therefore, processors can discard checkpoints and messages based on their knowledge of the simulation boundary.

To compute the simulation boundary using the definition in Section 3.5, processors have to know Φ^*, the maximum number of global ticks that elapse between processors' successive simulated steps, and Δ, the maximum message transmission time. Since processors in real distributed systems do not usually have such knowledge, they cannot compute the simulation boundary. We will show, however, that processors can compute "reasonable" estimates of the simulation boundary without any knowledge of their execution speeds and message transmission times.

Reasonable estimates have two properties: correctness and non-triviality. Let $\hat{B}$ denote an estimate of the simulation boundary computed by some processor at global tick u. $\hat{B}$ is *correct*, if after u, the simulated clocks of all processors will always be greater than or equal to $\hat{B}$; i.e., if $\hat{B}$ is correct, at global tick u, all processors have executed $\hat{B}$ simulated steps that will never be undone. Thus, at global tick u, the checkpoints that a processor made and the messages that it received at simulated times $1, \cdots, \hat{B} - 1$ can be discarded.

Estimates of the simulation boundary are *non-trivial*, if they grow without bound over time on the global clock. With non-trivial estimates, we can guarantee that checkpoints and messages that can be discarded at global tick u are identifed at some later global tick.

In the following section, we describe a boundary estimation algorithm and prove that it computes correct and non-trivial estimates of the simulation boundary.

4.2.1 Boundary Estimation Algorithm

One processor is chosen to be the initiator of the boundary estimate algorithm. The initiator begins the algorithm by sending messages to all processors requesting them to compute *local estimates* of the simulation boundary. After receiving the request message, each processor i computes a local estimate of the simulation boundary $\hat{B}_i$, and sends this estimate to the initiator. Once the initiator has received local estimates from all the processors, it computes a *global estimate* of the simulation boundary, $\hat{B}$, which is defined to be $\min_{1 \leq i \leq n} \hat{B}_i$. Then, it sends $\hat{B}$ to all processors, each of which can discard checkpoints and messages according to $\hat{B}$.

The initiator invokes the boundary estimation algorithm periodically. Furthermore, the initiator assigns a unique number to each invocation.

Thus, different invocations of the algorithm can be distinguished. In the remainder of this section, we consider only one invocation of the algorithm.

Our discussion of the boundary estimation algorithm proceeds as follows. We first specify the properties of local estimates in Section 4.2.1.1. Using these properties, in Section 4.2.1.2, we show that $\hat{B}$ is a correct estimate of the simulation boundary. Then, in Section 4.2.1.3, we describe how the local estimates are computed and show that they indeed have the specified properties. Finally, in Section 4.2.1.4, we show that $\hat{B}$ is non-trivial.

4.2.1.1 Properties of local estimates

For all $1 \leq i \leq n$, let u_i denote the global tick when processor i computes $\hat{B}_i$, and $c_i(u_i)$ denote i's simulated clock at global tick u_i. $\hat{B}_i$ must have the following two properties:

1. $\hat{B}_i \leq c_i(u_i)$; and
2. For all $1 \leq j \leq n$, no message (simulated or rollback) that i sent to processor j before global tick u_i will cause j to roll back after global tick u_j to a simulated time before $\hat{B}_i$.

4.2.1.2 Correct Global Estimates

Consider an execution of our boundary estimate algorithm. For all $1 \leq i \leq n$, let u_i denote the global tick when processor i computes its local estimate $\hat{B}_i$, and u denote the global tick when the initiator computes the global estimate $\hat{B}$, during this execution. Note that $u_i < u$. Furthermore, consider the simulated steps that processor i ($1 \leq i \leq n$) takes after global tick u_i and in which i rolls back or initiates a rollback. Let S be the set of these steps. We order the steps in S by the global tick at which they occur such that for all $l \geq 1$, the global tick when the l^{th} step occurs is less than or equal to the global tick when the $(l+1)^{st}$ step occurs. See Figure 4.1 for an illustration of S. We first prove that if a processor rolls back or initiates a rollback to time c in a step in S, then $c \geq \hat{B}$. Then we show that the global estimate $\hat{B}$ computed by the initiator at global tick u is correct.

In the following proofs, we assume that at any given global tick, a processor may run either the boundary estimation algorithm or the translation mechanism, but not both.

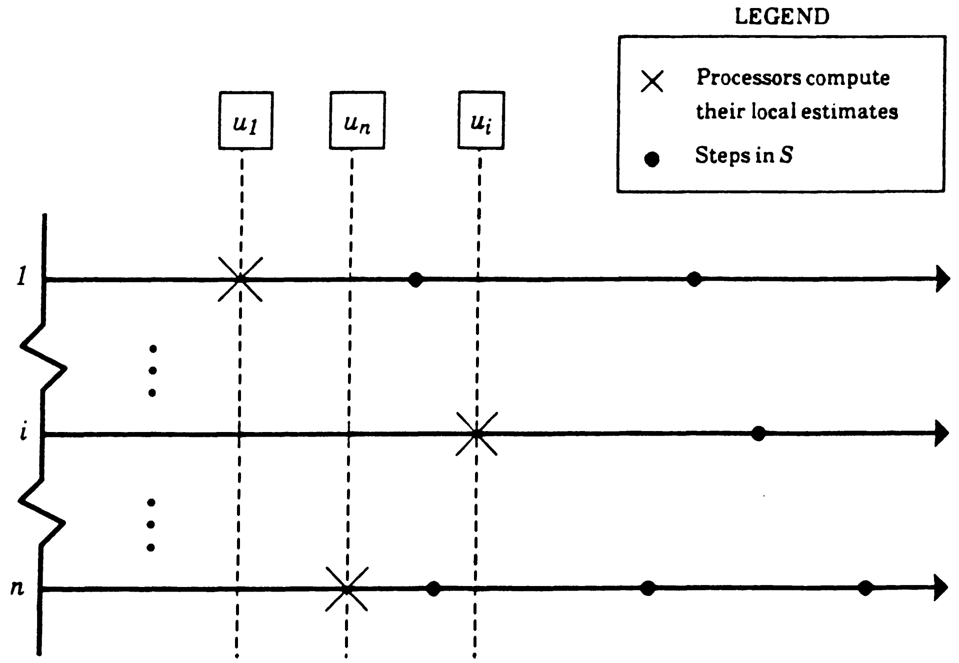

Figure 4.1: An example of set S.

Lemma 4.2.1 *Let $l \geq 0$ and p be the $(l+1)^{st}$ step in S. Suppose that for all steps q before p in S, if a processor rolls back or initiates a rollback to time d in q, $d \geq \hat{B}$. Then, a processor initiates a rollback to time c in step p, only if $c \geq \hat{B}$.*

Proof: Let $[m, k, c]$ be the late simulated message that causes a processor i to initiate a rollback to c in step p, and j be the sender of $[m, k, c]$. We need to show that $c \geq \hat{B}$. There are two cases to consider, depending on whether j sends $[m, k, c]$ before or after computing $\hat{B}_j$; i.e., before or after global tick u_j. Let w be the global tick at which i takes step p. Note that since $p \in S$, $u_i < w$ where u_i is the global tick at which i computes $\hat{B}_i$.

Case 1: j sends $[m, k, c]$ before global tick u_j. By the second property of $\hat{B}_j$, processor i cannot roll back to a simulated time before $\hat{B}_j$ after global tick u_i. Thus, $c \geq \hat{B}_j$. And since $\hat{B}_j \geq \hat{B}$, $c \geq \hat{B}$.

Case 2: j sends $[m, k, c]$ after global tick u_j. Since processor i processes $[m, k, c]$ at global tick w, processor j must send $[m, k, c]$ during the global tick interval $[u_j + 1, w - 1]$. We claim that j's simulated clock c_j in $[u_j + 1, w - 1]$ is always greater than or equal to $\hat{B}$. If j does not roll back in $[u_j + 1, w - 1]$, then for all $u_j + 1 \leq u' \leq w - 1$, $c_j(u') \geq c_j(u_j)$. By the first property of $\hat{B}_j$, $c_j(u_j) \geq \hat{B}_j$; hence, $c_j(u') \geq \hat{B}_j \geq \hat{B}$. Therefore, assume that j rolls back in $[u_j + 1, w - 1]$. By the assumption of the lemma, when j rolls back in $[u_j + 1, w - 1]$, it rolls back only to times that are greater than or equal to $\hat{B}$. Thus, for all $u_j + 1 \leq u' \leq w - 1$, $c_j(u') \geq \hat{B}$. The claim is proved. As a result, the timestamp of $[m, k, c]$ is greater than or equal to $\hat{B}$; i.e., $c \geq \hat{B}$. □

Lemma 4.2.2 *Let $l \geq 0$ and p be the $(l+1)^{st}$ step in S. Suppose that for all steps q before p in S, if a processor rolls back or initiates a rollback to time d in q, $d \geq \hat{B}$. Then, a processor rolls back to time c in step p, only if $c \geq \hat{B}$.*

Proof: Let [rollback, k, c] be the rollback message that causes a processor i to roll back to c in p, and j be the sender of [rollback, k, c]. We need to show that $c \geq \hat{B}$. There are two cases to consider, depending on whether j sends [rollback, k, c] before or after computing $\hat{B}_j$; i.e., before or after global tick u_j.

Case 1: j sends [rollback, k, c] before global tick u_j. This proof is the same as that of case 1 in the proof of Lemma 4.2.1.

Case 2: j sends [rollback, k, c] after global tick u_j. Let w be the global tick at which i takes step p. Since [rollback, k, c] causes i to roll back in p, j must send [rollback, k, c] during the global tick interval $[u_j + 1, w - 1]$; i.e., j initiates the rollback to time c in $[u_j + 1, w - 1]$. Let q be the simulated step in which j initiates the rollback. Since j takes q in $[u_j + 1, w - 1]$, $q \in S$. Moreover, step q is before step p; thus, by the assumption of the lemma, the rollback that j initiates in q must be to a time greater than or equal to $\hat{B}$; i.e., $c \geq \hat{B}$. □

Corollary 4.2.1 *For all steps $p \in S$, if a processor rolls back to or initiates a rollback to time c in p, then $c \geq \hat{B}$.*

Proof: By induction. Lemmas 4.2.1 and 4.2.2 provide the induction argument. □

Now we can show that the global estimate $\hat{B}$ is correct.

Theorem 4.2.1 *The global estimate of the simulation boundary $\hat{B}$ is correct.*

Proof: Recall that the initiator computes $\hat{B}$ at global tick u. We need to show that the simulated clock of every processor is always greater than or equal to $\hat{B}$ after global tick u; i.e., for all $1 \leq j \leq n$ and $v > u$, $c_j(v) \geq \hat{B}$. Consider a processor j and a global tick $v > u$. If j does not roll back during the global tick interval $[u_j + 1, v]$, then obviously, $c_j(u_j) \leq c_j(v)$. And by the first property of $\hat{B}_j$ that $\hat{B}_j \leq c_j(u_j)$, $c_j(v) \geq \hat{B}_j \geq \hat{B}$. Therefore, assume that j rolls back during the global tick interval $[u_j + 1, v]$. However, by Corollary 4.2.1, j cannot roll back to a simulated time before $\hat{B}$ in $[u_j + 1, v]$; hence, $c_j(v) \geq \hat{B}$. □

Theorem 4.2.1 guarantees that it is "safe" to use the global estimates of the simulation boundary computed by our algorithm to discard checkpoints and messages.

```
Procedure PrepareQ_i()
Begin
{ process rollback messages in MsgQ_i }
   for each rollback message [rollback, k, c] in MsgQ_i do
     if { ready to begin incarnation k }
        k = incar_i + 1 and c_i ≥ c ⟶
*          acknowledge [rollback, k, c];
           remove [rollback, k, c] from MsgQ_i;
           { begin incarnation k }
           incar_i, Beg_i[k] := k, c;
           { roll back }
           c_i, s_i := c, SavedState_i[c];
           unmark simulated messages in MsgQ_i with timestamps ≥ c;
     [] { too soon to begin incarnation k }
        k > incar_i + 1 or c_i < c ⟶ skip;
     fi;
   od;
   { to be continued on the next page }
```

Figure 4.2: Modified Procedure PrepareQ$_i$.

Figure 4.2 (continued)

```
{ process simulated messages in MsgQ_i }
   RoTime := ∞;
   for each unmarked simulated message [m, k, c] in MsgQ_i do
*      acknowledge [m, k, c];
       if { [m, k, c] is orphan; discard it }
          ∃ l > k, Beg_i[l] < c ⟶ remove [m, k, c] from MsgQ_i;
       [] { [m, k, c] is late; keeps track of smallest rollback time }
          ∀ l > k, Beg_i[l] ≥ c and c < c_i ⟶ RoTime := min(RoTime, c);
       [] { [m, k, c] is early; leave it to be received later }
          ∀ l > k, Beg_i[l] ≥ c and c > c_i ⟶ skip;
       [] { [m, k, c] is on-time; receive it }
          ∀ l > k, Beg_i[l] ≥ c and c = c_i ⟶ add m to Q_i;
              mark [m, k, c];
       fi;
   od;
   { initiate rollback to RoTime }
   if RoTime ≠ ∞ ⟶ ∀ 1 ≤ j ≤ n, send(j, [rollback, incar_i+1, RoTime]) fi;
End of PrepareQ_i.
```

4.2.1.3 Computation of local estimates

To compute local estimates of the simulation boundary, processors have to acknowledge the simulated and rollback messages that they receive. When a processor processes a simulated message for the first time, it sends an acknowledgement to the sender of the message; and when a processor rolls back due to a rollback message, it sends an acknowledgement to the sender of the rollback message. This results in a small change of Procedure PrepareQ$_i$. These changes are marked by $*$ in Figure 4.2.

Every processor i keeps a list of unacknowledged messages UA_i. Its local estimate of the simulation boundary $\hat{B}_i$ is based on its simulated clock c_i and UA_i; i.e., $\hat{B}_i \equiv \min(c_i, U_i)$ where

$$U_i = \begin{cases} \infty & \text{if } UA_i = \emptyset \\ \min\{c \mid \exists\ [m,k,c] \text{ or } [\text{rollback}, k, c] \text{ in } UA_i\} & \text{otherwise} \end{cases} .$$

The intuition behind U_i is as follows. Our algorithm will guarantee that for all $1 \leq j \leq n$, if processor j sends an acknowledgement to i after it has computed $\hat{B}_j$, then i computes $\hat{B}_i$ (at some global tick u_i) before receiving the acknowledgement. Given this guarantee, among all the messages that i sent before global tick u_i, only the unacknowledged ones at u_i can cause other processors to roll back after they have already computed their local estimates. Thus, for all $1 \leq j \leq n$, after processor j has computed $\hat{B}_j$, U_i is the minimum time to which j may roll back due to messages in UA_i.

To satisfy the above guarantee, we require that after a processor has received a request from the initiator to compute a local estimate, and before it receives the global estimate from the initiator, it piggybacks the initiator's request on every message (rollback, simulated, and acknowledgement) that it sends. When the request piggybacked on a message, say m, arrives at a processor i, if i has not yet received the request, i will compute its local estimate $\hat{B}_i$ before processing the message m; on the other hand, if i has already received the request, it simply discards the piggybacked request.

Our description of our simulation boundary algorithm is now complete. The algorithm is illustrated in Figure 4.3. We now show that $\hat{B}_i$ has the properties specified in Section 4.2.1.1.

Consider an execution of our boundary estimate algorithm. For all $1 \leq i \leq n$, let u_i denote the global tick when processor i computes its local

Initiator:

while true **do**

$\forall\ 1 \leq i \leq n$, send(i, request to compute $\hat{B}_i$);

$\forall\ 1 \leq i \leq n$, receive(i, $\hat{B}_i$);

$\hat{B} := \min_{1 \leq i \leq n} \hat{B}_i$;

$\forall\ 1 \leq i \leq n$, send(i, $\hat{B}$);

od.

All processors i:

Upon receipt of a request to compute $\hat{B}_i$ **do**

$$U_i = \begin{cases} \infty & \text{if } UA_i = \emptyset \\ \min\{c \mid \exists\ [m, k, c] \text{ or } [\text{rollback}, k, c] \text{ in } UA_i\} & \text{otherwise} \end{cases}$$

$\hat{B}_i := \min(c_i, U_i)$;

send(*initiator*, $\hat{B}_i$);

od.

Figure 4.3: Boundary Estimation Algorithm.

estimate $\hat{B}_i$, and u denote the global tick when the initiator computes the global estimate $\hat{B}$, during this execution.

Lemma 4.2.3 *For all $1 \leq i \leq n$, $\hat{B}_i$ has the following two properties:*

1. *$\hat{B}_i \leq c_i(u_i)$; and*
2. *For all $1 \leq j \leq n$, no message (simulated or rollback) that i sent to processor j before global tick u_i will cause j to roll back after global tick u_j to a simulated time before $\hat{B}_i$.*

Proof: Fix a processor i. Since $\hat{B}_i = \min(c_i, U_i)$, property (1) is easily satisfied. To prove property (2), suppose that a message M sent by i before global tick u_i causes a processor j to roll back after global tick u_j to a simulated time c. Note that M can be either a simulated message $[m, k, c]$ or a rollback message $[\text{rollback}, k, c]$. We need to show that $c \geq \hat{B}_i$.

Let u^* be the global tick when j receives the global estimate of the simulation boundary from the initiator. Note that for all $1 \leq l \leq n$, $u_l < u^*$. Furthermore, let v denote the global tick when M causes j to roll back to c; i.e., j sends an acknowledgement for M to i at v. If $u^* < v$, then M is unacknowledged at global tick u_i when i computes $\hat{B}_i$; thus, $c \geq \hat{B}_i$.

Suppose then that $u^* > v$. (Since we have assumed that at no global tick does a processor execute both the boundary estimation algorithm and the translation mechanism, the case when $u^* = v$ is impossible.) Since $u_j < v$ and j is required to piggyback the initiator's request on every message that j sends between global ticks u_j and u^*, the initiator's request is piggybacked on the acknowledgement for M. This piggybacked request ensures that i computes $\hat{B}_i$ before processing the acknowledgement; thus, $c \geq \hat{B}_i$. □

4.2.1.4 Non-trivial Boundary Estimates

We now prove that the global estimates of the simulation boundary are non-trivial by showing that successive global estimates will grow without bound. For this analysis, we assume that processor speeds and message transmission times are bounded by Φ and Δ, respectively, when the translation mechanism is executed. Moreover, we assume that processors take at least one simulated step every Φ^* global ticks, where Φ^* is a constant multiple of Φ. (The same Φ, Φ^*, and Δ have been used previously in Section 3.5.) To begin, note that in our boundary estimation algorithm, $\hat{B}$ is assigned $\min_{1 \le i \le n} \hat{B}_i$. Thus, it is sufficient to show that for all $1 \le i \le n$, the successive local estimates computed by processor i grow without bound.

Fix a processor i. Recall that $\hat{B}_i = \min(c_i, U_i)$. Since by the results of Section 3.5, c_i will grow without bound over time on the global clock, it suffices to show that U_i is an unbounded function of global clock. Let $U_i(u)$ denote the value of U_i at global tick u, and recall that $B(u)$ denotes the simulation boundary at global tick u.

Lemma 4.2.4 *For all* $u > 3(\Phi^* + \Delta)$, $U_i(u) \ge B(u - 3(\Phi^* + \Delta))$.

Proof: Fix a global tick u. Without loss of generality, assume that $U_i(u)$ is finite. There are two cases to consider.

Case 1: $U_i(u) = c$ where $[m, k, c]$ is a simulated message that processor i has sent to but not received an acknowledgement from processor j by global tick u. We need to show that $c \ge B(u - 3(\Phi^* + \Delta))$. Let p be the simulation step in which i sends $[m, k, c]$ to j, and v be the global tick when p occurs. Note that $p.clock = c$, and that by the definition of the simulation boundary, $B(v) \le p.clock$.

Message $[m, k, c]$ must arrive at j by global tick $v + \Delta$. Moreover, since j executes at least one simulated step in Φ^* ticks, j acknowledges $[m, k, c]$ at the latest by global tick $v + \Phi^* + \Delta$. Thus, by global tick $v + 2(\Phi^* + \Delta)$, i receives the acknowledgement for $[m, k, c]$. But $[m, k, c]$ is unacknowledged at global tick u; hence, $v + 2(\Phi^* + \Delta) > u$. This leads to the following derivation:

$$\begin{array}{rrcll} & v & > & u - 2(\Phi^* + \Delta) & \\ \Rightarrow & B(v) & \ge & B(u - 2(\Phi^* + \Delta)) & \text{since } B \text{ is non-decreasing} \\ \Rightarrow & p.clock & \ge & B(u - 2(\Phi^* + \Delta)) & \text{since } p.clock \ge B(v) \\ \Rightarrow & c & \ge & B(u - 3(\Phi^* + \Delta)). & \end{array}$$

Case 2: $U_i(u) = c$ where [rollback, k, c] is a simulated message that i has sent to but not received an acknowledgement from j by global tick u. We need to show that $c \ge B(u - 3(\Phi^* + \Delta))$. Let q be the simulation step in which j sends the acknowledgement to [rollback, k, c], and v be global tick when q occurs. This means that j rolls back to time c in q. Thus, $q.clock \le c + 1$. Since $q.clock \ge B(v)$, $c + 1 \ge B(v)$. The acknowledgement of [rollback, k, c] must arrive at processor i by global tick

$v + \Phi^* + \Delta$. And since [rollback, k, c] is unacknowledged at global tick u, $v + \Phi^* + \Delta > u$. Thus, $c + 1 \geq B(v) \geq B(u - \Phi^* - \Delta)$. By Lemma 3.5.4, $B(u - \Phi^* - \Delta) \geq 1 + B(u - 3(\Phi^* + \Delta))$; thus, $c \geq B(u - 3(\Phi^* + \Delta))$. □

Since $B(u) \to \infty$ as $u \to \infty$, Lemma 4.2.4 implies that U_i is an unbounded function of global ticks. Hence, our boundary estimates are non-trivial.

4.2.2 Discussion

Our boundary estimate algorithm is similar to the algorithm developed by Samadi [Sam85] to estimate the *global virtual time* during an execution of the *time warp* mechanism [Jef85]. Our algorithm requires that $3n$ messages be exchanged to compute and distribute one global estimate of the simulation boundary. Alternatively, one can use the consistent global state detection algorithm developed by Chandy and Lamport [CL85] to estimate the simulation boundary. This would require $n^2 + 2(n - 1)$ messages to compute and distribute one global estimate of the simulation boundary. The difference in the message overheads of the two algorithms is due to the fact that Chandy and Lamport's algorithm uses special messages to flush all the channels between all processors; whereas our algorithm uses acknowledgements and piggybacked requests and no flushing of channels.

4.3 Complete Version of Translation Mechanism

Call the translation mechanism described in the previous chapter our *basic* translation mechanism. In this section, we extend our basic translation mechanism to handle concurrent initiations of rollbacks. Recall that two initiations of rollbacks are concurrent, if each initiator starts rolling back before it receives a rollback message from the other initiator. The correctness of the basic mechanism depends on the fact that all initiations of rollbacks are serially ordered (see Section 3.4.2.2). Thus, one way to extend the basic translation mechanism is to serialize all concurrent initiations of rollbacks.

There are two general approaches to the serialization problem (more commonly known as the concurrency control problem). One is by locking and the other is by timestamping. Readers are referred to [BHG87] for a detailed discussion of locking and timestamping; we give just a cursory description here.

Locking can be used to serialize accesses to shared objects as follows. Every shared object is associated with a lock. A processor i seeking access to a shared object x must first acquire the lock associated with x. If another processor j already holds the lock, then j must release the lock before i can acquire the lock. This way, only one processor can hold the lock of object x at a time, and the accesses to x are serialized.

With timestamping, every access to a shared object is assigned a unique timestamp. Timestamps are totally ordered. An access with timestamp u to an object x is granted only if accesses with timestamps less than u to x have completed. Therefore, accesses to object x are totally ordered (serialized).

We will use locking instead of timestamping to serialize initiations of rollbacks, since locking requires fewer changes to our basic translation mechanism. Locking is incorporated into our basic translation mechanism as follows. In the basic mechanism, when a processor i in incarnation $incar_i$ initiates a rollback to time c, it simply sends the rollback message [rollback, $incar_i + 1, c$] to all the processors. In the complete mechanism, i will have to call a procedure NewIncarNo_i to determine the incarnation numbers of its rollback messages; i.e., when i initiates a rollback to time c, it first calls NewIncarNo_i to determine a new incarnation number $nincar$ and then sends [rollback, $nincar, c$] to all the processors.

A sketch of procedure NewIncarNo_i $(1 \leq i \leq n)$ is given in Figure 4.4. It is easy to verify that if the lock on *GIncar* provides processors exclusive access to *GIncar*, the following properties hold:

1. For all $1 \leq i, j \leq n$, if processors i and j call procedures NewIncarNo_i and NewIncarNo_j, respectively, they will receive different incarnation numbers in return; and

2. For all $1 \leq i \leq n$, the successive incarnation numbers that procedure NewIncarNo_i returns to processor i are strictly increasing.

The implementations of the object *GIncar* and the lock on *GIncar* are beyond the scope of this work. For a sample of the many possible im-

Procedure NewIncarNo$_i$()

Begin

{ *GIncar* is an integer object shared by all processors };

{ tmp_i is a local variable of processor i };

acquire_lock(*GIncar*);

increment(*GIncar*);

tmp_i := read(*GIncar*);

release_lock(*GIncar*);

return(tmp_i);

End.

Figure 4.4: Procedure NewIncarNo$_i$.

plementations, interested readers are referred to the work by Liskov and Scheifler on distributed programs [LS83], the work by Bernstein and Goodman on distributed databases [BG81], and the work by Birman et al. on distributed bulletin boards [BJS86].

Chapter 5

Conclusion

We have discussed two problems that complicate the design and verification of distributed programs: transient processor failures, and unknown variations of processor speeds and message transmission times. It is easier to derive and verify distributed programs with the simplifying assumptions that processors do not fail, and that processors execute in synchronized steps and messages take exactly one step to arrive.

In this thesis, we described a checkpointing/rollback mechanism that allows programmers to write distributed programs with the simplifying assumption that processors do not fail, and then run these programs correctly on systems with transient processor failures. We also described a translation mechanism that can be used to write programs with the simplifying assumptions that processors execute in synchronized steps and messages take exactly one step to arrive, and then run these programs correctly on systems that violate these assumptions. Both mechanisms are transparent to the programmer, and they can be applied to solve a large class of problems.

Our results can be extended in two directions. One direction is to integrate the checkpointing/rollback mechanism with the translation mechanism. This integrated mechanism would allow programmers to make *both* simplifying assumptions; i.e., they would be able to write programs with the assumption that processors do not fail, *and* that processors execute in synchronized steps and message take exactly one step to arrive. With this integrated mechanism, these programs could then be run correctly on systems with transient processor failures, variable processor speeds and message transmission times.

Another extension to the results of this thesis is to use experimental methods to investigate the costs and practicality of our mechanisms. As we mentioned before, these costs depend on the parameters of the underlying systems. For example, the costs of our translation mechanism depend on the magnitude of the deviation of the underlying real system from the ideal system. Implementing our mechanisms and experimenting with various system parameters could lead to a better understanding of these dependencies.

Bibliography

[ALS79] T. Anderson, P. A. Lee, and S. K. Shrivastava. System fault tolerance. In T. Anderson and B. Randell, editors, *Computing System Reliability*, pages 153–210, Cambridge University Press, 1979.

[Awe85] B. Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, 1985.

[BG81] P.A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, 1981.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Massachusetts, 1987.

[BJS86] K. Birman, T. Joseph, and P. Stephenson. *Programming with Shared Bulletin Boards in Asynchronous Distributed Systems.* Technical Report 86-772, Dept. of Computer Science, Cornell University, August 1986.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):204–214, February 1985.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of ACM*, 34(1), January 1987.

[DS83] C. Dwork and D. Skeen. The inherent cost of nonblocking commitment. In *Proc. ACM Symposium on Principles of Database Systems*, 1983.

[FGL82] M. Fischer, N. Griffeth, and N. Lynch. Global states of a distributed system. *IEEE Transaction on Software Engineering*, SE-8(3):198–202, 1982.

[Had82] V. Hadzilacos. An algorithm for minimizing rollback cost. In *Proc. ACM Symposium on Principles of Database Systems*, March 1982.

[Had84] V. Hadzilacos. *Issues of Fault Tolerance in Concurrent Computations*. Ph.D. thesis, Harvard University, June 1984. Harvard University Technical Report 11-84.

[HM85] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. 1985. To appear in JACM. An earlier version of the paper appeared in the Third ACM Symposium on Principles of Distributed Computing, 1984.

[JB86] T. Joseph and K. Birman. Low cost management of replicated data in fault-tolerant distributed systems. *ACM Transactions on Computer Systems*, 4(1):54–70, 1986.

[Jef85] D. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.

[KT87] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transaction on Software Engineering*, SE-13(1):23–31, January 1987.

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

[LF82] L. Lamport and M. Fischer. *Byzantine Generals and Transaction Commit Protocols*. Technical Report 62, SRI International, April 1982.

[LS79] B. Lampson and H. Sturgis. *Crash recovery in a distributed storage system.* Technical Report, Xerox Palo Alto Research Center, April 1979.

[LS83] B. Liskov and R. Scheifler. Guardians and actions: linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[Lun86] J. Lundelius. Simulating synchronous processors. 1986. Unpublished manuscript.

[NT86] G. Neiger and S. Toueg. *Substituting for real time and common knowledge in asynchronous distributed systems.* Technical Report 86-790, Computer Science Department, Cornell University, November 1986.

[Pre83] D. L. Presotto. *Publishing: A reliable broadcast communication mechanism.* Technical Report UCB/CSD 83-165, Computer Science Division, University of California, Berkeley, December 1983.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of ACM*, 27(2):228–234, April 1980.

[PT86] K.J. Perry and S. Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering*, SE-12(3):477–482, March 1986.

[Ran75] B. Randell. System structure for software fault tolerance. *IEEE Transactions On Software Engineering*, SE-1(2):226–232, June 1975.

[RLT78] B. Randell, P.A. Lee, and P.C. Treleaven. Reliability issues in computing system design. *ACM Computing Surveys*, 10(2):123–166, June 1978.

[Rus77] D. L. Russell. Process backup in producer-consumer systems. In *Proc. ACM Symposium on Operating Systems Principles*, 1977.

[Rus80] D. L. Russell. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, SE-6(2):183–194, March 1980.

[Sam85] B. Samadi. *Distributed Simulation, Algorithms, and Performance Analysis.* Ph.D. thesis, Computer Science Department, University of California, Los Angeles, 1985.

[Ske82] D. M. Skeen. *Crash recovery in a distributed database system.* Ph.D. thesis, Computer Science Division, University of California, Berkeley, 1982.

[SY85] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 204–226, August 1985.

[Tan81] A. S. Tanenbaum. *Computer Networks.* Prentice Hall, New Jersey, 1981.

[TS84] Y. Tamir and C. H. Sequin. Error recovery in multicomputers using global checkpoints. In *Proc. of 13th International Conference on Parallel Processing*, August 1984.

[Woo81] W. G. Wood. A decentralized recovery control protocol. In *Proc. of the 11th Annual International Symposium on Fault-Tolerant Computing*, June 1981.