Optimistic Protocols for Fault-Tolerance in Distributed Systems

by

Om Prakash Damani, B.Tech., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 1999

Optimistic Protocols for Fault-Tolerance in Distributed Systems

Approved by
Dissertation Committee:

Dedicated to My Causal Past

Optimistic Protocols for Fault-Tolerance in Distributed Systems

Publication No. ______________________

Om Prakash Damani, Ph.D.
The University of Texas at Austin, 1999

Supervisor: Vijay Kumar Garg

This dissertation focuses on the use of message logging for recovering from process failures in distributed systems. Optimistic message logging protocols assume that failures are rare. Based on this assumption, they try to reduce the failure-free overhead. We have proved several fundamental results about optimistic logging protocols.

We have designed a protocol that allows the user of a system to tune the degree of optimism. This protocol provides a trade-off between failure-free overhead and recovery efficiency. The special cases of this protocol include an existing optimistic protocol and an existing pessimistic protocol.

We have also studied extensions of optimistic protocols to multi-threaded environments. The natural extensions offer a trade-off between the false causality and the failure-free overhead. We avoid this trade-off by treating threads as the unit of recovery and processes as the unit of failure.

The protocols mentioned so far are independent of any particular application characteristics. The fault-tolerance overhead can sometimes be reduced by exploiting the specific characteristics of an application. We have demonstrated this reduction in the context of optimistic computations. Specifically, we have developed a new fault-tolerant optimistic simulation protocol.

Contents

Chapter 1

Introduction

The increasing popularity of intra-nets has caused a shift from main-frames to clusters of workstations. In fact, with the growth of the Internet and the World Wide Web, it has become possible to coordinate machines that are scattered across the globe. For example, the 56-bit DES encryption key [9] was broken by a coordinated effort of thousands of computers spread across the U.S. and Canada [20].

The power of distributed computing, however, comes at a cost. Distributed systems are vulnerable to many forms of failures. At times, the entire system may become unusable due to a failure of even a single link or a single processor. Thus, in order to fully exploit the power of distributed computing, the problem of fault-tolerance needs to be addressed.

1.1 Fault-Tolerance Techniques

The concept of fault-tolerance is as old as the digital computer itself. In a seminal report in 1946, Burks, Goldstine and von Neumann suggested the use of two computers acting in parallel and checking each other's result [15]. The term *fault-tolerance* was formally introduced in 1967 by A. Avizienis [1]. In 1973, the concept of *recovery block* was introduced by Randell and others [34]. By 1975, they extended the recovery block concept to a set of cooperating processes. This is, to our knowledge, the earliest work on fault-tolerant distributed systems. Since that first work, a number of techniques have been developed.

When reducing down-time is of paramount importance, redundant hardware is used to mask as many failures as possible. The ESS electronic telephone switching system developed by AT&T is a prime example of such systems [62]. Both Tandem and Stratus corporations have developed a number of products that include redundant power supplies, backplane buses and dual paths to all system elements including disks and I/O controllers. These techniques can reduce the number of crashes due to hardware, but when a program does crash, some other means of fault-tolerance is required.

There are two classes of techniques for providing fault-tolerance to distributed applications. Techniques in the first class assume a specific programming model and require application developers to use that particular model. Techniques in the second class are general purpose and work for any message-passing application.

1.1.1 Programming Model Specific Techniques

Remote procedure call, transactions, distributed shared memory, distributed objects, etc. are examples of specific programming models for which fault-tolerance techniques have been developed. Of these, we discuss transactions and distributed objects.

Transactions

A large number of distributed applications use the transactional approach [30]. A transaction has following properties: atomicity, consistency, independence, and durability. It is well suited for applications whose prime activities are maintaining and updating a database. However, it is overly restrictive for applications that are not so data-centric but are computation-centric and require cooperation among a set of peer processes. The *Independence* part of transactions does not work well for message-passing distributed applications because processes in these applications are designed to be inter-dependent.

Distributed Object Systems

With the increasing popularity of distributed objects, many applications are being written in CORBA, DCOM or Java RMI. A number of researchers have directed their efforts towards providing fault-tolerance for these applications [14, 41, 49, 65].

Our focus in on developing general purpose techniques that can be used with any message-passing application.

1.1.2 General Purpose Techniques

A number of generic methods have been developed for message-passing applications. These methods can be customized to implement programming model specific techniques like fault-tolerant RPC. They can be divided in two categories: spatial redundancy and temporal redundancy.

Spatial Redundancy

In addition to hardware, software can be replicated too. Multiple copies of a program can be executed concurrently. If one of the replicas fail the remaining replicas can continue execution without stopping the applications. This method is called *active replication*. The Isis toolkit [16] provides active replication.

Replication techniques work well in client-server settings where servers are replicated for high availability. In a general purpose distributed system, however, these strategies require complex multi-way synchronization between the replicas of senders and the replicas of receivers. The complexity involved and high resource requirement make replication unattractive for general purpose, peer-to-peer distributed computing.

Temporal Redundancy

The algorithms developed in this dissertation are based on temporal redundancy and therefore we describe this method in greater details. Compared to spatial redundancy, this method requires fewer physical resources. This advantage comes at the price of reduced availability upon the occurrence of a failure. Therefore, this approach is suited for non-mission critical applications where low-cost fault-tolerance is important.

In this approach, the state of each process is periodically recorded on stable storage. The saved state is called a *checkpoint*. A technique that is used in conjunction with checkpointing is *message logging*. In message logging, the contents and processing orders of the received messages are also saved in volatile and stable storage. While data saved on volatile storage are lost in a process crash, data saved on stable storage are assumed to remain unaffected. Upon recovery, a process restores a checkpointed state and replays the messages logged on stable storage.

The reconstructed state of the failed process may be *inconsistent* with the states of the surviving process [17]. The goal of a recovery protocol is to bring the system back to a *consistent* state after one or more processes fail. A *consistent* state is one where the send of a message is recorded in the sender's state if the receipt of the message has been recorded in the receiver's state. A more general definition of the recovery problem is given in Section 2.2, where the system model is introduced.

Message logging based recovery is especially useful for distributed applications that frequently interact with the outside world [24]. It can be used either to reduce the amount of work lost due to failures in long-running scientific applications [24], or to enable fast and localized recovery in continuously-running service-providing applications [35].

Depending on when and where the received messages are logged, the message logging schemes can be divided into three main categories: pessimistic, optimistic, and causal [24]. In pessimistic logging, each message is logged on stable storage before it is processed [33]. This ensures that all pre-failure states can be recreated after a failure. Some pessimistic logging protocols reduce the overhead by delaying the logging till the point where a message dependent on unlogged messages needs to be sent [37, 39].

In optimistic logging, messages are logged in volatile storage which is periodically flushed to stable storage [61]. Thus, optimistic logging protocols incur lower failure-free overhead compared to pessimistic logging protocols. On a failure, however, messages in volatile log are lost. Therefore, the entire system has to roll back to a consistent state that occurred prior to the lost messages. This implies that failure recovery in optimistic protocols may be slower than that in pessimistic protocols.

Causal logging protocols are based on the observations that a process may log messages in the volatile log of another process and the time of logging can be delayed until the point of actual dependency creation [5, 25]. These observations are used to avoid rolling back non-failed states as in pessimistic logging while achieving low failure-free overhead as in optimistic logging.

1.2 Motivation for Optimistic Protocols

In this dissertation, we focus on optimistic message logging protocols. These protocols have the advantage of low failure-free overhead. In addition, there are many scenarios in which optimistic logging schemes are desirable.

1. **Non-crash failures:** Traditional logging protocols are based on the assumption that processes fail by simply crashing, without causing any other harm such as sending incorrect messages. In practice, there is some latency between a fault-occurrence and the fault-detection. Optimistic protocols can handle this problem, when possible, by identifying and rolling back the faulty states [61].

2. **Software bugs:** Traditional logging protocols assume that successive failures of a process are independent. On restarting a failed process, the cause of the last crash is not expected to lead to another crash. However, when a software bug crashes a program, deterministically recreating the pre-failure computation results in the same bug leading to the same crash. A way to avoid this is to replay the last few messages in a different order, thereby potentially bypassing the bug that caused the original crash [61, 67].

3. **Optimistic computations:** Many applications employ techniques similar to optimistic logging and require rollback capability. (e.g., optimistic distributed simulation [36].) In such applications, the fault-tolerance overhead can be reduced by employing the same dependency tracking mechanism for both the application and the recovery system.

4. **Distributed debugging:** If a program needs to be tested under different message orderings, a technique similar to optimistic recovery can be used. After the result for a particular message ordering is available, a failure can be simulated and a different message ordering can be tried.

5. **Input message cancellation:** Traditional recovery protocols assume that messages from the environment are irrevocable. However, many new classes of distributed applications are emerging that allow the environment to revoke input messages but still do not allow the environment to be modeled

as one of the application process. One example is an application based on the integration of log based techniques with transaction processing. For such applications, revoking of an input message can be modeled as a failure in an optimistic system.

1.3 Dissertation Contributions

In this dissertation, we have developed both application-transparent and application-specific recovery protocols. In the context of application-transparent protocols, we have designed a K-optimistic protocol that offers a trade-off between failure-free overhead and recovery efficiency and have extended this protocol to multi-threaded systems. We have also developed a new distributed simulation scheme and have integrated it with the recovery layer. This demonstrates how to reduce the cost of fault-tolerance for optimistic computations. Next, we briefly discuss each of these contributions. In later chapters, we discuss these contributions in detail and present comparisons with related work.

1. **Efficient recovery in optimistic protocols:** Many traditional optimistic protocols treat a failure and a rollback almost identically. We have established that if transitive dependency tracking is employed, a process can correctly recover by learning only about failures in the system and not all rollbacks. This results in more efficient recovery. Other researchers have also used our result to improve their protocols [58]. We have further shown that any protocol that employs transitive dependency, need not track dependencies on stable states. This result further reduces the failure-free overhead of optimistic logging protocols.

2. **Bridging the gap between optimism and pessimism:** Although pessimistic and optimistic protocols together provide a trade-off between failure-

free overhead and recovery efficiency, it is only a coarse-grained trade-off. The application has either to tolerate the high overhead of pessimistic logging or to accept the potentially inefficient recovery of optimistic logging. To address this issue, we have introduced the concept of K-optimistic logging where K is a tunable parameter.

The concept of K-optimism provides a fine-grained trade-off between recovery time and logging overhead with traditional optimistic and pessimistic logging being the two end-points of the spectrum. The parameter K can be dynamically tuned to adjust to a changing environment. The trade-off provided is probabilistic in nature. In the worst case, for any value of K, the recovery time can be as bad as that for a completely optimistic protocol. This is not surprising,x because in the worst case, pessimistic protocols can have as bad a recovery time as optimistic protocols.

3. **Extending optimistic protocols to multi-threaded systems:** Optimistic logging protocols can be extended to multi-threaded processes in two natural ways: process-centric and thread-centric. These two approaches together present a trade-off between false causality and dependency tracking overhead. We avoid this trade-off by establishing that it is sufficient to track the dependency of threads on processes. The main intuition is that processes fail independently and are thus failure units and that threads may be rolled back independently and are thus rollback units. Based on this result, our protocol eliminates false causality while incurring low overhead.

4. **Efficient recovery for optimistic computations:** In optimistic computations, a process avoids blocking until the outcome of an event by guessing the outcome. If the guess turns out to be correct, the optimism pays off. However, if the guess turns out to be wrong, all the computation that follows the wrong guess needs to be undone. Optimistic recovery schemes rely on a

similar concept, and therefore can be conveniently integrated with optimistic computations. We have demonstrated this integration in the context of distributed simulation. We have also developed an efficient optimistic simulation protocol.

1.4 Dissertation Outline

Chapter 2 presents the system model that is used throughout the dissertation. In this chapter, Lamport's happened before relation is extended to failure-prone systems. A mechanism to track causal dependencies is presented and its properties are discussed.

Application-transparent protocols are presented in Chapters 3 and 4. In Chapter 3, a K-optimistic protocol that bridges the gap between optimism and pessimism is presented and proved correct. This chapter also presents an experimental evaluation of the protocol. This protocol assumes that processes are single-threaded. In Chapter 4, optimistic protocols are extended to multi-threaded systems. A detailed discussion of process-centric and thread-centric approaches is presented and a balanced protocol combining the advantages of both of these approaches is designed.

Chapter 5 illustrates how to reduce the recovery overhead by exploiting application-specific knowledge. This chapter presents the details of an optimistic simulation protocol and shows how to integrate this protocol with optimistic recovery protocols. Finally, Chapter 6 summarizes the dissertation contributions and presents directions for future research.

Chapter 2

Theoretical Framework

In this chapter, we introduce the formal model of the system that is used throughout the dissertation. We formally define what it means for a state to be dependent on another state. This dependency relation is an extension of the Lamport's happened before relation [42] to failure prone computations.

2.1 Abstract Model

A process execution is a pair $(S, \prec)$. S is a set of elementary entities called state interval (*si* for short). There exists a boolean-valued function *init* that takes a *si* as its input and returns *true* for exactly one of the members of S. The relation $\prec$ is an acyclic binary relation on S satisfying following conditions:

- $\forall s : |\{u : init(s) \wedge u \prec s\}| = 0$
- $\forall s : |\{u : \neg init(s) \wedge u \prec s\}| = 1$

The relation $\prec$ induces a tree on S. If $u \prec s$, u is a parent of s and s is a child of u.

In an online execution, new elements can be added to S at any time with an accompanying strengthening of the relation $\prec$.

A *si* can have one of three labels: *useful*, *lost* and *rolled_back*. Every *si* starts as *useful*. A newly added *si* becomes child of a *useful si* that has no *useful* child. The label of only a *useful*, non-*init si* can be changed. When the label of a *si* is changed, labels of all its *useful* children are also changed in the same way. This change propagates recursively to all descendents.

Consider a system consisting of n processes $P_1, .., P_n$. Let the execution of P_i be $(S_i, \prec_i)$. The system execution is a triplet $(H, \prec, \leadsto)$. The set H is defined as $H \equiv \bigcup_i S_i$. The acyclic binary relation $\prec$ is defined on H as $\prec \equiv \bigcup_i \prec_i$. The relation $\leadsto$, another acyclic binary relation defined on H, satisfies the following conditions[1]:

- $\forall s : |\{u : init(s) \wedge u \leadsto s\}| = 0$
- $\forall s : |\{u : \neg init(s) \wedge u \leadsto s\}| = 1$

Let the relation $\rightarrow$ be the transitive closure of $\prec \cup \leadsto$. Two system executions are considered equivalent if their $\rightarrow$ relations, restricted to *useful si*, are same.

2.2 Physical Model and the Recovery Problem

We consider an application system consisting of n processes communicating only through messages. The communication system used is unreliable in that it can lose, delay or duplicate a message. The environment also uses messages to provide inputs to and receive outputs from the application system. Each process has its own volatile storage and also has access to stable storage [46]. The data saved on volatile storage is lost in a process crash, while the data saved on stable storage remains unaffected by a process crash.

The state of a process consists of values of all program variables and the program counter. A *state interval* is a sequence of states between two consecutive

[1]As an aside we note that just like $\prec$, $\leadsto$ also induces n disjoint trees on H.

message receipts by the application process. The execution within each interval is assumed to be completely deterministic, i.e., actions performed between two message receives are completely determined by the content of the first message received and the state of the process at the time of the first receive. In Chapter 4 we relax this assumption and consider effects of non-deterministic thread scheduling in multi-threaded systems. For the purpose of recovery, we are interested in state intervals only and not in states, and therefore for convenience, we use the term *state* instead of *state interval*.

A state interval here corresponds to a state interval (si) in the abstract model. If in the abstract model, $s \prec u$, then the interval corresponding to s immediately precedes the interval corresponding to u. If $s \rightsquigarrow u$ then a message is send in the interval corresponding to s and the receive of that message results in the interval that corresponds to u. From now on, when there is no confusion, we use the term 'state s' instead of saying 'state interval that corresponds to si s.'

Although an abstract process execution is a tree, a physical process execution is a sequence of state intervals in real time. All n process executions together constitute a system execution. Two physical system executions are considered equivalent if their abstract counterparts are equivalent.

We assume perfect failure detection [19], i.e. each non-failed process eventually learns about all failures in the system and no process falsely assumes that a non-failed process has failed. A process fails by simply crashing. In a crash failure, a process stops executing and loses the data in its volatile storage. The process does no other harm, such as sending incorrect message. Pre-failure states of a process that cannot be recreated after a failure are called lost states. A lost state gets the label *lost* in the abstract model.

The application system is controlled by an underlying recovery system. The type of control may be of various forms, such as saving a checkpoint of the application process, stopping an application process, adding control information to the state of an application process, adding control information to a message, rolling back the application to an earlier state, etc.

If an application state is rolled back by the recovery system then that state is called *rolled_back*.

The *recovery problem* is to specify the behavior of a recovery system that controls the application system to ensure that despite crash failures, the system execution remains equivalent to a possible crash-free execution of the stand-alone application system.

From here on, when there is no confusion, instead of saying 'the system does something for the corresponding process', we will say 'a process does something'. We next give a general description of optimistic protocols in this model.

2.2.1 Optimistic Recovery

Optimistic recovery is a special class of log-based rollback recovery, where the recovery system employs checkpointing and message logging to control the application [24]. In optimistic recovery, received messages are logged in volatile storage. The volatile log is periodically written to stable storage in an asynchronous fashion. By asynchronous, we mean that a process does not stop executing while its volatile log is being written to stable storage. Each process, either independently or in coordination with other processes, takes periodic checkpoints [24].

After a crash, a process is *restarted* by restoring its last checkpoint and replaying logged messages that were received after the restored checkpoint. Since some messages might not have been logged at the time of the failure, some pre-failure states, called *lost* states, cannot be recreated. States in other processes that causally depend on lost states are called *orphan*. Causal dependency corresponds to the $\rightarrow$ relation in the abstract model. A message sent by a lost or orphan state is called an orphan message. If the current state of a process is orphan then the process itself is called orphan. All orphan states are rolled back. All orphan messages are also discarded. Each restart or rollback starts a new *incarnation* of the process. A failure or a rollback does not start a new interval. It simply restores an old interval.

Traditional optimistic protocols treat rollback of a failed process as if the process has failed and restarted. We note the distinction between a restart and a rollback. A failed process restarts whereas a rollback is done by a non-failed process. Information stored in volatile memory before a failure is not available at restart. In a rollback, no information is lost. Unlike in traditional protocols, in our protocols, a process informs other processes about its failures only and not about rollbacks.

In all optimistic protocols (or all log-based recovery protocols), the recovered state could have happened in a failure-free execution of the application, with relatively slower processor speed and relatively increased network delays. Therefore, in an asynchronous system, optimistic protocols solve the recovery problem.

Output Commit

Distributed applications often need to interact with "the outside world." Examples include setting hardware switches, performing database updates, printing computation results, displaying execution progress, etc. Since the outside world in general does not have the capability of rolling back its state, the applications must guarantee that any output sent to the outside world will never need to be revoked. This is called the *output commit problem*.

In optimistic recovery, an output can be committed when the state intervals that the output depends on have all become *stable* [61]. An interval is said to be stable if it can be recreated from the information saved on stable storage. To determine when output can be committed, each process periodically broadcasts a logging progress notification to let other processes know which of its state intervals have become stable. Such information is accumulated at each process to allow output commit.

Example

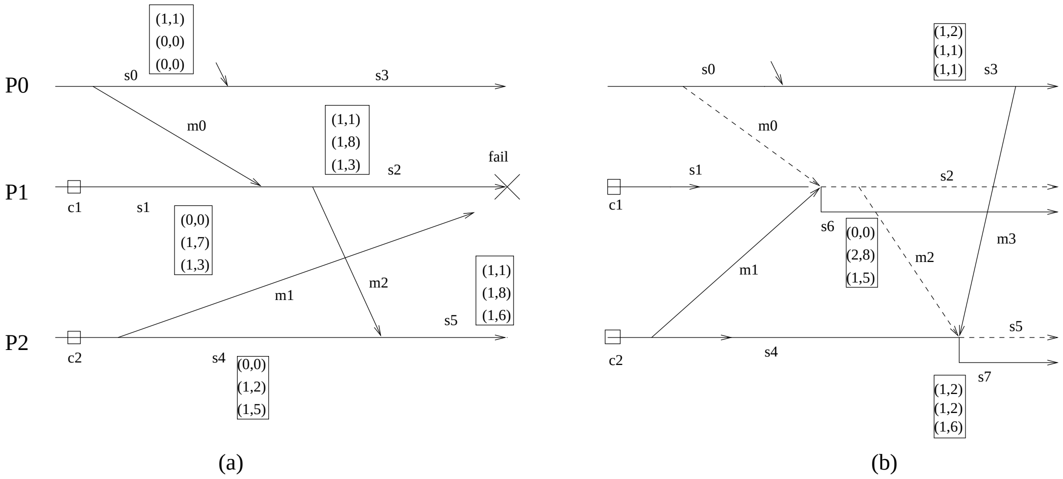


Figure 2.1: Example: Optimistic Recovery in Action

An example of an optimistic recovery system is shown in Figure 2.1. Solid horizontal lines show the useful computation, and dashed horizontal lines show the computation that is either lost in a failure or rolled back by the recovery protocol. In the figure, $c1$ and $c2$, shown by squares, are checkpoints of processes $P1$ and $P2$ respectively. State intervals are numbered from $s0$ to $s7$ and they extend from one message receive to the next. The numbers shown in rectangular boxes will be explained later in this chapter.

In Figure 2.1(a), process $P1$ takes a checkpoint $c1$, acts on some messages (not shown in the figure) and starts the interval $s0$. $P1$ logs to stable storage all messages that have been received so far. It starts interval $s2$ by processing the message $m0$. In interval $s2$, message $m2$ is sent to $P2$. $P1$ then fails without logging the message $m0$ to stable storage or receiving the message $m1$. It loses its volatile memory, which includes the knowledge about processing the message $m0$. During this time, $P2$ acts on the message $m2$.

Figure 2.1(b) shows the post-failure computation. On restarting after the failure, $P1$ restores its last checkpoint $c1$, replays all the logged messages and restores the interval $s1$. It then broadcasts a failure announcement (not shown in Figure 2.1). It continues its execution and starts interval $s6$ by processing $m1$. $P2$ receives the failure announcement in interval $s5$ and realizes that it is dependent on a lost state. It rolls back, restores its last checkpoint $c2$, and replays the logged messages until it is about to process $m2$, the message that made it dependent on a lost state. It discards $m2$ and continues its execution by processing $m3$. The message $m2$ is not regenerated in post-failure computation. $P0$ remains unaffected by the failure of $P1$.

Notations

We next define notations that are used throughout the dissertation.

- i, j, k refer to process identification numbers.
- t refers to the incarnation number of a process.
- s, u, v, w, z refer to a state (or a state interval).
- P_i refers to the i'th process.
- $P_{i,t}$ refers to incarnation t of P_i.
- $s.p$ denotes the identification number of the process to which s belongs, that is, $s.p = i \Rightarrow s \in S_i$.
- x, y refer to state sequence numbers.
- $(t, x)_i$ refers to the x'th state of the t'th incarnation of process P_i.

- m refers to a message.

- c refers to a dependency vector (defined in Section 2.2.3).

2.2.2 Causal Dependency Between States

In the previous section, we talked about one state being dependent on another. The application state resulting from a message delivery depends on (is determined by) the content of the message delivered and therefore depends on the state sending the message. This dependency relation is transitive. It corresponds to the $\rightarrow$ relation defined in the abstract model. Lamport defined the *happened before* relation [42] for a failure-free computation. Our dependency relation is an adaptation of the happened before relation to a failure-prone systems. The physical meaning of the abstract relation $\rightarrow$ is as follows. In a failure-prone system, happened before ($\rightarrow$) is the transitive closure of the relation defined by the following two conditions:

- $u \rightarrow v$, if the processing of an application message in state u results in state v, (for example, $s1 \rightarrow s6$ in Figure 2.1(b)),

- $u \rightarrow v$, if the processing of an application message sent from u starts v (for example, $s2 \rightarrow s5$ in Figure 2.1(a)).

We say that u is transitively dependent or simply dependent on s if s happened before u. By $s \underline{\rightarrow} u$, we mean $s \rightarrow u$ or $s = u$. By $s \not\rightarrow u$ we mean s did not happen before u. For example, in Figure 2.1(b), $s2 \not\rightarrow s6$.

Only application messages contribute to the happened before relation. The recovery protocol might also send some messages. These messages do not contribute to the happened before relation.

Earlier we mentioned that a state dependent on a lost sate is called orphan. We can now formally define *orphan* as:

Definition 1 $orphan(s) \equiv \exists u : lost(u) \ \wedge \ u \underline{\rightarrow} s$

To detect orphans, we need a mechanism to track dependencies between states.

2.2.3 Dependency Tracking Mechanism

We use dependency vectors to track transitive dependencies between states in a failure-prone system. Although dependency vectors have been used before [61], their properties have not been discussed.

A dependency vector has n entries, where n is the number of processes in the system. Each entry contains an *incarnation number* and a *state sequence number* (or simply *sequence number*). Let us consider the dependency vector of a process P_i. The incarnation number in the i'th entry of P_i's dependency vector (its own incarnation number) is equal to the number of times P_i has failed or rolled back. The incarnation number in the j'th entry is equal to the highest incarnation number of P_j on which P_i depends. Let entry e correspond to a tuple (incarnation t, sequence number seq). Then, $e_1 < e_2 \equiv (t_1 < t_2) \vee [(t_1 = t_2) \wedge (seq_1 < seq_2)]$.

A process sends its dependency vector along with every outgoing message. Before delivering a message to the application, a process updates its dependency vector with the message's dependency vector by taking the componentwise maximum of all entries. The process then increments its own sequence number.

To start a new incarnation, a process increments its incarnation number (it leaves the sequence number unchanged). A new incarnation is always started after a rollback or a failure.

The dependency tracking mechanism is given in Figure 2.2. An example of the mechanism is shown in Figure 2.1. The dependency vector of each state is shown in a rectangular box near it. The row i of the dependency vector corresponds to P_i (P_i is shown as Pi in Figure 2.1.) .

```
Process P_i :

type entry =    (int inc, int seq)      // incarnation, sequence number
var c :         array[n] of entry       // n : number of processes in system


Initialize :
    ∀ j : c[j] := (0,0) ;
    c[i] := (1,1) ;
Send_message :
    send (data, c) ;
Process_message (m) :
    // P_i receives the dependency vector 'm.c' with incoming message
    ∀ j: c[j] := max(c[j],m.c[j]) ;
    c[i].seq := c[i].seq + 1 ;
Start_incarnation :
    // A new incarnation is started after a failure or a rollback
    c[i].inc := c[i].inc + 1 ;
```

Figure 2.2: Dependency vector algorithm

Properties of Dependency Vectors

Dependency vectors have properties similar to Mattern's vector clocks [48]. They can be used to detect transitive dependencies between *useful* states (states which are neither lost nor orphan).

We define an ordering between two dependency vectors $c1$ and $c2$ as follows.

$$c1 < c2 \equiv (\forall i : c1[i] \leq c2[i]) \wedge (\exists j : c1[j] < c2[j])$$

Let $s.c$ denote the dependency vector of $P_{s.p}$ in state s. The following lemma gives a necessary condition for the $\not\rightarrow$ relation between two *useful* states.

Lemma 1 *Let s and u be distinct useful states (neither lost nor orphan). Then,* $s \not\rightarrow u \Rightarrow u.c[s.p] < s.c[s.p]$

Proof: Let $s.p = u.p$. Since s and u are distinct useful states, it follows that $u \rightarrow s$. During processing of a message, $P_{s.p}$ takes the maximum of dependency vectors and then increments the sequence number of its own component. On restart after a failure or a rollback, $P_{s.p}$ increments its incarnation number. Since for each state transition along the path from u to s, the local dependency vector is incremented, $u.c[s.p] < s.c[s.p]$.

Let $s.p \neq u.p$. As $s \not\rightarrow u$, $P_{u.p}$ could not have seen $s.c[s.p]$, the local dependency vector of $P_{s.p}$. Hence $u.c[s.p] < s.c[s.p]$. ■

As shown in the next theorem, the above condition is also sufficient for the $\not\rightarrow$ relation. The next theorem shows that, despite failures, dependency vectors keep track of causality for *useful* states.

Theorem 1 *Let s and u be useful states in a distributed computation. Then, $s \rightarrow u$* **iff** $s.c < u.c$

Proof: If $s = u$, then the theorem is trivially true. Let $s \rightarrow u$. Since both s and u are useful, there is a message path from s to u such that none of the intermediate states are either lost or orphan. Due to monotonicity of dependency vectors along each link in the path, $\forall j : s.c[j] \leq u.c[j]$. Since $u \not\rightarrow s$, from Lemma 1, $s.c[u.p] < u.c[u.p]$. Hence, $s.c < u.c$.

The converse follows from Lemma 1. ■

Dependency vectors do not detect the causality for either lost or orphan states. To detect causality for lost or orphan states, we use an *incarnation end table*, as explained in Section 3.3.

2.3 Related Work

The model described here is similar to but more general than the one given in the original optimistic work by Strom and Yemini [61]. They assumed reliable message delivery and therefore they considered a global state to be consistent if all the sent messages have been received. We assume that messages can be lost.

Johnson and Zwaenepoel established that the set of recoverable states form a lattice [40]. This result implies that after any failure in an optimistic system, there exists a maximum recoverable state with respect to that failure. They however used direct dependency tracking instead of transitive dependency tracking. They also used a centralized protocol to compute the maximum recoverable state.

Smith, Johnson and Tygar explicitly distinguished between application computation and the recovery system computation [59]. In their model causality is separately identified for each level of computation. They used a tree diagram similar to the one used in this dissertation. They used a time-tree to track complete causality. Unlike dependency vectors, which can track causality only between useful states, time-trees can track causality between any two states. However, the time-tree mechanism results in much higher overhead and therefore we do not use that mechanism.

Alvisi and Marzullo gave the first formal specification of the necessary and sufficient condition for the eventually no-orphan property satisfied by the message logging protocols [7]. Based on that specification, they derived a causal logging protocol.

Chapter 3

Transparent Recovery for Single-Threaded Processes

In this chapter, we describe an optimistic recovery scheme that is independent of any particular application. The scheme assumes that all application processes are single-threaded. Next chapter discusses the issue of multi-threading. Application-specific techniques are discussed in Chapter 5.

We first prove several fundamental properties about optimistic recovery. Using these properties, we design a K-optimistic protocol that bridges the gap between optimism and pessimism. This protocol provides a trade-off between recovery time and failure-free overhead. For K equal to n, the protocol reduces to the optimistic protocol presented in [21], while for K equal to 0, it reduces to the pessimistic protocol presented in [37].

3.1 Motivation

Traditional pessimistic logging and optimistic logging provide a coarse-grain trade-off between failure-free overhead and recovery efficiency: the application has to either tolerate the high overhead of pessimistic logging or accept the potentially inefficient recovery of optimistic logging. For long-running scientific applications, the primary performance measure is the total execution time. For these applications, minimizing failure-free overhead is more important than improving recovery efficiency because failures are rare events. Hence, optimistic logging is a better choice. In contrast, for continuously-running service-providing applications, the primary performance measure is the service quality. Systems running such applications are often designed with extra capacity which can absorb reasonable overhead without causing noticeable service degradation. On the other hand, improving recovery efficiency to reduce service down time can greatly improve service quality. As a result, many commercial service-providing applications have chosen pessimistic logging [35].

The above coarse-grain trade-off, however, may not provide optimal performance when the typical scenarios are no longer valid. For example, although hardware failures are rare, programs can also fail or exit due to transient software or protocol errors such as triggered boundary conditions, temporary resource unavailability, and by-passable deadlocks. If an application suffers from these additional failures in a particular execution environment, slow recovery due to optimistic logging may not be acceptable. Similarly, for a service-providing application, the initial design may be able to absorb higher run-time overhead incurred by message logging. However, as more service features are introduced in later releases, they consume more and more computation power and the system may no longer have the luxury to perform pessimistic logging.

These observations motivate the concept of K-optimistic protocol where K is the degree of optimism that can be tuned to provide a fine-grain trade-off. The basic idea is to ask each message sender to control the maximum amount of risk placed on each message. A sender can release a message only after it can guarantee that failures of at most K processes can possibly revoke the message (see Theorem 6).

This protocol provides a trade-off between recovery time and logging overhead, with traditional optimistic and pessimistic protocols being two extremes. As the value of K moves from n to 0, the recovery time goes down with a corresponding increase in the logging overhead. The parameter K can be dynamically changed to adjust to a changing environment.

The trade-off provided is probabilistic in nature. In the worst case, for any value of K, the recovery time can be as bad as that for a completely optimistic protocol. This is not surprising because in the worst case, pessimistic protocols can have as bad a recovery time as optimistic protocols.

3.2 Theoretical Basis

In Chapter 2.2, we presented the distinction between restart due to a process's own failure and rollback due to some other process's failure. Traditional optimistic recovery protocols [59, 61] blur this distinction and refer to lost states as rolled back states. In order to relate our results to those in the literature, we use the following terminology. A state satisfies predicate *rolled_back* if it has been either lost in a failure or explicitly rolled back by the recovery protocol. In traditional protocols, any state dependent on a rolled back state is called an *orphan*. The following predicate formally defines an orphan state for these protocols.

Definition 2 $orphan(s) \equiv \exists u : rolled\_back(u) \;\wedge\; u \underline{\rightarrow} s$

We have presented above definition only for an understanding of traditional protocols and for proof of theorems in this section. In rest of the dissertation, we use the orphan definition given in Section 2.2.2. For emphasis, we reproduce that definition here:

Definition 1 $orphan(s) \equiv \exists u : lost(u) \;\wedge\; u \underline{\rightarrow} s$

For orphan detection, traditional optimistic protocols usually require every non-failed rolled back process to behave as if it itself has failed [59, 61]. After each rollback, a process starts a new incarnation and announces the rollback. We observe that announcing failures is sufficient for orphan detection. We give a proof of this observation in the following theorem.

Theorem 2 *With transitive dependency tracking, announcing only failures (instead of all rollbacks) is sufficient for orphan detection.*

Proof. Let a state interval v be orphan because of rollback of another interval u. Interval u rolled back either because $P_{u.p}$ failed or because a rollback of another interval z made u orphan. By repeatedly applying this observation, we find an interval w whose rollback due to $P_{w.p}$'s failure caused v to become orphan. Because of transitive dependency tracking, $P_{v.p}$ can detect that v depends on w. Therefore, $P_{v.p}$ will detect that v is orphan when it receives the failure announcement from $P_{w.p}$. ∎

The above observation was first used in [21] and later used in [58]. We carry this observation even further in Theorem 3, by proving that any dependencies on stable intervals can be omitted without affecting the correctness of a recovery protocol which tracks transitive dependencies. A state interval is said to be *stable*, if it can be reconstructed from the information saved in stable storage.

We say that v is *commit dependent on* w if v is transitively dependent on w and w is not stable. A system is said to employ *commit dependency tracking* if it can detect the commit dependency between any two state intervals. The following theorem suggests a way to reduce dependency tracking for recovery purposes. It states that if all state intervals of P_j, on which P_i is dependent, are stable then P_i does not need to track its dependency on P_j.

Theorem 3 *Commit dependency tracking and failure announcements are sufficient for orphan detection.*

Proof. Once a state interval becomes stable, it cannot be lost in a failure. It can always be reconstructed by restarting from its previous checkpoint and replaying the logged messages. Following the proof in Theorem 2, an orphan interval v must transitively depend on an interval w that is lost in $P_{w.p}$'s failure. This implies that w had not become stable when the $P_{w.p}$'s failure occurred. By definition of commit dependency tracking, $P_{v.p}$ can detect that v transitively depends on w. Therefore, on receiving the failure announcement from $P_{w.p}$, $P_{v.p}$ will detect v to be orphan. ∎

A process can explicitly inform other processes of new stable state intervals by periodically sending logging progress notifications. Such information can also be obtained in a less obvious way. A failure announcement containing index $(t, x')_i$ indicates that all states of incarnation t of P_i with sequence number greater than x' have been lost in a failure. Since the state with sequence number x' has been restored after a failure, the announcement also serves as a logging progress notification that interval $(t, x')_i$ has become stable. Corollary 1 summarizes this result.

Corollary 1 *Upon receiving a failure announcement containing index $(t, x')_i$, a process can omit the dependency entry $(t, x)_i$ if $x \leq x'$.*

Corollary 1 is implicitly used by Strom and Yemini [61] to allow tracking dependency on only one incarnation of each process so that the size of dependency vector always remains n: *when process P_j receives a message m carrying a dependency entry $(t, x)_i$ before it receives the rollback announcement for P_i's incarnation $(t-1)$, P_j should delay the delivery of m until that rollback announcement arrives.* This in fact implicitly applies Corollary 1.

We can further apply corollary 1 to eliminate unnecessary delays in message delivery. Suppose P_j has a dependency on $(t-2, x)_i$ when it receives message m carrying a dependency on $(t, x+10)_i$. According to Theorem 3, P_j only needs to be informed that interval $(t-2, x)_i$ has become stable. It does not need to be informed anything about incarnation $(t-1)$ before it can acquire the dependency on $(t, x+10)_i$ and overwrite $(t-2, x)_i$. P_j can obtain that information when it receives either a logging progress notification or a failure announcement from P_i. A more interesting and useful special case is when P_j does not have any dependency entry for P_i at all and so the delay is altogether eliminated.

Based on these results, we have developed an efficient optimistic protocol, which is described next.

3.3 The Protocol

```
Process P_i :

type  entry:               (inc int, seq int)          // incarnation, sequence number
var   K:                   int;                        // the degree of optimism
      c :                  array[n] of entry;          // transitive dependency vector
      State_list :         list of state;              // list of non-stable states
      Receive_buffer :     buffer;                     // keeps received messages
      Send_buffer :        buffer;                     // messages to be sent
      log_prog :           array[n] of set of entry;   // logging progress information

      // The following data structures are stored in stable storage
      cur_inc :            int ;                       // current incarnation number
      iet :                array[n] of set of entry;   // incarnation end table
      Stable_state_list :  list of state;              // states saved on stable storage
```

Figure 3.1: Variables maintained by a process

3.3.1 Data Structures

The variables maintained by a process in this protocol are shown in Figure 3.1. The integer K is the degree of optimism. Dependency tracking is done by the dependency vector c. In *log_prog*, logging progress information is maintained by keeping an entry for the highest known stable interval of each known incarnation of each process. The received failure announcements are stored in an incarnation end table (*iet*). Variable *cur_inc* stores the current incarnation number in stable storage. This avoids the loss of incarnation number information in a failure. A simplification that we use to clarify the correctness proof is that of replacing checkpointing and message logging with saving of entire states. Our implementation indeed uses checkpointing and message logging. We discuss this point in detail in Section 3.3.10.

3.3.2 Auxiliary Functions and Predicates

Figure 3.2 shows predicates and functions used in the protocol. We next explain each of them.

- $knows\_orphan$: If a state s knows that a state u is orphan then the predicate $knows\_orphan(s, u)$ is true. This is the case, when the *iet* of s shows u to be dependent on a lost state.

- $stable$: If s belongs to Stable_state_list of $P_{s.p}$, then $stable(s)$ is said to be true.

- seq_num: This function takes a set of entries and an incarnation number and returns the sequence number associated with the given incarnation number in the set.

- $knows\_stable$: A state u is said to correspond to entry e if $u.c[u.p]$ is equal to e. If a state s knows that P_j's state corresponding to entry e is stable then predicate $knows\_stable(s, e, j)$ is true.

- *admissible*: The predicate $admissible(m, s)$ is true if a message m can be processed in a state s. The message can be processed if no dependency on any unstable interval will be overwritten in taking maximum of $m.c$ and $s.c$.

- *get_state*: This function takes a process id and an entry and returns the state interval of the given process that corresponds to that entry.

- *Insert*: This function inserts an entry (t, x) in a set se. If an entry (t, y) for incarnation t already exists in se, then that entry is replaced by $(t, max(x, y))$. This ensures that the set se contains the latest information about incarnation t.

- NULL: A NULL entry is defined to be lexicographically smaller than any non-NULL entry.

In the protocol, unspecified state variable s stands for the current state unless otherwise stated. In a predicate, if a message m is used instead of a state u then $u.c$ in predicate definition is replaced by $m.c$.

3.3.3 Initialization

We next describe the actions taken by a process P_i upon the occurrence of different events. The initialization routine is given in Figure 3.3.

Initialize: Upon starting the execution, a process has no dependency on any other process. Therefore, P_i sets all dependency vector entries, except its own, to NULL. Since each process execution can be considered as starting with an initial checkpoint, the first state interval is always stable. Therefore, P_i updates its *log_prog* accordingly. We show the initial state being added to the *State_list*. In practice, this is not done as the program itself serves as the initial state.

$\text{knows\_orphan}(s,u) \equiv \exists j\text{: } \exists\ (t,x) \in s.iet[j]\text{: } (t = u.c[j].inc) \wedge (x < u.c[j].seq)$

$\text{stable}(s) \equiv s \in \text{Stable\_state\_list}$

$\text{seq\_num}(se, t)\text{: } \textbf{return}\ x\ \textbf{where}\ (t,x) \in se$

$\text{knows\_stable}(s,e,j) \equiv \text{seq\_num}(s.\text{log\_prog}[j], e.inc) \geq e.seq$

$\text{admissible}(m,s) \equiv \forall j\text{: } [s.c[j].inc \neq m.c[j].inc \Rightarrow$

$\quad \text{knows\_stable}(s, \min(s.c[j], m.c[j]), j)]$

$\text{get\_state}(j,e)\text{: } \textbf{return}\ s\ \textbf{where}\ s.p = j \wedge s.c[j] = e$

$\text{Insert}(se, (t,x))\text{: } \textbf{if}\ \exists\ y\text{: } (t,y) \in se\ \textbf{then}\ se := (se - \{(t,y)\}) \cup \{(t, \max(x,y))\}$

$\quad \textbf{else}\ se := se \cup \{(t,x)\}\ ;$

$e = \text{NULL} \equiv (e.inc = \text{NULL} \wedge e.seq = \text{NULL})$

$x = \text{NULL} \equiv (\forall y \neq \text{NULL}\text{: } x < y)$

Figure 3.2: Predicates and functions used in the protocol

```
Initialize :
    ∀j : c[j] := NULL;
    c[i] := (1,1);
    ∀j : iet[j] := log_prog[j] := {} ;                // empty set
    Insert(log_prog[i],(1,1)); cur_inc := 1 ;
    State_list := {s} ; Stable_state_list := {} ;
```

Figure 3.3: K-optimistic protocol: Initialization routine

3.3.4 Message Manipulation

Routines that manipulate messages are given in Figure 3.4.

Send_message: To send a message, the current dependency vector is attached to the message and the message is added to *Send_buffer*. The message is held in *Send_buffer* if the number of non-NULL entries in its dependency vector is greater than K. Messages held in *Send_buffer* are sent in the routine *Check_send_buffer* (in Figure 3.5).

Receive_message: A received message is discarded if it is known to be orphan. Otherwise, it is added to *Receive_buffer*.

Process_message: When the application needs to process a message, any of the admissible messages among the received ones is selected. A message is admissible, if its delivery does not result in the overwriting of any non-stable entry in the dependency vector. In other words, if delivering a message to the application would cause P_i to depend on two incarnations of any process, P_i waits for the interval with the smaller incarnation number to become stable. This information may arrive in the form of a logging progress notification or a failure announcement. Such situation

```
Send_message(data) :
    Send_buffer := Send_buffer ∪ {(data, c)} ;
    Check_send_buffer ;

Receive_message(m) :
    if ¬knows_orphan(s,m) then
        Receive_buffer := Receive_buffer ∪ {m} ;

Process_message(m) :
    if admissible(m) then
        c := max(c,m.c) ; c[i].seq := c[i].seq + 1 ;
            // Application acts on the message
        State_list := State_list ∪ {s} ;
```

Figure 3.4: K-optimistic protocol: Routines that manipulate messages

may arise only for a small time interval after a failure and failure are expected to be rare, hence such blocking will rarely occur. After application processes a message, the current state is included in volatile log.

3.3.5 Routines Executed Periodically

We now describe the routines in Figure 3.5. These routines are executed periodically.

Check_orphan: This routine is called to discard orphan messages from the receive and the send buffers.

Check_send_buffer: This routine updates the dependency vectors of messages in Send_buffer. It is invoked by the events that can announce new stable state intervals, including: (1) *Receive_log_prog* for receiving logging progress notifi-

Check_orphan :

// Discard orphan messages from the receive and the send buffer.

Send_buffer := { m ∈ Send_buffer | ¬knows_orphan(s,m)} ;

Receive_buffer := { m ∈ Receive_buffer | ¬knows_orphan(s,m)} ;

Check_send_buffer :

// Check and send messages held in Send_buffer, if possible

∀ m ∈ Send_buffer: ∀ j: **if** knows_stable(s,m.c[j],j) **then** m.c[j] := NULL ;

∀ m ∈ Send_buffer:

if Number of non-NULL entries in m.c is at most K **then** send m ;

Broadcast_log_prog :

Broadcast(log_prog) ;

Log_state :

Stable_state_list := Stable_state_list ∪ State_list ;

State_list := {} ;

Insert(log_prog[i],c[i]) ;

Check_send_buffer ;

Figure 3.5: K-optimistic logging protocol: Routines invoked periodically.

cation; (2) *Receive_failure_ann* (according to Corollary 1); and (3) *Log_state*. When a message's dependency vector contains K or less non-NULL entries, it is sent.

Broadcast_log_prog: P_i informs other processes about its logging progress by broadcasting its *log_prog*. However, logging progress notification is in general less frequent than the logging of states.

Log_state: This routine is called to save volatile states on stable storage.

```
Receive_log_prog(mlog_prog) :
    ∀ j,t : (t,x) ∈ mlog_prog[j] : Insert(log_prog[j],(t,x)) ;
    ∀ j ≠ i: if knows_stable(s,c[j],j) then c[j] := NULL ;
    // i'th entry is not set to NULL as it will be needed to start the next interval
    Check_send_buffer ;
```

Figure 3.6: K-optimistic logging protocol: Routine for receiving logging notification.

3.3.6 Handling a Logging Notification

On receiving a logging progress notification, the routine in Figure 3.6 is called.

Receive_log_prog: Upon receiving a logging notification, a process updates its *log_prog*. It also sets the stable entries in its dependency vector to NULL. The *log_prog* is periodically flushed to stable storage. As some part of the *log_prog* may get lost in a failure, a process needs to collect the logging information from other processes on restarting after a failure.

```
Restart :          // after failure
    s := head(Stable_state_list) ;
    Insert(iet[i], c[i]) ;
    Broadcast_failure(c[i]) ;
    Start_incarnation ;

Receive_failure_ann (t,x,j) :       // called by state s on receiving (t,x) from P_j :
    Insert(iet[j],(t,x)) ; Insert(log_prog[j],(t,x)) ;
    Check_orphan ;
    Check_send_buffer ;
    if knows_orphan(s,s) then Rollback ;

Rollback :
    Log_state ;
    s := maximum{u ∈ Stable_state_list | ¬knows_orphan(s,u) } ;
    Stable_state_list := Stable_state_list - {u ∈ Stable_state_list | s → u} ;
    Start_incarnation ;

Start_incarnation
    cur_inc := cur_inc + 1 ;
    c[i].inc := cur_inc ;
```

Figure 3.7: K-optimistic logging protocol: Routines involving failure.

3.3.7 Handling a Failure

We next describe the routines in Figure 3.7. These routines are executed in case of a failure.

Restart: On restarting after a failure, P_i restores its last stable state and broadcasts the index of this state as a failure announcement. We assume that the reliable broadcast of a failure includes the execution of the routine *Receive_failure_ann* by all processes. P_i starts a new incarnation by incrementing its incarnation number in the routine *Start_incarnation*.

Receive_failure_ann: On receiving a failure announcement, P_i updates its incarnation end table. As explained in Section 3.2, this announcement also serves as a logging progress notification. P_i also discards orphan messages in Send_buffer and Receive_buffer by calling *Check_orphan* (in Figure 3.5). If the current state of P_i has become orphan due to this failure, then P_i rolls back by calling *Rollback*.

Rollback: Before rolling back, P_i logs its volatile states in stable storage. Clearly, an implementation will log only the non-orphan states. The highest non-orphan stable state is restored and the orphan states are discarded from stable storage. A new incarnation is started. No rollback announcement is send to other processes, which is a distinctive feature of our protocol.

Start_incarnation: This routine increments the current incarnation number, which is saved in stable storage as the variable *cur_inc*. This ensures that the current incarnation number is not lost in a failure. This routine also updates the dependency vector.

3.3.8 Adapting K

Note that there is nothing in the protocol to prevent a change in the value of K. Therefore, the value of K can be changed dynamically in response to changing system characteristics. Also, different processes can have different value of K. A process that is failing frequently may choose to become completely pessimistic by setting its K value to 0 while other processes in system may continue to be optimistic. On the other hand, if the stable storage manager becomes busy, a process may choose to increases its K value.

3.3.9 Output Commit

If a process needs to commit output to external world, it maintains an Output_buffer like the Send_buffer. This buffer is also updated whenever the Send_buffer is updated. An output message is released when all entries in message's dependency vector become NULL. It is interesting to note that an output can be viewed as a 0-optimistic message, and that different values of K can in fact be applied to different messages in the same system. In our implementation described in Section 3.8, we do not use the Output_buffer. Instead, we attach a K value with each message with 0 being assigned to output messages.

In practice, the concept of K-output commit may also be useful. Although strict output commit may be necessary for military or medical applications, most service-providing applications can revoke an output, if absolutely necessary, by escalating the recovery procedure to a higher level which may involve human intervention. Therefore, K-output commit can be useful to provide a trade-off between the commit latency and the degree of commitment.

3.3.10 Using Checkpoints and Message Logs

A simplification that we have used to clarify the correctness proof is that of replacing checkpointing and message logging with the saving of entire states. In our presentation, we save all states in volatile and stable storage. This is useful only in the unlikely case of the average state size being much smaller than the average message size. Otherwise, an implementation should save the received message instead

```
Log_message :
    Stable_message_list := Stable_message_list ∪ Message_list ;
    Message_list := {} ;
    Insert(log_prog[i],c[i]) ;
    Check_send_buffer ;

Checkpoint :
    Log_messages ;
    Stable_state_list := Stable_state_list ∪ {s} ;
```

Figure 3.8: K-optimistic protocol routines that use checkpointing.

of the states in volatile memory. Periodically, the current state should be saved on stable storage as a checkpoint. Any state can be reconstructed by restoring the highest checkpoint prior to that state and replaying the messages that have been received between the checkpoint and the state. Instead of a volatile *State_list*, a volatile *Message_list* is used. A *Stable_message_list* is also used. Checkpoints are stored in *Stable_state_list*. Instead of routine *Log_state*, two new routines are used. These routines are given in Figure 3.8. The old routines that are modified by this implementation strategy are shown in Figure 3.9.

So far, we have discussed the design of the K-optimistic protocol. There are a number of implementation issues that have been avoided for clarity. We now take a look at these issues.

3.4 Implementation Notes

There are a number of policy decisions and optimizations that are available to a system designer.

```
Process_message(m) :
    if admissible(m) then
        c := max(c,m.c) ; c[i].seq := c[i].seq + 1 ;
        Message_list := Message_list ∪ {m} ;
            // Application acts on the message

Restart :        // after failure
    s := head(Stable_state_list) ;
    replay the logged messages that follow;
    Insert(iet[i], c[i]) ;
    Broadcast_failure(c[i]) ;
    Start_incarnation ;

Rollback :
    Log_message ;
    s := maximum{u ∈ Stable_state_list | ¬knows_orphan(s,u) } ;
    Stable_state_list := Stable_state_list - {u ∈ Stable_state_list | s → u} ;
    Replay the messages logged after s in the original receipt order,
        till the current state remains non-orphan;
    Among remaining logged messages, discard orphans and
        add non-orphans to Receive_buffer ;
    Start_incarnation ;
```

Figure 3.9: Modified K-optimistic logging protocol routines.

3.4.1 Policy Decisions

1. While broadcasting the logging progress information, a process can choose to broadcast either its own logging information only or the information about all processes that it knows of. Similarly, at the time of failure announcement, logging information about all processes can be broadcast.

2. In general, logging progress need not be broadcast reliably. For a given incarnation, logging progress is monotonic. Therefore, future notification will supply the missing information. However, if an implementation does not broadcast the information about previous incarnations, then in the routine *Start_incarnation*, logging information of previous incarnation needs to be broadcast reliably.

3. We maintain the dependency vector as a vector of n entries. However, dependency vector can also be viewed as a set of triplets of the form (process number, incarnation number, sequence number). Depending on the relative values of K and n, more efficient form should be used.

3.4.2 Optimizations

1. In Figure 3.9, in routine *Restart*, failure broadcast is done after replaying the messages. An implementation will compute the index of the maximum recoverable state and broadcast it before replaying the messages.

2. In Figure 3.9, routine *Log_message* is called in the routine *Rollback* to log all unlogged messages. An implementation will log only non-orphan messages.

3. When a process sets its K value to 0, it needs to reliably broadcast a logging progress notification. After that it does not need to send a logging notification as no other process will be commit dependent on its future intervals. With

this optimization, our protocol behaves like the pessimistic protocol in [37] for K equal to 0.

3.4.3 Other Issues

In this dissertation, our focus is on the design of efficient optimistic protocols. There are a number of implementation issues that are not addressed here. We next give a partial list of these issues. These and many other issues are discussed in detail in [24].

- **Failure detection:** In theory, it is impossible to distinguish a failed process from a very slow process [28]. In practice, many failure detectors have been built that work well for practical situations [33]. Most of these detectors use a timeout mechanism.

- **Garbage collection:** Some form of garbage collection is required to reclaim the space used for checkpoints and message logs [61].

- **Stable storage:** Logging protocols require some form of stable storage that remains available across failures. In a multi-processor environment local disk can be used, because as other processors can access the local disk even if one of the processors fails. In a networking environment, the local disk may be inaccessible when the corresponding processor fails. Therefore, a network storage server is required. The storage server itself can be made fault-tolerant by using the techniques presented in [46].

- **Network Address:** When a failed process is restarted, it may have a different network address. Therefore, location independent identifiers need to be used for the purpose of inter-process communication.

- **Environment Variables:** If a failed process is restarted on a processor different from the one used before the failure then some inconsistency may arise due to mismatch of the values of environment variables in pre- and post-failure computation. In such scenario, logging and resetting of environment variables is required.

3.5 A Detailed Example

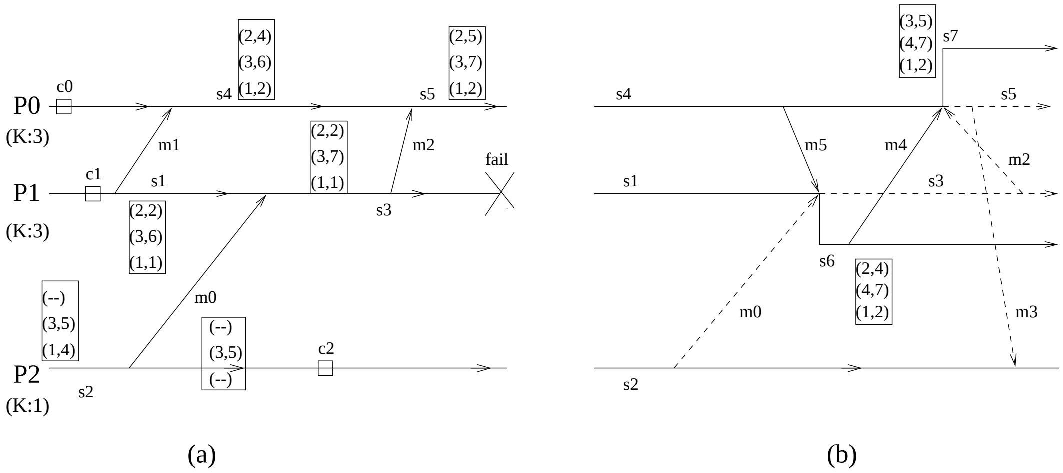


Figure 3.10: K-optimistic recovery: (a) Pre-failure computation (b) Post-failure computation

Figure 3.10 shows an example of the protocol execution. Dependency vectors are shown only for some states and messages. To avoid cluttering the figure, some messages causing state transitions are not shown. The K value for $P0$ and $P1$ is 3 and that for $P2$ is 1.

In Figure 3.10(a), $P1$ sends the message $m1$ to $P0$ in the state interval $s1$. $P0$ processes this message, starts the state interval $s4$ and sends a message $m5$ to $P1$ ($m5$ is shown in the Figure 3.10(b) only). In the state interval $s2$, $P2$ sends the message $m0$ to $P1$. However, the recovery layer delays the sending of $m0$ as it is dependent on two non-stable intervals. The message is sent after $P2$ makes its own interval (1,4) stable. $P1$ processes this message and sends the message $m2$ to $P0$. It performs some more computations and fails (shown by a cross). At the time of failure, it has logged all the messages received till the interval $s1$ and has not logged the message $m0$. During this time, $P0$ acts on the messages $m2$ and starts the interval $s5$.

The post-failure computation is shown in the Figure 3.10(b). On restart, $P1$ restores its last checkpoint $c1$, replays the logged messages and recreates the interval $s1$. It broadcasts the failure announcement (3,6) to other processes and increments its own incarnation number. $P1$ now processes message $m5$ resulting in the interval $s6$. $P1$ sends message $m4$ to $P0$. During this time, $P0$ sends the message $m3$ to $P2$. The recovery layer of $P0$ receives the message $m4$ before it receives the failure announcement from $P1$. Note that the message $m4$ is received by the recovery layer in state $s5$, but it is not delivered to the application. In the figure, arrows point to the state in which a message is delivered and not the state in which they are received. The second entry in dependency vector of $m4$ is $(4, 7)$, while the second entry in $P0$'s dependency vector in state $s5$ is $(3, 7)$. Therefore, $P0$ decides that $m4$ is inadmissible. Later, when $P0$ receives the failure announcement, it rolls back. $P0$ restores the checkpoint $c0$ and replays the logged messages, until, in state $s4$, it is about to process the message $m2$ that made it orphan. It discards $m2$ and increments its incarnation number. It does not send out any rollback announcement. Now message $m4$ is processed and interval $s7$ is started. On receiving message $m3$, $P2$ detects that $m3$ is orphan and discards it.

3.6 Properties of the Protocol

In this section, we prove our protocol correct and discuss its properties. All the lemmas and invariants presented before the Theorem 4 are used to for the correctness proof in the Theorem 4. The Theorem 5 shows that our protocol does not indefinitely postpone the sending of a message. The Theorem 6 establishes the meaning of K, i.e., given any message m released by its sender, the number of processes whose failure can revoke m is at most K.

In Figure 3.2, the function *get_state* returns a state interval of a given process corresponding to a given entity. This function is well defined only if the entries corresponding to the states of a process are unique. The following lemma shows that the function *get_state* is indeed well defined.

Lemma 2 *Different states of the same process have different entries.*

Proof. Whenever a new state is started in the routine *Process_message*, sequence number is incremented. In *Start_incarnation*, incarnation number is incremented. Storing *cur_inc* on stable storage prevents the loss of incarnation information in a failure. A failure before the completion of the routine *Start_incarnation* will result in the restoration of the same state. Therefore, reuse of the incarnation number does not matter. ∎

The converse of this lemma is not true. A state can have more than one entry. This happens to a state restored after a failure or rollback when the state's incarnation number is incremented.

The following lemma shows that stability is monotonic with respect to the happened before relation within the same process.

Lemma 3 $(s \underline{\rightarrow} w \wedge s.p = w.p \wedge stable(w)) \Rightarrow stable(s)$

Proof. First note that whenever an interval is started, it is added to the *State_list*. Also note that either *State_list* is empty or all the states in it belong to the same incarnation and are consecutive. This is because routine *Log_state* is called in the routine *Rollback*. Lemma is trivially true when s is same as w. Now we prove the lemma by induction on the number of states between distinct s and w.

Base Case (0 states): If s is not made stable in the call to *Log_state* that made w stable, then s must have been made stable in an previous call to *Log_state*.

Induction Step: Let $s \rightarrow w$. Let u be the state immediately preceding w. Now $stable(w)$ implies $stable(u)$ by induction hypothesis. Also, $stable(u)$ implies $stable(s)$ by induction hypothesis. Hence the result. ∎

The following lemma shows that the predicate *knows_stable* correctly detects a stable state.

Lemma 4 $knows\_stable(s, e, j) \Rightarrow stable(get\_state(j, e))$.

Proof. A state enters its entry into its *log_prog* in the routine *Log_state.* It does so, only after adding the current state to the stable state list. Therefore, lemma is true when $s.p$ is equal to j. Now we consider the case when $s.p$ is different from j.

$knows\_stable(s, e, j)$

$\Rightarrow$ { definition of *knows_stable* }

$seq\_num(s.log\_prog[j], e.inc) \geq e.seq$

$\Rightarrow$ { Definition of *seq_num* }

$\exists x : (e.inc, x) \in s.log\_prog[j] \ \wedge \ x \geq e.seq$

$\Rightarrow$ { e belongs to $s.log\_prog[j]$ implies that P_j has broadcast it. }

$\exists w : w.p = j \ \wedge \ w.c[j] = (e.inc, x) \ \wedge \ stable(w)$

$\Rightarrow$ { Definition of *get_state* }

$\exists w : stable(w) \ \wedge \ w.p = j \ \wedge \ get\_state(j, e).c[j].inc = w.c[j].inc$

$\wedge \ get\_state(j, e).c[j].seq \leq w.c[j].seq$

$\Rightarrow$ { Lemma 3 }

$stable(get\_state(j, e))$ ∎

The following lemma proves a similar result in the other direction. Before we continue, we need one more predicate definition. A set or an entry satisfies the predicate *broadcast* if it has been broadcast by some state.

Lemma 5 $stable(get\_state(j, e)) \Rightarrow \forall i : \exists s \in P_i : knows\_stable(s, e, j)$

Proof. Let w be the state returned by $get\_state(j, e)$. Let u be the first state of P_j

after w in which *Log_state* is called. We next prove that w and u have the same incarnation number. A new incarnation is started only due to a failure or a rollback. By our choice of u, a failure before u means that w is not stable. A rollback before u means that u is not the first state after w in which *Log_state* is called as *Log_state* is called in *Rollback*. Therefore, u and w have same incarnation number. Now,

$stable(w)$

$\Rightarrow$ { Above argument, *Log_state* is atomic }

$u.c[j].inc = e.inc \;\wedge\; u.c[j].seq \geq e.seq \wedge u.c[j] \in u.log\_prog[j]$

$\Rightarrow$ { Future insertions maintain the following property }

$\forall v : v.p = j \;\wedge\; u \underline{\rightarrow} v : [\exists x : (e.inc, x) \in v.log\_prog[j] \wedge x \geq e.seq\ ]$

$\Rightarrow$ { *Broadcast_log_prog* is called eventually. }

$\exists log\_prog : broadcast(log\_prog) : [\exists x : (e.inc, x) \in log\_prog[j] \wedge x \geq e.seq]$

$\Rightarrow$ {*Receive_log_prog* is called eventually }

$\forall i : \exists s \in P_i : \exists x : (e.inc, x) \in s.log\_prog[j] \wedge x \geq e.seq$

$\Rightarrow$ { Definition of *seq_num* }

$\forall i : \exists s \in P_i : seq\_num(s.log\_prog[j], e.inc) \geq e.seq$

$\Rightarrow$ { Definition of *knows_stable* }

$\forall i : \exists s \in P_i : knows\_stable(s, e, j)$ ■

The following invariant is at the heart of our protocol. It ensures that dependencies on unstable states are never set to NULL. Therefore when an unstable state is lost in a failure, this invariant helps in the detection of orphan states.

Invariant 1 $\forall s, u : (s \underline{\rightarrow} u \;\wedge\; \neg stable(s)) \Rightarrow ((u.c[s.p].inc = s.c[s.p].inc)$
$\wedge\; (u.c[s.p].seq \geq s.c[s.p].seq))$

Proof. Invariant holds trivially if s is same as u. Therefore, we consider s, such that s happened before u. We show that the above invariant holds initially. We also show that the invariant holds after the execution of a routine, if it holds before the routine is called. It is sufficient to consider the routines that create a new interval or modify the dependency vector.

Initialize: Invariant is trivially true, because for any initial state u and any state s, $s \not\rightarrow u$.

Process_message: Let the message m be sent by an interval w. Let interval v process m and start interval u. Incrementing the sequence number in the second operation in the routine leaves the invariant unaffected. Therefore, it is sufficient to show that the invariant holds after the *max* operation.

In general, dependency vector of w and m can be different as some entries in the dependency vector of m might be set to NULL in the routine *Check_send_buffer*. Happened before relation is defined between w and u, whereas dependency vector of u is updated by taking maximum of dependency vectors of v and m. In order to reason about happened before relation, sometimes we would like to use the dependency vector of w instead of dependency vector of m. The following claim shows that dependency vectors of m and w agree on the dependencies on the non-stable intervals.

Claim 1: $s \underline{\rightarrow} w \;\wedge\; \neg stable(s) \Rightarrow m.c[s.p] = w.c[s.p]$

Proof. $s \underline{\rightarrow} w \;\wedge\; \neg stable(s)$

$\Rightarrow$ { Invariant holds before this routine is called }

$(w.c[s.p].inc = s.c[s.p].inc) \;\wedge\; (w.c[s.p].seq \geq s.c[s.p].seq)$

$\Rightarrow$ { Definitions of *get_state* and happened before }

$s \underline{\rightarrow} get\_state(s.p, w.c[s.p])$

$\Rightarrow$ { $\neg stable(s)$, Lemma 3 }

$\neg stable(get\_state(s.p, w.c[s.p]))$

$\Rightarrow$ { Lemma 4 }

$\forall r : \neg knows\_stable(r, s.p, w.c[s.p])$

$\Rightarrow$ { First if condition in *Check_send_buffer* is not satisfied }

$m.c[s.p] = w.c[s.p]$ { End of Claim }

Now, $s \rightarrow u$

$\Rightarrow$ {By definition of happened before}

$s \underline{\rightarrow} v \vee s \underline{\rightarrow} w$ ... (I)

We consider the following three cases separately:

1) $v.c[s.p].inc = m.c[s.p].inc$

2) $v.c[s.p].inc > m.c[s.p].inc$

3) $v.c[s.p].inc < m.c[s.p].inc$

Case 1: $v.c[s.p].inc = m.c[s.p].inc$. From (I), there are two subcases to consider: 1.a) $s \underline{\rightarrow} v$, 1.b) $s \underline{\rightarrow} w$.

Subcase 1.a: $s \underline{\rightarrow} v \ \wedge \ \neg stable(s)$

$\Rightarrow$ { Invariant holds before this routine is called. }

$(v.c[s.p].inc = s.c[s.p].inc) \ \wedge \ (v.c[s.p].seq \geq s.c[s.p].seq)$

$\Rightarrow$ { $u.c[s.p] = max(v.c[s.p], m.c[s.p])$, case 1 }

$(u.c[s.p].inc = s.c[s.p].inc) \ \wedge \ (u.c[s.p].seq \geq s.c[s.p].seq)$

Subcase 1.b: By Claim 1, $m.c[s.p]$ is equal to $w.c[w.p]$. The rest of the proof is similar to the subcase 1.a .

Case 2: $v.c[s.p].inc > m.c[s.p].inc$. From (I), there are two subcases to consider: 2.a) $s \underline{\rightarrow} v$, 2.b) $s \underline{\rightarrow} w$.

Subcase 2.a: It is similar to subcase 1.a .

Subcase 2.b: $s \underline{\rightarrow} w \ \wedge \ \neg stable(s)$

$\Rightarrow$ { Invariant holds before this routine is called }

$(w.c[s.p].inc = s.c[s.p].inc) \ \wedge \ (w.c[s.p].seq \geq s.c[s.p].seq)$

$\Rightarrow$ { Claim 1: $m.c[s.p] = w.c[s.p]$ }

$(m.c[s.p].inc = s.c[s.p].inc) \ \wedge \ (m.c[s.p].seq \geq s.c[s.p].seq)$

$\Rightarrow$ { $admissible(m, v)$, case 2 }

$(m.c[s.p].inc = s.c[s.p].inc) \ \wedge \ (m.c[s.p].seq \geq s.c[s.p].seq)$

$\wedge \ knows\_stable(v, m.c[s.p], s.p)$

$\Rightarrow$ { Definition of $knows\_stable$ }

$knows\_stable(v, s.c[s.p], s.p)$

$\Rightarrow$ { Lemma 4 }

$stable(s)$

Case 3: $v.c[s.p].inc < m.c[s.p].inc$. Proof is similar to that of case 2.

This concludes the proof for the routine *Process_message*.

Start_incarnation: Let u be the state in which this routine is called. This routine is called inside routines *Restart* and *Rollback* only. Therefore, u is stable. We consider the following two cases: 1) $s.p = u.p$, 2) $s.p \neq u.p$.

Case 1: $s.p = u.p$

$s \rightarrow u \Rightarrow s \underline{\rightarrow} u$

$\Rightarrow$ { u is stable, stability is monotonic }

$stable(s)$

case 2: $s.p \neq u.p$

$s \rightarrow u$

$\Rightarrow$ { Antecedent: $\neg stable(s)$ }

$s \rightarrow u \ \wedge \ \neg stable(s)$

$\Rightarrow$ { Invariant holds before this routine, $s.p \neq u.p$, only $u.c[u.p]$ is modified in this routine }

$(u.c[s.p].inc = s.c[s.p].inc) \;\wedge\; (u.c[s.p].seq \geq s.c[s.p].seq)$

Receive_log_prog: Let u be the interval calling this routine. Modification of dependency vector of u can cause violations of invariant involving two kinds of states: 1) $s \to u$, 2) $u \to s$. As $u.p$'th entry is not modified in this routine, so it is sufficient to consider the first case. We need to consider state s only if the $s.p$'th entry is modified in this routine.

Now, $s.p$'th entry is modified.

$\Rightarrow$ { Test in the if condition }

$knows\_stable(u, u.c[s.p], s.p)$

$\Rightarrow$ { Antecedent, invariant holds before this routine is called }

$(u.c[s.p].inc = s.c[s.p].inc) \;\wedge\; (u.c[s.p].seq \geq s.c[s.p].seq)$

$\wedge\; knows\_stable(u, u.c[s.p], s.p)$

$\Rightarrow$ { Definition of $knows\_stable$ }

$knows\_stable(u, s.c[s.p], s.p)$

$\Rightarrow$ { Lemma 4 }

$stable(s)$ ■

The following invariant shows that a process never sets its own entry to NULL, as its own entry is needed to start its next interval.

Invariant 2 $\forall s : s.c[s.p] \neq NULL$.

Proof. Process $P_{s.p}$ starts with a non-NULL $c[s.p]$. It updates it by taking maximum with another entry or by incrementing it. Hence the result. ■

The following invariant ensures that our dependency tracking mechanism does not introduce any false dependencies.

Invariant 3 $s \underline{\not\to} u \Rightarrow ((s.c[s.p].inc \neq u.c[s.p].inc) \vee (s.c[s.p].seq > u.c[s.p].seq))$

Proof. If distinct s and u belong to the same incarnation of a process and s did not happen before u then u happened before s. Now within the same incarnation, sequence number is only increased. Hence the invariant holds in this case. Therefore, we assume that s and u belong to different processes.

We show that the above invariant holds initially. We also show that the invariant holds after the execution of a routine, if it holds before the routine is called. It is sufficient to consider the routines that create a new interval or modify the dependency vector.

Initialize: Let u be the initial interval.

$\Rightarrow$ { Assumption: $s.p \neq u.p$ }

$u.c[s.p] = NULL$

$\Rightarrow$ { Invariant 2 }

$u.c[s.p].inc \neq s.c[s.p].inc$

Process_message: Let the message m be sent by an interval w. Let interval v process m and start interval u. Incrementing the sequence number in the second operation in the routine leaves the invariant unaffected as $s.p$ is not equal to $u.p$. Therefore, it is sufficient to show that the invariant holds after the max operation.

Now, $u.c[s.p] = max(v.c[s.p], m.c[s.p])$

$\Rightarrow$ { Definition of max }

$u.c[s.p] = v.c[s.p] \vee u.c[s.p] = m.c[s.p]$

We consider the following two cases: 1) $u.c[s.p] = v.c[s.p]$, 2) $u.c[s.p] = m.c[s.p]$

Case 1: $u.c[s.p] = v.c[s.p]$.

Now, $s \not\rightarrow u$

$\Rightarrow$ { Definition of happened before }

$s \not\rightarrow v$

$\Rightarrow$ { Invariant holds before this routine is called }

$(s.c[s.p].inc \neq v.c[s.p].inc) \vee (s.c[s.p].seq > v.c[s.p].seq)$

$\Rightarrow$ { Case 1 }

$(s.c[s.p].inc \neq u.c[s.p].inc) \vee (s.c[s.p].seq > u.c[s.p].seq)$

Case 2: $u.c[s.p] = m.c[s.p]$. If $m.c[s.p]$ is NULL then $u.c[s.p]$ is NULL. This implies that $v.c[s.p]$ is also NULL and the case 1 applies. Therefore, we consider the case when $m.c[s.p]$ is not NULL. This implies that $m.c[s.p]$ is same as $w.c[s.p]$. Rest of the proof is similar to the case 1 .

Start_incarnation: let u be the state in which this routine is called.

Now, $s \not\rightarrow u$

$\Rightarrow$ { Invariant holds before this routine is called }

$(s.c[s.p].inc \neq u.c[s.p].inc) \vee (s.c[s.p].seq > u.c[s.p].seq)$

$\Rightarrow$ { Assumption $s.p \neq u.p$, only $u.c[u.p]$ is modified in this routine }

$(s.c[s.p].inc \neq u.c[s.p].inc) \vee (s.c[s.p].seq > u.c[s.p].seq)$

Receive_log_prog: Let u be the interval calling this routine. Modification of dependency vector of u can cause violations of invariant involving two kinds of states: 1) $s \not\rightarrow u$, 2) $u \not\rightarrow s$. As $u.p$'th entry is not modified in this routine, so it is sufficient to consider the first case. We need to consider state s only if the $s.p$'th entry is set to NULL in this routine. By Invariant 2, $s.c[s.p]$ is non-NULL. Hence the result follows. ■

The following lemma shows that the predicate *knows_orphan* correctly detects an orphan state.

Lemma 6 *$knows\_orphan(s, u) \Rightarrow orphan(u)$.*

Proof. $knows\_orphan(s, u)$

$\Rightarrow$ { Definition of *knows_orphan* }

$\exists j : \exists (t, x) \in s.iet[j] : (t = u.c[j].inc) \ \wedge \ (x < u.c[j].seq)$

Let w be the minimum state lost in the failure of P_j that resulted in the failure announcement entry (t, x). Now $w.p$ is same as j.

Then, $w.c[w.p] = (t, x + 1)$

$\Rightarrow$ { Definition of $knows\_orphan(s, u)$ }

$w.c[w.p].inc = u.c[w.p].inc \ \wedge \ w.c[w.p].seq \leq u.c[w.p].seq$

$\Rightarrow$ { Invariant 3 }

$\neg(w \not\rightarrow u)$

$\Rightarrow$ { w is a lost state }

$w \rightarrow u \ \wedge \ rolled\_back(w)$

$\Rightarrow$ { Definition of *orphan* }

$orphan(u)$ ■

The following lemma proves a similar result in the other direction.

Lemma 7 $orphan(u) \Rightarrow \forall i : \exists w \in P_i : knows\_orphan(w, u)$

Proof. Let v be lost in a failure and s be the state restored after that failure. Then we prove that s and v have the same incarnation number before the routine *Start_incarnation* is called by s. If s and v have different incarnation number then the routine, *Start_incarnation* must have been called by an intermediate state. Since s is the first state to call *Restart*, after loss of v, the routine *Rollback* must have been called in some intermediate state. As *Log_state* is called in *Rollback*, so all unlogged states including v are made stable. Then v cannot be lost in a failure. Therefore, s and v have the same incarnation number.

Now, $orphan(u)$

$\Rightarrow$ { Definition of orphan }

$\exists v : lost(v) \wedge v \underline{\rightarrow} u$

$\Rightarrow$ { Above argument, *Restart* is atomic, idempotent. }

$\exists v, s \; : \; lost(v) \wedge v \underline{\rightarrow} u \wedge broadcast(s.c[v.p]) \wedge s.c[v.p].inc = v.c[v.p].inc \wedge v.c[v.p].seq > s.c[v.p].seq$

$\Rightarrow$ { Reliable broadcast includes the execution of *Receive_failure_ann.* }

$\forall i : \exists w \in P_i : \exists e \in w.iet[v.p] \wedge e.inc = v.c[v.p].inc \wedge v.c[v.p].seq > e.seq$

$\Rightarrow$ { Invariant 1, not stable v }

$\forall i : \exists w \in P_i : \exists e \in w.iet[v.p] \wedge e.inc = u.c[v.p].inc \wedge u.c[v.p].seq > e.seq$

$\Rightarrow$ { Definition of $knows\_orphan$ }

$\forall i : \exists w \in P_i : knows\_orphan(w, u)$ ■

Two executions of a process are considered equivalent if their stable states and the sets of messages sent to other processes are same.

Lemma 8 *Given an arbitrary execution of a process and an arbitrary point of failure, there exists an equivalent execution in which the failure occurs just before the routine* Log_state *is called.*

Proof. A failure during execution of any routine that modifies volatile storage only is same as the failure before the execution of that routine.

Using the techniques in [46], all routines that modify stable storage can be made atomic. These routines are idempotent as well. A repeated execution of any routine in Figure 3.7 has the same effect as that of executing it only once, provided no new failure announcements are received during the repeated execution. If new failure announcements are received then repeated execution is equivalent to an execution in which all failure announcements are received before executing the routine under consideration. This follows from the fact that the set of the stable states is totally ordered and execution of the routine *Receive_failure_ann* has either no effect or causes one of the stable states to be restored. So repeated execution of a number of failure announcements is same as executing them once in an arbitrary order. Therefore, any failure is equivalent to a failure just before the routine *Log_state* is called. ■

The following theorem proves the correctness of our protocol.

Theorem 4 *The protocol rolls back all orphan states and orphan states only.*

Proof. As per Lemma 8, any failure is equivalent to a failure before executing the routine *Log_state*. Therefore, the only effect of a failure is that unlogged states are lost.

We show below that all orphan states are rolled back when failure announcement arrives. Also all orphan messages are discarded whether they arrive before failure announcement or after. Therefore, no state becomes orphan with respect to a failure after the arrival of the corresponding announcement.

In routine *Restart*, an announcement is made about the entry of the last stable interval of the last incarnation. On receiving this announcement, all processes roll back all states that satisfy the predicate $knows\_orphan$. By Lemma 7, all orphan states satisfy this predicate. Further, by Lemma 6, only orphan states satisfy this predicate. This ensures that the resulting system state is consistent. This point requires some elaboration. Let interval s be lost in a failure. One can imagine a protocol which rolls back a state u that is not dependent on s. A protocol that rolls back all states dependent on a lost state s can be wrong in the following way. It may not roll back a state w that is dependent on u. As it rolled back u, resulting system state will not be consistent.

As proved below, all messages that are orphan with respect to this failure are discarded. A message is orphan if its sending interval is orphan. A sending interval's dependency vector is attached with the message. As lost states are not stable, their entry in message's dependency vector is not set to NULL in the routine *Check_send_buffer*. Hence by using the Invariant 1, all orphan messages that have been received before the corresponding failure announcement are discarded when that failure announcement is received.

We assume that reliable delivery of failure announcement includes the atomic execution of the routine *Receive_failure_ann*. This means that received failure announcements are not lost afterwards. So all messages that are orphan w.r.t. this failure and that arrive after the failure announcement will be discarded upon their arrival in the routine *Receive_message*.

This completes the proof obligations stated earlier. ■

The following theorem shows that our protocol does not indefinitely postpone the sending of a message.

Theorem 5 *Each message in Send_buffer is either lost in a failure, or discarded as an orphan, or eventually sent.*

Proof. Consider a message m that is not lost in a failure and is not discarded as an orphan and is present in the *Send_buffer* of P_i. Let w be the maximum state of a process P_j on which m is dependent. If w is lost in failure then by Lemma 7, P_i will detect that m is orphan and will discard it. Else w will eventually become stable and by Lemma 5, P_i will eventually set $m.c[j]$ to NULL. As our choice of j was arbitrary, so the number of non-NULL entries in the dependency vector of m will become at most K and m will be sent. ■

The following theorem shows the meaning of K.

Theorem 6 *Given any message m released by its sender, the number of processes that can make the message orphan on their failure is at most K.*

Proof. In *Check_send_buffer*, the j'th entry of the dependency vector of a message m is set to NULL when the corresponding interval in P_j becomes stable. As per proof of Theorem 3, a failure of P_j cannot cause m to become an orphan. Since m is released when the number of non-NULL entries become at most K, the result follows. ■

3.7 Variations of the Basic Protocol

The K-optimistic protocol presented in previous sections is one of the possible applications of Theorem 3. This theorem can be used to implement many different policies. For example, P_i may be unwilling to roll back due to a failure of P_j. This can be enforced by P_i by blocking the delivery of any message that is commit dependent on any interval of P_j till that interval becomes stable. Interestingly, P_i may choose to become commit dependent on P_k while avoiding commit dependency on P_j, even though P_k may become commit dependent on P_j. This is because, on receiving a message from P_k, P_i can detect that P_k is passing an unstable dependency on P_j. That message delivery can be blocked by P_i till the dependency on P_j becomes stable.

Simulating a Failure

As discussed in Section 1.2, there are many scenarios like non-crash failures and software error recovery, where recreating pre-failure states is undesirable. This poses a problem for our protocol, because we are setting stable entries to NULL under the assumption that they can never be lost in a failure. But, now we need to simulate the loss of stable intervals. To do this, we add one more bit (initially 0) to each entry in the dependency vector. Instead of setting an entry to NULL, we simply set the corresponding bit to 1. Lexicographic comparison operation still remains the same. Incarnation numbers and state indices are compared to determine the maximum of two entries. Everything else in the protocol remains the same except that for the purpose of orphan detection, all entries including the stable ones need to be inspected. For example, suppose the second entry of $P1$ dependency vector is (2,6,0). It corresponds to entry (2,6) in the old notation. Now $P1$ receives the logging notification (2,8) from $P2$. Instead of setting (2,6,0) to NULL, it is changed to (2,6,1). Later on, if $P2$ were to simulate a failure and send the announcement

(2,4), $P1$ will know that it is an orphan by comparing (2,4) to the entry (2,6,1) in its dependency vector.

There is a clear limitation of this approach. It does not work when an entry from a lower incarnation is overwritten by an entry from a higher incarnation. This means that failures can be simulated only within the current incarnation.

An alternative approach to failure simulation is that in addition to logging on stable storage, an application may also need to satisfy some other conditions before it can declare an interval stable. For example, with latent errors, an interval becomes stable only after the maximum error detection latency period.

3.8 Experimental Results

So far we have mainly discussed the theoretical issues related to K-optimistic logging. This section presents the experimental results of a prototype implementation.

To our knowledge, there is no general answer to the question: What value of K should one use? It depends entirely on the application characteristics and the application. Some of the factors that play a crucial role are: communication pattern, message size distribution, message arrival rate, network bandwidth, stable storage server load, and failure probability. Given the wide range of these parameters, it is not possible to come up with a table showing the failure-free overhead and the recovery time for combinations of particular values of these parameters. Instead, we recommend that a prototype of the application be run with different values of K and be tested for different failure scenarios. Based on the observed behavior and the application requirements regarding maximum down time and failure-free overhead, appropriate value of K can be chosen. In the following sections, we discuss some particular applications and present the failure-free overhead and recovery time for the single failure case.

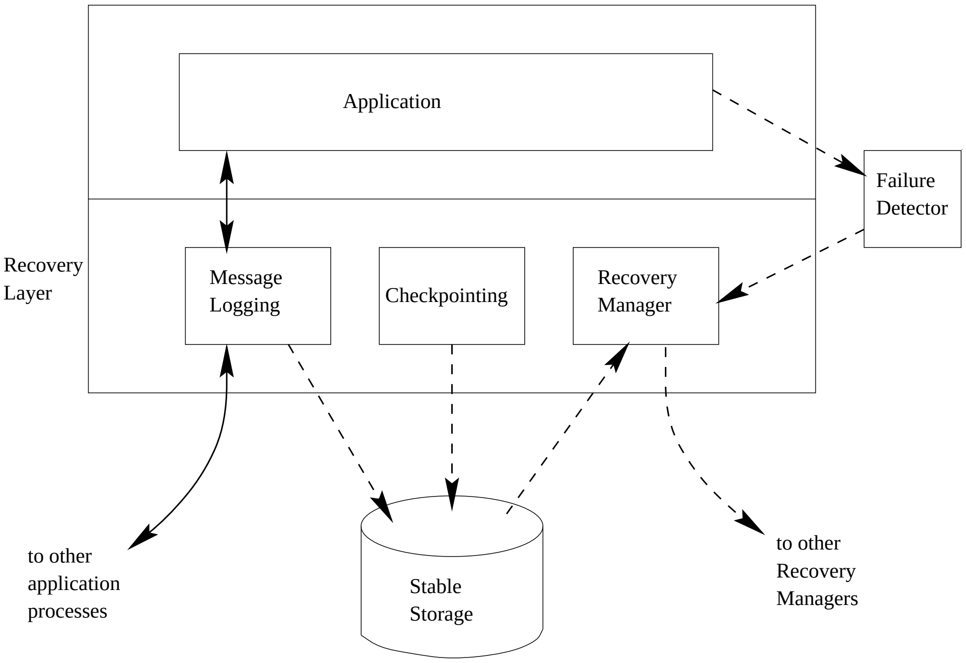


Figure 3.11: Recovery layer architecture.

3.8.1 Architecture

We have implemented a prototype of the K-optimistic protocol. Our architecture is shown in the Figure 3.11. An application is compiled with a recovery library that consists of a logging server, a checkpointing server and a recovery manager. Solid arrows show the flow of application messages while the dashed arrows show the messages required for the fault-tolerance. Application should periodically send an *I-am-alive* message to the failure detector. As per the diagram, recovery manager and application belong to the same process. Therefore, on detecting many consecutive missing I-am-alive message, the failure detector will start a new recovery manager which will load the latest checkpoint and pass the control to the application. Since our focus is on the message logging part, we have not implemented the checkpointing server and the failure detector. We simulate them appropriately, as explained in the Section 3.8.5.

3.8.2 Message Logging Policy

In traditional optimistic schemes, the volatile log is periodically flushed to stable storage. However logging at fixed interval does not work well for the K-optimistic scheme. This is because for the lower values of K, the logging needs to take place more often to give acceptable performance. For example, consider the case of K being 0. If messages are logged at a fixed interval of say 300 millisecond, then no process can send out messages faster than once in 300 millisecond.

The application progress is affected in a non-linear fashion with the varying logging frequency. Higher logging frequency may result in non-optimal use of the file server. Also, the application and the logging server may compete for the processor cycles and the network bandwidth. On the other hand, lower logging frequency may result in messages being held in *Send_buffer* for a long time. This implies that for a given value of K, one needs to experiment with different values of logging frequency to select the optimal value. However, this method of determining logging frequency does not work in presence of different message sizes, changing message arrival rates and varying system load.

To solve this problem, we have designed a novel message logging policy. Our policy asynchronously logs the very first message, right after it is received. After that, whenever a notification from the file server is received that the previous logging completed successfully, all the received messages since the previous logging are submitted to the file server for asynchronous logging. This policy automatically adapts to the changing system load. For a lightly loaded system, messages will be logged frequently. As the system load increases, logging frequency decreases correspondingly.

For K value of n, above logging policy is similar in spirit to the logging policy used in traditional optimistic protocols. Even for K value of 0, this policy works like the pessimistic protocol in [37]. In that protocol, the logging overhead is reduced by delaying the logging till the point where a message dependent on unlogged messages needs to be sent.

A related issue is that of logging progress notification frequency. It offers trade-offs similar to those discussed for the logging frequency. However, as the size of the logging notification message is much smaller than a typical application message (8 to $8n$ bytes, depending on the implementation), frequent notification results in negligible overhead compared to frequent logging. We also piggyback the logging progress information about the highest known incarnation of each process on every outgoing message. We found that this adds very little to the message processing overhead but helps in fast logging notification.

We have chosen a period of 500 millisecond for the logging notification, except when K equals n, for which notification period is 1 second. In the latter case, logging notification is needed only for the output commit and not for the progress of the computation.

3.8.3 Test Scripts

In our experiments, each process receives a message, sleeps for a while, sends a message and then blocks for the next message receive. In the beginning, an initiator process sends a message to all other processes. For a given experiment, the message size is fixed but compute time is chosen uniformly from a range. Compute time is the time between the processing of a message and the send of next message. It is inversely related to message frequency.

3.8.4 Application Parameters

We vary following parameters in our tests: message size, compute time and communication pattern. We consider these parameters because they are the main determinant of the trade-off provided by the K-optimistic protocol. The trade-off depends on two factors: how fast messages can be logged and how fast messages need to be logged. How fast messages can be logged depends on the message size. How fast the messages need to be logged depends on the message frequency. We later show that communication pattern also determines how fast messages can be logged.

Communication Pattern

We have tested our protocol for two different communication patterns. In test *Random*, the receiver of a message is chosen randomly by the sender, whereas in test *Neighbor*, processes are arranged in a ring and they alternately send messages to their left and right neighbors only. Tests *Random* and *Neighbor* were chosen because they are representatives of the many different applications studied in the literature [26, 55]. They represent two extreme communication patterns for distributed applications and if our protocol works well for these extremes, then it should work well for the patterns in between the extremes.

In *Random*, each process receives messages from all other processes. Therefore, if a single process were to fail, other processes will not slow down much as they will still be able to communicate with each other. On the other hand, in *Neighbor*, failure of a single process changes the topology from a ring to a doubly-linked list. The neighbors of the failed process are expected to slow down as their message intake is reduced by half. As a result, other neighbors of these neighbors will also slow down and so on, resulting in a slowdown of the entire application. In *Neighbor*, all processes receive equal number of messages and so they block and compute for approximately equal periods. On the other hand, in *Random*, some processes receive a little more messages than average while others receive a little less. This implies that some process may be blocked waiting for a message to process while some other process may always have a message to process when it needs one.

Message Size and Compute Time

We have selected specific message sizes and compute time to illustrate a wide range of applications. If the message logging time for most messages is less than the minimum computation time, then most of the messages will be logged before the application finishes processing them. As a result, very few messages will get lost in a failure and the recovery time will be dominated by the checkpoint restoration and the message replay time for the failed process. Therefore, the recovery time for different values of K should be similar. Also, during failure-free computation, very few messages should be held in the send buffer. Therefore, the overhead for different values of K should be similar and little. This overhead can be made arbitrarily small by selecting very high compute times. If the message logging time for most messages is more than the maximum compute time, then the overhead for lower values of K can be arbitrarily large depending on the actual values of logging and computation time.

For message size of 1K, compute time of 80 to 100 millisecond is more than the average logging time for both the tests. For message size of 10K, compute time of 80 to 100 millisecond is less than the average logging time for both the tests. For message size of 4K, compute time of 50-70 millisecond is less than the average logging time for *Random* and more than the average logging time for *Neighbor*.

3.8.5 Performance Evaluation

Performance Metrics

We measure the failure-free overhead by running the test with and without K-optimistic logging for different values of K. We measure recovery time as the difference in the average values of execution time without any failure and the execution time with a single failure. This definition implies that the recovery time may be similar for different values of K, even though the number of processes rolling back are different. This is because processes may roll back concurrently. Also, when one process rolls back, other processes may block. Therefore, the recovery time may change little if the blocked processes were to actually roll back a little.

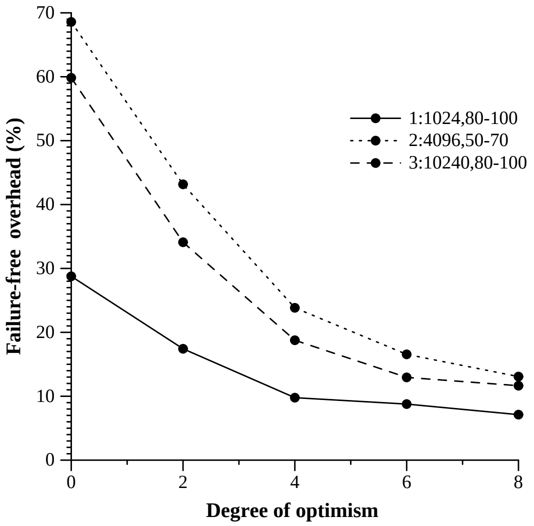
Experimental Settings

We use a network of IBM-250/25T workstations connected by a 10Mb/s Ethernet. Each workstation runs AIX 4.1.5 and is equipped with a 66-MHz PowerPC 601 processor, 32KB of data cache and 256MB of memory. A highly available NFS is used for stable storage.

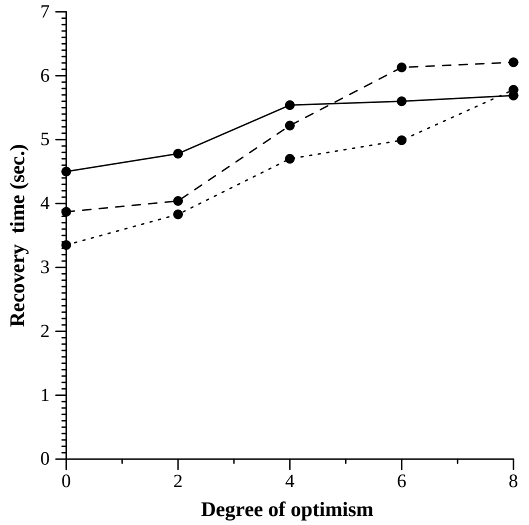
Measurement Methodology

All measurements are made with applications distributed across 4 machines with 2 processes per machines. All tests involved 20 trials. All results presented here are averages of middle 10 values for each test. The duration of a trial for the case of no message logging ranged from 24 to 40 seconds for different combinations of test parameters. Standard deviations for most measurements are under 1% of the average.

The occurrence of a failure is simulated after a process has processed 30 messages since a designated checkpoint. The latency of failure detection is heavily dependent on the implementation of the failure detector and the application environment. Since we consider only the single failure case, we have chosen to ignore this latency in comparing the recovery time for different values of K. Since the checkpoint size in our tests is very small, we simulate the cost of restoring a large checkpoint and reading the message log by blocking the application for 2 seconds while recovering from a failure. The value of 2 second was chosen based on the values reported in [55].

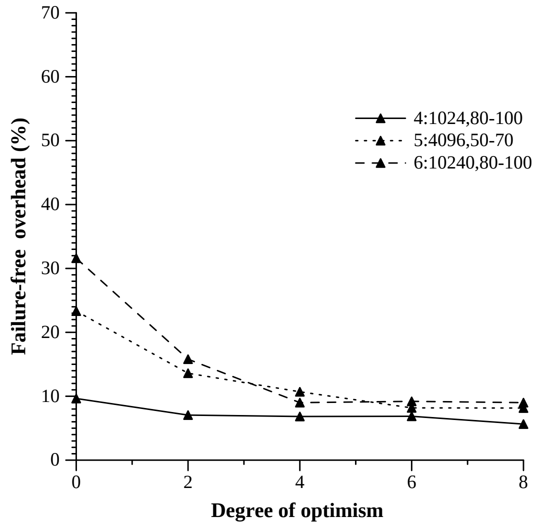


(a)

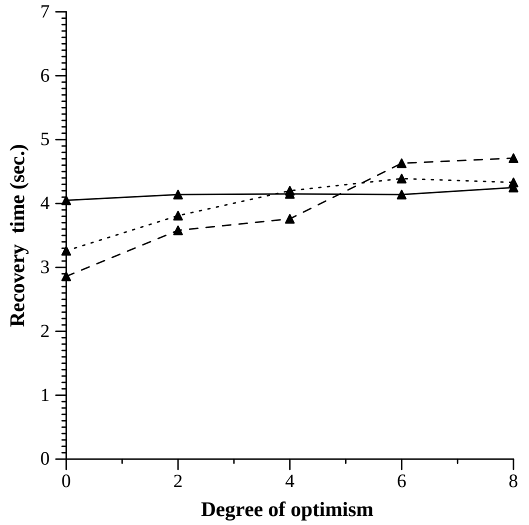


(b)

Figure 3.12: Performance results for application *Neighbor*. Legend consists of the experiment number, the message size in bytes and the compute time range in milliseconds.



(a)



(b)

Figure 3.13: Performance results for application *Random*. Legend consists of the experiment number, the message size in bytes and the compute time range in milliseconds.

Results

The results of our experiments are shown in Figures 3.12 and 3.13. Part (a) of the figures shows the failure-free overhead in percentage terms for various values of K. Part (b) shows the recovery time in seconds for the corresponding values of K. First entry in the legends is an experiment number that we use to discuss the results. Next two entries show the message size in bytes and the range in millisecond from which computation time for each message is uniformly chosen.

The failure-free overhead for traditional optimistic ($K = 8$) and pessimistic ($K = 0$) logging ranges from 6-14% and 8-66% respectively. These ranges are completely arbitrary and they can be much larger or smaller, depending on the application characteristics. For example, in [67], overhead of 3% is reported for completely pessimistic logging with a compute time of 2 seconds. The recovery time measurement should be used for illustration only, and not as the absolute values since we simulate the failure.

Discussion

Both failure-free overhead and recovery time graphs for experiment 4 are almost flat. This is to be expected, since in this experiment, almost all messages are logged before the application finishes processing them. Same logging pattern implies same failure-free overhead. Since no messages are lost in a failure, recovery time is also same for different values of K. In general, whenever compute time is much more than the average message logging time, we should expect the failure-free overhead and the recovery time graphs to be flat.

All other parameters being equal, the failure-free overhead for *Neighbor* is always more than that for *Random*. In *Neighbor*, all processes block and compute for similar period of time. Therefore, there are times when no process is trying to log messages while at other times, many processes try to log messages concurrently.

Compared to this, access to stable storage is more uniformly distributed over time for *Random*. Another reason is that in *Neighbor*, if one process is waiting for the current logging to finish to release its messages from the send buffer, then its neighbors also slow down. However, in *Random*, neighbors can receive messages from any other processes and can make progress.

We also note that the recovery time for *Neighbor* is always more than the recovery time for *Random*. This is because while a process is recovering, its neighbors receive messages only from their other neighbor, while in *Random*, they receive messages from many other processes.

For lower values of K, the time to restore the checkpoint and replay the message logs is always greater than the recovery time, which is defined as the difference in the running time of a failure-free run and a single-failure run. This is because while a process is recovering, other processes are making progress. Also, other processes are sending messages to the recovering process. Therefore, when the failed process finishes its replay, it will have many more messages to process without blocking for the want of a message to process.

Another interesting trend is that for the lower values of K and the same value of compute time, computation with smaller message size takes longer to recover. This is contrary to what one would expect if one were to define recovery time as the time to restore the checkpoint plus the time to replay the message logs. We explain this with reference to the experiments 1 and 3. But before that, let us understand the effect of a failure on an application completion time.

In general, processes compute most of the time and when they have no messages to process, they remain idle. When a failed process is recovering, it is also receiving messages from other processes. After the end of the replay, the failed process acts on these received messages and does not remain idle for quite some time. In this period, other processes receive message from the failed process at a rate faster than normal. As a result, overall computation proceeds at a rate faster than normal. More time a computation takes to finish in a failure-free run, less is the increase in completion time caused by a failure because the computation has more time to adjust to the disturbance caused by a failure.

Since messages of size 1K are logged faster than messages of size 10K, in the absence of any failure, experiment 1 takes less time to complete than experiment 3. Experiment 1 takes 37 seconds whereas experiment 3 takes 49 seconds. A failed process takes almost same time (5 seconds) to restore a checkpoint and replay messages in both experiments. Compared to experiment 1, experiment 3 has more time to adjust to the disturbance caused by this blocking. Therefore, the extra time taken to finish is more in experiment 1.

For higher values of K and same compute time, recovery time is lower for experiments with lower message sizes. This is expected because for lower message sizes, fewer messages are lost in a failure and as a result, fewer processes roll back compared to higher message sizes.

The failure-free overhead varies with varying message sizes in an intuitive manner. For *Random*, the failure-free overhead does not increase much as the K changes from 8 to 4. This is probably because most messages are dependent on at most 4 non-stable intervals. Therefore, for K value of 4 to 8, most messages are never held in the send buffer, resulting in similar failure-free overhead. However, logging progress in *Neighbor* is slow compared to *Random* for the reasons discussed earlier. Therefore, for experiments 2 and 3, failure-free overhead changes as K changes from 8 to 4. For experiment 1, small message size results in little change of failure-free overhead for higher values of K.

Finally, note that the configurations that give similar failure-free performance (5,6) give different recovery characteristics.

3.8.6 Selecting K

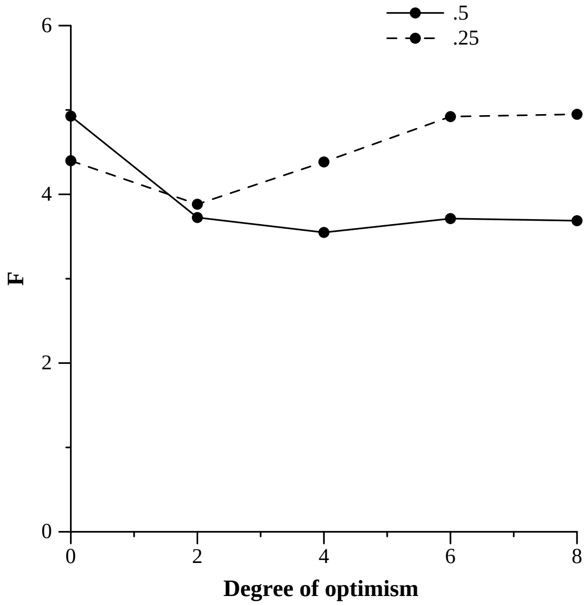


Figure 3.14: Selecting a K value for a given α. Numbers in the legend represent the α values.

At the beginning of this section, we proposed that to select the appropriate value of K for a given application, a prototype of the application should be run with different values of K and failure-free overhead and recovery time graphs should be obtained. After that, K can be chosen based on the constraints on the failure-free overhead or recovery time. For example, let us consider the results of the experiment 2, shown in Figure 3.12. If the system designer wants to minimize the recovery time while keeping failure-free overhead under 20% then he can choose K equal to 6. On the other hand if the goal is to have the maximum recovery time of 4 seconds while minimizing the failure-free overhead then K value of 2 can be used.

Another approach is to minimize an objective function of the failure-free overhead and recovery time. For example, let the objective function F be:

$$F(O, R) = \alpha O + (1 - \alpha)R$$

Arguments O and R represent the failure-free overhead and the recovery time in some normalized form and α is a parameter to be chosen by the system designer. As α changes from 0 to 1, we should expect the optimum K value to change from 0 to n. In Figure 3.14, we plot the function F for two different values of α. Argument O is obtained by dividing the failure-free overhead for experiment 2 by 10% and argument R is taken as recovery time. As both curves have a unique minimum, we select the K value of 4 for α equal to .5 and K value of 2 for α equal to .25.

3.9 Related Work

Strom and Yemini [61] started the area of optimistic message logging. The protocol presented in this chapter is similar in spirit to their protocol. They, however, did not define orphans properly and did not distinguish between failures and rollbacks. As a result, their protocol suffers from the *exponential rollback* problem, where a single failure of a process can roll back another process exponential number of times. For the same reason, they could not omit dependency tracking on stable states. They also assumed FIFO message ordering which is not required in optimistic protocols.

Johnson and Zwaenepoel [40] present a centralized optimistic protocol. Unlike most optimistic protocols including ours, their protocol does not require every received message to be logged. This can be advantageous in case average message size is much larger than average checkpoint size. They also use direct dependency tracking instead of transitive dependency tracking. This implies that instead of dependency vectors, only a small constant amount of information is piggybacked on each message. These advantages come at the expense of a centralized recovery manager, which itself needs to be made fault-tolerant.

Smith, Johnson and Tygar [59] present the first completely asynchronous optimistic protocol. Their protocol is completely asynchronous in that, in their system, neither a process is ever blocked from progressing, nor a message is ever blocked from being delivered. This is achieved by piggybacking on each message a data structure that is similar to our incarnation end table. This results in high failure-free overhead. In their protocol, no failure announcement is used. Since the incarnation end table is piggybacked on every message, each process learns about a failure when a message path is established between the restarted failed process and every other process. We believe that rather than letting orphan computation continue for a long time, it is better to announce failure and let every process know as soon as possible. In Strom and Yemini's and in our protocol, a process does not block after a failure, but a message may be blocked from delivery. In Chapter 4, we go one step further and block the failed process from recomputing till all non-failed processes acknowledge the receipt of failure announcement. This obviates the need for checking whether any message delivery should be blocked. Thus, it reduces the failure-free overhead at the expense of recovery time.

To address the scalability issue of dependency tracking for large systems, Sistla and Welch [60] divided the entire system into clusters and treated inter-cluster messages as output messages. Lowry et al. [45] introduced the concept of *recovery unit gateways* to compress the vector at the cost of introducing false dependencies. Direct dependency tracking techniques [60, 40, 38] piggyback only the sender's current state interval index, and are more scalable in general. The trade-off is that, at the time of output commit or recovery, the system needs to assemble direct dependencies to obtain transitive dependencies.

In Table 3.1 we present a comparison of our protocol with other optimistic protocols that track transitive dependencies. Since no other protocol bridges the gap between optimism and pessimism, we consider our protocol for K equal to n. Note that, using our ideas presented in [21], Smith and Johnson reduced the size of dependency vectors in their algorithm [58].

Our protocol generalizes several previously known protocols. For K equal

	Message ordering	Number of integers piggybacked	Number of concurrent failures	Number of rollbacks per failure	Asynchronous Recovery
Strom and Yemini 85	FIFO	$O(n)$	n	$O(2^n)$	Mostly
Sistla and Welch 89	FIFO	$O(n)$	1	1	No
Johnson and Zwaenepoel 90	None	$O(1)$	n	1	No
Peterson and Kearns 93	FIFO	$O(n)$	1	1	No
Smith, Johnson and Tygar 95	None	$O(n^2 f)$	n	1	Yes
Damani and Garg 96	None	$O(n)$	n	1	Mostly
Smith and Johnson 96	None	$O(nf)$	n	1	Yes

Table 3.1: Comparison with related work. (n is the number of processes in the system and f is the maximum number of failures of any single process.)

to n, our protocol reduces to the optimistic protocol presented in [21], while for K equal to 0, it reduces to the pessimistic protocol presented in [37]. The protocol in [6] can be thought of as a variant of our protocol where the server processes set K to 0 and the clients set K to n.

Although no parallel exists to our protocol in the area of message logging, in the area of checkpoint-based rollback-recovery, the concept of lazy checkpoint coordination [66] has been proposed to provide a fine-grain trade-off in-between the two extremes of uncoordinated checkpointing and coordinated checkpointing. An integer parameter Z, called the laziness, is introduced to control the degree of optimism by controlling the frequency of coordination. The concept of K-optimistic logging can be considered as the counterpart of lazy checkpoint coordination for the area of log-based rollback-recovery.

3.10 Summary

In this chapter, we have proved two fundamental results. First, with transitive dependency tracking, only the failures and not all rollbacks need to be announced. Second, only the dependencies on non-stable intervals need to be tracked. Based on these results, we have introduced the concept of K-optimistic logging that allows a system to explicitly fine-tune the trade-off between failure-free overhead and recovery efficiency. In such a system, given any message, the number of processes whose failures can revoke the message is bounded by K, and therefore K indicates the maximum amount of risk that can be placed on each message or equivalently the degree of optimism in the system. Traditional pessimistic logging and optimistic logging then become the two extremes in the spectrum spanned by K-optimistic logging.

Chapter 4

Transparent Recovery for Multi-Threaded Processes

In the previous chapter, we presented optimistic protocols for the systems where each process is single-threaded. The natural extensions of single-threaded optimistic protocols to multi-threaded processes lead to a trade-off between failure-free overhead and the degree of false causality in the system. We present a *balanced* approach that eliminates false causality while incurring low overhead.

4.1 System Model

The system model in Chapter 2.2 assumed that the processes are single-threaded. In this section, we extend that model to incorporate multi-threaded processes.

Each process contains a set of threads and a set of shared objects. Threads of different processes communicate only through messages. Threads of the same process communicate through shared objects and messages. Any other form of communication is allowed between threads, as long as it can be modeled using shared objects or messages. For example, wait-notify synchronizations can be modeled using messages.

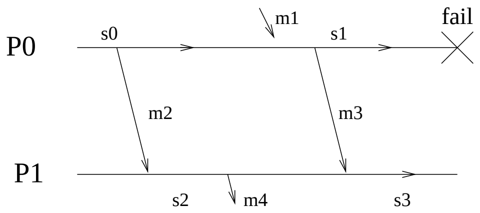
Threads of a process crash together. This happens not only in hardware crashes but also in most software crashes because threads share the same address space and are not protected from each other by the operating system.

The recovery system can restore the state of an individual thread or shared object to an old state without affecting the other threads or shared objects. This assumption will be discussed in Section 4.3.3.

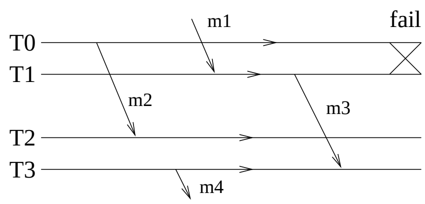
4.2 Extending Optimistic Recovery

We now extend the optimistic recovery protocol in Chapter 3 to multi-threaded environments. For the sake of simplicity, we consider the optimistic protocol for K equal to n. The protocol can be easily extended to incorporate the concept of K-optimism.

Strom and Yemini [61] presented the original optimistic protocol not in terms of processes, but in terms of *recovery units*. A recovery unit is a single unit of execution in optimistic recovery systems. A recovery unit fails as a unit and rolls back in response to another unit's failure. In Chapter 3, we chose individual processes as recovery units. In a multi-threaded environment, there are two natural candidates for the recovery unit: a process or a thread.



(a) Process-centric Logging



(b) Thread-centric Logging

Figure 4.1: Extending Optimistic Logging

4.2.1 Process-centric Logging

In treating a process as a recovery unit in a multi-threaded system, there is another source of non-determinism apart from the order of message receives. Depending on the scheduling, the threads may access shared objects in a different order. Therefore, after a failure, replaying the message log to a process is not sufficient to recreate the desired states.

To solve this problem, Goldberg et. al. [29] require that shared objects be accessed only in locked regions. The order in which threads acquire locks is logged. During a replay, the same locking order is enforced. This trace-and-replay technique has also been used in concurrent debuggers [44, 63].

Another approach has been used by Elnozahy and Slye [57]. They focus on uniprocessor multi-threaded environments in which the points of non-determinism can be reduced to the thread switches. Therefore, they log the order of thread switches and ensure that thread switches occur in the same order during replay. Again, this approach has been used in concurrent debuggers [50, 56].

Given that the non-determinism due to thread scheduling can be tracked and replayed, the general optimistic recovery approach described in Chapter 3 can be used with a process as a recovery unit.

The False Causality Problem:

An example of process-centric approach is shown in Figure 4.1(a). Receives of messages $m1$, $m2$ and $m3$ start the intervals $s1$, $s2$ and $s3$ respectively. When $P0$ fails and loses the state interval $s1$, $P1$ has to roll back the state interval $s3$. In the figure, the threads in each process are not shown, since processes are the recovery units.

Figure 4.1(b) shows the same scenario at level of threads. Process P1 consists of two threads T2 and T3. Therefore, $s3$ is an interleaving of the states of $T2$ and

$T3$. Suppose that after $m3$ was received, there was no shared object interactions between $T2$ and $T3$. So, only the states on $T3$ were really caused by $m3$ and needed to roll back. The states in $s3$ belonging to $T2$ were rolled back unnecessarily. This is due to *false causality* induced between the states in the two threads.

Besides causing unnecessary rollbacks during recovery, false causality has an unwanted effect in failure-free mode as well. In Figure 4.1(a), the output message $m4$ from $s2$ cannot be committed until $s0$ has become stable. However, the thread view in Figure 4.1(b) shows us that this was, in fact, unnecessary. The waiting was a result of false causality induced between $m2$ and $m4$. Thus, false causality also increases the latency of output commits.

How often does false causality arise? The answer depends on the frequency of shared object interactions between threads. Lewis and Berg [43] have divided multi-threaded programs into two main categories: *inherently multi-threaded programs* and *not obviously multi-threaded programs*. Inherently multi-threaded programs require multi-threading for ease of programming and not for speedup. These programs have highly independent tasks that are naturally expressed as threads. Some examples of such programs are: servers which handle multiple requests simultaneously, debuggers which monitor a program while keeping multiple displays active at the same time, and simulators which simulate different entities that operate simultaneously. The class of "not obviously multi-threaded programs" are those that require multi-threading for speedup on a multi-processor machine. Such programs have tightly-coupled threads that interact frequently through shared memory. Some examples are numerical programs and fine-tuning bottlenecks in existing code. Of these two categories, the inherently multi-threaded programs have highly independent threads which do not interact frequently, and therefore, would display false causality more often. Our focus is on this important category of multi-threaded applications.

Given that the false causality problem is an important concern, how can it be addressed? The problem arises because threads, which are independent units, are grouped together as a single unit. To solve this problem, we now study an approach that models each thread as a recovery unit.

4.2.2 Thread-centric Logging

Threads can be modeled as recovery units since they can be rolled back independently. Failure of a process is modeled as concurrent multiple failure of all its threads. This approach has been taken in [8] in the context of causal logging.

A number of important issues arise when treating threads as recovery units in the optimistic logging scheme. First, in addition to dependencies between threads due to messages, there are also dependencies caused by shared objects. These new dependencies must be tracked. Second, on a failure, just as a thread may have to roll back, a shared object may also have to roll back. Thus, orphan detection must be carried out for threads, as well as for shared objects. Third, both threads and shared objects must be restored to a checkpoint and replayed.

To address these issues, we describe a way to model shared objects using messages and threads. A fictitious thread is associated with each shared object whose state is the same as that of the corresponding object. Each method invocation on a shared object is modeled as a pair of messages between the invoking thread and the thread associated with the object. The first message is sent by the invoking thread and contains the method identifier and the method parameters. The second message is sent by the thread associated with the object and contains the return value of the method. This simplifies presentation of the protocols because messages are the only form of communication in this system.

Further, the simplified model deals with all the three issues mentioned above. The new dependencies are tracked by treating the shared object accesses as messages and associating a vector with each shared object. Similarly, orphan detection and replay of shared objects is done just as in threads.

Since we have dealt with the new issues, all that remains is to apply the general optimistic logging scheme. The computation in Figure 4.1(a) appears as that in Figure 4.1(b). When $T0$ and $T1$ fail, the thread-interval sending $m2$ is not lost in the failure. Therefore, only $T3$, is rolled back and $T2$ is not rolled back. Also, message $m4$ is committed without waiting for the interval sending $m2$ to become stable.

Therefore, the thread-centric approach reduces false causality. The dependency tracking overhead, however, is greatly increased. A main factor in this overhead is that, instead of $O(n)$ entries, each dependency vector now has $O(mn)$ entries (where m is the maximum number of threads per process). A more comprehensive discussion on this overhead will be presented in Section 4.4.

An Inherent Trade-off?

The process-centric and thread-centric approaches offer a trade-off between dependency tracking overhead and extent of false causality. This trade-off seems to be an inherent one as it arises from the choice of granularity of the recovery unit. A larger recovery unit introduces more false causality and has lower tracking overhead than a smaller one. In database systems, an analogous trade-off exists between lock maintenance overhead and extent of false causality while choosing the lock granularity. Surprisingly, in multi-threaded recovery, this trade-off can be avoided by a scheme, described next.

4.3 The Balanced Protocol

We observe that a recovery unit plays two distinct roles in optimistic recovery. The first role is that of a *failure unit*. The defining characteristic of a failure unit is that it fails as a unit. The second role is that of a *rollback unit*. A rollback unit can be rolled back and restored to a previous state independently. For example, in the process-centric protocol, the process was both the failure unit and the rollback unit, whereas in the thread-centric protocol, the thread was the failure unit and rollback unit.

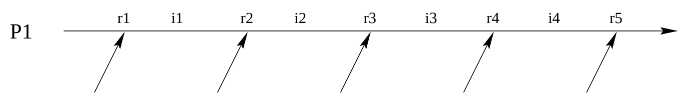
A general observation we make about optimistic recovery is that *to detect orphans, it is sufficient for a rollback unit to track its transitive dependency on a failure unit*. Then, the failure of a failure unit causes all orphaned rollback units to rollback, bringing the system back to a consistent state.

Thus, choosing a larger granularity failure unit reduces the dependency tracking overhead since there are fewer entities to track. Also, choosing a larger granularity rollback unit increases the extent of false causality since multiple entities are forced to roll back together. In the previous section, we saw that the trade-off between dependency tracking overhead and false causality depended on the granularity of the recovery unit. The separation of roles into failure units and rollback units avoids the trade-off, by choosing a larger granularity failure unit and a smaller granularity rollback unit. Applying this idea, our new *balanced* protocol uses a process as the failure unit and a thread as the rollback unit.

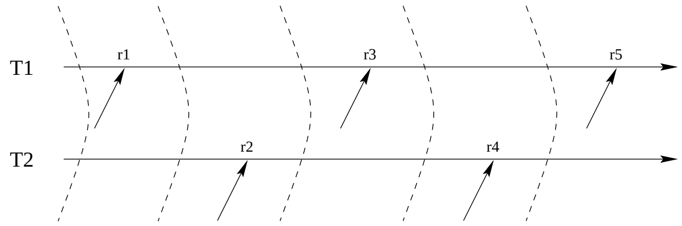
There is one issue that must be resolved before a protocol based on the balanced approach can be designed. In the balanced approach, our notion of state interval of a process must be reviewed. We now see how this is done using a general concept called *monotonic receive sets*.

4.3.1 Monotonic Receive Sets

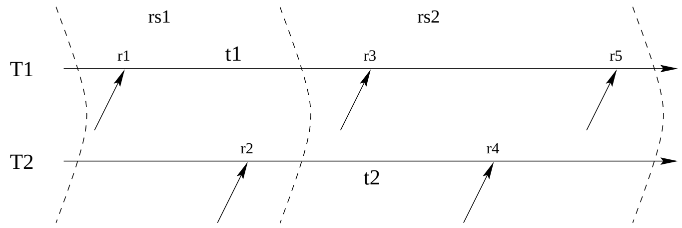
In the process-centric protocol, a state interval of a process is the set of states between two receive events. A state interval is caused by the receive event that precedes it. Thus, tracking dependencies on the state interval is really meant to track dependencies on the preceding receive event. For example, in Figure 4.2(a), the state interval i_1 of process P_1 corresponds to the receive event r_1.



(a) A Process-centric View of Receive Events



(b) A Total Order of Receive Events



(c) A Total Order of Receive Sets

Figure 4.2: Monotonic Receive Sets

We split process P_1 into its constituent threads T_1 and T_2 as shown in Figure 4.2(b). Here, the receive events are partially ordered. The process view in Figure 4.2(a) merely totally ordered the receive events in their order of occurrence in real time. This total ordering may also be represented by an advancing frontier that starts at the beginning of the process execution and advances one receive event at a time. This total order is followed when tracking causal dependencies so that only the maximum interval of a process as per the total order is tracked. This requires that the logging of these messages to stable storage must also follow the same total order and this is indeed the case in the process-centric protocol.

Now note that the frontier may be advanced multiple receive events at a time, resulting in a total order of "sets of receive events", called *receive sets*. For example, in Figure 4.2(c), receive set rs_1 contains two receive events r_1 and r_2. Further, the way in which the frontier advances from left to right ensures that the receive sets satisfy a special property that we call *monotonicity*. Monotonicity ensures that a receive event in a higher receive set cannot causally precede a receive event in a lower receive set.

Formally, let $\langle R, \rightarrow \rangle$ be the partially ordered set of receive events in a process, where $\rightarrow$ is the causally precedes relation within a process. Let $\langle RS, < \rangle$ be a partially ordered set where RS is a partition of R into sets of receive events, called *receive sets*. Let $rs(r)$ denote the receive set in RS that contains the receive event r. $\langle RS, < \rangle$ satisfies the *monotonicity property* if $<$ is a total order, and

$$\forall r_1, r_2 : \; r_1 \rightarrow r_2 \;\Rightarrow\; rs(r_1) \leq rs(r_2)$$

Although it is possible for each receive set to contain exactly one receive event, thus totally ordering all receive events, the advantages of grouping receive events into receive sets are two-fold:

1. First, while logging messages to disk, we notice that threads may naturally batch their receives together. Receive events logged in the same batch must be either both stable or both volatile. Thus it is sufficient to track dependencies on them together.

2. Second, totally ordering all receive events requires synchronization between threads. This is because each thread needs to be aware of the order of its receives relative to those of the other threads. This synchronization limits concurrency. By allowing groups of receive sets, we may reduce the synchronization since receive events in the same group need not be aware of their order relative to each other. In particular, if the ordering is maintained by a global counter, the number of times the counter needs to be updated is reduced.

We next define a new notion of state interval of a process. Just as the old notion of state interval was based on a receive event, the new notion is based on a receive set. A *state interval* of a process is defined with respect to a receive set as the union of the thread state intervals corresponding to the receive events in the receive set. A state interval is *monotonic* if its corresponding receive set is *monotonic*. For example, in Figure 4.2(c), the monotonic state interval corresponding to the monotonic receive set, rs_1, is the union of thread state intervals t_1 and t_2. Notice how this differs from the notion of process state interval in Figure 4.2(a) used in the process-centric protocol.

This new notion of state intervals along with the monotonic receive sets is sufficient for orphan detection, as shown in the next theorem.

Theorem 7 *In a multi-threaded system, tracking dependency on monotonic receive sets is sufficient for orphan detection.*

Proof. Consider a logging system that atomically logs all events belonging to a receive set. Also, the order in which receive sets are logged is same as the total order imposed on them. After a failure, all the logged events are replayed. We claim that the thread states restored in this fashion are non-orphan. If not, then let s be a restored thread state which is dependent on a lost state u. By definition of monotonicity, the receive set containing u is not ordered after the receive set containing s. Therefore, by the assumed logging property, u cannot be lost in a failure.

Given that a consistent state is restored, on receiving the failure announcement, states of other processes that track dependency on receive sets can find whether they are orphan. ■

In practice, it is easy to implement monotonic state intervals. Since threads share the same clock, one way to partition receive events is to use real time. The recovery system periodically marks the start of a new receive set by incrementing a global counter. All threads must read this counter whenever they need to be aware of the state interval they currently belong to. Since $(r_1 \rightarrow r_2)$ implies that r_1 occurs before r_2 in real time, this technique implements monotonic state intervals. We will use this technique in our protocol in the next sub-section. Clearly, by starting a new receive set at each receive event, we have the old notion of state interval used in the process-centric protocol.

4.3.2 The Balanced Protocol: Details

The protocol specifies the actions to be taken by the recovery system. The actions are divided into two categories, those for a process (failure unit) and those for a thread (rollback unit).

type *entry*: (*inc*: int, *seq*: int); // type representing state index with
// incarnation and sequence numbers

Thread T_l		
c	array[n] of entry;	// dependency vector
iet	array[n] of set of entry;	// incarnation end table

Process P_i		
sii	entry;	// state interval index
log	list of untyped objects;	// volatile log for messages and sii values
Stable_log	list of untyped objects;	// stable log for messages,
		// sii values, and checkpoints

Figure 4.3: Variables Used in the Protocol

Figure 4.3 shows the variables used in the protocol. Each thread maintains a dependency vector c and an incarnation end table *iet* in order to detect orphans. The *sii* indicates the current process-wide monotonic state interval index.

Thread T_l

Process_message:

log := log + (m, j);

c := max(c, m.c);

c[j] := sii;

Process P_i

Checkpoint:

log := log + sii;

Stable_log := Stable_log + log;

log := null;

Stable_log := Stable_log + s; // s: current state

sii.seq := sii.seq + 1;

Log_message:

Stable_log := Stable_log + log;

log := null;

Start_state_interval:

log := log + sii;

sii.seq := sii.seq + 1;

Figure 4.4: Protocol for Normal-mode Operation

The actions of the protocol may be divided into two types: normal mode and recovery mode. Figure 4.4 lists the normal mode protocol. For simplicity, all actions are assumed to be atomic.

First, in order to implement monotonic state intervals, periodically the recovery system starts a new state interval (**Start_state_interval**) by incrementing the global *sii* value. The old *sii* is queued in the log marking the end of the previous receive set in the log. On processing the next message, each thread T_l assigns its local $c[j]$ entry the value of this global *sii* (**Process_message**), thus keeping track of the monotonic state interval it belongs to. It is easy to verify that this results in monotonic state intervals as defined in the previous section.

Each thread keeps track of the highest monotonic state interval of each process that it is aware of using its local dependency vector c. The dependency vector mechanism is the same as before.

A received message is logged in the *log* (**Process_message**). This log totally orders all receives of all threads in their real time order, marking the end of receive sets by storing their *sii* values (**Start_state_interval**). Periodically, this volatile log is flushed to stable storage (**Log_message**).

Periodically, checkpoints are also taken. The volatile log is flushed to the stable log and a checkpoint is appended (**Checkpoint**). The old *sii* value is logged before the checkpoint and incremented after the checkpoint. This ensures that every checkpoint is exactly between two state intervals.

The recovery mode protocol is presented in Figure 4.5. On a crash failure, the crashed process restores its last checkpoint in the stable log, *Stable_log* (**Restart**). All threads are replayed from this point using messages and *sii* values from *Stable_log* up to the last complete monotonic receive set. Note that the monotonicity property of state intervals ensures that this recovered state is consistent (i.e. in the recovered state, each message received has been sent). This would not

Thread T_l

Receive_fail_ann(t,x,j): // receive (t,x) from P_j

Insert(iet[j],(t,x));

if (knows_orphan(s,s))

Check_orphan; // deletes orphan messages and states from the log

restore last checkpoint of T_l in Stable_log;

locally replay thread T_l using messages and sii

values after last checkpoint of T_l in Stable_log and log;

send *Ack_Announce* to P_j

Process P_i

Restart:

restore last checkpoint in Stable_log;

globally replay all threads in P_i using messages

and sii values after last checkpoint in Stable_log;

Broadcast_failure(sii)

wait for *Ack_Announce* from all;

sii.inc := sii.inc + 1;

sii.seq := 1;

Figure 4.5: Protocol for Recovery-mode Operation

be true, in general, if a partial monotonic receive set were replayed.

Next, the crashed process broadcasts an announcement of its failure to all threads of all other processes. The announcement includes its recovered state interval index indicating the end of that incarnation. This broadcast must be reliable in order to ensure that the system returns to a consistent state. Reliability may be ensured by repeating the broadcast periodically. The process then blocks, waiting for an acknowledgment from all threads. Once all of these are received, it starts a new incarnation by appropriately updating its state interval index.

When a thread receives a failure announcement, it first records the announcement in its incarnation end table *iet* (**Receive_fail_ann**). This will be later used in normal mode to discard orphan messages. It then decides if it is an orphan based on its dependency vector and its newly updated *iet*. If it is, it deletes all orphan entries from its stable and volatile logs. Next, it restores the last thread checkpoint from the stable log. It then replays itself to its latest state using the messages and *sii* values in the stable and volatile logs. This will bring it to the latest state that is not an orphan with respect to the received failure announcement. Note that the other threads and, in particular, the global *sii* remain unaffected by this action. Finally, it sends an acknowledgment to the sender of the failure announcement.

To complete the protocol, we must add logging progress notification to accommodate output commit. However, we omit these details because they are identical to those in traditional optimistic protocols.

An Example

An example of our protocol is shown in Figure 4.6. Threads $T0$, $T1$ belong to process $P0$ and $T2$, $T3$ belong to $P1$. The dashed arcs show the ends of the process state intervals $s1$, $s2$ and $s3$. Thread state intervals are $t1$ to $t6$. State interval indices of $s2$ and $s3$ are (1,7) and (1,4) respectively. When $P0$ fails, it loses the

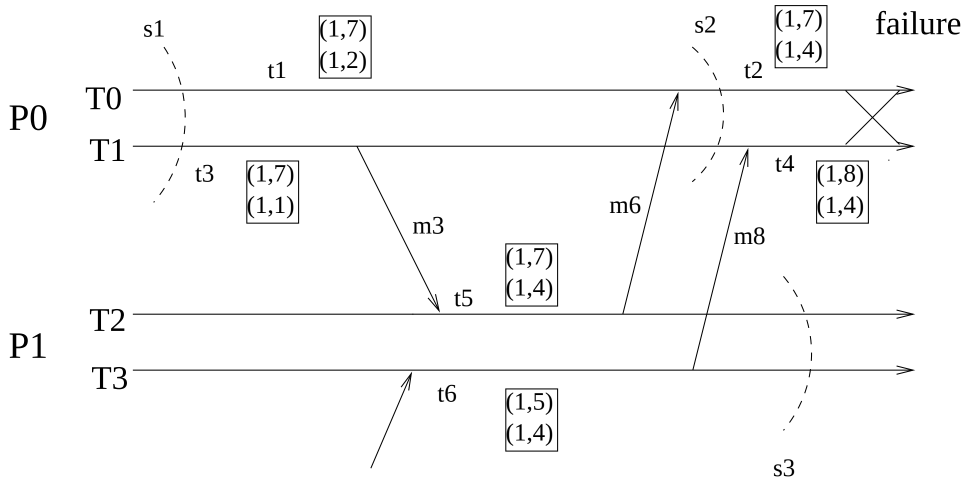


Figure 4.6: Example: Balanced Protocol

state interval s2. It broadcasts a failure announcement containing the index (1,6), corresponding to the state interval index of $s1$. On receiving this announcement, thread $T2$ rolls back the orphan state interval $t5$. Thread $T3$ remains unaffected by this failure. Note that, if $P1$ were to fail instead of $P0$, and lose the state interval $s3$, then both $T0$ and $T1$ will detect that they are orphans due to the entry (1,4) in their c's. This illustrates an important point of our protocol: in spite of belonging to the same process state interval and sharing a common index, thread intervals $t5$ and $t6$ act as independent rollback units and a single failure unit at the same time.

4.3.3 Implementation Issues

An important issue in implementing shared objects is defining a shared object access. Lower the number of accesses, lower the overhead in the balanced protocol. Frequently, multi-threaded programs are written using a discipline that ensures that each shared object is accessed only through methods, made mutually exclusive using an object lock. These mutually exclusive methods increase the granularity of a shared object access and provide a clear interface for tracking. Thus, instead of tracking every read and write to shared objects, we can track each method invocation on a shared object.

Each method invocation on a shared object is treated as a message from the thread to the shared object and the return from the shared object is treated as a message from the shared object to the thread. Thus, the recovery manager must intervene at both these points, either by code instrumentation or by modifying the run-time environment [18]. The log consists of the method and parameter values in case of the invocation and the returned values in the case of the return. While replaying a thread or a shared object, the logged values are replayed so that the thread or shared object recovers its state. This can again be achieved by either code instrumentation or run-time environment modifications.

4.4 Comparison with Related Work

There are two factors of interest while comparing various protocols: false causality and dependency tracking overhead.

We have already discussed the false causality problem in Section 4.2.1. To summarize: the false causality problem arises in the process-centric approach because threads are forced to roll back together even when they have low interactions between them. False causality is particularly a problem for a large class of applications that have low interactions between threads. The observable effects of false causality are: (1) delayed output commits, and (2) unnecessary rollbacks after a failure. Both the thread-centric and balanced approaches avoid false causality by allowing threads to roll back independently.

The price paid for avoiding false causality is the higher dependency tracking overhead. This overhead is in three forms: space overhead, time overhead, and message size overhead. Table 4.1 summarizes the relative overheads of the various protocols. The overhead of checkpointing is common to all protocols and hence it is not shown in the table.

	Space Overhead	**Time Overhead**	**Message Size Overhead**	**False Causality**
Process-centric(I)	$O(s + en)$	$O(c + en)$	$O(n)$	yes
Process-centric(II)	$O(o + en)$	$O(o + en)$	$O(n)$	yes
Thread-centric	$O(mn(o + e))$	$O(mn(o + e))$	$O(mn)$	no
Balanced	$O(n(o + e))$	$O(n(o + e))$	$O(n)$	no

Process-centric(I) is the process-centric protocol using Slye & Elnozahy [57]

Process-centric(II) is the process-centric protocol using Goldberg et al. [29]

n is the number of processes

m is the maximum number of threads and shared objects per process

e is the maximum number of message receive events per process execution

o is the maximum number of shared object accesses per process execution

s is the maximum number of thread switches per process execution

c is the time overhead for maintaining a software counter

Table 4.1: Comparative Evaluation of Overheads

As discussed before, there have been two implementations of the process-centric approach: Slye & Elnozahy [57], and Goldberg et al. [29]. Slye & Elnozahy use a software counter to track the thread switches. Therefore, the space overhead consists of $O(s)$ space to log all thread switching information and $O(en)$ space to store dependency vectors for each receive event. The time overhead consists of the total extra time the recovery protocol requires to execute. This involves the time to save checkpoints, log thread switching information, log dependency vectors. Therefore, the time overhead is proportional to the space overhead. The message size overhead is $O(n)$ since the dependency vector has n entries, one per process.

Goldberg et al. log the order of shared memory accesses so that they can be deterministically replayed. Therefore, the space overhead is $O(o+en)$ with the $O(o)$

component accounting for the log made on each shared memory access. The time overhead is proportional to space overhead. The message overhead remains $O(n)$ as before.

For the thread-centric approach, we assume that the overheads of checkpointing are similar for thread and process checkpoints. In practice, thread checkpoints may take additional time overhead to separate the thread local state from the process address space. Another method would be to simply take process checkpoints and extract the thread checkpoints when required. Since shared object accesses are treated as message receives, the space overhead to log the dependency vectors is $O(mn(o+e))$ since each vector has $O(mn)$ entries. The time overhead is similar. The message size overhead is now $O(mn)$.

The balanced approach reduces the vector size from $O(mn)$ in the thread-centric approach to $O(n)$. All overheads are similar to the thread-centric case replacing mn by n.

The saving in space, time and message size overhead of the balanced protocol with respect to the thread-centric protocol is substantial because mn is potentially a very large quantity compared to n. Each individual thread and shared object in the system is accounted for in mn. Since both protocols achieve the same elimination of false causality, the balanced protocol should always be preferred to the thread-centric protocol.

Compared to the process-centric protocol of Goldberg et al., the balanced protocol has the same message size overhead, but higher space and time overhead. This is because each shared object access in the balanced approach logs a $O(n)$ vector instead of constant information. With respect to the process-centric protocol using Slye & Elnozahy's technique, the space and time overhead is also expected to be higher since there are usually much fewer thread switches than shared memory accesses. However, as in Section 4.2.1, only the applications that have low thread interaction suffer greatly from false causality. For these applications, the increase in time and space overheads of balance protocol is low because the number of shared object accesses is low. Thus, the process-centric protocol should be used for applications with high thread interaction, and, therefore, low false causality effects. The balanced protocol should be used for the class of applications that have low thread interaction, where the extra space and time overhead is outweighed by the saving in false causality effects.

4.5 Summary

Attempting to extend traditional optimistic protocols to multi-threaded scenarios leads to a choice of either process-centric logging or thread-centric logging. This leaves us with a trade-off between false causality and high tracking overhead ($O(nm)$ dependency vector size).

This trade-off is avoided by noting the distinction between failure units and rollback units, and choosing processes as failure units and threads as rollback units. This allows threads to *do the tracking* while processes are the ones *being tracked*. We design a protocol based on these observations that reduces the dependency vector size to $O(n)$ without incurring false causality.

Finally, we introduce the general notion of monotonic state intervals as a generalization of the usual notion of state intervals. It is sufficient to track dependencies on monotonic states intervals rather than state intervals. This coincides with the batching of message logs to disk and also reduces the extent of synchronization between threads in determining state intervals.

Chapter 5

Application-Specific Recovery for Optimistic Computations

The protocols presented in Chapters 3 and 4 are independent of any particular application characteristics. They can be used with any message passing distributed application. The overhead of providing fault-tolerance can sometimes be reduced by exploiting the particular characteristics of a given application. Optimistic computations present one such opportunity. In optimistic computations, a process avoids blocking until an event by guessing its outcome. If the guess turns out to be correct, the optimism pays off. However, if it is wrong, all computation that follows the wrong guess is undone. Optimistic recovery schemes rely on a similar concept, and therefore can be conveniently integrated with optimistic computations. We demonstrate this integration in the context of distributed simulation.

5.1 Overview

In an event-driven distributed simulation [27], each logical process (LP) represents a simulation object and schedules events for other objects by sending messages. The correctness of a simulation requires that each object should execute all of its events in the order of increasing time-stamps. In conservative distributed simulation, an object processes an event only when it is sure that no event with a higher time-stamp will arrive in future. In contrast, in optimistic distributed simulation, each object optimistically processes the next available event, guessing that an event with a lower time-stamp will not arrive. If this guess turns out to be wrong and a *straggler*, i.e., an event with a lower time-stamp arrives, the object has to roll back all the processed events that have time-stamps higher than that of the straggler.

Although an application-independent recovery layer can be implemented below the optimistic simulation system described above, more efficient recovery can be achieved by modeling a failure as a straggler event. This straggler has a time-stamp equal to the time-stamp of the latest checkpoint saved on stable storage. In this model, computation lost due to a failure can be treated in the same way as computation rolled back due to a straggler. No extra dependency tracking needs to be done for recovery purpose. Thus, the fault-tolerance overhead is reduced.

5.2 Model of Simulation

We consider an event-driven optimistic simulation. Each simulation object corresponds to a logical process (LP). The execution of an LP consists of a sequence of states and a state transition is caused by an event execution. In addition to causing a state transition, executing an event may also schedule new events for other LPs (or the local LP) by sending messages. When an LP $P1$ acts on a message from $P2$, $P1$ becomes dependent on $P2$. This dependency relation is transitive.

An LP periodically saves its checkpoints on stable storage. After a failure, an LP restores its last checkpoint from stable storage and starts executing from there.

The arrival of a straggler causes an LP to roll back. A state that is lost in a failure, rolled back, or transitively dependent on a lost or rolled back state, is called an *orphan state*. We model a failure as a straggler event with a time-stamp equal to the time-stamp of the latest checkpoint saved on stable storage. We formally define *orphan* states as follows:

$straggled(s) \equiv$ state s was rolled back due to a straggler

$stable(P) \equiv$ time-stamp of the last stable checkpoint of LP P

$failure(P) \equiv$ event scheduled for LP P at time $stable(P)$

$orphan(s) \equiv \exists u : straggled(s) \ \wedge \ u \underline{\rightarrow} s$

In this model lost states are treated as straggled states. A message sent from an orphan state is called an *orphan message*. For correctness of a simulation, an LP must execute all its events in the order of increasing time-stamp, all orphan states must be rolled back, and all orphan messages must be discarded. To distinguish the computation before and after a rollback, we say that an LP starts a new *incarnation* after each rollback.

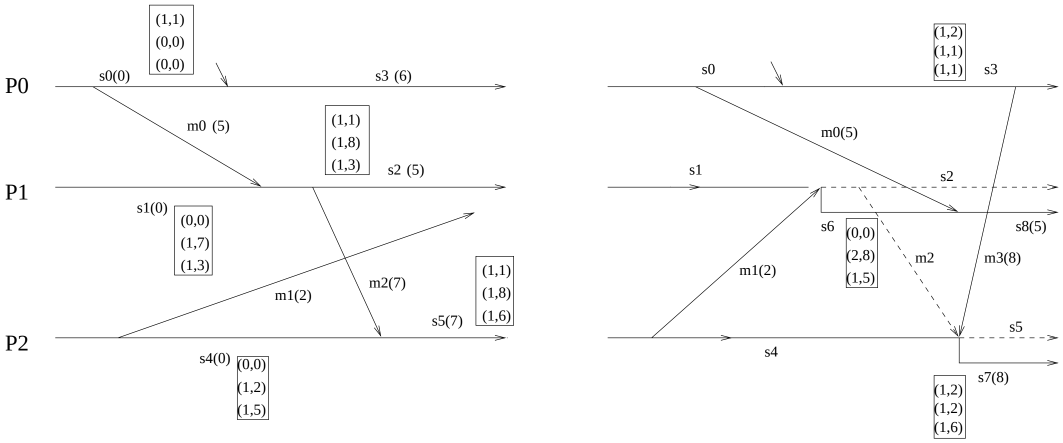


Figure 5.1: A Distributed Simulation. (a) Pre-straggler(failure) computation. (b) Post-straggler(failure) computation

An example of distributed simulation is shown in Figure 5.1. Solid horizontal lines indicate useful computation, and dashed horizontal lines indicate rolled back computation. Simulation state intervals are numbered from $s0$ to $s8$ and they extend from one message receive to the next. Next to a state, its virtual time is shown in parentheses. In optimistic simulation, simulation time is called virtual time. Next to a message, virtual time of the event scheduled by the message is shown. The rectangles in the figure correspond to dependency vectors and will be explained later.

In Figure 5.1(a), $s0$ schedules an event for $P1$ at time 5 by sending the message $m0$. $P1$ optimistically executes this event, resulting in the state transition from $s1$ to $s2$. $P1$ schedules an event for $P2$ at time 7 by sending message $m2$. Then $P1$ receives the straggler message $m1$, which schedules an event at time 2. Execution after the arrival of this straggler is shown in Figure 5.1(b). $P1$ rolls back and restore the state $s1$. It takes the actions needed for maintaining the correctness of the simulation, which, for our scheme, consists of broadcasting a rollback announcement. $P1$ acts on $m1$, resulting in state $s6$. $P1$ then acts on $m0$ and starts the interval $s8$. Upon receiving the rollback announcement from $P1$, $P2$ realizes that it is dependent on a rolled back state and it rolls back and restores the state $s4$. The orphan message $m1$ is discarded. $P2$ then acts on $m3$, resulting in state $s7$.

We now describe a simulation in a failure-prone system. Assume that the system has performed the computation shown in Figure 5.1(a), but $P1$ has not yet received the message $m1$. Let $P1$ fail before it receives the message $m1$. It loses its volatile memory, which includes the knowledge about processing message $m0$. Figure 5.1(b) shows the post-failure computation. $P1$ restores its last stable state $s1$. It broadcasts a failure announcement. On receiving this announcement, $P0$ and $P2$ resend the messages $m0$ and $m1$ as they might have been lost in the failure. $P2$ realizes that it is dependent on a lost state and rolls back and restores state $s4$. This time, $P1$ processes $m0$ and $m2$ in the correct order. This shows how we handle a failure and a straggler in the same way.

5.3 The Fault-Tolerant Optimistic Simulation Protocol

In this section, we present the details of an optimistic simulation protocol in which the recovery layer is integrated with the simulation layer. Although the protocol is similar to protocols given in previous chapters, there is an important optimization that is useful for simulations.

The number of logical processes in a simulation can be very high. The overhead of separately accessing stable storage for each checkpoint of each LP may be prohibitive. Therefore, we club LPs into clusters and take checkpoint of entire clusters. The idea of clustering has already been used in [54] and [10]. In [54], inter-cluster execution is conservative, whereas intra-cluster execution is optimistic. In [10], inter-cluster execution is optimistic, whereas intra-cluster execution is sequential. In a sequential execution, across LPs, events are processed in the order of increasing time-stamps. We employ optimistic inter-cluster execution. Our scheme works with both sequential and optimistic policy for intra-cluster execution. For purpose of exposition, we assume that intra-cluster execution is sequential.

Since the simulation inside a cluster is sequential, the state of a cluster at a given virtual time is well-defined. This state includes the input messages in all the LPs input queues. The state also includes the cluster output queue, that is described later. Clusters periodically save their state on stable storage. For presentation purpose, we use the term 'state of a cluster'. In an implementation, state of an LP needs to be saved on stable storage only if it has changed since the last state saving operation.

We need to modify the notations used so far in the dissertation. Since a cluster is implemented as a physical process, i,j refer to cluster numbers. Remaining notations remain the same: t refers to incarnation number; s,u refer to state; P_i refers to the cluster i ; m refers to a message and e refers to an event. We refer to intra-cluster messages as 'internal' messages and inter-cluster messages as 'external' messages.

```
Cluster P_i :

type entry = (int inc, int seq) ;          // incarnation, state sequence number
var  cvt : real;                           // cluster virtual time
     c : array[n] of entry;                // dependency vector
     iet : array[n] of set of entry;       // incarnation end table
     CIQ : set of pointers ;               // cluster input queue for external messages
     COQ : set of messages ;               // cluster output queue for external messages

          // Following variable is stored in stable storage
     cur_inc : int;                        // current incarnation number
Initialize :
    cvt = 0 ;        cur_inc = 1 ;
    ∀ j : c[j] = (0,0) ;
    c[i] = (1,1) ;
    ∀ j : iet[j] = {} ; // empty set
    CIQ = {} ; COQ = {} ;
```

Figure 5.2: Data Structures used by a cluster

5.3.1 Data Structures

The data structures used by a cluster are shown in figure 5.2. We describe the main data structures below:

Dependency Vector: A dependency vector c is used to keep track of transitive dependencies. In general, the dependency vector of the sending state needs to be attached with each message to correctly track transitive dependencies. We reduce this overhead by making the observation that with clustering, dependency vector needs to be attached with inter-cluster messages only. For intra-cluster messages, it is sufficient to track send time because the receiving state's dependency vector is always greater than or equal to the sending state's dependency vector.

Incarnation End Table: To detect orphan messages, an *incarnation end table* (*iet*) is used. It is similar to *iet*'s used in previous chapters.

Cluster Input Queue: All received messages, whether external or internal, are kept in an LP's input queue. In addition to this, references to received external messages are kept in a cluster level input queue (CIQ).

Cluster Output Queue: There is one output queue per cluster called Cluster Output Queue (COQ). Only inter-cluster messages are kept in COQ. Intra-cluster messages are not saved in any per LP output queue.

5.3.2 Actions Taken by an LP

Actions taken by an LP are shown in Figure 5.3. Before carrying out an event scheduled by a message, the dependency vector of the cluster is updated (**Execute_message**). Messages are send by calling the routine **Send_message**. In addition to the receive time (virtual time at which corresponding event should be executed), internal messages also carry the send time, which is same as the local virtual time (*lvt*) of the sending LP. Instead of send time, external messages carry the dependency vector of the sending cluster. If a cluster receives a straggler announcement (called *token*) from another cluster then all the LPs in the cluster need to roll back orphan states and discard orphan messages (**Rollback**).

5.3.3 Actions Taken by a Cluster

Actions taken by a cluster on receiving an external message are shown in the figure 5.4. Upon receiving an external message m, P_i discards m if m is an orphan.

Logical Process (LP) : // belongs to cluster P_i

var lvt : real ; // local virtual time

$input\_queue$: set of message ;

Execute_message(m) :

$lvt = cvt = m.receive\_time$; // cvt is the cluster virtual time

if m is an external message **then**

$\forall\ j$: $c[j] = \max(c[j], m.c[j])$;

$c[i].seq = c[i].seq + 1$;

Act on the event scheduled by m ;

Send_message(m)

if intra_cluster(m) **then** send($m, lvt, m.receive\_time$);

if inter_cluster(m) **then** send($m, c, m.receive\_time$);

Rollback(ts) // Roll back all states with virtual time $> ts$

Restore the latest state s such that $s.lvt \leq ts$;

Discard the states that follow s ;

// Discard orphan messages

$\forall m \in$ input_queue: **if** $m.send\_time > ts$ **then** discard m;

Figure 5.3: Actions of an LP

Receive_message(m) :

if $\exists j, x : ((m.c[j].inc, x) \in iet[j]) \wedge (x < m.c[j].seq)$ **then**

discard m ; // m is orphan

return ;

Copy m in input queue of the destination LP;

In the CIQ, add a reference to m;

if $m.receive\_time < cvt$ **then** //m is a straggler

Let s be the latest cluster checkpoint such that $s.cvt \leq m.receive\_time$;

Broadcast($(s.c[i])$);

// P_i receives its own broadcast and rolls back.

Block till all clusters acknowledge ;

Figure 5.4: Cluster actions on receiving an external message

This is the case when, for some j, P_i's iet and the j'th entry of m's dependency vector indicate that m is dependent on a rolled back state of P_j. A straggler for any LP in the cluster is considered a straggler for the entire cluster. If P_i detects that m is a straggler, it broadcasts a token containing the i'th entry (t, seq) of the dependency vector of its highest checkpoint s such that the virtual time of s is no greater than the receive time of m. The token indicates that all states of incarnation t with sequence number greater than seq are orphans. States dependent on any of these orphan states are also orphans. For simplicity of presentation, we show P_i rolling back in figure 5.5 (after it receives its broadcast). In practice, P_i should roll back immediately.

Steps taken by a cluster on receiving a token are shown in figure 5.5. On receiving a token (t, seq) from P_j, P_i acknowledges the receive and enters the token in its iet. P_i discards all orphan messages in the cluster input queue. A message is an orphan if it is dependent on a rolled back state of P_j. If the cluster is orphan then it restores the maximal non-orphan state. This is done by rolling back orphan LPs and discarding orphan messages in each LPs input queue in routine **Rollback**. All orphan external messages are discarded using CIQ. After the cluster's rollback, cluster incarnation number (saved in cur_inc) is incremented. To survive failures, cur_inc is stored in stable storage.

```
Receive_token(t, seq) from P_j :
  Send acknowledgment ;
  insert(iet[j], (t, seq)) ;
  ∀m ∈ CIQ :        // discard orphans
    if (m.c[j].inc == t) ∧ (m.c[j].seq > seq) then discard m ;
  if (c[j].inc == t) ∧ (c[j].seq > seq) then
      // the cluster is orphan
    Let s be the latest checkpoint such that
      s.c[j] ≤ (t, seq) ; // s: highest non-orphan state
    ∀lp ∈ cluster: lp.Rollback(s.cvt) ;
    c = s.c ;
    // start a new incarnation
    cur_inc = cur_inc + 1;
    c[i].inc = cur_inc ;
```

Figure 5.5: Cluster actions on receiving a token

Steps taken by a cluster on restarting after a failure are shown in figure 5.6. A failure is handled in the same way as a straggler. After a failure, the cluster is restarted from its last checkpoint on stable storage. The cluster broadcasts a token containing the index of the restored state. Other clusters react to this token in the

Restart // after failure

Restore last checkpoint s from stable storage;

Broadcast($c[i]$);

Block till all clusters acknowledge;

Figure 5.6: Cluster actions on restarting after a failure

same way as they do to the token due to a straggler.

5.3.4 Differences Between a Failure and a Straggler

There is one important difference between a failure and a straggler, which we have not shown in the protocol for clarity. In a failure, a cluster loses its volatile state, i.e., its *iet* and all messages that it has received but not acted on till its last stable checkpoint. Therefore, on learning about the failure, other clusters must resend messages to the failed cluster. Of these messages, the failed cluster should replay only those messages, which it did not act on before the last checkpoint.

This protocol can handle an arbitrary number of concurrent failures. When a processor fails, all clusters on that processor need to be restarted as if each one of them have failed independently.

5.3.5 Properties of the Protocol

A problem with optimistic protocols is that they are more prone to software crashes because a programmer may not anticipate all unexpected messages that may arrive in an optimistic system. In particular, the probability of crash increases if a state is allowed to become dependent on two *rollback inconsistent* states [51]. A rolled back state is called rollback inconsistent with the states that occur after the corresponding rollback [51]. For example, in Figure 5.1, $s2$ is rollback inconsistent with $s8$. The next theorem shows that our protocol avoids this problem.

Theorem 8 *A state cannot become dependent on two rollback inconsistent states.*

Proof: After a rollback, a process blocks till it receives acknowledgment of its rollback announcement from all processes. Therefore, all processes receive the rollback announcement before becoming dependent on a post-rollback state. Now, as per the routine **Receive_token** in Figure 5.5, any state dependent on a rolled back state is rolled back on receiving the corresponding token. Hence no state can become dependent on two rollback inconsistent states. ■

The next theorem shows that our protocol correctly completes simulation in the presence of failures.

Theorem 9 *At the end of a simulation, the following conditions are true:*

- *All LPs have received all the messages that they would have received in a sequential simulation.*
- *All LPs have processed all the messages in the increasing order of their receive time.*
- *All orphan states have been rolled back.*
- *All orphan messages have been discarded.*

Proof: To simplify the presentation, we assume that each cluster contains only a single LP. The proof can easily be extended to multiple LPs. Actions taken after receiving two tokens commute with each other and also the actions taken after receiving a token commute with a failure. Therefore, f concurrent failures are not different from the case where f processes fail one after another, such that between failures each process takes no action other than receiving failure announcements. Hence, we only consider the single non-concurrent failure case.

We have modeled a failure as a straggler event. A failure also results in loss of volatile state. We do not use any of the lost information even if a failure were truly a straggler event. The only information we need is the received messages and the *iet* entries. This information is collected from other processes. The only tricky case is when the sender itself has failed and it cannot resend some message as the state that sent that message is *lost*. This is harmless because according to the protocol, messages sent from a lost state are also *orphan* and they anyway need to be discarded upon their receive.

Our remaining proof obligation is to show that in absence of failures, our protocol handles the straggler messages and orphan states correctly. This follows from the proof of Theorem 4 in Chapter 3. ■

5.3.6 Stable Global Virtual Time (SGVT)

Global Virtual Time (GVT) is defined as the virtual time such that no state prior to GVT will ever be rolled back [11]. Traditional methods for computing GVT do not work in presence of crashes. A crash of a cluster may result in the restoration of a state with the virtual time less than the GVT.

Our modeling of failure as a straggler event can be directly used in the standard GVT algorithm to come up with a value, which we call SGVT, such that no state with virtual time less than SGVT will ever be rolled back. Since failure is treated as a straggler, a potential failure of process P can be treated as an unacknowledged message with time-stamp $stable(P)$.

GVT is approximated by taking the minimum of receive times of all unacknowledged messages and the local virtual times of all process. We make the observation that GVT can be approximated by taking the minimum of receive times of all unacknowledged messages and all unprocessed messages in input queues of all processes, which for a process P, is denoted by $unacked(P)$ and $unprocessed(P)$

respectively. We define SGVT as follows:

$$T_i \equiv \min\{stable(P_i), unacked(P_i), unprocessed(P_i)\}$$

$$SGVT \equiv \min\{\forall i : T_i\}$$

Theorem 10 *No state with virtual time less than SGVT can ever be rolled back.*

Proof: Every rollback has a first cause in a straggler or a failure. A failure cannot restore a state with a time-stamp less than the global minimum of *stable*. A straggler cannot have a time-stamp less than the global minimum of *unacked* and *unprocessed*. Hence the result follows. ■

In addition to being useful for fossil collection (garbage collection in recovery terminology) and output commit, SGVT has another application. We make the observation that only those entries need to be kept in the dependency vector whose associated states have virtual time greater than SGVT. Dependency on a state with virtual time less than SGVT need not be tracked because the corresponding state will never be rolled back. This results in the reduction of the overhead associated with dependency vectors. In fact, dependency vectors start with only one entry (process's own entry). As processes interact with one-another, size of dependency vector starts increasing. However, SGVT also keeps on increasing. Therefore, we expect the average number of entries in dependency vectors to be small.

5.3.7 Overhead Analysis

Our scheme incurs the following overheads for providing fault-tolerance:

Accessing stable storage: We need to periodically save checkpoints on stable storage. This is a necessary cost in absence of redundant resources like those used for replication. We save checkpoints asynchronously. Therefore, computation is not blocked when stable storage is being accessed.

Dependency information: We tag a dependency vector with each inter-cluster message. We expect the number of inter-cluster messages to be much smaller than the total number of messages. The size of dependency vector is $O(n)$ entries, where n is the number of clusters in the system. But as explained in section 5.3.6, we expect the number of entries to be much smaller in practice.

Cluster output queue: Inter-cluster messages are saved in a per cluster COQ. This overhead is much smaller than the overhead of per LP output queue in traditional optimistic simulation protocol like Time Warp [36].

Clustered rollback: Rollback of a single LP means rollback of the entire cluster. This slows down the simulation. Each cluster, however, rolls back at most once in response to each straggler or failure. There is no possibility of avalanche of antimessages or echoing [47]. This compensates for the slowdown owing to the clustered rollback.

5.4 Multi-Threaded Systems

The protocol presented in this chapter is in terms of single-threaded processes. If multi-threading is available, then a cluster can be implemented using threads. Although, in Chapter 4, we eliminated false causality between threads, a multi-threaded simulation cluster will have false causality problem. This is because, the elimination of false causality in Chapter 4 was based on the assumption that threads of a process fail together. In a simulation, however, one object may receive a straggler without other objects receiving a straggler. Therefore, either the complete dependency tracking needs to be done, or the false causality among objects will be present. These remarks are true not only for optimistic simulation, but for any optimistic computation.

5.5 Related Work

Distributed simulation methods can be divided in two main categories: conservative and optimistic. In conservative simulation, an object processes an event only when it is sure that no event with a higher time-stamp will arrive in future. A detailed discussion of different conservative simulation methods can be found in [27]. That survey also describes several optimistic simulation methods.

Time Warp is the most popular optimistic simulation scheme [36]. In Time Warp, dependency tracking is not used. Each LP maintains an output queue, where it keeps a copy of each message that it has sent. On receiving a straggler, each LP sends an antimessage corresponding to each orphan message in its output queue. Each antimessage annihilates the corresponding message. Except for that, an antimessage acts just like an ordinary message. Therefore, an antimessage can also become a straggler. This may result in an avalanche of antimessages and cascading rollbacks. These problems are not present in our scheme. Our scheme has less memory overhead and simple memory management algorithms. Time Warp may perform better than our scheme if the number of objects in the system is very large.

There has not been much discussion in simulation literature about fault-tolerance. In the seminal paper on Time Warp [36], Jefferson mentions that processes may coordinate to take a consistent snapshot and save it on stable storage. When any process fails, the entire system may roll back to the last snapshot. In contrast, our method rolls back only those states that are dependent on a lost state, and thus minimizes the wasted computation. In [4], a replication based strategy is presented to provide fault-tolerance. Our scheme has much lower overhead than that of the replication.

Some degree of fault-tolerance was built in the original Time Warp Operating System. Signal handlers were installed to catch exceptions. Once a signal indicating an error was caught, partially processed events were cleaned and processing was resumed. This approach takes care of some errors, but not all. It very well supplements our solution but does not replace it.

5.6 Summary

In this chapter we demonstrated that the cost of recovery in optimistic computation can be reduced by integrating the recovery layer with the application layer. This concept was illustrated with the help of optimistic distributed simulation. In the process we also developed a new simulation protocol that does not use anti-messages and avoids cascading rollbacks. The techniques used are generic and can be used for other optimistic computations as well.

Chapter 6

Conclusions

In this dissertation, we have developed a number of optimistic protocols. These protocols are based on message logging and checkpointing. This chapter presents a summary of the main contributions of the dissertation and discusses the directions for possible future research.

6.1 Contributions

We have developed a formal model to reason about failure-prone computations. In this model, a number of efficient optimistic protocols are designed. We have used the same model to show that the recovery can be made more efficient by exploiting the application-specific characteristics. We next discuss the contributions in more detail.

6.1.1 Efficient Optimistic Protocol in Single Threaded Systems

We have established that if transitive dependency tracking is employed, then announcing only failures, instead of all rollbacks, is sufficient for correct recovery. Since each process receives only one failure announcement, it rolls back at most once in response to a given failure. This results in more efficient recovery compared to traditional methods where a process can roll back exponential number of times in response to a single failure. Other researchers have also used our result to improve their protocols [58]. We have further shown that any protocol that employs transitive dependency, need not track dependencies on stable states. This result reduces the amount of information that needs to be piggybacked on each message.

6.1.2 Bridging the Gap Between Optimism and Pessimism

We have introduced the concept of K-optimism that provides a trade-off between recovery time and logging overhead, with traditional optimistic and pessimistic logging being the two end-points of the spectrum. As the value of K moves from 0 to n, the recovery time goes up with a corresponding decrease in the logging overhead. The parameter K can be dynamically tuned to adjust to a changing environment.

6.1.3 Efficient Optimistic Protocol in Multi-threaded Systems

Optimistic logging protocols can be extended to multi-threaded processes in two natural ways: process-centric and thread-centric. These two approaches together present a trade-off between false causality and tracking overhead. We have designed a *balanced* protocol that eliminates false causality while incurring low overhead. The main intuition is that processes fail independently and are thus failure units and that threads may be rolled back independently and are thus rollback units.

6.1.4 Efficient Recovery for Optimistic Computations

We have shown how to achieve low cost fault-tolerance by integrating optimistic recovery and optimistic simulation. The techniques used are generic and can be extended to other optimistic computations.

6.2 Future Research Directions

We next discuss a number of ways in which the results in this dissertation can be extended.

6.2.1 Generalizing the Balanced Approach

In Chapter 4, we considered failure and rollback units in the context of processes and threads. We can also select these units at coarser granularities. For simplicity, let us consider a system consisting of non-threaded processes. In such a system, a process can be a rollback unit while a processor can be a failure unit. To achieve this, an implementation of monotonic receive sets is required. This can be done in a number of ways: by using a centralized server or by explicit cooperation of processes. Further research is required to evaluate these methods. Regardless of the implementation of monotonic receive sets, dependency tracking overhead is likely to decrease , while the increase in false causality among processes will increase.

The issue of false causality needs further elaboration. In case of a hardware crash, entire processor indeed crashes as a unit. In case of a software crash of a single process, all processes on the corresponding processor have to simulate a crash, and hence the false causality between them.

Further generalizing, a local area network can be a failure unit in a wide area environment. A process can still be a rollback unit.

6.2.2 Recovery in Weak Consistency Systems

In this dissertation, we assumed that system state needs to be consistent at all times. Certain applications may be willing to tolerate certain inconsistencies if that will result in a substantial reduction in the fault-tolerance overhead. For example, a phone company may be willing to lose occasional call records to reduce operation costs. Some other application may be willing to tolerate a small fraction of outputs as orphans. New recovery techniques need to be designed for these applications.

6.2.3 Combining Optimistic and Causal Techniques

In Section 1.2, we discussed a number of scenarios where ability to roll back a process is desirable. In Section 3.7, we presented a method to introduce rollback capability in pessimistic protocols. In [3], the author states that a causal protocol can be modified to have roll back capability. It is desirable to develop a causal protocol with explicit rollback capability, that will handle most failures without rolling back non-failed processes but will be able to roll back processes in special circumstances.

6.2.4 Reconstructing Dependency Information from Other Processes

In optimistic logging, information about unlogged messages of a failed process is assumed to be completely lost. Some information, however, is available with other non-failed processes. All orphan processes in the system have received dependency vectors along with the messages that made those processes orphan. Although, complete dependency information cannot always be reconstructed from these vectors, it can be done some times. Even when it cannot be done, partial dependency information can still be reconstructed. A new optimistic protocol can be designed to take advantage of this information.

Bibliography

[1] A. Avizienis. Design of Fault-Tolerant Computers. *AFIPS Conference Proceedings, 1967 Fall Joint Computer Conference*, Vol. 31, 1967, 733-743.

[2] A. Avizienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, Vol. 11, No. 12, 1491-1501, Dec. 1985.

[3] L. Alvisi. Understanding the Message Logging Paradigm for Masking Process Crashes. Ph. D. Thesis, Dept. of Computer Science, Cornell University, January 1996.

[4] D. Agrawal and J. R. Agre. Replicated Objects in Time Warp Simulations. *Proc. Winter Simulation Conference*, 657-664, 1992.

[5] L. Alvisi, B. Hoppe, and K. Marzullo. Nonblocking and Orphan-Free Message Logging Protocols. *Proc. 23rd Fault-Tolerant Computing Symp.* , 145-154, 1993.

[6] M. Ahamad and L. Lin. Using Checkpoints to Localize the Effects of Faults in Distributed Systems. *Proc. 8th Symp. on Reliable Distributed Systems*, 66-75, 1989.

[7] L. Alvisi and K. Marzullo. Message Logging: Pessimistic, Optimistic, and Causal. *Proc. 15th Intl. Conf. on Distributed Computing Systems*, 229-236, 1995.

[8] L. Alvisi, and K. Marzullo. Deriving Optimal Checkpoint Protocols for Distributed Shared Memory Architectures. *Selected Papers, International Workshop in Theory and Practice in Distributed Systems*, K. Birman, F. Mattern and A. Schiper editors, Springer-Verlag, 111-120, 1995.

[9] ANSI X3.105. American National Standard for Data Encryption Algorithm (DEA), American National Standards Institute, 1981.

[10] H. Avril and C. Tropper. Clustered Time Warp and Logic Simulation. *Proc. 9th Workshop on Parallel and Distributed Simulation*, 112-119, 1995.

[11] S. Bellenot. Global Virtual Time Algorithms. *Proc. Multiconference on Dist. Simulation*, 122-127, 1990.

[12] A. Borg, J. Baumbach, and S. Glazer. A Message System Supporting Fault Tolerance. *Proc. 9th ACM Symp. on Operating Systems Principles*, 90-99, 1983.

[13] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault Tolerance Under UNIX. *ACM Trans. Comput. Syst.*, 7(1):1–24, February 1989.

[14] A. Baratloo, P.-Y. Chung, Y. Huang, S. Rangarajan, and S. Yajnik. FilterFresh: Hot Replication of Java RMI Server Objects. *Proc. USENIX The Fourth Conference on Object-Oriented Technologies and Systems (COOTS)*, 1998.

[15] A. W. Burks, H. G. Goldstine, and J. von Neumann. Preliminary Discussion of the Logical Design of an Electronic Computing Instrument. Draft Report 1946. Published in C. G. Bell and A. Newell, *Computer Structures: Reading and Examples*, 92-119, McGraw-Hill, 1971.

[16] K. P. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*, New York: IEEE Computer Society Press, 1994.

[17] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. on Computer Systems*, 3(1): 63-75, Feb. 1985.

[18] J. D. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. *2nd SIGMETRICS Symp. on Parallel and Distr. Tools* 48 - 59, Aug, 1998.

[19] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, March 1996.

[20] *www.zdnet.com/pcweek/news/0223/26mrsa.html*

[21] O. P. Damani and V. K. Garg. How to Recover Efficiently and Asynchronously When Optimism Fails. *Proc. IEEE Int. Conf. Distributed Comput. Syst.*, pages 108–115, 1996.

[22] O. P. Damani and V. K. Garg. Fault-Tolerant Distributed Simulation. *Proc. 12th Workshop on Parallel and Distributed Simulation*, 38-45, 1998.

[23] O. P. Damani, Y. M. Wang and V. K. Garg. Optimistic Distributed Simulation Based on Transitive Dependency Tracking. *Proc. 11th Workshop on Parallel and Distributed Simulation*, 90-97, 1997.

[24] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Tech. Rep. No. CMU-CS-96-181, Dept. of Computer Science, Carnegie Mellon University (also available at ftp://ftp.cs.cmu.edu/user/mootaz/papers/S.ps), 1996.

[25] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent Rollback Recovery with Low Overhead, Limited Rollback and Fast Output Commit. *IEEE Trans. on Computers*, 41 (5): 526-531, May 1992.

[26] E. N. Elnozahy and W. Zwaenepoel. On the Use and Implementation of Message Logging. In *Proc. 24th IEEE Fault-Tolerant Computing Symp.*, pages 298–307, 1994.

[27] R. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10), 30-53, Oct. 1990.

[28] M. J. Fischer, N. Lynch and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2), 374-382, April 1985.

[29] A. P. Goldberg, A. Gopal, K. Li, R. E. Strom, and D. F. Bacon. Transparent Recovery of Mach Applications. *1st USENIX Mach Workshop*, 1990.

[30] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* San Mateo, CA: Morgan Kaufmann, 1993.

[31] V. K. Garg and A. I. Tomlinson. Using Induction to Prove Properties of distributed Programs. *Proc. 5th IEEE Symp. on Parallel and Distributed Processing*, 478-485, 1993.

[32] V. K. Garg and B. Waldecker. Detection of Weak Unstable Predicates in Distributed Programs. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 3, pp. 299-307, March 1994.

[33] Y. Huang and C. Kintala. Software Implemented Fault-Tolerance: Technologies and Experience. *Proc. IEEE Fault-Tolerant Computing Sym.*, 2-9, 1992.

[34] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith and B. Randell. A Program Structure for Error Detection and Recovery. *Lecture Notes in Computer Science*, 16, 177-193, 1974.

[35] Y. Huang and Y. M. Wang. Why Optimistic Message Logging has not been Used in Telecommunications Systems. *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 459–463,

[36] D. R. Jefferson. Virtual Time. *ACM Trans. Prog. Lang. and Sys.*, 7(3), 404-425, 1985.

[37] P. Jalote. Fault Tolerant Processes. *Distributed Computing.* 1989.

[38] D. B. Johnson. Efficient Transparent Optimistic Rollback Recovery for Distributed Application Programs. *Proc. 12th IEEE Symp. on Reliable Distributed Systems*, 86-95, 1993.

[39] D. B. Johnson and W. Zwaenepoel. Sender-Based Message Logging. *Proc. 17th Intl. Symp. on Fault-Tolerant Computing*, 14-19, 1987.

[40] D. B. Johnson and W. Zwaenpeoel. Recovery in Distributed Systems using Optimistic Message Logging and Checkpointing. *Journal of Algorithms*, 11: 462-491, September 1990.

[41] M. O. Killijian, J. C. Fabre, J. C. R. Garcia, and S. Chiba. A Metaobject Protocol for Fault-Tolerant CORBA Applications. *Proc. 17th IEEE Symposium on Reliable Distributed Systems*, 127-134, 1998.

[42] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, vol. 21, no. 7, 558-565, 1978.

[43] B. Lewis and D. J. Berg. *Threads Primer: A Guide to Multithreaded Programming*. Sunsoft Press, 18 – 20, 1996.

[44] T. J. Leblanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Trans. on Computers*, C-36(4):471-481, Apr. 1987

[45] A. Lowry, J. R. Russell, and A. P. Goldberg. Optimistic Failure Recovery for Very Large Networks. *Proc. IEEE Symposium on Reliable Distributed Systems*, 66–75, 1991.

[46] B. W. Lampson and H. E. Sturgis. Crash Recovery in a Distributed Data Storage System. Unpublished Technical Report, Xerox Palo Alto Research Center. April 1979. *http://research.microsoft.com/users/blampson/Publications.html*

[47] B. D. Lubachevsky, A. Schwartz, and A. Weiss. Rollback Sometimes Works ... if Filtered. *Proc. Winter Simulation Conference*, 630-639, 1989.

[48] F. Mattern. Virtual Time and Global States of Distributed Systems. *Parallel and Distributed Algorithms: Proc. of the Intl. Workshop on Parallel and Distributed Algorithms, Elsevier Science Publishers B. V.(North Holland)*, 215-226, 1989.

[49] S. Maffeis. Piranha: A CORBA Tool For High Availability. *IEEE Computer*, 59-66, April 1997.

[50] J. M. Mellor-Crummey and T. J. Leblanc. A Software Instruction Counter. *Proc. of the 3rd Symp. on Architectural Support for Programming Languages and Operating Systems*, 78-86, Apr. 1989.

[51] D. M. Nicol and X. Liu. The Dark Side of Risk (What Your Mother Never Told You About Time Warp). *Proc. 11th Workshop on Parallel and Distributed Simulation (PADS)*, 188-195, 1997.

[52] S. L. Peterson and P. Kearns. Rollback Based on Vector Time. *Proc. 12th IEEE Symp. on Reliable Distributed Systems*, 68-77, 1993.

[53] M. L. Powell and D. L. Presotto. Publishing: A Reliable Broadcast Communication Mechanism. *Proc. 9th ACM Symp. on Operating Systems Principles*, 100-109, 1983.

[54] H. Rajaei, R. Ayani, and L. E. Thorelli. The Local Time Warp Approach to Parallel Simulation. *Proc. 7th Workshop on Parallel and Distributed Simulation (PADS)*, 119-126, 1993.

[55] S. Rao, L. Alvisi and H. Vin. The Cost of Recovery in Message Logging Protocols. Tech. Rep. No. TR-98-02, Dept. of Computer Sciences, Univ. of Texas at Austin, 1998.

[56] M. Russinovich and B. Cogswell. Replay for Concurrent Non-deterministic Shared-memory Applications. *Proc. ACM SIGPLAN Conf. on Programming Languages and Implementation (PLDI)*, 258-266, 1996.

[57] J. H. Slye and E. N. Elnozahy. Supporting Nondeterministic Execution in Fault-Tolerant Systems. *Proc. 26th Fault Tolerant Computing Symposium*, 250-259, 1996.

[58] S. W. Smith and D. B. Johnson. Minimizing Timestamp Size for Completely Asynchronous Optimistic Recovery with Minimal Rollbacks. *Proc. 15th Symp. on Reliable Distributed Systems*, 66-75, 1996.

[59] S. W. Smith, D. B. Johnson, and J. D. Tygar. Completely Asynchronous Optimistic Recovery with Minimal Rollbacks. *Proc. 25th Intl. Symp. on Fault-Tolerant Computing*, 361-370, 1995.

[60] A. P. Sistla and J. L. Welch. Efficient Distributed Recovery Using Message Logging. *Proc. 8th ACM Symp. on Principles of Distributed Computing*, 223-238, 1989.

[61] R. E. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Trans. on Computer Systems*, 204-226, August 1985.

[62] W. Toy. AT&T Case Part I: Fault-Tolerant Design of AT&T Telephone Switching System Processors. *Reliable Computer Systems: Designs and Evaluations*, Butterworth Publishing, Newton MA, 1992, 533-670.

[63] K. C. Tai, R. H. Carver, and E. E. Obaid. Debugging Concurrent Ada Programs by Deterministic Execution. *IEEE Trans. on Software Engineering*, 17(1):45-63, Jan. 1991.

[64] Y. M. Wang, O. P. Damani, and V. K. Garg. Distributed Recovery with K-Optimistic Logging. *Proc. 17th Intl. Conf. Dist. Comp. Sys.*, 60-67, 1997.

[65] Y. M. Wang, O P. Damani, and W. J. Lee. Reliability and Availability Issues In Distributed Component Object Model (DCOM). *Proceedings of the 4th International Workshop On Community Networking (CN4'97)*, 59-63, 1997.

[66] Y. M. Wang and W. K. Fuchs. Lazy Checkpoint Coordination for Bounding Rollback Propagation. *Proc. IEEE Symposium on Reliable Distributed Systems*, 78-85, 1993.

[67] Y. M. Wang, Y. Huang, W. K. Fuchs, C. Kintala, and G. Suri. Progressive Retry for Software Failure Recovery in Message-Passing Applications. *IEEE Trans. on Computers*, Vol. 46, No. 10, pp. 1137-1141, Oct. 1997.

Vita

Om Prakash Damani, the son of Shree Lal Damani and Raj Kumari Damani, was born in Calcutta, India on March 11, 1973. Since the age of 1, he stayed with his maternal grand-parents Dau Lal Bahety and Rama Devi Bahety. He passed his secondary examination from Tantia High School, Calcutta and higher secondary examination from St. Lawrence High School, Calcutta. From 1990 to 1994, he studied Computer Science and Engineering at Indian Institute of Technology Kanpur, India and received the Bachelor of Technology degree. In 1994, he joined the University of Texas at Austin for graduate studies. He did summer internships at Lucent Bell-Labs, New Jersey; AT&T Labs-Research, New Jersey; and Ubilab, Switzerland. In 1997, he married Seema.

Permanent Address: 39A Sir Hariram Goenka St., Calcutta 700007, India.

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin.