Completely Asynchronous Optimistic Recovery with Minimal Rollbacks

Sean W. Smith[*], *David B. Johnson, and J. D. Tygar*

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213-3891 USA

Abstract

Consider the problem of transparently recovering an asynchronous distributed computation when one or more processes fail. Basing rollback recovery on message logging *and replay is desirable since failure-free operation requires no synchronization between processes, and since logging a received message is cheaper than recording a checkpoint. Furthermore, surviving processes have the ability to recreate states other than those recorded in checkpoints—so only computation that depends on the failure must be rolled back. Although* optimistic rollback recovery *protocols make failure-free operation even cheaper by logging received messages* asynchronously, *optimism complicates recovery. Previous optimistic rollback recovery protocols have either required synchronization during recovery, or have permitted a failure at one process to potentially trigger an exponential number of process rollbacks. In this paper, we present an optimistic rollback recovery protocol that provides* completely asynchronous recovery, *while also reducing the number of times a process must roll back in response to a failure to at most one.*

1. Introduction

1.1. The Problem

Consider a long-running application program on an asynchronous distributed system. Suppose a process p fails, and recovers by rolling back to a previous state. Process p's computation since it first passed through the restored state has become lost. The failure and rollback of p may cause the state at a surviving process to become an *orphan*: a state that causally depends on lost computation. The existence of orphan states causes the system state to be *inconsistent*. The challenge of *rollback recovery* is to restore and maintain a consistent system state when one or more processes fail and roll back.

One of the desirable properties of a rollback recovery protocol is the ability to perform recovery *transparently* to the application program. Another desirable property is minimizing the amount of computation wasted due to rollback. For example, after a process failure, we might measure the number of surviving processes that must roll back, the number of times each process must roll back, and the amount of rolled-back computation beyond that which causally depends on the computation lost due the failure. An extreme case of such wasted computation is the *domino effect* [21, 22], in which all processes are forced to roll back to their initial states regardless of the amount of progress made or the amount of state individually saved for each process. A third desirable property of rollback recovery is minimizing the overhead incurred by the protocol, both during failure-free execution and during recovery. During failure-free execution, synchronization between processes slows down the application program; during recovery, synchronization between processes slows down recovery and may prevent recovery from concurrent failures from proceeding concurrently.

Optimally, surviving processes roll back at most once, and only roll back the portion of their own computation that has become an orphan. In this paper, we present a protocol that provides this property, while also providing complete asynchrony. Processes do not synchronize with each other during failure-free execution or during recovery, and processes record all recovery information during failure-free operation (logged messages and checkpoints) on stable storage asynchronously.

This research was sponsored in part by the Advanced Research Projects Agency under contract number F19628-93-C-0193, IBM, Motorola, the National Science Foundation under Presidential Young Investigator Grant CCR-8858087, TRW, and the U. S. Postal Service. The first author also received support from an ONR Graduate Fellowship.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

[*]Dr. Smith is currently affiliated with the Computer Research and Applications group at Los Alamos National Laboratory, Mail Stop M986, P.O. Box 1663, Los Alamos, NM 87545 USA.

The protocol is based on *optimistic message logging*. In the general *message logging* approach to recovery, processes log their received messages and occasionally checkpoint their local state. A process may recover to any past state by restarting from an earlier checkpoint and then replaying from the log the sequence of messages it originally received after that checkpoint. This approach to recovery assumes that the execution of each process is *piecewise deterministic* [26], in that its execution between successive received messages is completely determined by the process state before the first of these messages is received and by contents of that message. After receiving a message, a process performs a sequence of deterministic state transitions, some of which may involve sending messages to other processes. The process then attempts to receive another message, and blocks until one is available. This scheme can also be extended to handle some nondeterminism [8, 13] by treating each nondeterministic influence as a message, logging it and replaying it during recovery.

The message logging approach allows states of a process in addition to those saved in a checkpoint to be recovered. Recovery protocols based instead on checkpointing without message logging (e.g., [1, 4, 5, 6, 7, 14, 15, 16, 27]) can only recover the process states that have been checkpointed, often forcing processes to roll back further than otherwise required after a failure. Message logging thus allows each process to be checkpointed less frequently, and may in general reduce failure-free overhead since logging a message is less expensive than recording a checkpoint. Message logging also avoids the need for process synchronization during checkpointing; protocols using only checkpointing, on the other hand, either require some form of synchronization during checkpointing in order to record a consistent set of process checkpoints, or cannot guarantee to avoid the domino effect during recovery since no consistent set of checkpoints may exist.

Optimistic message logging protocols (e.g., [11, 12, 13, 19, 23, 26]) buffer received messages in volatile storage and log them to stable storage asynchronously in order to avoid blocking the process due to logging. Unlike *pessimistic* message logging protocols (e.g., [2, 3, 8, 10, 20]) which log each message synchronously, optimistic protocols will let a process receive a message and continue execution before the message is saved to stable storage. As a result, a failure at a process that has not yet logged some received messages may create orphans at other processes—since the failed process may not be able to restore its last state before failure, but other processes may depend on these lost states. In order to restore the system to a consistent state, an optimistic recovery protocol must be able to detect and eliminate orphans throughout the system. Although optimism thus complicates recovery, optimistic rollback protocols are cheaper during failure-free operation due to their asynchronous operation. By also using asynchronous checkpointing techniques [16, 7], all causes of process blocking due to fault tolerance during failure-free operation may be avoided.

1.2. Asynchronous Recovery

Existing optimistic rollback protocols have required synchronization between processes during recovery. This synchronization has been necessary in order to ensure that the system recovers to a consistent state, and that the system state remains consistent in spite of the effects of any orphan processes or messages remaining after the failure.

Strom and Yemini [26] initiated the area of optimistic rollback recovery and presented the most asynchronous protocol prior to the completely asynchronous protocol that we present in this paper. In the Strom and Yemini protocol, processes use timestamp vectors to track dependency. When a process rolls back, it begins a new *incarnation* and sends announcements to the other processes. (These announcement messages are not part of the failure-free computation, and thus do not carry dependency.) When a process receives a rollback announcement, it uses its timestamp vector to determine if its current state is an an orphan; if so, this process rolls back to its maximal state that it believes is not an orphan. The process restarts from an old checkpoint and replays from the log its received messages until it reaches one known to be an orphan based on the incarnation start information contained in the rollback announcements it has received.

The Strom and Yemini protocol usually requires no process synchronization during recovery, but may in some cases need to block a process. Processes need the incarnation start information from rollback announcements in order to compare timestamp vectors received on messages; if an announcement is delayed, any process that has not yet received the announcement may be forced to block when it needs to make a timestamp vector comparison. This behavior can occur even though the protocol assumes FIFO message ordering between each pair of processes, since a timestamp vector entry referring to the new incarnation may arrive at this process indirectly through a chain of messages.

In addition, the asynchrony present in the Strom and Yemini protocol can permit a single failure at one process to cause other processes to roll back an exponential number of times. This behavior occurs because an orphan state at a surviving process r may depend on the lost computation at another process through multiple paths: directly from the failed process, and indirectly through intermediate processes. The protocol may generate rollback announcements in such a way that process r rolls back in response to the rollbacks of intermediate processes, and then in response to the rollback of the failed process. Figure 1 shows a simple scenario in which process r rolls back twice in response to a single failure at process p. Sistla and Welch [23] claim an upper bound of $O(2^n)$ rollbacks in the worst case

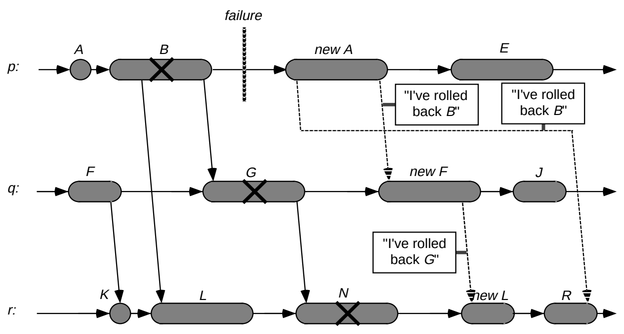

Figure 1 The Strom and Yemini protocol may cause surviving processes to roll back multiple times in response to a single failure. This diagram shows how one failure at process p causes process r to roll back twice. Process p fails and rolls back state interval B. (An "X" marks each rolled back interval.) This failure makes state interval G at process q an orphan, since G depends on the lost state interval B. The failure at p also makes process state interval N at process r an orphan, since it depends directly on B and indirectly on B through q. When process q receives p's announcement about B, q rolls back to its most recent state that does not depend on B. Unfortunately, q's announcement may arrive at process r before p's announcement does. When process r receives q's announcement about G, r rolls back to its most recent state that does not depend on G, and then proceeds with its computation. Process r does not know that its restored state and subsequent states are still orphans until after the delayed p announcement arrives.

	Strom and Yemini	Johnson and Zwaenepoel	Sistla and Welch	Peterson and Kearns
Assumptions	FIFO	None	FIFO	None
Asynchronous recovery?	Mostly	No	No	No
Maximum rollbacks at one process from one failure?	$\Theta(2^n)$	1	1	1
Entries in timestamps	$O(n)$	$O(1)$	$O(n)$	$O(n)$

Table 1 Our protocol improves on previous optimistic message logging protocols by providing completely asynchronous recovery, while also requiring a process to roll back at most once in response to any failure. The principal drawback of our protocol is timestamp size, since the protocol requires vector clocks for two levels of partial order time.

for the Strom and Yemini protocol, where n is the number of processes in the system. In [25], we provide a constructive proof for a lower bound of $2^n - 1$.

1.3. Our Results

In this paper, we present a new protocol for optimistic rollback recovery. Previous work in optimistic recovery has modeled the application program with partial order time, and used the standard technique of timestamp vectors [26, 9, 17] to track causal dependency. Our work exploits the insight that the transparent recovery protocol itself is also an asynchronous distributed computation. This recovery computation can also modeled by a partial order—but one that differs from the partial order for the user application computation. Our protocol maintains vector clocks for both levels; comparing vectors across time levels optimally characterizes when a given state is an orphan.

Our new protocol improves on previous optimistic rollback protocols by requiring no synchronization during recovery. In comparison specifically to the Strom and Yemini protocol (which otherwise requires the least synchronization during recovery), our protocol reduces the worst case number of rollbacks per process after any failure from exponential to *one*, and requires neither FIFO messages nor blocking of any process. Our protocol also does not require the sending of any messages other than those sent by the application program. Table 1 compares our protocol to four principal optimistic message logging protocols (discussed further in Section 6).

Section 2 of this paper presents our new recovery protocol. Section 3 presents its theoretical basis, and Section 4 uses this basis to establish the properties of our protocol. Section 5 sketches some data structures for implementation. Section 6 compares this protocol to previous rollback recovery protocols, and finally, Section 7 presents conclusions. We do not consider the issues of failure detection and reconfiguration after a failure in this paper, as these issues are largely orthogonal to the recovery protocol used in the system.

2. The Protocol

2.1. Definitions

Two Levels of Computation Besides performing the application program, processes also perform recovery. We formalize this duality by discussing two distributed computations:

- the *user* application computation, and
- the *system* recovery computation.

The system computation consists of the user computation along with extra management information. The system state at process p consists of the user state plus some extra state. All user messages are carried by system messages, but some messages may be exclusively system-only. The system process at p *implements* the user process (but the system process is transparent to the user process). Only user messages are logged.

State Intervals A *state interval* is a period of deterministic execution at a process. In our model, each process has a current *system state interval* a current *user state interval*. A process beings a new system state interval each time the system process receives a new system-level message, each time the user process receives a new user-level message, and each time the system process rolls back the user process. A process beings a new user state interval each time the user process receives a new user-level message; rollback restores an old user state interval.

Each state interval at a process has a *state interval index*. We use capital Roman letters to denote state interval indices, and use subscripts to indicate whether the interval is from the system level or from the user level. For example, A_S denotes a system state interval index, and B_U denotes a user state interval index.

At each process, the system state interval indices form a *timeline*: a linear sequence. The user state interval indices, on the other hand, form a *timetree*, since each rollback begins a new branch. Figure 2 illustrates this branching. We use $\preceq_S$ and $\preceq_U$ to indicate comparison on system state interval indices and user state interval indices, respectively.

Since (at each process) a new system state interval begins whenever a new user state interval begins, a given system state interval is associated with exactly one user state interval. We can use this association to compare system state interval indices to user state interval indices. We denote this comparison by $\preceq_*$. For any state interval indices A_U and B_S, we define $A_U \preceq_* B_S$ to hold if and only if $A_U \preceq_U B_U$, where B_U is the user state interval index associated with B_S.

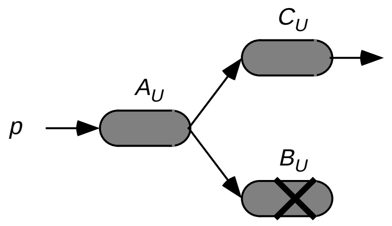

Figure 2 Rollback places the user state interval indices at a process into a timetree. Here, process p executes user state interval A_U, then B_U. Process p then rolls back to A_U, and subsequently executes C_U. We have $A_U \preceq_U B_U$ and $A_U \preceq_U C_U$. However, we also have both $B_U \not\preceq_U C_U$ and $C_U \not\preceq_U B_U$.

Vectors of Indices A *vector* is an array of state interval indices, one for each process. For a vector X, $X[p]$ denotes the process p entry of X. We use the $\preceq_S$, $\preceq_U$, and $\preceq_*$ comparisons on state interval indices to define entry-wise comparisons on vectors of state interval indices. We also use the $\preceq_S$ and $\preceq_U$ comparisons to define the entry-wise vector maximizations MAX_S and MAX_U, respectively. However, since user state interval indices form a tree, it is possible that two user state interval indices at a process may be incomparable. MAX_U is undefined in such situations.

2.2. The Protocol

This section presents our protocol in terms of system and user state indices, functions to compare them, and functions to generate new indices. Section 5 will present some approaches to implementing these functions.

Overhead Each process maintains timestamp vectors V_S and V_U for the system and user computations, respectively. At process p, the p entries of these vectors are its current state interval indices.

Sending Messages Figure 3 shows how messages are sent. To send a system-level message, a process p sends its own name, the message text, and the timestamp vector of the current system state interval. To send a user-level message, a process p sends a system-level message whose text contains two items: the user message text, and the timestamp vector of the current user state interval.

Receiving a System-level Message Figure 4 shows how processes receive system-level messages.[1] Process p first

```
/* process $p$ sends a system message to process $q$ */
procedure SEND_SYS $(M_S, q)$
    send $(p, M_S, V_S)$ to process $q$

/* process $p$ sends a user message to process $q$ */
procedure SEND_USR $(M_U, q)$
    SEND_SYS$((M_U, V_U), q)$
```

Figure 3 When sending a system message, a process includes the current system timestamp vector. When sending a user message, a process packages it with the current user timestamp vector, and sends this package as a system message.

[1]The assumption of piecewise determinism requires user-level processes to perform blocking receives, and for clarity of presentation, we have also assumed that system-level processes perform blocking receives. However,

```
/* process $p$ receives a system message */
function RECEIVE_SYS
    wait until $(q, M_S, X_S)$ arrives
    /* update system timestamp vector */
    $V_S \leftarrow MAX_S(V_S, X_S)$
    /* begin new system state interval */
    $V_S[p] \leftarrow NEW\_SYS\_INDEX(V_S[p], V_U[p])$
    asynchronously log $V_S$ to stable storage
    /* check if $p$ is an orphan */
    if $V_U \not\preceq_* V_S$
        then ROLL_BACK
    return $(q, M_S)$
```

Figure 4 When receiving a system-level message, a process begins a new system state interval, and rolls back if the new system timestamp vector indicates that the current user state is an orphan.

uses the timestamp vector on the message to update the system timestamp vector at p. Process p then begins a new system state interval and asynchronously logs the timestamp vector of the new interval. (Only the maximum system timestamp vector from each process need be retained on stable storage.) The updated system timestamp vector gives process p new information about the recovery activity of other processes. Process p compares its current user timestamp vector to its new system timestamp vector to determine if this new information indicates that the user state at p is now an orphan. (Section 3 explores this comparison in more detail.) If so, process p calls *ROLL_BACK* to roll itself back to its most recent non-orphan user state interval. In any case, *RECEIVE_SYS* at process p returns the source of the system message and the message text.

RECEIVE_SYS uses the *NEW_SYS_INDEX* function to generate new system state interval indices. *NEW_SYS_INDEX*(A_S, B_U) returns a system state interval index that immediate follows A_S, and whose unique user state interval is B_U.

Rollback Figure 5 shows the *ROLL_BACK* procedure, in which a process p restores its most recent available user state interval that is not an orphan. If p has failed and lost volatile storage, then the most recent user state intervals may not be available.

we could obtain increased performance by having the system-level process perform interrupt-driven receives; the system process would maintain a buffer of messages for the user process, and on each V_S update could re-examine the buffer for orphans.

```
/* process $p$ rolls back to most recent non-orphan */
procedure ROLL_BACK
    /* restore maximal non-orphan user state */
    find most recent checkpoint $(C, X_U)$ with $X_U \preceq_* V_S$
    discard the checkpoints that follow this one
    restore the user state to $C$
    find first logged message $(M_U, Y_U)$ following $X_U[p]$
    while $Y_U \preceq_* V_S$
        replay the receive of message $M_U$
        $V_U \leftarrow Y_U$
        get next $(M_U, Y_U)$
    discard remaining logged messages
    /* begin new system interval */
    $V_S[p] \leftarrow NEW\_INCARNATION(V_S[p], V_U[p])$
    write $V_S$ to stable storage before proceeding
    return
```

Figure 5 A process p rolls back to the most recent non-orphan user state interval by restoring the most recent non-orphan checkpoint, and replaying received messages whose sends are not orphans.

As with *RECEIVE_SYS*, process p determines orphans by comparing a user timestamp vector to the current system timestamp vector. Process p first obtains the most recent checkpointed user state interval that is not an orphan. Process p then replays the received messages logged after this checkpoint was recorded, until a message is reached whose send is an orphan. Process p discards any orphan checkpoints and logged messages, and begins a new system state interval. The discarding of orphan log data must be completed before any new data is logged.[2] *ROLL_BACK* also uses the *NEW_INCARNATION* function to generate new system state interval indices. The *NEW_INCARNATION*(A_S, B_U) function returns a system state index B_S whose user state is B_U. B_S follows A_S immediately—and also follows all other state intervals that have already occurred. (This is necessary since the process may have proceeded beyond A_S, but lost this information due to failure.)

Receiving a User-level Message Figure 6 shows how processes receive user-level messages. When receiving a user-level message, a process p waits for a system-level message that carries a user-level message whose send is not an orphan (according to p's current information). Process

[2]Old logged messages and checkpoints may also be discarded when no longer necessary for recovery from any possible future failure [26, 12].

```
/* process $p$ receives a user message */
function RECEIVE_USR

    /* loop until a suitable user message is available */
    DONE$\leftarrow$false
    while $\neg$DONE

        $(q, M_S)\leftarrow$RECEIVE_SYS

        /* if $M_U$ carries a user message, process it */
        if $M_S$ has format $(M_U, X_U)$
        then
            /* check if the send is an orphan */
            if $(X_U \preceq_* V_S)$
            then
                /* the send is not an orphan */
                DONE$\leftarrow$true        /* we can accept $M_U$ */
            else optionally inform process $q$

    /* formally receive user message $M_U$ */
    /* update user timetamp vector */
    $V_U\leftarrow$MAX$_U(X_U, V_U)$

    /* begin new user and system state intervals */
    $V_U[p]\leftarrow$NEW_USR_INDEX$(V_S[p], V_U[p])$
    $V_S[p]\leftarrow$NEW_SYS_INDEX$(V_S[p], V_U[p])$
    asynchronously log $(M_U, V_U)$ and $V_S$ to stable storage

    return $(q, M_U)$
```

Figure 6 To receive a user-level message, a process waits for a system-level message that contains a user-level message whose send was not an orphan according to the process's current information.

p then uses the user timestamp vector on the user message to update the user timestamp vector at p. Process p then begins a new user state interval and a new system state interval.

RECEIVE_USR uses the *NEW_USR_INDEX* function to generate new user state interval indices. The *NEW_USR_INDEX*(A_S, B_U) function returns a user state interval index that immediately follows B_U, in the context of the system state interval index A_S.

Recovering from Failure For process p to recover from its own failure, it simply reloads the current system timestamp vector from stable storage into V_S, and then calls *ROLL_BACK*.

An Example Figures 7 and 8 illustrate how our protocol avoids the multiple rollback problem of Figure 1. This example also illustrates a simple optimization to the protocol: a process rolling back may optionally explicitly announce the rollback, in order to reduce the latency before which any new orphan processes also roll back. If the announcement message has not yet been received by some process, the next message sent to that process by a process that has heard of the rollback (such as an announcement of its own rollback, or even a system-level message sending a user-level message to that process) will cause the process to learn of the rollback. For efficiency, this announcement could be sent as a single unreliable broadcast system-level message, if supported by the underlying system; if the broadcast message is not received by some process, the next message to that

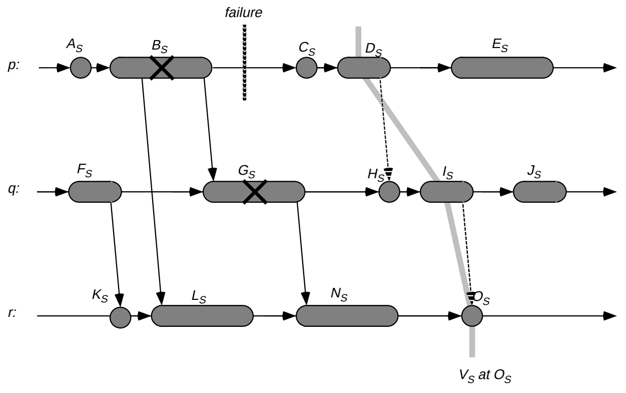

Figure 7 Our protocol avoids the multiple rollbacks that the Strom and Yemini protocol had in Figure 1. This diagram shows the first step. Here, process p fails, rolls itself back, and announces this fact to process q. (Dashed arrows indicate system-only messages). Process q rolls itself back and announces this fact to process r. When process r receives this announcement at state interval O_S, its system timestamp vector has D_S as its p entry and I_S as its q entry.

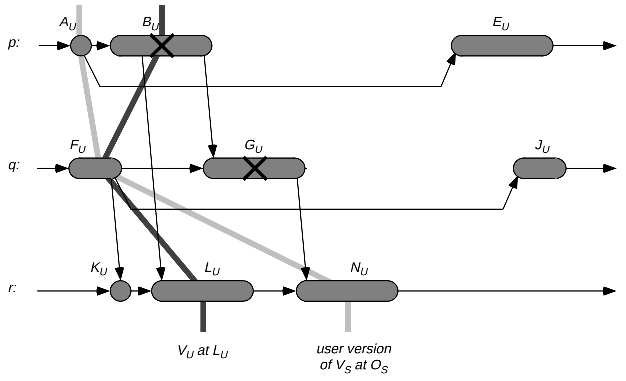

Figure 8 This diagram shows the second step of the example started in Figure 7. The user timestamp vector of user state interval L_U at r does not $\preceq_U$ -precede the user version of the system timestamp vector of system state interval O_S at r. (In particular, precedence fails for the process p entries.) Thus, when r receives the system-level message announcing q's rollback, r can determine that L_U is an orphan. Unlike the Strom and Yemini protocol, our protocol causes process r to roll back far enough the first time.

process will also serve to inform it of the rollback. These announcement messages, though, are only an optimization, and are not required for correct operation of the protocol.

3. Theoretical Basis

The protocol tests for orphans by comparing timestamp vectors across two levels of partial order time [24, 25]. Section 3.1 discusses the two levels of time. Section 3.2 shows how this comparison is an optimal test.

3.1. Preliminaries

State Intervals We have observed that each system state interval corresponds to exactly one user state interval, but each user state interval corresponds to at least one system state interval. Let *SYS_TO_USR* map each system state interval to its user state interval, and let *USR_TO_SYS* map each user state interval to its set of system state intervals.

Suppose user state interval A_U and system state interval B_S both occur at process p. We say that B_S *rolls back* A_U when B_S is the state interval that process p started while executing *ROLL_BACK*, and A_U was one of the user state intervals that was rolled back as part of that execution. Since process p can only roll back something that has already happened, any $A_S \in$ *USR_TO_SYS*(A_U) satisfies $A_S \preceq_S B_S$.

Two Partial Orders We build partial order time models for the two levels of computation in our system. We obtain these partial orders by constructing directed acyclic graphs whose nodes represent state interval indices and whose edges represent precedence.

For the system computation, we construct a node for each system state interval index. Let A_S and B_S be distinct state interval indices. We draw an edge from A_S to B_S when $A_S \preceq_S B_S$, or when A_S sends a message that initiates B_S. We say that A_S precedes B_S in the *system partial order* when a path exists from A_S to B_S in this graph. For the user computation, we construct a node for each user state interval index. Let A_U and B_U be distinct state interval indices. We draw an edge from A_U to B_U when $A_U \preceq_U B_U$, or when A_U sends a message that initiates B_U. We say that A_U precedes B_U in the *user partial order* when a path exists from A_U to B_U in this graph. We use $\longrightarrow$ to indicate precedence in these partial orders, and $\overset{=}{\longrightarrow}$ to indicate precedence or equality. Which partial order should be clear from the subscripts on the variable names.

The fact that rollback places the user state interval indices at a process into a tree permits some pathological situations. To restrict these situations, we say that a user state interval A_U is *valid* when its past in the user partial order touches only one root-leaf path at any process. That is, if state intervals B_U and C_U at process p satisfy $B_U \overset{=}{\longrightarrow} A_U$ and $C_U \overset{=}{\longrightarrow} A_U$, then $B_U \preceq_U C_U$ or $C_U \preceq_U B_U$.

Comparing the timestamp vectors for two state interval indices determines their partial order relation. Suppose system state intervals A_S and B_S have timestamp vectors X_S and Y_S, respectively. Then $A_S \overset{=}{\longrightarrow} B_S$ if and only if $X_S \preceq_S Y_S$. Similarly, suppose valid user state intervals A_U and B_U have timestamp vectors X_U and Y_U, respectively. Then $A_U \overset{=}{\longrightarrow} B_U$ if and only if $X_U \preceq_U Y_U$.

3.2. The Orphan Comparison

Potential Knowledge Suppose A_U is a user state interval at process p. When can another process potentially be aware that A_U is no longer part of the failure-free virtual computation?

We define a predicate *KNOWABLE_ORPHAN*(A_U, B_S) that is true when A_U is an orphan or has been rolled back, and process q during system state interval B_S can potentially be aware of this fact. For the predicate to be defined, process q must be aware of A_U; hence we require the precondition that $A_S \overset{=}{\longrightarrow} B_S$ for some $A_S \in$ *USR_TO_SYS*(A_U). For the predicate to be true, there must exist a user state interval C_U and a system state interval D_S satisfying the conditions:

- $C_U \overset{=}{\longrightarrow} A_U$,
- D_S rolls back C_U, and
- $D_S \overset{=}{\longrightarrow} B_S$.

The first two conditions are necessary for A_U to no longer be part of the valid computation. The last condition is necessary for process q to be potentially aware of this fact at B_S.

An Optimal Test Comparing timestamp vectors across the two levels of partial order time exactly captures this potential knowledge.

Theorem 1 Suppose user state interval A_U at process p and system state interval B_S at process q satisfy $A_S \longrightarrow B_S$, for some A_S in *USR_TO_SYS*(A_U). Let X_U be the user timestamp vector of A_U, and let Y_S be the system timestamp vector of B_S. Then

$$\textit{KNOWABLE\_ORPHAN}(A_U, B_S) \iff X_U \not\preceq_* X_S$$

Before proving this theorem, we establish three lemmas.

Lemma 1 Suppose B_S rolls back A_U. If $A_S \in$ *USR_TO_SYS*(A_U) then $A_S \preceq_S B_S$. If C_S satisfies $B_S \preceq_S C_S$, then $A_U \not\preceq_U$ *SYS_TO_USR*(C_S).

Proof Rolled-back states remain rolled-back, and we can only roll back states that have happened. $\square$

Lemma 2 If X_S is the timestamp vector for system state interval B_S, then a system state interval A_S at a process p satisfies $A_S \xrightarrow{\longrightarrow} B_S$ if and only if $A_S \preceq_S X_S[p]$. Similarly, if X_U is the timestamp vector for valid user state interval B_U, then a user state interval A_U at a process p satisfies $A_U \xrightarrow{\longrightarrow} B_U$ if and only if $A_U \preceq_U X_U[p]$.

Proof This follows inductively from the construction and maintenance of timestamp vectors. $\square$

Lemma 3 Suppose user state interval A_U and system state interval B_S satisfy

$$A_U \xrightarrow{\longrightarrow} \mathit{SYS\_TO\_USR}(B_S)$$

Let A_S be the minimal interval in *USR_TO_SYS*(A_U). Then $A_S \xrightarrow{\longrightarrow} B_S$.

Proof We establish this result by induction: If A_U and *SYS_TO_USR*(B_S) occur at the same process, this is easily true. If A_U sends a message that begins *SYS_TO_USR*(B_S), then some interval in *USR_TO_SYS*(A_U) precedes B_S so clearly A_S must. For more general precedence paths, choose an intermediate node C_U with $A_U \longrightarrow C_U \longrightarrow$ *SYS_TO_USR*(B_S), and choose the minimal C_S from *USR_TO_SYS*(C_U). Establish the result for A_U and *SYS_TO_USR*(C_S), and for C_U and *SYS_TO_USR*(B_S). $\square$

Proof (of Theorem 1) Suppose *KNOWABLE_ORPHAN*(A_U, B_S) holds. Then at some process r, there exists a user state interval C_U and system state interval D_S satisfying the statements: (1) $C_U \xrightarrow{\longrightarrow} A_U$, (2) D_S rolls back C_U, and (3) $D_S \xrightarrow{\longrightarrow} B_S$. Statement (1) implies that $C_U \preceq_U X_U[r]$. Statement (2) and Lemma 1 imply that $C_U \not\preceq_U$ *SYS_TO_USR*(E_S) for any E_S satisfying $D_S \preceq_S E_S$. Statement (3) implies that $D_S \preceq_S X_S[r]$. Hence $C_U \not\preceq_U$ *SYS_TO_USR*$(X_S[r])$. Thus $X_U \not\preceq_* X_S$.

Conversely, suppose $X_U \not\preceq_* X_S$. Then there exists a process r with $X_U[r] \not\preceq_U$ *SYS_TO_USR*$(X_S[r])$. Let $C_U = X_U[r]$; let C_S be the minimal state interval in *USR_TO_SYS*(C_U). By Lemma 2, $C_U \xrightarrow{\longrightarrow} A_U$. By Lemma 3 and hypothesis, $C_S \xrightarrow{\longrightarrow} B_S$. Thus by Lemma 2, $C_S \preceq_S X_S[r]$. Since by hypothesis $C_U \not\preceq_U$ *SYS_TO_USR*$(X_S[r])$ a D_S must exist such that $C_S \preceq_S D_S \preceq_S X_S[r]$ and D_S rolls back C_U. Lemma 2 gives $D_S \xrightarrow{\longrightarrow} B_S$, hence *KNOWABLE_ORPHAN*(A_U, B_S). $\square$

How quickly the system recovers from process failure depends on how quickly the processes whose user state intervals are orphans (or will become orphans) learn of the failure. Our protocol allows a range of alternatives, from broadcasting system-only messages, to letting the news percolate via the system timestamp data on user messages.

Theorem 1 establishes that the *MAX*$_U$ comparison in *RECEIVE_USR* is always well-defined. Suppose process p were to accept a user message with user timestamp vector X_U, and for some q, $X_U[q]$ and $V_U[q]$ were incomparable under $\preceq_U$. Then either $X_U[q]$ had been rolled back by the time $V_U[q]$ occurred (so process p would have discarded the message), or $V_U[q]$ had been rolled back by the time $X_U[q]$ had occurred (so process p would have rolled itself back).

4. Properties of the Protocol

Our new protocol is the first optimistic rollback protocol to implement completely asynchronous recovery effectively. We discuss the advantages.

Suppose a process p fails and rolls itself back. A surviving process q will roll back its own user state when this state fails the $\preceq_*$ comparison in *RECEIVE_SYS*. Hence:

- **Complete Asynchrony** When a process must roll back, it can roll back immediately and resume computation *without additional synchronization* with other processes.

Theorem 1 tells us that surviving process q will roll back its user state when this state becomes a knowable orphan: when it depends on a rolled back state, and a knowledge path exists from the rollback to q. Because of optimistic logging, a *surviving* process can always restore its maximal non-orphan state, so the orphans created by a process failure are exactly the state intervals that depend on the computation lost at the failed process. Because processes test user messages before receiving them, the state at process q never becomes an orphan due to the failure at p once a knowledge path is established. Hence:

- **Maximal Recovery** Like other optimistic rollback protocols, ours guarantees that a state is rolled back if and only if it causally depends on the computation lost at failed processes.

- **Minimal Rollbacks** Our protocol also guarantees that a failure at process p causes a process q to roll back at most once. Processes that do not depend on the failure will not roll back at all.

- **Speedy Recovery** Suppose process q must roll back because of a failure at process p. Process q will roll back as soon as any knowledge path is established from p's rollback.

- **Toleration of Network Partitions** Another side-effect of our asynchronous approach is that once initiated, recovery can proceed despite a partitioned network. The only processes that need to worry about recovery are those that may causally depend on lost states. Since each such process can recover asynchronously, the processes on the same side of the partition as the failure can recover immediately. Processes on the other side that need to recover can do so when the network is reunited. The remaining processes on either side may proceed unhindered. (This paper does not address the problem of *detecting* failure in a partitioned network.)

The preceding discussion considered the failure of a single process. Using the $\preceq_*$ test allows a surviving process to roll itself back to the maximal state that is not an orphan due to *any* rollback within the survivor's knowledge horizon. Hence our protocol provides:

- **Concurrent Recovery** Recovery from a process failure occurs as information about the failure propagates. Basing recovery on information flow rather than coordinated rounds directly allows recovery from concurrent failures to proceed concurrently: the recoveries merge and the protocol restores the maximum recoverable system state. (In particular, two processes that each need to roll back due to two failures do not need to react to the failures in the same order.)

5. Implementation Considerations

Our new protocol requires a way to represent user and system state indices, to compare these representations under the $\preceq_U$, $\preceq_S$ and $\preceq_*$ functions, and to generate new indices. This section provides a sample solution.

System Indices System state intervals are organized into a timeline. The system indices should reflect this linear structure—but also take into account the fact that a failed process may lose all state. The system index should also indicate the index of the corresponding user state interval. Following [26], we say that each rollback of a process begins a new incarnation of that process. We represent the index for system state interval A_S as a triple (k, m, A_U). Integer k represents the incarnation in which A_S occurs; integer m represents the sequence of A_S within that incarnation, and index A_U indicates the user interval corresponding to A_S.

This format supports our index generation functions. Let A_S be a system state interval with index (k, m, A_U). *NEW_SYS_INDEX*(A_S, B_U) returns an index $(k, m+1, B_U)$. *NEW_INCARNATION*(A_S, B_U) returns $(k', 0, B_U)$, where k' is one greater than the incarnation count at stable storage. This format also directly supports the $\preceq_S$ and $\preceq_*$ comparisons. For $\preceq_S$, we just lexicographically compare the first two entries in the indices for two intervals. For $\preceq_*$, we extract the the third entry of the system index, and use $\preceq_U$ (defined below).

User Indices User state intervals are organized into a timetree. In order to reflect this structure, we can use an index representation that indicates the position of a user state interval within the timetree. With every user interval, we associate two integers: the depth of the interval within the timetree, and the system incarnation in which this interval first started. (The incarnation changes only when rollback occurs.) We represent the index of a user state interval A_U as a triple (i, j, S). Integer i is the depth of A_U, integer j is its incarnation, and set S contains integer pairs indicating the depth and incarnation of all B_U satisfying two conditions:

1. $B_U \preceq_U A_U$
2. Either B_U has an incarnation different from its parent in the timetree, or B_U the root of the timetree.

By specifying each branch, this set determines the path from the root of the timetree to A_U.

This index format provides straightforward support for the $\preceq_U$ comparison. Suppose user intervals A_U and B_U have indices (i_A, j_A, S_A) and (i_B, j_B, S_B), respectively. To evaluate whether $A_U \preceq_U B_U$, we determine first if $i_A \leq i_B$, and then if $S_A \subseteq S_B$.

This index format also provides a straightforward way to generate new indices. Suppose system state interval A_S has incarnation and sequence numbers k and m, respectively; suppose user state interval B_U has index (i, j, S). If $k = j$, *NEW_USR_INDEX*(A_S, B_U) returns $(i+1, k, S)$; if $k \neq j$, *NEW_USR_INDEX*(A_S, B_U) returns $(i+1, k, S \cup \{(i+1, k)\})$.

Reducing the Size of User Indices The size of the set in the index of a user interval A_U is proportional to the number of rollbacks in the path from the root to A_U. If failures occur, this will not be constant; thus, for these indices, the size of user timestamp vectors will not be linear. Instead, the size will be proportional to the number of rollbacks in the system-past of the intervals recorded in the vector entries. (However, the number of vector entries will still be linear.)

We can reduce the amortized length of user indices by having processes avoid transmitting redundant data. One approach is to use a simple compression mechanism to abbreviate redundant byte sequences in the representation of timestamp vectors sent on a message. Another approach is to specify user index paths from intermediate nodes instead of from the root. Suppose process p wants to send the index of A_U to process q. Instead of sending the path from the root to A_U, process p can send the path from an intermediate interval B_U to A_U. If process q already knows the path from the root to B_U, then process q quickly reconstructs the full path. If not, process q recognizes that it is missing data and can request it of process p.

One example of this amortization technique is using a heuristic similar to Strom and Yemini's approach. Each time a process rolls back, it broadcasts the path to that rollback node along with its new incarnation count. Subsequent user indices consist solely of the system incarnation count and the position of the user interval within that incarnation. (This heuristic introduces blocking into our protocol, but still maintains the at-most-once lower bound on rollbacks at a process.) However, a wide range of other heuristics exists for this technique. At one extreme, process p transmits only the end of the path; at the other extreme, process p maintains the most recent system timestamp vector received from q, and uses the q entry as the intermediate node for a name sent to q. Commitment and garbage collection may integrate nicely with these amortization techniques, since processes may maintain a *log vector* of the maximal known logged nodes at other processes.

6. Comparison to Related Work

Strom and Yemini [26] initiated the area of optimistic rollback recovery. They presented optimistic techniques for surviving processes to ensure complete recoverability, and a rollback protocol[3] that allows processes to recover mostly asynchronously, although delayed transmission of incarnation start information may cause blocking. This protocol implicitly uses partial order time to track dependency on failed computation (and, to our knowledge, is the the earliest publication of the timestamp vector mechanism).

However, Strom and Yemini did not consider the flow of knowledge of rollback. They consequently built an orphan test that is strictly weaker than ours. Their protocol never falsely concludes that a non-orphan state is an orphan. However, their protocol will falsely conclude that some orphan states are not orphans—even when the testing process could potentially know otherwise. These false negatives make it possible for a single failure at one process to cause another process to roll back an exponential number of times, since the unfortunate process never rolls back far enough (until the last time).

Johnson and Zwaenepoel [11, 12] developed a general model for optimistic rollback recovery. They used state lattices from partial order time to show that a maximal recoverable system state exists, and presented synchronized protocols to recover this state—even without reliable message delivery. Sistla and Welch [23] presented two protocols for optimistic recovery that avoid the exponential worst case by using synchronization between processes during recovery; like Strom and Yemini, Sistla and Welch require reliable FIFO message channels. Peterson and Kearns [19] recently presented a recovery protocol using vector clocks that synchronizes during recovery by passing tokens. However, we improve even on the explicit vector time work of Peterson and Kearns by truly using the full power of temporal abstraction.

Recovery protocols based on checkpointing without message logging restore the system to a recovery line composed of local checkpoints. Organizing recovery lines into an increasing sequence (e.g., [4, 6]) may allow asynchronous recovery and may tolerate concurrent failures (since one recovery line will clearly be earliest). More complex structures of recovery lines require more synchronization upon recovery, but may allow some surviving processes to proceed without rolling back. However, unless for every state A_U, the maximal global state containing A_U is a recovery line, checkpointing-based recovery will force surviving processes to roll back computation that does not depend on the computation lost due to failure

[3]In some sense, Merlin and Randell [18] foreshadowed Strom and Yemini's work by presenting a protocol based on a representation similar to Petri Nets; this protocol could be transformed and optimized into one similar to Strom and Yemini's.

7. Conclusion

Optimistic rollback protocols improve on other recovery methods by requiring little synchronization during failure-free operation and by requiring only the theoretical minimum amount of computation to be rolled back (since the only computation that must be rolled back is the computation that depends on the computation lost due to failure). Our protocol improves on previous optimistic rollback protocols by providing both *completely asynchronous* recovery and a worst-case upper bound of *at most one rollback* at each process. The key to asynchronous optimistic rollback recovery is the realization that two levels of partial order time abstraction are relevant: causal dependency on rolled-back events and potential knowledge of rollbacks. Our protocol explicitly tracks these two levels of time.

References

[1] B. Bhargava and S. Lian. "Independent Checkpointing and Concurrent Rollback Recovery for Distributed Systems—An Optimistic Approach." *Seventh Symposium on Reliable Distributed Systems*. 3–12. IEEE, 1988.

[2] A. Borg, J. Baumbach and S. Glazer. "A Message System Supporting Fault Tolerance." *Ninth ACM Symposium on Operating Systems Principles*. 90–99. 1983.

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

[4] D. Briatico, A. Ciuffoletti, and L. Simoncini. "A Distributed Domino Effect Free Recovery Algorithm." *IEEE Symposium on Reliability in Distributed Software and Database Systems*. 207–215. October 1984.

[5] K. M. Chandy and L. Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems." *ACM Transactions on Computer Systems*. 3: 63–75. February 1985.

[6] A. Ciuffoelleti. "La Coordinazione Delle Attivita Di Ripristino Nei Sistemi Distribuiti." *A.I.C.A. Annual Conference Proceedings*. October 1989.

[7] E. N. Elnozahy, D. B. Johnson and W. Zwaenepoel. "The Performance of Consistent Checkpointing." *Eleventh IEEE Symposium on Reliable Distributed Systems*. 39–47. October 1992.

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

[9] C. J. Fidge. "Timestamps in Message-Passing Systems That Preserve the Partial Ordering." *Eleventh Australian Computer Science Conference*. 56–67. February 1988.

[10] D. B. Johnson and W. Zwaenepoel. "Sender-Based Message Logging." *Seventeenth Annual International Symposium on Fault-Tolerant Computing*. 14–19. 1987.

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

[12] D. B. Johnson and W. Zwaenepoel. "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing." *Journal of Algorithms*. 11: 462–491. September 1990.

[13] D. B. Johnson. "Efficient Transparent Optimistic Rollback Recovery for Distributed Application Programs." *Twelfth IEEE Symposium on Reliable Distributed Systems*. 86–95. October 1993.

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

[15] P. Leu and B. Bhargava. "Concurrent Robust Checkpointing and Recovery in Distributed Systems." *Fourth International Conference on Data Engineering*. 154–163. 1988.

[16] K. Li, J. F. Naughton and J. S. Plank. "Real-Time, Concurrent Checkpointing for Parallel Programs." *Second ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*. 79–88. 1990.

[17] F. Mattern. "Virtual Time and Global States of Distributed Systems." In Cosnard, et al, ed., *Parallel and Distributed Algorithms*. Amsterdam: North-Holland, 1989. 215–226.

[18] P. M. Merlin and B. Randell. "State Restoration in Distributed Systems." *International Symposium on Fault-Tolerant Computing*. June 1978.

[19] S. L. Peterson and P. Kearns. "Rollback Based on Vector Time." *Twelfth IEEE Symposium on Reliable Distributed Systems*. 68–77. October 1993.

[20] M. L. Powell and D. L. Presotto. "Publishing: A Reliable Broadcast Communication Mechanism." *Ninth ACM Symposium on Operating Systems Principles*. 100–109. 1983.

[21] B. Randell. "System Structure for Fault Tolerance." *IEEE Transactions on Software Engineering*. SE-1: 220–232, 1975.

[22] D. L. Russell. "State Restoration in Systems of Communicating Processes." *IEEE Transactions on Software Engineering*. 6 (2): 183–194. March 1980.

[23] A. P. Sistla and J. L. Welch. "Efficient Distributed Recovery Using Message Logging." *Eighth ACM Symposium on Principles of Distributed Computing*, 223–238. August 1989.

[24] S. W. Smith. *A Theory of Distributed Time.* Computer Science Technical Report CMU-CS-93-231, Carnegie Mellon University. December 1993.

[25] S. W. Smith. *Secure Distributed Time for Secure Distributed Protocols*. Ph.D. thesis. Computer Science Technical Report CMU-CS-94-177, Carnegie Mellon University. September 1994.

[26] R. Strom and S. Yemini. "Optimistic Recovery in Distributed Systems." *ACM Transactions on Computer Systems*. 3: 204–226. August 1985.

[27] Y.-M. Wang and W. K. Fuchs. "Lazy Checkpoint Coordination for Bounding Rollback Propagation." *Twelfth IEEE Symposium on Reliable Distributed Systems*. 78–85. October 1993.