Asynchronous Optimistic Rollback Recovery Using Secure Distributed Time

Sean W. Smith, David B. Johnson, J.D. Tygar

March 1994

CMU-CS-94-130

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597, and by the Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing," ARPA Order No. 7330, issued by ARPA/CMO under Contract MDA972-90-C-0035. Additional support was provided by NSF Grant CCR-8858087, by matching funds from Motorola and TRW, and by the U.S. Postal Service. The authors are grateful to IBM for equipment to support this research. The first author also received support from an ONR Graduate Fellowship.

Keywords: Distributed systems, concurrency, security and protection, checkpoint/restart, fault tolerance

Abstract

In an asynchronous distributed computation, processes may fail and restart from saved state. A protocol for *optimistic rollback recovery* must recover the system when other processes may depend on *lost* states at failed processes. Previous work has used forms of partial order clocks to track potential causality. Our research addresses two crucial shortcomings: the rollback problem also involves tracking a second level of partial order time (potential knowledge of failures and rollbacks), and protocols based on partial order clocks are open to inherent security and privacy risks. We have developed a *distributed time* framework that provides the tools for multiple levels of time abstraction, and for identifying and solving the corresponding security and privacy risks. This paper applies our framework to the rollback problem. We derive a new optimistic rollback recovery protocol that provides *completely asynchronous* recovery (thus directly supporting concurrent recovery and tolerating network partitions) and that enables processes to take full advantage of their maximum potential knowledge of orphans (thus reducing the worst case bound on asynchronous recovery after a single failure from exponential to at most one rollback per process). By explicitly tracking and utilizing both levels of partial order time, our protocol substantially improves on previous work in optimistic recovery. Our work also provides a foundation for incorporating security and privacy in optimistic rollback recovery.

1. Introduction

Optimistic rollback recovery allows distributed application programs to recover from the failure of one or more processes. Optimistic rollback recovery protocols have low failure-free overhead, but previously either required synchronization during recovery, or permitted (in the worst case) exponential rollbacks in order to recover from a single failure. We have developed a *distributed time* framework that provide tools for tracking multiple levels of temporal relations in a distributed computation. In this paper, we use this framework to build a simple protocol for optimistic rollback recovery that allows *completely asynchronous* recovery, with at most one rollback per process to recover from any failure. Furthermore, protocols, such as those for optimistic rollback recovery, that track partial order time are subject to inherent security and privacy risks. Our distributed time framework in which we developed the protocol provides a basis to systematically identify and protect against these risks.

The Recovery Problem Consider a distributed system consisting of processes that pass messages asynchronously. (To allow for full generality, we will assume nothing about the reliability of the network or the order of message delivery.) Suppose process p *fails* and recovers by restarting itself at an earlier, saved state. All activity by process p since it first passed through this restored state has been lost.

If the execution of the lost activity affected no process other than p, then the loss of this activity can affect no process except p. Suppose the lost activity had been entirely internal to p, or had included only the receipt of messages (if messages are not acknowledged and could be lost). Process p's failure and recovery will not hinder the overall computation.

However, suppose the lost activity at process p included the send of a message that was received by process q. Then the state of process q depends on activity at process p that has been rolled back. Process q has received a message that, in process p's view after recovery, was never sent. Distribution and asynchrony may make the situation even more complex. For example, if process q subsequently sends a message to process r, then process r also depends on events that never happened. Further, the lost activity at process p may include the send of a message to process r that, due to network delays, does not arrive until after p has rolled back and the system appears to have recovered.

The challenge of *rollback recovery* consists of correctly recovering the system when a failed process restarts an earlier state. *Pessimistic* rollback protocols (e.g., [BBG83, BBGH89, ElZw92, PoPr83]) prevent processes from acquiring dependencies on states that may become lost if a process fails. However, pessimism take a significant toll on performance [Jo89]. *Optimistic rollback protocols* (e.g., [BhLi88, Jo89, JoZw90, Jo93, KoTo87, PeKe93, SiWe89, StYe85]) optimistically bet that processes will not lose state, but then must consider the challenge of recovering the system when non-faulty processes may depend on lost states at the failed process.

The Security Problem Optimistic recovery requires determining which states depend on lost states. Existing protocols use *partial order time* [La78, Fi91, Ma89] to track this dependency, frequently employing variations on *vector clocks* [Fi88, Ma89]. However, tracking temporal relations different from real physical time creates security and privacy risks—whether or not a protocol is explicit about these relations [SmTy91, ReGo93, SmTy93]. A malicous process can disrupt the vector clock protocol by transmitting nonsense entries, or by more subtly altering its vector entries to fool honest processes into falsely believing temporal precedence (or concurrence) occurs when it does not. Even without active sabotage, a malicious process can exploit the private information shared in each timestamp vector to gain knowledge of other process's activities.

Attacks on partial order clocks translate to attacks on protocols built on these clocks. For example, during optimistic rollback recovery, a malicious process can cause honest processes to make incorrect decisions about whether they need to roll back.

Our Solutions Many problems in distributed systems depend on temporal relations more general than the linear order of real time, but tracking these relations creates security and privacy risks. We have addressed these issues by developing a *distributed time* framework [Sm93,Sm94] that provides tools to reason about multiple levels of time relations, to design protocols in terms of these relations, and to independently consider the inherent security and privacy risks.

In this paper, we use this framework to build a new optimistic rollback recovery protocol. The heart of the protocol is a simple procedure for processes to determine exactly when a given state depends on a lost state. The design and the correctness of this procedure follow directly from explicitly tracking both the partial order of causal dependency *and* the partial order of rollback knowledge. The completeness of this procedure (it reports no false negatives) allows *completely asynchronous* recovery while also ensuring each process rolls back at most once to recover from any failure. Our protocol thus substantially improves on previous optimistic rollback recovery protocols. Further, our distributed time framework provides a systematic way to add a level of security and privacy to any protocol that explicitly uses partial order time. This framework grants our recovery protocol an extra level of security since it is explicit about the multiple levels of partial order time involved.

This Paper Section 2 discusses asynchrony in optimistic rollback recovery. Section 3 presents the preliminaries of rollback and the use of our distributed time framework. Section 4 discusses the central role that *orphans* have in rollback protocols, and uses the distributed time tools to develop an optimal test for orphans. Section 5 uses this test to build the our new protocol. Section 6 discusses some security issues inherent in any protocol tracking nonlinear time, and Section 7 presents our conclusions.

2. Asynchronous Recovery

The more decentralized a distributed protocol is, the better it exploits the advantages of distribution (e.g., concurrency) and the more robust it is against the disadvantages (e.g., asynchrony, unreliable networks). Thus, in theory, the more decentralization a protocol brings to the task of optimistic rollback recovery, the better performance it achieves. In practice, getting asynchronous recovery to perform well is difficult.

Previous Approaches Strom and Yemini [StYe85] opened up the area of optimistic rollback recovery and presented a protocol that allows processes to recover mostly asynchronously. (Certain situations can require recovering processes to block.) However, their protocol assumes FIFO channels and deterministic processes. Their protocol also suffers from the drawback that, in the worst case, a single failure at one process can cause $\Theta(2^n)$ rollbacks at another process. (Sistla and Welch [SiWe89] cite $O(2^n)$; we have a simple construction showing $\Omega(2^n)$.)

To avoid these problems, subsequent work moved away from completely asynchronous recovery. Koo and Toueg [KoTo87] introduced a protocol based on two-phased commit. Bhargava and Lian [BhLi88] presented a synchronized recovery protocol that introduces some concurrency into recovery (and thus tolerates concurrent failures). Leu and Bhargava [LeBh88] dispensed with FIFO message orderong, and presented a synchronized protocol that introduces some concurrency into recovery and allows some toleration of network partition. Johnson and Zwaenepoel [Jo89, JoZw90] used state lattices from partial order time to show that a maximal recoverable system state exists, and present synchronized protocols to recover this state. Peterson and Kearns [PeKe93] recently presented a recovery protocol that uses vector clocks and synchronizes by passing tokens.

Asynchronous Recovery using Distributed Time The technique of using *partial order time* to track potential causality in a distributed system is well known. Rollback recovery requires determining which states have been potentially influenced by a lost state. Consequently, existing protocols use some form of partial order time (either implicitly or explicitly) to track this potential dependence. However, by dispensing with formal coordination, *asynchronous* rollback recovery requires being able to reason about and track potential knowledge of failures and restarts. This activity itself is an asynchronous distributed computation, and thus also trackable using partial order time.

However, this partial order differs from the partial order of events visible within the user's computation. For rollback recovery, *potential knowledge is not the same as causal dependency.* For example, suppose process q learns that its current state A depends on a lost state. Process q rolls back, and then enters state B. A knowledge path exists from state A to state B, but no causal dependency path exists.

Thus, to effectively implement asynchronous recovery, we need not only to move from viewing time as a linear order to viewing it as a partial order, but also to move away from viewing time as a single level of abstraction. Our framework of *distributed time* provides these tools, and allows us to build a new protocol that cleanly and elegantly solves the asynchronous recovery problem. Distributed time enables us to define when a state can be known to depend on a lost state, and to implement a test within the protocol that fully utilizes this potential knowledge.

Our new recovery protocol improves on previous work in optimistic rollback recovery in that it is the first protocol to effectively implement completely asynchronous recovery. It also compares favorably in many other aspects. We discuss some of the advantages:

Complete Asynchrony A failed process can restart immediately. When a process needs to roll back, it can roll back immediately and resume computation with *no additional synchronization*.

Minimal Rollbacks A failure at process p will cause process q to roll back at most once—and only when process q causally depends on rolled-back state at process p.

Speedy Recovery Suppose process q needs to 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 restart.

Concurrent Recovery Recovery from a failure occurs as knowledge of the failure propagates. Basing recovery on knowledge rather than coordinated rounds directly allows recovery from concurrent failures to procede concurrently. (In particular, two processes that each need to roll back due to two failures do not need to react to the failues in the same order.)

Toleration of Network Partitions Another side-effect of our asynchronous approach is that 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 recovery asynchronously, the processes on the same side of the network 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 can proceed unhindered.

A Framework for Security and Privacy Tracking partial order time relations creates security and privacy risks, since processes must share and trust private information. By building our protocol in terms of distributed time, we can transparently protect the protocol against these risks.

Our new protocol does require more timetamp information to be maintained, since two partial orders must be tracked simultaneously—but this bound is linear and thus is comparable to previous protocols that explicitly use vector time.

Orthogonal Issues A number of issues arise in our research that do not lie within the scope of this paper. We will not consider mechanisms for processes to detect failure, and for failed processes to migrate to non-faulty sites. We also make the simplifying assumption that processes restore state by retrieving a *checkpoint* from stable storage (although balancing checkpointing with message logging may provide better performance). We also do not worry here about committing output to the outside world. (Integrating checkpointing/logging techniques with our approach is an avenue for future work.)

3. Rollback and Distributed Time

3.1. Preliminaries

Partial order time provides a natural description of an asynchronous distributed computation. We consider a *state interval* as the fundamental unit of experience at a process. For a given computation, we build the partial order on state intervals in three steps. First, we can organize the state intervals at any one process into a linear sequence. We then link these timelines according to message traffic: if process p during p's state interval A sends a message which process q receives during q's state interval B, we let A precede B. Finally, we obtain the partial order by taking the transitive closure of this relation.

In a failure-free computation, this partial order expresses potential causal dependency. However, failure disrupts this dependency. When a process fails and restarts, it restores an earlier state and *rolls back* the states that previously had followed the restored state. Thus, the ability to fail and restart partitions the states at a process into sets: those that have been *rolled back* and those that are *live.*

Suppose states A, B, C, and D occur in that order at process p. Process p fails, restores A, and then begins a new state E. State A is the logical predecessor of state E; the *live history* of E includes A, but none of B, C, D.

To formalize this convention, Strom and Yemini partition the live history at a process into *incarnations*, the intervals from each restart to the subsequent failure. Incarnations are numbered sequentially at a process; restart begins a new incarnation. Suppose the rollback in the above example was the first one at process p. The interval up to and including state D constitutes the first incarnation at process p; the interval from the restarted A to E (and on the next failure) constitutes the second incarnation.

A state is *invalid* when it causally depends on two states at a process that could not both have been part of the live history at that process. In the above example, if state F at process q depends on state E at process p, then depending on state C at process p as well would make state F invalid. We assume that processes enforce the invariant that they never let their own state become invalid. (Later we will return to this topic.)

3.2. Using the Distributed Time Framework

3.2.1. Motivation

Distributed systems need distributed time. The linear order of real time is not sufficient (and is generally not observable). Frequently a more general relation also does not suffice, since multiple levels of abstraction require multiple levels of temporal relations. The framework of *distributed time* [Sm93] provides these tools. Distributed time represents computations as *computation graphs*, represents the full physical description of system traces as *ground-level computation graphs*, and uses *time models* to systematically transform graphs to the appropriate level of abstraction. Vector time is a special case of distributed time; the crucial difference for rollback recovery is that the modularity of distributed time easily supports *multiple levels of time*, which is the key to completely asynchronous recovery.

One Level is Insufficient Partial order time tracks causal dependence for failure-free computations. For any state A, we can tell whether state A potentally influenced some state B by examining their relation in the partial order. However, when failures and optimistic rollback recovery actually occur, this correspondence fails:

- This model places all states at a process in a linear sequence, which does not express the details of incarnations or live histories.
- This model regards all messages as carrying causal dependence. Messages exchanged as part of the recovery protocol (which should not carry dependence) are lumped in with the dependence-carrying messages of the application program.

A further complication is that in order to implement a recovery protocol, processes may need to perform recovery-managing computation that should not be considered part of the computation being recovered. Directly applying partial order time loses this distinction.

What is needed is an abstraction from the single level of partial order time.

3.2.2. Two Levels of Time

Two Levels of Abstraction Using a partial order to describe a computation implies abstraction: we neither know nor care which total order schedule actually occurred. Providing virtual failure-free computations through rollback recovery allows this abstraction to proceed indirectly—at the highest level, we neither know nor care whether the virtual failure-free computation was in fact failure-free. The problem of rollback recovery introduces two relevant levels of abstraction:

- The *user* level is the computation that is failing and being recovered. (Failures and recoveries are not visible to the user.)

- The *system* level is the fault-tolerant implementation of this computation using rollback recovery.

Two Levels of Computation Performing recovery may require computation not part of the user computation. This distinction gives processes a bipartite form: the *system process* (all state at the process) implements the *user process* (the subset of that state visible by the user computation).

This distinction introduces two corresponding types of state at processes, *user state* and *system state*. We will indicate user states with letters from the beginning of the alphabet, and system states with letters from the latter part of the alphabet (from Q on). We will use the letters G and H when the level does not matter.

The subset relation on states establishes a correspondence. Each system state Q has a well-defined "current" user state, which we indicate Q_{U}. Each user state A corresponds to at least one system state. We use A_{S} to indicate any such system state.

All messages exchanged in the system are, by definition, *system messages.* A message M sent by the user level of a process has two characterizations. If the user level of the destination process actually receives M, then M is also a *user message*. (That is, user messages are carried by system messages.) If the system level of the destination process receives M but does not forward it to the user level, then M is a system message only.

Two Levels of Time We build two levels of partial order time to reflect the two levels of computation. We construct *system partial order time* (*SYSTEM_TIME*) by taking the linear sequence of system states at a process, linking them with system messages, and taking the transitive closure.

Constructing *user partial order time* (*USER_TIME*) is similar, except the local structure generated at a process is a *tree* instead of a linear sequence. To build the user tree for a process, we put successive states in successive order—until *rollback*.

Rollback restores a previous user state. Suppose the process is in system state Q before rollback. After rollback, the process enters a special *restart* system state R, whose user state R_{U} does not follow Q_{U}, but instead is a copy of an earlier user state A that user-precedes Q_{U}. The next user state becomes a new child of A. Rollback thus terminates the current branch in the user tree, and grows a new branch from an earlier state. In the user tree at a process, every branch but one represents a failed and rolled-back part of the computation. (The remaining branch represents the computation currently live.) The leaves of a process tree are the states that have no logical successors (either maximal lost states, or the last state executed). The path from the root to a state A is the local *live* history of A.

In the *USER_TIME* model of the example of Section 3.1, states A through D are placed in sequential order, but after the failure/restart, state E is placed as another successor to state A, and subsequent computation extends from E.

Thus, using a tree as a "timeline" differs considerably from a traditional linear timeline. In a linear timeline, two states $A \neq B$ at the same process have two possible relations: either A precedes B or B precedes A. However, the user tree at a process admits a third relation: concurrency. If states A and B lie in different branches of the user tree at a process, then neither could have preceded the other. When one occurred, the other never occurred and never was going to occur.

To construct the full *USER_TIME* order, we link the user trees for each process by letting each *receive* of a *user message* follow its *send*, and then taking the transitive closure.

We indicate precedence between states by arrows ($G \longrightarrow H$) and precedence or equivalence by underscored arrows ($G \underline{\longrightarrow} H$).

The definitions immediately provide that system precedence follows from user precedence:

Theorem 1 Let A and B be user states in a computation. For any state B_{S} there exists a state A_{S} such that: $A \longrightarrow B$ in *USER_TIME* $\Longrightarrow$ $A_{\mathrm{S}} \longrightarrow B_{\mathrm{S}}$ in *SYSTEM_TIME*.

Vector Clocks The *vector clock* mechanism tracks relations in a standard partial order (such as *SYSTEM_TIME*) by equipping each state G with a *timestamp vector* $\mathbf{V}(G)$ that has one entry per process, with the property that the process p entry in $\mathbf{V}(G)$ is the maximal state H at process p with $H \underline{\longrightarrow} G$. Processes track these vectors incrementally: if state G_2 directly follows state G_1 at process q, process q obtains $\mathbf{V}(G_2)$ from $\mathbf{V}(G_1)$ as follows. Process q advances its own entry in $\mathbf{V}(G_1)$ to obtain interim vector W. If G_2 is a not a *receive*, then process q sets $\mathbf{V}(G_2)$ to W. If G_2 is a *send*, process q sends $\mathbf{V}(G_2)$ along with the message. If G_2 is a *receive*, process q strips off the timestamp vector X from the message, and sets $V(G_2)$ to be the entry-wise maximum of W and X.

Such vectors form a partial order based on the ordering of events at a process. Vector V precedes vector W when for each process p, the p entry of V precedes or equals the p entry of W in the execution order at p, but for some process, this inequality is strict. We write $V \prec W$ to indicate vector precedence.

Vector Clocks for System Time Since *SYSTEM_TIME* is a standard partial order model, we can use vector clocks to track system precedence. We write $\mathbf{V}_{\mathrm{sys}}(Q)$ to indicate the *SYSTEM_TIME* timestamp vector of a system state Q, $\prec_{\mathrm{sys}}$ to indicate *SYSTEM_TIME* vector precedence, and $\max_{\mathrm{sys}}$ to indicate entry-wise *SYSTEM_TIME* vector maximization.

Vector Clocks for User Time Using vector clocks to track *USER_TIME* is complicated by the fact that processes order local states into trees rather than linear sequences. However, the set of user states at a given process that user-precede a given user state are totally ordered within the tree at that process.

Theorem 2 Let B be a valid user state, and let p be a process (but not necessarily the one where B occurred). Let S be the set of all user states A at p such that $A \Longrightarrow B$ in *USER_TIME*. If S is nonempty, then S has a unique *USER_TIME* maximum.

As a consequence, the notion of timestamp vector is well-defined for valid user states. Thus we can use vector clocks to track *USER_TIME* as well. We write $\mathbf{V}_{\text{usr}}(A)$ to indicate the *USER_TIME* timestamp vector of a user state A, $\prec_{\text{usr}}$ to indicate *USER_TIME* vector precedence, and $\max_{\text{usr}}$ to indicate entry-wise *USER_TIME* vector maximization. (However, since the local ordering (used in vector precedence and maximization) derives from the user trees, mere integers will not suffice for vector entries. Section 5.2 considers these issues further.)

Validity The system level of a process can insure its user state never becomes invalid, simply by never accepting an incoming user message whose user timestamp has an entry that is not user-comparable with the corresponding entry of the process's current user timestamp. (We shall show shortly that there is a simpler way of obtaining this assurance.)

4. Orphans

An *orphan* is a state in a computation that causally depends on a state that has been lost. In terms of our time models, an orphan is a user state A such that some rolled-back user state B exists with $B \Longrightarrow A$ in *USER_TIME*.

This section discusses the central roll orphans play in optimistic rollback recovery in general, and asynchronous approaches in particular. This section then uses distributed time to characterize when a process can potentially know that a state is an orphan, and then to build a simple test that achieves this potential.

4.1. Orphans and Optimistic Recovery

A process p that initiates a recovery (that is, the process that actually fails) recovers by restoring earlier state and continuing user-level execution. This action causes one or more *live* states at process p to become *rolled-back*. The new rolled-back events are *orphans* by definition. However, the rollback action at p may also cause some states at other processes to become *orphans*.

The key to optimistic rollback recovery is the ability for processes to know when states have become orphans. This has two aspects:

Orphan Elimination When process q receives notification that process p has failed, process q needs to determine if its current user state has become an orphan. If so, process q needs to roll back—preferably back to the most recent state that is now *not* an orphan. Processes thus need to be able to test if *their own user states* are orphans.

Orphan Prevention The lost state at process p may have caused user state A at some process r to become an orphan. However, suppose user state A was the *send* of a message to process q. If process q user-accepts the message (whenever it arrives), then process q will become an orphan. Thus, to prevent becoming orphans, processes need to be able to test if *user states at other processes* are orphans.

Accurately testing for orphans is especially critical for asynchronous recovery, with multiple failures and minimal coordination.

4.2. Knowledge of Orphans

When can a process at system state Q know that a user state A is an orphan? We use distributed time to answer this question.

First, to even ask this question, state Q must know about state A. We must have the precondition that $A_{\mathrm{S}} \Longrightarrow Q$ in *SYSTEM_TIME*, for some A_{S}.

For A to be an orphan, a rolled-back state B must exist with $B \Longrightarrow A$ in *USER_TIME*. From Theorem 1 and transitivity, $B_{\mathrm{S}} \Longrightarrow Q$ in *SYSTEM_TIME* for some B_{S}. Thus, state Q can know about B.

However, for Q to know that A is an orphan, it must know that state B has been rolled back and is no longer part of the local live-history history at state B's process. If R is the restart system state following B's rollback, then we must have $R \Longrightarrow Q$ in *SYSTEM_TIME* as well.

We summarize this formally with the predicate *ORPHAN*(A, Q), which is defined only when $A_{\mathrm{S}} \Longrightarrow Q$ in *SYSTEM_TIME* for some A_{S}.

$$\mathit{ORPHAN}(A, Q) = \mathit{true} \iff \exists\, B, R \text{ such that } \begin{cases} 1.\ B \Longrightarrow A \text{ in } \mathit{USER\_TIME} \\ 2.\ R \Longrightarrow Q \text{ in } \mathit{SYSTEM\_TIME} \\ 3.\ R \text{ is a restart state rolling back } B \end{cases}$$

The *ORPHAN* predicate does *not* capture all the orphans in the computation—just all the orphans that a given process can know are orphans. If process p sends process q a user message but promptly rolls back without telling anyone, then q can not know that the *send* is an orphan. In *SYSTEM_TIME*, the timestamp vector on a state Q marks the information horizon of that state. State Q can not know about anything beyond this horizon—indeed, runs of the system could exist where every process pauses indefinitely after executing their $\mathbf{V}_{\mathrm{sys}}(Q)$ entry.

4.3. Testing for Orphans

We can use distributed time to build a test that exactly captures the *ORPHAN* predicate.

Let Q be a system state, and let A be a user state with $A_{\mathrm{S}} \Longrightarrow Q$ in *SYSTEM_TIME*, for some A_{S}.

Let R be the maximal system state at process p that state Q knows about. Then R_{U} must be live (to Q's knowledge), because a restart state rolling back R_{U} would system-follow R—contradicting the choice of R.

Indeed, to Q's knowledge, only those user states C at process p with $C \Longrightarrow R_{\mathrm{U}}$ in *USER_TIME* can be part of the live history at process p.

So, let A be a user state that Q knows about. Let C be the user-maximal user state at process p with $C \Longrightarrow A$ in *USER_TIME*. If $C \Longrightarrow R_{\mathrm{U}}$ in *USER_TIME*, then no restart at p that Q knows about makes A an orphan. If this relation holds for all processes, than Q must conclude that A has not been rolled back. Otherwise Q knows that A is an orphan.

Vector clocks permit an elegant statement of this test. For a vector W of system states, let W_{U} be the vector of user states obtained by taking the user part of each entry. Define the *DT_ORPHAN* test by:

$$\mathit{DT\_ORPHAN}(A,Q) = \mathit{true} \iff \mathbf{V}_{\mathrm{usr}}(A) \not\preceq_{\mathrm{usr}} \mathbf{V}_{\mathrm{sys}}(Q)_{\mathrm{U}}$$

That is, take the user timestamp of A, the system timestamp of Q, and do a *USER_TIME* vector comparison.

This test captures all potential knowledge of orphans.

Theorem 3 Suppose user state A and system state Q satisfy $A_{\mathrm{S}} \Longrightarrow Q$ in *SYSTEM_TIME*, for some A_{S}. Then $\mathit{ORPHAN}(A,Q) \iff \mathit{DT\_ORPHAN}(A,Q)$.

By not transitively propagating knowledge of orphans, Strom and Yemini use a strictly weaker orphan test. 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 $\Omega(2^n)$ times, since the unfortunate process never rolls back far enough (until the last time).

Examples Figure 1 demonstrates using the *DT_ORPHAN* test for orphan elimination and orphan prevention. The figure shows the *USER_TIME* partial order for a computation on three processes. Dashed arrows indicate *SYSTEM_TIME* precedence on corresponding system states, and Q is the sole system state for B_5 and R is the sole system state for C_6. Process p fails after A_4, rolls back to A_1, then executes A_5, and finally sends a system message to process r. Process r then rolls back to C_1, executes C_4 and C_5, and sends a system message to process q.

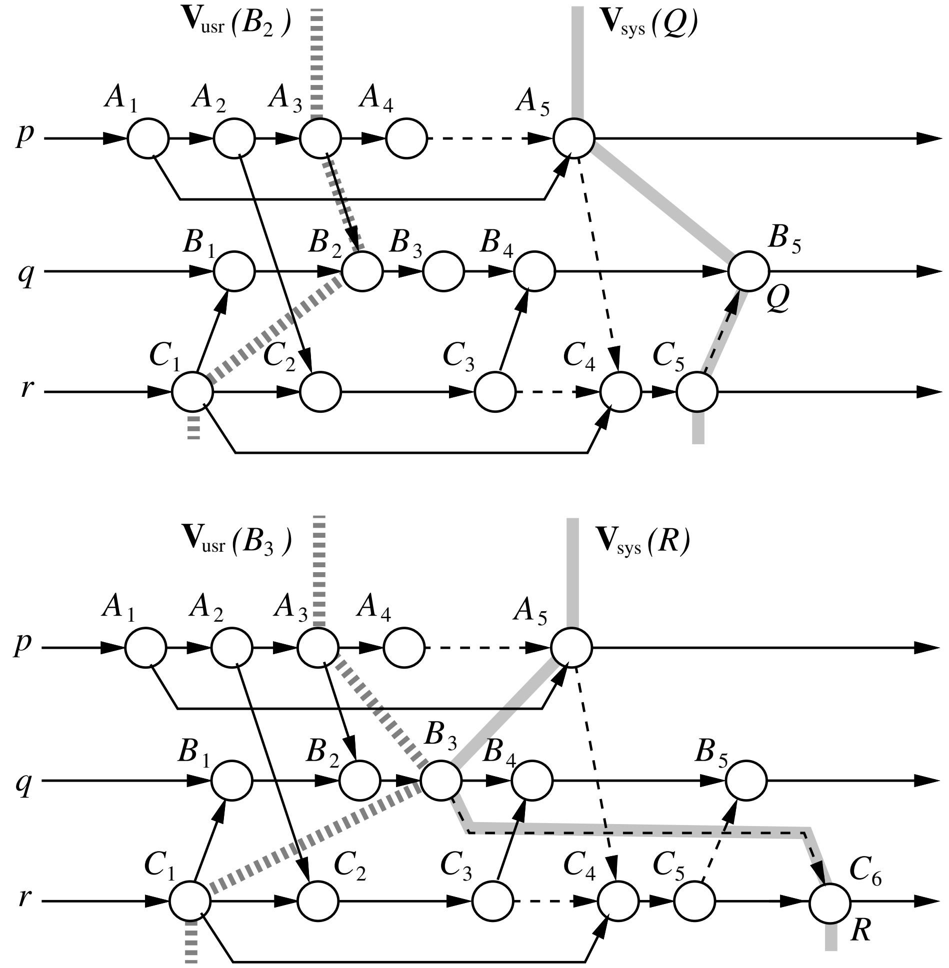

Figure 1 The *DT_ORPHAN* test allows processes both eliminate and prevent orphans. In the top illustration, process q uses the *DT_ORPHAN* test at B_5/Q to deduce that state B_2 is an orphan; in the bottom illustration, process r at C_6 uses the *DT_ORPHAN* test to reject the user message that just arrived from B_3. In both cases, user-precedence fails on the p entries of the vectors.

5. The Distributed Time Rollback Recovery Protocol

5.1. The Protocol

We build our protocol for optimistic rollback recovery by having the system processes maintain vector clocks for *USER_TIME* and *SYSTEM_TIME*, and use these clocks to test for orphans.

Sending a User Message Suppose process p in system state S decides to send a user message M to process q. System process p sends a system message containing M to the system process at q.

Sending a System Message When the system process at p sends a system message M to the system process q, it sends along the timestamp $\mathbf{V}_{\text{sys}}(S)$ (where S is the current system state at p). If M is a forwarded user message, then p includes the timestamp $\mathbf{V}_{\text{usr}}(S_{\text{U}})$. If M is exclusively a system message, then including the $\mathbf{V}_{\text{usr}}$ vector is optional.

/ the orphan test */*
function *DT_ORPHAN*$($*TESTED_STATE*$,$ *TESTING_STATE*$)$
    if $\mathbf{V}_{\text{usr}}(\textit{TESTED\_STATE}) \preceq_{\text{usr}} \mathbf{V}_{\text{sys}}(\textit{TESTING\_STATE})_{\text{U}}$
        then return *false*
        else return *true*

/ receive system message M sent in system state S */*
procedure *RECEIVE*(M)
    / update* SYSPOT *vector*/*
    $\mathbf{V}_{\text{sys}}(\textit{CURRENT}) \leftarrow \mathbf{V}_{\text{sys}}(\textit{CURRENT}) \max_{\text{sys}} \mathbf{V}_{\text{sys}}(S)$
    if *DT_ORPHAN*$(\textit{CURRENT}_{\text{U}}, \textit{CURRENT})$
        then rollback to maximal non-orphan
    if *DT_ORPHAN*$(S_{\text{U}}, \textit{CURRENT})$
        then inform the sender (optional)
        else if M contains a user message
            then
                / update the* USRPOT *vector*/*
                $\mathbf{V}_{\text{usr}}(\textit{CURRENT}_{\text{U}}) \leftarrow \mathbf{V}_{\text{usr}}(\textit{CURRENT}_{\text{U}}) \max_{\text{usr}} \mathbf{V}_{\text{usr}}(S_{\text{U}})$
                accept user message

Figure 2 In the distributed time protocol, a system process rolls itself back if its user state has become an orphan, and then accepts a user message only if its *send* is not an orphan. (Let *CURRENT* be the current event at the executing process.)

Receiving Messages Figure 2 shows the procedure used for receiving messages. Suppose process p receives a system message M sent by q at S. The system process updates the current $\mathbf{V}_{\text{sys}}$ vector. If a comparison of the process's current $\mathbf{V}_{\text{usr}}$ and $\mathbf{V}_{\text{sys}}$ vectors indicates the current user state is an orphan, the system process rolls itself back. If a comparison of the *USER_TIME* vector on the message with the process's current $\mathbf{V}_{\text{sys}}$ vector indicates the message's *send* is an orphan, the system process considers telling q about this. Otherwise, if M contains a user message, the system process forwards it to its user process.

(Suppose the *send* of a user message M user-followed from state A at process r, but process p's current user state depends on B at r, with A and B user-concurrent. At least one of A, B must have been rolled back, and the system timestamp on M will carry that information if p doesn't already know it. Thus, this protocol automatically enforces the invariant that user states are always valid.)

Rollback To roll back because of its own failure, a process restores a state in its live history and creates new incarnation.

To roll back because it discovers it's an orphan, a process needs to find a state in its live history that is not an orphan—that is, a state whose $\mathbf{V}_{\text{usr}}$ timestamp still *USER_TIME*-precedes the current $\mathbf{V}_{\text{sys}}$ vector. Clearly the initial state is not an orphan, and clearly once a user state is an orphan, subsequent user-states are orphans. Thus, for a given value of $\mathbf{V}_{\text{sys}}$, there exists a unique *USER_TIME*-maximal state in the live history that is not an orphan.

How quickly the system recovers depends on how quickly the processes that are (or may become) orphans learn of the restart.

5.2. Implementation Issues

For processes to track *SYSTEM_TIME* and *USER_TIME*, we need to use state labeling at processes that allows comparison both in the system timeline and user tree. We show some sample labeling schemes.

System Timeline We can order states in the system timelines by *index pairs* after Strom and Yemini. We label each system state at process p with two integers: the *incarnation index* and the *state index*. Both are initially 1. If the state following $[i, j]$ is not a restart state, we label it $[i, j + 1]$. If the state following $[i, j]$ is a restart state restoring the state $[i', k]$, then we label it $[i + 1, k]$.

Lexigraphic order sorts system states at a each process. If system state A has index pair $[i, j]$ and A' has $[i', j']$, then $A \prec_{\text{sys}} A'$ iff $(i < i') \ \vee \ (i = i' \ \wedge \ j < j')$. Index pairs also have the convenient property that a new incarnation of a process does not need to exactly how far the old incarnation got.

The User Tree The user tree at a process is a partial order. To sort states in this tree, we can use the familiar tool of vector clocks, with one entry per root-leaf branch. Within any one root-leaf path in the user tree, state indices sort user states. (If Q and R are two system states at a process with $Q_{\mathrm{U}} = R_{\mathrm{U}}$, then the state indices of Q and R will be the same.)

Our natural incarnation is use incarnations for vector entries, but two items require special attention:

1. If a system state A_{S} occurs before the final rollback of a process, then branches in the tree do not yet exist.
2. User states restored after rollback might be construed to be part of two different branches.

We resolve these difficulties by defining the *canonical* occurrence of a user state A to be the system-minimal A_{S}, and limiting our attention to canonical occurrences.

For a user state A at process p, we define the TREE-timestamp vector $\mathbf{V}_{\text{tree}}(A)$ as follows. Let B be the maximal user state at p with $B \Longrightarrow A$ in *USER_TIME* such that the canonical B_{S} occurs in incarnation i. Then the ith entry of $\mathbf{V}_{\text{tree}}(A)$ equals the state index of the canonical B_{S}. If no such B exists, then the i entry of $\mathbf{V}_{\text{tree}}(A)$ is 0. Straightforward integer comparisons on vector entries will determine tree precedence.

A straightforward implementation of *USER_TIME* timestamps using tree-vectors would require two integers for each restart in the computation, which is unreasonably large. However, many optimizations suggest themselves. Suppose the canonical A_{S} occurs in incarnation j. Then the ith entry of $\mathbf{V}_{\text{tree}}(A)$ will be zero for $i > j$, will equal the state index of A for $i = j$, and will be the same for all states in incarnation j for $i < j$. Thus, the suffix of $\mathbf{V}_{\text{tree}}(A)$ doesn't need to be transmitted at all, and the prefix only needs to be transmitted once per incarnation. An incremental approach would bring the information down to constant. This is essentially the technique that Strom and Yemini use—although at the price of having processes block if they need to perform a comparison requiring data they haven't yet received. With this tree-vector optimization, our timestamps are only twice as long as Strom and Yemini's; more careful management of tree vector prefix transmission will avoid blocking (at the price of slightly longer timestamps during recovery).

These implementations are only examples. The $\mathbf{V}_{\text{tree}}$ clocks can tolerate general partial orders, and (without prefix removal) will function for the more general version of rollback where a process wishes to restore a state that it had previously aborted.

6. Security and Privacy

A process can verify the passage of real time by independent physical hardware (e.g., a quartz clock). However, more distributed, virtual models of time do not allow such independent checking. Tracking relations in these models requires sharing private information and trusting the private information that is shared; this trust creates the opportunity for attacks on clocks that translate to attacks on protocols. Our distributed time framework [Sm93, Sm94] identifies these security and privacy problems and builds clocks that protect against them. Thus, the framework allows us to design protocols in terms of our general time formalism, and then transparently consider these security and privacy issues by installing secure and private clocks.

6.1. Attacks

The proper functioning of vector clocks depends on the accuracy of the timestamps on messages. This dependence creates a window for malicious (or merely faulty) processes to disrupt the vector clock protocol. When process p in state G sends a message M to process q, it is supposed to include the timestamp vector $\mathbf{V}(G)$ listing, for each process, the maximal state influencing state S. A malicious process p could lie about its vector entries; a malicious q could use this data for purposes other than sorting states.

An Example To illustrate the security risks of vector clocks, consider an example of running a commodities exchange over a public network (such as the Internet). Using vector clocks to track event ordering in such a system allows corrupt users to commit the crime of *options frontrunning*. Suppose a broker is allowed to trade both for himself and for his client. If a lucky broker happens to buy a small number of shares of an item (for example, orange futures) shortly before receiving a request from his client to buy a large number of shares for her, then he makes a nice profit—the large purchase by his client drives up the value of the broker's shares. This profit can motivate a corrupt broker to wait until he receives the client request B, and then forge a purchase order A of his own that appears not to follow B. This forgery is easy with vector clocks: the corrupt broker merely winds back the entries on the timestamp vector on A. (Options frontrunning occurs in the Chicago commodities exchange, and the technique the FBI uses in this physical environment is to place undercover agents in the pits to look for such "lucky" purchases.)

We briefly identify three classes of attacks on vector clocks:

Malicious Backdating A malicious process p can fool an honest process q into thinking that a state occurred earlier than it really did by saving and reusing old vector entries on messages that it sends. (The *options frontrunning* example demonstrates this technique.)

Malicious Postdating A malicious process p can fool an honest process q into thinking that a state occurred later than it really did by selectively advancing certain vector entries on messages that it sends. For example, a malicious process can leak an advance copy of a public announcement merely by postdating the timestamp vector on the leak. The recipient of this leaked announcement can act on the advanced warning (perhaps by being the first to respond to an enclosed offer), but appear to the rest of the system to have had the same chance as everyone else.

Compromised Privacy A malicious process p can extract information about the activities of process q (and other processes) by examining the changes in entries of timestamp vectors on messages received from process q. If process q is involved in two separate conversations, one with p and one with r, process p can detect the existence of the other conversation and the identity of r. This detection may have serious consequences—for example, retribution against whistleblowers promised anonymity.

6.2. Defenses

Signed Vectors We have identified some of the security and privacy risks associated with vector clocks, and presented the solution of *Signed Vector Timestamps* [SmTy91, see also ReGo93]. The key to this protocol is the realization that the process p entry in *any* timestamp vector should originate with process p. We use the cryptographic tool of *digital signatures* to prove this authorship. In a digital signature scheme, each process knows a function that only it can compute, but that anyone can check. That is, only p can calculate $\mathbf{S}_p(A)$ for a given A, but anyone can examine A and $\mathbf{S}_p(A)$ and know that they match.

Installing and checking signatures on vectors prevents any dishonest process from advancing vector entries for honest processes. Thus, the Signed Vector protocol allows honest processes to correctly determine precedence when the causal path touches only honest processes. However, this protection is still not sufficient: it still permits the three risk scenarios from Section 6.1, and it does not extend to dealing with dependencies that do not flow according to real time.

Sealed Vectors To provide protection against the risk scenarios of Section 6.1, we have developed the *Sealed Vector Timestamp* protocol [SmTy93]. This protocol uses inexpensive *secure co-processors* [TyYe93, Wein87, Wein91, WWAP91, Yee94], that can detect physical tampering and erase their memory. Although secure co-processors provide a limited secure environment, building protocols that effectively take advantage of this secure environment raises some subtle challenges. For example, the co-processors must maintain communication between each other despite malicious attacks, and protocols must be designed to prevent malicious processes from bypassing their co-processors. Through careful use of *bit-secure encryption* [Gold89] and *digital signatures*, the Sealed Vector protocol forces all to consult their co-processors in order to send and receive messages, and to generate and comparing timestamps. The Sealed Vector protocols provides secure and private clocks for arbitrary partial orders. (The presense of covert channels will weaken this protection, however.)

6.3. Rollback

Most existing rollback protocols use some type of vector clock to track causal dependency. These attacks on vector clocks can be used to attack rollback protocols. For example, by hiding a dependency, a malicious process can cause an honest one to delay rolling back. By forging a dependency, a malicious process can cause an honest one to roll back unnecessarily. Asynchronous recovery protocols are also susceptible to attacks based on the partial order of restart knowledge. For example, in the Strom-Yemini protocol, a malicious process can cause an honest process to block indefinitely by sending a user message depending on non-existent incarnations.

Our distributed time framework provides a systematic way to protect against these attacks. By its explicit use of distributed time on both user and system levels, our new protocol is particularly prepared for this protection. We can first design the protocol in terms of the time relations, and then transparently install secure partial order clocks.

7. Conclusions

Existing optimistic rollback recovery protocols use various forms of partial order time to track causal dependency on rolled-back states. Asynchronous recovery is desirable, since minimizing coordination should minimize the computational overhead and maximize the concurrency and flexibility in the protocol. 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 distributed time framework allows us to explicitly track these two levels of time. Applying these tools directly yields a simple optimistic rollback recovery protocol that allows completely asynchronous recovery while also limiting rollbacks to at most one per process after any failure. In addition, our distributed time framework independently addresses the security and privacy issues inherent in protocols based on partial order time. Future work includes using disributed time to explore more general versions of the rollback problem, as well as integrating this work with checkpointing techniques and output commitment.

References

[BhLi88] 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.

[BBG83] A. Borg, J. Baumbach and S. Glazer. "A Message System Supporting Fault Tolerance." *Ninth ACM Symposium on Operating Systems Principles.* 90-99. 1983.

[BBGH89] 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.

[EJZ92] E.N. Elnozahy, D.B. Johnson and W.Zwaenepoel. "The Performance of Consistent Checkpointing." *11th Symposium on Reliable Distributed Systems.* IEEE, 1992.

[ElZw92] 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

[Fi88] C.J. Fidge. "Timestamps in Message-Passing Systems That Preserve the Partial Ordering." *11th Australian Computer Science Conference.* 56-67. February 1988.

[Fi91] C.J. Fidge. "Logical Time in Distributed Computing Systems." *IEEE Computer.* 24 (8):28-33. August 1991.

[Gold89] O. Goldreich. *Foundations of Cryptology.* Computer Science Department, Technion, 1989.

[Jo89] D.B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing.* Ph.D. thesis, Rice University, 1989.

[JoZw90] D.B. Johnson and W. Zwaenepoel. "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing." *Journal of Algorithms.* 11: 462-491. September 1990.

[Jo93] D.B. Johnson. "Efficient Transparent Optimistic Rollback Recovery for Distributed Application Programs." *13th Symposium on Reliable Distributed Systems.* IEEE, October 1993.

[KoTo87] R. Koo and S. Toueg. "Checkpointing and Rollback-Recovery for Distributed Systems." *IEEE Transactions on Software Engineering.* 13 (1): 23-31. January 1987.

[La78] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System." *Communications of the ACM*. 21: 558-565. July 1978.

[LeBh88] P. Leu and B. Bhargava. "Concurrent Robust Checkpointing and Recovery in Distributed Systems." *Fourth International Conference on Data Engineering.* I54-163. IEEE, 1988.

[Ma89] 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.

[PeKe93] S.L. Peterson and P. Kearns. "Rollback Based on Vector Time." *12th Symposium on Reliable Distributed Systems.* IEEE, October 1993.

[PoPr83] M.L. Powell and D.L. Presotto. "Publishing: A Reliable Broadcast Communication Mechanism." *Ninth ACM Symposium on Operating Systems Principles.* 100-109. 1983.

[ReGo93] M. Reiter and L. Gong. "Preventing Denial and Forgery of Causal Relationships in Distributed Systems." *1993 IEEE Symposium on Research in Security and Privacy.*

[SiKs90] M. Singhal and A.D. Kshemkalyani. *An Efficient Implementation of Vector Clocks.* Computer Science Technical Report TR OSU-CISRC-11/90-TR34, Ohio State University. November 1990.

[SiWe89] A.P. Sistla and J.L. Welch. "Efficient Distributed Recovery Using Message Logging." *Eighth ACM Symposium on Principles of Distributed Computing*. 223-238, 1989.

[Sm93] S.W. Smith. *A Theory of Distributed Time.* Computer Science Technical Report CMU-CS-93-231, Carnegie Mellon University. December 1993.

[Sm94] S.W. Smith. *Secure Distributed Time for Secure Distributed Protocols.* Ph.D. thesis, School of Computer Science, Carnegie Mellon University. (In preparation, to appear in Summer 1994.)

[SmTy91] S.W. Smith and J.D. Tygar. *Signed Vector Timestamps: A Secure Protocol for Partial Order Time.* Computer Science Technical Report CMU-CS-93-116, Carnegie Mellon University. October 1991; version of February 1993.

[SmTy93] S.W. Smith and J.D. Tygar. *Sealed Vector Timestamps: Privacy and Integrity for Partial Order Time.* Carnegie Mellon University. November 1993.

[StYe85] R. Strom and S. Yemini. "Optimistic Recovery in Distributed Systems." *ACM Transactions on Computer Systems.* 3: 204-226. August 1985.

[TyYe93] J.D. Tygar and B.S. Yee. "Dyad: A System for Using Physcially Secure Coprocessors." *Proceedings of the Joint Harvard-MIT Workshop on Technological Strategies for the Protection of Intellectual Property in the Network Multimedia Environment.* April 1993. (A preliminary version is available as Computer Science Technical Report CMU-CS-91-140R, Carnegie Mellon University.)

[Wein87] S.H. Weingart. "Physical Security for the μABYSS System." *IEEE Computer Society Conference on Security and Privacy.* 1987.

[Wein91] S.H. Weingart. *Physical Security Devices for Computer Subsystems: A Survey of Attacks and Defenses.* IBM, internal use only. March 1991.

[WWAP91] S.R. White, S.H. Weingart, W.C. Arnold, and E.R. Palmer. *Introduction to the Citadel Architecture: Security in Physically Exposed Environments.* Technical Report, Distributed Security Systems Group, IBM Thomas J. Watson Reaserch Center. March 1991.

[Yee94] B.S. Yee. *Using Secure Coprocessors.* Ph.D. thesis, School of Computer Science, Carnegie Mellon University. (In preparation, to appear in Spring 1994.)