TITLE: MINIMIZING TIMESTAMP SIZE FOR COMPLETELY ASYNCHRONOUS OPTIMISTIC RECOVERY WITH MINIMAL ROLLBACK

AUTHOR(S): Sean W. Smith, David B. Johnson

SUBMITTED TO: IEEE International Symposium on Fault-Tolerant Computing
June 1996

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

By acceptance of this article, the publisher recognizes that the U.S. Government retains a nonexclusive royalty-free license to publish or reproduce the published form of this contribution or to allow others to do so, for U.S. Government purposes.

The Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy.

Los Alamos National Laboratory
Los Alamos New Mexico 87545

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED
Minimizing Timestamp Size for Completely Asynchronous Optimistic Recovery with Minimal Rollback

Sean W. Smith
Computer Research and Applications Group
Los Alamos National Laboratory
Mail Stop B265, Post Office Box 1663
Los Alamos, NM 87544 USA

David B. Johnson
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213 USA

November 30, 1995

Abstract

In this paper, we present a new protocol for optimistic rollback recovery in distributed systems. This protocol is completely asynchronous, minimizes rollback, and is independent of any particular underlying distributed computation to be made fault tolerant. This protocol improves on earlier work in asynchronous optimistic rollback recovery in that previous protocols either sacrificed some of these properties or required larger timestamps. Furthermore, we establish that this protocol is optimal, in that no rollback recovery protocol can achieve these properties and have asymptotically smaller timestamps.

1. Introduction

In this paper, we present a new protocol for optimistic rollback recovery in distributed systems. Our protocol achieves asynchronous recovery in that it neither requires processes to block nor to wait (nor even to send special messages) during recovery. Our protocol minimizes rollback by two metrics: First, our protocol requires a process to roll back at most once in response to a failure, and only if the failure makes the process an orphan. Second, our protocol minimizes the propagation of orphans, in that an orphan process rolls back as soon as it can potentially know it is an orphan, and any process refuses an orphan message as soon as the process can potentially know the message

This research was sponsored in part by the Advanced Research Projects Agency, under contract number DABT63-93-C-9954, and by the Department of Energy, under contract number W-7405-ENG-36. This paper is registered as Los Alamos Unclassified Report LA-UR-95-XXXX. 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. The authors are grateful to Doug Tygar and Vance Faber for their helpful discussions.
is an orphan. In addition, our protocol is independent of any particular underlying computation in that it does not examine the underlying state, program, or message contents; our protocol thus can transparently add fault tolerance to any asynchronous computation.

Our timestamp size is bounded by $O(v \log s)$ bits, where $v$ is the total number of process versions in the computation and $s$ is the maximal number of state intervals in any single version of any process. (A process starts a new version only after each of its own failures; $v = n + F$, where $F$ is the total number of failures in the system and $n$ is the number of processes in the system). Furthermore, we prove that any rollback protocol achieving the above properties (full asynchrony, minimal rollback, computation independence) cannot have a timestamp upper bound smaller than $\Omega(v \log s)$ bits.

This work differs from our earlier protocol [16] by using a third level of partial order time to achieve smaller timestamps; this work differs from [3] by sacrificing no asynchrony or minimality properties.

1.1. Asynchronous, Optimistic Recovery

Rollback recovery addresses the problem of adding fault tolerance to long-running applications on asynchronous distributed systems. An implicit goal of recovery is that the protocol be as transparent as possible to the underlying computation, both during failure-free operation and during recovery. Optimistic message logging is an approach to recovery that attempts to minimize the failure-free overhead, at the expense of complicating recovery from failure. Asynchronous optimistic recovery reduces this cost by allowing recovery to proceed without impacting the asynchrony of the underlying computation.

We assume that processes are piecewise deterministic: 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. We define a state interval to be the period of deterministic computation at a process that is started by the receipt of a message and continues until the next message arrives. If a process $p$ fails and then recovers by rolling back to a previous state, process $p$'s computation since it first passed through the restored state becomes lost. State at surviving processes is an orphan when it causally depends on lost computation.

In message logging protocols, processes checkpoint their local state occasionally, and log all incoming messages. Consequently, a process can restore a previous state by restoring a preceding checkpoint and replaying the subsequent logged messages. (The ability to restore arbitrary previous states eliminates the domino effect [11, 12].) In optimistic protocols, processes log messages by buffering them in volatile memory and sending them to stable storage asynchronously. As a
consequence, the failure of a process before the logging of received messages is successful can cause the state at other processes to become orphans.

1.2. Wishlist for Optimistic Recovery

Ideally, an optimistic recovery protocol should fulfill several criteria:

- **Complete Asynchrony.** The recovery protocol should have no impact on the asynchrony of the underlying computation. In particular, the protocol should meet the following conditions:
  - **No Synchronization.** Recovery should not require processes to synchronize with each other.
  - **No Additional Messages.** Recovery should require any messages to be sent beyond those in the underlying computation.
  - **No Blocking during Recovery.** Recovery should never require execution of the underlying computation to block.
  - **No Blocking during Failure-Free Operation.** Failure-free operation should never require execution of the underlying computation to block. In particular, the protocol should be use *bona fide* optimistic message logging: computation never waits for messages to be logged to stable storage.
  - **No Assumptions.** The protocol should make no assumptions about the underlying communication patterns or mechanisms.

- **Minimal Rollback.** The recovery protocol should minimize the amount of computation lost due to rollback. This property requires minimizing both the number of rollbacks as well as the propagation of orphans, as expressed in these conditions:
  - **Minimal Number of Rollbacks.** The failure of any one process should cause any other process to roll back at most once, and then only if that process has become an orphan.
  - **Immediate Rollback.** A process in an orphan state should roll back as soon as it can potentially know that its current state is an orphan.
  - **No Self-Contamination.** A process $p$ not in an orphan state should not accept a message sent from a process $q$ in an orphan state, if $p$ can potentially know that $q$ was an orphan.

- **Small Timestamp Size.** Optimistic recovery protocols typically require appending timestamp structures to messages. These timestamps should be as small as possible. Our previous work [16] characterized timestamp size in terms of the number of entries. Subsequent work
[3] characterizes timestamp overhead in terms of number of integers, since some entries might be bigger than others. In this paper, we characterize timestamps in terms of the number of bits, in order to maximize accuracy regarding the amount of information in them.

- **Independent of Underlying Computation.** The rollback computation itself is a distributed computation which has the following independence properties:

  - **Process State Opaque.** The user state of processes is opaque to the rollback computation.

  - **Message Content Opaque.** Except for the timestamp and the identity of the source and destination processes, the contents of messages are opaque to the rollback computation.

  - **Process Program Opaque.** The programs (state transition functions) governing the user computation is opaque to the rollback computation.

This independence serves to make the rollback protocol universal, in the sense that it can transparently add fault-tolerance to any underlying computation. Specifying the space of rollback protocols also leads to additional conditions:

  - **Piecewise Determinism.** At each process, the rollback state changes deterministically with each arriving message based on the visible components, with each new state interval, and with each timestamp generation. Each timestamp generation is determined by the state of the rollback computation and the identity of the destination process.

  - **No Needlessly Discarded Messages.** For each incoming message, the rollback protocol can decide to discard the message only when the message is a knowable orphan.

1.3. Previous Work

Strom and Yemini [17] opened up the area of optimistic recovery [1, 3, 5, 6, 7, 9, 10, 13, 16, 18]. Their protocol provided mostly asynchronous recovery, but required blocking and additional messages. Furthermore, their protocol permitted a worst-case scenario where one failure at one process could cause another process to rollback an exponential number of times; this pathology arose from the lack of the Immediate Rollback property. Strom and Yemini used timestamps of $O(n \log s')$ bits, where $s'$ is the maximum number of state intervals in any one process incarnation. Some subsequent work in optimistic recovery minimized the number of rollbacks by sacrificing asynchrony during recovery [6, 10, 13, 3]. Some even achieved $O(\log s')$ bit timestamps [6, 13].

Our earlier protocol [16] achieves fully asynchronous recovery while also minimizing rollbacks and wasted computation. However, we obtained this result by using large timestamps. We required a system timestamp vector consisting of $n$ entries of a pair of integers each, and a user timestamp
vector consisting of \( n \) entries whose total size was \( R \) integers, where \( R \) is the number of rollbacks that have occurred throughout the system. Thus, the number of integers in our timestamps is bounded by \( O(n + R) \). We show in Section 3.4 how system timestamps can be bounded by \( O(v \log s) \) bits. Thus the net size is \( O((v + R) \log s) \) bits; since \( R \leq nF \) this is also bounded by \( O(nF \log s) \) bits.

Damani and Garg [3] present an optimistic protocol that requires no synchronization, minimized the number of rollbacks, and requires timestamps consisting of a version index and a state index for each process. These timestamps are bounded by \( O(n \log v + n \log s) \) bits. (although the \( \log v \) factor might be reduced, since it cannot be the case that all versions occur at all processes.) We note that the same argument from Section 3.4 bounds this by \( O(v \log s) \).

However, the Damani and Garg protocol fails to meet other criteria from Section 1.2:

- **Assumptions.** It requires the assumption of reliable broadcast for failure announcements.
- **Extra Messages.** It requires the extra messages of failure announcements.
- **Blocking.** A process that has received a message from a rolled-back process without receiving the failure announcement will be forced to block if it executes a receive and no other messages have arrived.
- **Constraining Orphan Propagation.** Until it receives a failure announcement, an orphan process will keep on executing—and causing other processes to become orphans—even if it could potentially know that it is an orphan, due to a message that has arrived but is delayed until the announcement arrives. Similarly, even though it could potentially know otherwise, a process that has not received the failure announcement will accept messages sent by another process in an orphan state. In this sense, the protocol does not support minimal rollback.

Essentially the Strom and Yemini protocol with announcements sent only when restarting from failure, the Damani and Garg protocol critically depends on the assumption of reliable broadcast of failure announcements.

### 1.4. This Paper

The Damani and Garg protocol [3] obtains small timestamp size via the implicit use of a novel insight: that keeping track only of the failures of other processes, rather than of their rollbacks, suffices to detect orphans. The number of failures is smaller, since a process rolls back both in response to its own failures as well as in response to failures at other processes.

---

\(^1\) Damani and Garg [3] express this bound as \( O(n^2 f) \) integers, where \( f \) is the maximal number of times any one process has failed, by bounding \( R \) by \( nF \) and bounding \( F \) by \( nf \).
In this paper, we use this insight to reduce our timestamp size to $O(v \log s)$ bits. Furthermore, we establish that this timestamp size is optimal: any protocol meeting the criteria of Section 1.2 cannot have a smaller upper bound on timestamp size.

Section 2 reviews how our protocol functions. Section 3 establishes the correctness of our protocol. Section 4 establishes the optimality of our modified protocol.

2. The Protocol

The technique of using partial order time [4, 8] to describe distributed asynchronous computation is well-known. Time puts a total order on the state intervals at an individual process; the sending of a message makes the state interval containing the send precede the state interval begun by the receive. The transitive closure of the union of these two relations comprises a partial order on the state intervals of all processes.

In this paper we present a new optimistic rollback recovery protocol that explicitly tracks three levels of partial order time. In our previous work [15, 16], we used two levels of partial order time to minimize rollback in completely asynchronous recovery. The system level consists of both the recovery computation and the virtual failure-free computation that it implements; the user level consists of the application computation that fails and recovers from failure by restart and rollback. In this paper, we also introduce a third time model: the failure level, consisting of the application computation that fails, along with the portion of recovery consisting of restart only. The mechanics of partial order time apply to all these levels of time.

2.1. Partial Order Time and Vector Clocks

The motivation behind partial order time is the ability to express the temporal ordering on state intervals that occur at physically separate locations—pairs of state intervals that cannot have influenced each other are best left unordered. In its usual form, partial order time decomposes into a linear timeline for each process, and links (from messages) between each timeline. In previous work [14, 15], we have generalized this structure to allow for more general models at processes, and for hierarchies of time. We use $\prec$ and $\preceq$ to denote time orderings within a process, and $\rightarrow$ and $\iff$ to denote time orderings across processes.

In the context of partial order time, a vector is an array of state intervals (or, more precisely, names or indices of intervals), one per process. The total order on each timeline induces a natural partial order on vectors: we say that vector $V$ precedes vector $W$ when each entry of $V$ precedes or equals the corresponding entry of $W$ in the timeline for that entry. We use the same notation
to compare vectors \((\prec,\preceq)\) that we use for process time, since the vector comparison arises from process time.

For any state interval \(A\), we define its timestamp vector \(V(A)\) as follows: for each process \(p\), the \(p\) entry of \(V(A)\) is the maximal state interval \(B\) at process \(p\) such that \(B \preceq A\). These timestamp vectors function as clocks: for any \(A, B\), \(V(A) \preceq V(B)\) exactly when \(A \implies B\). When each process \(p\) can sort state intervals in the timeline of each other process \(q\), vector clocks are also implementable. Each process \(p\) maintains its current clock; when sending a message, process \(p\) includes the timestamp vector of the send event; when receiving a message, process \(p\) retains the entry-wise maximum of its old timestamp vector and the one on the received message.

In earlier work [14, 15], we show how this mechanism applies to more general forms of time, including partial orders where the local time at individual processes forms timetrees instead of timelines. The key requirement, again, is that processes have the ability to sort state intervals in the timetrees of other processes.

### 2.2. Three Levels of Time

System time organizes the system state intervals at each process into a linear sequence, reflecting the order in which they happened. User time organizes the user state intervals at each process into a tree, with a new branch beginning each time the process rolls back and restarts. The system-level computation implements the user-level computation. All user-level messages are carried on system-level messages, but system messages can have extra content, just as the system-level state at a process contains the user-level state, but can have extra state as well.

The failure-level computation forms an intermediate level between the user level and the system level. Failure time also applies to user state intervals, and also organizes the state intervals at each process into timetrees. However, failure time begins a new branch in the process timetree only when a process restarts after its own failure, not after rollback due to the failure of another process. Tracking failure time is possible: a process can number state intervals across these excluded rollbacks since the process itself did not lose state. Tracking failure time is sufficient: although processes need to know about rollbacks elsewhere, knowledge of failures elsewhere communicates equivalent information—since all rollbacks have a first cause in a failure.

When appropriate, we use subscripts to indicate whether a state interval or comparison is made in user time, system time, or failure time—e.g., \(A_s \prec_s B_s\) compares two system state intervals in their process timeline. We omit subscripts on the partial order time comparison, since the partial order time model is implied by the subscripts on the state intervals.

Each system state interval maps to exactly one user state interval (because we begin a new system interval each time we begin a new user interval); we denote this mapping by \(\text{SYS} \_\text{TO} \_\text{USR}\).
Each user state interval maps to the set of system state intervals that occurred during that user state interval; we denote this map by $USR \to SYS$.

An isomorphism exists between user state intervals and failure state intervals; we use $USR \to FLR$ to denote this map from user states to failure states, and $FLR \to USR$ to denote its inverse. We can use this isomorphism to define mappings between system state intervals and failure state intervals:

- We use $SYS \to FLR$ to indicate the map from system state intervals to failure state intervals: $SYS \to FLR = USR \to FLR \circ SYS \to USR$.

- We use $FLR \to SYS$ to indicate the map from a failure state interval to the set of corresponding system state intervals: $FLR \to SYS = USR \to SYS \circ FLR \to USR$.

The fact that a system state interval corresponds to exactly one user state interval means that we can use this correspondence to compare user states to system states within a process’s timetree. The fact that user state intervals are isomorphic to failure state intervals give us many ways to perform cross-model comparison:

- Suppose $A_U$ is a user state interval at a process, and $B_S$ is a system state interval at the same process. Since $B_S$ corresponds to exactly one user state interval $B_U = SYS \to USR(B_S)$, we can compare $A_U$ to $B_S$ in the user timetree at that process by comparing $A_U$ to $B_U$. We indicate comparison of user states to system states within the timetree at one process by $\prec_{US}$.

- We use $\prec_{FS}$ and $\preceq_{FS}$ to indicate the comparison of a failure state interval with the unique failure state interval that corresponds to a system state interval. That is,

$$A_F \prec_{FS} B_S \iff A_F \prec_F SYS \to FLR(B_S)$$

- We use $\prec_{UFS}$ and $\preceq_{UFS}$ to indicate the comparison of a user state interval to a system state interval by converting both to failure state intervals. That is,

$$A_U \prec_{UFS} B_S \iff USR \to FLR(A_U) \prec_U SYS \to FLR(B_S)$$

We extend these comparisons (defined for state intervals at a single process) to compare vectors in the natural way.
2.3. **Knowable Orphans**

The heart of our protocol is an insight into when a process can know a user state is an orphan, and a novel method of testing when this property holds.

Suppose $A_U$ is a user state interval at process $p$, and $B_S$ is a system state interval at process $q$. Process $q$ in state $B_S$ can know that user state $A_U$ is an orphan when:

- Process $q$ in $B_S$ knows about $A_U$.
- State $A_U$ has been made an orphan by causally following state lost due to a restart $R$ (after either rollback or failure).
- Process $q$ in $B_S$ can know about $R$.

We define a predicate $KNOWABLE\_ORPHAN(A_U, B_S)$ to capture this property. The predicate $KNOWABLE\_ORPHAN(A_U, B_S)$ is defined when $A_S \Rightarrow B_S$ for some $A_S \in USR\_TO\_SYS(A_U)$. $KNOWABLE\_ORPHAN(A_U, B_S)$ is true iff there exists, at some process $r$, a user state interval $C_U$ and system state interval $D_S$ satisfying:

- $C_U \Rightarrow A_U$;
- $D_S$ rolls back $C_U$; and
- $D_S \Rightarrow B_S$

Each time a process $q$ receives a system-level message, it checks whether its current user state is a knowable orphan; before a process $q$ accepts a user-level message, it checks whether the user state that sent the message is a knowable orphan.

Tracking the system-level and the user-level of partial order time allows process $q$ in system state $B_S$ to determine if a user state $A_U$ is a knowable orphan. Process $q$ merely needs to map each entry of the system timestamp vector on $B_S$ to its corresponding user state interval, and then do a vector comparison with the user timestamp vector on $A_U$. However, when the protocol also tracks failure time and meets some minimal rollback conditions, process $q$ can perform this comparison of vectors via their failure-time images.

**Theorem 1** Suppose a rollback protocol only rolls back two classes of state intervals:

- those that are lost due to failure of their processes, and
- those that are knowable orphans.
Suppose user state interval $A_U$ at process $p$ and system state interval $B_S$ at process $q$ satisfy $A_S \rightarrow B_S$, 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 $KNOWABLE\_ORPHAN(A_U, B_S)$ is true if and only if $X_U \not\rightarrow_{UFS} Y_S$.

(We prove Theorem 1 in Section 3.3.)

3. Proof of Correctness

3.1. Relations among Time Models

User precedence implies failure precedence.

**Lemma 1** Let $A_U$ and $B_U$ be user state intervals; let $A_F = USR\_TO\_FLR(A_U)$ and $B_F = USR\_TO\_FLR(B_U)$. If $A_U \rightarrow B_U$ then $A_F \rightarrow B_F$.

*Proof* Each branch in a failure timetree also is a branch in the corresponding user timetree. Consequently, each path in a user timetree is also a path in the failure timetree. Thus the lemma holds for state intervals at any one process; since the cross-process links are the same for both time models, the lemma holds in general. □

Failure precedence implies system precedence.

**Lemma 2** Suppose failure state interval $A_F$ and system state interval $B_S$ satisfy $A_F \rightarrow SYS\_TO\_FLR(B_S)$. Let $A_S$ be the minimal interval in $FLR\_TO\_SYS(A_F)$. Then $A_S \rightarrow B_S$.

*Proof* Let $B_F = SYS\_TO\_FLR(B_S)$. We establish this result by induction: If $A_F$ and $B_F$ occur at the same process, this is easily true. If $A_F$ sends a message that begins $B_F$, then some interval in $FLR\_TO\_SYS(A_F)$ precedes $B_S$, so clearly $A_S$ must also. For more general precedence paths, choose an intermediate node $C_F$ with $A_F \rightarrow C_F \rightarrow B_F$, and choose the minimal $C_S$ from $FLR\_TO\_SYS(C_F)$. Establish the result for $A_F$ and $C_F$, and for $C_F$ and $B_F$. □

User precedence also implies system precedence.

**Lemma 3** Suppose user state interval $A_U$ and system state interval $B_S$ satisfy $A_U \rightarrow SYS\_TO\_USR(B_S)$. Let $A_S$ be the minimal interval in $USR\_TO\_SYS(A_U)$. Then $A_S \rightarrow B_S$.

(The proof can be found in [16], although the lemma follows directly from the previous two.)
3.2. Knowable Orphans and Failure Time

The knowable orphan definition is given in terms of rollbacks. We establish that knowable orphans can be characterized in terms restart after a process's own failure (a subset of rollbacks).

**Lemma 4** Suppose the only the user state intervals rolled back are those that are lost due to failure of their processes, and those that are knowable orphans. Then $KNOWABLE\_ORPHAN(A_U, B_S)$ is true iff there exists a $C_U$ and $D_S$ (both at some process $r$) such that:

1. $C_U \rightarrow A_U$
2. $C_U$ is lost due to failure of process $r$, and $D_S$ is the restart after that failure
3. $D_S \rightarrow B_S$

**Proof** If such a $C_U, D_S, r$ exist, then $KNOWABLE\_ORPHAN(A_U, B_S)$ clearly holds, since restart after failure is a special case of rollback.

Conversely, suppose $KNOWABLE\_ORPHAN(A_U, B_S)$ holds. By definition, there exists a $C_U^1$ and $D_S^1$ at process $r^1$ such that

- $C_U^1 \rightarrow A_U$
- $D_S^1$ rolls back $C_U^1$; and
- $D_S^1 \rightarrow B_S$

By the assumed causes of rollback, at least one of the following statements must be true:

- $C_U^1$ is lost due to failure of $r^1$, and $D_S^1$ is the restart after that failure
- $KNOWABLE\_ORPHAN(C_U^1, D_S^1)$ is true

If the latter, then we can iterate; since computations are finite, eventually we reach some $C_U^k, D_S^k, r^k$ such that former rollback cause holds. \n
We also establish some relations among lost states and failure time.

**Lemma 5** Suppose $A_U$ is lost due to failure, and $B_S$ is the restart after that failure. Then:

- If $A_S \in USR\_TO\_SYS(A_U)$ then $A_S \prec_S B_S$.
- If $C_S$ satisfies $B_S \preceq_S C_S$ and $A_F = USR\_TO\_FLR(A_U)$, then $A_F \not\preceq_F S C_S$.

**Proof** The first statement holds because we can only restart after failure has occurred. The second statement holds because lost states remain lost. \n
11
3.3. The Orphan Test

We establish the correctness of our new protocol.

Proof (of Theorem 1) Suppose $KNOWABLE\_ORPHAN(A_U, B_S)$ holds. Lemma 4 gives us that at some process $r$, there exists a user state interval $C_U$ and system state interval $D_S$ satisfying the statements:

- (1) $C_U \rightarrow A_U$,
- (2) $C_U$ is lost due to failure, whose restart was $D_S$, and
- (3) $D_S \rightarrow B_S$.

Let $C_F = USR\_TO\_FLR(C_U)$. Statement (1) implies that $C_U \preceq X_U[r]$, and thus $C_F \preceq USR\_TO\_FLR(X_U[r])$. Statement (2) and Lemma 5 imply that $C_F \not\preceq_{FS} E_S$ for any $E_S$ satisfying $D_S \preceq_{FS} E_S$. Statement (3) implies that $D_S \preceq_{FS} Y_S[r]$. Hence $C_F \not\preceq_{FS} Y_S[r]$. If $X_U[r] \preceq_{US} Y_S[r]$, then $C_F \preceq_{FS} Y_S[r]$ since a failure time path exists from $C_F$ to $X_U[r]$ in the failure timetree at $r$. Thus $X_U[r] \not\preceq_{US} Y_S[r]$.

Conversely, suppose $X_U \not\preceq_{US} Y_S$. Then there exists a process $r$ with $X_U[r] \not\preceq_{US} Y_S[r]$.

Let $C_U = X_U[r]$; let $C_F = USR\_TO\_FLR(C_U)$; let $C_S$ be the minimal state interval in $USR\_TO\_SYS(C_U)$.

By hypothesis, some $A_S \in USR\_TO\_SYS(A_U)$ satisfies $A_S \rightarrow B_S$. By the definition of a timestamp vector, $C_U \preceq A_U$. By Lemma 3 $C_S \preceq A_S$. Thus $C_S \rightarrow B_S$. Applying the definition of timestamp vector again yields $C_S \preceq_{FS} Y_S[r]$.

Since by hypothesis $C_F \not\preceq_{FS} SYS\_TO\_FLR(Y_S[r])$, a $D_S$ must exist such that $C_S \prec_{FS} D_S \preceq_{FS} Y_S[r]$ and $D_S$ restarts $r$ after a failure that lost $C_U$. Since $D_S \rightarrow B_S$, we have $KNOWABLE\_ORPHAN(A_U, B_S)$.

\[\square\]

3.4. Bookkeeping

At first glance, the new protocol may appear to require that processes still be able to sort within user timetrees, since processes still maintain user timestamp vectors. However, we establish that, for purposes of user timestamp vectors on state intervals that cannot be known to be orphans, tracking failure time suffices.

Lemma 6 Suppose user state intervals $A_U$ and $B_U$, and system state interval $C_S$ satisfy:
• \( A_S \rightarrow C_S \), for some \( A_S \in USR\_TO\_SYS(A_U) \)
• \( B_S \rightarrow C_S \), for some \( B_S \in USR\_TO\_SYS(B_S) \)
• neither KNOWABLE\_ORPHAN\((A_U, C_S)\) nor KNOWABLE\_ORPHAN\((B_U, C_S)\) hold.

Let \( A_F = FLR\_TO\_USR(A_U) \) and \( B_F = FLR\_TO\_USR(B_U) \). Then \( A_F \rightarrow B_F \) iff \( A_U \rightarrow B_U \).

Proof  Lemma 1 provides \( A_U \rightarrow B_U \Rightarrow A_F \rightarrow B_F \). So we only need to establish the converse. Suppose \( A_F \rightarrow B_F \). The failure time path from \( A_F \) to \( B_F \) decomposes into a sequence of one or more failure timetree paths, each separated by a message. If \( A_U \nleftrightarrow B_U \), then at least one of these failure timetree paths is not a user timetree path. Suppose \( D_F \leq E_F \) at process \( r \) is the first such path from \( A_F \); let \( D_U = FLR\_TO\_USR(D_F) \) and \( E_U = FLR\_TO\_USR(E_F) \); let \( E_S \) be the minimal interval in \( FLR\_TO\_SYS(E_F) \). By choice of \( r \), \( A_U \rightarrow D_U \). By Lemma 2, \( E_S \rightarrow B_S \), thus by hypothesis, \( E_S \rightarrow C_S \). However, since \( D_U \nleq E_U \), a state interval in that segment must have rolled back \( D_U \). Hence \( KNOWABLE\_ORPHAN\((A_U, C_S)\) \).  

As a consequence, the new protocol needs to be able track system timestamp vectors, failure timestamp vectors, and to map from system event intervals to failure event intervals.

• **System Vectors.** Tracking system vectors takes one entry for each process, with two integers for each entry. One integer is bounded by \( O(\log v_i) \) bits (where \( v_i \) is the number of versions at process \( P_i \)); the other is bounded by \( O(\log s) \) bits. The net contribution of the timestamp is bounded by \( O(n \log s + \sum_i \log v_i) \) bits. Since \( \log v_i = O(v_i) \), we have \( O(v \log s) \) bits for system vectors.

• **Failure Vectors.** Tracking failure vectors takes one entry for each process. That entry consists of one index of \( O(\log s) \) bits for each version at the process; thus the net contribution is \( O(v \log s) \) bits.

• **Mapping.** Mapping a system vector to a failure vector consists of, at worse, carrying around an extra copy of the failure vector.

Thus, the straightforward implementation of tracking indices requires the total timestamp size to be bounded above by \( O(v \log s) \) bits.
4. Asymptotic Optimality

We now establish that, for any optimistic recovery protocol meeting the requirements of Section 1.2, computations exist where the upper bound on timestamp size must be at least $\Omega(v \log s)$ bits. This result establishes the asymptotic optimality of timestamp size in our new protocol.

A restarted state interval occurs when a process restarts after its own failure. At each process, the first version begins with state interval 0. The $j$th restarted state interval (ordered by time) begins version $j + 1$. Each new version must begin with the restart of a state interval that was active in the previous version. As a consequence, for any one process, we can unambiguously label the first interval in each version with an index relative to the start of computation. These indices form a non-descending sequence. For a state interval $S$ at a process, define $\Delta S$ to be the index of $S$ relative to the most recent preceding element in this sequence. For completeness, we define $\Delta S = 0$, where $S$ is the initial state interval of a process.

Suppose $M$ is a message sent in state interval $S$ at process $p$. Define $\mathcal{F}(M)$ to be the set of restarted intervals that causally precede the state interval in which $M$ was sent. Define $\mathcal{V}(M)$ to be the set of state intervals in the timestamp vector of $S$.

Since many definitions of asymptotic complexity only discuss functions of one variable, we review the more general definitions [2]:

- A function $f(v, s)$ is in $\Omega(g(v, s))$ when there exist constants $c, v_0, s_0$ such that for any pair $v, s$ with $v \geq v_0$ and $s \geq s_0$, $0 \leq cg(v, s) \leq f(v, s)$.

- A function $f(v, s)$ is in $O(g(v, s))$ when there exist constants $c, v_0, s_0$ such that for any pair $v, s$ with $v \geq v_0$ and $s \geq s_0$, $0 \leq f(v, s) \leq cg(v, s)$.

**Theorem 2** There exists a function $g(v, s)$ in $\Omega(v \log s)$ such that for any rollback protocol satisfying the criteria in Section 1.2 and for any $v, s$, there exists a computation in which

- some message $M$ must be timestamped with at least $g(v, s)$ bits
- $v$ is the number of process versions in the computation perceivable by $M$, and
- $s$ is the maximum number of state intervals in any one version in this computation.

**Proof** For any $v, s$ (where $v$ and $s$ are each beyond some constant), we construct a class $\mathcal{C}(v, s)$ of computations where $v$ is the number of versions and $s$ is the maximum number of state intervals in any one version as follows.

Choose $k \leq v - 1$ and $n = k + 3$. Let us distinguish processes:

- $P_S$, the sender;
• $P_R$, the receiver;
• $P_C$, the clock, and
• $P_i$ through $P_k$, the failers.

(We use the clock solely to send out the messages that begin state intervals.) Distribute $v - (k + 1)$ failures among the $P_i$. Let each version run out to $s$ state intervals. Let us assume that the $P_i$ only ever restart from even state intervals, and only send messages out in odd state intervals. Furthermore, suppose in each odd state interval in each version, each $P_i$ sends messages both to $P_S$ and $P_R$. For each $i$, at least one message has made it from $P_i$ to $P_S$, and all messages that do arrive have not been lost. For each $i$, let the most recent message to arrive be $M_i$, sent in interval $S_i$ in version $V_i$.

Now, in state interval $S$, $P_S$ is preparing to send a message $M$ to $P_R$. Define the configuration of $P_S$ at this point to consist of the following:

• For each $i$, the sequence $\Delta F$ for $F \in \mathcal{F}(M_i)$
• For each $i$, the value $\Delta S_i$

We now establish that $P_S$ cannot send the same timestamp on $M$ in two different configurations. Suppose $P_S$ uses the same timestamp in two different configurations. One of two cases holds:

1. At some $P_i$, the $\Delta F$ sequence differs. Let $j$ be the first restart where a difference occurs: the $j$th version in configuration $C_1$ began earlier than the $j$th version in configuration $C_2$. By assumption, there exists at least one odd state interval in version $j - 1$ between these restarts, and a message $M'$ was sent to $P_R$ during this interval. Since the configurations do not differ until later and since the rollback protocol is piecewise deterministic, the timestamp on $M'$ is the same in both configurations. However, $M'$ is rolled-back in $C_2$ but not in $C_1$. Suppose $M'$ is the only message $P_R$ has actually received. Should it then receive $M$, whether $P_R$ needs to roll back or not depends on the configuration—which $P_R$ cannot distinguish if $P_S$ uses the same timestamp on $M$ in both.

2. The $\Delta F$ sequences are identical, but at some $P_i$, the $\Delta S_i$ has a different value. Suppose WLOG that this occurs at process $P_i$, in version $j$: the successful send in configuration $C_1$ occurs earlier than the successful send in configuration $C_2$. By assumption, there exists at least one even state interval between the index of the $S_i$ intervals in the two configurations. In either configuration, the computation might can by having version $j + 1$ begin from this interval. Then $S_1$ is rolled-back in $C_2$ but not in $C_1$. Suppose $M$ is the only message that $P_R$ actually receives, until it later receives a message $M'$ directly from $P_1$, sent in version
$j + 1$. $P_R$ can accept $M'$ in $C_1$ but must first roll back in $C_2$. Since $P_1$ has no information to contribute regarding whether $P_S$ was in $C_1$ or $C_2$ when it sent $M$, $P_R$ must get this information from the timestamp on $M$.

Let $W(v, s)$ be the number of configurations for $C(v, s)$. $W(v, s)$ equals the number of ways the restarts and the $S_i$ could have been laid out. Since each restart and each $s_i$ can occur at any even interval among $s$, we have:

$$W(v, s) \in \Omega((\frac{s}{2})^v)$$

Thus for some $c$ and for $W(v, s)$ sufficiently large, the number of bits necessary to distinguish membership in a set of $W(v, s)$ objects is at least $cv \log \frac{s}{2}$ for at least some of these objects. □

This lower-bound proof based on failures does not generalize to the case of rollbacks because all rollbacks have first causes. If two configurations differed solely because of a rollback at some $P_i$, process $P_S$ would still have the same knowledge of failures. The behavior of process $P_R$ would be fixed on the basis of this knowledge; however, in one configuration, $P_i$ would be an orphan. (This scenario suggests a potential desired characteristic for rollback: processes that can know other processes are orphans telling them about this orphan status.)

**Corollary 1** For any rollback protocol satisfying the conditions of Section 1.2, the upper bound on timestamp size is at least $\Omega(v \log s)$.

We could generalize the above theorem to computations based on the number of state intervals in each version (instead of having each be length $s$). We note that our new protocol uses timestamps within a constant size of the minimal indexing for these configurations—which suggests that our protocol is close to optimal even the maximum $s$ is not often achieved.

## 5. Future Directions

Previous work has shown how timestamp size can be reduced by sacrificing asynchrony or minimal rollback. Our results yield an optimal timestamp size while preserving asynchrony and minimal rollback. However, our lower bound proof holds only asymptotically, and for independent, deterministic rollback protocols. Each of these conditions suggests an avenue for further research:

- **Relaxing Determinism.** Optimistic rollback protocols that use randomness might achieve lower timestamp size.

- **Reducing Practical Size.** Optimistic rollback protocols might use timestamps with the same asymptotic bound but with a smaller constant. Optimistic rollback protocols might also reduce the average size of timestamps.
• **Relaxing Independence.** Optimistic rollback protocols might exploit properties of the underlying computation to reduce timestamp size (essentially by re-using information present in the messages themselves and in the process states).

References


