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

AUTHOR(S): S. W. Smith, D. B. Johnson

SUBMITTED TO: IEEE 1996 15th Symposium on Reliable Distributed Systems
Niagara, Canada
October 23-25, 1996
DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.
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.
Minimizing Timestamp Size for Completely Asynchronous Optimistic Recovery with Minimal Rollback

Sean W. Smith
IBM Research Division
T. J. Watson Research Center
Hawthorne, NY 10532
sean@watson.ibm.com

David B. Johnson
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213
dbj@cs.cmu.edu

Abstract
Basing rollback recovery on optimistic message logging and replay avoids the need for synchronization between processes during failure-free execution. Some previous research has also attempted to reduce the need for synchronization during recovery, but these protocols have suffered from three problems: not eliminating all synchronization during recovery, not minimizing rollback, or providing these properties but requiring large timestamps. This paper makes two contributions: we present a new rollback recovery protocol, based on our previous work, that provides these properties (asynchronous recovery, minimal rollback) while reducing the timestamp size; and we prove that no protocol can provide these properties and have asymptotically smaller timestamps.

1. Introduction
Rollback recovery can provide fault tolerance for long-running applications in asynchronous distributed systems. Basing rollback recovery on optimistic message logging and replay avoids the need for synchronization during failure-free operation, and can add fault tolerance transparently. In their seminal paper, Strom and Yemini [28] removed most synchronization from recovery, but permitted a worst case in which a single failure could lead to an exponential number of rollbacks. In our 1995 paper [27], we eliminated all synchronization and minimized the number of rollbacks, but used large timestamps. Damani and Garg [7], in a subsequent paper, further reduced timestamp size, but sacrificed some asynchrony and minimality properties.

In this paper, we make two contributions. First, we present a new optimistic rollback recovery protocol, based on our earlier work [27], that preserves all properties of asynchronous recovery and minimal rollback, but reduces the timestamp size over our previous protocol. Second, we prove that no optimistic recovery protocol can have a smaller bound on timestamp size and still preserve all of these properties.

1.1. Asynchronous, Optimistic Recovery
In an asynchronous distributed computation, processes pass messages that either arrive after some unbounded, unpredictable positive delay, or never arrive at all. Rollback recovery may be used to add fault tolerance to long-running applications on asynchronous distributed systems. An implicit goal of this 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 removing the need for synchronization between processes during recovery, and allowing recovery to proceed without impacting the asynchronous nature of the underlying computation.

We assume that processes are piecewise deterministic: a process’s execution between successive received messages is completely determined by the process’s 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. The state at a surviving process is an orphan when it causally depends on such lost computation.
A process begins a new incarnation when it rolls back and restarts in response to anyone's failure [28]. A process begins a new version when it rolls back and restarts only in response to its own failure [7].

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 in the order originally received. (The ability to restore arbitrary previous states, between checkpointed states, eliminates the domino effect [22, 23].) In optimistic protocols, processes log messages by buffering them in volatile memory and later writing them to stable storage asynchronously. As a consequence, the failure of a process before the logging of some received messages completes can cause the state at other processes to become orphans—since the failed process may have sent messages during a state interval (now lost) begun by the receipt of such an unlogged message.

1.2. Wishlist for Optimistic Rollback Recovery

Ideally, an optimistic rollback 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 not require any messages to be sent beyond those in the underlying computation.

No Blocking During Recovery. Recovery should not force execution of the underlying computation to block.

No Blocking During Failure-Free Operation. Failure-free operation should not force execution of the underlying computation to block. (In particular, computation should never wait for messages to be logged to stable storage.)

No Assumptions. The protocol should make no assumptions about the underlying communication patterns or protocols.

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 New 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 some type of timestamp structures to messages. These timestamps should be as small as possible.

Independence of Underlying Computation. The rollback computation itself is a distributed computation, which should have the following independence properties:

Process State Opacity. The user state of processes is opaque to the rollback computation.

Message Content Opacity. 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 Opacity. The programs (state transition functions) governing the user computation are opaque to the rollback computation.

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

Piecewise Determinism of Rollback Computation. At each process, the state of the rollback computation 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

In this paper, we concentrate on rollback based on optimistic message logging and replay. Recovery protocols based instead on checkpointing without message logging (e.g., [1, 3, 4, 5, 8, 15, 16, 29]) may force processes to roll back further than otherwise required, since processes can only recover states that have been checkpointed. Recovery protocols based on pessimistic message logging (e.g., [2, 9, 11, 21]) can cause processes to delay execution until incoming messages are logged to stable storage. In this section, we discuss previous work in optimistic message logging and replay, for protocols that reduce the need for synchronization during recovery. Table 1 summarizes this work and compares it to the work described in this paper.

Parameters. To discuss the timestamp size required by an optimistic recovery protocol, we need to introduce some parameters. Let \( n \) be the number of processes in the system. In a particular execution of the system, let \( F, R, V \) be the total number of failures, rollbacks, and versions (respectively)
across all processes \((V = n + F)\). We introduce three measures of state intervals: let \(s_I\) be the maximal number of state intervals in any single incarnation of any process \([28]\); let \(s_V\) be the maximal number in any single version \([7]\); and let \(s_L\) be the maximal number in any single live history \([28]\). We have \(s_I \leq s_V\), since many incarnations may comprise a single version.

Additionally, let \(s_Y\) be the maximum number of system state intervals (defined below) in an incarnation; \(s_I \leq s_Y\). Let \(v_i\) be the number of versions at the \(i\)th process, and let \(r_i\) be the number of rollbacks. Let \(r_M\) be the maximal number of rollbacks at any one process.

Of the previous work shown in Table 1, Strom and Yemini \([28]\) use the smallest timestamps, followed by Damani and Garg \([7]\), followed by our previous protocol \([27]\). The timestamp size required by the protocol presented in this paper is substantially less than in our previous protocol. But this timestamp size is still larger than in Damani and Garg’s protocol, although unlike their protocol, our protocol fully preserves all properties of asynchronous recovery and minimal rollback.

### Strom and Yemini

Strom and Yemini \([28]\) opened the area of optimistic recovery. Their protocol provided mostly asynchronous recovery, but required some blocking and additional messages. Furthermore, their protocol permitted a worst-case scenario in which 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 described in Section 1.2. Strom and Yemini used timestamps of size \(O(n \log s_L)\) bits. Some subsequent work in optimistic recovery minimized the number of rollbacks by sacrificing asynchrony during recovery \([13, 20, 24, 7]\), and some of these even reduced the timestamp size to \(O(\log s_I)\) bits \([13, 24]\).

### Smith, Johnson, and Tygar

Smith, Johnson, and Tygar \([27]\) achieve fully asynchronous recovery while also minimizing rollbacks and wasted computation. However, we obtained this result by using large timestamps.\(^1\) We introduced a second level of partial order time, separating the user computation from the system computation of the rollback recovery protocol itself that is transparent to the user computation. 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 \(O(R)\) integers. Thus, the number of integers in our timestamps is bounded\(^2\) by \(O(n + R)\). In terms of bits, the system timestamp vector is bounded by \(\sum_i (\log r_i + \log s_Y)\) bits; as written, the user timestamp vector is bounded by \(\sum_i (\log r_i + \log s_I)\), but a straightforward modification replaces the \(s_I\) by \(s_Y\). Together, the timestamps require \(O(n \log s_Y + R \log s_I + R \log r_M)\) bits.

### Damani and Garg

Damani and Garg \([7]\) present an optimistic protocol that requires little synchronization, minimizes the number of rollbacks, and requires timestamps consisting of a version index and a state index for each process. These timestamps are bounded by \(O(n \log V + n \log s_Y)\) bits (although the \(\log V\) factor might be reduced, since it cannot be the case that all versions occur at all processes.)

However, the Damani and Garg protocol fails to meet other criteria from Section 1.2. In particular, it requires extra messages for failure announcements and assumes reliable broadcast for them. In addition, it may cause blocking during recovery, as 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. Finally, the protocol allows new contamination by orphan processes, since an orphan process will continue executing until it receives a failure announcement, and a process that has not yet received the failure announcement will accept messages sent by an orphan process, even if either could potentially know that the process is in fact an orphan.

### 1.4. This Paper

Section 2 presents our new recovery protocol, and Section 3 demonstrates how it reduces timestamp size to \(O(V \log s_Y)\) bits. Section 4 establishes that this timestamp size is optimal, in that any protocol meeting the criteria of Section 1.2 cannot have a smaller upper bound on timestamp size. The Appendix presents the proofs of these arguments.

---

\(^1\)In that paper, we characterized timestamp size in terms of the number of entries. Damani and Garg \([7]\) characterize timestamp size in terms of number of integers, since some entries may require more than one integer.

\(^2\)Damani and Garg \([7]\) 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 \(nf\).
2. The Protocol

The technique of using partial order time [10, 18, 25] to describe distributed asynchronoum computation is well-known. Experience puts a total order on the state intervals at each individual process; the sending of a message makes the state interval containing the send preceed 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. As described in our earlier work [26], issues such as failure require generalizations such as timetrees (partial orders on the state intervals at individual processes) and multiple levels of time.

Section 2.1 reviews partial order time. The optimistic rollback recovery protocol presented in this paper is defined in terms of four levels of partial order time, and Section 2.2 describes these four levels. Section 2.3 reviews the concept of knowable orphans and how to write rollback protocols in terms of knowable orphan tests. Section 2.4 uses vector clocks for these levels of time to build a more efficient test for knowable orphans. Plugging this test into the scheme of Section 2.3 produces our new protocol.

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—if two state intervals cannot have influenced each other, then neither interval should precede the other in the partial order. In its usual form, partial order time decomposes into linear timelines (one for each process) and links (one for each received message) between each timeline. In previous work [25, 26], 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 single process, and \( \rightarrow \) and \( \mapsto \) to denote time orderings across two or more 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 \( \prec \), \( \preceq \), \( \rightarrow \), and \( \mapsto \) 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 \( A \) and \( B \), \( V(A) \preceq V(B) \) exactly when \( A \mapsto B \). 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 on the message, and when receiving a message, process \( p \) sets its own timestamp vector to the entry-wise maximum of its current value and the timestamp vector on the received message.

In earlier work [25, 26], we show how this mechanism applies to more general forms of time, including partial orders in which 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. Four Levels of Time

Our earlier protocol [26, 27] introduced the notion of system time and user time. System time organizes the 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 timetree, with a new branch beginning each time the process rolls back and restarts. The system-level computation implements the user-level computation, and there may thus be a number of individual states in system time corresponding to each state in user time. All user-level messages are carried in system-level messages, but system messages can have extra content, just as the user-level state at a process is contained within the system-level state, which itself can contain extra information.

In this paper, we introduce two intermediate levels, as illustrated in Figure 1 for a single process, \( p \).

The first new level is failure time, which reproduces the relevant properties of user time but is more efficient to track. 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.

---

**Figure 1. Four levels of time at a process \( p \).**
not after rollback due to the failure of another process. That
is, in user time, a new branch begins with each process
incarnation, whereas in failure time, a new branch begins
with each process version. Tracking failure time is possible
because a process does not lose system-level state when it
rolls back due to a failure of a process other than itself; the
process can thus continuously number its own state intervals
across such rollbacks. As we shall show later, 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 some failure.

The second new level is compressed system time\(^3\), which
reproduces the relevant properties of system time but is more
efficient to track. In system time, process state consists of
the user state, plus additional information including which
version the process is in, and which incarnation within that
version. In compressed system time, we compress state
to exactly this information. These compressed states are
ordered linearly, as the original system states are.

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., \( A_S \sim_S B_S \) compares two system
state intervals in their process timeline. When it is clear, we
omit subscripts on the partial order time comparison, since
the partial order time model is implied by the subscripts on
the state intervals.

Mapping States Across Levels. Figure 2 shows how the
bits at a process comprise the various levels of state. As this
structure indicates, a natural mapping exists from "lower"
level state to "higher" level state. We define names for these functions: \( S \to C \) maps each system state to a unique
compressed system state; \( C \to F \) maps each compressed
system state to a unique failure state; \( F \to U \) maps each
failure state to a unique user state. We compose these maps
in the obvious way, to obtain \( U \to C, U \to S, \) and \( F \to S \).

Since these maps are not in general bijective (one-to-one),
moving in the other direction is a bit more complicated.
Since user states are the same as failure states, we still have
that each user state maps to a unique failure state. We denote
this mapping by \( U \to F \). However, \( F \to C \) maps each fail-
ure state to at least one (and potentially many) compressed
system states. (The number may be more than one, since a
process may return to a user state after rollback.) Similarly,
\( C \to S \) maps each compressed system state to at least one
(and potentially many) system states. (The number may be
more than one, since a process may go through several sys-

\(^3\)Treated informally in earlier versions of this paper, compressed system
time is necessary for the protocol to have sufficiently small timestamp size;
furthermore, explicit treatment adds clarity.

\[\begin{array}{|c|c|c|c|}
\hline
\text{user data} & \text{version} & \text{incarnation} & \text{extra data} \\
\hline
\text{user state} & & & \\
\hline
\text{failure state} & & & \\
\hline
\text{compressed system state} & & & \\
\hline
\text{system state} & & & \\
\hline
\end{array}\]

\textbf{Figure 2. Different subsets of the bits at a}
\textbf{process form states for the different levels of time.}

Figure 3 illustrates the relationships between these mapping
functions.

2.3. Rollback using Knowable Orphans

Multiple levels of time permits an insight into when a process
can know a user state is an orphan. 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 the following conditions all hold:
when process \( q \) in \( B_S \) knows about \( A_U \); when state \( A_U \)
has been made an orphan by causally following state lost due to
a restart \( R \) at some possibly different process (after either
rollback or failure there); and when process \( q \) in \( B_S \) can
know about \( R \).

As in our earlier work [27], we define a predicate
\texttt{KNOWABLE.ORPHAN}(\( A_U, B_S \)) to capture this property.
The predicate \texttt{KNOWABLE.ORPHAN}(\( A_U, B_S \)) is defined
when \( A_S \Rightarrow B_S \) for some \( A_S \in U \to S(A_U) \). When
defined, \texttt{KNOWABLE.ORPHAN}(\( A_U, B_S \)) is true if and only
if there exists, at some process \( g \), 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 \( D_S \Rightarrow B_S \).

The ability to test for knowable orphans enables asyn-
chronous rollback recovery. Each time a process \( q \) receives
a system-level message, it checks whether its current user
state is a knowable orphan—if so, \( q \) rolls back to its most

\[\begin{array}{|c|c|c|}
\hline
\text{user states} & \text{failure states} & \text{compressed system states} \\
\hline
\text{U} & \text{F} & \text{C} \\
\hline
\end{array}\]

\[\begin{array}{|c|c|c|}
\hline
\text{system states} & \text{compressed system states} & \text{system states} \\
\hline
\text{U} & \text{C} & \text{U} \\
\hline
\end{array}\]

\textbf{Figure 3. Maps take states across levels; dashed}
\textbf{lines indicate one-to-many maps.}
recent state interval that is not 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—if
so, \( q \) rejects the message.

2.4. An Efficient Test for Knowable Orphans

Mapping Ordering Across Levels. Our earlier protocol
[27] worked because we tracked system time and user time,
and were able to compare states across these levels. Our
new protocol works because it suffices to track compressed
system time instead of system time; and to track failure time
instead of user time. To establish these facts, we need to
establish first how orderings map across levels of time.

Both failure precedence and user precedence imply sys-

tem precedence:

Lemma 1  (1) Let \( A_S \) be the minimal inter-

val in \( F \to S(A_F) \) and \( B_S \) be any interval in

\( F \to S(B_F) \). If \( A_F \iff B_F \) then \( A_S \iff B_S \).

(2) Let \( C_S \) be the minimal interval in \( U \to S(C_U) \)

and \( D_S \) be any interval in \( U \to S(D_U) \). If

\( C_U \iff D_U \) then \( C_S \iff D_S \).

System precedence corresponds to compressed system
precedence:

Lemma 2  For any \( A_S, B_S, A_C, B_C \):

\[
A_S \to B_S \implies S \to C(A_S) \to S \to C(B_S)
\]

\[
A_C \to B_C \implies C \to S(A_C) \to C \to S(B_C)
\]

User precedence implies failure precedence; failure
precedence of the images of non-orphan user states implies
user precedence:

Lemma 3  Let \( A_F, B_F \) be the respective images

of \( A_U, B_U \) under \( U \to F \).

(1) If \( A_U \iff B_U \) then \( A_F \iff B_F \).

(2) If \( A_F \iff B_F \) and there exists a

\( C_S \) where Knowable_Orphan(\( A_U, C_S \)) and

Knowable_Orphan(\( B_U, C_S \)) are both false, then

\( A_U \iff B_U \).

Cross-Level Comparison. In our previous protocol, we
defined a way to compare state intervals between the user
and system levels [27]. Here, we extend this definition to
accommodate cross-level comparison through the interme-
diate levels. Let \( A_U, B_U, \) and \( C_S \) be state intervals
at some process, corresponding to unique failure state in-

tervals \( A_F, B_F, C_F \), respectively (under \( U \to F, C \to F, \)

and \( S \to F, \) respectively). We can compare \( A_U \) to \( B_C \)
by comparing \( A_F \) to \( B_F \) in the failure timetree; we denote this
by \( \prec_{UFC} \) and \( \preceq_{UFC} \). Similarly, we can compare \( A_F \) to \( C_S \)
by comparing \( A_F \) to \( C_F \) in the failure timetree; we denote this
by \( \prec_{UFS} \) and \( \preceq_{UFS} \).

Lemma 4  Suppose \( B_C = S \to C(C_S) \). Then

\( B_F = C_F \), so we have \( A_U \prec_{UFC} B_C \) if and only

if \( A_U \prec_{UFS} C_S \).

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

Testing Knowable Orphans via Failure Time. The
heart of our new protocol is a novel method of testing for
knowable orphans.

Our earlier work showed that 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, if all rollbacks have a first cause in some failure,
then comparing user state intervals to system state intervals
via their failure time images exactly captures the knowable
orphan property:

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 \( A_S \iff B_S \),
for some \( A_S \) in \( U \to S(A_U) \). 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 \( X_U \prec_{UFS} Y_S \).

Lemma 2 and Lemma 4 implies that we can substitute
compressed system time for system time.

The fact that our protocol meets the conditions in
Section 1.2 follows from these results.

3. Timestamp Size

Failure Time. The new protocol requires that processes still
be able to sort within user timetrees, but only for state in-
tervals that are not knowable orphans. Lemma 3 estab-
lished that, for purposes of user timestamp vectors on state
intervals that cannot be known to be orphans, tracking failure
time suffices. Failure state intervals can be represented by
a pair of integers, representing the current version and the in-
dex within that version. Sorting within failure time requires
extending this index with a "version start array," showing
the tree structure of the version segments at a process.

Thus, tracking failure vectors takes one entry for each
process. That entry consists of one index of \( O(\log sv) \) bits
for each version at the process; thus the net contribution is
\( O(V \log sv) \) bits.
Compressed System Time. At first glance, the new protocol may also appear to require that processes maintain system timestamp vectors. However, Lemma 2 and Lemma 4 imply that tracking compressed system timestamp vectors suffices. Compressed system state intervals can be represented by a triple of integers, representing the current version, the current incarnation within that version, and the current index within that incarnation. Comparing these triples lexicographically captures the order. Thus, tracking compressed system vectors takes one entry for each process. The version count gives $\log v_i$ bits. The incarnation count is bounded by $s_v$, since each rollback must lose a state that is never restored, thus giving $\log s_v$ bits. The current index is also bounded by $s_v$, giving an additional $\log s_v$ bits. Thus the net contribution is bounded by $O(v \log s_v + \sum_i \log v_i)$, which is bounded by $O(V \log s_v)$, since $\sum_i \log v_i$ is bounded by $O(V)$.

Overall Timestamp Size. Thus, the straightforward implementation of tracking indices requires the total timestamp size to be bounded above by $O(V \log s_v)$ bits.

4. 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 $\Omega(V \log s_v)$ bits. This result establishes the asymptotic optimality of timestamp size in our new protocol.

Since many definitions of asymptotic complexity only discuss functions of one variable, we review the more general definitions [6]. A function $f(u, s)$ is in $\Omega(g(u, s))$ when there exist constants $c, v_0, s_0$ such that for any pair $u, s$ with $u \geq v_0$ and $s \geq s_0$, $0 \leq cg(u, s) \leq f(u, s)$. A function $f(u, s)$ is in $O(g(u, s))$ when there exist constants $c, v_0, s_0$ such that for any pair $u, s$ with $u \geq v_0$ and $s \geq s_0$, $0 \leq f(u, s) \leq cg(u, s)$.

**Theorem 2** There exists a function $g(V, s_v)$ in $\Omega(V \log s_v)$ such that for any rollback protocol satisfying the criteria in Section 1.2 and for any $V, s_v$, there exists a computation in which: some message $M$ must be timestamped with at least $g(V, s_v)$ bits (where $V$ is the number of process versions in the computation perceivable by $M$; and $s_v$ is the maximum number of state intervals in any one version in this computation).

As a consequence of this result, for any rollback protocol satisfying the conditions of Section 1.2, the upper bound on timestamp size is at least $\Omega(V \log s_v)$.

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 Complete Asynchrony. Our results yield completely asynchronous, minimal rollback always—but smaller timestamps are possible by sacrificing optimality performance in unlikely pathological cases. Exploring heuristics such as not sending vector entries the destination process is likely to have, and using unreliable broadcasts to more aggressively distribute some rollback and timestamp information, might yield better results most of the time. Extending our system model to incorporate probabilities of message delay and loss, as well as benchmarking to determine the failure patterns that arise in practice (and how our protocol performs then), would be fruitful areas of further work.

Relaxing Determinism. The lower-bound proof on timestamp size appeared to require that the rollback protocol be deterministic. Thus, 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).

Appendix: Proofs

**Proof of Lemma 1.** First we consider (1). 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 $F_{\rightarrow S}(A_F)$ 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 $F_{\rightarrow S}(C_F)$. Establish the result for $A_F$ and $C_F$, and for $C_F$ and $B_F$.

The proof of (2) can be found in [27]. □

**Proof of Lemma 2.** Two consecutive system states either map to the same compressed system state, or to consecutive compressed system states. □

**Proof of Lemma 3.** First we consider (1). Each branchpoint 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 statement holds for state intervals at any one process since the cross-process links are the same for both time models, the statement holds in general.

We now consider (2). Suppose $A_F \rightarrow B_F$ and such a $C_S$ exists. The failure time path from $A_F$ to $B_F$ de-
composes into a sequence of one or more segments, each contained within a timetree and each separated by a message. If \( A_U \not= B_U \), then at least one of these segments is not a user timetree path. Suppose \( D_F \leq E_F \) at process \( q \) is the first such segment from \( A_P \); let \( D_S, E_U \) be their respective images under \( \text{F-to-U} \). Since \( D_U \not= E_U \), some \( G_U \prec D_U \) must have been restored in a rollback \( H_S \) before \( E_U \) first occurred. By choice of \( q \), \( A_U \not= D_U \). Let \( E_S \) be the minimal interval in \( \text{F-to-S}(E_F) \). \( H_S \not= E_S \), so by Lemma 1 and hypothesis, \( H_S \not= C_S \). Hence \( \text{KNOWABLE-ORPHAN}(A_U, C_S) \). □

Proof of Lemma 4. This follows directly from the definitions. □

Correctness. 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 5 Suppose the only user state intervals rolled back are those that are lost due to failure of their processes, and those that are knowable orphans. Suppose also that some \( A_S \not= B_S \) for some \( A_S \) in \( \text{U-to-S}(A_U) \). Then \( \text{KNOWABLE-ORPHAN}(A_U, B_S) \) is true if and only if there exists a \( C_U \) and \( D_S \) (both at some process \( q \)) such that: (1) \( C_U \not= A_U \); and (2) \( C_U \) is lost due to failure of process \( q \), which then restarts in \( D_S \); and (3) \( D_S \not= B_S \).

Proof. If such a \( C_U, D_S, q \) exist, then the predicate \( \text{KNOWABLE-ORPHAN}(A_U, B_S) \) clearly holds, since restart after failure is a special case of rollback.

Conversely, suppose \( \text{KNOWABLE-ORPHAN}(A_U, B_S) \) holds. By definition, there exists a \( C_U^1 \) and \( D_S^1 \) at process \( q^1 \) such that: \( C_U^1 \not= A_U \); and \( D_S^1 \not= B_S \). 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 \( q^1 \) which then restarts in \( D_S^1 \); or \( \text{KNOWABLE-ORPHAN}(C_U^1, D_S^1) \) is true. If the latter, then we can iterate; since computations are finite, eventually we reach some \( C_U^k, D_S^k, q^k \) such that former rollback causes hold. □

We also establish some relations among lost states and failure time.

Lemma 6 Suppose \( A_U \) is lost due to failure, and \( B_S \) is the restart after that failure. (1) If \( A_S \in \text{U-to-S}(A_U) \) then \( A_S \not= B_S \). (2) If \( C_S \) satisfies \( B_S \not= C_S \) and \( A_F = \text{U-to-F}(A_U) \), then \( A_F \not= C_S \).

Proof. The first statement holds because we can only restart after failure has occurred. The second statement holds because lost states remain lost. □

Proof of Theorem 1. Suppose the predicate \( \text{KNOWABLE-ORPHAN}(A_U, B_S) \) holds. Then Lemma 5 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) \( C_U \not= A_U \); (2) \( C_U \) is lost due to failure, whose restart was \( B_S \); and (3) \( D_S \not= B_S \). Let \( C_F = \text{U-to-F}(C_U) \). Statement (1) implies that \( C_U \not= X_U[r] \), and thus \( C_F \not= U \cdot \cdot \cdot \text{F}(X_U[r]) \). Statement (2) and Lemma 6 imply that \( C_F \not= X_S \) for any \( E_S \) satisfying \( D_S \not= X_S \). Statement (3) implies that \( D_S \not= X_U[r] \). Hence \( C_F \not= X_S \). If \( X_U[r] \not= X_S[r] \), then \( C_F \not= X_S[r] \) since a failure time path exists from \( C_F \) to \( X_U[r] \) in the failure timetree at \( r \). Thus \( X_U[r] \not= X_S[r] \).

Conversely, suppose \( X_U \not= X_S \). Then there exists a process \( r \) with \( X_U[r] \not= X_S[r] \). Let \( C_U = X_U[r] \); let \( C_F = \text{F-to-U}(C_U) \); let \( C_S \) be the minimal state interval in \( \text{U-to-S}(C_U) \). By hypothesis, some \( A_S \in \text{U-to-S}(A_U) \) satisfies \( A_S \not= B_S \). By the definition of a timestamp vector, \( C_U \not= A_U \). By Lemma 1, \( C_S \not= A_S \). Thus \( C_S \not= B_S \). Applying the definition of timestamp vector again yields \( C_S \not= X_S[r] \). Since by hypothesis \( C_F \not= X_S[r] \), a \( D_S \) must exist such that \( C_S \not= D_S \). Then \( D_S \) restarts \( r \) after a failure that lost \( C_U \). Let \( D_S \not= B_S \); then we have \( \text{KNOWABLE-ORPHAN}(A_U, B_S) \). □

Optimality. 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 \( S \).

Proof of Theorem 2. For any \( V, s_V, n \) (where \( V \) and \( s_V \) are each beyond some constant and \( V \geq n \)), we construct a class \( C(V, s_V) \) of computations where \( V \) is the number of versions and \( s_V \) is the maximum number of state intervals in any one version as follows.

Let \( k = n - 3 \). Let us distinguish processes: \( P_S \), the sender; \( P_R \), the receiver; \( P_C \), the clock; and \( P_l \) through \( P_k \), the processes that failed. (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 $sv$ 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 $PS$ and $PR$. For each $i$, at least one message has made it from $P_i$ to $PS$, 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 $Si$ in version $V_i$.

Now, in state interval $S$, $PS$ is preparing to send a message $M$ to $PR$. Define the configuration of $PS$ at this point to consist of the following: for each $i$, the sequence $ΔF$ for $F \in F(M_i)$; and for each $i$, the value $ΔS_i$.

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

(1) At some $P_i$, the $ΔF$ 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 $PR$ 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 $C_1$. Suppose $M'$ is the only message $PR$ has actually received. Should $PR$ then receive $M$, whether $PR$ needs to roll back or not depends on the configuration—which $PR$ cannot distinguish if $PS$ uses the same timestamp on $M$ in both. (Figure 4 illustrates this case.)

(2) The $ΔF$ sequences are identical, but at some $P_i$, the $ΔS_i$ has a different value. Without loss of generality, suppose 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 $Si$ intervals in the two configurations. In either configuration, the computation might continue by having version $j+1$ begin from this interval. Then $Si$ is rolled-back in $C_2$ but not in $C_1$. Suppose $M$ is the only message that $PR$ actually receives, until it later receives a message $M'$ directly from $P_i$, sent in version $j+1$. $PR$ can accept $M'$ in $C_1$ but must first roll back in $C_2$. Since $P_i$ has no information to contribute regarding whether $PS$ was in $C_1$ or $C_2$ when $PS$ sent $M$, $PR$ must get this information from the timestamp on $M$. (Figure 5 illustrates this case.)

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

$$W(V, sv) \in \Omega \left( \frac{(sv)^v}{2} \right)$$

For some $c$ and for $W(V, sv)$ sufficiently large, the number of bits necessary to distinguish membership in a set of $W(V, sv)$ objects is at least $cV \log \frac{sv}{2}$ 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. Consider case (1) above: if $P_i$ rolled back but did not fail, then $PS$ when receiving $M_i$ knows about the failure elsewhere that caused this rollback. Thus $PR$ knows when receiving $M$, and can decide for itself whether the $M'$ it received was an orphan.

**Acknowledgements**

We are grateful to Doug Tygar and Vance Faber for their helpful discussions on this work. We would also like to thank the referees, whose comments helped to improve the clarity of the presentation.
References


