# Secure Distributed Time for Secure Distributed Protocols

Sean W. Smith

September 1994

CMU-CS-94-177

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy*

**Thesis Committee:**
Doug Tygar, chair
Stephen Brookes
David B. Johnson
Maurice Herlihy, Brown University



This research was sponsored in part 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. Support also was sponsored by the Air Force Materiel Command (AFMC) and the Advanced Research Projects Agency (ARPA) under contract number F19628-93-C-0193. In addition, IBM, Motorola, and the NSF/Presidential Young Investigator Award under Grant No. CCR-8858087, TRW, and the U.S. Postal Service gave their support. The author received support from an ONR Graduate Fellowship.

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

## Abstract

This thesis develops a framework for secure distributed time, and uses this framework to build secure protocols for practical problems. In distributed systems, many important problems—such as detecting potential causality, obtaining global states, and recovering from process failure—center on temporal relations more general than the linear order of real time. Systems with asynchronous message passing require a partial order time model, and systems with multiple levels of abstraction require multiple levels of time models. Building clock primitives for these time models facilitates building protocols for these application problems. However, protocols built (even tacitly) on such clocks open themselves to security and privacy risks, since tracking these temporal relations requires sharing and trusting private information.

This thesis addresses these issues of time and security by constructing a distributed time formalism that supports hierarchies of general time models, and then constructing clock primitives—the *Signed Vector Timestamp* protocol and the *Sealed Vector Timestamp* protocol—that provide security and privacy. Framing application problems in terms of this distributed time framework grants insight that often allows us to build protocols more general and flexible than were previously possible. Separating clocks from protocols grants additional flexibility by allowing us to keep their design issues mutually transparent.

This thesis explores three applications of this secure distributed time framework. We transparently add security and privacy to *immediate ordered service* protocols. We build basic *distributed snapshot* protocols and transparently add security, privacy, and increased flexibility. Finally, we use the framework to build a new *optimistic rollback recovery* protocol that substantially improves on previous work by providing full asynchrony while also reducing the worst-case bound for rollbacks after a failure from exponential to one per process; further, developing this protocol within the distributed time framework transparently allows for security and privacy.

# Contents

# List of Figures

# List of Tables

# Acknowledgments

A great many people deserve thanks for helping me reach this point.

First among these is my advisor, Doug Tygar, for his help throughout the research and writing of this thesis, and for his support, advice, and encouragement throughout my entire graduate career.

I also am grateful to the other members of committee—Steve Brookes, Maurice Herlihy, and Dave Johnson—for their support and advice. Dave deserves a special note of thanks, for his insistence that I explain what I mean (hence [Sm93]) and his insistence that I convince him that this mathematics solved real problems (hence [SJT94]). I also am in debt to Carl Sturtivant for his continued advice and collaboration.

I am grateful to Vaughan Pratt and Y. C. Tay for providing many helpful pointers during the initial phase of this work. During the three years I spent on this thesis, I also benefited greatly from the helpful comments of a number of other people, including Bob Baron, Sharon Burks, Jean Camp, Eric Cooper, Gene Cooperman, Catherine Copetas, Marian D'Amico, Nevin Heintze, Allan Heydon, Juan Leon, Francesmary Modugno, Greg Nelson, Michal Prussak, Sandy Ramos Thuel, *Bad Bob* Wheeler, Bennet Yee, and Marco Zagha. Francesmary deserves an extra note of thanks for translating [Ci89].

Most of all, without the prayers and love of my wife Nancy (who deserves a "co-doctorate" for all this), my family, and the Pittsburgh Oratory community, none of this would have been possible. I cannot begin to express gratitude.

# Chapter 1

# Introduction

Many problems in distributed systems center on temporal relations more general than the linear order of real time, or even on a single layer of a more general order. Put simply, *distributed* systems need *distributed* time. Recognizing the central role that distributed time plays creates opportunities:

- Using the appropriate temporal relations can allow deeper insight into the nature of application problems.
- Providing clock primitives for these temporal relations permits construction of clearer, more flexible, and more general protocols for these problems.

However, recognizing the central role that distributed time plays also leads to recognizing a significant problem:

- Protocols built (even implicitly) on distributed time are open to security and privacy risks, since tracking these temporal relations requires sharing private information, and requires trusting the information that is shared. Attacks on the lower-level clocks lead to attacks on the higher-level protocols.

This thesis identifies and resolves these issues by building a framework for secure distributed time, and by using this framework to build secure distributed protocols.

## 1.1. Distributed Time

Our first intuitions organize experience into a linear sequence of discrete events. However, this approach is inappropriate for asynchronous distributed systems, where information is distributed and perception is delayed. Distributed environments require a distributed notion of time, to abstract away not only irrelevant physical detail but also irrelevant temporal and computational detail. By better expressing distributed systems concepts that are difficult to talk about in terms of real time, and by distinguishing what "actually happens" from what physically occurs, a theory of distributed time provides a natural framework for solving problems in distributed environments.

Chapter 2 lays the groundwork for these tasks by reviewing the theory of distributed time we developed for this thesis. This theory improves on previous work on time in distributed systems by supporting temporal relations more general than partial orders, by supporting abstraction through multiple levels of temporal relations, by separating the family of temporal relations an application consults from the particular clock implementations that track them, and by providing a single arena in which to consider these issues for a wide range of applications.

One central claim of this thesis is that distributed time provides a framework for building general protocols for distributed systems application problems. We can first phrase problems in terms of distributed time, and then phrase protocols in terms of distributed time clock primitives. Chapter 2 through Chapter 4 develop this claim by considering several application problems:

- **Potential Causality** Determining whether one event could potentially have influenced another requires sorting events in the *partial order* determined by the asynchronous computation, rather than in the linear order determined by real time. Clocks for partial order time directly support building protocols for problems such as *orphan detection* and *immediate ordered service* that reduce to sorting based on potential causality.
- **Snapshots and Global States** Distribution and asynchrony make it difficult for a process to determine the state of the system at any given instant, since anything that the process can perceive about other processes will be out-of-date. However, phrasing snapshots as *timeslices* from a time model provides a way to use clocks for these models to capture general snapshots and to reason about global states. Phrasing the problem this way allows us to extend a basic protocol by substituting clocks for more general temporal relations, and to address performance concerns by substituting clock implementations.
- **Optimistic Rollback Recovery** The problem of *rollback recovery* arises when a process, due to some type of failure, must roll back events and restart execution (possibly with modified replay). Recovery is *optimistic* when other processes may depend on the lost events at the failed process. Since optimistic recovery requires tracking dependency, many previous approaches use some form of partial order clocks, and thus already dovetail nicely with our work. However, effectively performing this recovery *asynchronously* requires tracking potential knowledge of failures as well. This potential knowledge relation is also expressed by a partial order time model—but a lower-level model than the dependency model. The distributed time framework provides the tools needed to clearly talk about such hierarchies of time—and thus to develop new rollback protocols that improve on previous work.

The distributed time framework introduces orthogonality between clocks and the higher-level protocols that use them. Besides permitting more flexible protocols, this orthogonality has an additional benefit: we can consider clock issues on the clock level, independently of the protocol issues. This approach offers advantages:

- **Orthogonality between Time Models and Protocols** Separating clocks from protocols provides a separation between time models and protocols. We can transparently change the scope of a protocol by substituting clocks for a different time model—for example, a change to the model underlying the snapshot protocols allows us to capture snapshots satisfying the property of having no messages in transit.

- **Unification of Protocols** The distributed time framework unites protocols that individually deal with distributed time issues. This unification directly allows tasks such as taking an offline snapshot after rollback with modified replay. For example, rollback with modified replay creates three distinct versions of the computation: the failed computation, the virtual failure-free computation, and the underlying failure-plus-recovery computation. For each of these computations, scenarios exist where a snapshot would be useful. The distributed time framework directly supports this flexibility.

## 1.2. Security and Privacy

In a distributed system, a process can detect the local passage of real time by examining an independent physical device, such as a quartz clock. However, to track more general temporal relations, a process must collect and share private information. Consequently, dealing with these relations—even implicitly—exposes protocols to security and privacy risks:

- Is the information a process receives correct?

- Is the information a process shares being used for dishonest purposes?

This sharing and trusting creates opportunities for Byzantine (malicious) processes to manipulate the clock protocols, and consequently to manipulate application protocols built on these clock protocols. The orthogonality that distributed time introduces between clocks and protocols thus has the additional significant benefit of creating a single arena in which to examine and resolve these security issues. Installing clocks that protect against security and privacy attacks will transparently provide this protection to higher-level protocols.

The latter part of this thesis examines these security and privacy aspects of distributed time. Chapter 5 begins this examination by considering secure clocks. For example, the standard *timestamp vector* mechanism for partial order time permits numerous attacks. We catalog these attacks, and present two protocols that provide protection: the *Signed Vector Timestamp* protocol and the and *Sealed Vector Timestamp* protocol. We discuss scalability and implementation issues, and outline avenues for further research into secure clocks.

Chapter 6 then uses these techniques to add security and privacy protection to the distributed protocols developed in Chapters 2 through 4.

## 1.3. Overview of Previous Work

Previous work has explored partial order time for distributed systems, both through mathematical models and through protocol construction. The mathematical work provided foundational insights but did not support construction of clocks and protocols; the protocol work did not provide a fully general framework, and consequently did not exploit the full power of the temporal abstraction being performed. Further, the security challenges raised by using clocks for relations more general than linear time were unidentified and unsolved.

**Time** The notion that the linear order of real time may be inappropriate for asynchronous distributed systems emerges in earlier work. Jefferson [Je85] used linear time that departs from the real time order. Lamport [La78] used partial orders to track causal dependency in distributed systems. Pratt [Pr86] argued for the universality of partial order time. Partial order temporal relations have also emerged in the areas of semantics (e.g., [Gr75, Pe80, GaPr87, CCMP89, Win89]) and artificial intelligence (e.g., [Ba93, Bo93, Ts87, YaAl93]). Using partial orders for distributed systems is sometimes called *logical time*; Fidge [Fi91] presents a good survey paper, and very recently Yang and Marsland [YaMa93, YaMa94] have published a collection of some of the principal papers on these issues (and the orthogonal issues of total order clock synchronization).

**Asynchrony** Previous work [BiJo87, PBS89, SES89] has also explored the communication problems introduced by asynchrony: by the fact that the underlying temporal structure is not the linear order of real time. One proposed solution to this problem is to fix a partial order structure as the *causal order* and to *enforce* (via multicasting) that processes perceive a consistent view of this order. (The appropriateness and scalability of this solution has lately generated no small amount of controversy [ChSk93, Bi94, Co94, Re94].) Other approaches to this problem include frameworks to adapt protocols for the asynchronous partial order environment after developing them in simpler environments [Aw85, Mo85, NeTo90] .

**Protocols** Partial order time has also appeared in various forms in distributed systems applications. Some of these areas include distributed debugging [Fi89, Sp89], distributed snapshots [ChLa85, AhKs89, Ma93] and the use of distributed snapshots in debugging [CoMa91, MaNe91, MaSa91, GaWa94]. Partial order time has also been used in deadlock detection [Ma87, TaLo91], immediate ordered service [KeKo89], and rollback recovery [StYe85, Jo89, JoZw90, ElZw92, PeKe93].

**Clocks** Lamport [La78] proposed a clock mechanism that allows processes in an asynchronous distributed system to track a total order consistent with the underlying partial order. Fidge [Fi88] and Mattern [Ma89] formally explored partial order time and concurrently introduced the *vector timestamp* mechanism. (Protocols essentially identical to the vector timestamps mechanism also independently appeared in other work [StYe85, KeKo89]). Other research has explored optimizations to the vector clock protocol [SiKs90], trading decreased accuracy for decreased size [ACGS91], and limits to vector size [CB91] and to clock accuracy [Va93].

**Security** The author [Sm91] identified security problems in the vector clock and Lamport clock mechanisms, and introduced the *Signed Vector Timestamp* protocol. Reiter and Gong [ReGo93] also explored this area and independently discovered this protocol. Amman and Jajordia [AmJa93] explored some issues in securely generating timestamps in the face of confinement levels.

## 1.4. Thesis Contributions

This thesis uses a framework for general temporal relations to advance the state of the art both in distributed protocol design and also in security and privacy for distributed systems.

**Time** To begin with, this thesis provides a fully-developed formalism to talk about clocks for temporal relations that differ from the linear order of real time. This formalism improves on the foundational work by allowing us to talk about *arbitrary* relations (not just partial orders, and not just the single partial order of information flow) and *hierarchies* of abstraction (not just a single level), and allowing us to build *clocks* for these relations and *protocols* based on these clocks.

**Protocols** This thesis then applies this framework to the example problems of distributed snapshots and optimistic rollback recovery. We can define global states in terms of distributed time relations, and build snapshot protocols in terms of clock queries; this approach allows us to substitute clock implementations (e.g., for increased security) and to substitute underlying time models (e.g., to capture specialized properties or to examine alternate virtual computations). The ability to talk about multiple levels of time allows us to build an optimistic rollback recovery protocol that provides fully asynchronous recovery while also reducing the worst case number of rollbacks after a failure from exponential (as in Strom and Yemini's asynchronous protocol [StYe85]) to at most one per process. Further, the single framework of distributed time allows us to consider in one place problems and protocols separately affecting time abstraction .

**Secure Clocks** This research was the first to identify security and privacy problems inherent in partial order time. This thesis presents both the first secure partial order clock protocol, as well as the most secure clock protocol to date. This latter protocol provides security and privacy despite any number of corrupt agents–and extends to partial order temporal structures that differ from the underlying partial order of information flow.

**Secure Protocols** This thesis demonstrates a systematic and transparent way to add security and privacy protection to protocols developed within this secure distributed time framework. Consequently, this work shows how to solve application problems using partial order time—while also defending against espionage and Byzantine attacks. We show how to add this protection to example protocols for providing immediate ordered service, taking distributed snapshots, and performing optimistic rollback recovery.

As computer systems become increasingly distributed and user applications become more attractive to attack, the issues of time and security will only become more important. This thesis lays the groundwork for solving these problems.

(A glossary follows the text of this thesis. This glossary presents four lists: terms, clock primitives, time models, and symbols.)

# Chapter 2

# Distributed Time

This chapter reviews the theory of *distributed time*, a general framework (developed as part of this thesis research) for temporal relations in distributed systems. Section 2.1 presents the motivation behind the theory. Section 2.2 presents tools for representing and abstracting computation. Section 2.3 discusses *timeslices* and *global states*. Section 2.4 specifies and builds some clock primitives. Section 2.5 examines some example applications. Section 2.6 relates this chapter to our earlier, more detailed publication on distributed time [Sm93].

## 2.1. Overview

**Beyond Real Time** Normally we think of a computation as a sequence of states and events ordered by real time. However, even this natural view performs abstraction from full physical detail to discrete events. Describing asynchronous distributed computation requires extending this abstraction to time: if two events occur without knowledge of each other, then their real time sequence does not matter and also may not be observable [La78, Pr86]. Consequently, many application problems are simplified by thinking of time as the partial order determined by potential information flow. (This relation is sometimes referred to as the "Lamport order," after [La78], and also the "causal order," since it expresses potential causality.)

**Beyond Partial Order Time** Pioneering work in partial order time [Fi88, Ma89] leaves us thinking about computation as a temporal relation on a set of objects—except each object actually represents the activity in a region of space-time, and the relation does not follow directly from real time order on these regions. Many application issues suggest that we should continue removing irrelevant temporal and computational detail—that we should continue the process of abstraction:

- Using multiple levels of partial order time clarifies distributed computations that fail and recover.

- Omitting the details of recovery facilitates describing the failure-free virtual computation.

- The temporal relations of interest may not necessarily follow from the information flow partial order. One example is the partial order describing the virtual computation after recovery; another are the zigzag paths in Xu and Netzer's recent work [XuNe93] in checkpoint coordination.

- The temporal relations of interest may not necessarily be a mathematical *order*. (As Section 2.2.3 discusses, an order is a relation both transitive and antisymmetric.) For example, relaxing the transitivity requirement clarifies discussion of confinement barriers and individual steps in information flow. Relaxing the acyclic requirement allows a natural way to unite sets of events into atomic units: cycles.

Extending partial order time to a general framework for temporal abstraction provides the tools to talk about these scenarios. Put simply, describing a *distributed* computation requires a theory of *distributed time*.

**Distributed Time for Distributed Protocols** A theory of *distributed time* has practical motivations and uses. Consider the computation performed by asynchronous distributed systems, with processes that possess no common clock, that fail and restart, and that frequently may be disconnected or even powered down. Many application problems that arise in these systems reduce to asking questions about temporal relations other than the natural real time sequence. Thinking in terms of these alternative temporal relations clarifies these problems; providing clocks for these relations simplifies protocol design. Indeed, building protocols for these problems requires confronting these clock issues in one form or another. However, exploiting the power of alternative temporal relations requires understanding the underlying framework. The remainder of this chapter develops these formal mechanisms of distributed time.

This research improves on earlier work by providing a single, general theory of distributed time suitable for a wide range of applications. By supporting temporal relations more general than partial orders and by supporting hierarchies of temporal abstraction, this theory can express the computational abstraction appropriate for families of application problems. By providing a general approach to distributed time, this theory allows us to unify in a single framework protocols that separately consult and affect time, and to consider once the clock issues central to each separate protocol. By introducing orthogonality between temporal relations and the clocks that track them, this theory allows us to consider (and alter) clock implementations without changing higher-level protocols.

Considering these goals raises some critical issues:

- We want to represent a computation as some abstract set of "things that happened," with a relation indicating the temporal order in which these things happened.

- The components in these abstractions themselves represent various parts of a literal description of what physically happened.

- These abstractions should permit temporal relations more general than that of linear time.
- We need to distinguish between the way we obtain the abstract representations and the representations themselves, since we may have multiple routes to the same representation.
- We want to be able to apply abstractions to abstractions.

We conclude that a general theory of *distributed time* should contain three components:

- a standard format for these abstract representations (so we can talk about computations);
- a way to specify *time models*: representational transformations on these objects (so we can abstract from one representation to another); and
- a way to translate some level of physical description into this format (so our chains of abstraction have some footing in reality).

Distributed *time models* provide several advantages:

- If the desired physical description is unavailable, our time model should express the best observable approximation.
- If the complete physical description obscures key concepts, then our time model should abstract to a more appropriate description.
- If the processes collectively pretend that the “current” computation differs from the one a complete physical description would record, then our time model should express this abstraction.

## 2.2. Description and Abstraction

### 2.2.1. Systems

In the theory of distributed time, we model the system as a collection of process automata that send and receive messages asynchronously (and unreliably). Each process has a send queue and a receive queue, not necessarily FIFO. When a process sends a message, it appends the message to its send queue. At some undetermined time later, the network removes the message from the send queue. Eventually the message may appear in the receive queue of the destination process, which may then receive the message. (Thus, each message may be received at most once.) We assume that each message is sufficiently distinct (perhaps using identifier tags) so that we can unambiguously identify the *send* corresponding to a given *receive*.

Each process also has a state transition rule $\delta$. The $\delta$ rule may differ at each process and may even be nondeterministic (specifying a set of possible new states at each transition). We constrain the $\delta$ rule to force processes to behave reasonably with respect to messages. That is, during a transition, a process may do one of three things:

- try to receive a message from its input queue,
- send a message, or
- perform internal computation.

Sending or receiving a message changes the internal state at a process. A process may receive messages by periodically polling its queue, or by continually looping on a poll (thereby blocking) until a message is received. We also permit interrupt-driven receive events: a process may attempt to perform an internal computation, with the caveat that if a message is present in the input queue, the process will receive that instead.

A process operates in real time and changes state at indeterminate intervals. We model this behavior by saying that each process has a black box that generates *ticks* nondeterministically (but generating only finitely many in any finite period of real time). When receiving a tick, a process transforms state instantaneously according to its state transition rule. If a tick arrives at real time $u$, the old state persists for times $t$ satisfying $t \leq u$; the new state exists for $t > u$.

A *system computation* is what happens when the processes are set to their initial conditions and fed with nondeterministic ticks.

### 2.2.2. Traces

Probably the most practical ground-level view of computation is a linear *trace*. A trace is an exhaustive physical description analogous to a movie reel, each frame stamped with a real time value and recording the states of each process. We require that the cameraman obtaining the trace to be lucky but not necessarily regular: the interval of time between frames need not be constant, but at least one frame must be taken between any two consecutive ticks (or message arrivals/departures) in the system. Table I shows an example.

So that traces have non-zero duration, we require that they have at least two frames.

In some sense, a trace is a hypothetical construct, since obtaining one requires access to the complete physical state of each process at any instant in real time. Nevertheless, traces serve as a starting point, describing the physical action in a computation.

| time | $t = 0.0$ | $t = 0.8$ | $t = 1.5$ | $t = 2.2$ | $t = 3.0$ | $t = 4.2$ | $t = 4.8$ |
|---|---|---|---|---|---|---|---|
| $p$: state | *initial* | 17 | 17 | 17 | 23 | 23 | 23 |
| $p$: send queue | $\emptyset$ | $\{M\}$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |
| $p$: receive queue | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |
| $q$: state | *initial* | *initial* | *initial* | 32 | 32 | 32 | 12 |
| $q$: send queue | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |
| $q$: receive queue | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\{M\}$ | $\emptyset$ |

**Table I** In this simple example of a system trace, process $p$ sends a message $M$ to process $q$.

### 2.2.3. Computation Graphs

In order to express computation as a temporal relation on some set of abstract, discrete objects, distributed time uses a *computation graph* format where nodes represent the objects, and directed edges represent precedence. This construction is similar to ordered multisets, but allows us to express relations more general than orders, and to use language already in the common parlance of systems scientists.

**Notation** An *atom* of a graph is a node or an edge. A *minimal* node in a computation graph is one that has no node preceding it: a node with in-degree zero. Similarly, a *maximal* node in a computation graph is one that has no node following it. We usually use lower-case Greek letters to refer to computation graphs, upper-case Roman from the front of the alphabet to refer to nodes, and upper-case Roman from the end of the alphabet to refer to sets of nodes.

**Precedence and Concurrence** For nodes $A$ and $B$ in a computation graph, we write $A \longrightarrow B$ to indicate that node $A$ precedes node $B$, and $A \not\longleftrightarrow B$ to indicate that $A$ and $B$ are incomparable: neither precedes the other. We say that incomparable nodes are *concurrent*.

We write $A \Longrightarrow B$ to indicate that either $A \longrightarrow B$ or $A = B$.

The precedence relation specified by a computation graph is an *order* when it satisfies two conditions:

- The relation is *antisymmetric* (or *acyclic*): for any $A, B$, if $A \longrightarrow B$ and $B \longrightarrow A$ then $A = B$.
- The relation is *transitive*: for any $A, B, C$, if $A \longrightarrow B$ and $B \longrightarrow C$ then $A \longrightarrow C$.

We use graphs to permit temporal relations more general than orders. In particular, defining precedence by edges, rather than paths, permits nontransitive relations.

**Prefixes and Past-Closure** Suppose $\alpha'$ is a subgraph of computation graph $\alpha$. We say that $\alpha'$ is a a *prefix* of $\alpha$ when $\alpha'$ is connected and contains all minimal nodes of $\alpha$. We say that $\alpha'$ is *past-closed* when common nodes have the same history in $\alpha$ and $\alpha'$. (That is, for any node $B$ in $\alpha'$, if node $A$ precedes $B$ in $\alpha$, then $A$ exists and precedes $B$ in $\alpha'$.) The *past-closure* of a subgraph $\alpha'$ is the the intersection of all past-closed subgraphs of $\alpha$ that contain $\alpha'$.

**Ground-level Computation Graphs** Directly translating traces into computation graph format yields *ground-level* computation graphs. Ground-level graphs have six types of nodes: a *photo* node, representing the state of a process captured in a frame of the trace, and nodes representing each way that process state might transform:

- when a process *sends* a message,
- when a process *receives* a message,
- when a process *computes* something internally (i.e., a state transition not involving input or output),
- when a message *departs* from the send queue at a process, or
- when a message *arrives* at a receive queue at a process.

We transform a trace into its ground-level graph by constructing a photo node for each process in each frame of the trace. Should two consecutive photos of a process indicate a state change, we insert the appropriate transition node. Directed edges connect the consecutive nodes at each process.

Figure 2.1 shows an example of this construction.

**Representation** Each atom in a ground-level computation graph represents some part of the computational space-time expressed by the trace. The space coordinate of the region an atom represents is determined by the process: the process $p$ atoms represent activity at process $p$. The time span is determined by the following rules:

- Each photo node represents the instant in time of that frame.
- Each transition node represents the unknown instant in time the transition occurred.
- Each directed edge between two nodes represents the open interval between the instants represented by the endpoints.

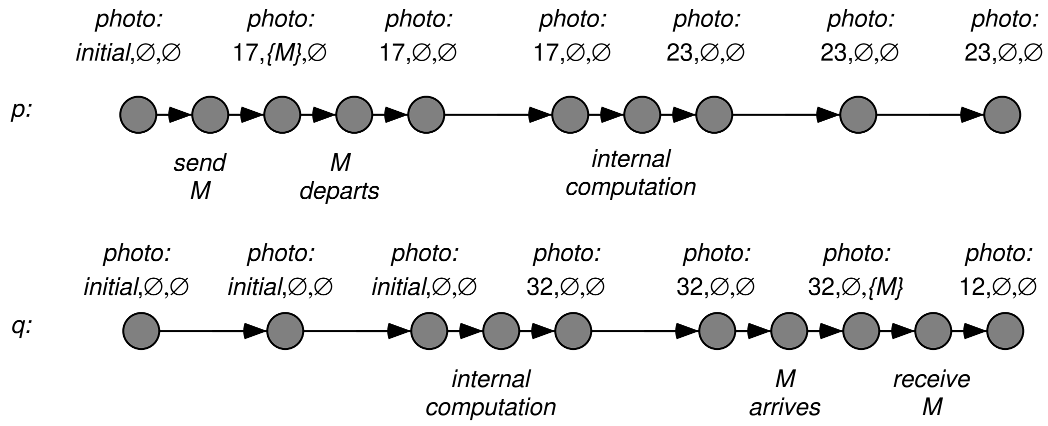


**Figure 2.1** A ground-level computation graph is the lowest level abstraction of a computation. This sketch shows the ground-level graph for the computation whose trace appears in Table I.

Figure 2.2 shows an example of this representation.

**Event vs. State** Considering how to build computation graphs brings up an important question [Pr92]: should the fundamental object (the nodes) represent events or states? Should the main unit of description be the dynamic "thing that happens" at a process, or the static "interval of holding a specified bit-pattern?" Ground-level graphs admit both types of objects: photo nodes describe process state, while the other nodes describe inferred (rather than directly observed) state transitions.

Each approach can be useful, and the distributed time formalism supports both.

### 2.2.4. Time Models

**Representative Transformations** A ground-level graph provides too much detail. A *time model* is a mechanism to generate more abstract descriptions. Formally, time models are representative transformations on computation graphs. This description highlights the two key properties:

- **Transformation** A time model $\mathbf{M}$ is a partial function on computation graphs. Applying $\mathbf{M}$ to a graph $\alpha$ (for which $\mathbf{M}$ is defined) produces a new, more abstract graph $\mathbf{M}(\alpha)$.
- **Representation** If model $\mathbf{M}$ is defined on graph $\alpha$, each atom of $\mathbf{M}(\alpha)$ may represent atoms in the original graph $\alpha$. However, this representation may be a Chicago-style democracy: some atoms of $\mathbf{M}(\alpha)$ may represent no one, and some atoms of $\alpha$ may have multiple representatives. We formalize this arrangement by saying that the application of $\mathbf{M}$ to $\alpha$ induces a *representation map* from the atoms of $\mathbf{M}(\alpha)$ to sets of atoms of $\alpha$. This map, which

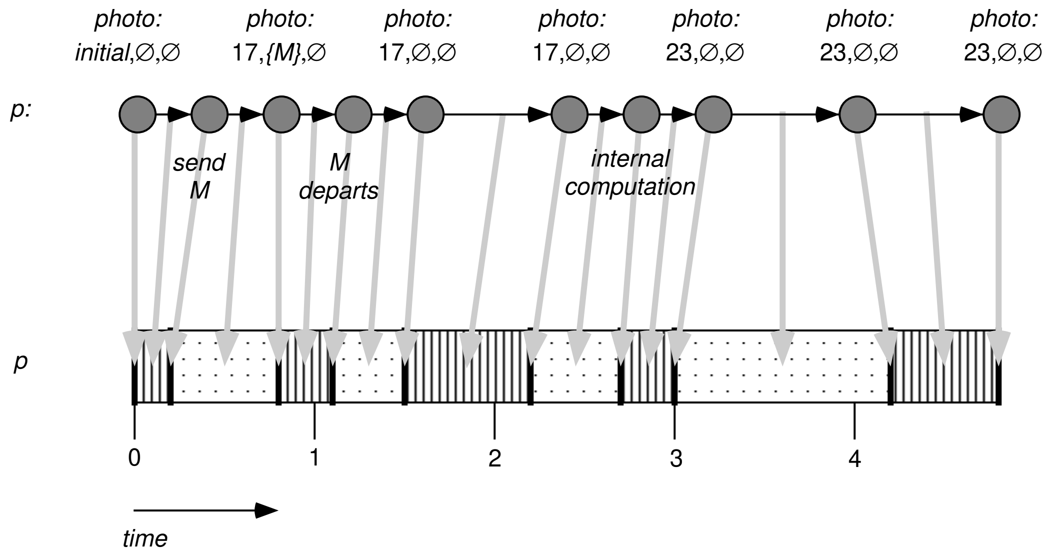


**Figure 2.2** Each atom in a ground-level computation graph represents part of the space-time region in which the computation occurs. This diagram shows how the process $p$ part of the ground-level graph from Figure 2.1 partitions the time experience of process $p$ in the computation from Table I.

we denote as $\langle\, \mathbf{M},\ \alpha\, \rangle$, takes each atom in the new graph to the set of atoms it represents in the original graph.

Figure 2.3 shows an example of this relationship.

**Composition and Hierarchies** The functional nature of time models allows us to compose them. This allows us to place models—and computation graphs—into hierarchies. For example, $\mathbf{M}_1$ might take a set of ground-level computation graphs $\mathcal{G}_0$ to a set of more abstract graphs $\mathcal{G}_1$. This abstraction might lose information, in the sense that $\mathbf{M}_1$ might take several graphs in $\mathcal{G}_0$ to a single graph in $\mathcal{G}_1$. Model $\mathbf{M}_2$ may abstract from $\mathcal{G}_1$ to $\mathcal{G}_2$; the composition model $\mathbf{M}_2 \circ \mathbf{M}_1$ takes the ground-level graphs directly to $\mathcal{G}_2$.

The representation map for composed models follows naturally. For a computation graph $\alpha$, let $\beta = \mathbf{M}_1(\alpha)$ and $\gamma = \mathbf{M}_2(\beta) = (\mathbf{M}_2 \circ \mathbf{M}_1)(\alpha)$. We find out what an atom $A$ in $\gamma$ represents under the representation map for $\mathbf{M}_2 \circ \mathbf{M}_1$ by the following steps:

1. We apply the map for $\mathbf{M}_2$ to find out what $A$ represents in $\beta$.

2. We then apply the map for $\mathbf{M}_1$ to find out what each atom in this set represents in $\alpha$.

3. We take the union of the results of this second round.

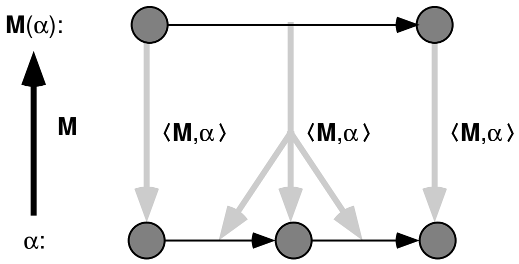


**Figure 2.3** Time model $\mathbf{M}$ transforms computation graph $\alpha$ to computation graph $\mathbf{M}(\alpha)$. The representation map $\langle\, \mathbf{M},\, \alpha \,\rangle$ takes each atom of $\mathbf{M}(\alpha)$ back to the set of atoms in $\alpha$ it represents. The bold arrow indicates the action of $\mathbf{M}$; the gray arrows indicate the action of $\langle\, \mathbf{M},\, \alpha \,\rangle$.

We state this formally in the equation:

$$\langle\, \mathbf{M}_2 \circ \mathbf{M}_1,\, \alpha \,\rangle(A) \;\equiv\; \bigcup_{B \in \langle\, \mathbf{M}_2,\, \mathbf{M}_1(\alpha) \,\rangle(A)} \langle\, \mathbf{M}_1,\, \alpha \,\rangle(B)$$

Figure 2.4 illustrates this construction.

**Refinement** Suppose two models $\mathbf{M}_1, \mathbf{M}_2$ have the property that for all computation graphs $\alpha$ and $\alpha'$, $\mathbf{M}_1(\alpha) = \mathbf{M}_1(\alpha')$ implies $\mathbf{M}_2(\alpha) = \mathbf{M}_2(\alpha')$. Knowing the $\mathbf{M}_1$ image of a graph is sufficient to determine the $\mathbf{M}_2$ image. We say that $\mathbf{M}_1$ *refines* to $\mathbf{M}_2$, and write $\mathbf{M}_1 \triangleright \mathbf{M}_2$.

When $\mathbf{M}_1 \triangleright \mathbf{M}_2$, model $\mathbf{M}_2$ provides a more abstract view of the underlying computation, but in a way that is still well-defined in terms of the view $\mathbf{M}_1$ provides.

**Abstraction Hierarchies** Refinement is clearly transitive. This fact allows us to put models into *abstraction hierarchies*: chains of models that successively refine to each other.

### 2.2.5. Properties of Time Models

We now define several time model properties that we will use in this thesis. Temporal relations determine two of these properties:

- A model is *transitively bounded* when its transitive closure has a unique maximum node and a unique minimum node.
- A model is *acyclic* when its transitive closure has no cycles. (A node in a graph is *acyclic* when it does not precede itself.)

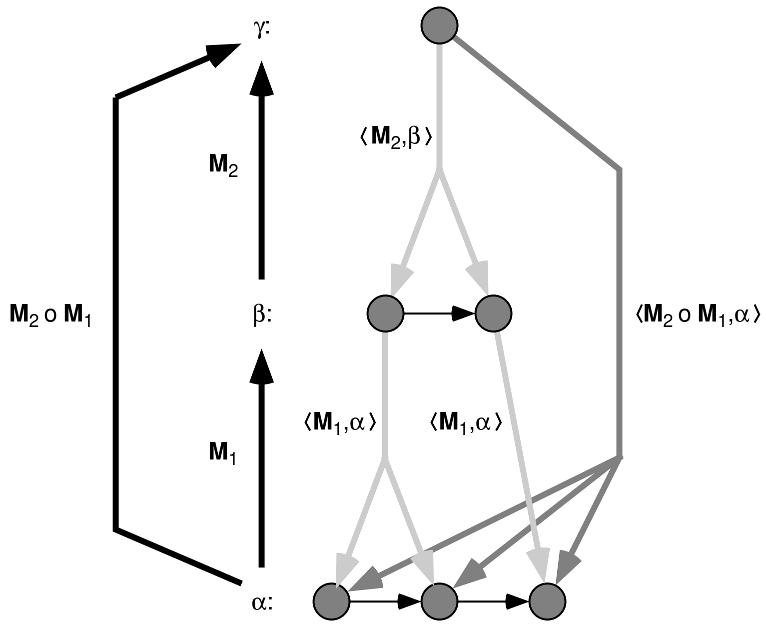


**Figure 2.4** To obtain the image of graph $\alpha$ under the composition $\mathbf{M}_2 \circ \mathbf{M}_1$, we first obtain $\beta = \mathbf{M}_1(\alpha)$, and then obtain $\gamma = \mathbf{M}_2(\beta)$. To apply the representation map $\langle\, \mathbf{M}_2 \circ \mathbf{M}_1,\, \alpha\, \rangle$ to an atom in $\gamma$, we first apply $\langle\, \mathbf{M}_2,\, \beta\, \rangle(A)$ to that atom. We then apply $\langle\, \mathbf{M}_1,\, \alpha\, \rangle$ to each atom in the resulting subset of $\beta$, and take the union of the result. In this diagram, solid arrows indicate the action of the time models; gray arrows indicate the action of the representation maps.

The remaining properties involve relating the graphs a model produces to the computation underlying this graph (through the trace and the ground-level computation graph.) First, moving from an $\mathbf{M}$ graph back to the *static* underlying computation allows us to define two properties:

- We say a time model $\mathbf{M}$ is *flow-supported* when transitive precedence implies information flow. Every precedence path is *supported* by potential information flow. That is, suppose that $A$ and $B$ are two nodes in a graph $\beta$ produced by $\mathbf{M}$, and that ground-level graph $\alpha$ satisfies $\mathbf{M}(\alpha) = \beta$. If $A \longrightarrow B$ in $\beta$, then an information flow path exists from the space-time region $A$ represents (through $\alpha$) to the space-time region that $B$ represents (through $\alpha$).
- We say a time model $\mathbf{M}$ is *flow-virtual* when information flow does not necessarily imply precedence. Such a model may express the information flow in a simulated *virtual* computation.

We also need to move from an $\mathbf{M}$ graph back to the *dynamic* underlying computation. A trace of a computation corresponds to a ground-level graph. A computation in progress induces a sequence of increasing finite traces; hence we can think of an unfolding computation as the sequence of ground-level graphs

$$[\alpha_i] \quad = \quad \alpha_0, \alpha_1, ..., \alpha_i, ...$$

corresponding to this sequence of traces. For a graph $\beta$ produced by a time model $\mathbf{M}$, we can define the set $\mathcal{S}_{\mathbf{M},\beta}$ of ground-level graph sequences corresponding to unfolding computations that, at some point, generate $\beta$ through $\mathbf{M}$.

$$\mathcal{S}_{\mathbf{M},\beta} \quad = \quad \{[\alpha_i] \;\;:\;\; \exists\, k\; \mathbf{M}(\alpha_k) = \beta\}$$

We use this set $\mathcal{S}_{\mathbf{M},\beta}$ to define several types of *monotonicity*:

- A model $\mathbf{M}$ is *node-monotonic* when, for any graph $\beta$ it produces, each node in $\beta$ never vanishes once it exists.
$$\forall \text{ nodes } A \text{ in } \beta \;\; \forall\, [\alpha_i] \in \mathcal{S}_{\mathbf{M},\beta} \;\; \exists\, k \;\; \forall\, j \;\; : \;\; A \in \mathbf{M}(\alpha_j) \iff (j \geq k)$$
- A model $\mathbf{M}$ is *weakly edge-monotonic* when, for any graph $\beta$ it produces, each edge in $\beta$ never vanishes once it exists.
$$\forall \text{ edges } E \text{ in } \beta \;\; \forall\, [\alpha_i] \in \mathcal{S}_{\mathbf{M},\beta} \;\; \exists\, k \;\; \forall\, j \;\; : \;\; E \in \mathbf{M}(\alpha_j) \iff (j \geq k)$$
- A model $\mathbf{M}$ is *strongly edge-monotonic* when an edge exists between two nodes in $\beta$ only if it always exists in all graphs containing those two nodes.
$$\begin{aligned} &\forall \text{ nodes } A, B \text{ in } \beta \;\; \forall\, [\alpha_i] \in \mathcal{S}_{\mathbf{M},\beta} \;\; \forall\, j \quad : \\ &\qquad A, B \in \mathbf{M}(\alpha_j) \quad \Longrightarrow \quad ((A \longrightarrow B \text{ in } \mathbf{M}(\alpha_j)) \iff (A \longrightarrow B \text{ in } \beta)) \end{aligned}$$
- A model is *weakly monotonic* when it is node-monotonic and weakly edge-monotonic.
- A model is *strongly monotonic* when it is node-monotonic and strongly edge-monotonic.

### 2.2.6. Parallel Pairs

Frequently the single perspective from a single time model is not sufficient. A distributed system provides two simple examples:

- We may want to distinguish between "node $B$ happened *immediately* after node $A$" and "node $B$ happened after node $A$."
- We may want to distinguish between the ordering of nodes at an individual process (the *local* computation) and the system-wide ordering of nodes at all processes (the *global* computation).

Distributed time allows such multiple perspectives.

Composition allows us to distinguish between the basic steps in a computation—e.g., events $A, B, C$ happened in sequence—and the general ordering. We simply build our model $\mathbf{M}$ to draw edges for the basic steps and build a standard model TRANS to take the transitive closure of a graph. Then we can talk about basic steps using $\mathbf{M}$, and full transitive precedence using $\overline{\mathbf{M}} = \text{TRANS} \circ \mathbf{M}$.

Events/states in a distributed system can exist and be ordered on two levels: locally, in the timelines of their processes, and globally, in terms of the entire system. A graph describing the local computations clearly relates to a graph describing the global computation: join the local graphs "in parallel," merge some events, and possibly add some edges.

A *parallel pair* is such a pair of models $(\mathbf{M}, \mathbf{M}')$. Both models act on ground-level graphs. Model $\mathbf{M}$ produces a graph describing the global computation; model $\mathbf{M}'$ produces a graph composed of disjoint straightline graphs, each describing a local process timeline. The models in a parallel pair must satisfy one additional rule: minimal events at processes must correspond to minimal events in the global graph, and similarly for maximal events. When $\mathbf{M}'$ is the local model in a parallel pair, we denote its process $p$ component by $\pi_p\, \mathbf{M}'$.

The two models in a parallel pair must closely correlate. This closeness allows us to define a time model taking graphs produced by the local model to graphs produced by the global one. We call this model the *factoring model* $\mathbf{M}/\mathbf{M}'$. The factoring model satisfies the equation:

$$\mathbf{M} \;=\; (\mathbf{M}/\mathbf{M}') \circ \mathbf{M}'$$

Figure 2.5 illustrates the four perspectives that a parallel pair provides.

### 2.2.7. Properties of Parallel Pairs

Time model properties directly lead to several parallel pair properties:

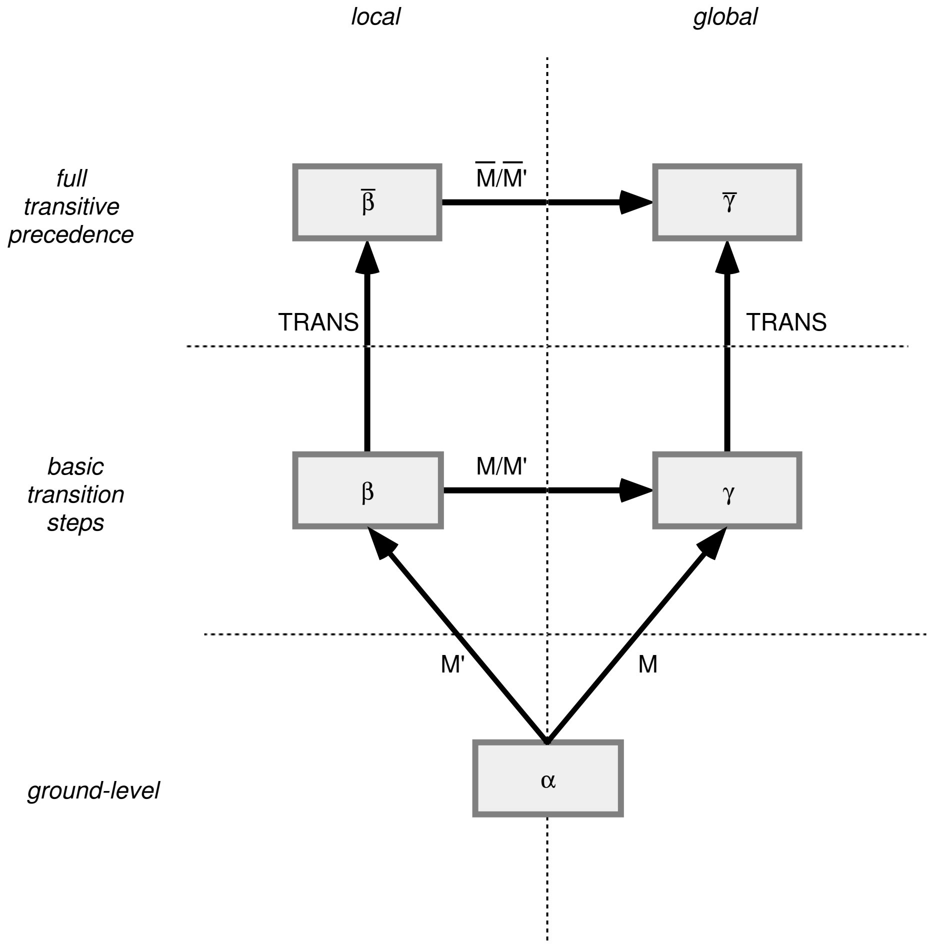


**Figure 2.5** A time model generates an abstract view of a computation. A parallel pair generates four views, according to the two independent choices: whether we use the process timelines or the overall system graph, and whether we consider basic transitions or transitive precedence. Here, the parallel pair $(\mathrm{M}, \mathrm{M}')$ acts on ground-level $\alpha$, the computation graph corresponding to system trace $T$. The local model $\mathrm{M}'$ takes $\alpha$ to $\beta = \mathrm{M}'(\alpha)$, the collection of process timelines. The global model $\mathrm{M}$ takes $\alpha$ to $\gamma = \mathrm{M}(\alpha)$, the overall system description. We can take the transitive closure of either of these graphs—and of either of these models. The graph $\overline{\beta} = \overline{\mathrm{M}'}(\alpha)$ expresses the full transitive relation induced by the basic steps in $\beta$; the graph $\overline{\gamma} = \overline{\mathrm{M}}(\alpha)$ expresses the full transitive relation induced by the basic steps in $\gamma$. The factoring model $\mathrm{M}/\mathrm{M}'$ takes the $\mathrm{M}'$ image to the $\mathrm{M}$ image; the factoring model $\overline{\mathrm{M}}/\overline{\mathrm{M}'}$ takes the $\overline{\mathrm{M}'}$ image to the $\overline{\mathrm{M}}$ image.

- A parallel pair is *transitively bounded* when its global model is transitively bounded.
- A parallel pair is *acyclic* when its global model is acyclic.
- A parallel pair is *weakly monotonic* when the transitive closure of each model is weakly monotonic.
- A parallel pair is *strongly monotonic* when the transitive closure of each model is strongly monotonic.
- A parallel pair is *flow-supported* when each model is flow-supported.
- A parallel pair is *flow-virtual* when the transitive closure of each model is flow-virtual.

The pairing of time models leads to other properties:

- Each atom at a process affords some view of the activity at the other processes. Two such atoms at a process are *externally equivalent* when they afford the same view: either both are cyclic or both are acyclic, and both have the same transitive global relation to each node at all other processes. A graph $\alpha$ from the global model in a parallel pair is *view-complete* when any edge at any process has, in $\overline{\alpha}$, an externally equivalent node at that process. That is, if any basic step at a process affords some external view in the transitive global graph, a node exists at that process giving the same view. A parallel pair $(\mathbf{M}, \mathbf{M}')$ is *view-complete* when all graphs produced by the global model $\mathbf{M}$ are view-complete.
- A *consistent* parallel pair is one that is view-complete and transitively bounded.
- In an *independent* parallel pair, each non-extremal node in the global model represents a unique node in the process model.

**Types of Parallel Pairs** This thesis will focus primarily on parallel pairs of four types:

- *Type 1*: those that are consistent;
- *Type 2*: those that are consistent and independent;
- *Type 3*: those that are strongly monotonic, consistent, and independent;
- *Type 4*: those that are flow-supported, strongly monotonic, consistent, and independent.

For $n \in \{1, 2, 3\}$, any Type $n$ pair is a Type $n-1$ pair, but some Type $n-1$ pairs may not necessarily be Type $n$ pairs.

We will also consider independently when a parallel pair is *flow-virtual*.

The Type 1 and Type 2 conditions describe the internal structure of time models. The Type 3 and Type 4 conditions describe properties useful for specifying clocks for these models. The flow-virtual condition will be useful in considering security properties of these clocks.

When a parallel pair $(\mathbf{M}, \mathbf{M}')$ is Type 2, we will informally identify the nodes in the $\mathbf{M}$ graph with their mates in the $\mathbf{M}'$ graph.

### 2.2.8. Nonlinear Pairs

A *nonlinear pair* is a pair of models $(\mathbf{M}, \mathbf{M}')$ that meets the definition of parallel pair, except for the requirement that $\mathbf{M}'$ produce straightline graphs. The definitions of Section 2.2.6 and Section 2.2.7 apply to nonlinear pairs as well.

### 2.2.9. Examples

Thinking about time in asynchronous distributed computations as a partial order, determined by the asynchrony and distribution, holds a number of advantages over thinking of time as a total order, determined by real time. This section develops time models to transform ground-level computation graphs to graphs depicting their natural partial order time descriptions.

Partial order time abstracts away irrelevant temporal detail. As we shall see in subsequent chapters, frequently we need to abstract away irrelevant computational detail as well—deriving temporal relations more general than the standard partial order, as well as deriving instances of the standard partial order that do not arise directly from the actual computation.

This section proceeds by removing the irrelevant detail of the network activity, and then building a partial order time model that will be standard for this thesis.

**Abstracting Away Network Activity** The goal of partial order time is to express the temporal ordering perceived by the processes themselves. The first step toward building such models consists of abstracting away details imperceivable by the processes: the state and transformations of their queues. Thus we begin by defining the NET_ABSTRACT time model which acts on ground-level computation graphs.

The NET_ABSTRACT model abstracts away network activity as follows. In a ground-level graph, the photo nodes record both the automata state and the queue state at the process. For each photo node, we retain only the recorded automata state. We delete the nodes marking arrive transitions and depart transitions. For each process, the nodes in the NET_ABSTRACT image correspond to a subsequence of the nodes in the ground-level graph; we draw edges connecting the nodes in this sequential order.

Basically, the NET_ABSTRACT image of a graph consists of a copy of the original graph, with the photo nodes relabeled and the irrelevant transition nodes deleted. The representation map follows this description. Let $\alpha$ be a ground-level graph, and $\beta$ = NET_ABSTRACT($\alpha$). The representation map $\langle$ NET_ABSTRACT, $\alpha \rangle$ takes each atom in $\beta$ to its original image in $\alpha$, with one exception—deleted transition nodes. Suppose node $A$ in $\alpha$ is a transition node that NET_ABSTRACT deletes. Let $E_1$ and $E_2$ be the edges incident to $\alpha$. Let $E$ be the edge in $\beta$ where $A$ would have been. Then

$$\langle \text{NET\_ABSTRACT}, \alpha \rangle (E) \quad = \quad \{E_1, A, E_2\}$$

Figure 2.6 shows how the NET_ABSTRACT model applies to the sample ground-level computation graph from Figure 2.1. Figure 2.7 clarifies the representation map.

**Timelines** The TIMELINES model organizes individual process activity into linear timelines. We obtain the TIMELINES image of a ground-level computation graph $\alpha$ in several steps:

- We apply NET_ABSTRACT to $\alpha$. Let $\beta$ be the resulting graph.
- At each process, we create a $\perp$ node for the first photo node in $\beta$, and a $\top$ node for the last photo node in $\beta$.
- We copy each send and receive node in $\beta$.
- Removing the send nodes, receive nodes, and extremal photo nodes from $\beta$ would leave us with a collection of maximal connected sequences of atoms, each occurring at only one process. For each such sequence, we create a *state* node reflecting the process state. (This state is well-defined: each sequence will have at least one photo node, except possibly the

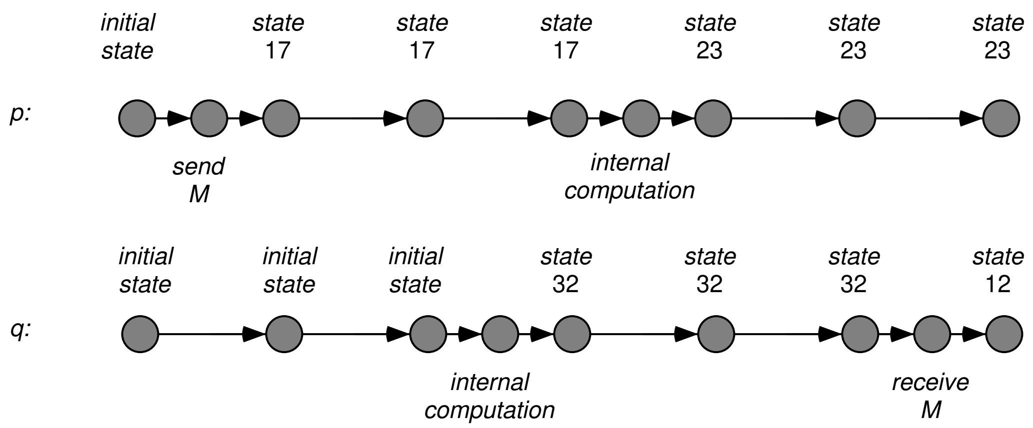


**Figure 2.6** The NET_ABSTRACT model removes irrelevant network detail. This computation graph shows the result of applying NET_ABSTRACT to the graph of Figure 2.1.

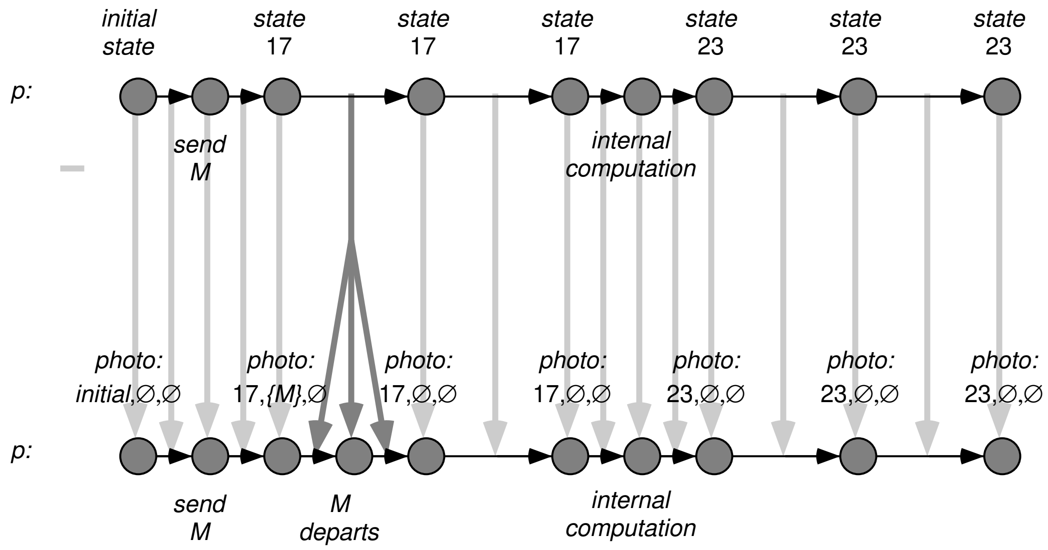


**Figure 2.7** Representation under the NET_ABSTRACT model is practically the identity. This diagram shows how atoms in the process $p$ part of the NET_ABSTRACT graph of Figure 2.6 represent atoms in the process $p$ part of the ground-level graph from Figure 2.1. The darker gray arrows leading to the deleted *depart* node are the only significant change.

first and last sequences at a process. These extremal sequences will pick up their values from $\perp$ and $\top$.)

- We connect consecutive nodes at each process with directed edges.

Figure 2.8 sketches this construction.

Representation follows from this construction. Suppose $\alpha$ is a ground-level graph, and we apply the time models to obtain the graphs:

$$\begin{aligned}\beta &= \text{NET\_ABSTRACT}(\alpha)\\ \gamma &= \text{TIMELINES}(\alpha)\end{aligned}$$

Each node $A$ in $\gamma$ replaces a sequence $S$ of atoms in $\beta$. Node $A$ represents in $\alpha$ the union of what the elements of $S$ represent.

$$\langle\, \text{TIMELINES},\ \alpha\,\rangle(A) \;=\; \bigcup_{B\in S} \langle\, \text{NET\_ABSTRACT},\ \alpha\,\rangle(B)$$

Figure 2.9 sketches this relation.

For process $p$, the model $\text{TIMELINES}_p$ produces only the timeline belonging to process $p$.

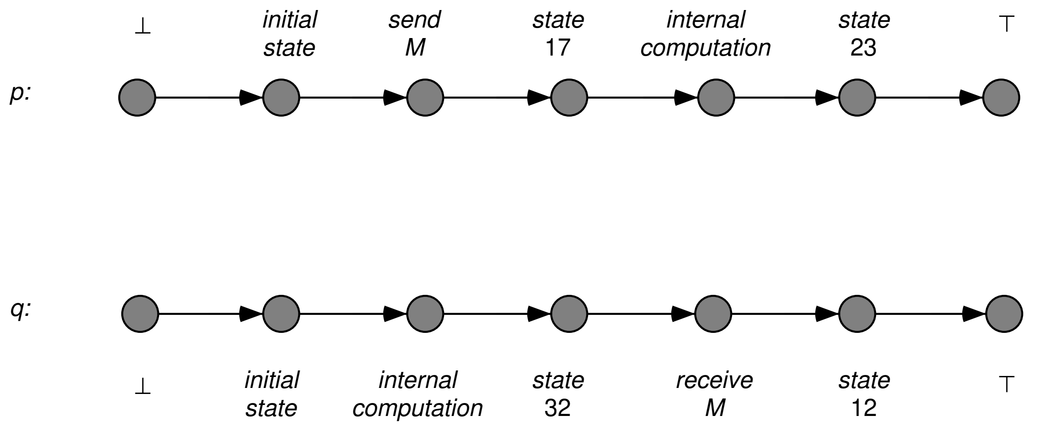


**Figure 2.8** The TIMELINES model produces this graph when applied to the ground-level graph of Figure 2.1.

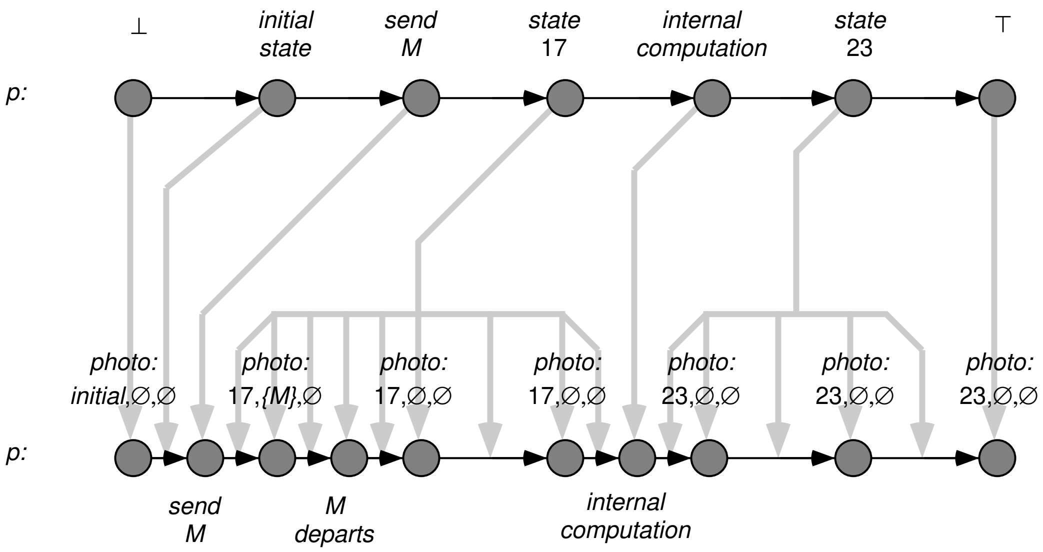


**Figure 2.9** Each node in the TIMELINES model represents a sequence of atoms in the original ground-level graph. This diagram shows how atoms in the process $p$ part of the TIMELINES graph of Figure 2.8 represents atoms in the process $p$ part of the ground-level graph from Figure 2.1.

**The Partial Order** The PARTIAL_ORDER_TIME model organizes the timelines of TIMELINES into a system-wide partial order. We obtain the PARTIAL_ORDER_TIME image of a ground-level computation graph $\alpha$ in several steps:

- We apply TIMELINES to $\alpha$.
- We merge the $\perp$ nodes into a single global $\perp$ node.
- We merge the $\top$ nodes into a single global $\top$ node.
- For each received message, we draw a directed edge from its send node to its receive node.

Representation follows directly from TIMELINES representation: the $\perp$ and $\top$ nodes represent the union of what the merged nodes represent, and the message edges represent nothing. Figure 2.10 sketches this construction.

Since a trace must have at least two frames, we observe that the minimal PARTIAL_ORDER_TIME graph consists of $\perp$, $\top$, and a state node for each process.

This construction ensures that a process cannot have two consecutive "external" nodes (that is, extremal or message event nodes).

The models (PARTIAL_ORDER_TIME, TIMELINES) form a Type 4 parallel pair: consistency, independence, flow-support, and strong monotonicity are all easily established. Indeed, we could *define* flow-support in terms of $\overline{\text{PARTIAL\_ORDER\_TIME}}$: a graph $\mathbf{M}(\alpha)$ is flow-supported iff

$$A \longrightarrow B \text{ in } \mathbf{M}(\alpha) \quad \Longrightarrow \quad A \longrightarrow B \text{ in } \overline{\text{PARTIAL\_ORDER\_TIME}}(\alpha)$$

The construction of PARTIAL_ORDER_TIME naturally suggests how to obtain the factoring model PARTIAL_ORDER_TIME/TIMELINES .

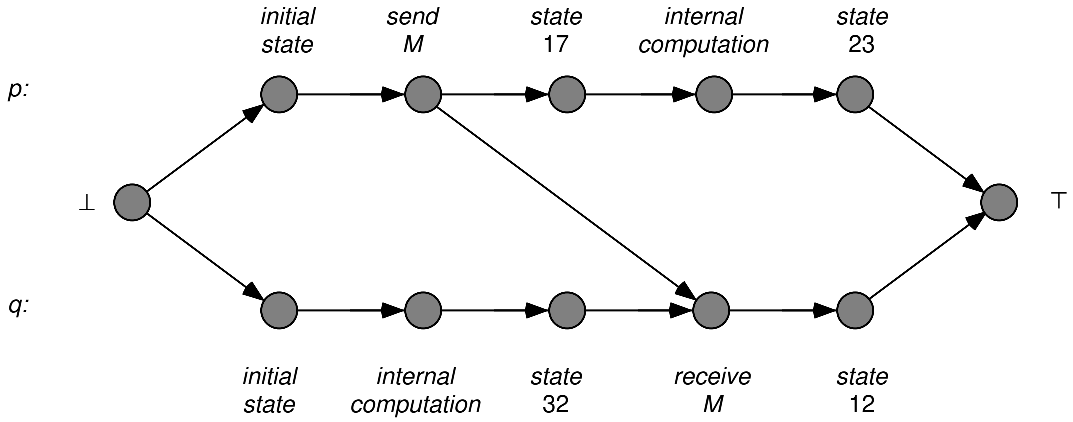


**Figure 2.10** The PARTIAL_ORDER_TIME model produces this graph when applied to the ground-level graph of Figure 2.1.

The transitive closure $\overline{\text{TIMELINES}}$ builds a total order on the nodes at each process. The transitive closure $\overline{\text{PARTIAL\_ORDER\_TIME}}$ builds a partial order on the nodes at all processes.

## 2.3. Timeslices and Global States

This section discusses how the mechanics of distributed time extend to handle the problems of real and apparent simultaneity in asynchronous distributed systems. Section 2.3.1 defines *timeslices* in computation graphs. Section 2.3.2 discusses global states in computations. Section 2.3.3 discusses the relation between global states and timeslices, and Section 2.3.4 discusses the finer structure of timeslices.

### 2.3.1. Timeslices

We construct time models to package periods of activity at processes into events or states, which appear in the computation graph as nodes. Two nodes that a computation graph leaves unordered are logically concurrent, in that the graph does not specify one happening before another. A maximal set of mutually concurrent nodes represents a logical slice of time across this computation; this meaning follows naturally from the semantics of the computation graph: any other node must happen either before or after some node in the set.

We define a *timeslice*[1] to be a maximal mutually concurrent set of nodes. That is, $X$ is a timeslice iff $X$ satisfies two conditions:

1. $X$ is *mutually concurrent*: no $A, B \in X$ satisfy $A \longrightarrow B$, and

2. $X$ is *maximal*: no mutually concurrent $Y$ exists properly containing $X$.

This definition of mutually concurrent automatically prohibits cyclic events from timeslices.

A *partial timeslice* is a subset of a timeslice—that is, a set of mutually concurrent nodes that is not necessarily maximal. (If the precedence relation from a computation graph were guaranteed to be an order,[2] then a partial timeslice is simply an *antichain*.)

[1]Spezialetti [Sp89] uses the term "timeslice," and Mattern [Ma89] uses "time slice"; the timeslices there are special cases of the timeslices here.

[2]Section 2.2.3 presented a formal definition.

### 2.3.2. Global States

**The Physical Computation** In the physical system, computation takes place in the space-time region consisting of the cross product of the set of processes with a continuous interval of real time. A physical global state consists of the state of the entire system at some point in time—that is, a slice of the space-time region.

**Ground-Level Graphs** Each atom of a ground-level computation graph $\alpha$ implicitly represents some subset of the computation space-time region. The collection of subsets represented by all the atoms in $\alpha$ constitutes a partition of the space-time region. For the space-time slice corresponding to a physical global state, we can find sets of atoms from the ground-level graph $\alpha$ that represent a subset of the space-time region that contains this slice. (For a trivial example, consider the set of *all* the atoms in the graph.) We say that a set $X$ of atoms of ground-level $\alpha$ is a *global state* when it is the minimal subset representing a slice: when $X$ contains the slice but no proper subset of $X$ does.

**Abstract Graphs** Suppose time model $\mathbf{M}$ is defined for ground-level graphs. A computation graph $\beta$ produced by $\mathbf{M}$ is supposed to "forget" which ground-level graph generated it. The graph $\beta$ is also supposed to express the objects of interest as nodes. The model $\mathbf{M}$ has an explicit representation map to tell us what these nodes represent in pre-images of $\beta$. To talk about global states in $\beta$, we want to talk about three aspects:

- a set $X$ of *nodes* in $\beta$
- that minimally represents a ground-level *global state*
- in *some* ground-level graph that $\mathbf{M}$ transforms to $\beta$.

Formally, suppose that time model $\mathbf{M}$ is defined on ground-level graphs. A graph $\beta$ that $\mathbf{M}$ produces is the $\mathbf{M}$ image of at least one ground-level graph $\alpha$. A set $X$ of nodes in $\beta$ *minimally represents a global state* when some ground-level graph $\alpha$ exists satisfying the conditions:

- $\mathbf{M}(\alpha) = \beta$;
- $X$ represents a global state $Y$ in $\alpha$:

$$Y \subseteq \bigcup_{A \in X} \langle\, \mathbf{M},\, \alpha\, \rangle(A)$$

- however, no proper subset of $X$ represents $Y$.

Figure 2.11 illustrates how node sets from higher-level graphs correspond to global states.

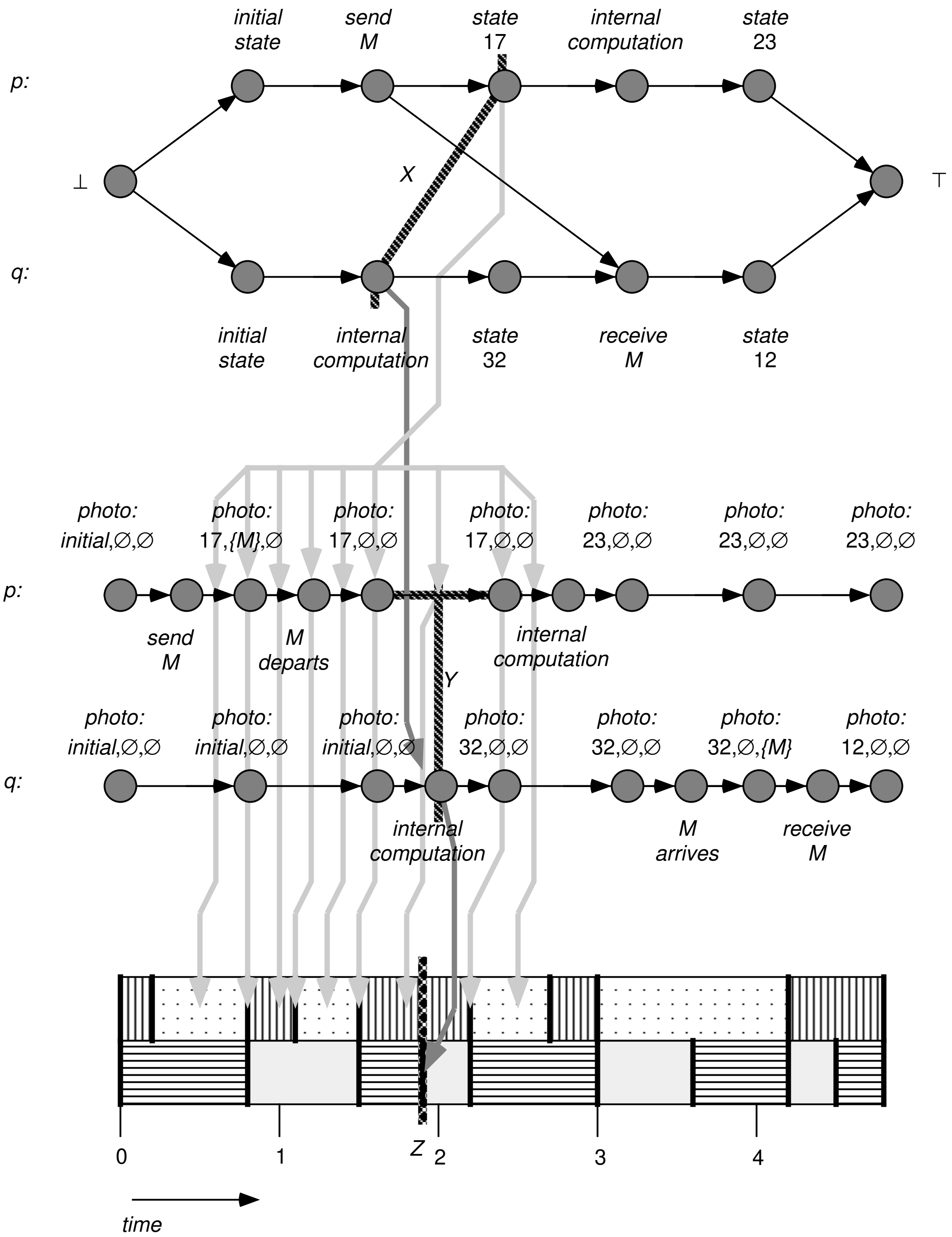


**Figure 2.11** Global states arise from real simultaneity. Here, the region $Z$ in the space-time diagram at the bottom indicates the activity at time $t = 1.9$. The atom set $Y$ in the ground-level graph in the middle is the minimal set mapping to this instant, and thus is a global state. The node set $X$ in the PARTIAL_ORDER_TIME graph at top minimally represents the global state $Y$. (The set $X$ also is a timeslice.)

### 2.3.3. The Relation Between Timeslices and Global States

A ground-level graph $\alpha$ expresses the physical computation. Its abstraction under the time model $\mathbf{M}$ is the graph $\beta = \mathbf{M}(\alpha)$. In general, time models will not be injective: many ground-level graphs may map to $\beta$ under $\mathbf{M}$. If the set of ground-level graphs describes "possible" computations, the set

$$\mathcal{G} \quad = \quad \{\mathbf{M}(\alpha) \ : \ \alpha \text{ is a ground-level graph}\}$$

describes the possible computations when viewed through the model $\mathbf{M}$.

**The Timeslice Condition** If $\mathbf{M}$ is well-constructed, then the timeslices in a graph that $\mathbf{M}$ generates represent exactly the significant global states in the physical computations from which this graph abstracts. Formally, suppose time model $\mathbf{M}$ on ground-level graphs generates the set $\mathcal{G}$. We develop criteria for a model to have timeslices with the appropriate semantics. Model $\mathbf{M}$ satisfies the *Timeslice Condition* iff for each $\beta \in \mathcal{G}$, $\mathbf{M}$ satisfies these requirements:

1. For each set $X$ of nodes in $\beta$, the following are equivalent statements:
   - $X$ minimally represents a global state $Y$ in some ground-level graph $\alpha$ with $\mathbf{M}(\alpha) = \beta$.
   - $X$ is a timeslice in $\overline{\beta}$.
2. Each ground-level graph $\alpha$ with $\beta = \mathbf{M}(\alpha)$ and each global state $Y$ in $\alpha$ satisfy the statement:
   - If $\langle\, \mathbf{M},\, \alpha \,\rangle(A) \cap Y \neq \emptyset$ for some node $A$, then some timeslice in $\overline{\beta}$ minimally represents $Y$.

Some partial order models fail to meet the Timeslice Condition. For example, a version of PARTIAL_ORDER_TIME that omitted the state nodes would fail: if a process $p$ executes a receive immediately after a send, then global states corresponding to the real time interval between those events cannot be represented by timeslices. Figure 2.12 sketches an example.

The view-completeness property from Section 2.2.6 prevents these scenarios where timeslices cannot extend to all processes.

**Theorem 2.1** Suppose $(\mathbf{M}, \mathbf{M}')$ is a Type 1 parallel pair. Then all timeslices in $\overline{\mathbf{M}}$ touch every process.

*Proof* Suppose timeslice $X$ does not touch process $p$. Let $A$ be the maximal node at $p$ that precedes or equals some node in $X$. Let $B$ be the minimal node at $p$ that follows some node in $X$. We must have $A \longrightarrow B$, for if $B \Longrightarrow A$ then $X$ could not be a timeslice. All nodes and edges between $A$ and $B$ must be mutually concurrent with each node in $X$. Further, the first

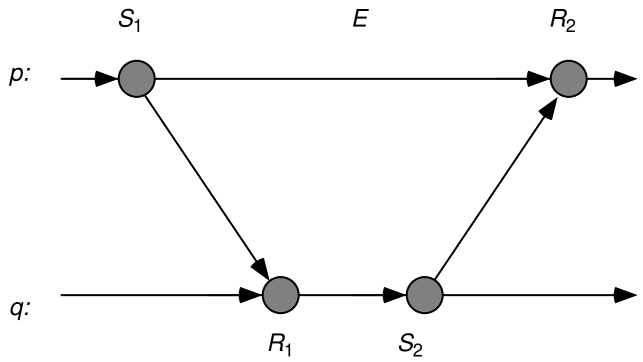


**Figure 2.12** Consider the partial order produced by the transitive closure of this event-only graph. Edge $E$ at process $p$ is concurrent with both $R_1$ and $S_2$ at process $q$. However, all nodes at process $p$ either precede $R_1$ or follow $S_2$. Consequently, this graph is not view-complete. As a result, no timeslice can minimally represent a global state containing $R_1$ or $S_2$, since any corresponding process $p$ node will not be concurrent.

and last edges must be acyclic (otherwise $A$ would advance and/or $B$ would move back). View-completeness gives the existence of nodes at $p$ with the same properties, thus $X$ could not have been a timeslice. □

By including state nodes, the PARTIAL_ORDER_TIME model of Section 2.2 is easily view-complete and thus consistent. The construction of PARTIAL_ORDER_TIME provides some additional properties:

- Precedence of two nodes in $\overline{\text{PARTIAL\_ORDER\_TIME}}$ implies real-time precedence of the activity those nodes represent in any underlying computation.
- Each node in PARTIAL_ORDER_TIME represents a connected region of activity at a process.
- The activity of each process any point in time is represented by some node in PARTIAL_ORDER_TIME.

These properties serve to establish the following result:

**Theorem 2.2** The PARTIAL_ORDER_TIME model satisfies the Timeslice Condition.

*Proof* Let $\beta$ be a graph generated by PARTIAL_ORDER_TIME.

Suppose node set $X$ is not a timeslice. Then either $X$ does not touch every process (in which case it cannot represent a global state), or $X$ is not mutually concurrent (in which case one node of $X$ must precede another in $\overline{\beta}$, and thus in real time in all traces).

Suppose node set $X$ is a timeslice. We construct a trace where the activities $X$ represents are simultaneous. Assign integers to the nodes of $\beta$ by first setting each node of $X$ to 0, then setting each node $A$ following $X$ to be one greater than the maximum value of its predecessors and each $A$ preceding $X$ to be one less than the maximum value of its successors. If $-j$ is the value on $\perp$, add $j$ to each value. A trace exists that schedules each instantaneous node ($\perp$, $\top$, and transition nodes) labelled $i$ at $t = i$, and each state node labelled $i$ in the open interval $(i - 1, i + 1)$. Then timeslice $X$ describes the state of the system at $t = j$.

In any computation generating $\beta$, a physical global state generates a node set $X$ in $\beta$ touching every process. $\square$

### 2.3.4. The Structure of Timeslices

By definition, a timeslice is maximal set of mutually concurrent nodes. What do these timeslices look like? If acyclic, the singletons $\{\top\}$ and $\{\perp\}$ are trivially timeslices: no concurrent nodes exist. What about the other timeslices?

Naively, a timeslice should consist of one node per process. In general models, nodes may represent activity at multiple processes. Hence in general, the informal "one-per-process" tuple has two formal characterizations:

- as a *vector*—an array of nodes, with the constraint that the process $p$ entry occurs at $p$; and
- as a *cut*—a set of nodes that contains, for each process $p$, exactly one node occurring at $p$.

A cut is the node set of a unique vector, but the node set of an arbitrary vector is not necessarily a cut. In either case, we can use projection to isolate particular entries—e.g., $\pi_p X$ is the process $p$ entry of $X$.

The literature uses *consistent cut* for a cut that is also a timeslice in the global model. If the node set of a vector is a timeslice, then it is also a consistent cut (because in parallel pairs, the local process models are total orders, so distinct nodes at the same process cannot be concurrent). However, not all timeslices will be consistent cuts—Figure 2.12 shows a counter-example. View-completeness eliminates this problem for our partial order models, as Theorem 2.1 showed. View-completeness also provides a convenient extension property for partial timeslices.

> **Corollary 2.3** Let $(\mathbf{M}, \mathbf{M}')$ be a Type 1 parallel pair. Any set of mutually concurrent nodes from $\overline{\mathbf{M}}$ extends to a full consistent cut.

The timeslices in a partial order graph will be the extrema singletons (which are easily consistent cuts, since the extrema represent every process), and the sets consisting of a non-extremal node from each process.

**Timestamp Vectors** Let $(\mathbf{M}, \mathbf{M}')$ be a parallel pair. We define the *timestamp vector* for a node $A$ to be the vector $\mathbf{V}(A)$ consisting of the maximal node at each process that precedes or equals $A$ in the global model. That is, let $B$ be the process $p$ entry $\pi_p\,\mathbf{V}(A)$. Then $B \Longrightarrow A$ in $\overline{\mathbf{M}}$, and each node $C$ at each process $p$ satisfies

$$C \Longrightarrow A \text{ in } \overline{\mathbf{M}} \quad \Longrightarrow \quad C' \Longrightarrow B' \text{ in } \pi_p\,\overline{\mathbf{M}'}$$

(where $C'$ is the maximal $\pi_p\,\overline{\mathbf{M}'}$ node that $C$ represents there, and $B'$ is the minimal).

When the global model of a parallel pair is transitively bounded, all entries of all timestamp vectors are defined. When the parallel pair is Type 2 as well, the definition becomes much simpler, since each non-extremal node in $\mathbf{M}$ corresponds to a unique node in $\mathbf{M}'$.

View-completeness endows timestamp vectors with another useful property:

> **Theorem 2.4** Suppose parallel pair $(\mathbf{M}, \mathbf{M}')$ is Type 1. Let $A_1, \ldots, A_k$ be mutually concurrent nodes in a graph from $\overline{\mathbf{M}}$; let $p$ be a process at which no $A_i$ occurs, and let node $B$ be the $p$-maximal node among the $p$ entries of the vectors $\mathbf{V}(A_i)$.
>
> Then there exists a minimal acyclic node $C$ following $B$ at $p$, and $C$ is concurrent with each $A_i$.

*Proof* This result follows directly from the proof of Theorem 2.1. □

Suppose $A$ is a node in a PARTIAL_ORDER_TIME graph that does not occur at process $p$. One implication of Theorem 2.4 is that the node following the $p$ entry of $\mathbf{V}(A)$ is mutually concurrent with $A$.

**Rollback Vectors** We can define *rollback vectors* as the dual to timestamp vectors. The rollback vector for a node $A$ is the vector $\mathbf{R}(A)$ consisting of the minimal node at each process that follows or equals $A$. That is, let $B$ be the process $p$ entry $\pi_p\,\mathbf{R}(A)$. Then $A \Longrightarrow B$ in $\overline{\mathbf{M}}$, and each node $C$ at each process $p$ satisfies the statement:

$$A \Longrightarrow C \text{ in } \overline{\mathbf{M}} \quad \Longrightarrow \quad B' \Longrightarrow C' \text{ in } \pi_p\,\overline{\mathbf{M}'}$$

(Again, let $B'$ be the maximal node that $B$ represents in the $p$ timeline, and let $C'$ be the minimal that $C$ that $C$ represents.)

Just as timestamp vectors describe the maximal history cone of an node, rollback vectors describe the minimal future cone. The term "rollback vector" originates in this fact: if $A$ were to be instantaneously undone, $\mathbf{R}(A)$ describes the frontier of the region to be rolled back. The dual of Theorem 2.4 holds for rollback vectors.

**Precedence of Vectors** The linear order on individual timelines induces a natural relation on vectors: we say that $V \prec W$ when $\pi_p\,V \Longrightarrow \pi_p\,W$ for each process $p$, but $V \neq W$.

**The Timeslice Lattice** Vectors of nodes, one per process, induce some natural entry-wise operations—one of which we have already done. For vectors $X$ and $Y$, define their *meet* $X \sqcap Y$ to be the vector obtained by taking, for each process $p$, the minimal $p$ entry from $X$ and $Y$. Define the *join* $X \sqcup Y$ symmetrically by taking the entry-wise maximum.

We know that timeslices from Type 1 parallel pairs are consistent cuts, and thus have a vector structure. In Type 2 parallel pairs, the meet and join operations preserve this property.

> **Theorem 2.5** Suppose $X$ and $Y$ are consistent cuts in a Type 2 parallel pair. Then $X \sqcap Y$ and $X \sqcup Y$ are both consistent cuts.

*Proof* Suppose $Z = X \sqcap Y$ is not a timeslice. Then $Z$ must equal neither $X$ nor $Y$. There must exist processes $p$ and $q$ such that $X$ contributes the process $p$ entry of $Z$ and $Y$ contributes the $q$ entry, but these entries are not mutually concurrent. Let $A, B$ be the $p$ entries of $X, Y$ (respectively), and $C, D$ be the $q$ entries. By hypothesis, $A \longrightarrow B$ but $D \longrightarrow C$. If $A$ and $D$ are not concurrent, then either $A \longrightarrow C$ (so $X$ is not a timeslice) or $D \longrightarrow B$ (so $Y$ is not a timeslice).

The case for join is symmetric. $\square$

Entry-wise precedence $\prec$ partially orders consistent cuts; in this order, $X \sqcup Y$ is the least consistent cut dominating consistent cuts $X$ and $Y$ and $X \sqcap Y$ is the greatest consistent cut dominated by $X$ and $Y$. These observations, along with Theorem 2.5, suffice to establish that timeslices form a lattice: a nonempty, partially-ordered set, such that each pair of elements has a least upper bound and greatest lower bound in the set [DaPr90].

> **Theorem 2.6** Timeslices in Type 2 parallel pairs form a lattice.

**Adjusted Vectors** An easy variation of Theorem 2.6 is that the set of timeslices containing some specified node also forms a lattice (since meet and join will preserve this membership). The bounds on this lattice derive directly from timestamp and rollback vectors.

Let $A$ be a non-extremal node at process $p$ in a graph from a Type 1 parallel pair. Theorem 2.4 tells us that for each $q \neq p$, a minimal acyclic node exists in the $q$ timeline following the $q$ entry of $\mathbf{V}(A)$. Define the *adjusted timestamp vector* $\mathbf{V}^*(A)$ by replacing each non-$p$ entry in $\mathbf{V}(A)$ by this "successor." Similarly define the *adjusted rollback vector* $\mathbf{R}^*(A)$ by replacing each non-$p$ entry with its "predecessor": the maximal acyclic node preceding the $\mathbf{R}(A)$ entry.

For acyclic Type 2 models, this construction is stated more simply: if $A$ occurs at $p$, obtain $\mathbf{V}^*(A)$ by replacing each each non-$p$ entry of $\mathbf{V}(A)$ by its immediate successor, and obtain $\mathbf{R}^*(A)$ by replacing each non-$p$ entry of $\mathbf{R}(A)$ by its immediate successor.

**Theorem 2.7** Let $(\mathbf{M}, \mathbf{M}')$ be a Type 1 parallel pair. Let $A$ be an acyclic node from an $\overline{\mathbf{M}}$ graph. Let $\{X_1, ..., X_k\}$ be the set of all timeslices containing $A$. Then

$$\begin{aligned}\mathbf{V}^*(A) &= X_1 \sqcap X_2 \sqcap ... \sqcap X_k \\ \mathbf{R}^*(A) &= X_1 \sqcup X_2 \sqcup ... \sqcup X_k\end{aligned}$$

*Proof* From Theorem 2.4, $A$ is either concurrent with or equals each element in its adjusted vector. From Corollary 2.3, timeslices exist containing $A$ and each of these elements. Thus the bound can be achieved. By definition, no element from $\mathbf{V}(A)$ or $\mathbf{R}(A)$ except $A$ can be in a timeslice with $A$. Further, no cyclic node can be in a timeslice with $A$. Thus, these bounds are tight. □

## 2.4. Clocks for Distributed Time

Section 2.4.1 sketches some clock primitives for time models. Section 2.4.2 sketches some clock primitives for parallel pairs. Section 2.4.3 considers issues of when clocks have sufficient information to answer these queries. Section 2.4.4 discusses how timestamp vectors form a basis for an implementation of these primitives.

**An Implicit Parameter** The behavior of clock primitives will all be specified in terms of the ground-level computation graph current at the time of execution. We denote this graph by *CUR_GRAPH*. We do not include this graph as an explicit parameter since the processes that will invoke these primitives will not have explicit access to this graph.

### 2.4.1. Primitives for Time Models

Suppose time model $\mathbf{M}$ acts on ground-level computation graphs. We define the most fundamental clock primitive:

- *PRECEDES*$(A, B, \mathbf{M})$ returns *true* iff $A$ and $B$ are nodes in the graph $\mathbf{M}($*CUR_GRAPH*$)$ and an edge in this graph connects $A$ to $B$.

*PRECEDES* allows us to implement two other primitives:

- *CONCURRENT*$(A, B, \mathbf{M})$ returns *true* iff $A$ and $B$ are nodes in graph $\mathbf{M}($*CUR_GRAPH*$)$, and in this graph, $A$ and $B$ are concurrent.

$$\begin{aligned}&\mathit{CONCURRENT}(A, B, \mathbf{M}) \equiv \\ &\quad \neg \mathit{PRECEDES}(A, B, \mathbf{M}) \;\wedge\; \neg \mathit{PRECEDES}(B, A, \mathbf{M})\end{aligned}$$

- *ACYCLIC*$(A, \mathbf{M})$ returns *true* iff node $A$ is acyclic in $\mathbf{M}($*CUR_GRAPH*$)$.

$$\textit{ACYCLIC}(A, \mathbf{M}) \quad \equiv \quad \neg \textit{PRECEDES}(A, A, \mathbf{M})$$

This specification raises some questions. Are the primitives well-defined? How do processes provide these parameters? The remainder of this section considers these issues.

**Well-defined Answers** Suppose system computation extends the current ground-level computation graph from $\alpha$ to $\alpha'$. If $A \longrightarrow B$ in $\alpha$, will $A \longrightarrow B$ in $\alpha'$? If $A \not\longrightarrow B$ in $\alpha$, will $A \not\longrightarrow B$ in $\alpha'$? The monotonicity definitions in Section 2.2.5 provide some answers:

- If the time model $\mathbf{M}$ is strongly monotonic, then the *PRECEDES* primitive is well-defined.
- If the time model $\mathbf{M}$ is only weakly monotonic, then the *PRECEDES* primitive still behaves reasonably, with the exception of occasionally changing from *false* to *true* as computation progresses.

We make the implicit assumption that the models we define primitives for are strongly monotonic. However, we note that the weakly monotonic case can also be made to work once we handle the problem of *convergence*: knowing when a precedence answer become stable.

**Node Names** Processes using these primitives must specify nodes as parameters. Specifying these primitives begged the question of how processes themselves should refer to nodes. We assume that nodes in a computation have unique *names*. Whether names should be mere identifiers (e.g., "node 73 at process 12") or more complete descriptions (e.g., "node 73 at process 12: state change from $q_3$ to $q_7$") is another issue. This naming convention carries an implicit assumption: from the information in a node name, one may extract the process at which the node occurred.

**Shifting Models** We use these simple primitives to ask about precedence in a model $\overline{\mathbf{M}}$. However, a natural extension is to ask about other types of precedence using other models. For example, in a parallel pair $(\mathbf{M}, \mathbf{M}')$, we can ask about individual steps at processes using $\mathbf{M}'$, or about precedence at process $p$ using $\overline{\pi_p \mathbf{M}'}$. The format of *PRECEDES* and *CONCURRENT* already grants this ability: we use the model parameter to specify the appropriate model. However, shifting nodes between levels in a parallel pair can be tricky, because a node from a parallel pair exists on three levels:

- as a node in the global graph;
- as the set of nodes it corresponds to in the disjoint union of the local graphs;
- as the set of nodes, if any, it corresponds to in each process graph.

In general, shifting levels requires some care to avoid ambiguity. For Type 2 parallel pairs such as $(\text{PARTIAL\_ORDER\_TIME}, \text{TIMELINES})$, this multiplicity is simple: each node in the global partial order represents exactly one node at one process, except for $\perp$ and $\top$.

### 2.4.2. Primitives for Pairs

We define some additional primitives for parallel pairs and nonlinear pairs. We assume that our pair is Type 3: both strongly monotonic and Type 2. (Strong monotonicity assures us that precedence relations are well-defined; Type 2 provides convenient node structure.)

**A Primitive for "Now"** First, processes need access the name of their current node. We specify a primitive:

- *CUR_NODE*$(p, (\mathbf{M}, \mathbf{M}'))$ returns the name of the current process $p$ node in the graph $\mathbf{M}'($*CUR_GRAPH*$)$.

We allow only process $p$ to ask *CUR_NODE*$(p, (\mathbf{M}, \mathbf{M}'))$.

**Vector Operations** Processes need to perform vector operations in nonlinear pairs. We specify two primitives:

- *MAX*$(V, W, (\mathbf{M}, \mathbf{M}'))$ is defined for vectors $V$ and $W$ of nodes from $\mathbf{M}($*CUR_GRAPH*$)$, and returns the entry-wise maximum (using $\overline{\mathbf{M}'}$ to sort entries).
- *COMPARE*$(V, W, (\mathbf{M}, \mathbf{M}'))$ is defined for vectors $V$ and $W$ of nodes from $\mathbf{M}($*CUR_GRAPH*$)$, and is true iff $V \preceq W$ (using $\overline{\mathbf{M}'}$ to sort entries).

**Meta-Primitive** We want to define enumerative primitives for our clock suite. We begin by defining two "meta-primitives" as building blocks. Let $A$ be a variable representing an unspecified node, and $\Phi$ be a predicate on $A$. We specify two meta-primitives:

- *LIST*$(A, \Phi(A), \beta)$ returns the set of nodes in graph $\beta$ that, when substituted for $A$, satisfy $\Phi(A)$.
- *NODE*$(A, \Phi(A), \beta)$ returns the single node $A$ from $\beta$ satisfying $\Phi(A)$ (and is undefined otherwise).

These meta-primitives themselves are off-limits for processes.

**Primitives that Enumerate** We use *LIST* and *NODE* to build clock primitives that enumerate nodes, rather than merely providing Boolean answers. These primitives apply to Type 3 parallel pairs only. (Recall that in a Type 2 parallel pair $(\mathbf{M}, \mathbf{M}')$, we identify nodes in $\mathbf{M}$ with nodes in $\mathbf{M}'$.) We specify three primitives:

- *NEXT*$(p, A, (\mathbf{M}, \mathbf{M}'))$ returns the $\mathbf{M}$ node that follows node $A$ in the process $p$ timeline.

$$\textit{NEXT}(p, A, (\mathbf{M}, \mathbf{M}')) \equiv$$

$$\textit{NODE}(B, \textit{PRECEDES}(A, B, \pi_p \mathbf{M}'), \pi_p \mathbf{M}'(\textit{CUR\_GRAPH}))$$

- *PREVIOUS*$(p, A, (\mathbf{M}, \mathbf{M}'))$ returns the $\mathbf{M}$ event that precedes node $A$ in the process $p$ timeline.

$$\textit{PREVIOUS}(p, A, (\mathbf{M}, \mathbf{M}')) \equiv$$

$$\textit{NODE}(B, \textit{PRECEDES}(B, A, \pi_p \mathbf{M}'), \pi_p \mathbf{M}'(\textit{CUR\_GRAPH}))$$

- *LIST_CONCURRENT*$(p, A, (\mathbf{M}, \mathbf{M}'))$ returns the acyclic $\mathbf{M}$ nodes at process $p$ that are concurrent with event $A$.

$$\textit{LIST\_CONCURRENT}(p, A, (\mathbf{M}, \mathbf{M}')) \equiv$$

$$\textit{LIST}(B, \textit{CONCURRENT}(A, B, \overline{\mathbf{M}}) \wedge \textit{ACYCLIC}(B, \overline{\mathbf{M}}), \pi_p \mathbf{M}'(\textit{CUR\_GRAPH}))$$

### 2.4.3. Knowable Pursuits

Section 2.4.1 considered when the temporal relation that a clock primitive examines is well-defined. However, we have not examined when when a process executing of a clock primitive in an unfolding computation will have sufficient information to obtain this well-defined answer. For example:

- When should the clock at process $p$ be expected to handle queries about a node $A$?
- What precedence relations should the clock at $p$ be expected to know about?

To answer these questions, we informally consider an "Elephant-Pig Paradigm": processes never forget anything, and always piggyback each link in a precedence path with complete knowledge. Although this paradigm would not be met by real implementations, it serves to a starting point. Suppose we use parallel pair $(\mathbf{M}, \mathbf{M}')$ to describe computation, and node $C$ occurs at process $p$. We specify some clock guidelines:

- If $A \longrightarrow C$ in $\overline{\mathbf{M}}$, then process $p$ at node $C$ may ask about $A$.
- If $A$ and $B$ both precede $C$ in $\overline{\mathbf{M}}$, then process $p$ at node $C$ may ask about the relation between $A$ and $C$.

For our models, flow-support reasonably approximates the Elephant-Pig Paradigm. If we restrict the knowability questions to a parallel pair $(\mathbf{M}, \mathbf{M}')$ is also flow-supported (e.g., $(\mathbf{M}, \mathbf{M}')$ is a Type 4 parallel pair), this sketch provides some answers:

- Process $q$ at node $C \in \mathbf{M}(\textit{CUR\_GRAPH})$ will get an answer from the queries

  $$\textit{PRECEDES}(A, B, \overline{\mathbf{M}'})$$

  $$\textit{PRECEDES}(A, B, \overline{\mathbf{M}})$$

  iff $B \rightrightarrows C$ in $\overline{\mathbf{M}}(\textit{CUR\_GRAPH})$.

- Process $q$ at node $C \in \mathbf{M}(\textit{CUR\_GRAPH})$ will get an answer from

  $$\textit{NEXT}(p, A, (\mathbf{M}, \mathbf{M}'))$$

  iff this node exists, and precedes or equals $C$ in $\overline{\mathbf{M}}(\textit{CUR\_GRAPH})$.

- Process $q$ at node $C \in \mathbf{M}(\textit{CUR\_GRAPH})$ will get an answer from

  $$\textit{PREVIOUS}(p, B, (\mathbf{M}, \mathbf{M}'))$$

  iff $B \rightrightarrows C$ in $\overline{\mathbf{M}}(\textit{CUR\_GRAPH})$.

- Realistically, it seems unreasonable for a process to know *everything* in its past. Consequently, we restrict *LIST_CONCURRENT* to examine local nodes only. Only process $p$ can query $\textit{LIST\_CONCURRENT}(p, A, (\mathbf{M}, \mathbf{M}'))$, only for $A$ preceding the query node.

### 2.4.4. An Implementation

Vector clocks provide a natural approach for tracking temporal precedence in parallel pairs. Historically, research in partial order time includes vector-based clock implementations [StYe85, Fi88, Fi91, Ma87, KeKo89, Ma89]. Indeed, the term "vector time" has surfaced for partial order time, although we feel this is a misnomer as it confuses an implementation with the underlying structure. (However, these particular implementations do permit extra elegance in some applications.)

The vector relation on timestamp vectors follows the temporal relation on the events.

**Theorem 2.8** Any two nodes $A$ and $B$ from a Type 1 parallel pair. satisfy the statement:

$$\mathbf{V}(A) \prec \mathbf{V}(B) \iff (A \longrightarrow B \ \wedge \ B \not\longrightarrow A)$$

*Proof* Each entry of a timestamp vector for a given event precedes or equals that event. If $A \longrightarrow B$ then, then the $p$ entry of $\mathbf{V}(A)$ precedes or equals $B$, and thus precedes or equals the maximal node at $p$ preceding $B$. Conversely, suppose $\mathbf{V}(A) \prec \mathbf{V}(B)$ and $A$ occurs at process $q$. Then $A$ precedes or equals the $q$ entry of $\mathbf{V}(A)$, which precedes or equals the $q$ entry of $\mathbf{V}(B)$, which precedes or equals $B$. $\square$

(In fact, this theorem holds for parallel pairs more general than Type 1. Only transitive bounding is required.)

Strong monotonicity implies that the timestamp vector for an event can actually be defined at some point in the computation. Flow-completeness implies that the timestamp vector for an event can actually be defined when the event occurs. Consequently, to implement clocks for Type 4 parallel pairs using timestamp vectors, we just have each process maintain a local counter and a "current" timestamp vector. When a process sends a message, it piggybacks the timestamp vector of the send; when a process receives a message, it updates its current vector to be the entry-wise maximum. Timestamp vectors allow direct implementation of the *PRECEDES* and *CONCURRENT* primitives, and, along with some facility for remembering history and event descriptions, allows implementation of the remainder of the primitives.

Timestamp vectors also function as clocks for more general types of parallel pairs, such as those lacking flow-support, and those whose process timelines are themselves partial orders. The implementation becomes somewhat more complicated in these scenarios, however. For example, non-flow-supported models suffer from an information gap: when event $A$ occurs at process $p$, process $p$ may not have sufficient information to sort $A$. The answer to the query *PRECEDES*$(\mathbf{M}_2, A, B, \alpha)$ depends on when the query is made—and we need a time model $\mathbf{M}_1$ that refines to $\mathbf{M}_2$ to capture this parameter. (This scenario is an example of *parameterized clocks.*) Alternatively, when a process timeline is itself a partial order, we need to distribute information so that other processes can perform the vector clock algorithm—sorting two events at process $p$ is no longer a matter of comparing two scalars. (Chapter 4 and Chapter 5 discuss these issues in more detail.)

In principle, rollback vectors also function as clocks (the dual to Theorem 2.8 holds), but information gaps makes implementation impractical.

## 2.5. Example Applications

### 2.5.1. Orphan Detection

An immediate application of distributed time is accurate *orphan detection*. When an event is aborted, any event that could have been influenced by the aborted event is an orphan and should be undone.

Tracking this dependence in an asynchronous distributed system is difficult. For example, using real time to label as an orphan any event with a timestamp greater than the aborted event will generate false positives, and not extend to work in environments lacking synchronized real time clocks. Using a total order consistent with the underlying computation also generates false positives—and fails to extend to scenarios such as rollback recovery, in which the final (replayed) instance of an event may actually occur later than an event it influenced.

The tools of distributed time solve these problems by allowing us to talk about time as a partial order, and by allowing us to move transparently from the partial order representing the physical computation to a more abstract partial order that represents a virtual computation.

### 2.5.2. Immediate Ordered Service

The problem of *immediate ordered service* consists of servers processing requests from clients in an asynchronous distributed system. Each server has a list of outstanding requests. How can the server choose the "earliest" entry to process without necessitating additional communication and discussion?

This problem can be solved by applying a partial order time model to the computation, and having servers use partial order clocks to sort the incoming requests. The immediate response time of vector clocks makes that implementation particularly attractive—especially in a distributed, asynchronous, and frequently disconnected environment. (Indeed, the published solution [KeKo89] to this problem is one of the independent discoveries of the vector clock mechanism.)

## 2.6. Comparison to our Earlier Publication

We presented much of this material in an earlier publication [Sm93]. That version was usually more detailed, but was also more preliminary. This section briefly discusses some of the differences.

When defining process automata, the earlier publication allowed processes to know their input queue was nonempty, but proceed without receiving any message. That approach unintentionally permitted anonymous influence: process $p$ may act on the knowledge that a message has arrived from process $q$, but our time models would not establish a precedence path from $q$ to $p$. The revised message rule in this thesis prohibits this scenario.

When developing time models, the earlier publication primarily took the event-based approach. This approach created problems with view-completeness and complicated discussion of certain application problems. This thesis avoids these problems by including both events and states as nodes in computation graphs. In this respect, the construction of PARTIAL_ORDER_TIME in this thesis differs from the POT model of the earlier publication. (In particular, PARTIAL_ORDER_TIME ensures that a state node separates any two event nodes.)

When defining properties of time models, the earlier publication did not formally examine issues of information flow. The definitions of flow-supported, flow-virtual, and monotonicity appear for the first time in this thesis.

Chapters 7 and 8 of the earlier publication gives a much fuller discussion of parallel pairs and factoring models than Section 2.2.6 of this thesis presents. However, the earlier publication did not explore nonlinear pairs, and took a different approach to examining the taxonomy of pairs. The definitions of Type 1 through Type 4 appear appear for the first time in this thesis.

The earlier publication provides a more detailed derivation of the timeslice results. Theorem 2.2 in this thesis reflects Theorems 13.6 through 13.8 of the earlier report.

Theorem 2.6 in this thesis considers only timeslices from Type 2 parallel pairs, although we can show that consistent cuts in general parallel pairs form lattices, as do timeslices from any transitive graph. The general case is difficult due to two facts:

- Some consistent cuts may contain nodes that touch more than one process, but not all of them.
- Some timeslices may not be consistent cuts.

In particular, the definitions here of the $\sqcap$ and $\sqcup$ operations and the $\prec$ relation work correctly for the well-behaved vector-like cuts in independent consistent parallel pairs. More general models require more careful definitions. Chapter 9 of the earlier publication provides the full details.

# Chapter 3

# Distributed Snapshots

## 3.1. Overview

The *distributed snapshot* problem provides a straightforward application of the distributed time framework. In an asynchronous distributed system, what one process perceives about the rest of system is always out-of-date. This limitation complicates the problem of capturing a snapshot: a mosaic depicting the global state of the system at some instant.

In real life, we think of time as a linear sequence of moments. Consequently, we find it only natural to think of computations as linear sequences of global states. Barring anything unusual, this linear model actually describes the behavior of the system. Unfortunately, the asynchrony and the distribution in the system make it difficult for processes within a system to obtain global states. For example, suppose process $p$ at time $t$ wants to take a snapshot of of the global state of the system at time $t$, or even at some unspecified time close to $t$. Although process $p$ needs knowledge of the other processes in order to put together this picture, any knowledge it may obtain will be stale, because information travels at a finite speed. Further, the unpredictable message delays mean process $p$ cannot even know how stale this knowledge is.

**The Traditional Solution** In their foundational paper on snapshots, Chandy and Lamport [ChLa85] present an elegant marker-pushing protocol that works despite this limitation.[1] A process initiating the protocol receives an approximately current snapshot with a counter-intuitive correctness property: while this snapshot may not necessarily describe the state of the system at any single instant, it describes a *consistent* state of the system.

That is, suppose process $p$ initiates a snapshot protocol at time $t_0$, and at time $t_1$ receives a snapshot: a tuple $X$ describing the local state at each process. There exists a well-defined history function $H$ taking each $t$ in the interval $[t_0, t_1]$ to its global state $H(t)$: the tuple consisting of the local process states at $t$. Intuition suggests that the snapshot $X$ ought to be the value of $H$ at some instant in this interval. Asynchrony causes this intuition to fail. Lacking perfect knowledge, process $p$ cannot obtain the $H$ values; lacking real time clocks, process $p$ cannot even obtain the

[1] Their system model—lossless FIFO message channels—somewhat constrains the asynchrony.

$t$ values. A process's only sense of time derives from the messages the process receives and the actions it takes.

Here lies the rub: many valid histories $H'$ exist with $H(t_0) = H'(t_0)$, $H(t_1) = H'(t_1)$ and where each process perceives the same temporal relations. The global state $X$ may not necessarily be an intermediate value from the history that actually occurred, but it will be an intermediate value from an equivalent[2] history.

What is more, this is the best we can do. A consistent global state is consistent with the processes' observations. Hence a snapshot recording a consistent global state is the most accurate picture a process can obtain: anything more accurate would require more detailed observations—which would change the computation.

Even though it may never have occurred, a consistent system state still says useful things about the computation in progress. For example, if property $\Phi$ is *stable*—it remains true once it becomes true—then examining a past consistent system state for $\Phi$ may suffice to determine if $\Phi$ holds at the current instant.

**Subsequent Research** Subsequent research in snapshots explored variations on marker-pushing protocols [SpKe86, LaYa87, Ve89, NeTo90, Ma93], characterized the state lattice that arises from slices across partial order time [Ma89, Jo89, JoZw90], and modified the message delivery model by relaxing the FIFO requirement, and by adding various flushing primitives. Work also progressed in developing applications of distributed snapshots in deadlock detection [Ma87], in checkpointing [Jo89, JoZw90, Jo93] and in distributed debugging [Fi89, Sp89], including efforts to use timestamp vectors to capture consistent states with specific properties [CoMa91, MaNe91, MaSa91, ToGa93, GaWa94]. (Taylor's work [Ta89, CrTa90] uses a more deterministic notion of snapshot—processes must know at the time a state occurs that that state is part of a snapshot—and thus her results do not apply here.)

**Using Distributed Time** The snapshot problem demonstrates that the standard way of thinking about computation—as a linear progression of system states—does not work in an asynchronous distributed system. The unsuitability of linear time makes the snapshot problem an attractive demonstration area for the distributed time framework.

In Chapter 2 we phrased global states in terms of timeslices from a computation graph, and specified clocks for these temporal relations. This framework allows straightforward snapshot

[2]This phenomenon is seductively similar to particle-wave duality. Processes may construct a set of possible paths for the computation. Although the computation takes one path in particular, processes can never know which one: each snapshot causes the set to "jump" to one value, not necessarily the real one. Manthey [MaMo83, Ma90a, Ma90b] has explored the use of *computational* abstractions to model *physics*, and compiled a list of physical phenomena that arise as side-effects of computational behavior. The snapshot problem suggests an interesting extension to this research: exploring what physical phenomena may arise as side-effects of temporal behavior.

protocols: processes use their distributed time clocks to assemble timeslices. This approach offers three significant advantages:

- **Flexibility** Giving processes the ability to sort events and states in terms of the logical time model permits snapshot protocols that are much more flexible than the traditional marker-pushing protocols. For example, we can take multiple snapshots and snapshots containing an arbitrary past event.

- **Orthogonality Between Protocols and Clocks** By encapsulating the problems of tracking time in alternative models into *clocks* for these models, we separate the implemented from the implementations. This orthogonality allows us to modify clock protocols—perhaps due to changing system environments or efficiency goals—without modifying the higher level application protocols. For example, we can transparently add security and privacy to these snapshot protocols by using more secure clocks. (Chapters 5 and 6 consider these issues.)

- **Orthogonality Between Protocols and Time Models** We define global states and snapshot protocols relative to a *time model*. This model encapsulates the logical timing issues: physical reality may determine some linear order, but we pretend the model describes what actually happens. However our notion of what actually happens may change. For example:

  - We may want to abstract further than this level—perhaps by pretending only global states with certain properties occur.
  - We may want to increase the separation between this level and physical reality—perhaps by allowing for rollback with modified replay.

  Using the distributed time framework allows transparent alteration of the level of abstraction in a snapshot protocol by using more abstract time models. (For example, suppose a process rolls back and performs different computation. At least three virtual computations may arise: the failed computation, the failure-free virtual computation, and the recovery computation itself. The distributed time framework allows us to use the same protocol to take snapshots of all three levels. Chapter 4 discusses these issues further.)

Figure 3.1 sketches this approach.

**This Section** Section 3.2 defines the snapshot problem in terms of distributed time, sketches a simple protocol to find snapshots containing any arbitrary event or state (even without FIFO messages), and uses some of our theoretical results to improve this basic protocol. Section 3.3 considers the implications of using basic snapshot protocols with more abstract time models, and shows two examples. Section 3.4 explores some advanced issues.

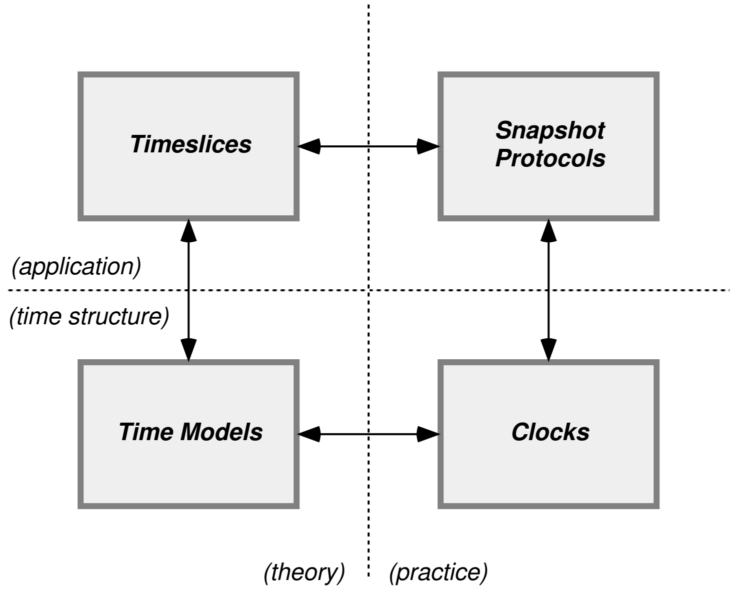


**Figure 3.1** Distributed time simplifies protocol design. In the snapshot application, we can describe the target in terms of distributed time: snapshots are *timeslices*, instants of logical simultaneity in the temporal relations expressed by a *time model*. *Clocks* for these time models permit thinking directly in terms of these relations, and thus provide the necessary primitives for *snapshot protocols*.

## 3.2. The Basic Problem

This section uses distributed time to examine the basic problem of taking a snapshot. Section 3.2.1 builds a basic snapshot protocol. Section 3.2.2 uses vector clocks and the lattice structure of timeslices to simplify this protocol.

### 3.2.1. Building a Basic Protocol

Informally, we think of a global state as what is happening everywhere at some moment in real time. However, we do not want a description of everything happening everywhere (where would we write it all down?) but rather a list of convenient abstractions. The restrictions of distributed asynchrony confine the basic snapshot protocol to capturing what is happening at some moment in time in some computation consistent with what processes observe. Thus for snapshot applications, a well-constructed time model should produce graphs that have two properties:

- the nodes express the desired abstractions, and
- the temporal precedence expresses only the observable orderings.

The PARTIAL_ORDER_TIME model built in Chapter 2 has these properties.

When a process takes a snapshot, it wants to find a global state from a computation consistent with what it and the other processes are observing. By Theorem 2.2, these consistent global states are exactly the timeslices from the global partial order.

**The Round Robin Protocol** An interesting consequence of Corollary 2.3 is that for Type 2 (consistent and independent) parallel pairs, any set of mutually concurrent nodes—even a singleton—extends to a consistent cut. This means that the following naive protocol suffices for a process $p$ to take a snapshot. Let $(\mathbf{M}, \mathbf{M}')$ be a Type 3 (consistent, independent, and strongly monotonic) parallel pair. The protocol assumes the processes are organized into a directed cycle, and performs the following steps:

1. Process $P_1$ chooses an acyclic node $A_1$ and sends $\{A_1\}$ as a partial timeslice to $P_2$.
2. For each $i$ with $1 < i \leq n$, process $P_i$ receives a partial timeslice from $P_{i-1}$ and appends a local acyclic node mutually concurrent with each node in the timeslice. If $i < n$, process $P_i$ sends the new partial timeslice on to $P_{i+1}$. If $i = n$, process $P_n$ sends the completed snapshot back to $P_1$.

Figure 3.2 presents a more complete description. Processes use *LIST_CONCURRENT* to enumerate nodes from their own timelines (consistent with the knowledge restriction from Section 2.4.3). Corollary 2.3 guarantees that $U_i$ will be non-empty.

With modification, the Round Robin Protocol extends to more general parallel pairs. For example, if the *LIST_CONCURRENT* call were guaranteed to be answerable, we could relax the $(\mathbf{M}, \mathbf{M}')$ requirement to Type 1 (consistent). If we rewrote the protocol to allow missing entries from the $S_i$ and to allow some $U_i$ to be empty, we could even dispense with consistency requirement.

Unlike the traditional protocol, the Round Robin Protocol does not require FIFO message delivery, is offline (in that the initiating process may specify any arbitrary seed node), and allows each process some leeway in choosing what node to include in the snapshot.

### 3.2.2. Shortcuts

The Round Robin Protocol is simple and clear; each process receives a partial timeslice, then finds and appends a local node mutually concurrent with that timeslice. The protocol is also fairly inefficient; the multiple *LIST_CONCURRENT* calls followed by the set intersection take time, and $n$ rounds of communication must take place before the initiating process receives the desired snapshot.

However, examining the Round Robin Protocol in terms of vector clocks reveals shortcuts that improve efficiency.

```
/* process P_1 initiates protocol */
procedure INITIATE

    /* find acyclic node to seed the snapshot */
    repeat
        CHOOSE  A_1 ≠ ⊥
    until ACYCLIC(A_1, M̄)

    /* create partial timeslice S_1 and send it off */
    S_1 ← {A_1}
    SEND  S_1 to P_2

/* for each i > 1, process P_i receives a set S_{i-1} and cooperates */
procedure COOPERATE

    /* find the local nodes that are concurrent with each node in S_{i-1} */
    for j = 1 to i - 1
        A_j ← process P_j entry of S_{i-1}
        T_j ← LIST_CONCURRENT(P_i, A_j, (M, M'))

    /* find the intersection */
    U_i ← ⋂_{1≤j<i} T_j

    /* extend the partial timeslice */
    CHOOSE A_i ∈ U_i
    S_i ← S_{i-1} ∪ {A_i}

    if  i < n
        /* if incomplete, send the partial timeslice to the next process */
        then SEND  S_i to P_{i+1}

        /* if complete, send the snapshot back to P_1 */
        else SEND  S_i to P_1
```

**Figure 3.2** In the Round Robin Protocol for distributed snapshots, we assume the processes are organized into a cycle $P_1, ..., P_n$. Process $P_1$ initiates the protocol by choosing a local node that is acyclic, and passing a partial timeslice on to $P_2$. For $i > 1$, process $P_i$ receives a partial timeslice and cooperates by extending it, and passing it on.

**The Reduced Round Robin Protocol** Theorem 2.4 provides a way to combine the timestamp vectors for partial timeslices, and to find nodes with which to extend the partial timeslice. This result allows us to reduce both the messages and the computation in the Round Robin Protocol. Let $(\mathbf{M}, \mathbf{M}')$ be a Type 3 (consistent, independent, and strongly monotonic) parallel pair. The protocol assumes the processes are organized into a directed cycle, and performs the following steps:

1. Process $P_1$ chooses an acyclic node $A_1$ and sends the vector $\mathbf{V}(A_1)$ to $P_2$.
2. For each $i$ with $1 < i \leq n$, process $P_i$ receives a vector from $P_{i-1}$ and replaces the $i$ entry with the next acyclic node $A_i$. $P_i$ them maximizes this vector against the timestamp vector for $A_i$. If $i < n$, process $P_i$ sends the new vector on to $P_{i+1}$. If $i = n$, process $P_n$ sends the completed snapshot back to $P_1$.

Figure 3.3 presents a more complete description.

This protocol improves on the Round Robin Protocol by encoding the partial timeslice $S_i$ as the first $i$ entries of vector $V_i$, and using the remaining entries of the vector to mark the upper bound of the set of nodes preceding $S_i$.

**The Completely Reduced Round Robin Protocol** Some convenient properties of vector clocks made the Reduced Robin Protocol possible. As Chapter 2 explained, these properties have a solid theoretical foundation:

- Timeslices form a lattice.
- The timestamp vector and rollback vector of a node delineate the bounds of the sublattice of timeslices containing that node.

Theorem 2.7 tells us that if a process wants to know a snapshot containing a node, then the adjusted timestamp vector of that node suffices. Thus we can reduce the Round Robin Protocol even further. Let $(\mathbf{M}, \mathbf{M}')$ be a Type 3 (consistent, independent, and strongly monotonic) parallel pair. Our completely reduced protocol performs the following step:

- For a process $p$ to find a snapshot containing node $A$ at $q$, process $p$ independently asks each process $r \neq q$ for the value *NEXT*$(r, \pi_r \mathbf{V}(A), (\mathbf{M}, \mathbf{M}'))$.

This protocol dispenses with the assumption that processes are organized into a cycle.

In general, concurrence is not transitive—to find the next element of a partial timeslice, a process must check every element. But snapshots from the adjusted vectors have the advantage of being canonical, in the sense that the identity of the vector (e.g., "the adjusted timestamp vector of $A$") is sufficient to determine membership. The initiating process still must query other processes, but these queries may now be independent rather than sequential.

```
/* process P_1 initiates protoco1 */
procedure INITIATE

    /* find acyclic node to seed the snapshot */
    repeat
        CHOOSE  A_1 ≠ ⊥
    until  ACYCLIC(A_1, M̄)

    /* send off a vector */
    V_1 ← V(A_1)
    SEND V_1 to P_2

/* for each i > 1, process P_i receives a vector V_{i-1} and cooperates */
procedure COOPERATE

    /* advance P_i entry to next acyclic node */
    A_i ← π_{P_i} V_{i-1}
    repeat
        A_i ← NEXT(P_i, A_i, (M, M'))
    until  ACYCLIC(A_i, M̄)

    /* obtain new vector */
    V_i ← MAX(V_{i-1}, V(A), (M, M'))

    if   i < n
        /* if incomplete, send the new vector to next process */
        then SEND  V_i to P_{i+1}

        /* if complete, send the snapshot back to P_1 */
        else SEND  V_i to P_1
```

**Figure 3.3** In the Reduced Round Robin Protocol for distributed snapshots, we assume the processes are organized into a cycle $P_1, ..., P_n$. Process $P_1$ initiates the protocol by choosing a local node that is acyclic, and its timestamp vector to $P_2$. For each $i > 1$, process $P_i$ receives a vector whose first $i - 1$ entries comprise a partial timeslice, and whose remaining entries are the maximal nodes preceding this partial timeslice. Process $P_i$ cooperates by extending the partial timeslice, updating its vector encoding, and passing it on. The vector maximization step cannot affect the first $i$ entries, since these form a partial timeslice.

The most straightforward way to assemble an adjusted vector takes $2 \log n$ communication steps—use a binary tree to send out the requests, and another to collect them. However, several natural optimizations suggest themselves. For example, if $A$ is sufficiently far in the past, then process $p$ may already know some values. Also, various processes may know the components for other processes.

Theorem 2.7 also gives us ways to collect snapshots without actually having access to clocks. For example, process $p$ could obtain $\mathbf{V}^*(A)$ (the adjusted timestamp vector of $A$) by searching backwards along the paths of messages arriving before $A$. In fact, the traditional snapshot protocol uses essentially this technique: initiate broadcast at node $B$, and obtain $\mathbf{R}^*(B)$ (the adjusted rollback vector of $B$).

## 3.3. Snapshots from Higher-level Models

Both the Round Robin variants and the traditional snapshot protocol obtain a single snapshot. A single snapshot is satisfactory from a linear view: the question of "what is happening right now" should only have a single answer. But distributed asynchrony gives multiple correct answers. This fact made Chandy and Lamport's work appear counter-intuitive, and helps motivate distributed time theory.

Many global states may contain some specific node from process $p$. However, process $p$ may need access to states different from the one that particular run of a single-snapshot protocol provides. Process $p$ has only a couple of approaches:

- The desired states may be exactly the timeslices in some higher-level time model supported by clock primitives. In this case, taking a snapshot in this alternate model suffices.

- Otherwise, process $p$ needs either to extend its snapshot protocol to handle search-and-backtrack, or to find an efficient way to collect sets of global states (so it can perform the search locally).

This section focuses on this *general snapshot problem* (finding a global state satisfying some arbitrary predicate $\Phi$) by taking snapshots from higher-level models. Section 3.3.1 outlines an easy example: when snapshots satisfying $\Phi$ are exactly the timeslices from a higher-level model. Section 3.3.2 presents a more complicated example: capturing a large class of snapshots by capturing a single snapshot from a higher-level model.

### 3.3.1. The Easier Case

Section 2.3.1 showed that if a time model is doing its job, then its timeslices correspond exactly to the global states in the underlying computations. This correspondence assures a process that by obtaining one of these timeslices, it is taking a snapshot of a global state.

**The Special Timeslice Condition** Suppose that a process wants snapshots of global states that additionally satisfy some arbitrary predicate $\Phi$. In some sense, the process wants to pretend the only moments of simultaneity that exist are ones that satisfy $\Phi$. It would be convenient if a time model would express this pretense by admitting only the interesting global states as timeslices. To this end, we revise the earlier Timeslice Condition.

Suppose time model $\mathbf{M}$ on ground-level graphs generates the set $\mathcal{G}$, and predicate $\Phi$ on specifies which global states are of interest. Model $\mathbf{M}$ satisfies the *Special Timeslice Condition* for $\Phi$ iff for each $\beta \in \mathcal{G}$, $\mathbf{M}$ satisfies these requirements:

1. For each set $X$ of nodes in $\beta$, the following are equivalent statements:
   - $X$ minimally represents a global state $Y$ satisfying $\Phi$ in some ground-level graph $\alpha$ with $\mathbf{M}(\alpha) = \beta$.
   - $X$ is a timeslice in $\overline{\beta}$.
2. Each ground-level graph $\alpha$ with $\beta = \mathbf{M}(\alpha)$ and each global state $Y$ in $\alpha$ with $\Phi(Y)$ satisfy the statement:
   - if $\langle \mathbf{M}, \alpha \rangle(A) \cap Y \neq \emptyset$ for some node $A$, then some timeslice in $\overline{\beta}$ minimally represents $Y$.

We build time models by specifying nodes and precedence; timeslices follow from this basic construction. Hence the arbitrary predicate $\Phi$ must possess a fair degree of structure (and the model builder a fair degree of insight) in order to lead to a time model satisfying the Special Timeslice Condition. For this approach to work, the predicate must decompose to incomparability under some nicely behaved precedence relation.

**Example: No Messages In Transit** As an example, suppose a process wanted only global states where no messages are in transit. No send event preceding a snapshot $X$ has its receive event following $X$. Whether send and receive events themselves ought to be permitted to be members of such snapshots is another issue: if process $p$ is in the very act of sending or receiving a message, is that message in transit or not? We choose the cleanest approach: we permit the node before the send or after the receive, but not the transitional events themselves.

In the $\overline{\text{PARTIAL\_ORDER\_TIME}}$ model, a consistent cut $X$ with this no-transit property is genuinely a cut of the PARTIAL_ORDER_TIME graph: any path from $\perp$ to $\top$ must touch a node in $X$. (Other consistent cuts will not partition the graph, since message edges connect the past of the cut to the future of the cut.)

If process $p$ never wants to see a send event having occurred without also seeing the corresponding receive event, it essentially wants to pretend that corresponding send and receive events occur simultaneously. We formalize this pretense by defining the STRONG model, that adds an edge from each receive event to its send event. The STRONG model also must handle the case of unreceived messages. We take the approach of adding an edge from $\top$ to the send event of each unreceived message.[3] We then merge the atoms cyclic with $\top$ into $\top$. We define the STRONG_PARTIAL_ORDER model to be the composition:

$$\text{STRONG\_PARTIAL\_ORDER} \equiv \text{STRONG} \circ \text{PARTIAL\_ORDER\_TIME}$$

The new STRONG_PARTIAL_ORDER model exhibits much of the same theoretical structure as its original version. For example, (STRONG_PARTIAL_ORDER, TIMELINES) is still a consistent and independent parallel pair, so Theorem 2.1, Corollary 2.3, Theorem 2.4, Theorem 2.5, Theorem 2.6, and Theorem 2.7 all still hold.

Conveniently, the STRONG model prohibits send and receive events from STRONG timeslices, since these events are cyclic. This model thus obtains the desired result:

**Theorem 3.1** The STRONG_PARTIAL_ORDER model satisfies the Special Timeslice Condition for global states with no messages in transit.

*Proof* Suppose message $M$ from $p$ to $q$ is sent at $S$ and received at $R$. In $\overline{\text{STRONG\_PARTIAL\_ORDER}}$, any node preceding $R$ precedes any node following $S$. If message $M$ is sent by process $p$ at $S$ but is unreceived, then all nodes following $S$ in $\overline{\text{PARTIAL\_ORDER\_TIME}}$ are cyclic, and prohibited from timeslices. $\square$

Cycles of nodes become atomic units—if timeslices define perceivable moments, then cyclic sets can never be perceived as only partially complete. This observation suggests that transaction behavior may fit nicely into the framework of distributed time. (Section 7.2 discusses this topic further.)

Suppose the clock primitives for the global order extend to the STRONG version. A process may obtain a snapshot guaranteed to be a global state with neither send nor receive, nor message in transit, simply by carrying out a partial order time snapshot protocol, modified by substituting clock primitives for the new model.

Unfortunately, the cyclic STRONG_PARTIAL_ORDER model no longer satisfies the convenient information properties of Section 2.4.3. For example, suppose process $p$ executes a send event $S$

[3] According to this fix, all messages are received, only some messages are not received until the end of time.

of a message that is later received by process $q$ in receive event $R$. Process $p$ at $S$ may know that $R \longrightarrow S$ in $\overline{\text{STRONG\_PARTIAL\_ORDER}}$—however, process $p$ may not know anything else about $R$.

More precisely, $(\text{STRONG\_PARTIAL\_ORDER}, \text{TIMELINES})$ is neither strongly monotonic nor flow-supported (although it at least offers the advantage that processes can know that unmatched send events exist, so at least convergence can actually be determined). Specifying and implementing clocks for such cyclic models becomes rather tricky. In our original scheme, the ***PRECEDES*** primitive has two answers. However, the query "does $A$ precede $B$ in $\overline{\text{STRONG\_PARTIAL\_ORDER}}$?" admits a third answer: "not enough information yet."

The proper answer to the new query depends on *when* it is asked. Consequently, we need a third time model to express the degree of information flow for clock primitives.[4] This model should fit between the original partial order and its composition with STRONG; the abstraction hierarchy framework from distributed time theory provides the necessary machinery. This concept of *parameterized clocks* also extends to handle the case when various circumstances (such as faults or malice) prevent the convenient information assumptions in well-behaved partial order models from holding.

### 3.3.2. A Harder Case

Theorem 2.7 tells us that the adjusted timestamp vector of a node is a timeslice. One might conjecture that *any* nontrivial timeslice is the adjusted vector of one of its nodes. In fact, this conjecture is false—Figure 3.4 shows a counter-example. However, we can still establish something rather interesting: that adjusted vectors uniquely describe any nontrivial timeslice, and that we can obtain these descriptions by taking timeslices from higher-level time models.

For the snapshot problem, these results have two implications:

- A process might obtain snapshot $X$, but determine that a different one is necessary. The descriptions give a way of quickly specifying and obtaining a new one.
- A process can obtain a group of related snapshots by taking a single snapshot in the higher-level model.

(Charron-Bost [CB89] establishes a related result: in partial orders, a bijection exists between antichains (i.e., partial timeslices) and past-closed graph prefixes. Besides being developed in a different framework, our results are distinct because past-closed graph prefixes do not map injectively to timeslices.)

[4] Section 6.2 explores these issues further.

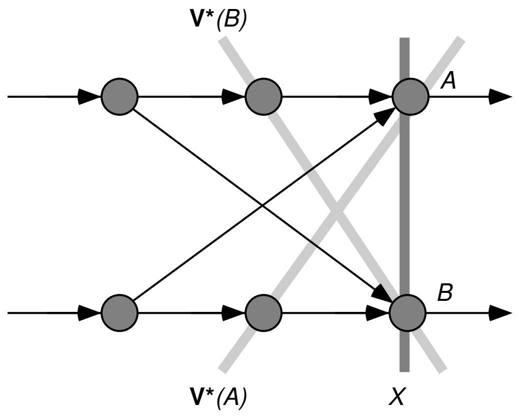


**Figure 3.4** Not all timeslices are adjusted vectors. Timeslice $X = \{A, B\}$ equals neither $\mathbf{V}^*(A)$ nor $\mathbf{V}^*(B)$. This example disproves the conjecture that Theorem 2.7 might describe all timeslices.

**The Description** Let $(\mathbf{M}, \mathbf{M}')$ be a Type 2 parallel pair (consistent and independent) that is also acyclic. Let $Y$ be a non-trivial partial timeslice: a non-empty set of non-extremal nodes that are mutually concurrent. Let $X$ be a non-trivial timeslice from $\overline{\mathbf{M}}$. A *generating subset* of $X$ is a $Y \subseteq X$ satisfying the equation:

$$X \quad = \quad \bigsqcup_{A \in Y} \mathbf{V}^*(A)$$

A *minimal generating subset* of $X$ is a generating subset of $X$, with the additional property that no proper subset is a generating subset of $X$.

The remainder of this section establishes two key results:

1. Each timeslice $X$ has a unique minimal generating subset.
2. A set $Y$ is a minimal generating subset of a timeslice iff $Y$ is a non-trivial partial timeslice in a higher-level model.

**The Blocking Model** Establishing these results hinges on drawing edges from a node $A$ to node $B$ when, in the transitive global order, the local predecessor of $A$ precedes $B$. We can express this enhancement itself as a model, BLOCKED. The BLOCKED model operates on a graph by copying it, and for each cross-process edge from node $A$ to node $B$, adding another edge the local successor of $A$ to $B$. The name of the BLOCKED model derives from its function (which we will demonstrate shortly). If $A \longrightarrow B$ in $\overline{\text{BLOCKED}} \circ \mathbf{M}$, then the presence of $B$ in an $\overline{\mathbf{M}}$ timeslice $X$ *blocks* node $A$ from being part of the minimal generating subset for $X$.

**Results** When an acyclic parallel pair $(\mathbf{M}, \mathbf{M}')$ is Type 2 (that is, consistent and independent), then $(\text{BLOCKED} \circ \mathbf{M}, \text{BLOCKED} \circ \mathbf{M}')$ is a parallel pair that is still independent, transitively-bounded, and acyclic. However, this new parallel pair may not necessarily be view-complete. Consider a graph from the PARTIAL_ORDER_TIME model. Suppose at process $p$, only a single state node $A$ separates a send node $S$ from a subsequent message node $B$. The BLOCKED model will slide the $S \longrightarrow R$ message edge up to $A \longrightarrow R$, and the edge from $A$ to $B$ may not necessarily have an externally equivalent node.

Precedence in BLOCKED expresses when the presence of one node in a timeslice forces the presence of another.

**Theorem 3.2** Suppose $(\mathbf{M}, \mathbf{M}')$ is an acyclic Type 2 (consistent and independent) parallel pair. If non-extremal distinct $A$ and $B$ satisfy $A \not\longleftrightarrow B$ in $\overline{\mathbf{M}}$, then

$$A \longrightarrow B \text{ in } \overline{\text{BLOCKED} \circ \mathbf{M}} \iff A \in \mathbf{V}^*(B)$$

*Proof* Let $A'$ be the node immediately preceding $A$.

Suppose $A \longrightarrow B$ in $\overline{\text{BLOCKED} \circ \mathbf{M}}$. Since $A \not\longrightarrow B$ in $\overline{\mathbf{M}}$, the precedence path from $A$ to $B$ must have been created by BLOCKED moving ahead the in-node of a message edge. This could only have made a difference when $A' \longrightarrow B$ in $\overline{\mathbf{M}}$. Thus $A' \in \mathbf{V}(B)$, hence $A \in \mathbf{V}^*(B)$.

Conversely, suppose $A \in \mathbf{V}^*(B)$. Then in $\overline{\mathbf{M}}$, $A' \longrightarrow B$ but $A \not\longrightarrow B$. Thus $A'$ must be the send event of a message that is received, so BLOCKED will copy and shift this message edge, giving $A \longrightarrow B$ in $\overline{\text{BLOCKED} \circ \mathbf{M}}$. $\square$

To obtain our main results, we first establish a condition on generating subsets.

**Theorem 3.3** Suppose $(\mathbf{M}, \mathbf{M}')$ is an acyclic Type 2 (consistent and independent) parallel pair. Let $X$ be a non-trivial timeslice from $\overline{\mathbf{M}}$. Any generating subset $Y$ of $X$ satisfies the statement:

$$\forall\, A \in X \quad \exists\, B \in Y \quad A \stackrel{=}{\longrightarrow} B \text{ in } \overline{\text{BLOCKED} \circ \mathbf{M}}$$

*Proof* Suppose the condition fails for $A \in X$. Then $A \notin Y$, and (from Theorem 3.2), for no $C \in Y$ is $A \in \mathbf{V}^*(C)$. Thus $A$ is not in the join of the adjusted timestamp vectors of $Y$, so $Y$ is not a generating subset. $\square$

We then construct a generating subset for a given timeslice.

**Theorem 3.4** Suppose $(\mathbf{M}, \mathbf{M}')$ is an acyclic Type 2 (consistent and independent) parallel pair. Let $X$ be a non-trivial timeslice from $\overline{\mathbf{M}}$. Let $Y$ be the set of $\overline{\text{BLOCKED} \circ \mathbf{M}}$ sinks in $X$.

$$Y \equiv \{A \in X \ : \ \forall\, B \in X,\ A \not\longrightarrow B \text{ in } \overline{\text{BLOCKED} \circ \mathbf{M}}\}$$

Then $Y$ is a generating subset of $X$.

*Proof* Let $Z$ be the join of the $\overline{\mathbf{M}}$ adjusted timestamp vectors of nodes in $Y$. For each process $p$, let $A_p$ and $B_p$ be the $p$ entries of $X$ and $Z$, respectively. We establish that $A_p = B_p$ by considering the two cases:

1. Suppose $A_p \in Y$. Let $C$ be another node in $Y$, let $C_p$ be the $p$ entry in its $\overline{\mathbf{M}}$ adjusted timestamp vector. If $A_p = C_p$ then (by Theorem 3.2), $A_p \longrightarrow C$ in $\overline{\text{BLOCKED} \circ \mathbf{M}}$, violating the construction of $Y$. If $A_p \longrightarrow C_p$ in $\overline{\mathbf{M}}$ but $A_p \neq C_p$, then $A_p \longrightarrow C$ in $\overline{\mathbf{M}}$, violating the fact that $X$ is a timeslice. Thus $A_p$ properly follows from each $C_p$, so $A_p = B_p$.
2. Suppose $A_P \notin Y$. By construction of $Y$, there exists a $C \in Y$ such that $A_p \longrightarrow C$ in $\overline{\text{BLOCKED} \circ \mathbf{M}}$. Since $A_p$ and $C$ are both members of timeslice $X$, Theorem 3.2 gives that $A_p \in \mathbf{V}^*(C)$. If $D \neq C$ in $Y$ has $\mathbf{V}^*(D)$ dominating $\mathbf{V}^*(C)$ at $p$, then $A_p$ precedes or equals the $p$ entry of $\mathbf{V}(D)$. Thus $A_p \longrightarrow D$ in $\overline{\mathbf{M}}$, violating the fact that $X$ is a timeslice. Thus $A_p = B_p$.

□

We use the condition from Theorem 3.3 to show that the generating subset from Theorem 3.4 is unique and minimal.

**Theorem 3.5** Suppose $(\mathbf{M}, \mathbf{M}')$ is an acyclic Type 2 (consistent and independent) parallel pair. Any non-trivial timeslice from $\overline{\mathbf{M}}$ has a unique minimal generating subset.

*Proof* Let $X$ be a non-trivial timeslice from $\overline{\mathbf{M}}$. Let $Y$ be the generating subset from Theorem 3.4. Theorem 3.3 provides two facts:

1. Any generating subset $Y'$ of $X$ has $Y \subseteq Y'$.
2. Any proper subset of $Y$ cannot be a generating subset of $X$.

Hence $Y$ is the unique minimal generating subset. □

Finally, we show that being a unique minimal generating subset is equivalent to being a partial timeslices under BLOCKED.

**Theorem 3.6** Suppose $(\mathbf{M}, \mathbf{M}')$ is an acyclic Type 2 (consistent and independent) parallel pair. Let $Y$ be a set of non-extremal nodes. $Y$ is the unique minimal generating subset of a timeslice in $\overline{\mathbf{M}}$ iff $Y$ is a partial timeslice in $\overline{\text{BLOCKED} \circ \mathbf{M}}$.

*Proof* Apply Theorem 3.5. The unique minimal generating subset of a timeslice $X$ is partial timeslice in $\overline{\text{BLOCKED} \circ \mathbf{M}}$. Any not-trivial partial timeslice $Y$ from $\overline{\text{BLOCKED} \circ \mathbf{M}}$ is the unique minimal generating set of

$$X = \bigsqcup_{A \in Y} \mathbf{V}^*(A)$$

□

**Implications** Suppose $(\mathbf{M}, \mathbf{M}')$ is an acyclic Type 2 (consistent and independent) parallel pair. A timeslice containing $k$ nodes in $\overline{\text{BLOCKED} \circ \mathbf{M}}$ is a shorthand representation of $2^k$ timeslices in $\overline{\mathbf{M}}$. If the clock primitives extend to handle queries about $\overline{\text{BLOCKED} \circ \mathbf{M}}$, then process $p$ can capture a large class of snapshots in $\overline{\mathbf{M}}$ by asking for a single one in $\overline{\text{BLOCKED} \circ \mathbf{M}}$. Naturally, we can establish dual results for adjusted rollback vectors.

## 3.4. Further Issues

To find out the state of the system, a process takes a snapshot. The fact that the snapshot does not necessarily describe a real state creates some complications. Section 3.4.1 considers the problem of resolving the parallax between inconsistent states, and Section 3.4.2 considers some areas for future work.

### 3.4.1. Resolving Parallax

The fact that standard snapshot protocols only guarantee consistent global states allows some potentially difficult *parallax* situations: two snapshots taken in the same computation may not mesh. Process $p$ may take snapshot $X$; process $q$ may take snapshot $Y$. The global states $X$ and $Y$ will each be genuine global states in some physical computation underlying the unfolding partial order—but they might not both appear in the same physical computation. For example, perhaps there exists a pair of nodes such that at instant $X$, one lies in the future and one in the past, but at instant $Y$ the positions are reversed. Figure 3.5 sketches a simple example.

Distributed time theory gives us a way to understand parallax; distributed time primitives form the basis for a simple protocol to resolve parallax.

**Why Does Parallax Occur?** Timeslices in a computation graph form a sublattice of a divided hypercube in $k$ dimensions, where $k$ is the number of messages. We obtain this hypercube by drawing nodes at each point with coordinates from the set $\{0, 1, 2\}$, and drawing an edge from node $N_1$ to node $N_2$ when they differ in exactly one coordinate, and the $N_2$ value there is one

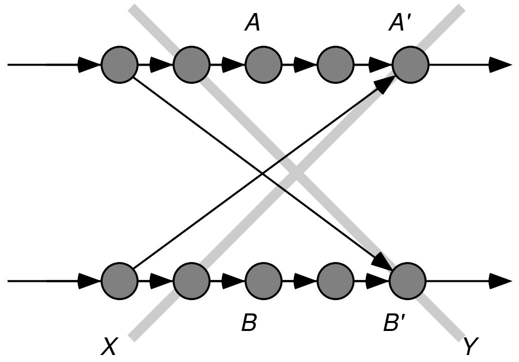


**Figure 3.5** Parallax occurs when snapshots appear to be inconsistent. Consider the positions of nodes $A$ and $B$ with respect to timeslices $X = \mathbf{V}^*(A')$ and $Y = \mathbf{V}^*(B')$. Both timeslices represent logically simultaneous instants. However, at instant $X$, $A$ has occurred but $B$ has not, while at instant $Y$, $B$ has occurred but $A$ has not.

greater than the $N_1$ value. Each dimension represents a message $M$, and the coordinate values represent that message's status: unsent, en route, or received.

Under the simplifying assumption that no two message events occur simultaneously, each source-sink traversal of this graph corresponds to a real time trace of a computation generating this graph.

Timeslice precedence captures computation paths. Any timeslice follows or equals the minimal timeslice and precedes or equals the final; timeslices $X$ and $Y$ satisfy $X \prec Y$ when a computation path exists from $X$ to $Y$. In general, the timeslice lattice is not a straightline graph; timeslice precedence is a not a total order. Parallax follows from the existence of timeslices that are "concurrent," in the sense that they are incomparable under the $\prec$ order. (This structure on timeslices is reminiscent of the time model structure on nodes.)

**Resolving the Inconsistency** Once again, lattices come to the rescue. Suppose $X$ and $Y$ are timeslices in a view-complete, transitively bounded graph. We already know that $X$ and $Y$ are consistent cuts. We can directly establish that the set of all timeslices $Z$ such that $Z \subseteq X \cup Y$ forms a finite lattice: this set is closed under $\sqcap$ and $\sqcup$.

To simplify presentation, we assume without loss of generality that $X$ properly dominates $Y$ at each process. (In the general case, we would take the join and meet of the two timeslices, and restrict our attention the processes where the values differ.)

Clock primitives provide a basis for constructing a simple graph that expresses the sublattice of timeslices contained in the set $X \cup Y$. Construct the graph $G$ by creating one node for each process, and drawing an edge from process $p$ to process $q \neq p$ when $\pi_p X \longrightarrow \pi_q Y$. If there are $n$

processes, this construction takes $O(n^2)$ operations; access to vector operations and vector clocks can bring this down to linear.

The graph $G$ concisely expresses the sublattice. To obtain a timeslice $Z$, a process follows these steps:

1. Choose a node $p$ in $G$.
2. Select either $\pi_p\, X$ or $\pi_p\, Y$ for $Z$.
   (a) If $\pi_p\, X$, then for each node $q$ that (transitively) follows $p$, select $\pi_q\, X$.
   (b) If $\pi_p\, Y$, then for each node $q$ that (transitively) precedes $p$, select $\pi_q\, Y$.

   Delete the nodes for which we just selected values.
3. Repeat until all entries of $Z$ are chosen.

### 3.4.2. Future Work

Fully generalizing the protocols and primitives requires confronting unresolved obstacles—and also suggests interesting new structures. These issues provide topics for further research.

**Convergence and Knowability** The snapshot protocols of Section 3.2 implicitly assume that a given process has sufficient information to decide clock queries. As Section 2.4.3 discusses, this implicit assumption may fail. We present three such scenarios:

- cyclic models;
- clock implementation where faults or malice or efficiency prevent complete knowledge; and
- snapshot queries about recent nodes.

These scenarios require more active consideration of *convergence*: when information about the past of a node catches up with the future of a node. These scenarios also require a more detailed exploration of the knowability issues of Section 2.4.3.

**Observation Effects** Consider again a snapshot protocol that assumes all processes have heard of node $A$. If a precedence path does not exist from node $A$ to process $q$, should the arrival of the snapshot query establish one? That is, how should the act of examining the computation interact with the computation itself?

Suppose a process $p$ uses a snapshot protocol determines if a global state satisfying a particular property exists. Process $p$ obtains its result at node $A$. What should happen to process $p$ if this query fails—but later another process rolls back and changes history? If a suitable timeslice now exists, the answer process $p$ received is incorrect, so node $A$ now depends on incorrect data. The need to express this dependence suggests that, for time models expressing perception, every node examined in a snapshot search should precede the response node. Is this synchronization desirable? Should we use the abstraction hierarchy techniques of distributed time to capture this influence in a higher-level model? What should happen in snapshot protocols where the existence of a snapshot does not change, but the actual snapshot returned does?

**Global States and Guaranteed Pasts** The abstract computation graph describing a computation induces a lattice of timeslices. The actual physical computation that occurs determines which path through this lattice the system actually takes. The limitations of distributed asynchrony prevent the system from ever finding out which path this is; as a consequence, taking a snapshot to determine some property of the computational path is difficult in the general case.

Recent work in global state detection (e.g., [CoMa91, MaNe91, MaSa91, ToGa93, GaWa94]) has explored this area, both by explictly searching the timeslice lattice and by defining classes of predicates (in particular, by relaxing the stability requirement) that may be examined by snapshot techniques. Integrating this work into our framework would be an interesting area for future research.

Another research direction comes from the observation that although a process $p$ cannot find a recent global state guaranteed to have occurred, it may have some use for a concise set $\{X_1, ..., X_k\}$ of recent global states, such that one of them definitely occurred. In the general case, these sets will be minimal graph-cuts of the timeslice lattice. With some simplifying assumptions, these sets have a more familiar form: a "maximal set of $X_i$ such that no $X_i \prec X_j$"—that is, a timeslice of timeslices. Can we directly generalize distributed time to build higher-level time models whose nodes represent timeslices in lower-level time models?

# Chapter 4

# Optimistic Rollback Recovery

## 4.1. Overview

### 4.1.1. The Basic Problem

The problem of *rollback* arises when a process $p$ in a distributed computation rolls back[1] to a previous state. This problem typically appears when providing fault tolerance for distributed computation: physical failure of process $p$ might force $p$ to roll back, since the most recent state of $p$ that can be recreated may not necessarily be the state $p$ held when it failed. Some applications might also permit rollback in non-failure scenarios; for example, process $p$ might roll back its computation if $p$ discovers that critical input data was corrupted. For clarity of presentation, this chapter assumes that rollback occurs because of process *failure*; however, our techniques also apply to the more general case.

**Failure and Recovery** Suppose process $p$ fails and recovers by restarting itself from an earlier, saved state. All activity by process $p$ since it first passed through this restored state has been lost. (Figure 4.1 illustrates this scenario.) If the original execution of this lost activity affected no process other than $p$, then the loss of this activity can affect no process other than $p$. For example, 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 also be lost for other reasons). In this case, the rest of the system may proceed without ever knowing about process $p$'s failure and recovery.

**Dependence** However, suppose the lost activity at process $p$ included the send event of a message that was received by process $q$. Then the state of process $q$ depends on activity at process $p$ that led up to the sending of that message, but some of this activity—including the send event itself—has been rolled back due to the failure. Process $q$ has received a message that, in process $p$'s view after recovery, was never sent. The computation at process $q$ after this receive event must

[1] "Rollback" is the noun; "roll back" is the verb phrase.

also be rolled back in order to restore the system to a consistent state. (Figure 4.2 illustrates this scenario.)

**Transitive Dependence** Distribution and asynchrony may make the situation even more complicated. For example, suppose process $q$ receives a message from process $p$, but $p$ then rolls back its send event. However, before learning of process $p$'s rollback, process $q$ sends a message to another process $r$. Then process $r$ depends on computation that has been rolled back—even though process $r$ may not have directly received a message from $p$. (Figure 4.3 illustrates this scenario.)

**Orphans** In rollback recovery, an *orphan* is an event or state[2] that causally depends on (or equals) computation that has been rolled back. This terminology and the use of PARTIAL_ORDER_TIME to express potential causality simplifies the above discussion. In Figure 4.1, nodes $A_2$ through $A_5$ are orphans, since they depend on prior computation that has been rolled back. Figures 4.2 and 4.3 show orphans at other processes: the rollback in Figure 4.2 causes nodes $B_3$ and $B_4$ to become orphans; the rollback in Figure 4.3 causes nodes $C_3$ and $C_4$ to become orphans, along with nodes $B_3$ through $B_6$.

### 4.1.2. Further Issues

**Delayed Messages** Rollback can also give rise to some pathological scenarios. For example, the lost activity at process $p$ may include the send event of a message to process $q$ that, due to network delays, does not arrive at $q$ until after $p$ has rolled back and the system appears to have recovered. Accepting this message may cause process $q$ to become an orphan, as Figure 4.4 shows. However, addressing this problem by blindly discarding *all* messages sent before rollback may lead to discarding valid messages, as Figure 4.5 shows.

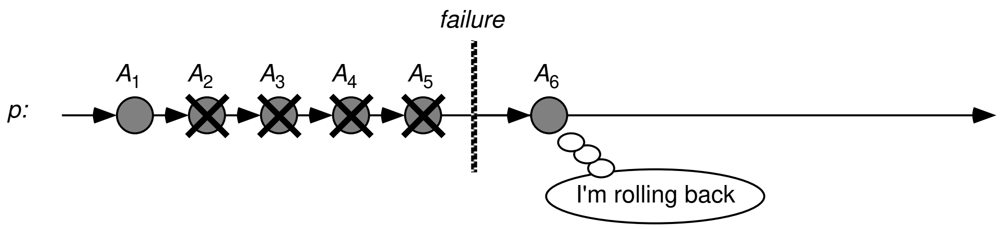


**Figure 4.1** The problem of rollback arises when a process fails and restarts from an earlier state. Here, process $p$ fails; it recovers at state $A_6$ by restarting from the state from $A_1$. A large "X" marks each node that has been rolled back.

[2]The literature sometimes extends the notion of orphan to include processes (whose current state is an orphan) and messages (whose send events are orphans).

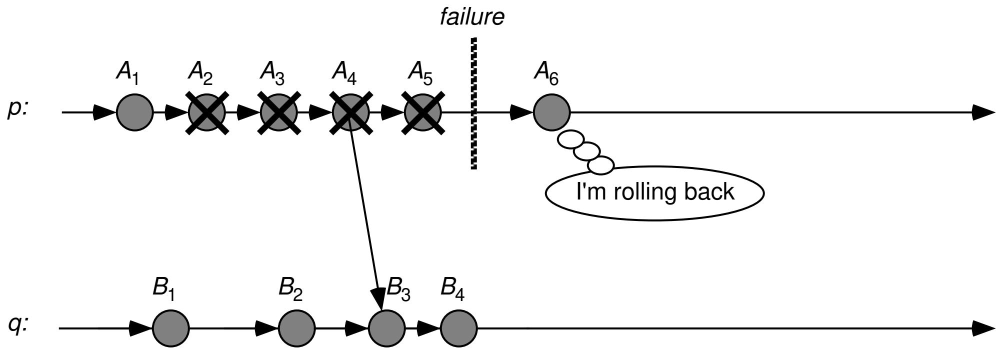


**Figure 4.2** The problem of rollback becomes complicated when another process depends on events and states that the failed process has rolled back. Here, process $p$ has failed and restored state $A_1$. However, process $q$ at $B_3$ has received a message whose send event has been rolled back. Hence, $B_3$ and $B_4$ depend on computation that has no longer happened. Consequently, $B_3$ and $B_4$ should also be rolled back.

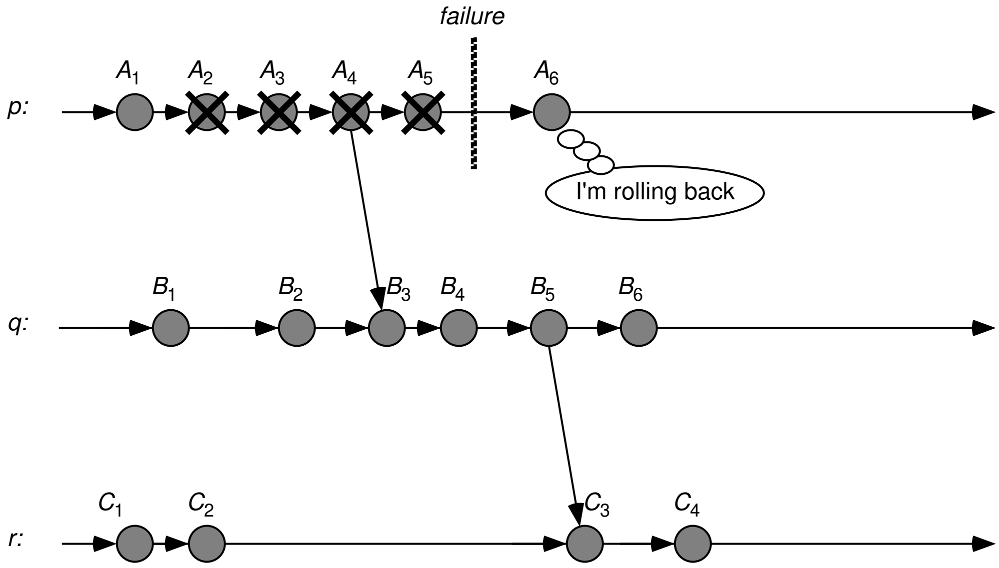


**Figure 4.3** Transitive dependence further complicates rollback. Process $p$ has failed and rolled back. Process $r$ depends on computation at process $q$ that in turn depends computation that process $p$ has rolled back. Thus, the failure at process $p$ makes it necessary for process $r$ to roll back, even though process $r$ has never received any message directly from process $p$.

**Rollback with Modified Replay** After a process $p$ rolls back and restores some earlier state $A$, it has a number of options. Process $p$ could re-execute the same computation beginning at state $A$ that it originally executed. Alternatively, process $p$ could intentionally execute a computation beginning at state $A$ that differs at some point from its original execution; this approach is termed *rollback with modified replay*. For example, perhaps process $p$ alters its activity in order to avoid the conditions that led to the failure, or perhaps process $p$ rolled back explicitly to take another course of action (rather than to recover from failure).

**Concurrent Rollbacks** The possibility that rollback may occur asynchronously raises some questions:

- Multiple processes could initiate rollback concurrently—perhaps to recover from the same failure, or perhaps to recover from different failures. Can the recoveries be merged? If not, which recovery is performed first? Do the others still need to be performed?
- A process might fail and initiate rollback before recovery from some earlier rollback at another process is complete. Can these two recoveries proceed concurrently?

### 4.1.3. Rollback Recovery Protocols

Beyond recovering the system when one or more processes fail, rollback recovery protocols have several implicit goals:

- minimizing the computation lost due to failure (e.g., the interval from the original execution of the restored state to the failure);

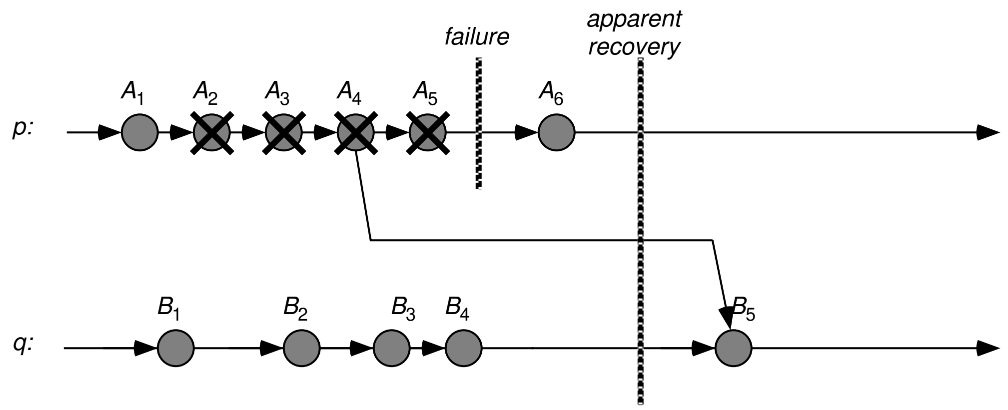


**Figure 4.4** Successful recovery protocols need to consider delayed messages. Believing the system is recovered and blindly accepting this message will cause process $q$ to become an orphan.

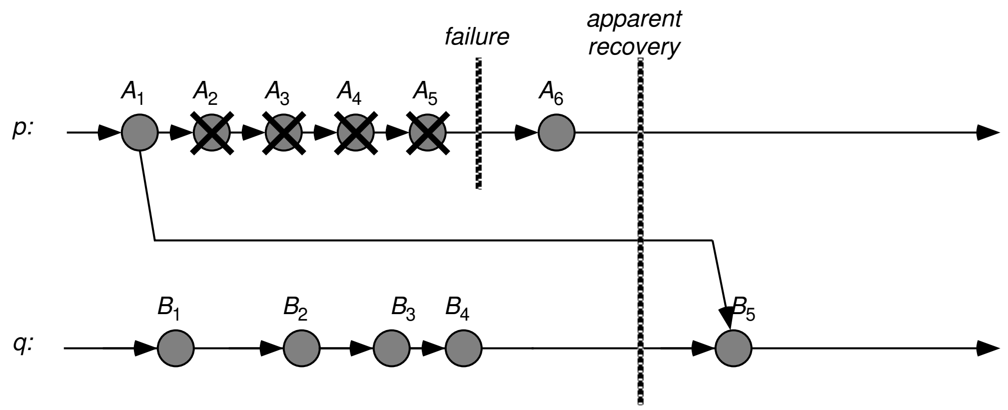


**Figure 4.5** Fixing the problem of Figure 4.4 by discarding *all* messages sent before rollback can lead to discarding valid messages. Here, process $q$ should accept the delayed message—even though the message was sent by process $p$ before recovery.

- minimizing the computation wasted due to rollback (e.g., the number of surviving processes that must be rolled back, the number of times they must roll back, the delay before which they begin their rollbacks, and the amount of rolled-back computation that did not depend on computation lost due to the failure); and
- minimizing the overhead of the recovery protocol during failure-free execution.

**Checkpointing** One approach to recovery is based on *checkpointing*: processes periodically checkpoint their local state to stable storage. Rollback recovery protocols based on checkpointing (e.g., [BhLi88, BCS84, Ci89, EJZ92, KoTo87, LeBh88, LNP90]) organize local checkpoints into system-wide global checkpoints, and recover from failure by rolling back all processes to one of these *recovery lines*. Protocols use varying degrees of synchronization in establishing global checkpoints. Using too little synchronization permits pathological scenarios where a single failure could lead to the *domino effect* [Ra75, Ru80] in which all processes are forced to roll back to their initial states regardless of the amount of progress made before the failure. Careful use of synchronization avoids the domino effect. Nevertheless, checkpointing-based recovery wastes computation by rolling back beyond the theoretical minimum. Processes that have dependence on the failure must (in general) roll back computation that occurred before dependence was established; processes that have no dependence may also need to discard their progress and roll back.

**Message Logging and Replay** Another approach to recovery is based on *message logging* and *replay*. Processes log their received messages and occasionally checkpoint their local state. Consequently, processes may recreate any past state—not just the ones saved as checkpoints—by restoring an earlier checkpoint and replaying the received messages from the log. This approach offers two significant advantages:

- Logging a message is cheaper than recording a checkpoint.
- Message logging reduces wasted computation, since surviving processes only roll back computation that depends on the computation lost to failure.

*Pessimistic* rollback protocols (e.g., [BBG83, BBGH89, ElZw92, JoZw87, PoPr83]) synchronize message logging with the underlying computation. For example, a process may not proceed beyond the receipt of a message until that message is successfully logged to stable storage. Pessimistic protocols simplify recovery, since a surviving process never depends on computation lost due to process failure. However, the logging synchronization needed by pessimistic protocols leads to decreased performance [Jo89].

*Optimistic* rollback protocols (e.g., [Jo89, JoZw90, Jo93, PeKe93, SiWe89, StYe85, Zw88]) buffer received messages in volatile storage, and asynchronously log them to stable storage. A process may proceed beyond the receipt of a message before the message is successfully logged. These protocols optimistically bet that a process will not fail before the logging of its received messages is complete. However, a failure at a process that has not finished logging may create orphans at other processes. Consequently, optimism complicates recovery, since protocols must be able to detect and eliminate orphans throughout the system. However, optimistic protocols are cheaper during failure-free operation.

### 4.1.4. Asynchronous Optimistic Rollback Recovery

This chapter uses the framework of distributed time to consider optimistic rollback recovery. Optimistic protocols already have low failure-free overhead. Our tools for time abstraction allow us to improve on previous work by simplifying the task of recovery.

Most optimistic rollback rollback protocols require synchronization in recovery. However, the more decentralized a distributed protocol is, the better its potential for exploiting the advantages of distribution (e.g., concurrency) and being robust against the disadvantages (e.g., asynchrony and unreliable networks). Strom and Yemini [StYe85] initiated the area of optimistic rollback recovery and presented the most asynchronous protocol prior to ours.

**Strom and Yemini** In the Strom and Yemini protocol, processes use timestamp vectors to track dependency. When a process rolls back, it begins a new *incarnation* and sends announcements to the other processes. (These announcement messages are not part of the failure-free computation, and thus do not carry dependency.) When a process receives a rollback announcement, it uses its timestamp vector to determine if it is currently an orphan; if so, this process rolls back to its maximal non-orphan state by restoring an old checkpoint and replaying its received messages until a message is reached whose send event is an orphan. A process receiving a rollback announcement also saves the incarnation start information from the announcement for use in subsequent vector sorting; delayed announcements may require non-faulty processes to block.

Strom and Yemini do not require processes to synchronize during recovery. This asynchrony offers several advantages:

- processes can recover without the delay of synchronization;
- recovery from concurrent failures can proceed concurrently; and
- once initiated, recovery can sometimes proceed despite network partitions.

However, the Strom and Yemini protocol has a significant disadvantage: a single failure at one process may lead to $\Theta(2^n)$ rollbacks (where $n$ is the number of processes in the system). This behavior occurs because an orphan state at a surviving process $r$ may depend on the lost computation through multiple paths: directly from the failed process, and indirectly through intermediate processes. Even with its assumption of FIFO message ordering, Strom and Yemini's protocol may generate failure announcements in such a way that process $r$ rolls back in response to the rollbacks of intermediate processes, and then in response to the rollback of the failed process. Figure 4.6 shows a simple scenario in which process $r$ rolls back twice in response to a single failure at process $p$; Section 4.3.6 presents an inductive construction showing an exponential number of rollbacks.

**Distributed Time** The framework of distributed time allows us to talk about time abstraction on *multiple* levels:

- We can use one level of partial order time to describe potential causality in the failed computation.
- We can use another level of partial order time to describe potential knowledge in the recovery computation.

Using timestamp vectors for both levels allows us to build an orphan test exploiting all potential information. This ability directly leads to an optimistic recovery protocol that provides completely asynchronous recovery but requires surviving processes to roll back at most once in response to the failure of any process. (Figure 4.7 provides a rough sketch.)

Our new recovery protocol improves on Strom and Yemini's work by reducing the worst case from exponential to constant, and improves on other optimistic recovery protocols by requiring no synchronization during recovery. Our protocol also provides additional flexibility: messages need not be FIFO, and no extra messages need to be transmitted. Further, developing our protocol in the framework of distributed time allows transparent integration with other applications based on partial order time, and transparent protection against clock-based security and privacy attacks. Like other optimistic approaches, our protocol does not require any process to roll back computation that does not depend on lost computation at the failed process. Table II presents a table comparing our protocol to three principal optimistic rollback protocols, and to a sample checkpointing protocol. (Section 4.3.6 provides a fuller discussion of these protocols.)

**This Chapter** Section 4.2 discusses the relevance of distributed time to rollback recovery. Section 4.3 presents our new protocol. Finally, Section 4.4 uses distributed time to derive a general framework for rollback problems and recovery protocols. (Chapter 5 and Chapter 6 will explore the security issues.)

We presented a preliminary version of our new protocol in an earlier publication [SJT94].

### 4.1.5. Assumptions

**Recoverability** Section 4.1.1 and the remainder of this chapter implicitly assume *complete recoverability*: each state at every non-faulty process can be recovered.

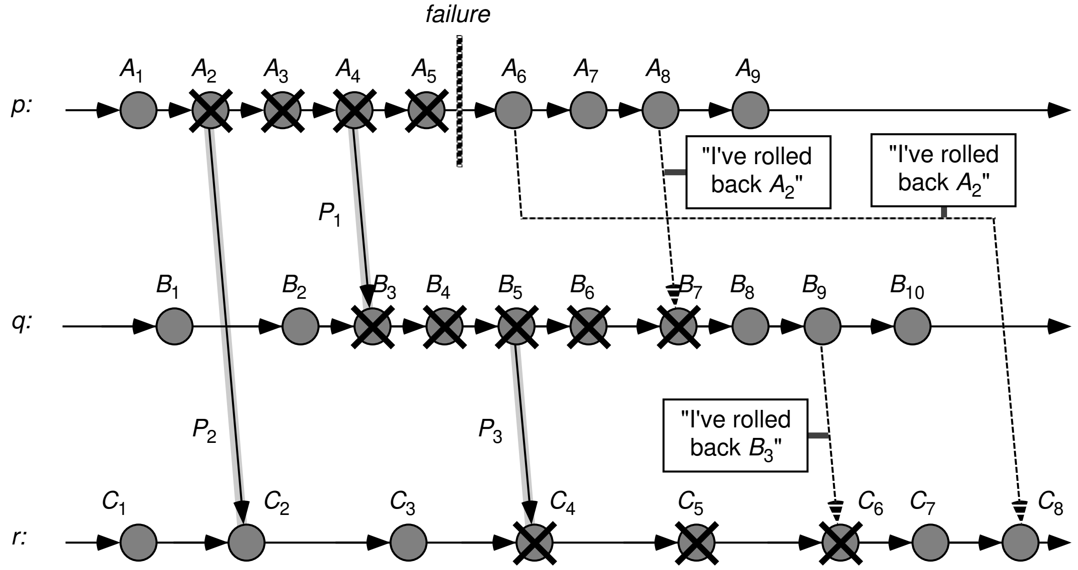


**Figure 4.6** The Strom and Yemini protocol may cause surviving processes to roll back multiple times in response to a single failure. This diagram shows how one failure at process $p$ causes process $r$ to roll back twice. Process $p$ fails and rolls back $A_2$ through $A_5$. This failure makes process $q$ an orphan (since $q$ depends on the lost computation via path $P_1$) and also makes process $r$ an orphan (directly through path $P_2$, and indirectly through paths $P_1$ and $P_3$). When process $q$ receives $p$'s announcement about $A_2$, $q$ rolls back to its most recent state that does not depend on $A_2$. Unfortunately, $q$'s announcement may arrive at process $r$ before $p$'s announcement does. When process $r$ receives $q$'s announcement about $B_3$, $r$ rolls back to its most recent state that does not depend on $B_3$. Process $r$ does not know that its restored state is still an orphan until after the delayed $p$ announcement arrives (at $C_8$).

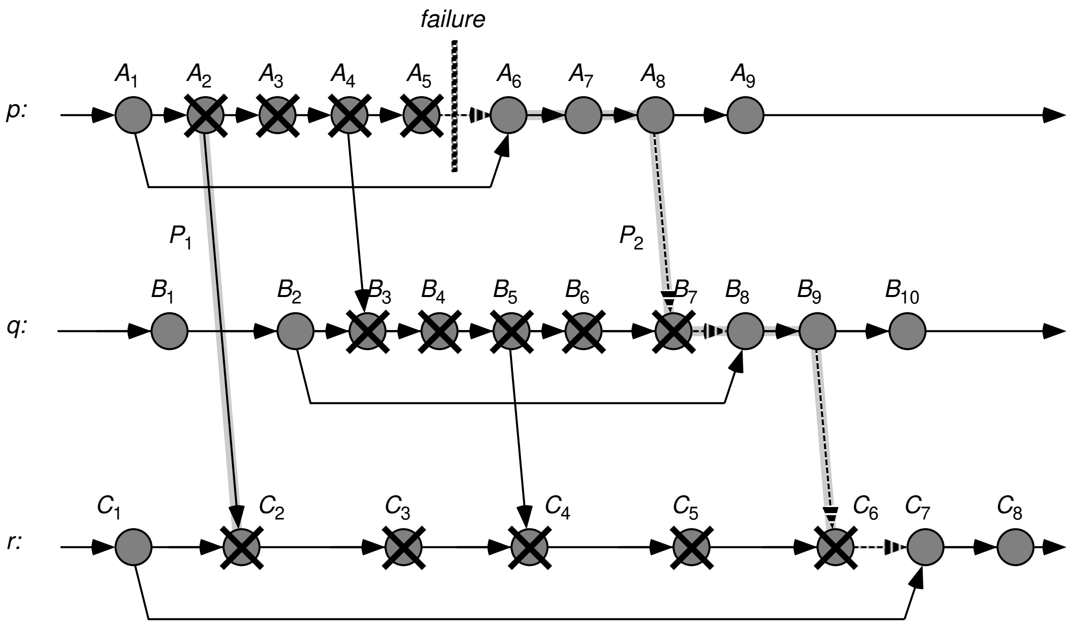


**Figure 4.7** Using two levels of partial order time allows asynchronous recovery while avoiding the problem of multiple rollbacks. This diagram roughly sketches the principles involved in our new protocol. Solid arrows indicate both potential causality in the failed computation and potential knowledge in the recovery computation. Dashed arrows indicate only potential knowledge in the recovery computation. As in Figure 4.6, process $p$ fails and rolls back $A_2$ through $A_5$. Thus $A_6$ logically succeeds $A_1$ rather than $A_5$; hence the dashed edge from $A_5$ to $A_6$ and the solid edge from $A_1$ to $A_6$. Process $p$ sends an announcement to process $q$; since this announcement is not part of the underlying computation, we use a dashed edge. Process $q$ rolls back, and sends an announcement that process $r$ receives at $C_6$. Via dependence path $P_1$, $C_2$ depends on lost $A_2$ and is an orphan. However, via *knowledge* path $P_2$, process $r$ at $C_6$ is potentially aware that $A_2$ has been lost. Comparing timestamp vectors across partial orders allows process $r$ at $C_6$ to determine that $C_2$ is an orphan. Thus, unlike Figure 4.6, process $r$ rolls back far enough the first time.

| | Strom and Yemini | Koo and Toeug | Johnson and Zwaenepoel | Peterson and Kearns | Distributed Time Protocol |
|---|---|---|---|---|---|
| Assumptions | FIFO | FIFO | None | None | None |
| Asynchronous recovery? | Partially | No | No | No | Yes |
| Concurrent recovery? | Yes | No | No | No | Yes |
| Maximum rollbacks at one process from one failure? | $\Theta(2^n)$ | 1 | 1 | 1 | 1 |
| Entries in timestamps | $O(n)$ | $O(1)$ | $O(1)$ | $O(n)$ | $O(n)$ |

**Table II** The distributed time protocol for rollback recovery compares favorably to previous work in many aspects. Its principal drawback is timestamp size, since the protocol requires vector clocks for two levels of partial order time.

As we have discussed, optimistic recovery protocols typically provide complete recoverability of states at non-faulty processes by asynchronously taking local checkpoints at each process, and by asynchronously logging the messages each process receives. To restore a state $A$, a process $p$ rolls back to its most recent checkpointed state preceding or equaling $A$, and then re-executes its computation (replaying received messages from its log) until it reaches state $A$. This approach requires that process execution be *piecewise deterministic* (that is, deterministic between message receive events). When a process fails, it may lose recent logging information, since logging proceeds asynchronously with the underlying computation. (This fact distinguishes optimistic recovery from pessimistic recovery.)

This logging and replay approach can be extended to nondeterministic execution by having processes treat nondeterministic influences as incoming messages [ElZw92, Jo93]. For example, if a process state enables a transition to multiple states, the process might asynchronously log the index of the choice that is made. Our automata model of Section 2.2.1 permits each state transition at a process to be non-deterministic. The extended logging-and-replay approach would provide complete recoverability for our model, and for the protocols presented in this chapter. With some modifications, the simpler piecewise deterministic model (with its simpler logging scheme) also suffices. Section 4.3.5 discusses these modifications.

**Commitability** The examples in Section 4.1.1 also implicitly assume that any state or event can be rolled back. Achieving this assumption in practice is difficult. Rolling back arbitrary nodes may be impossible: for example, interaction with the outside world may lead to activity (e.g., launching a missile) that cannot easily be undone. Providing the ability to roll back arbitrary nodes may be expensive, since processes can then never throw out any checkpoints or log data. Rollback recovery for fault tolerance requires only keeping sufficient data to restore the maximal recoverable system state; however, the question still arises of determining which data this is.

Due to these problems, realistic protocols also need to consider stability and commitability. A state or event is *stable* when it has been successfully logged to stable storage; a state or event is *commitable* when it will never be rolled back [JoZw90]. If rollback only occurs to recover from process failure, then a node is commitable when every node in its timestamp vector is stable. When a stable node $A$ becomes commitable at a process, the process will never need to recreate an earlier node, and thus may discard all earlier log data (except that which is necessary to recreate $A$). Furthermore, activity with potentially permanent side-effects may proceed safely. For space efficiency, recovery protocols should allow processes to discard unnecessary data. For example, in the Strom and Yemini protocol, each process maintains a vector indicating the logging status of the other processes, and uses this vector to determine when node is commitable (and thus when previous log data may be discarded).

For clarity of presentation, our protocols do not address the issue of commitability. However, a vector solution similar to Strom and Yemini's easily incorporates into our framework.

**Failure Detection and Reconfiguration** We also do not consider mechanisms for processes to detect failure, nor for selecting the physical site where a failed a process should restart. (However, since our framework provides tools for hierarchies of abstraction, it may simplify many issues in process/processor mapping.)

## 4.2. Rollback and Distributed Time

This section applies the framework of distributed time to the rollback problem. Section 4.2.1 discusses the relevance of distributed time. Section 4.2.2 introduces the idea that processes performing rollback have two levels of consciousness—the system level of a process implements the user level. Section 4.2.3 builds a time model for the computation performed by the system level of processes; Section 4.2.4 builds a time model for the computation performed by the user level. Section 4.2.5 introduces some notation for mapping between the system level and the user level. Section 4.2.6 discusses the mechanics of retroactive change—how rollback protocols might alter the computation in progress. Section 4.2.7 discusses how the failure-free virtual computation arises from the user-level computation.

### 4.2.1. The Relevance of Distributed Time

Optimistic rollback recovery changes the history that user processes perceive. Distributed time provides abstraction tools that apply on several levels:

- **Dependence on Failure** Optimistic rollback recovery permits orphans to exist at processes other than the process that failed. If we have accurate timestamps from real time and no prior failures have occurred, then we might try using real time to test for orphans: anything that occurred after the failure. This test will detect all orphans, but has some substantial flaws. First, using real time is wasteful—many states that did not depend on the failure will be rolled back unnecessarily. Further, as Section 2.5.1 observed, this approach easily breaks down in realistic scenarios:

  - Not all processes may receive word of the failure simultaneously. Real time does not distinguish between states that depend on the failed state, and states that occur after recovery from dependence on the failed state.
  - Suppose a failure occurs at process $q$ before recovery from a failure at process $p$ is complete. Real time is not sufficiently articulate to express the resulting nuances. For example, paerhaps the current node at process $r$ was an orphan due to both failures, but process $r$ rolled back in response to process $p$'s failure. Is the current node at $r$ still an orphan due to $q$'s failure?
  - Some state restoration mechanisms require a process to re-execute events. Such a re-executed event may have a later timestamp than events that it influenced.

  The nuances of dependence are better captured by a partial order time model (possibly, as the last example illustrates, a flow-virtual model that does not follow directly from the real timelines at processes).

- **The Failure-Free Virtual Computation** Recovery from failure changes the underlying computation. The failure-free virtual computation that appears to have happened after recovery is complete is also expressed by a partial order—but this partial order differs from the one tracking dependency in the failed computation. For example, suppose we wanted to use the results of Chapter 3 to take a snapshot from the logical past of the virtual computation, or suppose we need to recover from a second failure. Applying protocols based on partial order time to the failure-free virtual computation requires access to this second partial order.

- **The Recovery Computation** The recovery computation is itself a distributed computation, different from both the original failed computation and the failure-free virtual computation. The recovery computation is expressed by a *third* partial order—one that would be constructed by an external observer who did not know that the system was performing a recovery algorithm. Reasoning about the progress of recovery (e.g., "who knows about what rollbacks?") and integrating the recovery computation with other applications (e.g., snapshots) requires using the recovery partial order.

**A Single Framework** Distributed time provides the tools to represent computation at all the levels of abstraction that arise when considering rollback recovery. The remainder of Section 4.2 will develop these levels of abstraction.

## 4.2.2. Bipartite Processes

The most straightforward representation of the state at a process is as a set of bits. However, our distributed time theory introduces the abstraction that process clocks track temporal relations. A firewall limits the interaction between a process and its clock to formal queries and responses. Figure 4.8 illustrates this view.

Both the decision to roll back and the inability to directly control the state of the network highlight the need for managing rollback at a process. This need introduces a second firewall inside a process: in order for a simple process of the form of Figure 4.8 to exist in the virtual computation, it must have with it another process that handles the management. Figure 4.9 illustrates this revised view: an *implementing process* supports the *implemented process*.

The implemented process—the state and action of the process above the firewall of Figure 4.9—is the *user* level of that process. The state and action of the entire process is the *system* level. The *management* state is the portion of process state exclusively part of the system level.

Defining rollback requires the use of the Figure 4.8 view of a process. Implementing rollback requires the use of the Figure 4.9 view. Multiple levels of abstraction at a process mesh nicely with multiple levels of abstraction and time (as the following sections discuss).

## 4.2.3. The System Computation

An external observer who did not know that failure and recovery was occurring would be oblivious to the process structure of Figure 4.9. This observer's point of view would provide no distinction between the implemented process and the implementing process.


**Figure 4.8** Encapsulating time services into a clock module revises our view of process: we now now can think of the internal state of the process as separate from the state of the process clock.

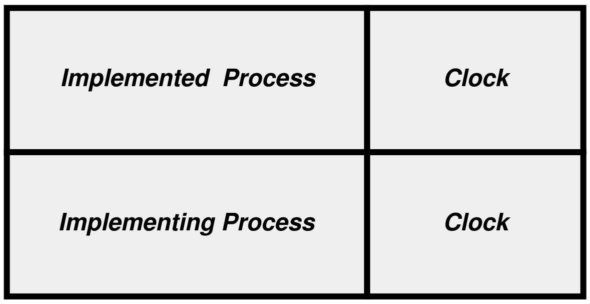


**Figure 4.9** Managing the virtual existence required by rollback introduces another firewall: between the implemented process and the implementing process.

We define

SYSTEM_PARTIAL_ORDER

to be the time model obtained by applying PARTIAL_ORDER_TIME without distinguishing process levels. (That is, we ignore the firewall between the implemented process and the implementing process.) Similarly, we define

SYSTEM_TIMELINES

to be the model obtained by applying TIMELINES without distinguishing process levels.

We use the notation $\mathbf{V}_{\text{sys}}(A)$ to indicate the timestamp vector of a node $A$ a graph from SYSTEM_PARTIAL_ORDER.

The pair of models (SYSTEM_PARTIAL_ORDER, SYSTEM_TIMELINES) forms a Type 3 parallel pair—consistent, independent, and strongly monotonic. The possibility that process failure may disrupt information flow prevents the pair from being flow-supported, and thus from being Type 4. Clocks for SYSTEM_PARTIAL_ORDER must be designed around this limitation. Section 4.3.4 considers these issues.

### 4.2.4. The User Computation

In this section, we build a USER_PARTIAL_ORDER model to express the user-level computation performed by the user levels of processes. This construction is a bit more complicated than the construction in Section 4.2.3: we obtain the USER_PARTIAL_ORDER model as the composition of an IMPLEMENT model with the SYSTEM_PARTIAL_ORDER model.

**Implementation** The system level of process computation implements the user level. Building the user model requires defining this implementation.

In terms of our time models, a user process does four things:

- It holds a *state*.
- It performs *internal computation*.
- It *sends* a message.
- It *receives* a message.

The system-level computation of a process implements these four things:

- A user *state* node consists of a maximal sequence of system-level state nodes and system-level transition nodes that do not change the user state.
- A user *internal computation* transition occurs when the user state changes, due to an implemented internal transition.
- A user *send* event consists of a change in user state along with the transfer of the message to the management state (the virtual *send*); the system state subsequently sends the message out as part of a system message. (The potential exists here for the system process to suppress the message.)
- A user *receive* event occurs when the system process receives a user message and decides to pass it on to the user-level process. In the virtual *receive*, the management state changes (to reflect the dequeuing of the message) and the user state changes (to reflect the *receive*).

The system-level computation at a process can also perform *rollback*: a discontinuous change in the user state.

**Nodes** The preceding discussion of how the system-level implements the user-level directly tells what nodes the IMPLEMENT model should produce when it is applied to a SYSTEM_PARTIAL_ORDER graph, and what these nodes should represent. This mapping is described in the remainder of this section, and is illustrated in Figure 4.10 through Figure 4.14.

**Timelines as Trees** With one exception, the logical ordering of user nodes follows from the semantics of implementation—thus the IMPLEMENT model draws directed edges between consecutive user nodes, and draws a directed edge to each user receive event from the corresponding user send event.

The exception is the rollback transition. Rollback requires a user process to restore an earlier state and continue execution from there. Logically, the restored state node becomes a sibling[3] of

[3]Section 4.3.5 considers the implications of restoring the original node itself.

its original instance; unless rollback occurs again, subsequent user nodes form a linear sequence extending from this restored state node.

This branching constitutes a departure from the linear timeline basis of parallel pairs. The *live history* of a user node consists of its past in the transitive closure in its process timetree.

**The User Model** The IMPLEMENT model acts on SYSTEM_PARTIAL_ORDER graphs to abstract away the implementation details of the user computation. Figure 4.10 sketches the production of state nodes; Figure 4.11 sketches internal transitions; Figures 4.12 and 4.13 sketch the send and receive events for user messages; and Figure 4.14 sketches the rollback transition. We define the USER_PARTIAL_ORDER model as a composition:

$$\text{USER\_PARTIAL\_ORDER} \;\equiv\; \text{IMPLEMENT} \;\circ\; \text{SYSTEM\_PARTIAL\_ORDER}$$

We define the TIMETREES model as USER_PARTIAL_ORDER, less the message edges.

The models (USER_PARTIAL_ORDER, TIMETREES) form a Type 4 (consistent, independent, strongly monotonic and flow-supported) nonlinear pair: a parallel pair, less the requirement that process graphs be linear.[4] This will be the only nonlinear pair considered in this thesis.

**Timestamp Pseudo-vectors** Because nonlinear pairs do not place the nodes at a process in a linear order, we cannot guarantee that any collection of nodes at a process has a minimal and a maximal element. This uniqueness property is key to defining timestamp vectors and rollback vectors for nodes—without it, the definitions collapse.

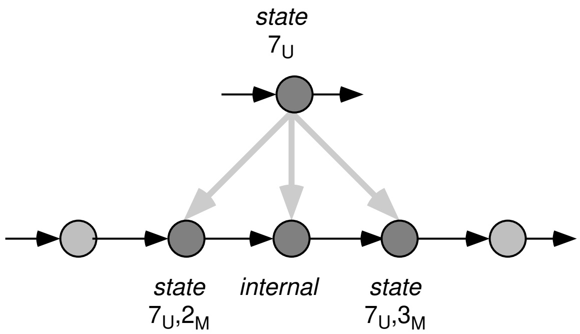


**Figure 4.10** Under the IMPLEMENT map, each USER_PARTIAL_ORDER state node (top) represents a maximal sequence of SYSTEM_PARTIAL_ORDER nodes that have no user state changes (bottom). Subscripts on the state labels distinguish the user part of process state from the management part.

[4]Section 2.2.8 discussed nonlinear pairs.

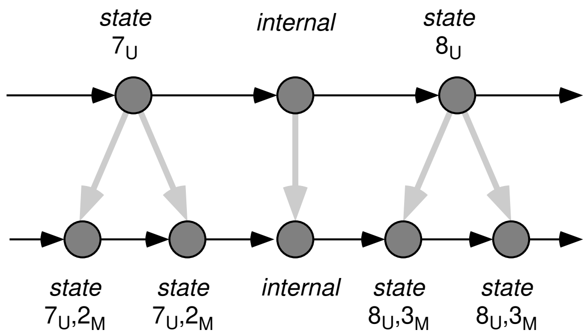


**Figure 4.11** Under the IMPLEMENT map, each USER_PARTIAL_ORDER internal node (top) represents a SYSTEM_PARTIAL_ORDER internal node that implements a user state change (bottom).

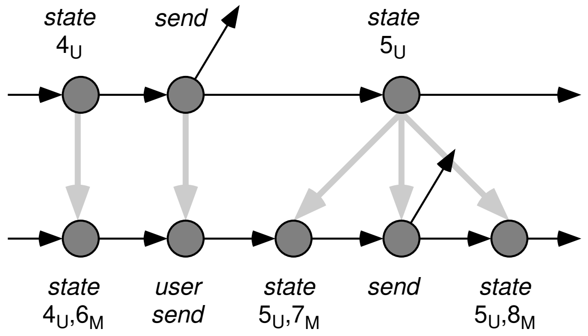


**Figure 4.12** To implement a USER_PARTIAL_ORDER *send* (top), the system process lets the user process send the message virtually. The system process then takes care of the details of actually sending the message (bottom).

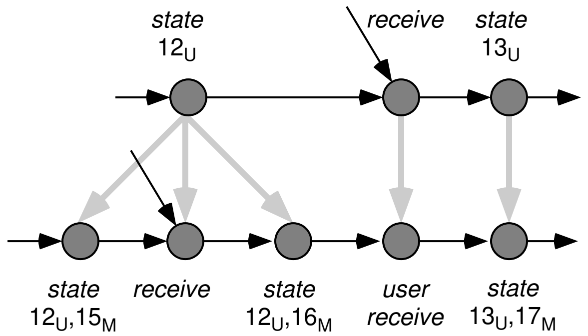


**Figure 4.13** To implement a USER_PARTIAL_ORDER *receive* (top), the system process receives the message and forwards it to the user process (bottom).

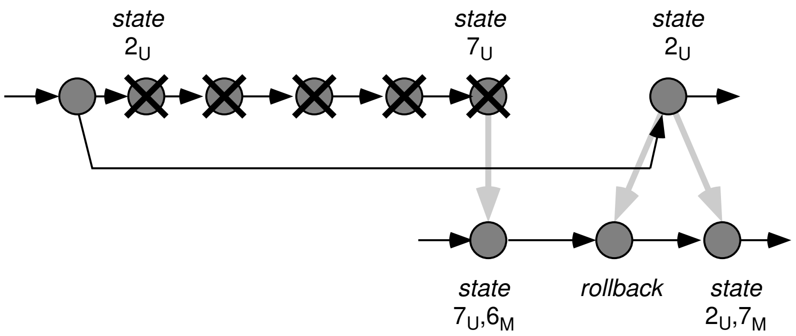


**Figure 4.14** A system process performs a *rollback* transition by restoring an earlier user state (bottom). The implemented user transition falls outside the normal rules of transition of user state; thus the new user state is a logical sibling of its earlier instance. We adopt convention that the restored *user state* node represents the *rollback* transition. A large "X" marks each node that has been rolled back.

For a given node $A$ in USER_PARTIAL_ORDER, we can still define a *timestamp pseudo-vector* $\mathbf{V}'(A)$ as the $\overline{\text{TIMETREES}}$-maxima of the live history of $A$. The timestamp pseudo-vector will not necessarily be a true vector, since it may contain multiple nodes from the same process (but from different branches of the timetree). If a timestamp pseudo-vector $\mathbf{V}'(A)$ is in fact a vector, we use the notation $\mathbf{V}_{\text{usr}}(A)$.

(Section 4.2.7 will discuss further properties of timestamp pseudo-vectors, and will observe why generalizing rollback vectors to rollback pseudo-vectors is difficult.)

### 4.2.5. Mapping Between the System and User Computation

We will need to map between the USER_PARTIAL_ORDER and the SYSTEM_PARTIAL_ORDER levels of abstraction. This section introduces some tools for this mapping. We show that user precedence implies system precedence (of corresponding nodes); we introduce some shortcuts for graphical notation; and we provide some clock primitives for processes to perform this mapping explicitly.

**Precedence** User precedence implies system precedence.

**Theorem 4.1** Let $\beta$ be a SYSTEM_PARTIAL_ORDER graph, and let $\gamma$ be the corresponding USER_PARTIAL_ORDER graph. Let $A_U, B_U$ be nodes in $\gamma$. Any choice of $A_S$ from $\langle$ IMPLEMENT, $\beta\rangle(A)$ and $B_S$ from $\langle$ IMPLEMENT, $\beta\rangle(B)$ satisfy the statement:

$$A_U \longrightarrow B_U \text{ in } \overline{\gamma} \quad \Longrightarrow \quad A_S \longrightarrow B_S \text{ in } \overline{\beta}$$

*Proof* This follows from the definition of IMPLEMENT. ☐

**Graphical Shorthand** Theorem 4.1 implies that, when we care about transitive precedence and the node set is unambiguous, we may use the same drawing to represent relations from both the SYSTEM_PARTIAL_ORDER model and the USER_PARTIAL_ORDER model. Figure 4.15 shows an example. For these drawings, we adopt the convention of dashed arrows for system-only edges, and solid arrows for edges that carry precedence in both the system and user models. System precedence corresponds to any path; user precedence corresponds to paths composed of solid edges only.

Usually, we can build these combined diagrams showing USER_PARTIAL_ORDER nodes. Potential for ambiguity arises when we want to consider the system activity that a user node represents. We are particularly interested in three areas:

- the send event for a system message (and the decision to send it);
- the receive event for a system message (and subsequent processing); and
- the receive event for a user message that a system process decides to discard (so the message becomes a system-only message).

When relevant, we indicate the interesting sequence of SYSTEM_PARTIAL_ORDER nodes that a USER_PARTIAL_ORDER node represents by a "peapod" drawing (such as Figure 4.16). As system-only edges, SYSTEM_PARTIAL_ORDER messages are indicated by dashed arrows. This convention fails to distinguish a system-only message from a rejected user message; where relevant, this distinction will be made clear in discussion.

**Primitives for Mapping** Some of our protocols require processes themselves to map between levels. We now specify some explicit primitives for this task. (Recall from Section 2.4 that *CUR_GRAPH* returns the current ground-level computation graph.)

Processes need to map nodes back and forth. We define two primitives:

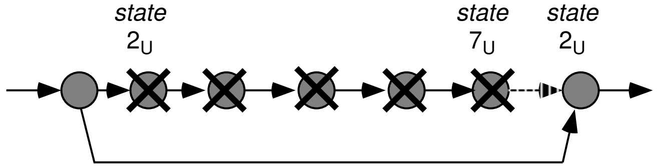


**Figure 4.15** Since Theorem 4.1 shows that user precedence implies system precedence, the same drawing can show both $\overline{\text{USER\_PARTIAL\_ORDER}}$ and $\overline{\text{SYSTEM\_PARTIAL\_ORDER}}$ information. We use the convention that dashed edges carry precedence in SYSTEM_PARTIAL_ORDER only. This sketch shows that, after rollback, the restored state system-follows the aborted state.

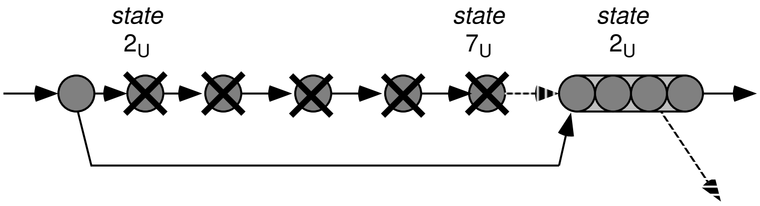


**Figure 4.16** When relevant, a “peapod” drawing reveals the implementation detail of a user node. Suppose upon rollback, the system process here sends a system-only message. We can indicate that in our combined drawing by expanding the node for the restored state into a “peapod” indicating the subsequence of interesting system nodes—*rollback*, *state*, *send*, and *state*—that this user node represents. The dashed message arrow indicates the system-only message.

- *USER*$(A)$ returns the user node representing system node $A$.

$$\mathit{USER}(A) \equiv \mathit{NODE}(B, A \in \langle \text{IMPLEMENT}, \beta \rangle(B), \text{IMPLEMENT}(\beta))$$

  (Here, $\beta = \text{SYSTEM\_PARTIAL\_ORDER}(\mathit{CUR\_GRAPH})$).

- *SYSTEM*$(A)$ returns the set of system nodes that user node $A$ represents.

$$\mathit{SYSTEM}(A) \equiv \langle \text{IMPLEMENT}, \text{SYSTEM\_PARTIAL\_ORDER}(\mathit{CUR\_GRAPH}) \rangle(A)$$

Processes need to work with vectors on both levels. We define a primitive:

- *USER_VECTOR*$(V)$ takes system vector $V$ and maps each entry $A$ to its user version *USER*$(A)$.

Finally, processes need to work with messages on both levels. We define three primitives:

- *USER_MESSAGE_TEST*$(M)$ returns *true* iff system message $M$ is also a user message.
- *USER_MESSAGE*$(M)$ extracts the user message from system message $M$.
- *SEND_EVENT*$(M, \mathbf{M})$ returns the send event in $\mathbf{M}$ associated with message $M$.

These primitives form part of our clock suite for processes. However, we also use *USER*, *SYSTEM*, and *USER_VECTOR* as informal shorthand for the operations they carry out.

### 4.2.6. Retroactive Change

Section 4.2.3 and Section 4.2.4 introduced two time models for optimistic rollback recovery. Understanding how recovery leads to a failure-free virtual computation arising from these models is critical to defining and solving the rollback problem. This section explores these issues.

**Failure-Free Computations** The goal of rollback recovery is to establish a failure-free virtual computation. To facilitate this work, we define a *failure-free trace* to be a trace of a system consisting of processes with implemented state only, that never fail. We define

FAILURE_FREE_PARTIAL_ORDER

to be the PARTIAL_ORDER_TIME model applied to failure-free traces.

**Extracting Failure-Free Computations** The SYSTEM_PARTIAL_ORDER model constructs the standard partial order for the system-level computation, and the USER_PARTIAL_ORDER model constructs the dependency partial order for the user nodes. In the terminology of distributed time, we say that the SYSTEM_PARTIAL_ORDER model refines to the USER_PARTIAL_ORDER model.

$$\text{SYSTEM\_PARTIAL\_ORDER} \quad \rhd \quad \text{USER\_PARTIAL\_ORDER}$$

The system-level computation determines the user-level computation.

However, if failure has actually occurred, then the USER_PARTIAL_ORDER graph will not be a FAILURE_FREE_PARTIAL_ORDER graph, because USER_PARTIAL_ORDER is constructed from the TIMETREES process structures, showing all rolled-back computation. Furthermore, extracting a FAILURE_FREE_PARTIAL_ORDER graph from USER_PARTIAL_ORDER leads to tricky situations:

- If recovery proceeds correctly but more than one process must roll back, then the graph from the USER_PARTIAL_ORDER model will generate a unique recovered failure-free computation, but may generate multiple "older" computations.
- Consequently, designing correct recovery protocols (or even unambiguously specifying the rollback problem) can be difficult. A particular challenge is getting a distributed collection of processes to agree, based on knowledge of one failure and differing views on the unfolding USER_PARTIAL_ORDER computation, on which failure-free computation to restore.

**An Example** Consider the SYSTEM_PARTIAL_ORDER computation described by Figure 4.17. Process $q$ decides to roll back the send event $B_3$, and establishes a copy $B'_2$ of earlier state node $B_2$. Process $q$ sends a message to process $p$, who cooperates. Process $q$ performs modified replay, and executes $B_5$ instead.

This example provides a clear distinction between the old virtual user computation and the new virtual user computation. Figure 4.18 shows the FAILURE_FREE_PARTIAL_ORDER computation before recovery; Figure 4.19 shows the FAILURE_FREE_PARTIAL_ORDER computation after recovery.

Identifying the recovery period in the USER_PARTIAL_ORDER computation of Figure 4.17 is also straightforward. From a real-time perspective, recovery should begin at the real time $t_s$ that process $q$ first rolls back, and end at the real time $t_f$ that process $p$ rolls back. Distributed time lets

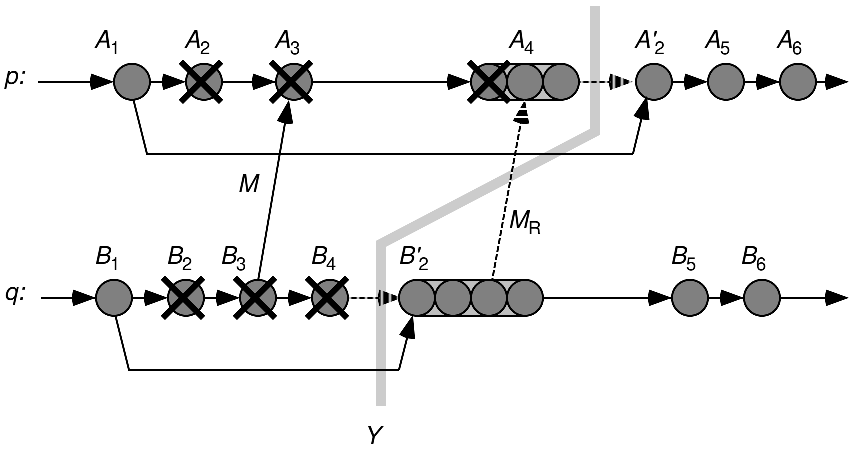


**Figure 4.17** This combined drawing shows an example of rollback with modified replay. After user state $B_4$, process $q$ decides to roll back node $B_3$. Process $q$ restores a copy of state $B_2$, and informs process $p$, who cooperates and then proceeds with its own modified replay. The pair of timeline edges marked $Y$ delineates the transition from the old computation to the new computation.

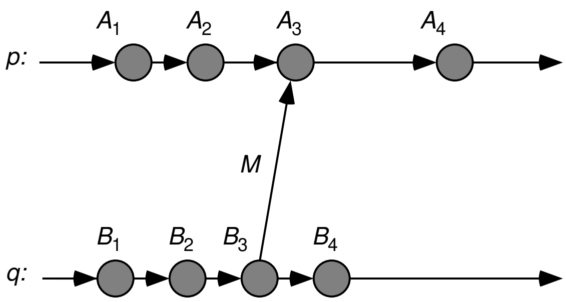


**Figure 4.18** This graph shows the failure-free virtual computation from Figure 4.17 before process $q$ initiates recovery.

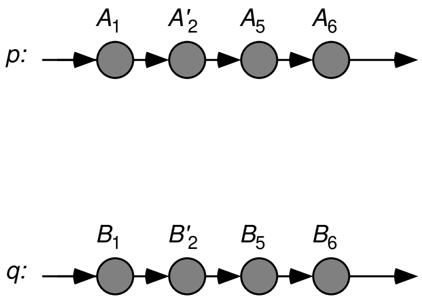


**Figure 4.19** This graph shows the failure-free virtual computation from Figure 4.17 after rollback recovery is complete.

us draw even tighter boundaries: the pair of timeline edges $Y$ in Figure 4.17 marks the transition from the old computation to the new computation.

In this example, both process $p$ and process $q$ perform rollback. The user computation at each is a tree; Figure 4.20 shows the USER_PARTIAL_ORDER graph. Before process $q$ initiates rollback recovery, the user computation consists of $A_4, B_4$ and their pasts (the FAILURE_FREE_PARTIAL_ORDER graph of Figure 4.18). After recovery, the user computation consists of to $A_6, B_6$ and their pasts (the FAILURE_FREE_PARTIAL_ORDER graph of Figure 4.19). However, the USER_PARTIAL_ORDER graph of Figure 4.20 also admits a *third* FAILURE_FREE_PARTIAL_ORDER graph: that determined by $A_6, B_4$ and their pasts. Figure 4.21 shows this computation, where message $M$ is sent but never received.

Scenarios exists where the computation of Figure 4.21 may be the correct virtual user computation arising from the USER_PARTIAL_ORDER graph of Figure 4.20. Suppose process $q$ decides that its initial decision to roll back node $B_2$ was incorrect, and wants to restore its earlier computation. What computation should the system establish? The most straightforward answer is to return from the recovered computation of Figure 4.19 to the older computation of Figure 4.18. However, the computation of Figure 4.21 might be a more reasonable result: fewer nodes need to be rolled back, and no extra messages need to be transmitted.

**Questions** Considering this example raises a number of questions:

- How do FAILURE_FREE_PARTIAL_ORDER computations arise from a USER_PARTIAL_ORDER computation? When are FAILURE_FREE_PARTIAL_ORDER computations incompatible? The

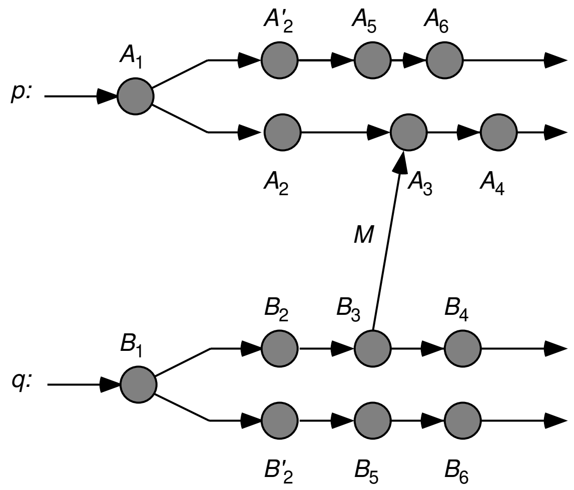


**Figure 4.20** The SYSTEM_PARTIAL_ORDER computation of Figure 4.17 maps to this USER_PARTIAL_ORDER graph. The recovery changed the current frontier from $A_4, B_4$ to $A_6, B_6$.

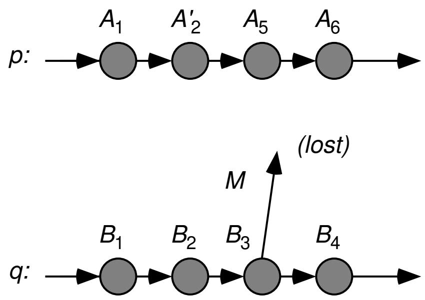


**Figure 4.21** The USER_PARTIAL_ORDER computation of Figure 4.20 admits a third failure-free virtual computation: one where message $M$ is sent but never received.

three virtual computations arising from Figure 4.20 each have subgraphs that are valid FAILURE_FREE_PARTIAL_ORDER graphs. Intuitively, we reduce these myriad graphs to three distinct computations. Why these three? Why are they distinct?

- How should we specify rollback problems? When a process changes the computation in progress by moving to another branch in its USER_PARTIAL_ORDER tree, what overall change in the virtual user computation should result?

- How do we design recovery protocols? How should separate processes agree on the current virtual user computation?

- How should we evaluate the performance of rollback protocols? Our discussion suggests several somewhat independent parameters:

  - how quickly recovery completes;
  - how many processes are involved with recovery;
  - how many nodes need to be rolled back;
  - how many messages become "lost;" and
  - how many system messages need to be sent as part of recovery.

  What tradeoffs exist? Which parameters are most important?

In Section 4.2.7, we begin answering these questions.

### 4.2.7. Validity and Consistency

In this section, we define *valid* user nodes, and we show how valid nodes comprise *consistent* virtual computations.

**Validity** Identifying a system-wide virtual user computation begins by selecting user nodes at individual processes. A user node is *valid* when (from its perspective) it is part of a failure-free virtual computation. That is, user node $A$ is valid when its live history forms a past-closed prefix of a graph generated by FAILURE_FREE_PARTIAL_ORDER.

> **Theorem 4.2** A user node $A$ is valid iff its timestamp pseudo-vector $\mathbf{V}'(A)$ is a vector.

*Proof* The pseudo-vector $\mathbf{V}'$ is a vector exactly when the past-closure of the live history of $A$ touches at most one branch in the timetree at each process. $\square$

**Consistency** A set $S$ of nodes in a USER_PARTIAL_ORDER graph is *consistent* iff the nodes could all have been part of the same failure-free virtual computation. That is, the graph formed by taking the nodes in $S$ along with their live histories forms a past-closed prefix of a graph from FAILURE_FREE_PARTIAL_ORDER.

Timestamp pseudo-vectors provide a nice way to describe consistency.

> **Theorem 4.3** A set $S$ of nodes in a USER_PARTIAL_ORDER graph is consistent iff each node in $S$ is valid, and the TIMETREES-maximum of the timestamp vectors $\mathbf{V}(A)$ (for all $A \in S$) is a vector.

*Proof* Let $\beta$ be the USER_PARTIAL_ORDER graph, and $\beta'$ be the subgraph obtained by taking $S$ and their past-closure. If $S$ is consistent, then $\beta'$ is a valid PARTIAL_ORDER_TIME graph, so each node $A \in S$ must be valid, and for each $p$, the $p$ entries of the timestamp vectors are orderable within a single branch of the $p$ timetree. If each $A \in S$ is valid and their timestamp vectors join to a vector, then the past-closure of $S$ touches exactly one branch in each timetree, and so $S$ is consistent. $\square$

(That is, the timestamp vectors of consistent nodes form a lattice.)

Consistency directly generalizes from validity: the singleton set $\{A\}$ is consistent iff the node $A$ is valid. Consistency of sets also builds in a nice way: a set $S$ of valid nodes is consistent iff each pair of nodes in $S$ is consistent.

Some straightforward approaches to describing consistency actually fail. For example:

- A set $S$ is not necessarily consistent if the USER_PARTIAL_ORDER-maxima of its past form a vector. (Figure 4.22 provides a counterexample.) Using TIMETREES rather than USER_PARTIAL_ORDER ordering is important if one branch of a tree can develop a dependence on another branch.

- A set $S$ is not necessarily inconsistent even if it is not TIMETREES-dominated by a set of consistent leaves. (Figure 4.23 provides a counterexample.) Thinking about computations as arising from a set of process *incarnations*—maximal root-leaf branches—leads to this incorrect description.

- While consistency of a set follows from pairwise consistency, evaluating whether a node $A$ at process $p$ is consistent with a node $B$ at process $q$ still requires system-wide data—nodes $A$ and $B$ may be inconsistent because they depend on concurrent branches of the timetree at a third process $r$. (The timestamp vectors provide the system-wide data.)

The timestamp pseudo-vector of a valid node $A$ marks the lower bound of the events concurrent and consistent with $A$—the adjusted timestamp vector of a valid node is the minimal consistent timeslice containing that event. The asymmetry of time in USER_PARTIAL_ORDER makes it difficult to define a similar "rollback pseudo-vector" having a similar property.

**Generation** A failure-free virtual computation arises from a USER_PARTIAL_ORDER graph through *consistency*. A set of consistent nodes in the USER_PARTIAL_ORDER graph determines

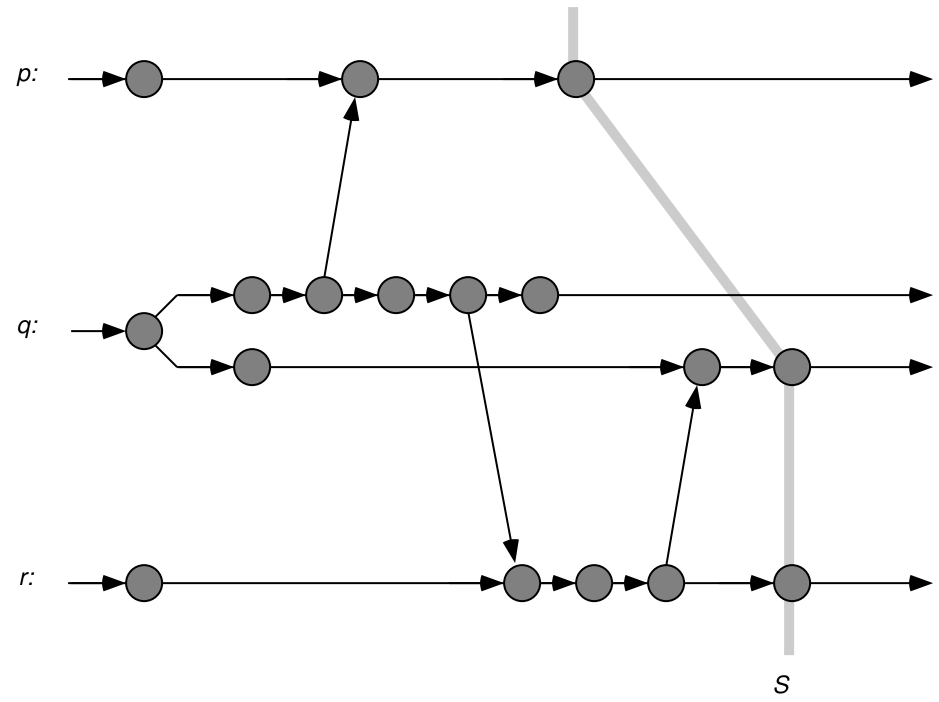


**Figure 4.22** Even if the USER_PARTIAL_ORDER-maxima of the past of a set forms a vector, the set itself may not be consistent. This graph shows a counterexample: the past of the mutually concurrent vector $S$ is not a valid prefix of a failure-free virtual computation.

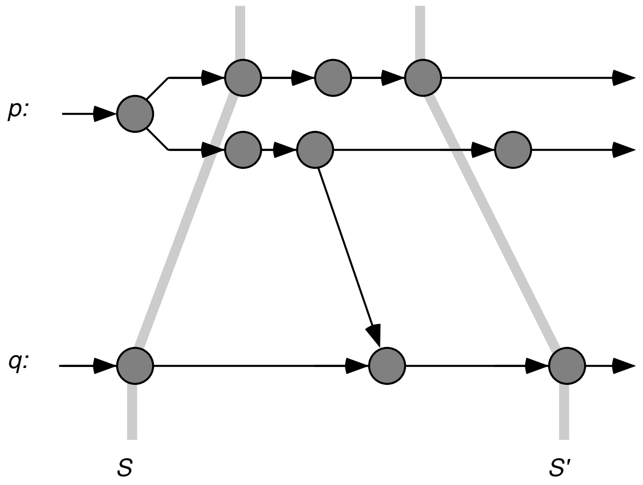


**Figure 4.23** Not being dominated by a consistent leaf vector does not imply inconsistency. This graph shows a counterexample: the vector $S$ is consistent, even though the only dominating leaf-vector $S'$ is not consistent.

a past-closed prefix of a FAILURE_FREE_PARTIAL_ORDER graph—the nodes, along with their live histories. When two consistent sets are distinct but have a consistent union, then these sets represent different intermediate versions of the same computation. The three virtual computations we extracted from Figure 4.17 arise from the three maximal distinct consistent sets.

The goal of optimistic rollback recovery to restore consistency to the system computation: to ensure that the current user nodes at the processes form a consistent set.

## 4.3. Asynchronous Optimistic Rollback Recovery Using Distributed Time

The distributed time framework developed in this thesis provides tools for reasoning about multiple levels of time relations, for designing protocols in terms of these relations, and for considering independently the inherent security and privacy risks. Section 4.3 uses 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 or event is an *orphan*. 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. This procedure is complete in that it reports no false negatives. It thus allows *completely asynchronous* recovery while also ensuring that each process rolls back at most once to recover from any failure—and that processes that do not depend on the failure need not roll back at all.

This protocol thus substantially improves on previous optimistic rollback recovery protocols.

Section 4.3.1 provides an overview of this work. Section 4.3.2 discusses the orphan detection test. Section 4.3.3 presents the complete protocol. Section 4.3.6 compares our protocol to previous work.

### 4.3.1. Overview

Rollback recovery requires determining which states and events have been potentially influenced by lost activity. Many 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 also requires the ability to reason about and track potential knowledge of failures and restarts. This activity itself is an asynchronous distributed computation, and is 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 at the system level is not the same as causal dependency at the user level.* For example, suppose process $q$ learns that its current state $A$ depends on a lost state. Process $q$ rolls back, and then enters state $B$. Although a knowledge path exists from state $A$ to state $B$, no causal dependency path exists.

For effective implementation of asynchronous recovery, we need to move from viewing time as a linear order to viewing it as a partial order, and we *also* need to move away from viewing time as a single level of abstraction. The 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.

**Advantages** Our new protocol is the first optimistic rollback protocol to implement completely asynchronous recovery effectively. 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 must roll back, it can roll back immediately and resume computation *without additional synchronization* with other processes.
- **Maximal Recovery** Like other optimistic rollback protocols, ours guarantees that a state or event is rolled back iff it causally depends on the computation lost at failed processes.
- **Minimal Rollbacks** Our protocol *also* guarantees that a failure at process $p$ causes a process $q$ to roll back at most once. Processes that do not depend on the failure will not roll back at all.
- **Speedy Recovery** Suppose process $q$ must roll back because of a failure at process $p$. Process $q$ will roll back as soon as any knowledge path is established from $p$'s rollback.

- **Concurrent Recovery** Recovery from a process failure occurs as information about the failure propagates. Basing recovery on information flow rather than coordinated rounds directly allows recovery from concurrent failures to proceed concurrently: the recoveries merge and the protocol restores the maximum recoverable system state [Jo89]. (In particular, two processes that each need to roll back due to two failures do not need to react to the failures in the same order.)

- **Toleration of Network Partitions** Another side-effect of our asynchronous approach is that once initiated, recovery can proceed despite a partitioned network. The only processes that need to worry about recovery are those that may causally depend on lost states. Since each such process can 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 may proceed unhindered. (However, this work does not address the problem of detecting failure in a partitioned network.)

- **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 provide transparent protection against these risks.

**Drawbacks** Our new protocol does require timestamp information to be maintained, since processes must track relations in both the user and system partial orders. Vector clock implementations for these models require one entry per process. For SYSTEM_PARTIAL_ORDER, these entries can be a pair of scalars. A straightforward implementation of USER_PARTIAL_ORDER clocks would require that the size of the entry for process $p$ be proportional to the number of rollbacks process $p$ has performed. However, optimizations may substantially reduce this size. For example, Strom and Yemini obtain constant size entries by transmitting the extra data incrementally (at the cost of not always having sufficient data to make a comparison). Using similar implementations will keep timestamp size in our protocol within a factor of two of Strom and Yemini. Section 4.3.4 considers these issues in more detail.

### 4.3.2. Orphan Detection

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

**Preliminaries** We assume that processes enforce the invariant that their user state is always valid, according to the definition in Section 4.2.7. (Section 4.3.3 will show this assumption is easily satisfied.) We also assume that processes only restore states from their current live history.

We discuss our protocol in terms of distributed time, which describes computations as graphs. Consequently, we sometimes informally identify a *node* in a computation graph with the *state* or *event* it represents.

Discussing two levels of time sometimes make the use of Roman letters for node names ambiguous. For example, is the node "$A$" a system-level node or a user-level node? Where a simple name may be misleading, we adopt the convention of using subscripted Roman letters; e.g., "$A_S$" will be a system node, and "$B_U$" will be a user node. We adopt a similar convention for messages and vectors.

**Why Orphan Testing is Crucial** Suppose $p$ is the process that actually failed. The system process at $p$ initiates recovery by restoring earlier user state and continuing user-level execution. This action causes one or more *live* nodes at process $p$ to become *rolled-back*. These rolled-back nodes are *orphans* by definition. However, the rollback action at $p$ may also cause nodes at other processes to become *orphans*.

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

- **Orphan Elimination** When process $q$ learns that process $p$ has failed, process $q$ must determine if its current user state has become an orphan. If so, process $q$ must 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 nodes* are orphans. Figure 4.24 shows a detailed example of this situation.
- **Orphan Prevention** The rollback at process $p$ may have caused user node $A_U$ at some process $r$ to become an orphan. However, suppose $A_U$ was the send of a message to process $q$. If the user process at $q$ accepts the message, then $q$ will become an orphan. Thus, to prevent their current user nodes from becoming orphans, processes need to be able to test if *user events at other processes* are orphans. Figure 4.25 shows a detailed example of this situation.

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

**Knowledge of Orphans** Suppose the system process at $q$ is in node $B_S$. When could $q$ know that a user node $A_U$ is an orphan? We use distributed time to answer this question.

In order to test $A_U$, the system process at $q$ must be aware of $A_U$. We must have the precondition that for some system node $A_S$ in *SYSTEM*$(A_U)$, $A_S \Longrightarrow B_S$.

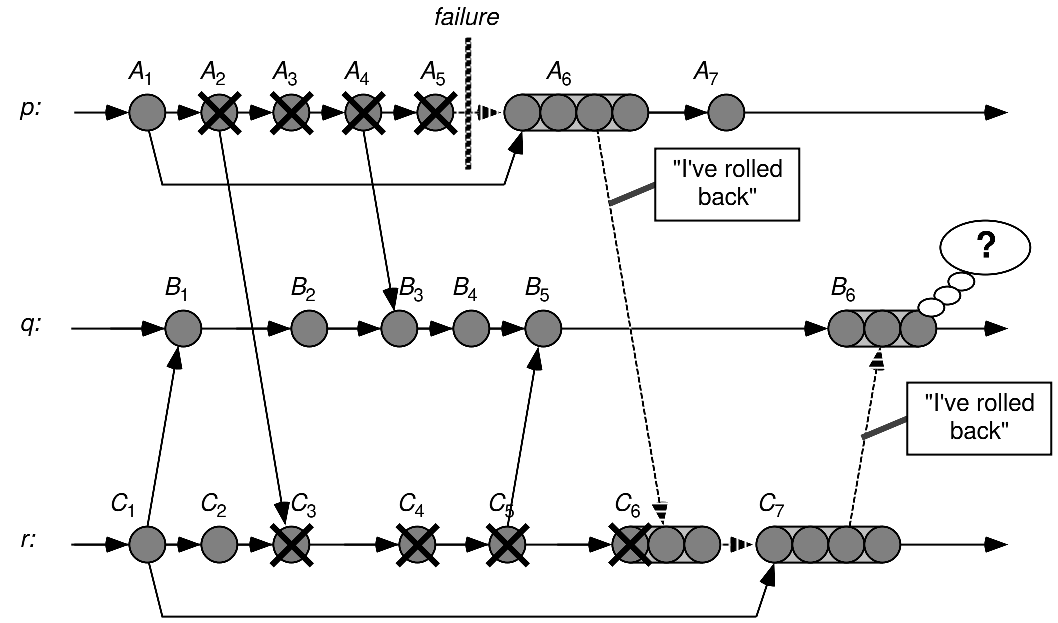


**Figure 4.24** Optimistic rollback recovery raises the challenge of orphan elimination: when processes learn of failure, they need to determine their most recent node that is not an orphan. In this diagram, all named nodes are user nodes. Process $p$ fails and restarts, and informs process $r$, who restores a copy of the state at $C_2$ and informs process $q$. When it learns of process $r$'s rollback, process $q$ must decide if and how far it should roll back. Process $q$ depends directly on rolled back nodes at process $r$, so a naive analysis would suggest rolling back to before $B_5$. In actuality, process $q$ should roll back to before $B_3$, since that node has a direct dependence on rolled-back node $A_4$ at process $p$, whose failure triggered the rollback at process $r$.

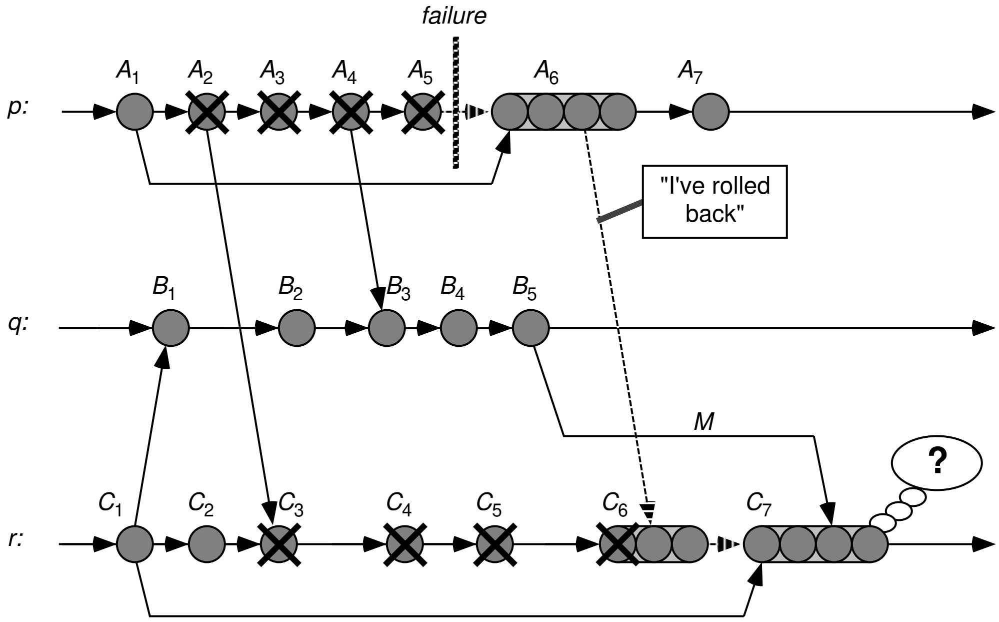


**Figure 4.25** Optimistic rollback recovery also raises the challenge of orphan prevention: before formally receiving an arriving user message, a process should determine if the send event is an orphan. Again, all named nodes are user nodes. Process $p$ fails and rolls back, and informs process $r$ who rolls back and restores a copy of the state at $C_2$. Process $r$ then receives user message $M$ from process $q$. The send event of $M$ is an orphan, since it user-follows from a rolled-back node $A_4$ at process $p$. Accepting this message would cause the user process at $r$ to become an orphan.

For $A_U$ to be an orphan, a rolled-back user node $C_U$ must exist with $C_U \Longrightarrow A_U$. From Theorem 4.1 and transitivity, $C_S \Longrightarrow B_S$ for any system node $C_S$ in *SYSTEM*$(C_U)$. A path of potential information flow exists from the rolled-back $C_U$ to the system process at $q$.

However, for the system process at $q$ to know that dependence on the rolled-back $C_U$ makes $A_U$ an orphan, $p$ must know that that $C_U$ has been rolled back. If $D_S$ is the system node that rolled back $C_U$, then we must have $D_S \Longrightarrow B_S$ as well.

We summarize this formally with the predicate *ORPHAN*$(A_U, B_S)$, which is defined only when $A_S \Longrightarrow B_S$ for some $A_S \in$ *SYSTEM*$(A_U)$.

$$\textit{ORPHAN}(A_U, B_S) \equiv \textit{true} \iff$$
$$\exists\, C_U, D_S \text{ such that } \begin{cases} 1.\ C_U \Longrightarrow A_U \text{ in the USER\_PARTIAL\_ORDER model} \\ 2.\ D_S \Longrightarrow B_S \text{ in the SYSTEM\_PARTIAL\_ORDER model} \\ 3.\ D_S \text{ rolls back } C_U \end{cases}$$

The *ORPHAN* predicate does not capture *all* the orphans in the computation—just all the orphans that a given system process may potentially know are orphans. If process $p$ sends process $q$ a user message but promptly rolls back without telling anyone, then process $q$ can not know that the send is an orphan. In the SYSTEM_PARTIAL_ORDER model, the timestamp vector on a node $B_S$ marks the information horizon of that node. At node $B_S$, the system process cannot know about anything beyond this horizon.

**An Optimal Orphan Test** We can use distributed time to build a test that captures the *ORPHAN* predicate exactly. First, we build a test that lets a system process determine if (to its current potential information) a node has been rolled back. Then, we generalize this test to let a system process determine if a given node depends on a node that has been rolled back.

Let $B_S$ be a system node at process $q$. Let $E_S$ be the $p$ entry of $\mathbf{V}_{\text{sys}}(B_S)$, and let $E_U = \textit{USER}(E_S)$. At $B_S$, process $q$ has no information that $E_U$ is not live.[5] Any $F_S$ rolling back $E_U$ would system-follow $E_S$; if $q$ could know about such an $F_S$, then $E_S$ would not have been the $p$ entry in $\mathbf{V}_{\text{sys}}(B_S)$.

Further, process $q$ at $B_S$ can know of a user node $C_U$ at process $p$ iff for some $C_S \in$ *SYSTEM*$(C_U)$, $C_S$ precedes $B_S$. Process $q$ at $B_S$ can sort these user nodes at $p$ into two groups:

- those that user-precede $E_U$ (the user version of the $p$ entry of $\mathbf{V}_{\text{sys}}(B_S)$), and
- those that do not.

[5] By definition, node $E_U$ is live iff its process $p$ has not rolled it back. A live node may be an orphan; knowing that a node is live is not the same as knowing that it is not an orphan. For example, process $s$ may have rolled back an ancestor $G_U$ of $E_U$. Process $q$ may perceive that $p$ has not rolled back $E_U$ but $s$ has rolled back $G_U$, and consequently the currently live node at $p$ is an orphan.

Process $q$ at $B_S$ knows that each node in the second group has been rolled back. Process $q$ treats each node in the first group as if it were live, since $q$ has no information otherwise. For example, suppose $C_U$ is a user node that $q$ knows about at $B_S$. Consider the two cases:

- If $C_U \Longrightarrow E_U$, then either $C_U$ has not been rolled back, or information about this rollback (which would also roll back $E_U$) has not reached process $q$ at state $B_S$.

- If $C_U \not\longrightarrow E_U$, then by $E_S$, the system process at $p$ has rolled back $C_U$. This rollback event must precede or equal $E_S$ and thus $B_S$, so process $q$ knows about it.

Figure 4.26 sketches this scenario.

This reasoning shows how the system process at $q$ can determine if a specified node has been rolled back (according to the information potentially available to $q$). Since an *orphan* is a node that depends on a rolled-back node, this reasoning extends to allow $q$ to test for orphans. Let $A_U$ be a user state at process $r$ that process $q$ knows about at $B_S$. Let $p$ be an arbitrary process. Let $E_S$ be the $p$ entry of $\mathbf{V}_{\text{sys}}(B_S)$, and let $E_U = \textit{USER}(E_S)$. Let $C_U$ be the user-maximal user state at process $p$ with $C_U \Longrightarrow A_U$.

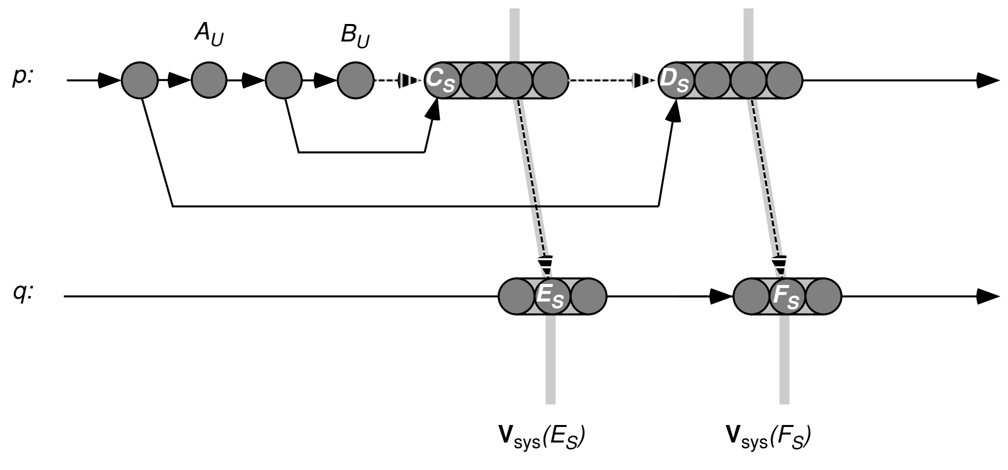


**Figure 4.26** The SYSTEM_PARTIAL_ORDER timestamp vector of a system process determines what states it can know to have been rolled back. Here, system process $q$ at $E_S$ knows about user nodes $A_U$ and $B_U$ at process $p$, since they lie within the system-horizon of $E_S$. (That is, paths exists from all nodes in *SYSTEM*$(A_U)$ and *SYSTEM*$(B_U)$ to $E_S$.) At $E_S$, process $q$ also knows that $B_U$ has been rolled back, since the rollback event $C_S$ also lies within the system-horizon of $E_S$. However, at $E_S$ process $q$ believes $A_U$ is still live, since the rollback event $D_S$ that undoes it lies beyond $\mathbf{V}_{\text{sys}}(E_S)$—and thus outside the knowledge of $E_S$. Process $q$ does not learn that $A_S$ has been rolled back until $F_S$.

If $C_U \Longrightarrow E_U$, then process $q$ at $B_S$ perceives no rollback at $p$ that makes $A$ an orphan. If this relation holds for all processes $p$, than process $q$ cannot perceive that $A_U$ is an orphan; according to $q$'s potential information at $B_S$, nothing that $A_U$ depends on has been rolled back. If this relation fails for any process, then process $q$ knows that $A_U$ is an orphan.

Vector clocks permit an elegant statement of this test. For a vector $W_S$ of system nodes, *USER_VECTOR*$(W_S)$ is the vector of user nodes obtained by applying *USER* to each entry. Let $\prec_{\text{usr}}$ denote the vector precedence relation under USER_PARTIAL_ORDER; vectors $U_U$ and $V_U$ satisfy $U_U \preceq_{\text{usr}} V_U$ when for each $p$, the $p$ entry of $U_U$ precedes or equals the $p$ entry of $V_U$ in $\overline{\text{TIMETREES}}$.

Define *DT_ORPHAN_TEST* by the following comparison:

$$\textit{DT\_ORPHAN\_TEST}(A_U, B_S) = \textit{true} \iff \mathbf{V}_{\text{usr}}(A_U) \not\preceq_{\text{usr}} \textit{USER\_VECTOR}(\mathbf{V}_{\text{sys}}(B_S))$$

That is, take the system timestamp of $B_S$, map each entry to its user equivalent, and do a TIMETREES vector comparison with the user timestamp of $A_U$.

This test captures all potential knowledge of orphans.

> **Theorem 4.4** If system node $B_S$ and valid user node $A_U$ satisfy $A_S \Longrightarrow B_S$ for some $A_S \in \textit{SYSTEM}(A_U)$, then they satisfy the statement:
>
> $$\textit{ORPHAN}(A_U, B_S) \iff \textit{DT\_ORPHAN\_TEST}(A_U, B_S)$$

*Proof* Let $A_U$ occur at process $p$ and $B_S$ at process $q$.

Suppose *ORPHAN*$(A_U, B_S)$ holds. Then at some process $r$, there exists a user node $C_U$ and system node $D_S$ satisfying the following statements:

1. $C_U \Longrightarrow A_U$,
2. $D_S$ rolls back $C_U$, and
3. $D_S \Longrightarrow B_S$.

Let $E_U$ be the $r$ entry of $\mathbf{V}_{\text{usr}}(A_U)$ (which exists, since $A_U$ is valid). Let $F_S$ be the $r$ entry of $\mathbf{V}_{\text{sys}}(B_S)$, and let $F_U = \textit{USER}(F_S)$. By (1) and the definition of timestamp vector, $C_U \Longrightarrow E_U$. Hence, $E_U \Longrightarrow F_U$ would imply $C_U \Longrightarrow F_U$. By (2), $C_U$ cannot precede or equal the user version of any system node $G_S$ at $r$ with $D_S \Longrightarrow G_S$ (since rolled-back nodes stay rolled back). By (3) and the definition of timestamp vector, $D_S \Longrightarrow F_S$. Thus $C_U$ cannot precede or equal $F_U$, so $E_U$ cannot precede or equal $F_U$. Thus *DT_ORPHAN_TEST*$(A_U, B_S)$ holds.

Conversely, suppose *DT_ORPHAN_TEST*$(A_U, B_S)$ holds. Then there exists a process $r$ such that $C_U$ fails to precede or equal $F_U$, where $C_U$ is the $r$ entry of $\mathbf{V}_{\text{usr}}(A_U)$, $F_S$ is the $r$ entry of $\mathbf{V}_{\text{sys}}(B_S)$, and $F_U = \textit{USER}(F_S)$. Let $C_S$ be the minimal element of *SYSTEM*$(C_U)$. Since the definition of timestamp vector provides $C_U \Longrightarrow A_U$, Theorem 4.1 provides $C_S \Longrightarrow A_S$ for any $A_S \in \mathit{SYSTEM}(A_U)$. Hypothesis then provides $C_S \Longrightarrow B_S$. The definition of timestamp vector then provides that $C_S \Longrightarrow F_S$. Since $C_U$ neither precedes nor equals $F_U$, there must exist a $D_S$ at $r$ in the range $C_S \longrightarrow D_S \Longrightarrow F_S$ such that $D_S$ rolls back $C_U$. By the definition of timestamp vector, $D_S \Longrightarrow B_S$. Hence $\mathit{ORPHAN}(A_U, B_S)$ holds. $\square$

Figure 4.27 shows how *DT_ORPHAN_TEST* resolves the orphan elimination problem from Figure 4.24. Figure 4.28 shows how the test resolves the orphan prevention problem from Figure 4.25.

### 4.3.3. The Protocol

We build our protocol for optimistic rollback recovery by having the system processes maintain vector clocks for the user and system partial orders, and then using these clocks to test for orphans.

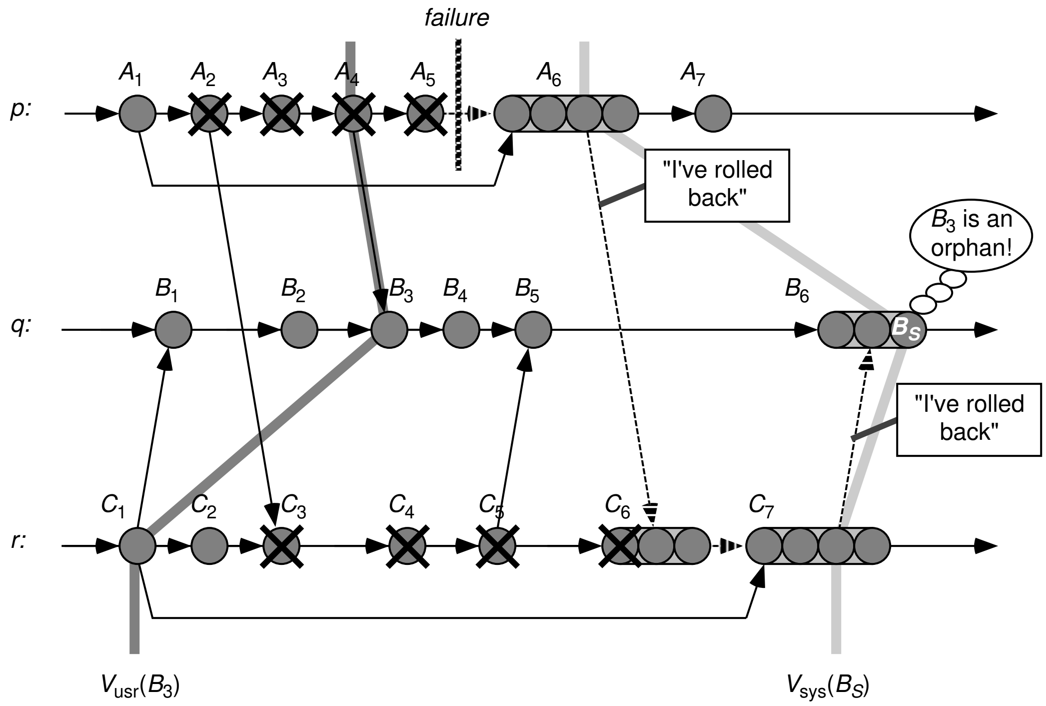


**Figure 4.27** *DT_ORPHAN_TEST* allows accurate orphan elimination. Here, node $B_S$ is the only named system node. At $B_S$, the system process at $q$ can know that user node $B_3$ is an orphan, because in the process $p$ timetree, node $A_4$ (the $p$ entry of $\mathbf{V}_{\mathrm{usr}}(B_3)$) does not precede node $A_6$ (the *USER* image of the $p$ entry of $\mathbf{V}_{\mathrm{sys}}(B_S)$).

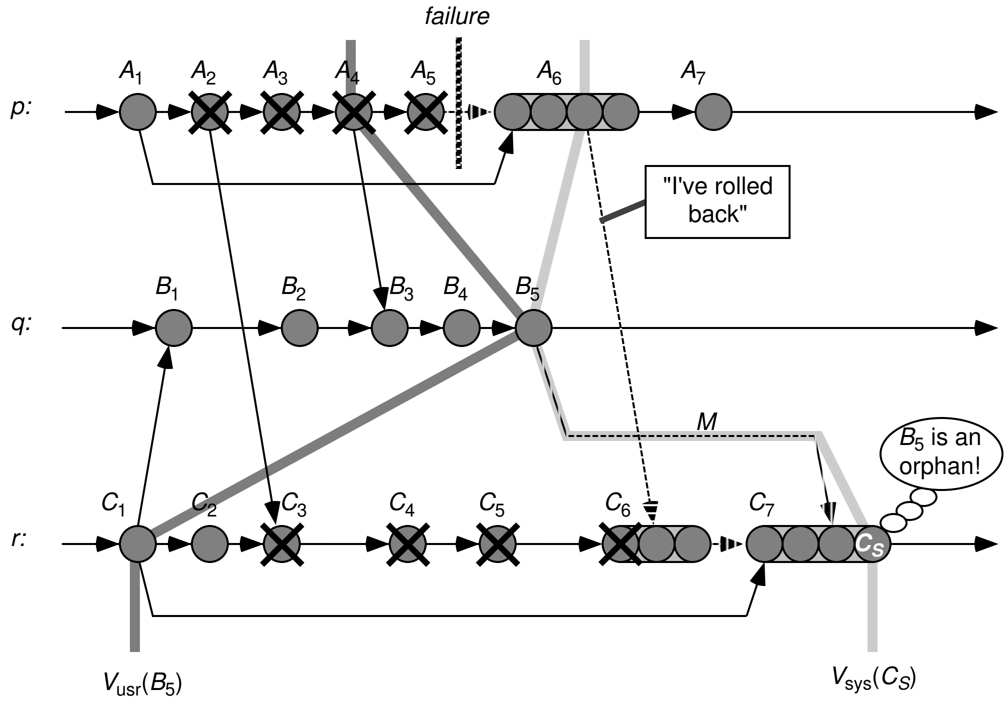


**Figure 4.28** *DT_ORPHAN_TEST* allows accurate orphan prevention. Here, node $C_S$ is the only named system node. At $C_S$, the system process at $r$ can know that the send $B_5$ of user message $M$ is an orphan, because in the process $p$ timetree, node $A_4$ (the $p$ entry of $\mathbf{V}_{\text{usr}}(B_5)$) does not precede node $A_6$ (the *USER* image of the $p$ entry of $\mathbf{V}_{\text{sys}}(C_S)$).

*/* the orphan test */*
**function** *DT_ORPHAN_TEST*(*TESTED_STATE*, *TESTING_STATE*)

  $V \leftarrow \mathbf{V}_{\text{usr}}(\textit{TESTED\_STATE})$
  $W \leftarrow \textit{USER\_VECTOR}(\mathbf{V}_{\text{sys}}(\textit{TESTING\_STATE}))$
  **return** $\neg \textit{COMPARE}(V, W, (\text{USER\_PARTIAL\_ORDER}, \text{TIMETREES}))$

*/* process $p$ receives system message $M$ */*
**procedure** *RECEIVE*($M_S$)

  */* set pointers to current nodes */*
  $A_S \equiv \textit{CUR\_NODE}(p, \text{SYSTEM\_PARTIAL\_ORDER})$
  $A_U \equiv \textit{CUR\_NODE}(p, \text{USER\_PARTIAL\_ORDER})$

  */* update* SYSTEM_PARTIAL_ORDER *vector*/*
  $S_S \leftarrow \textit{SEND\_EVENT}(M_S, \text{SYSTEM\_PARTIAL\_ORDER})$
  $\mathbf{V}_{\text{sys}}(A_S) \leftarrow \textit{MAX}(\mathbf{V}_{\text{sys}}(A_S), \mathbf{V}_{\text{sys}}(S_S), (\text{SYSTEM\_PARTIAL\_ORDER}, \text{SYSTEM\_TIMELINES}))$

  */* is $p$ now an orphan? */*
  **if** $\textit{DT\_ORPHAN\_TEST}(A_U, A_S)$
    **then** roll back to maximal non-orphan state

  */* was sender's current user node an orphan? */*
  **if** $\textit{DT\_ORPHAN\_TEST}(\textit{USER}(S_S), B_S)$
    **then** optionally inform the sender

  */* did $M_S$ include a user message? */*
  **if** $\textit{USER\_MESSAGE\_TEST}(M_S)$ **then**
    $S_U \leftarrow \textit{SEND\_EVENT}(\textit{USER\_MESSAGE}(M_S), \text{USER\_PARTIAL\_ORDER})$
    */* accept it if the user send is not an orphan */*
    **if** $\textit{DT\_ORPHAN\_TEST}(S_U, B_S)$
      **then** optionally inform the sender
      **else** accept $\textit{USER\_MESSAGE}(M_S)$

**Figure 4.29** 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. (We use $\equiv$ to indicate assignment by reference, and $\leftarrow$ to indicate assignment by value.)

**Sending a User Message** Suppose the user process at $p$ decides to send a message $M_U$ to the user process at $q$. The user process $p$ packages $M_U$ along with $\mathbf{V}_{\text{usr}}(A_U)$ (where $A_U$ is the user send event) and routes it to the system process at $p$, who sends the package as a system message.

**Sending a System Message** When the system process at $p$ sends a system message $M_S$ to the system process at $q$, it sends along the timestamp $\mathbf{V}_{\text{sys}}(B_S)$ (where $B_S$ the system send event). The system process at $p$ may optionally include the user timestamp vector of *USER*$(B_S)$.

**Receiving Messages** Figure 4.29 shows the procedure used for receiving messages. Suppose the system process at $p$ receives a system message $M_S$ sent by the system process at $q$. The system process at $p$ updates its current $\mathbf{V}_{\text{sys}}$ vector. If *DT_ORPHAN_TEST* indicates that $p$'s current user node is an orphan, the system process at $p$ performs rollback. If *DT_ORPHAN_TEST* indicates that the user node corresponding to the send of $M_S$ is an orphan, the the system process at $p$ may optionally inform $q$. If $M_S$ contains a user message $M_U$, then the system process at $p$ applies *DT_ORPHAN_TEST* to the send of $M_U$. If this event is an orphan, $p$ may optionally inform $q$; if not, the system process at $p$ lets the user process at $p$ receive the message.

(Suppose the send of a user message $M_U$ user-followed from node $A_U$ at process $r$, but process $p$'s current user node depends on $B_U$ at $r$, with $A_U$ and $B_U$ concurrent in the timetree. At least one of $A_U$, $B_U$ must have been rolled back, and the system timestamp on $M_U$ will carry that information if the system process at $p$ does not already know it. Thus, this protocol automatically enforces the invariant that user states are always valid.)

**Rollback** A process rolls back in two situations: when it fails, and when it discovers it is an orphan.

To roll back because of its own failure, a process restores the maximal recoverable state in its live history.

To roll back because it discovers it is an orphan, a process must find a state in its live history that is not an orphan—that is, a state whose $\mathbf{V}_{\text{usr}}$ timestamp still user-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-maximal state in the live history that is not an orphan.

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

### 4.3.4. Implementation Details

Implementing this protocol requires solving a number of problems:

- Building vector clocks for SYSTEM_PARTIAL_ORDER requires the ability to sort system states in terms of system timelines.
- Building vector clocks for USER_PARTIAL_ORDER requires the ability to sort user states in terms of user timetrees.
- Performing *DT_ORPHAN_TEST* requires the ability to map system states to user states (that is, to perform *USER*).

This section provides one possible solution. We build a *SYS_NAME* data structure for each system node, a *USR_NAME* data structure for each user node, and show how to perform the above functions in terms of these data structures.

**The System Timeline** System states at a process occur in consecutive order, so a simple scalar counter will suffice. The only complication arises because failed processes will not know how high their system state counter was before failure.

Consequently, we have each process maintain two counters: *INCARNATION_COUNT* tracks the current incarnation of the process [StYe85], and *SYS_COUNT* tracks the current node within that incarnation. Startup initializes each counter to zero. The *SYS_COUNT* counter is incremented with each subsequent system node, unless the subsequent node is a rollback node, in which case *SYS_COUNT* resets to zero, and *INCARNATION_COUNT* is incremented.

The *SYS_NAME* for a node consists of three items: the *INCARNATION_COUNT* value, the *SYS_COUNT* value, and the *USR_NAME* of the node's current user state. To sort two system nodes at the same process, we perform lexicographic comparison of the

$$(\mathit{INCARNATION\_COUNT}, \mathit{SYS\_COUNT})$$

pairs. To implement *USER*, we return the *USR_NAME* entry.

**The User Timetree** Comparing nodes in user timetrees is more challenging than comparing nodes in system timelines, because trees do not guarantee that two nodes can even be ordered. The restriction that we must generate *USR_NAME* values on-line further complicates matters.

We can begin by having each process maintain a *USR_COUNT* variable, initially zero, indicating the count of the current user node in the currently live history. The process increments *USR_COUNT* with each subsequent user node—except one obtained through rollback.

The *USR_COUNT* values suffice to sort two user nodes within the same live history. However, we need to be able to determine if two nodes are in the same history—that is, if one is a descendent of the other in the user timetree.

Conceptually, processes could track this information by maintaining the path from the root to the current node in the user timetree. Label the nodes in the user timetree with their *USR_COUNT* value, and the edge into each node $A$ with the *INCARNATION_COUNT* value active when that node was executed. The *INCARNATION_COUNT* value is fixed until rollback occurs—then we create a node for the new instance of restored state, and add it as a sibling of its earlier instance. The edge from its parent to the new node is labeled with the new *INCARNATION_COUNT* value.

The path for node $A$ is just the sequence of pairs of node and edge labels

$$(N_0, E_0), ..., (N_{k-1}, E_{k-1})$$

necessary to reach $A$ from the root. Figure 4.30 shows the labelling on a timetree for a computation that rolls back twice.

We make a couple of observations:

- *Paths are sufficient to sort nodes.* If $A$ and $B$ are two nodes in the user timetree, $A$ precedes $B$ iff the path for $A$ is a prefix of the path for $B$.
- *Paths can be greatly condensed.* The $i$th node label in a path is the integer $i - 1$. The $i$th edge label in a path is the same as the label on edge $i - 1$, unless edge $i$ leads to a node restored by rollback. Unless the computation has rolled back to initial conditions, all paths start with $(0, 0)$.

Let $A$ have *USR_COUNT* $= k$. Then the path from the root to $A$ has the form

$$(0, E_0), ..., (k - 1, E_{k-1})$$

We condense this path by deleting $(i, E_i)$ if $E_i = E_{i-1}$, and deleting the leading pair if it is $(0, 0)$. The *USR_NAME* of $A$ consists of the *USR_COUNT* value, and this condensed path. (Figure 4.30 shows this construction.)

To compare two user nodes in the timetree at a process, we check whether one node's path is a prefix of the other. Suppose $k, P$ and $k', P'$ are the *USR_NAME* values for nodes $A$ and $A'$, and $k < k'$. We determine if $A \longrightarrow A'$ by removing from $P'$ any pairs $(m, E_m)$ with $m \geq k$, and then checking if the resulting list is identical to $P$.

The length of the condensed path in the *USR_NAME* for node $A$ is proportional to the number of rollbacks in the path from the root to $A$. If failures occur, this will not be constant; thus, for the implementation we sketched above, USER_PARTIAL_ORDER timestamp vectors will not be linear. The size instead will be proportional to the number of rollbacks in the SYSTEM_PARTIAL_ORDER past of the nodes in the vector.

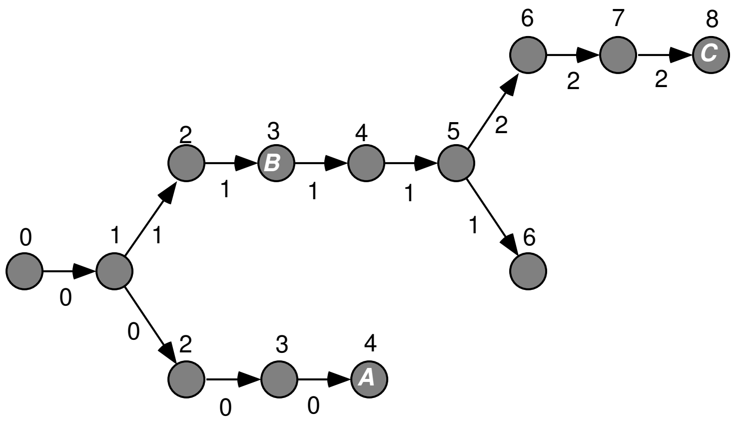


**Figure 4.30** Path information allows sorting in user timetrees. In this sample tree, we have labeled nodes with their *USR_COUNT* value and edges with their *INCARNATION_COUNT* value. The *USR_NAME* for $A$ is $4, \emptyset$; for $B$ is $3, \{(1,1)\}$, and for $C$ is $8, \{(1,1),(5,2)\}$. We can determine that $A$ does not precede $C$, since the condensed path for $A$ does not equal $\{(1,1)\}$ (the condensed path for $C$ trimmed for $A$). We can determine that $B$ precedes $C$, since the condensed path for $B$ does equal the condensed path for $C$, trimmed for $B$.

We can reduce the amortized length of *USR_NAME* values by having processes try to avoid transmitting redundant data. Suppose process $p$ wants to send the *USR_NAME* of $A$ to process $q$. Instead of sending the path from the root to $A$, process $p$ can send the path from an intermediate node $B$ to $A$. If process $q$ already knows the path from the root to $B$, then process $q$ quickly reconstructs the full path. If not, process $q$ recognizes that it is missing data and blocks until it can obtain it.

One example of this amortization technique is using a heuristic similar to Strom and Yemini's approach. Each time a process rolls back, it broadcasts the path to that rollback node along with its new incarnation count. Subsequent *USR_NAME* values consist solely of *INCARNATION_COUNT* and *USR_COUNT*. (This heuristic introduces blocking into our protocol, but still maintains the at-most-once lower bound on rollbacks at a process.) However, a wide range of other heuristics exists for this technique. At one extreme, process $p$ transmits only the end of the path; at the other extreme, process $p$ maintains the most recent system timestamp vector received from $q$, and uses the $q$ entry as the intermediate node for a name sent to $q$.

Commitment and garbage collection may integrate nicely with these amortization techniques, since processes may maintain a *log vector* of the maximal known logged nodes at other processes.

### 4.3.5. Piecewise Determinism and State Intervals

The presentation of our protocol allowed transitions between states to be nondeterministic. As Section 4.1.5 observed, providing complete recoverability under this model requires logging every transition. However, realistic distributed systems often guarantee that process execution is deterministic between message receives; models for these systems focus on state intervals instead of states. With a few modifications, our protocols adapt to this environment.

The framework of Section 2.2.1 can express piecewise determinism by restricting processes to blocking receives—that is, if a process attempts to receive a message, it pauses until a message is available. (This approach contrasts with more flexible polling or interrupting approaches.) We require all other process transitions to be deterministic. A *state interval* is the period of deterministic execution between successive receive events. We can build a simple time model to collect the sequence of nodes between successive receives into a state interval node.

The coarser granularity of state intervals makes logging and replay easier. However, this granularity also changes how rollback should affect the mapping between the system and user time models. In the state model, when a process restores state $A_U$, it establishes a sibling of $A_U$ in the user timetree. However, this approach does not work for state intervals, since the state at other processes may depend *directly* on $A_U$, rather than indirectly through a subsequent state transition node. Restoring a *sibling* of $A_U$ incorrectly makes the state at these processes orphans.

The solution to this problem is for rollback to restore state interval $A_U$ itself, not a sibling. The interval following the re-execution of $A_U$ begins the new timetree branch. Consequently, the set of system interval nodes that a user interval node represents is not necessarily a connected sequence. This fact has some implications for Section 4.2.5. Our graphical shorthand no longer applies, since Theorem 4.1 no longer holds; however, we can still establish a weaker version of Theorem 4.1.

**Theorem 4.5** Suppose $A_U$ and $B_U$ are user state intervals. Let $B_S$ be any system state interval corresponding to $B_U$, but let $A_S$ be the *minimal* system state interval corresponding to $A_U$. If $A_U \rightleftarrows B_U$ then $A_S \rightleftarrows B_S$.

*Proof* This result follows from induction on precedence paths. If $A_U$ and $B_U$ occur at the same process, then the result easily follows. If $A_U$ sends a message that begins $B_U$, then some system state interval following or equaling $A_S$ must also precede $B_S$.

For more general paths, choose an intermediate node $C_U$ with $A_U \longrightarrow C_U \longrightarrow B_U$, and let $C_S$ be the minimal system state node corresponding to $C_U$. Establish the result for $A_U$ and $C_U$, and then for $C_U$ and $B_U$. $\square$

Theorem 4.4 holds for state intervals, since we may substitute Theorem 4.5 for Theorem 4.1 in its proof.

### 4.3.6. Comparison to Related Work

**Checkpointing** As we note in Section 4.1.3, recovery protocols based on checkpointing restore the system to a recovery line composed of local checkpoints. Organizing recovery lines into an increasing sequence (e.g., [BCS84, Ci89]) may allow asynchronous recovery and may tolerate concurrent failures (since one recovery line will clearly be earliest). More complex structures of recovery lines require more synchronization upon recovery, but may allow some surviving processes to proceed without rolling back. However, unless every adjusted rollback vector is a recovery line, checkpointing-based recovery will force surviving processes to roll back computation that does not depend on the computation lost due to failure

The distinction between checkpointing-based protocols and the message logging family sometimes blurs. (Johnson [Jo93] presents a protocol that is explicitly hybrid.) A checkpointing scheme in which processes checkpoint every local state to stable storage before proceeding would be similar to pessimistic rollback; a checkpointing scheme in which processes checkpoint every local state to volatile storage (and eventually to stable storage) would be similar to optimistic rollback. Our protocol adapts to this latter environment.

Ciuffoletti [Ci84] proposed a checkpointing protocol for *synchronous* communication: with each message, processes use a heavy-weight scheme to exchange history and checkpoint information between sender and receiver. Although some aspects of this scheme foreshadow the user and system levels in our work, Ciuffoletti's protocol is inherently synchronous, and the model of synchronous communication does not apply to realistic distributed systems.

**Optimistic Rollback Recovery** Strom and Yemini [StYe85] initiated the area of optimistic rollback recovery. They presented optimistic techniques for surviving processes to ensure complete recoverability, and a rollback protocol[6] that allows processes to recover mostly asynchronously, although delayed transmission of incarnation start information may cause blocking. This protocol implicitly uses partial order time to track dependency on failed computation (and, to our knowledge, is the the earliest publication of the timestamp vector mechanism).

However, Strom and Yemini did not consider the flow of knowledge of rollback. They consequently built an orphan test that is strictly weaker than ours. Their protocol never falsely concludes that a non-orphan state is an orphan. However, their protocol will falsely conclude that some orphan states are not orphans—even when the testing process could potentially know otherwise. These false negatives make it possible for a single failure at one process to cause another process to roll back an exponential number of times, since the unfortunate process never rolls back far enough (until the last time). Sistla and Welch [SiWe89] claim an $O(2^n)$ upper bound for the worst case in the Strom and Yemini protocol. We prove an $\Omega(2^n)$ lower bound by construction in Figures 4.31 through 4.33.

[6]In some sense, Merlin and Randell [MeRa78] foreshadowed Strom and Yemini's work by presenting a protocol based on a representation similar to Petri Nets; this protocol could be transformed and optimized into one similar to Strom and Yemini's.

Johnson and Zwaenepoel [Jo89, JoZw90] developed a general model for optimistic rollback recovery. They used state lattices from partial order time to show that a maximal recoverable system state exists, and presented synchronized protocols to recover this state—even without reliable message delivery. Sistla and Welch [Se89] presented two protocols for optimistic recovery that avoid the exponential worst case by using synchronization during recovery; like Strom and Yemini, Sistla and Welch require reliable FIFO message channels. Peterson and Kearns [PeKe93] recently presented a recovery protocol using vector clocks that synchronizes during recovery by passing tokens.

**Summary** Optimistic rollback protocols improve on other recovery methods by requiring little synchronization during failure-free operation and by requiring only the theoretical minimum amount of computation to be rolled back (only the computation that depends on the computation lost due to failure). Our protocol improves on previous optimistic rollback protocols by providing both *completely asynchronous* recovery and a worst-case upper bound of at most one rollback at each process. The key to asynchronous optimistic rollback recovery is the realization that two levels of partial order time abstraction are relevant: causal dependency on rolled-back events and potential knowledge of rollbacks. Our distributed time framework allows us to explicitly track these two levels of time. We improve even on the explicit "vector time" work of Peterson and Kearns by truly using the full power of temporal abstraction.

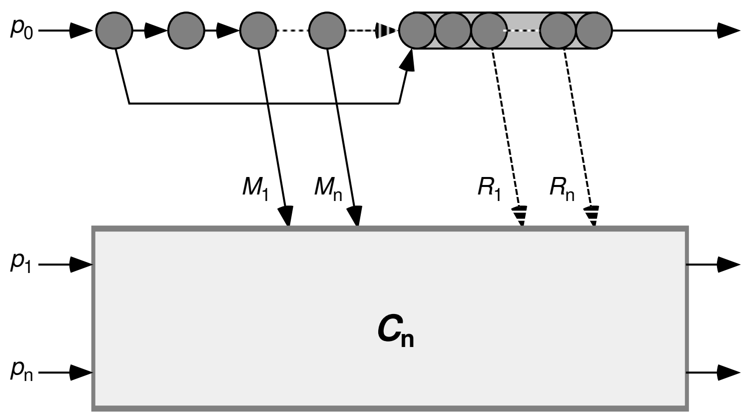


**Figure 4.31** The failure of one process may lead to $\Omega(2^n)$ rollbacks using Strom and Yemini's protocol. This diagram shows how to construct computations exhibiting this behavior. We build this computation inductively. This diagram shows the hypothesis: the existence of a computation $C_n$ on $n$ processes that accepts $n$ user messages $M_1, ..., M_n$, then $n$ system messages $R_1, ..., R_n$ announcing the rollback of the send events of the user messages. We assume that the single failure at process $p_0$ triggers $2^n - 1$ failures in computation $C_n$, and that $2^{n-1}$ of these failures occur at process $p_n$. Figure 4.32 shows how to build computation $C_{n+1}$ from two copies of computation $C_n$. Figure 4.33 shows the base for $n = 1$.

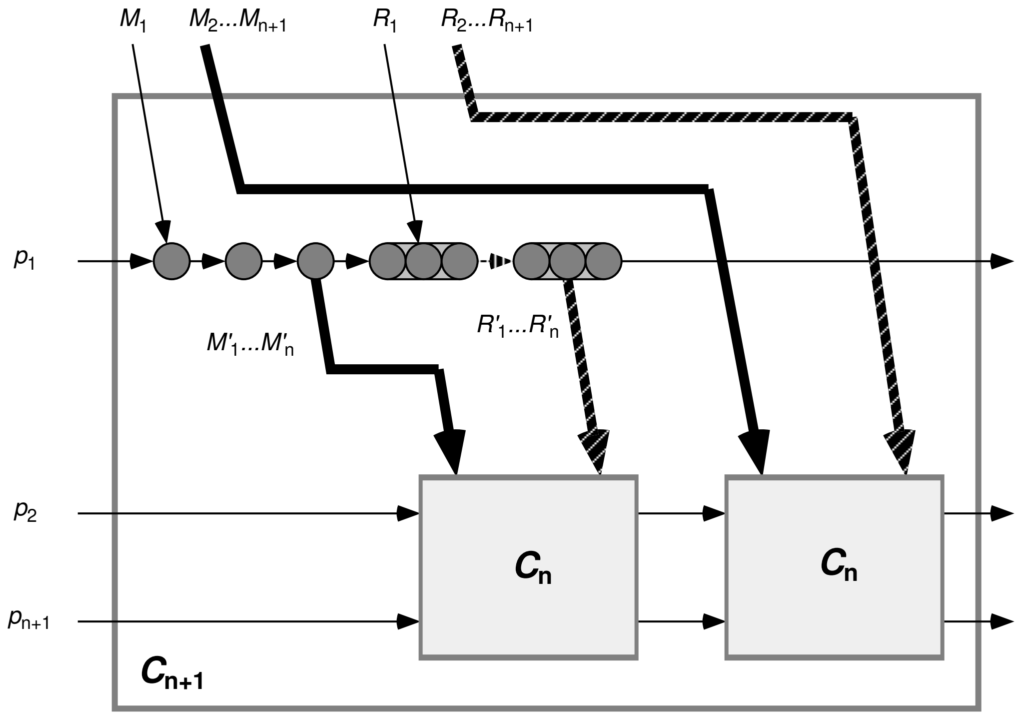


**Figure 4.32** This diagram shows how to build $C_{n+1}$ from two copies of $C_n$. Computation $C_{n+1}$ receives $n+1$ user messages $M_1, ..., M_{n+1}$, then receives $n+1$ system messages $R_1, ..., R_{n+1}$ announcing the rollback of the user send events. Process $p_1$ receives $M_1$, establishing a dependency, then sends $n$ messages $M_1', ..., M_n'$ to the first copy of $C_n$. This establishes dependency on $p_1$, and transitive dependency on the send event of $M_1$. Process $p_1$ then receives rollback announcement $R_1$, rolls back, and sends the announcements out to the first copy of $C_n$. The $n$ remaining user messages $M_2, ..., M_{n+1}$ are then fed directly to the second copy of $C_n$, followed by the remaining rollback announcements. (We cannot repeat the $p_1$ trick since $p_1$ now knows about the initial failure. However, processes $p_2$ through $p_n$ only know about the failure at $p_1$.) The assumption that $C_n$ rolls back $2^n - 1$ times puts the number of rollbacks in $C_{n+1}$ at $2(2^n - 1) + 1 = 2^{n+1} - 1$. The assumption that the last process in $C_n$ rolls back $2^{n-1}$ times gives $2(2^{n-1}) = 2^n$ rollbacks at $p_{n+1}$ in $C_{n+1}$. Hence this construction establishes the induction.

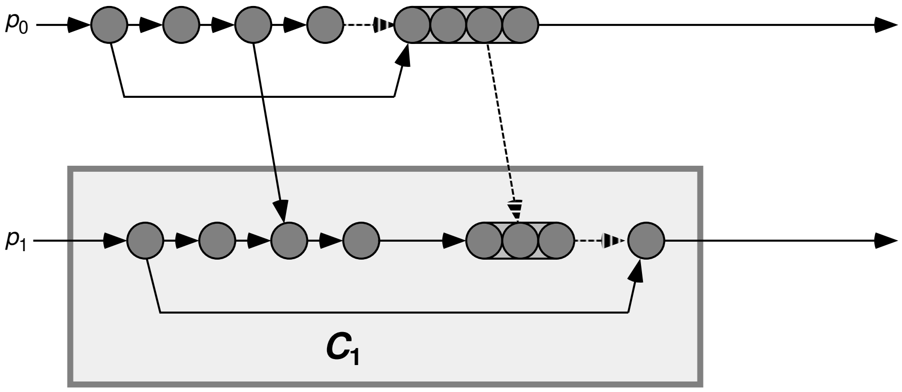


**Figure 4.33** This diagram shows the construction of $C_1$, the base for the inductive construction of Figure 4.32.

## 4.4. A General Framework

From a high level, a rollback protocol consists of the initiating process requesting that the system roll back to some point, and each of the other processes receiving this request and cooperating. This sketch raises some questions:

- How does the initiating process specify the state to be restored?
- How should the other processes react?

The protocol in Section 4.3 (as well as most of those in the literature) uses the *current past-current past* (CP-CP) paradigm: the initiator chooses a state that it currently regards as being in its past (that is, a state USER_PARTIAL_ORDER-preceding the decision to roll back), and the other processes each choose a state from their current pasts.

The CP-CP paradigm has the advantage of being well-defined. Suppose the system is currently consistent, and that the initiating process restores state $A$. Then the adjusted rollback vector $\mathbf{R}^*(A)$ (from USER_PARTIAL_ORDER) will be consistent and concurrent with restored $A$ (and subsequent computation). When recovery is complete, the virtual FAILURE_FREE_PARTIAL_ORDER computation will consist of the portion of the initial failed computation preceding $\mathbf{R}^*(A)$, with the revised computation appended from there.

Phrasing rollback in terms of the CP-CP paradigm immediately suggests alternative paradigms: letting the initiator and/or the other processes choose from their *general pasts*. Allowing the initiating process to restore any state from its timetree permits the flexibility of rolling back rollback. (The implementation sketch of Section 4.3.4 allows sorting of events in user-trees even if the trees grow through general-past rollback.) We can construct scenarios where this might be useful. For example, suppose process $p$ has been performing some valuable computation in silence. The system assumes $p$ has failed and restarts a new version—but when the old version speaks up, the system decides it would prefer to discard the rollback and re-incorporate the old $p$.

As we have seen in Section 4.2.6, allowing both the initiator and the others to choose states from their general pasts permits ambiguity. Implementing this approach in a distributed fashion is difficult: processes must disjointly choose consistent paths. (Figure 4.34 shows an example.) Constraining the other processes to choose from their current pasts (but consistently with the general past state chosen by the initiator) also creates problems. For example, the other processes may not be able to choose states that permit the initiator's choice to exist. Figure 4.35 shows one such situation.

One interesting avenue for future work lies in having the initiator choose not a state but a predicate describing a system state it would like restored. (Of course, such an approach requires that the predicate is satisfiable.)

Another interesting avenue is to implement general-past rollback by formally rolling back rollback. We might build a third level of partial order time to express the meta-recovery computation, and use our earlier protocols to roll back the recovery computation that performed the original rollback.

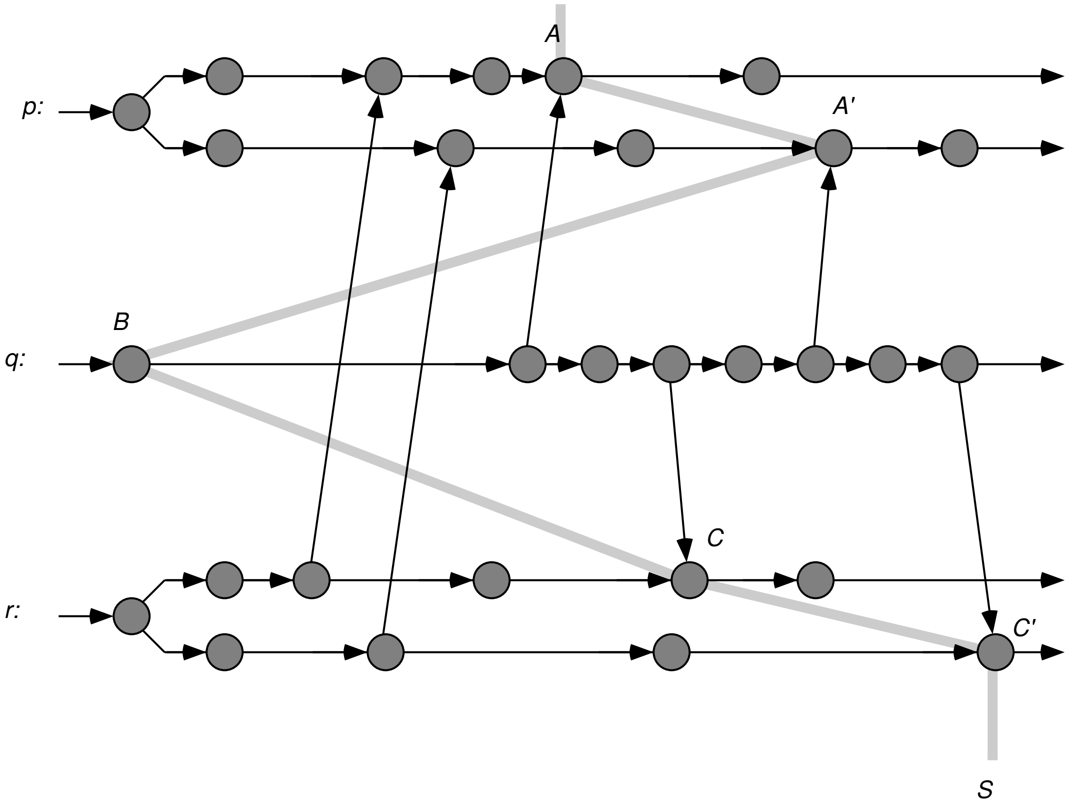


**Figure 4.34** During rollback recovery, allowing processes other than the initiator to choose from their general pasts creates difficulty. For example, suppose process $q$ decides to roll back $B$. The naively defined rollback pseudo-vector of $B$ is the set $S$. Since $S$ touches multiple branches at processes $p$ and $r$, allowing these processes to restore states from their general pasts creates ambiguity: e.g., if process $p$ chooses its $A$ branch and process $r$ chooses its $C'$ branch, then the resulting system state will not be consistent. For system consistency, processes $p$ and $r$ must both choose their primed branches or both choose their unprimed branches.

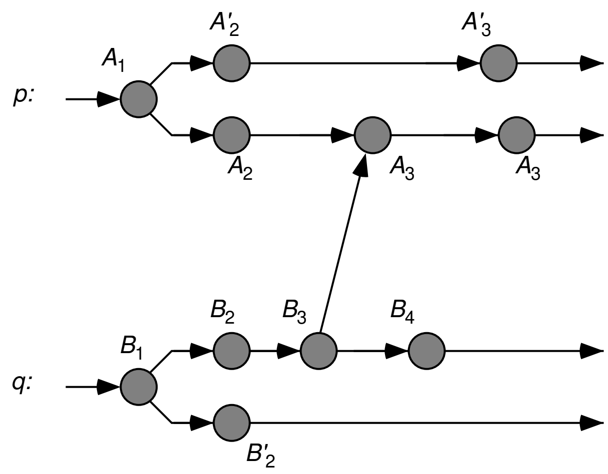


**Figure 4.35** During rollback recovery, allowing the initiator to choose from its general past while constraining other processes to their current past creates difficulty. For example, suppose the current virtual computation has frontier $A_3', B_2'$, but process $p$ wishes to restore the state $A_3$. No state at process $q$ in the USER_PARTIAL_ORDER past of $B_2'$ is both consistent and concurrent with process $p$'s new state.

# Chapter 5

# Security and Privacy for Distributed Time

## 5.1. Overview

Systems of time more general than the linear order of real time are central to solving application problems in asynchronous distributed systems. Since protocols for these applications require examining the underlying distributed time models, explicitly providing a distributed time service simplifies and clarifies the task of protocol design.

However, while real time can be determined from an independent physical device, relations such as partial order time cannot be determined in isolation. Tracking relations such as the PARTIAL_ORDER_TIME model requires collecting and sharing information; tracking relations in time models that dispense with transitivity or permit cycles involves even more local information. Thus dealing with distributed time exposes protocols to security risks. Is the information a process receives correct? Can shared information be used for dishonest purposes?

Encapsulating a system's dealings with partial order time into a single time service provides an arena to examine and resolve security and temporal issues for protocol design.

The proposal document [Sm91] for this thesis recognized the central role of partial order clocks, cataloged some of the security and privacy risks, and gave the original presentation of the *Signed Vector Timestamp* protocol, which protects against some of these risks. While this protocol prevents dishonest processes from forging causal dependence on nodes at honest processes, it suffers from some drawbacks:

- The Signed Vector Timestamp protocol cannot guarantee detection of causal paths touching dishonest processes. Consequently, Signed Vectors cannot be used to build secure protocols for problems such as *distributed snapshots* requiring accurate detection of non-precedence.
- The Signed Vector Timestamp protocol leaks private information, since vector entries are publicly readable.
- The Signed Vector Timestamp protocol requires each process to check $n$ signatures.

- The Signed Vector Timestamp protocol requires that the temporal relation being tracked express all paths of information flow; thus the protocol does not extend to more general relations (such as USER_PARTIAL_ORDER from Chapter 4).

**This Chapter** In this chapter, we use new developments in inexpensive tamper-proof hardware to build the *Sealed Vector Timestamp* protocol, which provides stronger security and privacy protection than any previous protocol. Sealed Vectors solve previously open problems by preventing dishonest processes from forging dependence on *any* events, and by preventing dishonest processes from denying dependence (if malicious processes cannot communicate covertly). (Even with covert communication, Sealed Vectors provide some protection against denying dependence.) Sealed Vectors also move beyond previous work by addressing privacy risks, and by providing secure clocks for partial orders where information flow does not imply precedence.

The proposal document opened up this area of research. This chapter presents the most secure protocols to date, and solves problems other researchers left open [ReGo93]. Section 5.2 discusses the inherent security and privacy risks for partial order time. Section 5.3 surveys the defenses and presents our new protocol. Section 5.4 discusses our new protocol, and Section 5.5 considers some directions for future research. For clarity of presentation, most of this chapter considers the problems of tracking temporal relations in a Type 4 parallel pair such as $\langle$PARTIAL_ORDER_TIME, TIMELINES$\rangle$. Chapter 6 will consider the implications of this work for more general time models.

(Preliminary versions of some this material appeared in earlier publications [SmTy91, SmTy94].)

## 5.2. Security and Privacy Attacks

Partial order time draws on data distributed throughout the system. Consequently, building partial order clocks requires that processes share private information, and trust the private information shared with them. This opens opportunities for Byzantine (malicious) processes to manipulate the clock protocols, and consequently to manipulate application protocols built on these clock protocols.

We sketch four such attacks on vector clocks.

**Nonsense Attacks** Malicious processes can send arbitrary vector entries. Since honest processes will dutifully copy and pass on these values, a single act by a single malicious process can destroy the validity of many vectors throughout the system. (Lamport total order clocks [La78] are particularly vulnerable to these attacks.) Simple sanity checks fail to combat this problem. Suppose vector entries are integers. If honest processes refuse to accept vector entries that have increased more than $N$, a dishonest process can repeatedly increase an entry by $N - 1$. The next honest process the victim talks to may then mistakenly identify the honest victim as corrupt.

**Malicious Backdating** Malicious processes can selectively reduce vector entries, and thus fool honest processes into thinking events happened earlier than they actually did. Consider the application of trading commodities options on a public network. Figure 5.1 shows how *Malicious Backdating* permits the crime of *options frontrunning*, which can occur when brokers may trade both for themselves and for their clients. (One place where options frontrunning occurs is the Chicago commodities exchange.) If a broker happens to buy a small quantity of shares for himself before his client requests a large number of shares, then the broker will make a tidy sum. Consequently, on receiving a client request, a dishonest broker has incentive to issue a request of his own that appears not to have followed the client request. In an electronic exchange using vector clocks, a malicious broker can do this by re-using an old vector on his purchase request.[1]

**Malicious Postdating** Malicious processes can selectively inflate vector entries, and thus fool honest processes into thinking events happened later than they actually did. Figure 5.2 shows how such *Malicious Postdating* permits *insider trading*. A malicious process can send a cohort an advance copy of an announcement *along with an advanced vector*. The cohort can act on this

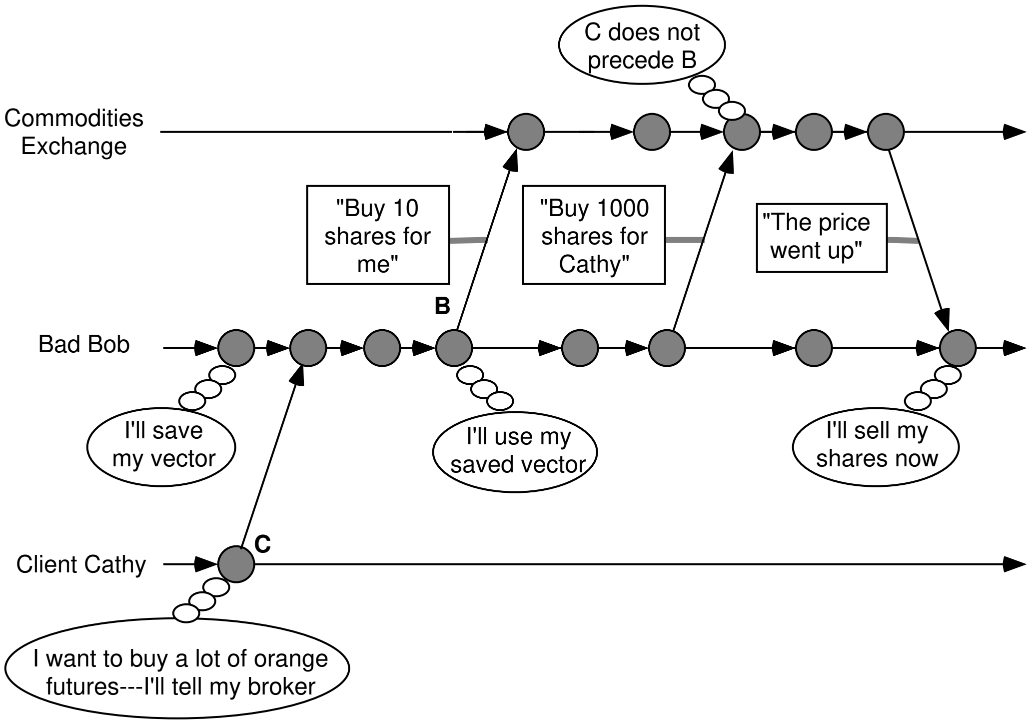


**Figure 5.1** Malicious processes can selectively backdate nodes. Here, *Bob* commits the crime of *options frontrunning* by making his own purchase appear not to follow his client's request.

[1]In the physical Chicago exchange, the only defense the FBI has against options frontrunning is placing undercover agents in the pit to look for unusually lucky brokers.

data, but use the advanced vector to hide her headstart. (The cohort could even be unwitting; the malicious process might frame her now, in order to spread the blame should the ruse be discovered later.)

**Compromised Privacy** Malicious processes can correctly perform the vector clock protocol, but use the vector entries to gain illicit knowledge. Figure 5.3 shows how this technique reveals anonymous whistleblowers. Changes in subsequent timestamp vectors sent from *Alice* to *Bob* show the identities of processes communicating with *Alice*.

## 5.3. Defenses

An ideal clock should report "$A \longrightarrow B$" exactly when $A$ precedes $B$, even if processes perform malicious actions. An ideal clock should also confine private information. We can evaluate clock protocols by this standard: against decreasing amounts of honesty, how well do clocks perform?

Many application protocols use forms of partial order time and vector clocks. A clock protocol meeting this ideal transparently protects higher-level applications against the security and privacy risks of Section 5.2.

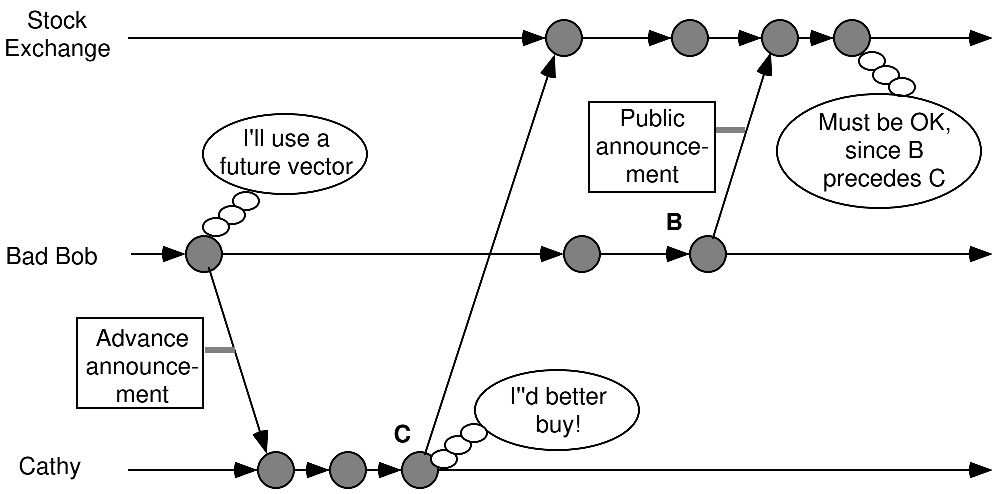


**Figure 5.2** Malicious processes can selectively postdate nodes. Here, *Bob* leaks an advance copy of his public announcement to *Cathy* in such a way that allows her to act on the data first, without appearing to have had a headstart.

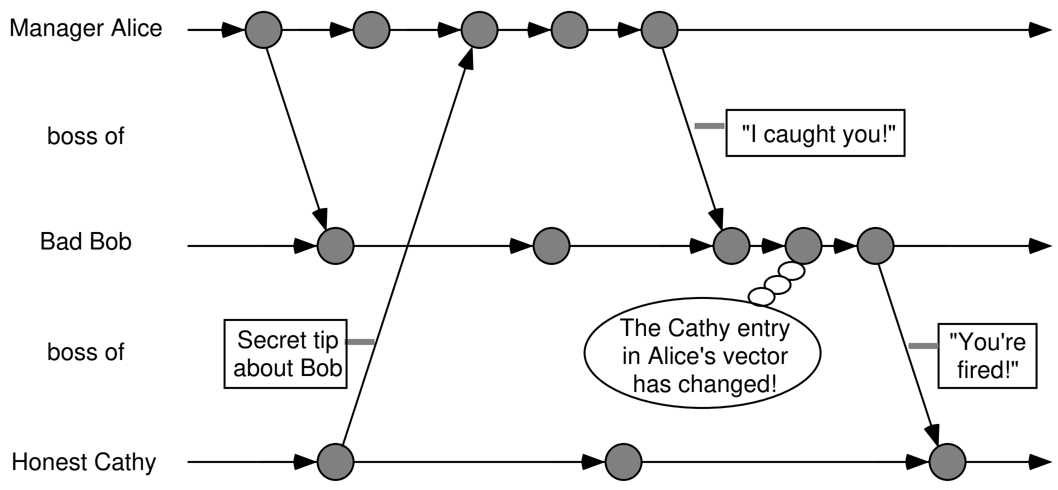


**Figure 5.3** Malicious processes can exploit vector data for illicit purposes. Here, *Bob* uses the timestamp vectors from *Alice* to learn the identity of whistleblower *Cathy*.

### 5.3.1. Previous Work

If all processes are honest, then the process $p$ entries in all vector timestamps originate at process $p$. Our *Signed Vector Timestamp* protocol [SmTy91, ReGo93] builds on this observation by requiring each process to digitally sign[2] its entries in outgoing timestamp vectors. That is, the process $p$ entry in a timestamp vector now consists of the name of a node at process $p$, *and* a signature from $p$ on that name. This scheme prevents malicious processes from advancing vector entries belonging to honest processes. If an event $A$ occurs at an honest process and our time model expresses all information flow paths, then possession of a signed entry for $A$ is proof of dependence on $A$. With Signed Vectors, $A \longrightarrow B$ when an honest clock reports "$A \longrightarrow B$" (and $A$ occurs at an honest process). If all processes along a precedence path from $A$ to $B$ are honest, the converse is also true: an honest clock reports "$A \longrightarrow B$" when $A \longrightarrow B$.

However, Signed Vectors may fail if precedence paths go through malicious processes. For example, a malicious process can use old values in the vector entries for honest processes, as long as the malicious process has retained the matching signatures. Signed Vectors still permit the Malicious Backdating and Malicious Postdating attacks. Signed Vectors do not even attempt to address the Compromised Privacy attack. These problems migrate to higher-level applications. Inability to detect non-precedence reliably can result in inefficiency (in *optimistic rollback recovery*, processes may mistakenly conclude they depend on failed states) or complete incorrectness (in *global state* protocols, processes may make incorrect decisions regarding "concurrent" events).

[2]Section 5.3.3 will discuss *digital signatures* in more detail.

The security of the Signed Vector protocol depends on the fact that precedence paths and information flow paths coincide. If precedence and information flow do not coincide, then Signed Vectors do not provide secure clocks. For example, consider the partial order describing the virtual computation arising after rollback with modified replay.

Three additional protocols exist for the special case of a process sorting the send events of two messages it has received [ReGo93]. The *Piggybacking* protocol generalizes the vector timestamp protocol by timestamping each event $E$ with a signed record of all messages whose *send* events precede $E$. Piggybacking (like Signed Vectors) ensures that if a clock reports "$A \longrightarrow B$" and $A$ occurs at an honest process, then $A \longrightarrow B$; Piggybacking further limits the possible actions of a dishonest $A$ process conspiring to make a clock falsely report "$A \longrightarrow B$." However, the Piggybacking protocol (also like Signed Vectors) cannot reliably detect precedence paths touching malicious processes, and does not address the issue of privacy. The other two protocols from [ReGo93] alter the order in which messages are received. These protocols address the problem of detecting the partial order by changing the partial order; further, they do not accurately report non-precedence. The *Conservative* protocol requires that before sending a new message, a process wait for acknowledgements of any previous messages it sent. The *Causality Server* protocol assumes secure FIFO channels, and relies on a trusted central intermediary to impose a total order on all message traffic.

### 5.3.2. The Sealed Vector Timestamp Protocol

The *Sealed Vector Timestamp* protocol has security properties that solve previously open problems:

- Our protocol accurately reports "$A \longrightarrow B$" or "$A \not\longrightarrow B$," in the presence of arbitrary malicious processes (including the $A$ process).
- Our protocol does not leak private information.

The Sealed Vector Timestamp protocol satisfies the ideal (assuming no covert channels), and protects privacy of vector entries as well. Further, this protocol extends to time models where information flow does not imply precedence. Table III compares our new protocol to previous work.

**Overview** Our new protocol rests on the the technology of *secure coprocessors* [TyYe93, Yee94]: inexpensive physically secure devices with a CPU, ROM, and non-volatile RAM. A host processor interacts with its secure coprocessor through formal I/O channels. Any other method of determining the internal state of the coprocessor—including physically penetrating the hardware—results in the erasing of RAM and CPU registers. Secure coprocessors are being deployed rapidly; commercial secure coprocessor products are available from IBM ($\mu$ABYSS [Wein87], Citadel [WWAP91]), and have been announced by other vendors including National Semiconductor [Va94], Semaphore,

| | path from $A$ to $B$ is honest | $A$ is honest | no one (but you) is honest |
|---|---|---|---|
| "$A \longrightarrow B$" $\Longrightarrow\ A \longrightarrow B$ | *Signed, PB, Sealed* | *Signed, PB, Sealed* | *Sealed* |
| "$A \longrightarrow B$" $\Longleftarrow A \longrightarrow B$ | *Signed, PB, Sealed* | *Sealed* | *Sealed* |
| "$A \longrightarrow B$" $\Longleftrightarrow A \longrightarrow B$ | *Signed, PB, Sealed* | *Sealed* | *Sealed* |
| privacy of data | *Sealed* | *Sealed* | *Sealed* |

**Table III** This table compares how, against decreasing amounts of honesty, partial order clock protocols meet the clock ideal: reporting "$A \longrightarrow B$" $\Longleftrightarrow A \longrightarrow B$ while protecting the privacy of vector entries. *Signed* denotes the Signed Vector Timestamp protocol; *Sealed* denotes the Sealed Vector Timestamp protocol; *PB* denotes the Piggybacking protocol.

Telequip, and Wave Systems. Various protection technologies exist. For example, IBM wraps circuit boards in nichrome wire and then seals them with an epoxy mixture chemically stronger than the wire. A detection circuit monitors the resistance of this wire wrapping; penetration attempts will disrupt the wire wrapping and alter the resistance (e.g., by shorting the wire or by cutting it).

Secure coprocessors only possess *limited* amounts of power. We cannot secure an entire workstation—even if we could, we could not secure the user. Bootstrapping from this small amount of physical security into full protocol security raises subtle issues. For example, malicious processes might attempt to bypass coprocessors, or to attack communication lines. (Recent work [TyYe93, Yee94] shows how to protect against these attacks.)

In the Sealed Vector Timestamp protocol, each process runs on a host processor with a secure coprocessor. The secure coprocessor creates timestamp vectors and *seals* them so that processes cannot read them. Although processes can store and exchange timestamps, they need to query a secure coprocessor in order to compare them.

The security of Sealed Vectors follows from a number of properties:

- No party (except a secure coprocessor) can obtain information about the contents of any vector entry from a sealed timestamp, even if the party knows the other entries.
- All processes must route incoming and outgoing messages through secure coprocessors.
- A secure coprocessor must be able to verify that a timestamp was properly sealed by another secure coprocessor.

- Given a sealed timestamp and an event, a secure coprocessor must be able to verify that they match.

### 5.3.3. Cryptographic Tools

We build a timestamp scheme meeting this description using two common cryptographic tools: *digital signatures* and *bit-secure public key cryptography* [DiHe76, RSA78]. A digital signature is a function $S$ from a value space to a signature space meeting the following conditions:

- Given a value $v$ and a signature $s$, any party can determine whether $s$ is a valid signature of $v$: whether $S(v) = s$.
- However, it is intractable for any party (except the privileged signing party) to take a set of value-signature pairs and produce a pair not in this set.

Public key cryptography consists of a function $E$ (from the plaintext space to the cipherspace) and a function $D$ (from the cipherspace to the plaintext space) meeting the following conditions:

- For any plaintext value $v$, any party can calculate $E(v)$.
- For any plaintext value $v$, $D(E(v)) = v$.
- It is intractable for any party (except for the privileged decrypting party) to take a set of plaintext-ciphertext pairs and produce a pair not in this set.

Standard public key cryptography requires only that inverting $E$ is difficult (without the privilege of knowing $D$). Bit-secure public key cryptography requires an additional level of security. Roughly speaking, from a given ciphertext, a malicious process should gain no information about the plaintext that it did not know *a priori*. ([Gold89] presents formal definitions.) Some popular cryptosystems (like [Ra79] and [RSA78]) are known to leak number-theoretic properties of the plaintexts and thus fail to meet this condition [ACGS88, Li81]. For the Sealed Vector protocol to attain its full security potential, it should be implemented using strong cryptosystems such as [BlGo84] or [GoMi82].

**Operation** We use cryptography and signatures both on messages ($E_{\text{msg}}$, $D_{\text{msg}}$ and $S_{\text{msg}}$) and on timestamps ($E_{\text{tst}}$, $D_{\text{tst}}$ and $S_{\text{tst}}$).[3] Each process $p$ has a name, which we denote as $p$. Each process $p$ runs on a host processor with a secure coprocessor, which we denote as $p_{\mathsf{SC}}$. Each secure coprocessor knows that name of its process.

[3]This presentation assumes global schemes for all processes. In practice, giving each process its own key scheme adds flexibility and another level of security; Section 5.4.2 discusses these issues.

Let $\mathcal{P}$ be the set of process names, let $\mathcal{E}$ be the set of event names, let $\mathcal{V}$ be the set of possible timestamp vectors, and let $\mathcal{M}$ be the set of possible message texts. Let $\mathcal{G}_{\text{msg}}$ and $\mathcal{G}_{\text{tst}}$ be the signatures spaces for messages and timestamps, respectively; let $\mathcal{C}_{\text{msg}}$ and $\mathcal{C}_{\text{tst}}$ be the cipherspaces for messages and timestamps. Our signature and encryption functions act according to these rules:

$$S_{\text{tst}} \; : \; \mathcal{E} \times \mathcal{V} \; \mapsto \; \mathcal{G}_{\text{tst}}$$

$$E_{\text{tst}} \; : \; \mathcal{E} \times \mathcal{V} \times \mathcal{G}_{\text{tst}} \; \mapsto \; \mathcal{C}_{\text{tst}}$$

$$S_{\text{msg}} \; : \; \mathcal{P} \times \mathcal{P} \times \mathcal{M} \times \mathcal{C}_{\text{tst}} \; \mapsto \; \mathcal{G}_{\text{msg}}$$

$$E_{\text{msg}} \; : \; \mathcal{P} \times \mathcal{P} \times \mathcal{M} \times \mathcal{C}_{\text{tst}} \times \mathcal{G}_{\text{msg}} \; \mapsto \; \mathcal{C}_{\text{msg}}$$

The functions $E_{\text{msg}}$ and $E_{\text{tst}}$ are public. Each secure coprocessor $p_{\mathsf{SC}}$ has the ability to calculate $D_{\text{msg}}$, $D_{\text{tst}}$, $S_{\text{msg}}$, and $S_{\text{tst}}$; the coprocessor $p_{\mathsf{SC}}$ also maintains the current process $p$ timestamp vector, which we denote as $V_p$.

**Obtaining Timestamps** Suppose process $p$ wants to obtain a timestamp for its current event $A$. Process $p$ submits the request to $p_{\mathsf{SC}}$, which obtains $\mathbf{V}(A)$ by incrementing the $p$ entry of $V_p$. The coprocessor $p_{\mathsf{SC}}$ then returns the sealed timestamp:

$$T(A) \quad = \quad E_{\text{tst}}\left(A, \mathbf{V}(A), S_{\text{tst}}(A, \mathbf{V}(A))\right)$$

Figure 5.4 illustrates this structure.

The signature plays two roles here. First, it proves that this vector belongs to this event. Secondly, its presence *inside the plaintext* protects against a malicious process guessing the value of the vector, and verifying this guess using $E_{\text{tst}}$.

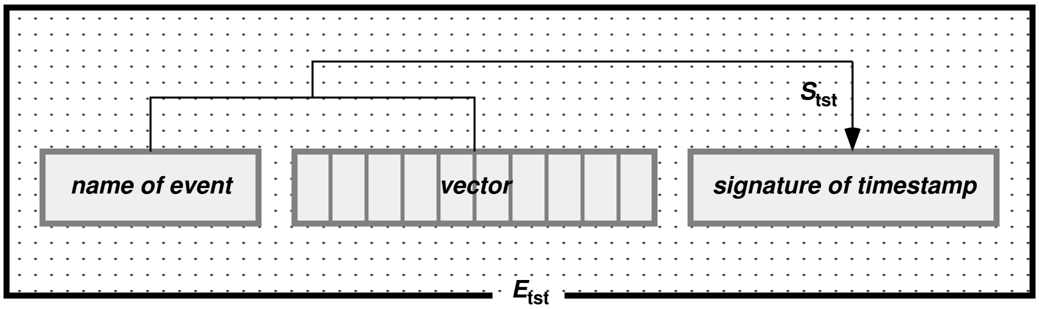


**Figure 5.4** A sealed timestamp consists of the encryption of three items: the name of an event, its timestamp vector, and a signature on this pair. The signature certifies that this vector belongs to this event, and also protects against guessing the plaintext: verifying a guessed vector requires guessing the correct signature.

**Comparing Timestamps** When process $p$ wants to compare events $A$ and $B$, it sends $T(A)$ and $T(B)$ to $p_{\mathsf{SC}}$. The coprocessor applies $D_{\text{tst}}$ to extract the event names, vectors and signatures. If the signatures are valid, the coprocessor then compares $\mathbf{V}(A)$ and $\mathbf{V}(B)$, and reports the result: either "$A \longrightarrow B$," "$B \longrightarrow A$" or "$A \not\longleftrightarrow B$."

**Sending Messages** Suppose process $p$ wants to execute a send event $S$, sending a message with text $M$ to process $q$. Process $p$ submits $M$ and $q$ to the secure coprocessor $p_{\mathsf{SC}}$, which calculates the timestamp[4] $T(S)$, and returns the ciphertext

$$M' \quad = \quad E_{\text{msg}}\,(p, q, M, T(S), S_{\text{msg}}(p, q, M, T(S)))$$

Figure 5.5 illustrates this structure. Process $p$ then transmits the message.

A malicious process might still be able to suppress this message $M$. (For example, in Figure 5.1, *Bad Bob* could have his purchase order sealed, but only introduce it into the network if he receives an order from his client.) The secure coprocessor $p_{\mathsf{SC}}$ can protect against loss by requiring a signed acknowledgement from $q_{\mathsf{SC}}$. If the acknowledgement does not arrive, $p_{\mathsf{SC}}$ can retransmit the message—perhaps incrementally, as part of other sealed packets. A malicious process can successfully suppress a message only by permanently partitioning itself from the network.

**Receiving Messages** Suppose a process $p$ receives a ciphertext message $M'$. To read $M'$, process $p$ needs to send it to the secure coprocessor $p_{\mathsf{SC}}$. The coprocessor applies $D_{\text{msg}}$ to obtain the source and destination process, the plaintext $M$, the timestamp $T(S)$ of the send event, and the $S_{\text{msg}}$ signature of this data. The coprocessor verifies that the $S_{\text{msg}}$ signature is valid and that $p$ is the intended destination process. The coprocessor then applies $D_{\text{tst}}$ to the timestamp, checks its signature, and obtains the vector $\mathbf{V}(S)$. The coprocessor then performs the vector timestamp

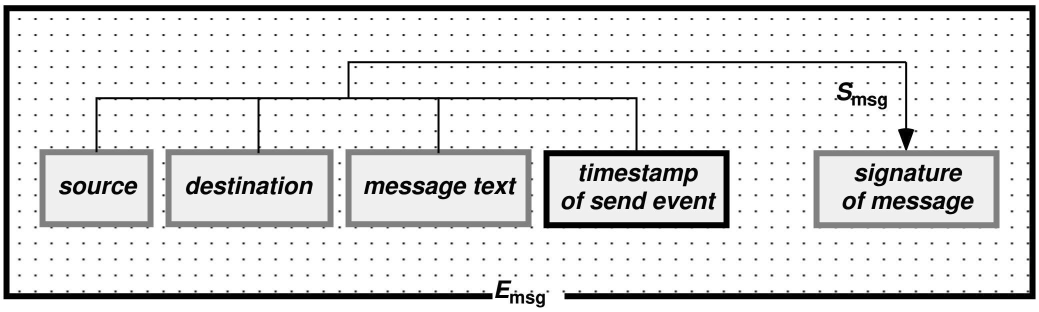


**Figure 5.5** The message ciphertext encrypts the message information (source and destination processes, message text), along with the sealed timestamp of the *send* and a signature of these values.

[4] Since messages are tagged with a signature before encrypting, using the unsealed timestamp $\mathbf{V}(S)$ would suffice here.

protocol: replacing its current vector $V_p$ with the entry-wise maximum of $V_p$ and $\mathbf{V}(S)$. Finally, $p_{\mathsf{SC}}$ returns to $p$ the name of the source process, the plaintext $M$, and (optionally) the timestamp $T(S)$.

## 5.4. Discussion

### 5.4.1. Results

We make some preliminary observations:

- **The coprocessors carry out the vector timestamp protocol.** This follows directly from the description.
- **Only secure coprocessors can unseal messages and timestamps.** A process may be able to guess some or all of the entries of a given timestamp vector. If timestamps were merely vectors encrypted with a public key, then a process could guess a possible vector, encrypt the guess, and compare the result to the ciphertext. However, in our scheme, timestamps are the encryption of a vector along with a signature of that vector. Without knowing the signature function, a process cannot verify that $V$ is the vector in the timestamp $E_{\text{tst}}(A, V, S_{\text{tst}}(A, V))$. Timestamps are truly sealed.

  Similarly, with high probability a process cannot decrypt an encrypted message by making some lucky guesses, since that would require breaking the message signature $S_{\text{msg}}$.
- **Only the secure coprocessor at the source process may seal messages.** Messages arriving at an honest process will be routed to the secure coprocessor, which will ignore messages that do not include both a valid timestamp and a valid signature on the message and the timestamp together.
- **Only the secure coprocessor at the intended destination process may unseal a message.** Sealed messages must be decrypted to be intelligible. The receiving process must consult its secure coprocessor, since the encrypted message includes the name of the intended destination process. (However, a malicious process can receive and discard an encrypted message without consulting its coprocessor. Section 5.4.2 considers this avenue.)

Together, these assertions imply the following result:

**Theorem 5.1** Suppose the following are true statements:

- All messages to or from honest processes are routed through through secure coprocessors.
- The encryption and signature functions are not breakable.

- The integrity of the secure coprocessors is not compromised.

Then Sealed Vectors guarantee the following properties:

- If a clock reports "$A \longrightarrow B$" then $A \longrightarrow B$.
- If node $A$ precedes node $B$ along a path where each message edge touches an honest process, then clocks will report "$A \longrightarrow B$."

*Proof* Let $\beta$ be the PARTIAL_ORDER_TIME graph of this computation. To construct a graph $\gamma$ that reflects the computation perceived by the secure coprocessors, we perform these steps:

1. Copy the entire timeline belonging to each honest process.
2. For each message edge incident to an honest process, copy the edge, and the node at the other end (if it is not already in $\gamma$).
3. Add each node that a dishonest process registers with its coprocessor.
4. At each dishonest process, connect the $\gamma$ nodes in their $\beta$ sequence.

A coprocessor reports "$A \longrightarrow B$" in $\overline{\beta}$ iff $A \longrightarrow B$ in $\overline{\gamma}$. □

**Corollary 5.2** Suppose that, in addition to the hypothesis of Theorem 5.1, malicious processes cannot communicate without using the sealed message protocol. Then Sealed Vectors guarantee that clocks report "$A \longrightarrow B$" iff $A \longrightarrow B$.

*Proof* Construct $\gamma$ as in the proof of Theorem 5.1, only add *all* message edges and their incident nodes (if they are not already in $\gamma$). □

This protocol improves on prior work by offering security advantages:

- **Complete Results** If a clock reports "$A \longrightarrow B$," then $A \longrightarrow B$. If a clock reports "$A \not\longleftrightarrow B$" (and malicious processes cannot communicate using covert channels) then $A \not\longleftrightarrow B$.
- **No Spoofing** Even with covert channels, a malicious process cannot deny having received a message from an honest process.
- **Privacy** The private information shared in timestamps is confined to the secure coprocessors.
- **Wider Application** The Sealed Vector Timestamp protocol does not require that the partial order directly arise from information flow.

In particular, Sealed Vectors protect against *all* the attacks catalogued in Section 5.2, and provide secure clocks for scenarios such as the partial order arising after rollback with modified replay.

Sealed Vectors also improve on Signed Vectors in terms of scalability: the number of decryptions required on incoming messages decreases from linear to constant.

### 5.4.2. Implicit Assumptions

This chapter has made several implicit assumptions open to challenge. We discuss these challenges.

**No Covert Channels** Precedence corresponds to paths through the PARTIAL_ORDER_TIME graph. The Sealed Vector protocol prevents a single malicious process from masking its presence in such paths. However, if malicious processes can communicate without using official (that is, coprocessor-sealed) messages, then they can cooperatively hide their presence in paths—since communication outside of the coprocessors is invisible to the clocks.

One approach to this problem is to make such communication very difficult: for example, by having the secure coprocessors handle net traffic (and perhaps snoop on Ethernet packets), malicious processes would be forced to communicate outside the network.

Covert communication is also possible using *in-band signaling*, since it may be possible to extract information from sealed messages without consulting secure coprocessors. For example, a malicious process might draw conclusions from the existence of the message, the length of the message (real encryption usually breaks long text into blocks and encrypts each block separately) or the frequency of multiple messages.

**Security of Coprocessors** The protocol depends on the physical security of the coprocessors. In practice, secure coprocessors are extremely difficult to penetrate. However, as with any security mechanism (physical or computational), it may be possible to compromise the system if the attacker is willing to pay tremendous amounts of money. (For a detailed analysis of the cost, see [Wein91].) What do we do if the exception case occurs—if a coprocessor is compromised? One way to limit the damage is to use separate $S_{\text{msg}}$, $S_{\text{tst}}$ and $E_{\text{msg}}$ functions for each process. This technique prevents a compromised coprocessor from impersonating someone else or performing message decryption for someone else. Using separate $E_{\text{tst}}$ functions prevents the compromised coprocessor from doing comparisons for someone else, but requires re-encrypting forwarded timestamps. (Section 5.5 considers some further defenses.)

**Validity of Keys** Giving each coprocessor its own keys raises the issue of key management: a new coprocessor must somehow announce its public keys. A straightforward technique to prevent dishonest processes from impersonating a "new coprocessor" is to have new coprocessors obtain certificates, signed by a universally trusted agent, listing their identity and public keys.

## 5.5. Future Work

**Limiting Penetration Damage** What can we do if the integrity of a coprocessor is compromised? Penetration exposes any data that a coprocessor has saved. However, an uncompromised coprocessor can securely *forget* data. This observation suggests an alternative *Give-and-Forget* timestamping scheme. Suppose process $p$ at event $S$ sends a message to process $q$, who receives it at event $R$. Process $p$ generates a key pair $K_{1,S}$, $K_{2,S}$. Process $p$ signs a certificate asserting that $K_{2,S}$ is its public key for event $S$, and sends this certificate along with the private key $K_{1,S}$ to process $q$ with the message. Process $q$ uses the private key $K_{1,S}$ to encrypt an identifier for $R$ and then *erases the key*. Process $q$ then has a universally verifiable certificate that it knew about $S$ when $R$ occurred. However, examining this certificate allows no one—not even process $q$—to forge a new certificate of knowledge of $S$ without the cooperation of process $p$.

This technique allows a secure coprocessor to generate *proof-of-timestamp* certificates showing the last message received from each uncompromised process. Should the coprocessor later be compromised, it cannot produce new certificates for these messages. To prevent a compromised coprocessor from rolling back timestamp entries, we can require all coprocessors to use these proof-of-timestamp certificates to prove the validity of each entry in their timestamp vectors.

Other approaches for pre-compromised coprocessors to limit the forging power of their compromised versions include the *Distributed Trust* and *Digital Timestamping* techniques of [BHS92, HaSt91], as well using data on acknowledgement packets.

**Improving Performance** A performance problem with vector clocks results from size: timestamps have $n$ entries; comparing timestamps requires $n$ comparisons. Charron-Bost's result [CB91] that partial order timestamps must be linear suggests two approaches to improving performance: implementing vector clocks more carefully (to reduce the actual data transmitted), and trading timestamp *size* for comparison *time*.

Singhal and Kshemkalyani [SiKs90] present a vector clock implementation where processes refrain from transmitting redundant data in vectors. Integrating this technique with Sealed Vectors would yield increased efficiency.

A more generalized approach would be to give processes more latitude in choosing which entries to transmit and which to withhold. Some entries in timestamp vectors might be marked with flags indicating that that value is merely a lower bound. This lower bound may suffice for many comparisons; if it does not, a secure coprocessor would need to consult other secure coprocessors to obtain the missing data. It would be interesting to develop good heuristics for deciding which entries to withhold and for determining when the expense of a "miss" outweighs the benefits of withholding.

Another interesting approach would be to implement vector clock protocol in a more centralized fashion. For the extreme case, suppose we had a single trusted *logging site*. When a process receives a message, its secure coprocessor sends a note to the logging site indicating the sending process, the receiving process, and the local indices of the send and receive events. The logging site then has sufficient information to maintain the timestamp vectors for each process. We obtain constant size timestamp data on messages—at the price of doubling the number of messages, and having processes need to consult a remote site to perform comparisons. This approach still requires coprocessor sealing in order to force a process not only to acknowledge receiving a message, but also to file a logging note. (This approach differs from the *Causality Server* protocol [ReGo93] in that messages are not routed through an an intermediary, but logged after the fact, that no FIFO nor secure channel assumptions are needed, and that the logging site protocol preserves the actual partial order, not just a consistent total order.)

Yet another technique (e.g., [ACGS91]) is to use vector clocks to track a coarser partial order—trading timestamp size for false positives in precedence detection. However, adapting these techniques (or the linear timestamping techniques of [BHS92, HaSt91]) creates the problem of proving the *absence* of a precedence path. Developing a hierarchical approach—to indicate the most "likely" precedence path, and then verify its correctness—is one path of future research.

**General Confinement Models** Another area for exploration is the use of more general confinement models. Coprocessor sealing provides control over the information a timestamp provides to a process. This control may provide more benefits than just suppressing vector entries—in particular, it may allow for anonymous or hidden causality [Gr75].

# Chapter 6

# Secure Distributed Time for Secure Distributed Protocols

## 6.1. Overview

Chapters 2 through 4 showed how framing application problems in terms of distributed time provides a deeper understanding of the problems, and allows the development of flexible and general protocols that access the distributed time structure by querying clock primitives. Separating the clocks from the higher-level protocols in this fashion allows us to change the clock implementations transparently to the higher-level protocols. However, the popular timestamp vector implementation of partial order clocks suffers from security and privacy risks, as Chapter 5 discussed.

These security and privacy risks for timestamp vectors create problems for higher-level protocols that use these clock implementations. For example, malicious clients can exploit the security and privacy risks of timestamp vectors in order to subvert the *immediate ordered service* protocol of Section 2.5.2. Standard attacks on timestamp vectors translate to higher-level protocol attacks:

- **Backdating** A malicious process $p$ could ensure that its requests receive undue priority by backdating the timestamp vectors on them.
- **Postdating** Alternatively, a malicious process $p$ could ensure that its requests always precede those from an honest process $q$ by sending postdated vectors on its messages to $q$.
- **Privacy** A malicious process could use the timestamp vectors sent as part of the protocol to spy on the activity of other processes.

Chapter 5 considered two approaches to provide secure clocks for the PARTIAL_ORDER_TIME model: the *Signed Vector Timestamp* protocol and the *Sealed Vector Timestamp* protocol. Using the security of these clocks to provide security for higher-level application protocols (such as those presented in Chapters 2 through 4) raises two critical issues:

- Do the security properties of the clocks protect the application protocols against clock-based attacks?

- Do the security properties of the clocks hold for the higher-level time models considered by some application protocols?

For example, the Signed Vector Timestamp protocol protects immediate ordered service only against some of the *postdating* risks—with Signed Vectors, a malicious process must confine its postdating to entries belonging to processes whose keys it knows. The Signed Vector Timestamp protocol provides even less protection if (due to failure and recovery) the partial order model is flow-virtual. On the other hand, the Sealed Vector Timestamp protocol eliminates all three risks.

Chapter 6 examines these issues of security and privacy for higher-level protocols and time models. Section 6.2 explores the protection that our secure vector protocols provide for the time models considered in this thesis. Section 6.3 and Section 6.4 consider the security implications for the application problems of distributed snapshots and optimistic rollback recovery, respectively.

## 6.2. Security, Timestamps, and Time Models

Section 6.2.1 discusses the general paradigm behind the clock schemes proposed in this thesis. Section 6.2.2 discusses some attacks permitted by this family. Section 6.2.3 discusses how the defenses proposed in Chapter 5 fare against these attacks, for various types of time models.

### 6.2.1. Timestamp Clocks

The clock protocols discussed in this thesis are based on timestamps: processes generate a *timestamp* associated with an event or state $A$, and this timestamp serves to sort $A$ relative to other events or states.

Such timestamp clocks are easily implemented for Type 4 parallel pairs—pairs that are consistent, independent, strongly monotonic and flow-supported. The ease of implementation follows from these properties:

- Strong monotonicity implies the relation between two nodes is established forever once they come into existence.
- Flow-support implies that a process has the potential to know all information necessary to create a timestamp for a node when the node comes into existence.

For example, consider generating the timestamp vector for a node $A$ in the PARTIAL_ORDER_TIME model. The timestamp vector $\mathbf{V}(A)$ is well-defined when $A$ occurs, due to strong monotonicity: when $A$ occurs, all the nodes that precede $A$ have occurred, and their precedence is established. The timestamp vector $\mathbf{V}(A)$ can be created when $A$ occurs, due to flow-support: information paths exist from every node in $\mathbf{V}(A)$ to $A$.

Timestamp clocks can also be implemented for some Type 2 parallel pairs—pairs that are only guaranteed to be consistent and independent. These pairs lack the convenient properties of Type 4, but we may compensate:

- Weak monotonicity implies that when a precedence relation is established between two nodes, the relation holds forever. Consequently, weak monotonicity coupled with a way to determine *when* all such relations have been established for a node still permits a timestamping scheme.
- Strictly speaking, only the agents that create timestamps require information flow. These agents do not need to be processes—for example, the Sealed Vector Timestamp protocol splits clocks from processes.

Before we can discuss example implementations for time models other than Type 4 parallel pairs, we need machinery to separate clock agents (and their experience) from process agents. The tools of distributed time provide an easy way to express this notion: we can build a Type 4 parallel pair

$$(\text{CLOCK\_PARTIAL\_ORDER}, \text{CLOCK\_TIMELINES})$$

to express the computational activity and information flow of the clock agents. We consider various pairs:

- For the PARTIAL_ORDER_TIME model with the processes themselves implementing clocks, the clock pair above is the same as $(\text{PARTIAL\_ORDER\_TIME}, \text{TIMELINES})$.
- For the SYSTEM_PARTIAL_ORDER and USER_PARTIAL_ORDER models, if processes themselves implement clocks, then the clock pair is the same as

  $$(\text{SYSTEM\_PARTIAL\_ORDER}, \text{SYSTEM\_TIMELINES})$$

  If processes use separate clock processors, then the clock pair is the partial order parallel pair obtained by treating clocks as separate processes.

Using $(\text{CLOCK\_PARTIAL\_ORDER}, \text{CLOCK\_TIMELINES})$ clarifies the discussion of when we can build timestamp clocks for a weakly monotonic model $\mathbf{M}$. Basically, we use the clock computation to simulate strong monotonicity and flow-support. We restate the earlier conditions in these terms:

- **Simulated Strong Monotonicity** Clocks in CLOCK_PARTIAL_ORDER generate timestamps for nodes $A$ and $B$ in $\mathbf{M}$ only when the relation between $A$ and $B$ is fixed.
- **Simulated Flow-Support** If a precedence path exists from node $A$ to node $B$ in $\overline{\mathbf{M}}(\mathit{CUR\_GRAPH})$, then a precedence path exists from $A$ to the clock node that generates a timestamp for $B$.

For example, consider generating timestamp vectors for the STRONG_PARTIAL_ORDER model. A send event $S$ depends on the corresponding receive event $R$, only no information path exists from $R$ to $S$. As a result, when a node $A$ occurs, the information necessary to create $\mathbf{V}(A)$ is not available, and in fact $\mathbf{V}(A)$ may not even be well-defined. However, a clock coprocessor could keep track of the set $X$ of receive events it depends on but does not know about, and generate for $A$ an interim timestamp consisting of a vector $V$ and this set $X$. This interim timestamp satisfies the invariant:

$$\mathbf{V}(A) \quad = \quad \left( \bigsqcup_{R \in X} \mathbf{V}(R) \right) \sqcup V$$

Independently, the clock coprocessors share interim timestamp information for receive events that have occurred, and transform interim timestamps to reflect this new information. If the set in an interim timestamp for node $A$ becomes empty at clock node $B_C$ at process $p$, then all nodes that will ever precede $A$ have occurred, and information paths exist to $B_C$ from each of these nodes. The clock at process $p$ may then generate the full timestamp vector $\mathbf{V}(A)$.

**Precedence Horizons** The timestamp vector protocols use timestamps that specify precedence horizons: the timestamp vector for node $A$ consists of the names of the process-maximal nodes that precede or equal $A$. Such precedence horizons function as clocks for parallel pairs where processes can sort events in the other process's local time structures. As Chapter 4 described, this approach also extends to restricted subgraphs of nonlinear pairs (e.g., when a well-defined valid computation emerges from USER_PARTIAL_ORDER).

### 6.2.2. Attacks

Clocks based on precedence horizons have three distinct tasks:

- **Generating Local Tokens** A clock at a process must generate a local token for each of its nodes. This token may an integer or a more complex identifier, and may include items such as signatures.
- **Assembling Timestamps** A clock at a process must assemble sets of these local tokens into a global timestamp.
- **Disassembling Timestamps** A clock at a process must disassemble a global timestamp into local tokens, some of which may be re-used when assembling subsequent timestamps.

These tasks generate the following security concerns:

- Is a given local token correct? Suppose the clock at process $q$ has a token for node $A$ at process $p$. Did node $A$ actually occur? Is this the correct token for $A$? Should the clock at $q$ even possess this data?

- Is the assembly correct? Clocks are supposed to follow some set of specified rules when assembling timestamps. Were these rules followed?
- Is the information released by disassembling a timestamp confined to appropriate agents?

These concerns create opportunities for malicious agents to attack clock protocols. Chapter 5 discussed three such attacks. *Compromised privacy* may occur when agents release data from disassembled timestamps. Violating the assembly rules (and creating fraudulent tokens) leads to *backdating* and *postdating* attacks; these violations can also lead to *concurrent-dating* attacks in which some vector entries are advanced and others reduced.

The PARTIAL_ORDER_TIME model alone provides a single partial order with straightline graphs at processes. Departing from this comfortable world permits two additional attacks:

- **Level-Mixing** When we deal with multiple levels of time without adequately distinguishing the levels, a malicious agent may assemble timestamps for one level using tokens for another.
- **Branch-Mixing** In nonlinear pairs such as USER_PARTIAL_ORDER, a malicious agent may assemble timestamps using at least one token from an incorrect process branch. Such "sidedating" places an event in a computation different from the one actually occurring.

### 6.2.3. Defenses

**Signed Vectors** The Signed Vector Timestamp protocol requires processes to implement their own clocks, and addresses the security concerns of Section 6.2.2 by using cryptography to verify the identity of the process creating the local tokens. Each process has its own private key; multiple levels of processes presumably have distinct private keys.

This approach raises two significant problems:

- The protocol restricts only *identity*, not *time*.
- The security of the protocol rests on an implicit assumption that the time model is *not* flow-virtual: that information flow implies precedence.

We now consider these problems in more detail. The Signed Vector Timestamp protocol leaves processes completely free to create arbitrary local tokens. This flaw permits the *postdating* attacks: a malicious process $p$ can advance its own local counter, sign it, and pass this along to a process $q$ as the "real" value. Processes are also free to create arbitrary global timestamps from the local tokens available. This flaw permits the *backdating* attacks on PARTIAL_ORDER_TIME: a malicious process $p$ can assemble an arbitrary timestamp from the signed entries it possesses.

The Signed Vector Timestamp protocol also implicitly assumes that, barring signature compromise, possession of a signed entry for a node implies precedence from that node. Suppose that node

$A$ occurs at an honest process $p$, that a process $q$ at clock node $B_C$ generates a timestamp for node $B$, and that this timestamp includes a signed entry from node $A$. For Signed Vectors, this is sufficient evidence to conclude that $A \longrightarrow B$ in the higher-level model $\overline{\mathbf{M}}$. However, all we are justified in concluding is that $A_C \longrightarrow B_C$ in the $\overline{\text{CLOCK\_PARTIAL\_ORDER}}$, where $A_C$ was the timestamp generation event for $A$. In flow-virtual time models (such as USER_PARTIAL_ORDER), precedence in $\overline{\text{CLOCK\_PARTIAL\_ORDER}}$ will not imply precedence in the higher-level $\overline{\mathbf{M}}$. In these cases, Signed Vectors permit branch-mixing attacks. (Figure 6.1 shows a simple example.) Branch-mixing may even masquerade as the postdating of the entries belonging to honest processes.

**Sealed Vectors** Using secure coprocessors to implement clocks allows reliable location of tokens in both space *and* time. Using secure coprocessors also ensures that no rules are broken in the assembly and disassembly of global timestamps, since we can trust the secure coprocessor at any process to track a local counter and (barring communication subversion) assemble the correct pieces into timestamps. Secure coprocessors could also be used to track relations in a model $\mathbf{M}$ more general than the underlying SYSTEM_PARTIAL_ORDER model, if the model is well-defined in terms of the SYSTEM_PARTIAL_ORDER. Thus the security properties of Sealed Vectors extend to models such as the USER_PARTIAL_ORDER and the STRONG_PARTIAL_ORDER.

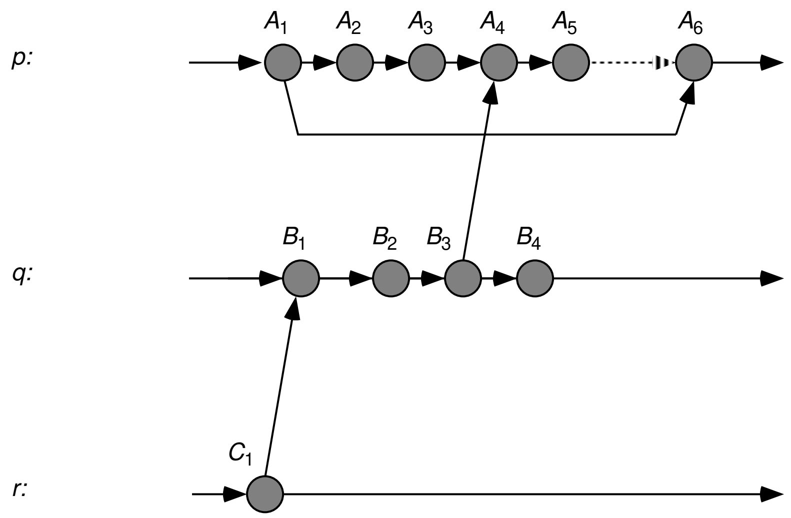


**Figure 6.1** The Signed Vector Timestamp protocol fails for flow-virtual time models, since processes may retain signed entries from previous lifetimes. Suppose process $p$ has rolled back for reasons other than local failure: either voluntarily, or in response to failure at another process. Process $p$ at node $A_6$ can forge USER_PARTIAL_ORDER dependence on nodes $B_1$ through $B_3$ at process $q$ and on node $C_1$ at process $r$, because an information path exists from $A_4$ to $A_6$. Even giving each process incarnation its own private key does not remove this problem.

By also functioning as reliable oracles at processes, secure coprocessors make new techniques possible. For example, the secure coprocessor at process $p$ will truthfully list a complete set of nodes at $p$ satisfying some particular property (provided that the nodes have been registered with the coprocessor, and that the property is something that the coprocessor has sufficient information to evaluate).

## 6.3. Distributed Snapshots

Chapter 3 discussed the problem of taking *distributed snapshots* in terms of the distributed time framework. This discussion took two paths: using clocks for partial order time to build *Round Robin* protocols assembling global states, and using such snapshot protocols with more general time models in order to capture global states with more specific properties. Their use of distributed time clocks makes these protocols susceptible to the security and privacy risks—and defenses—of Chapter 5.

This section considers these issues. Section 6.3.1 considers active attacks, and Section 6.3.2 considers passive ones. Section 6.3.3 discusses the security and privacy implications for the distributed time snapshot protocols using more abstract time models.

### 6.3.1. Active Attacks

Distributed snapshot protocols based on timestamp vectors inherit their security risks. Since taking a snapshot requires more than just sorting timestamps, these protocols are liable to some additional risks as well. That is, taking a distributed snapshot involves two somewhat orthogonal tasks:

- assembling a timeslice, and
- obtaining a description of the activity on this timeslice.

A malicious process may actively attack both tasks.

**Attacking Timeslice Assembly** The basic Round Robin snapshot protocol of Section 3.2.1 assembles a maximal set of nodes mutually concurrent in the transitive global time model. This basic protocol organizes processes into a directed cycle. Suppose process $P_k$ receives a set $S_{k-1}$ of mutually concurrent nodes from $P_1$ through $P_{k-1}$. Process $P_k$ is supposed to add one of its own nodes to form mutually concurrent set $S_k$; however, process $P_k$ may act instead with malice:

- **Backdating** Process $P_k$ could forge a backdated timestamp for some node $A$, and consequently include $A$ in $S_k$ even if $A$ follows some $B \in S_{k-1}$.

- **Postdating** Likewise, process $P_k$ could forge a postdated timestamp for some node $A$, and consequently include $A$ in $S_k$ even if $A$ precedes some $B \in S_{k-1}$.

This protocol gives each process freedom in selecting concurrent nodes. This freedom makes concurrency detection less robust against attack. The Signed Vector Timestamp protocol does not help: a malicious process $P_k$ can select arbitrary signed entries from the timestamps on the nodes in set $S_{k-1}$.

The Reduced Round Robin snapshot protocols of Section 3.2.2 achieve better performance than this basic protocol; this improvement exploits shortcuts: using concurrency information that timestamp and rollback vectors already contain. These shortcuts sometimes make concurrency detection more resilient. For example, suppose a malicious process $p$ fraudulently wishes to insert a node $A$ into a snapshot obtained from the adjusted timestamp vector $\mathbf{V}^*(B)$ of node $B$ at process $q$. Since process $q$ already "knows" the identity of $A$ (from the timestamp vector for $B$), process $p$ must manipulate vectors not only before $q$ asks for the snapshot, but also before $B$ even occurs. Process $p$ must forge the right sequence of outgoing messages—and must hope that other processes do not send messages that dispel the illusion that the node preceding $A$ at $p$ is the $p$-maximal node preceding $B$. On the other hand, taking a snapshot using an adjusted *rollback* vector $\mathbf{R}^*(B)$ does not provide as much resilience, since the potential delay between $B$ and $\mathbf{R}^*(B)$ gives malicious processes more flexibility.

**Attacking Descriptions** Taking a snapshot usually entails more than just collecting a set of mutually concurrent timestamps; we also want a description of the activity associated with each of these timestamps. This requirement creates another avenue of attack: a malicious process may attack a snapshot protocol by using legitimate timestamps but lying about the nodes and activity that belong to the timestamps. For example, in the Reduced Round Robin snapshot protocol, an honest process $q$ might ask a malicious process $p$ for the node following the one names by the $p$ entry in $\mathbf{V}(B)$, for a node $B$ at process $q$. Protocols such as Signed Vectors keep the timestamp separate from the node name—so process $p$ can reply to $q$ with the proper timestamp for the requested node, but may forge the name and description of the node itself.

**Defenses** Protecting against these attacks using the Sealed Vector Timestamp protocol is straightforward. Sealed Vectors protect against forging timestamps and subverting concurrency detection; the presence of a trusted agent (the secure coprocessor) to link timestamps to node names protects against description attacks.

Effectively protecting against these attacks without using secure coprocessors remains a research area. Expanding Signed Vectors to include more details of message paths might make them harder to forge. However, we still have the problem that (in terms of Section 6.2.1) the ability to assemble legitimate timestamps easily transforms into the ability fraudulent timestamps. Rather than using the correct set of local tokens, a malicious process may use a carefully chosen incorrect set.

The techniques of Haber and Stornetta [HaSt91, BHS93] provide some grounds for future work. Cryptographic linking techniques might prevent node-name attacks (honest processes can prove their allegation that a given node follows node $A$), but using these techniques requires forcing processes to exchange correct logging information. This exchange may be difficult to ensure without the trusted local agent of a secure coprocessor. Pseudorandom logging techniques may be more effective in these situations—but at the expense of increased communication and delayed verification, and also with the increased risk of espionage and sabotage that come with remote logging.

### 6.3.2. Passive Attacks

Snapshot protocols based on distributed time also permit passive attacks—both the standard timestamp vector attacks, and new ones raised by the snapshot problem.

**Privacy** Distributed snapshot protocols based on timestamp vectors inherit their privacy risks: vectors leak information. Problems also arise from observation effects:[1] the interaction between the act of observing and the computation being observed. Do the messages exchanged as part of taking a snapshot of a given computation belong to the computation? If not, then the data being exchanged creates serious potential for abuse. Participating in such a snapshot protocol provides processes with valid local tokens for nodes on which they have no precedence; a malicious process might use these tokens to forge timestamps. For example, using the Signed Vector Timestamp protocol here would distribute signed vector entries to processes that have no dependence on the nodes named by those entries.

**Spying on the Initiator** The preceding attacks come from spying on the data exchanged as part of a snapshot protocol. A malicious process may also gain unauthorized information from the fact that a snapshot protocol is being executed. For example, suppose auditor *Alice* is asking for a snapshot to verify that the electronic currency in circulation sums correctly. If counterfeiter *Bad Bob* knows this fact, then he may manipulate this probe to hide his crime (and subvert the purpose of *Alice* taking this snapshot).

**Spying on Other Processes** A distributed snapshot protocol may also be misused by its initiator to gain unauthorized information about other processes in the system. Of course, an authorization policy must exist for an action to be classified as misuse. If anyone is permitted to take any kind of snapshot at any time, then subversion is not necessary. However, more substantive authorization rules create the potential for both direct and indirect attacks. A malicious process $p$ might forge its own authorization proving the legitimacy of its snapshot request; alternatively,

[1] Section 3.4.2 discussed this issue.

a malicious process $p$ might spy on process $q$ by simulating participation in a legitimate snapshot protocol initiated by a different process.

**Defenses** One way to add resiliency to a snapshot protocol is to require each participant to identify the initiator. However, strengthening a protocol by including extra information suggests a fundamental tradeoff between privacy and security: information included is information leaked. As with the active attacks, using secure coprocessors appears to be the best defense. We can seal the entire snapshot protocol, and also use secure coprocessors to ensure the initial requests for snapshots agree with whatever policy we select for snapshot authorization.

### 6.3.3. Alternative Models

Chapter 3 introduces another approach to obtaining global states satisfying some particular property: taking a standard snapshot from a nonstandard time model. Chapter 4 shows how at least three distinct virtual partial orders arise from rollback with modified replay; a process may also wish to take a snapshot from one these alternative models.

This approach to snapshots follows directly from the orthogonality between clocks and higher-level time protocols. However, the *performance* orthogonality between clocks and protocols does not extend to a *security* orthogonality between clocks and models. As Section 6.2 discussed, how a temporal relation in an abstract time model arises from the real-time partial order (e.g., is it flow-virtual?) influences how its clocks may be attacked.

**Blocked Partial Order Time** Theorem 3.5 from Section 3.3.2 repeats a result from [Sm93]: each timeslice from a Type 2 (consistent and independent) parallel pair has a unique subset of nodes that determine the timeslice. Since these subsets are partial timeslices from the composition of the BLOCKED model with the partial order, taking snapshots in this higher-level model yields an exponential number of snapshots in the original partial order. This technique creates the potential for security and privacy problems, because of the BLOCKED model itself, and because we have two levels of time.

One problem arises because of the lack of view-completeness. Applying BLOCKED to a Type 2 parallel pair $(\mathbf{M}, \mathbf{M}')$ does not preserve all properties of $(\mathbf{M}, \mathbf{M}')$; in particular, we lose view-completeness (as Section 3.3.2 observes). This adds a wrinkle to the Round Robin protocol: a process may not have any node to add to the partial timeslice. This wrinkle leads again to a security-privacy tradeoff: if we do not require such a process to provide proof of its necessary abstention, then we allow malicious processes to opt out of reporting sensitive data. The role of secure coprocessors as trusted oracles keeps this from being a problem for the Sealed Vector Timestamp protocol.

Flow-support is another property that the BLOCKED model does not preserve. Consider a message edge $S \longrightarrow R$ in a graph from the PARTIAL_ORDER_TIME model. Composing BLOCKED with this model draws an edge to $R$ from the local successor of $S$, but an information flow path does not exist. This is not a serious problem: the sending process can inform the receiving process of the identification of the next local node. For the higher-level model, the foundation of the Signed Vector Timestamp protocol still holds: possession of a signature for an honest node proves dependence on that node. We may thus use either Signed Vectors or Sealed Vectors to track $\overline{\text{BLOCKED} \circ \text{PARTIAL\_ORDER\_TIME}}$ relations.

The fact that two distinct levels of time are being tracked also permits level-mixing attacks. Consider again the example of PARTIAL_ORDER_TIME. The $\overline{\text{PARTIAL\_ORDER\_TIME}}$ and $\overline{\text{BLOCKED} \circ \text{PARTIAL\_ORDER\_TIME}}$ models describe sufficiently similar structures that privacy is not a problem. However, security risks might still exist: for example, with Signed Vectors, timestamps for one level could be used to construct timestamps for the other. As Section 6.2.3 observed, the use of Signed Vectors with multiple levels requires either distinct signature functions or distinct name spaces. (Otherwise, our trick for having possession imply precedence in BLOCKED $\circ$ PARTIAL_ORDER_TIME causes this property to fail for PARTIAL_ORDER_TIME.)

**Strong Partial Order Time** We developed the STRONG model to allow processes to take snapshots in which no messages are in transit. The STRONG model composes with a partial order by making message edges bidirectional; the resulting temporal relation has the property that its timeslices are exactly the timeslices from the original partial order in which no messages were in transit.

The STRONG model also alters the properties of the model to which it is applied. For example, $\overline{\text{STRONG} \circ \text{PARTIAL\_ORDER\_TIME}}$ differs from standard partial order time in some substantial ways: edges may flow backwards in time, and precedence no longer implies information flow. The most substantial difference is that the relation possesses cycles. Making message edges bidirectional ties together send events and receive events; sets of messages may interact in unexpected ways to form larger cycles.

The cycles in the $\overline{\text{STRONG} \circ \text{PARTIAL\_ORDER\_TIME}}$ model create opportunities for malicious processes to attack clock and snapshot protocols. As Section 6.2.1 discussed, clocks must keep track of the incoming edges with unknown originating nodes; clocks that know the identity of these nodes must see that this information eventually reaches the clocks that need it. Without secure coprocessors to keep them honest, malicious processes can lie on both ends of this task, and spy on the information itself.

Malicious processes can also subvert the model without attacking the clocks by making sure that at least one message is always in transit. Figure 6.2 sketches this scenario.

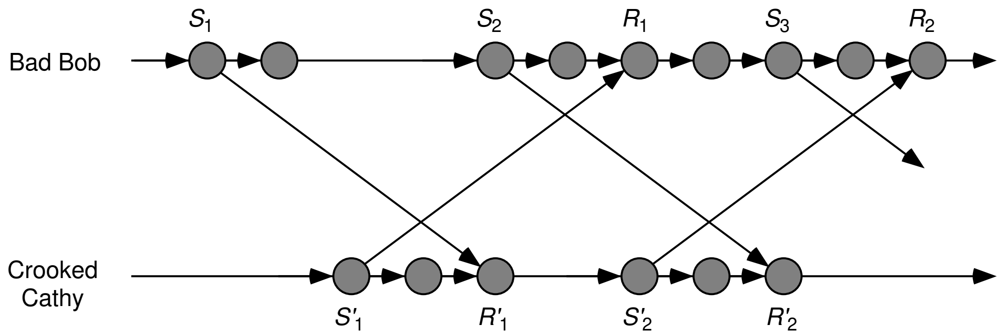


**Figure 6.2** Malicious processes may subvert the STRONG_PARTIAL_ORDER model by ensuring that at least one message is always in transit. This PARTIAL_ORDER_TIME graph illustrates the initial phases of such a conspiracy between *Bad Bob* and *Crooked Cathy*. For each $i$, $S_i \longrightarrow R_i$ and $S'_i \longrightarrow R'_i$ in the timelines. However, the STRONG model makes message edges bidirectional, so applying that would make $R_i \longrightarrow S'_i$ and $R'_i \longrightarrow S_i$. Hence, in $\overline{\text{STRONG\_PARTIAL\_ORDER}}$, each *Bad Bob* node from $S_i$ to $R_i$ (inclusive) is cyclic, and cannot be part of a timeslice. *Bad Bob* and *Crooked Cathy* collaborate to ensure that any *Bad Bob* node from $S_1$ on can never be part of a STRONG_PARTIAL_ORDER timeslice.

## 6.4. Optimistic Rollback Recovery

The optimistic rollback recovery protocol of Chapter 4 uses distributed time clocks, and thus is liable to security and privacy attacks on the clock mechanisms. Section 6.4.1 considers standard attacks on clocks, and Section 6.4.2 considers some attacks more specific to optimistic rollback recovery. Many of these issues are also relevant to previous rollback protocols; however, by its explicit foundation in two levels of partial order time, our protocol is a particularly appropriate scenario to discuss these issues.

### 6.4.1. Standard Attacks

Chapter 5 discussed three risks of partial order clocks: backdating, postdating, and privacy leaks. Section 6.2.2 discussed the additional problems of *level-mixing* (that arises when an application tracks multiple levels of time) and *branch-mixing* (that arises when an application deals with a nonlinear pair). These risks all apply to our optimistic rollback recovery protocol, which uses two levels of time: a nonlinear pair to track dependence on failed nodes, and a parallel pair to track knowledge of rollback.

The SYSTEM_PARTIAL_ORDER model tracking knowledge of rollbacks is a standard partial order model, producing the partial order that an external observer (unaware that recovery is taking place) would perceive. The standard backdating, postdating, and privacy risks apply.

In this context, backdating hides knowledge of rollbacks; the malicious process falsely deludes an honest process into thinking a node is not an orphan. This hoax may have direct consequences, such as the honest process failing to roll itself back or to discard an incoming orphan message, or may have more subtle consequences, such undeceived honest processes rejecting messages from the deceived honest process. Figure 6.3 shows an example of the first scenario; Figure 6.4 shows an example of the second.

Postdating a node in the SYSTEM_PARTIAL_ORDER order involves forging the future. Placing a user node artificially far in the SYSTEM_PARTIAL_ORDER future allows a malicious process to fool an honest one into accepting orphan messages. Figure 6.5 shows an example. The Signed Vector Timestamp protocol does not solve these problems, but the Sealed Vector Timestamp protocol does.

In the USER_PARTIAL_ORDER model tracking dependence on failed nodes, backdating and postdating have the more standard behavior of hiding or forging dependence on failed nodes. This model behaves like the standard partial order until rollback actually occurs.

**Branch-Mixing** Once user timelines turn into timetrees, the USER_PARTIAL_ORDER graph may generate a valid FAILURE_FREE_PARTIAL_ORDER graph. However, the fact that failure-free graph is flow-virtual complicates the task of reliably tracking it. As Section 6.2.3 observed and Figure 6.1

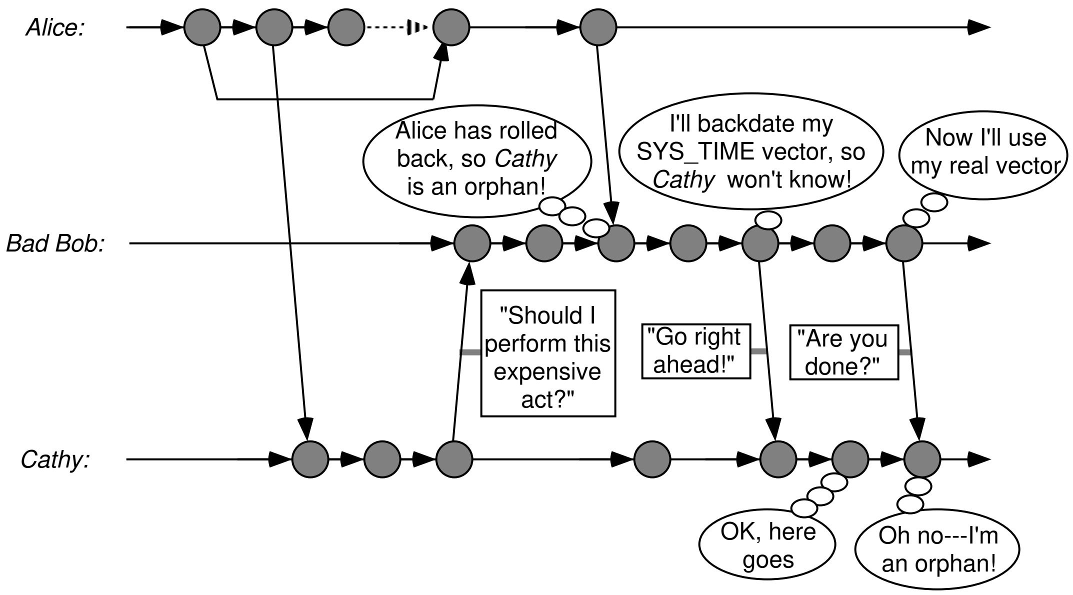


**Figure 6.3** Backdating SYSTEM_PARTIAL_ORDER relations can cause honest processes to waste computation. In this example, *Alice's* rollback makes *Cathy* an orphan. By backdating the SYSTEM_PARTIAL_ORDER vector on his first message to *Cathy*, *Bad Bob* prevents *Cathy* from learning that she is an orphan until after she has performed expensive computation that now must be discarded.

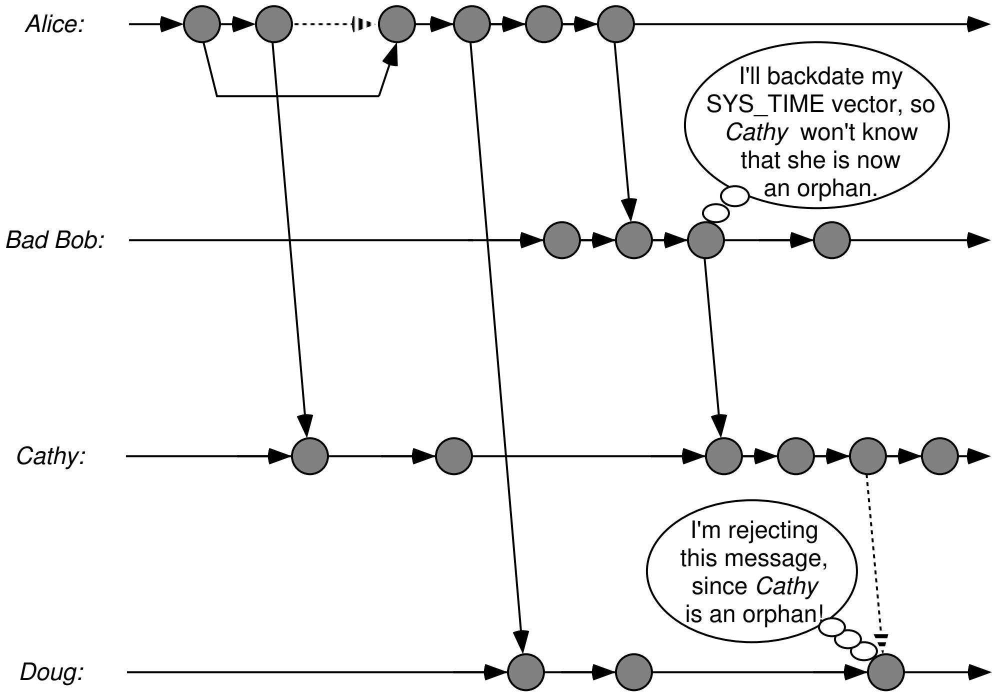


**Figure 6.4** If knowledge of rollback is propagated solely on system messages carrying user messages, then backdating SYSTEM_PARTIAL_ORDER can cause an honest process to remain an orphan indefinitely. In this example, *Alice's* rollback makes *Cathy* an orphan. By backdating his SYSTEM_PARTIAL_ORDER vector, *Bad Bob* prevents *Cathy* from learning this fact. All of her subsequent user messages will be rejected—*Cathy* loses all credibility with *Doug*.

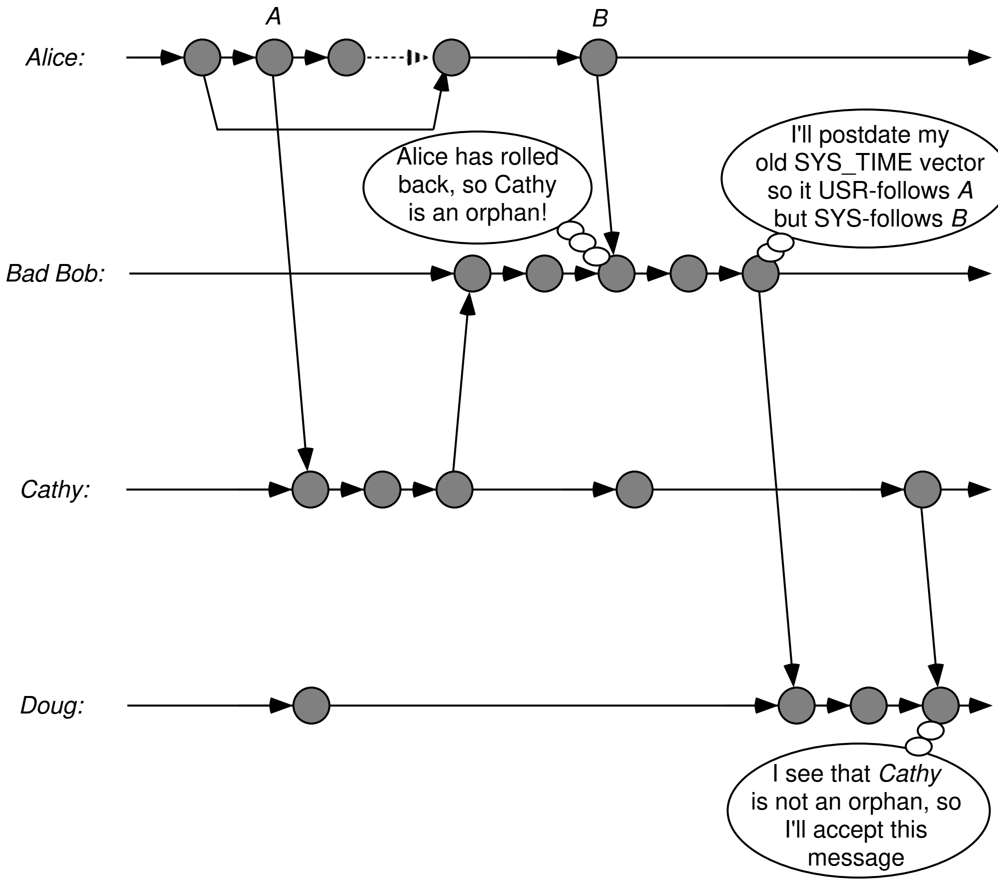


**Figure 6.5** Postdating SYSTEM_PARTIAL_ORDER relations fools honest processes into accepting orphan messages. In this example, *Alice's* rollback makes *Cathy* an orphan. By advancing his *Alice* entry in the system model but not in the user model, *Bad Bob* not only hides the rollback from *Doug*, he ensures that *Doug* will not listen to anyone else's announcement of the rollback.

illustrated, the Signed Vector Timestamp protocol breaks down when the a graph is generated virtually—possession of a signed entry for a node no longer implies dependence on that node.

The Sealed Vector Timestamp protocol still provides protection in this scenario.

**Level-Mixing** One way to subvert a protocol that requires accurate tracking of computation on two levels is to disrupt the correspondence between the levels. For example, the rollback protocol from Chapter 4 requires that processes be able to map two system nodes at another process to their corresponding user nodes, and be able to sort them in terms of the USER_PARTIAL_ORDER model. How can this mapping be made reliable? If it is each process's responsibility to report a node as a pair of identifiers, then malicious processes can avoid the problem of forging timestamps merely by mismatching valid timestamps.

Again, the Sealed Vector Timestamp protocol still provides protection in this scenario.

**Privacy Risks** Surviving processes may need to roll back in response to a failure. If a surviving process is malicious, it may retain and exploit old state. For example, one process in a poker game may mistakenly reveal a card, and call for rollback. How do we ensure the other processes actually "forget" the revealed card?

Banking systems provide another example. Suppose *Alice* deposits a large check for *Bad Bob* with banker *Cathy*. *Alice* then discovers that all her activity that day was incorrect, and rolls herself back. The current state at *Cathy* indicates that a large sum of money is in *Bad Bob*'s account—but *Alice*'s failure makes this state an orphan. If *Bad Bob* learns that *Cathy* is an orphan before *Cathy* does, then *Bad Bob* can exploit the incorrect balance by withdrawing the extra money.

To solve this problem, we need to introduce a complete discontinuity in the state of surviving processes that roll back. Perhaps we could force a site migration, and keep the location of the new site secret from the old site. We may need to extend this discontinuity to any process learning of rollback: the banking example did not specify whether *Bad Bob's* own state was an orphan. This problem raises similar issues as *commitment*, since transfer of knowledge is an action that is difficult to undo.

### 6.4.2. Other Avenues of Attack

Our framework of secure distributed time provides protection against clock-based security and privacy risks. However, optimistic rollback recovery protocols face other security risks. In this section consider, we discuss some of these areas for future research.

**Checkpointing** Rollback protocols assume some mechanism for processes to restore state. Usually this mechanism uses stable storage to preserve sufficient information for state restoration.

This information may consist of checkpointed images of local state, logs of incoming messages (for replay), logs of outgoing messages (for replay), or various combinations of these techniques.

The existence and use of this logged information creates security and privacy risks:

- **Forging Identity** A malicious process can forge someone else's checkpoint.
- **Forging Data** A malicious process can lie about the data it stores as its own checkpoint.
- **Forging Timestamps** In protocols that preserve more than just the most recent checkpoint at each process, a malicious process can attack the methods used to identify which checkpoint belongs to what point in (distributed) time.
- **Forging Storage Location** If stable storage servers are distributed throughout the system, a method must exist that, upon recovery of process $q$, specifies where the checkpoint for $q$ is saved. A malicious process can disrupt recovery by leaving $q$'s checkpoints untouched and attacking this mapping instead.
- **Espionage** A malicious process $p$ might gain unauthorized knowledge about the affairs of process $q$ by examining checkpoints belonging to $q$.
- **Interactions** The checkpointing policy at a physical process site cannot be completely orthogonal to the levels of processes at that site. For example, in a system using Signed Vectors, checkpointing a physical process would leak keys belonging to the system-level processes (since the checkpoint would include these keys).
- **Authority** The authority to read a checkpoint belonging to a process $q$ must be more than just the identity of $q$ or its physical site, since both these may vanish in a failure.

**Restart** As the last item above suggests, the mechanics of restart—especially in the face of failure of physical machines—creates risks:

- **Proving Legitimacy of Request** How does a system establish the legitimacy of a restart request? For example, consider the standard mechanism of a process $p$ calling for a restart of process $q$ if process $q$ has been silent for a while. (After all, indefinite silence is indistinguishable from failure.) A process $r$ that has heard from $q$ recently can veto this request. Once more, we face a tradeoff between security and privacy: preventing malicious vetoes requires that process $r$ present evidence (e.g., a timestamp) showing it has heard from $q$—which allows process $p$ to probe the behavior of processes $r$ and $q$ by "innocently" calling for restart.
- **Malicious Restart** A malicious process might be able to abuse the restart mechanism by convincing a sufficient quorum of processes that an honest, non-faulty process $q$ is dead.

- **Malicious Termination** Even with no malice, two versions of the same process may be alive simultaneously because silence and failure are indistinguishable. Rollback implementations thus must include a way to terminate honest processes. A malicious process might be able to abuse this machinery and terminate inconvenient honest processes.

- **Directing Migration to a Corrupted Site** When an honest process $q$ is restarted (either naturally, or through malice), a corrupt process $p$ might be able to direct the restarted version to a physical site that $p$ has compromised—thus gaining access to data and authority of process $q$.

- **Mutual Restart** Even if a majority of processes must agree to restart a silent process, a coterie of malicious processes could partition the honest processes and convince each half to restart the other.

- **Migration of Authority** Security and privacy techniques (such as the Signed Vector Timestamp protocol and the Sealed Vector Timestamp protocol) may assume that each honest process possesses secret keys. Realistic rollback protocols allow for a process to migrate to a new physical site when the original physical site fails. How does the new version of the process obtain the proper keys? If backup copies of the keys exist—even in a shared fashion [Sh79]—what protects them? If new keys are created, what prevents a malicious process from inventing and inserting new keys?

- **Revoking Authority** If a site is compromised, how do the surviving honest processes revoke its authority? Can the revocation mechanism be used to attack honest processes?

- **Migration of Identity** When a process migrates to a new site, how does it convince other processes of its new identity? Can this mechanism be abused to steal the identity of an honest process?

**Rollback** Even if we take care of these attacks on a rollback recovery protocol, the protocol can still be subverted simply by misusing it. For example, a malicious process can prevent the entire system from ever getting any work done simply by repeatedly sending messages to honest processes (establishing dependence) and then rolling itself back.

# Chapter 7

# Conclusion

## 7.1. Summary

Distributed time provides a general framework for building distributed protocols and for transparently adding security and privacy protection to these protocols. This thesis demonstrated these claims in three steps:

- We developed formal machinery to express the general temporal relations that arise in distributed application problems. We then built a suite of clock primitives for these relations.
- We analyzed application problems in terms of these general temporal relations. We then built protocol solutions using the clock primitives. The orthogonality between clocks and protocols allows transparently modifying the protocols by changing clocks and time models.
- We identified the security and privacy risks inherent to tracking general temporal relations, and built clock primitives that protect against these risks. We then provided security to our higher-level protocols by transparently substituting these secure clocks.

By providing insight into the underlying temporal relations and orthogonality between clocks and protocols, the distributed time framework permits us to build protocols that are more general, more flexible, and more secure than previous solutions. Furthermore, the security and privacy problems we identify—and the solutions we provide—also apply to less general frameworks.

Computational environments are becoming increasingly distributed, and applications are permeating social and financial arenas that are particularly sensitive to security and privacy attacks. The problems that this framework addresses are likewise becoming increasingly important.

### 7.1.1. Distributed Time

Distributed time improves on previous work in partial order time by providing a fully general temporal framework supporting protocol design and construction.

We defined a *computation graph* format to describe computation, and showed how to translate system traces into *ground-level computation graphs*. We then defined *time models* as representational transformations of computation graphs, and constructed a suite of *clock primitives* probing relations in these models.

Computation graphs allow us to consider temporal relations more general than partial orders—for example, non-transitive relations and cyclic relations. Time models provide a formal means to abstract away irrelevant temporal, physical, and computational detail. The ability to compose time models permits us to build hierarchies of temporal abstraction. The separation between time models and computation graphs allows us to consider computations that arise virtually, via composition of models.

### 7.1.2. Distributed Protocols

Distributed time supports protocol construction by providing an understanding of the general temporal relations underlying application problems, and by allowing processes to examine these relations via clock primitives. This thesis illustrates this support by applying the framework of distributed time to three application problems: detecting *potential causality*, and the more advanced examples of *distributed snapshots* and *optimistic rollback recovery*.

Distributed time permits accurate detection of potential causality in asynchronous distributed systems. Determining whether one event potentially influenced another reduces to querying a clock primitive. The orthogonality in our framework permits transparent extension of protocols using these clock queries. For example, changing the time model used in the clock primitives permits departing from real-time partial orders, allowing the detection of potential causality in a distributed computation that (perhaps via rollback and modified replay) never physically occurred.

By expressing temporal and computational abstraction, distributed time provides a framework for taking distributed snapshots. A *timeslice* in a well-constructed time model represents an instantaneous global state of the system in some underlying computation; clock primitives directly support assembly of timeslices. This framework permits increased flexibility. For example, we can take snapshots of the past, and (by using higher-level time models) we can take snapshots with specific properties.

By expressing multiple levels of temporal and computational abstraction, distributed time provides a framework for optimistic rollback recovery. This problem involves two distinct distributed computations: the *user* application level and the *system* recovery level. The ability to model both levels permits us to build an optimistic rollback recovery protocol that allows processes to fully exploit all potential information. Our new optimistic rollback recovery protocol is the first to provide both fully asynchronous recovery and optimality in the number of individual rollbacks at processes. In particular, we reduce the previous worst case for asynchronous optimistic rollback recovery from exponential to at most one rollback per process after any failure.

### 7.1.3. Security and Privacy

Tracking temporal relations more general than real time creates security and privacy risks. This thesis identified these risks and constructed clock primitives that protect against them. Because our framework provides orthogonality between clocks and protocols, using secure clocks can transparently provide security for higher-level protocols.

Unlike the passage of real time, the general temporal relations of distributed time cannot be verified independently. Processes must share private information, and must trust the information that is shared with them. This necessity of trust creates the potential for security and privacy risks for clocks for these relations: malicious processes may sabotage the clocks at honest processes by providing false information, and may spy on honest processes by misusing the information that honest processes provide. These risks for clocks translate to risks for protocols based on these clocks—such as the application protocols presented in this thesis, or other protocols based on querying temporal relations such as partial order time.

The proposal document for this thesis opened these questions and presented the *Signed Vector Timestamp* protocol, the first to provide security for partial order time clocks. This thesis used cryptographic and secure coprocessor techniques to develop the *Sealed Vector Timestamp* protocol that provides full security and privacy for time models more general than the standard real-time partial order. The generality and security of these techniques provides security and privacy protection for higher-level protocols built on these clocks. For example, we can provide immediate ordered service, take distributed snapshots, and recover from failure—while also protecting against espionage and Byzantine attacks.

### 7.1.4. A Single Arena for Time and Security

Previous work has used partial order time to analyze distributed computation and to construct distributed protocols. However, many distributed applications center on temporal relations more general than a single level of a partial order. Furthermore, separate applications may center on temporal relations that are separate but related.

The time hierarchies and secure clocks developed in this thesis provide a single framework to consider these separate issues. This framework allows us to integrate applications and solutions developed independently. For example, rollback with modified replay changes the underlying computation. By formally specifying how the rollback protocol changes the virtual partial order computation, and by writing snapshot protocols in terms of queries to clocks for a specific partial order time model, the distributed time framework lets us take snapshots without worrying about rollback. Furthermore, rollback creates several layers of partial order time; we can use distributed-time based snapshot protocols to take snapshots of each layer. This framework also allows us to consider the security and privacy issues for these various levels of time, independently of the particular applications and protocols.

## 7.2. Future Work

Future research in secure distributed time includes developing and testing new techniques for secure clocks, and using this framework to solve the time and security problems in new application areas. Section 7.2.1 considers some areas of work in clock techniques, and Section 7.2.2 discusses some particular application problems. However, the shape and scope of computation is changing, and the questions of security and privacy are becoming more urgent and harder to specify. Section 7.2.3 offers some speculation on the fundamental role that the secure distributed time framework may play in this emerging world.

### 7.2.1. Future Work: Techniques

Discussions of new clock and security techniques and areas for future research have occurred throughout this thesis. We summarize some of them here.

One of the principal drawbacks of any of the timestamp vector techniques is vector size: a vector has one entry for each process in the system, and these entries may even have nonconstant size. As Section 5.5 discussed, this property makes *scalability* a significant concern: how can these techniques extend to large networks? Avenues to solve this problem include adapting *cryptographic linking* and *distributed trust* techniques [HaSt91, BHS93], exploiting secure logging sites, and developing good heuristics for when information can be omitted. The discussion of timestamp clocks in Section 6.2.1 raises an additional question: do effective clock techniques exist that depart from the timestamp approach in any substantial way?

Some areas for security research include limiting potential damage when secure coprocessors are compromised (for example, fleshing out the Give-and-Forget approach of Section 5.5), detecting covert communication between malicious processes, and exploring what privacy can be attained without coprocessors. Another area for work is formulating effective privacy policies for coprocessor-based clocks such as Sealed Vectors. On a basic level, we need to develop formal (and enforceable) rules for precedence querying. How does process $p$ limit the use of timestamps it generates? How will this policy limit what a malicious process may learn via selective probing? On a more advanced level, we need to develop policies for snapshots and rollback that grant the initiator some degree of anonymity while also establishing the initiator's authority.

These issues all hint at a fundamental tradeoff between *security* and *privacy*. Including sufficient data in protocols to prevent malicious tampering creates privacy as well as efficiency concerns. Is this tradeoff unavoidable?

### 7.2.2. Future Work: Applications

The framework of secure distributed time is a powerful tool for solving application problem that depend on temporal relations more general than real time. As Section 7.1 discussed, this thesis applied this tool to the problems of potential causality, distributed snapshots, and optimistic rollback recovery. However, the framework of secure distributed time may be appropriate for many other application problems. We now consider some of these problems, and discuss the possible relevance of our framework and possible topics for future research.

**Rollback** Our research into optimistic rollback recovery suggests many directions for future work. One direction is exploring the area of commitability, especially in for situations where rollback recovery may be initiated for reasons other than process failure. Another direction is examining the list in Section 6.4.2 of possible security attacks on optimistic rollback recovery protocols.

A third direction consists of exploring more general versions of the problem: real world applications provide motivation for rolling back rollback. For example, users of word processors and graphics packages frequently attempt to UNDO previous UNDO commands. As Section 4.4 discussed, rolling back rollback requires considering *general-past* versions of the problem. It would be interesting to develop effective distributed techniques for these scenarios.

**Distributed Nested Transactions** Natural experience provides many examples of *atomic* actions: consider *Alice* physically giving a ten dollar bill to *Bob*. This exchange is either successfully completed, or it never happened. No intermediate views of this action are possible.

*Transactions* (e.g., [GrRe93]) are a standard tool for providing this abstraction in distributed systems. Without this framework, situations such as process failure, unreliable communication, and interactions between concurrent transactions may cause pathological behavior. A particular subcomputation may be distributed in time and space, and consequently may be susceptible to many failures. However, a transactional system guarantees *atomicity*, *consistency*, *independence*, and *durability*; a programmer may regard these subcomputations as durable, atomic actions that appear to happen in some linear sequence.

Essentially, transactions perform temporal and computational abstraction. During a transaction, certain processes may perceive individual steps occurring in a certain order; everyone else must perceive these actions as an atomic unit. *Nested* transactions allow additional levels of abstraction by permitting transactions to call lower-level transactions. Supporting nested transactions requires managing the interactions between child and parent transactions. One aspect of this management is *orphan elimination* (e.g., [HLMW87]); when a transaction is aborted, all subtransactions executing on its behalf must also be aborted.

Distributed time provides tools to support such temporal and computational abstraction. Hierarchies of time models permit the multi-level view necessary to implement individual steps as part of a single unit, and to support nested transactions. Cyclic time models support atomicity in a distributed environment. Partial order models support tracking dependency, for orphan elimination. Consequently, distributed time might provide a nice framework to implement distributed nested transactions. As an extra benefit, we can transparently provide security and privacy protection to these implementations. Fitting existing implementation into our framework and showing they already face these security risks would be an interesting research topic.

**Electronic Currency** Adapting the familiar paradigm of physical cash to a distributed electronic environment raises a number of challenges. Many properties of physical cash fail in the more general case of electronic currency.

As the initial example for transactions showed, natural experience with cash implicitly uses transactional behavior. For example, a dollar bill is a unique physical token; a faulty physical transaction will never cause this token to be duplicated. However, electronic transmission of a data packet (in general) leaves the sender with a copy of the packet. Further, electronic interactions may be subject to network and process failures. For example, if the communication line breaks while transferring cash, what happens to the cash? Consequently, robust electronic currency must provide fully transactional behavior; as we have discussed, the abstraction tools of distributed time have relevance to this area.

The temporal tools of distributed time also have relevance to electronic currency. For example, bookkeeping and auditing practices in the real world are implicitly based on the notion of perceivable real-time simultaneity. However, perceivable simultaneity is one of the first casualties of asynchronous distribution. The framework of distributed time provides support for obtaining and reasoning about possible simultaneous states, and consequently provides support for performing auditing and bookkeeping along timeslices.

Additionally, the framework of *secure* distributed time provides protection from acts of sabotage and espionage that money seems to motivate in humans. If communication failures may cause cash to be created (or destroyed), then malicious agents will simulate communication failures. If incorrect values in timestamp vectors prevent discovery of illegal cash activities (as Section 6.3 discussed), then malicious agents will use incorrect values in timestamp vectors.

The ability to model multiple levels of abstraction while providing some assurances of security and reliability provides the distributed time framework with another class of potential applications: balancing privacy of transactions with government law. For example, the Internal Revenue Service may have the right to examine and verify certain aspects of the flow of electronic currency. Integrating time models expressing currency flow with time models expressing knowledge rights might provide the necessary tools.

**Electronic Exchanges** Performing commodities trading on public networks raises a number of challenges (e.g., [SEC94]). As the examples of Chapter 5 indicate, these applications are susceptible to many explicitly clock-based security problems, since adversaries may reap rich rewards by subverting clock protocols. How do we accurately detect the order of actions? How do we prevent unauthorized leaking of information? The framework of secure distributed time provides a foundation for exploring these issues.

Considering real-world scenarios raises even more questions. For example, how do we enforce real-time fairness? Even though *Alice* in Albuquerque and *Bad Bob* in New York do not receive news of a stock offer at the same real time, they should have the same duration of time to consider the offer. If *Alice* responds one second after she receives the news and *Bad Bob* responds two seconds, *Alice's* response should take precedence, even though it may have been sent after *Bob's* was sent. (The real-time order in which the responses arrive is yet another issue.) These questions suggest another interesting research topic: incorporating real time into the framework of distributed time.

**Capabilities Management** A *capability* is an explicit authorization granting its bearer certain rights. As computation becomes more distributed and asynchronous, the problem of *capabilities management* becomes more complex. If *Alice* performs a task on behalf of *Bob*, how does *Alice* inherit the necessary capabilities? How does *Bob* later revoke the capabilities he has transferred to *Alice*?

Capabilities management in distributed systems raises several issues related to partial order time. We list some examples:

- Managing capability inheritance requires tracking a relation very similar to precedence paths.
- Revoking capabilities requires modifying these paths.
- Enforcing access rules requires using these paths to restrict the user-level computation.

The framework of distributed time provides tools for constructing and and tracking hierarchies of partial order time relations. In capabilities management, these tools should apply both to the time-like relations of capabilities, and to the interaction of these relations with time models describing computation. In particular, our framework may have relevance to the earlier examples:

- Tracking which agents inherit which capabilities requires distributed construction of a directed acyclic graph (e.g., [He91]). We could examine this problem in the distributed time framework by building time models that express authority instead of temporal precedence.
- Inheritance complicates revocation. Revoking a capability given to *Alice* should also revoke that capability from anyone who has inherited that capability (even indirectly) from *Alice*. However, this revocation should not result in any other capabilities being revoked. If we express capabilities using a time model, then revocation reduces to optimistic rollback recovery. Consequently, the framework of distributed time may have some bearing on this problem.

- In order for capabilities to have meaning, their possession (or lack thereof) should affect the underlying computation. Having access both to a time model describing the unfolding computation and also to a time model expressing authority might provide a way to express the semantics of capabilities; having access to clocks for these models might provide a way to implement these semantics.

Enforcing access rights via capabilities is meaningless if malicious agents can subvert the underlying management system. Using the framework of distributed time has the additional benefit of providing transparent security and privacy.

The framework of distributed time also provides the potential for integrating capabilities with other temporal issues. For example, Herlihy and Tygar [HeTy89] use an approximation of real time as a basis for revoking capabilities. Distributed time might provide a way to generalize such work based on linear time. For example, how would capabilities be temporarily restored during rollback with modified replay?

**Information Confinement** A long-considered issue (e.g., [La73]) in computer security is the *confinement* of information to appropriate agents. Indeed, some researchers regard information confinement to be synonymous with security (e.g., [NCSC90]).

Tracking the flow of information in order to enforce *information confinement* is an area in which secure distributed time has particular relevance. This relevance arises for many of the same reasons as in the capabilities management problem. Information confinement requires using causality-like relations both to describe and also to proscribe computation, and requires careful consideration of temporal abstraction and security issues. The framework of distributed time may provide appropriate tools:

- The ability to express partial order time allows the potential to track correctly who has seen what.
- The ability to support flow-virtual time allows us to extend this potential for computations whose history is altered.
- The ability to support multiple levels of time allows us to track independent flows.
- The ability to support relations more general than partial orders allows us to consider the semantics of computation while tracking information flow. For example, nontransitive relations express process actions that destroy data.

**Mobile Computing** The advent of mobile computing raises a number of challenges related both to abstraction and to security. The framework of distributed time might provide tools for these issues.

Besides distribution and asynchrony, mobile computing raises an additional level of abstraction: networks with mobile agents must abstract from a dynamic physical topology to the more stable logical topology. Providing a hierarchy of time models might express this abstraction; providing clocks for these models might facilitate protocol construction. The dynamic physical topology creates additional security risks: for example, if a mechanism exists for *Alice* to appear suddenly in Cedar Springs, Michigan and have her communication routed to the server there, then *Bad Bob* in New York may abuse this mechanism to intercept *Alice's* communication. Expressing the abstraction using distributed time may provide techniques that transparently protect against these risks.

Disconnected operation also raises challenges with time and with confinement. For example, if a partition temporarily distributes *Alice's* computation among several physical sites, then she must re-establish a consistent image upon repair of the partition. We might be able to address this problem using the consistent global state tools of distributed time. *Remote execution* subverts the standard client/server model, since a portion of the client's computation may run on the server's machine, or vice-versa. Sharing a machine creates a *mutual suspicion* problem: one computation might interfere with or spy on the other. (For example, *Trojan Horses* and *viruses* are examples of such attacks.) The security and privacy tools of distributed time might address this problem.

**Hidden Causality** Real distributed systems frequently provide the potential for anonymous or hidden causality [Gr75]. The semaphore mechanism is an example: the agent granted a lock by a semaphore knows neither who released the lock nor who else is waiting for it. Extending our framework to handle these issues would be an interesting research area. Suppose *Alice* releases a lock which the semaphore grants to *Bob*. Does *Bob* now depend on *Alice*? Would vector clocks leak the identity of *Alice* to *Bob*? If *Alice* fails and rolls back, what should *Bob* do?

**Distributed Optimistic Execution** Much research (including the recent work of Leon [LFS93] and Cowan [CLB94]) has explored the uses of highly optimistic execution in distributed environments. For example, allowing long-running application programs to execute based on speculation (and to roll back if the speculation proves false) may provide increased performance (if the speculation is correct sufficiently often). The abstraction tools of distributed time framework may handle this distribution and the multiple time levels that may arise in such implementations; the security tools might provide transparent tolerance against Byzantine attacks on the machinery of optimism.

### 7.2.3. A Framework for the Future

The abstraction from linear time to partial orders and beyond has a precedent in the shift in physics from the classical world-view to the relativistic world-view. The comfortable, familiar perspective fails when simultaneity of information vanishes. The right perspective clarifies otherwise baffling behavior and also provides a way to continue to apply the comfortable perspective, once formal tools exist for changing frames of reference.

Linear time does not adequately describe the behavior of distributed systems. When failure recovery is allowed, a single level of partial order time also does not suffice—time must have *depth* as well as width. Levels of temporal abstraction will only become more necessary as computational environments become more complex, such as by admitting mobile agents, anonymous paths of influence, and the potential for cross-channel communication. Multiple levels of abstraction will multiply the problems of specifying and providing security and privacy in protocols running in such environments.

However, understanding how virtual partial orders arise from a hierarchy of time levels allows us to model the underlying behavior of the system, and to relativize the protocols and tools developed for the comfortable world of partial order time. Understanding how partial order protocols relate to the underlying time models also allows us to relativize the security challenges of timekeeping.

This thesis provides the fundamental contributions of a framework to understand the general temporal relations and the concomitant security challenges that arise and will continue to arise in distributed computation.

# Glossary

## *Terms*

| | | |
|---|---|---|
| acyclic | when a node does not precede itself in a graph; when a time model produces graphs with no cycles; when the global model in a parallel or nonlinear pair is acyclic | 15, 18 |
| adjusted rollback vector | the vector obtained from the rollback vector for a node by moving the other entries to their maximal preceding acyclic nodes | 33 |
| adjusted timestamp vector | the vector obtained from the timestamp vector for a node by moving the other entries to their minimal following acyclic nodes | 33 |
| antichain | in an order, a set of mutually incomparable elements | 26 |
| atom | a node or edge in a graph | 11 |
| bit-secure | cryptographic functions that leak no information | 120 |
| commitable | a state or event that will never be rolled back | 73 |
| complete recoverability | the assumption that any state in the live history at a non-faulty process is recoverable | 70 |
| computation graph | a directed graph describing computation | 11 |
| concurrent | when a time model leaves two nodes unordered | 11 |
| consistent cut | a cut that is also a timeslice | 31 |
| consistent pair | a parallel or nonlinear pair that is view-complete and transitively-bounded | 20 |
| consistent set | a set of user nodes whose live histories together comprise a past-closed prefix of a graph from FAILURE_FREE_PARTIAL_ORDER | 87 |
| cut | a set of nodes, exactly one at each process | 31 |

| | | |
|---|---|---|
| digital signature | a function that only a privileged agent can perform, but anyone can check | 120 |
| externally equivalent | when two atoms at a process afford the same external view | 20 |
| factoring model | in a nonlinear or parallel pair, the model induced from the local model to the global model | 18 |
| flow-supported | when transitive precedence in a model implies information flow in any underlying computation; when each model in a parallel or nonlinear pair is flow-supported | 17, 18 |
| flow-virtual | when information flow does not necessarily imply time model precedence; when the transitive closure of each model in a parallel or nonlinear pair is flow-virtual | 17, 18 |
| global state | in a ground-level graph, a minimal set of atoms that represents an instant of time in an underlying computation | 27 |
| ground-level computation graphs | the "least abstract" computation graphs, constructed directly from traces | 12 |
| independent | when a parallel or nonlinear pair has the property that each non-extremal node in the global model represents a unique node in the local model | 20 |
| join | in lattices, the least upper bound of two elements; for vectors, the entry-wise maximum | 33 |
| lattice | a nonempty ordered set closed under meet and join | 33 |
| live history | a user node along with its past from the $\overline{\text{USER\_PARTIAL\_ORDER}}$ | 78 |
| maximal node | a node with no successors | 11 |
| meet | in lattices, the greatest lower bound of two elements; for vectors, the entry-wise minimum | 33 |
| minimal generating set | in a timeslice $X$, a minimal subset of nodes whose adjusted timestamp vectors join to yield timeslice $X$ | 55 |
| minimal node | a node with no predecessors | 11 |
| modified replay | when the computation after rollback differs from the original computation | 66 |

| | | |
|---|---|---|
| node-monotonic | when a time model on ground-level graphs has the property that once it produces a node, the node never vanishes | 17 |
| nonlinear pair | a pair of related models providing (respectively) system-wide and process-only descriptions of computation; the process descriptions need not be linear | 21 |
| optimistic rollback | approaches that bet failure will not happen, and allow orphans to develop at non-faulty processes | 68 |
| order | a relation that is transitive and antisymmetric | 11 |
| orphan | a node that depends on or equals a rolled-back node | 64 |
| parallax | when two snapshots of the same computation could not both have been real simultaneous states | 58 |
| parallel pair | a pair of related models providing (respectively) system-wide and process-only descriptions of computation; the process descriptions must be linear | 18 |
| partial timeslice | a subset (not necessarily proper) of a timeslice | 26 |
| past-closed | when nodes in a subgraph have the same history as they do in the original graph | 12 |
| past-closure | a subgraph minimally extended to make it past-closed | 12 |
| pessimistic rollback | approaches that bet failure will happen, and prevent orphans from developing at non-faulty processes | 68 |
| piecewise deterministic | when a process's computation between message receive events is completely determined by the state before the first receive and contents of the message | 72 |
| prefix | a subgraph that is connected and that contains the minimal nodes | 12 |
| pseudo-vector | an array of node sets, one for each process | 80 |
| public key cryptography | an encryption function that anyone can perform, but only a privileged agent can invert | 120 |
| refinement | time model $\mathbf{M}_1$ refines to time model $\mathbf{M}_2$ when $\mathbf{M}_1(\alpha) = \mathbf{M}_1(\alpha')$ always implies $\mathbf{M}_2(\alpha) = \mathbf{M}_2(\alpha')$ | 15 |

| | | |
|---|---|---|
| representation map | a function (induced by the application of a time model to a graph) that takes atoms in the image graph back to sets of atoms in the original graph | 13 |
| rollback vector | the minimal nodes at each process that follow or equal a given node | 32 |
| stable | when a property remains true once it becomes true; when a state or event has been successfully logged to stable storage | 44, 73 |
| state interval | a sequence of states and events representing a period of deterministic execution at a process | 105 |
| strongly edge-monotonic | when a time model on ground-level graphs has the property that once it produces two nodes, the relation between those nodes is fixed | 17 |
| strongly monotonic | when a time model is node-monotonic and strongly edge-monotonic; when the transitive closure of each model in a parallel or nonlinear pair is strongly monotonic | 17, 18 |
| time model | a representative transformation of computation graphs | 13 |
| timeslice | a maximal set of mutually concurrent (hence acyclic) nodes | 26 |
| timestamp pseudo-vector | in a nonlinear pair, the maximal nodes in the local model that precede or equal a given node in the global model | 80 |
| timestamp vector | the maximum node at each process that precedes or equals a given node | 32 |
| timetree | the tree-structure on a process's events that emerges instead of timelines in the USER_PARTIAL_ORDER | 78 |
| trace | an exhaustive real-time description of a computation | 10 |
| transitively bounded | when the transitive closure of a model produces unique maximal and minimal nodes; when the global model in a parallel or nonlinear pair is transitively bounded | 15, 18 |
| Type 1 | a parallel or nonlinear pair that is consistent | 20 |
| Type 2 | a parallel or nonlinear pair that is consistent and independent | 20 |

| | | |
|---|---|---|
| Type 3 | a parallel or nonlinear pair that is consistent, independent, and strongly monotonic | 20 |
| Type 4 | a parallel or nonlinear pair that is consistent, independent, strongly-monotonic, and flow-supported | 20 |
| valid | when the live history of a user node is a past-closed prefix of a graph from the FAILURE_FREE_PARTIAL_ORDER | 87 |
| vector | an array of nodes, one from each process | 31 |
| view-complete | when a graph from the global model in a pair has the property that for any edge at process, there exists a node that is externally equivalent in the transitive global graph; when a parallel or nonlinear pair always produces view-complete graphs | 20 |
| weakly edge-monotonic | when a time model on ground-level graphs has the property that once it produces an edge, the edge never vanishes | 17 |
| weakly monotonic | when a time model is node-monotonic and weakly edge-monotonic; when the transitive closure of each model in a parallel or nonlinear pair is weakly monotonic | 17, 18 |

## *Clock Primitives*

| | | |
|---|---|---|
| *ACYCLIC* | clock primitive testing if a node is acyclic | 35 |
| *COMPARE* | clock primitive comparing two vectors | 36 |
| *CONCURRENT* | clock primitive testing if two nodes are concurrent | 34 |
| *CUR_GRAPH* | universal variable for current ground-level graph | 34 |
| *CUR_NODE* | clock primitive returning current node | 36 |
| *LIST* | clock meta-primitive, listing nodes from a specified graph with a specified property | 36 |
| *LIST_CONCURRENT* | clock primitive listing all nodes at a process concurrent with a given node | 37 |

## *Time Models*

| | | |
|---|---|---|
| IMPLEMENT | time model expressing how to construct the USER_PARTIAL_ORDER from the SYSTEM_PARTIAL_ORDER | 77 |
| NET_ABSTRACT | time model abstracting away network activity | 21 |
| PARTIAL_ORDER_TIME | time model organizing process activity into a partial order | 25 |
| STRONG_PARTIAL_ORDER | a "partial order" time model with bidirectional message edges | 53 |
| STRONG | time model making cross-process edges bidirectional | 53 |
| SYSTEM_PARTIAL_ORDER | partial order time model for the recovery computation | 76 |
| SYSTEM_TIMELINES | timelines time model for the recovery computation | 76 |
| TIMELINES | time model organizing process activity into timelines | 22 |
| TIMETREES | "timelines" time model expressing logical local precedence for user computation—hence process structure is a tree, not a line | 78 |
| TRANS | time model performing transitive closure | 18 |
| USER_PARTIAL_ORDER | partial order time model that examines only the state of the *implemented* process, and expresses logical precedence | 78 |

# *Symbols*

| | | |
|---|---|---|
| $A \longrightarrow B$ | node $A$ precedes node $B$ | 11 |
| $A \mathrel{\underline{\longrightarrow}} B$ | node $A$ precedes or equals node $B$ | 11 |
| $A \not\longleftrightarrow B$ | nodes $A$ and $B$ are incomparable | 11 |
| $\mathbf{M}_1 \rhd \mathbf{M}_2$ | model $\mathbf{M}_1$ refines to model $\mathbf{M}_2$ | 15 |
| $\langle\, \mathbf{M},\, \alpha \,\rangle$ | the representation map induced by applying model $\mathbf{M}$ to graph $\alpha$ | 13 |
| $\overline{\mathbf{M}}$ | the transitive closure of model $\mathbf{M}$ | 18 |

| | | |
|---|---|---|
| $[\alpha_i]$ | a sequence of unfolding ground-level graphs representing a computation in progress | 17 |
| $\mathcal{S}_{\mathbf{M},\beta}$ | the set of ground-level graph sequences that, at some point, generate $\beta$ through $\mathbf{M}$ | 17 |
| $(\mathbf{M}, \mathbf{M}')$ | a parallel or nonlinear pair; $\mathbf{M}$ is the global system model and $\mathbf{M}'$ is the local process model | 18, 21 |
| $\mathbf{M}/\mathbf{M}'$ | the factoring model for pair $(\mathbf{M}, \mathbf{M}')$ | 18 |
| $\pi_p X$ | the process $p$ entry of $X$ | 31 |
| $\mathbf{R}(A)$ | the rollback vector of $A$ | 32 |
| $\mathbf{R}^*(A)$ | the adjusted rollback vector of $A$ | 33 |
| $\mathbf{V}(A)$ | the timestamp vector of $A$ | 32 |
| $\mathbf{V}^*(A)$ | the adjusted timestamp vector of $A$ | 33 |
| $\mathbf{V}'(A)$ | the timestamp pseudo-vector of $A$ | 80 |
| $X \prec Y$ | timeslice $X$ precedes timeslice $Y$ | 32 |
| $X \sqcap Y$ | the meet of $X$ and $Y$ | 33 |
| $X \sqcup Y$ | the join of $X$ and $Y$ | 33 |

# References


[ACGS91] M. Ahuja, T. Carlson, A. Gahlot and D. Shands. *Timestamping Events for Inferring "Affects" Relation and Potential Causality.* Computer and Information Science Technical Report OSU-CISRC-5/91-TR13, Ohio State. May 1991.

[AhKs89] M. Ahuja and A. D. Kshemkalyani. *Characterization of Global Snapshots and a Survey of Global Snapshot Algorithms.* Computer and Information Science Technical Report OSU-CISRC-10/89-TR46, Ohio State. October 1989.

[ACGS88] W. Alexi, B. Chor, O. Goldreich and C. P. Schnorr. "RSA and Rabin Functions: Certain Parts are as Hard as the Whole." *SIAM Journal on Computing*, 17:194-209. 1988

[AmJa93] P. Amman and S. Jajodia. "Distributed Timestamp Generation in Planar Lattice Networks." *ACM Transactions on Computer Systems.* Preprint.

[Aw85] B. Awerbuch. "Complexity of Network Synchronization." *Journal of the ACM.* 32: 804-823. October 1985.

[Ba93] F. A. Barber. "A Metric Time-Point and Duration-Based Temporal Model." *SIGART Bulletin*, 4 (3): 30-49. 1993.

[BHS93] D. Bayer, S. Haber, and W. S. Stornetta. "Improving the Efficiency and Reliability of Digital Time-Stamping." *Sequences II: Methods in Communication, Security, and Computer Science.* Springer Verlag, 1993.

[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.

[Bi94] K. P. Birman. "A Response to Cheriton and Skeen's Criticism of Causal and Totally Ordered Communication." *ACM Operating Systems Review.* 28: 11-21. January 1994.

[BiJo87] K. P. Birman and T. A. Joseph. "Reliable Communication in the Presence of Failures." *ACM Transactions on Computer Systems,* 5: 47-76. February 1987.

[BlGo84] M. Blum and S. Goldwasser. "An Efficient Probabilistic Public-Key Encryption Scheme which Hides All Partial Information." *Advances in Cryptology: Proceedings of Crytpo 84.* Springer Verlag LNCS 196.

[Bo93] M. Boddy. "Temporal Reasoning for Planning and Scheduling." *SIGART Bulletin*, 4 (3): 17-25. 1993.

[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.

[BCS84] D. Briatico, A. Ciuffoletti, and L. Simoncini. "A Distributed Domino Effect Free Recovery Algorithm." *IEEE Symposium on Reliability in Distributed Software and Database Systems.* October 1984.

[CCMP89] R. Casley, R. Crew, J. Meseguer, and V. Pratt. "Temporal Structures." *Category Theory and Computer Science*. Springer-Verlag LNCS 389, 1989

[Ch89] K. M. Chandy. *The Essence of Distributed Snapshots.* Computer Science Technical Report CS TR 89-5, Caltech. March 1989.

[ChLa85] K. M. Chandy and L. Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems." *ACM Transactions on Computer Systems.* 3: 63-75. February 1985.

[CB89] B. Charron-Bost. "Combinatorics and Geometry of Consistent Cuts: Application to Concurrency Theory." In J. C. Bermond and M. Raynal (ed.), *Proceedings of the Third International Workshop on Distributed Algorithms.* Springer-Verlag LNCS 392, 1989.

[CB91] B. Charron-Bost. "Concerning the Size of Logical Clocks in Distributed Systems." *Information Processing Letters*. 39: 11-16. July 1991.

[ChSk93] D. R. Cheriton and D. Skeen. "Understanding the Limitations of Causally and Totally Ordered Communication." *Fourteenth ACM Symposium on Operating Systems Principles.* December 1993.

[Ci84] A. Ciuffoletti. "Error Recovery in Systems of Communicating Processes." *IEEE International Conference on Software Engineering*. March 1984.

[Ci89] A. Ciuffoelleti. "La Coordinazione Delle Attivita Di Ripristino Nei Sistemi Distribuiti." *A.I.C.A. Annual Conference Proceedings.* October 1989.

[Co94] R. Cooper. "Experience with Causally and Totally Ordered Communication Support—a cautionary tale." *ACM Operating Systems Review.* 28: 28-32. January 1994.

[CoMa91] R. Cooper and K. Marzullo. "Consistent Detection of Global Predicates." *ACM SIGPLAN Notices.* 26 (12): 167-174. December 1991.

[CLB94] C. Cowan, H. Lutfiyya, and M. Bauer. "Increasing Concurrency through Optimism: A Reason for HOPE." *ACM Computer Science Conference*, March 1994.

[CrTa90] C. Critchlow and K. Taylor. "The Inhibition Spectrum and the Achievement of Causal Consistency." *Ninth ACM Symposium on Principles of Distributed Computing*, 1990.

[DaPr90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge: Cambridge University Press, 1990.

[DiHe76] W. Diffie and M. E. Hellman. "New Directions in Cryptography." *IEEE Transactions on Information Theory*. IT-22: 644-654. November 1976.

[EJZ92] E. N. Elnozahy, D. B. Johnson and W. Zwaenepoel. "The Performance of Consistent Checkpointing." *Eleventh IEEE Symposium on Reliable Distributed Systems.* 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." *Eleventh Australian Computer Science Conference.* 56-67. February 1988.

[Fi89] C. J. Fidge. "Partial Orders for Parallel Debugging." *ACM SIGPLAN Notices.* 24: 183-194. January 1989.

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

[GaPr87] H. Gaifman and V. Pratt. "Partial Order Models of Concurrency and the Computation of Functions." *Logic in Computer Science*, 72-85, 1987.

[GaWa94] V. K. Garg and B. Waldecker. "Detection of Weak Unstable Predicates in Distributed Systems." *IEEE Transactions on Parallel and Distributed Systems*, to appear. Reprinted in [YaMa94].

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

[GoMi82] S. Goldwasser and S. Micali. "Probabilistic Encryption and How to Play Mental Poker Keeping Secret All Partial Information." *Fourteenth ACM Symposium on Theory of Computing*, 1982.

[Gr75] I. G. Greif. *Semantics of Communicating Parallel Processes.* Ph.D. thesis, Massachusetts Institute of Technology. 1975.

[GrRe93] J. Gray and A. Reuter. *Transaction Processing*. Morgan-Kaufman, 1993.

[HaSt91] S. Haber and W. S. Stornetta. "How to Time-Stamp a Digital Document." *Journal of Cryptology.* 3 (2): 99-111. 1991.

[HLMW87] M. P. Herlihy, N. Lynch, M. Merritt and W. Weihl. *On the Correctness of Orphan Elimination Algorithms.* Computer Science Technical Report MIT LCS TM-329, Massachusetts Institute of Technology. 1987.

[HeTy89] M. P. Herlihy and J. D. Tygar. "Implementing Distributed Capabilities Without a Trusted Kernel." In Avizienis and Laprie (ed.), *Dependable Computing for Critical Applications.* Springer-Verlag.

[He91] C. A. Heydon. *Processing Visual Specifications of File System Security.* Ph.D. thesis, Computer Science, Carnegie Mellon University. 1991.

[Je85] D. R. Jefferson. "Virtual Time." *ACM Transactions on Programming Languages and Systems.* 7: 404-425. July 1985.

[JoZw87] D. B. Johnson and W. Zwaenepoel. "Sender-Based Message Logging." *Seventeenth Annual International Symposium on Fault-Tolerant Computing*. 14-19. 1987.

[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." *Twelfth IEEE Symposium on Reliable Distributed Systems.* October 1993.

[KeKo89] P. Kearns and B. Koodalattupuram. "Immediate Ordered Service in Distributed Systems." *Ninth IEEE Symposium on Reliable Distributed Systems.* 1989.

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

[KsSi90] A. D. Kshemkalyani and M. Singhal. *Efficient Detection and Resolution of Generalized Distributed Deadlocks*. Computer and Information Science Technical Report OSU-CISRC-10/90- TR30, Ohio State University. October 1990.

[LaYa87] T. H. Lai and T. H. Yang. "On Distributed Snapshots." *Information Processing Letters.* 25: 153-158. May 1987.

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

[La73] B. W. Lampson. “A Note on the Confinement Problem.” *Communications of the ACM*. 10: 613-615. October 1973.

[LFS93] J. Leon, A. Fisher and P. Steenkiste. *Fail-Safe PVM: A Portable Package for Distributed Programming with Transparent Recovery.* Computer Science Technical Report CMU-CS-TR-93-124, Carnegie Mellon University.

[LeBh88] P. Leu and B. Bhargava. “Concurrent Robust Checkpointing and Recovery in Distributed Systems.” *Fourth International Conference on Data Engineering.* 154-163. 1988.

[LRV87] H. F. Li, T. Radhakrishnan and K. Venkatesh. “Global State Detection in Non-FIFO Networks.” *Seventh International Conference on Distributed Computing Systems.* 1987.

[LNP90] K. Li, J. F. Naughton and J. S. Plank. “Real-Time, Concurrent Checkpointing for Parallel Programs.” *Second ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming.* 79-88. 1990.

[Li81] R. Lipton. *How to Cheat at Mental Poker.* Personal communication. 1981.

[MaMo83] M. J. Manthey and B. E. Moret. “The Computational Metaphor and Quantum Physics.” *Communications of the ACM*. 26: 137-144. February 1983.

[Ma90a] M. J. Manthey. *Synchronization: the Mechanism of Conservation Laws*. Mathematics and Computer Science Technical Report R90-16, the University of Aalborg. April 1990.

[Ma90b] M. J. Manthey. *Hierarchy and Convergence: a Computational View.* Mathematics and Computer Science Technical Report R90-25, the University of Aalborg. May 1990.

[MaNe91] K. Marzullo and G. Neiger. “Detection of Global State Predicates.” In Toueg, Spirakis and Kirousis (ed.), *Fifth International Workshop on Distributed Algorithms (WDAG-91).* Springer-Verlag LNCS 579. 1991.

[MaSa91] K. Marzullo and L. Sabel. “Using Consistent Subcuts for Detecting Stable Properties.” In Toueg, Spirakis and Kirousis (ed.), *Fifth International Workshop on Distributed Algorithms (WDAG-91).* Springer-Verlag LNCS 579. 1991.

[Ma87] F. Mattern. “Algorithms for Distributed Termination Detection.” *Distributed Computing*. 2: 161-175. 1987.

[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.

[Ma93] F. Mattern. “Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation.” *Journal of Parallel and Distributed Computing.* 18: 423-434. August 1993.

[MeRa78] P. M. Merlin and B. Randell. “State Restoration in Distributed Systems.” *International Symposium on Fault-Tolerant Computing.* June 1978.

[Mo85] C. Morgan. “Global and Logical Time in Distributed Algorithms.” *Information Processing Letters.* 20 : 189-194. May 1985.

[NCSC90] National Computer Security Center. *Trusted Network Interpretation Environments Guideline NCSC-TG-011.* United States Government Printing Office, 1990.

[NeTo90] G. Neiger and S. Toueg. *Simulating Synchronized Clocks and Common Knowledge in Distributed Systems.* Computer Science Technical Report TR-90-1086, Cornell University. January 1990.

[PBS89] L. L. Peterson, N. C. Bucholz and R. D. Schlichting. “Preserving and Using Context Information in Interprocess Communication.” *ACM Transactions on Computer Systems.* 7: 217-246. August 1989.

[PeKe93] S. L. Peterson and P. Kearns. “Rollback Based on Vector Time.” *Twelfth IEEE Symposium on Reliable Distributed Systems.* October 1993.

[Pe80] C. Petri. “Concurrency.” In Brauer, ed., *Net Theory and Applications.* Springer Verlag, 1980. Pp. 251-260.

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

[Pr86] V. Pratt. “Modeling Concurrency with Partial Orders.” *International Journal of Parallel Programming.* 15 (1): 33-71. 1986.

[Pr92] V. Pratt, personal communication, October 1992.

[Ra79] M. Rabin. *Digitalized Signatures and Public-Key Functions as Intractable as Factorization.* Laboratory for Computer Science Technical Report MIT/LCS/TR-212, Massachusetts Institute of Technology. January 1979.

[Ra75] B. Randell. “System Structure for Fault Tolerance.” *IEEE Transactions on Software Engineering.* SE-1: 220-232, 1975.

[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.*

[Re94] R. van Renesse. “Why Bother with CATOCS?” *ACM Operating Systems Review.* 28: 22-27. January 1994.

[RSA78] R. L. Rivest, A. Shamir and L. Aldeman. “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.” *Communications of the ACM.* 21 (2): 120-126. February 1978.

[Ru80] D. L. Russell. “State Restoration in Systems of Communicating Processes.” *IEEE Transactions on Software Engineering.* 6 (2): 183-194. March 1980.

[SES89] A. Schiper, J. Eggli and A. Sandoz. “A New Algorithm to Implement Causal Ordering.” In J. C. Bermond and M. Raynal (ed.), *Proceedings of the 3rd International Workshop on Distributed Algorithms.* Springer-Verlag LNCS 392, 1989.

[SEC94] Securities and Exchange Commission, Division of Market Regulation. *Market 2000: An Examination of Current Equity Market Developments.* United States Government Printing Office, 1994.

[Sh79] A. Shamir. “How to Share a Secret.” *Commuications of the ACM.* 22 (11): 612-613. November 1979.

[SiKs90] M. Singhal and A. D. Kshemkalyani. *An Efficient Implementation of Vector Clocks.* Computer and Information Science Technical Report 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*, 1989.

[Sm91] S. W. Smith. *Secure Clocks for Partial Order Time.* Thesis proposal, School of Computer Science, Carnegie Mellon University. October 30, 1991. (See [SmTy91].)

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

[SJT94] S. W. Smith, D. B. Johnson, and J. D. Tygar. *Asynchronous Optimistic Rollback Recovery using Secure Distributed Time.* Computer Science Technical Report CMU-CS-94-130, March 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. (The majority of [SmTy91] is drawn verbatim from [Sm91].)

[SmTy94] S. W. Smith and J. D. Tygar. “Security and Privacy for Partial Order Time.” *Seventh International Conference on Parallel and Distributed Computing Systems.* October 1994. (A preliminary version is available as Computer Science Technical Report CMU-CS-94-135, Carnegie Mellon University, April 1994.)

[SpKe86] M. Spezialetti and P. Kearns. “Efficient Distributed Snapshots.” *Sixth International Conference on Distributed Computing Systems.* 1986.

[Sp89] M. Spezialetti. *A Generalized Approach to Monitoring Distributed Computations for Event Occurrences.* Ph.D. thesis, University of Pittsburgh, 1989.

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

[TaLo91] Y. C. Tay and W. T. Loke. *A Theory for Deadlocks.* Computer Science Technical Report CS-TR-344-91, Princeton University. August 1991.

[Ta89] K. Taylor. "The Role of Inhibition in Asynchronous Consistent-Cut Protocols." In J. C. Bermond and M. Raynal (ed.), *Proceedings of the Third International Workshop on Distributed Algorithms.* Springer-Verlag LNCS 392, 1989.

[ToGa93] A. I. Tomlinson and V. K. Garg. "Detecting Relational Global Predicates in Distributed Systems." *ACM SIGPLAN Notices.* 28 (12): 21-31. December 1993.

[Ts87] E. P. K. Tsang. "Time Structures for AI." *Proceedings IJCAI-87.* 456-461.

[TyYe93] J. D. Tygar and B. S. Yee. "Dyad: A System for Using Physically 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.)

[Va93] C. Valot. "Characterizing the Accuracy of Distributed Timestamps." *ACM SIGPLAN Notices.* 28 (12): 43-52. December 1993.

[Va94] L. Van Valkenburgh, personal communication, July 12, 1994.

[Ve89] S. Venkatesan. "Message-Optimal Incremental Snapshots." *Ninth International Conference on Distributed Computing Systems.* 1989.

[Wein87] S. H. Weingart. "Physical Security for the $\mu$ABYSS System." *Proceedings of the 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 Reserch Center. March 1991.

[Win89] G. Winskel. *An Introduction to Event Structures.* Computer Science Technical Report 278, Aarhus University. April 1989.

[YaAl93] E. Yampratoom and J. F. Allen. "Performance of Temporal Reasoning Systems." *SIGART Bulletin*, 4 (3): 26-29.

[YaMa93] Z. Yang and T. A. Marsland. "Annotated Bibliography on Global States and Times in Distributed Systems." *ACM Operating Systems Review.* 27 (3): 55-74. July 1993.

[YaMa94] Z. Yang and T. A. Marsland. *Global States and Time in Distributed Systems.* IEEE Computer Science Press, 1994.

[Yee94] B. S. Yee. *Using Secure Coprocessors*. Ph.D. thesis. Computer Science Technical Report CMU-CS-94-149, Carnegie Mellon University. May 1994.

[Zw88] W. Zwaenepoel. *A Theoretical Framework for Optimistic Recovery in Distributed Systems.* Computer Science Technical Report TR88-64, Rice University. February 1988.