Carnegie Mellon Computer Science Technical Report CMU-CS-93-116.

Signed Vector Timestamps: A Secure Protocol for Partial Order Time

Sean W. Smith J.D. Tygar

October 1991; version of February 1993

CMU-CS-93-116

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The language of partial order time expresses the issues central to many problems in asynchronous distributed systems. A secure partial order time service would provide a general method to develop secure protocols for these problems. In this paper, we sketch out these issues and develop one such protocol: *signed vector timestamps.* The majority of this paper is drawn verbatim from the first author's October 1991 thesis proposal, the first research into security issues for non-scalar time services and the original presentation of the SVT protocol.

This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Order No. 7597. S. Smith also received support from an ONR Graduate Fellowship and J.D. Tygar from NSF Presidential Young Investigator Grant CCR-8858087.

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

Keywords: security, cryptographic controls, distributed systems, concurrency

1. Introduction

The language of partial order time expresses the issues central to many problems in asynchronous distributed systems. A secure partial order time service would provide a general method to develop secure protocols for these problems. In this paper, we sketch out these issues and develop one such protocol: *signed vector timestamps.* This paper is drawn verbatim from the first author's October 1991 thesis proposal[1] [20], except for minor edits, the concluding Sections 5 and 6, and this paragraph. The original proposal document gives the first research into security issues for non-scalar time services and the original presentation of the SVT protocol. (It has recently come to our attention that this protocol was later independently rediscovered. [18])

Traditionally, we regard time as a scalar value, totally ordering on the events in a system. However, the very nature of asynchronous distributed systems suggests that we should use an order that is partial, not total, so that we can deliberately leave unordered two separated events that have no knowledge of each other. In this partial order time model, both the presence and the absence of a path between two events carry meaning—whether one event necessarily precedes the other, or whether they are concurrent. If we use merely a total order, we lose the latter information.

Many problems in distributed systems reduce to questions about this partial order. Our current research explores building tools that explicitly grant these abilities, thus providing a general method to develop protocols to solve problems in this class—known problems that currently have separate *ad hoc* solutions, and also new problems that arise from this unified framework. Our research also explores making these tools robust for various models of Byzantine failure and information confinement; thus, protocols based on these tools will be secure and robust, since they will inherit the security properties already present in the toolkit.

2. Partial Order Time

Partial order time provides an alternative way to order events in an asynchronous distributed system. The goal of the first author's thesis [21] is to design a family of protocols that allow processes in a system to examine local events in terms of this time model.

The concept of partial order time solves some of the difficulties introduced by merging independent timelines into the same totally ordered stream. Using only a partial order on events lets us ensure that event a happens "after" event b if and only if a can observe the results of b—total orders only allow the converse direction. Deliberately leaving unordered two events that lie outside each other's "observation cone" frees us from the paradoxes of conflicting knowledge horizons.

[1] The proposal document is available by request from the School of Computer Science, Carnegie Mellon University, and also by ftp on `lunch.trust.cs.cmu.edu` as `/usr/smith/public/PROPOSAL.ps`.

A total order $<$ is *consistent* with partial order $\prec$ when $a \prec b \Longrightarrow a < b$. (If we think of orders as a set of ordered pairs, than a consistent total order is just a total order that contains the partial order as a subset.) Any partial order extends to a consistent total order; further, the set of consistent total orders uniquely characterizes a partial order. Research on concurrent systems raises ideas of partial order time precisely because of the need to reason about this entire set. Total order time—even the total order provided by real time—provides only one member.

2.1. Formal Definitions

We base our partial order time model on Lamport's. [11] Formally, let us define an event to be an instantaneous, atomic action within a system (as per Mattern [14]). Each event takes place at one specific process. We partition events into three categories:

1. *send* events, in which one process sends a message to another
2. *receive* events, in which one process receives a message from another
3. *internal* events—anything else that happens within a process

Send events take place at the sending process; *receive* events at the receiving process. Note that since communication is asynchronous, a *send* event does not have to be simultaneous with a *receive* event; depending on the failure model we use, a *send* event may not even have a corresponding *receive*.

Isolating each event in a distributed system—e.g., requiring each process to throw away its state after each event—would render irrelevant any discussion of event ordering. Only when events can observe the results of previous events does the issue arise of deciding which events are indeed "previous." To capture this notion of "previous," we will construct the *basic partial order* (BPO) on events: we will write $a \longrightarrow b$ to indicate the event b potentially depends on event a: that is, event a must be in the past in the timeline experienced by b. One interpretation of the BPO is that it expresses the basic flow of causality; a less mystical interpretation is that it specifies some minimum required level of structure in possible time sequences.

To define the BPO, we proceed from two basic rules:

- Recall that we assume that uniprocessors can totally order their own events. If events a and b occur on the same process and a precedes b in this order, then let $a \longrightarrow b$.
- Processes only influence other processes by sending messages, and are influenced only by receiving them. So, if a is the sending of a message and b is reception of that message, then $a \longrightarrow b$.

Formally, we let the BPO be the transitive closure of this relation. Note that for two events a and b, exactly one of three cases holds.

- $a \longrightarrow b$: b depends on a
- $b \longrightarrow a$
- $a \not\longrightarrow b$ and $b \not\longrightarrow a$; in this case, we say that a and b are *concurrent*.

We will write $a \not\longleftrightarrow b$ to indicate the latter relationship.

2.2. The Graph Interpretation

Interpreting the BPO as a directed acyclic graph (DAG) makes discussing some of its properties easier. Construct a node for each event in the system, and draw directed edges according to the two basic rules above. Then the relation $a \longrightarrow b$ holds exactly when a path exists from a to b.

Regarding the BPO as a graph—without transitive closure—allows us two different ways to define restrictions on a BPO. Let S be a subset of the events (perhaps those events occurring at some subset of the processes).

- We construct the *nontransitive restriction* of the BPO to S simply by deleting all nodes not in S, and all edges incident to these nodes.
- We construct the *transitive restriction* of the BPO by first taking the transitive closure of the graph, and then deleting the nodes and edges not in S. This is the standard restriction for partial orders.

We will use the notation $a \xrightarrow{S} b$ to indicate that event b depends on a under the nontransitive restriction of the BPO to S, and $a \xrightarrow{\hat{S}} b$ under the transitive restriction.

3. Secure Clocks for Partial Order Time

This paper proposes a secure toolkit for distributed partial clocks. We now offer a more detailed discussion on what we mean by this—Section 3.1 presents the basic issues involved in defining these clocks, and Section 3.2 examines security and robustness issues.

3.1. Clocks for Partial Order Time

The problem of robustly implementing a traditional clock on a distributed system (where by "clock" we refer to a global event counter, although some ideas extend to approximations of real time) is difficult but solvable (e.g., [12],[13],[22]). Researchers observe that a necessary condition for distributed clocks is that the total order calculated be consistent with the BPO.

That is, the system computes a time function T, mapping events to integer timestamps, such that for all events a, b,

$$T(a) < T(b) \implies a \longrightarrow b$$

However, we stress the importance of a system being able to calculate the BPO exactly. Our goal is to implement a *distributed partial clock*: we want a timestamp set, with partial order $\prec$; a function T from events to timestamps satisfying

$$T(a) \prec T(b) \iff a \longrightarrow b;$$

and the ability for processes *within the distributed system* to compute the function T and the comparison $\prec$.

More precisely, we want our partial clock toolkit to enable processes to be able to calculate these functions for the events they know about: process P_i need only calculate T and $\prec$ on some subset E_i containing the events *perceivable* by P_i. Defining this notion is a bit tricky. The weakest nontrivial definition follows:

- if event a occurs at P_i, then $a \in E_i$
- if event a is the sending of a message to P_i, which P_i received, then $a \in E_i$

Note that this is nontrivial because, if events a, b are the sending of messages to process P_i and event c is internal, then answering the questions of whether $a \longrightarrow b$ or $c \longrightarrow a$ may require information not easily available to this process. This definition is still rather weak: suppose the message sent to process P_i in event a at process P_j contains information about events preceding a? One could argue that P_i ought to be able order those event too.[2] Further, suppose that event b is the reception at process P_j of message m sent by event a at P_i. Should $b \in E_i$? Clearly P_i knows that its event a influenced b—but does P_i necessarily know that b exists?

In the spirit of saying that our definition of BPO is purely syntactic, we claim that this weak definition of E_i is the corresponding purely syntactic version. As with BPO, we can construct more complicated extensions of this basic concept by considering other issues.

3.2. Security Issues

The problems of robustness and security in distributed partial clocks take two forms: fault tolerance, and some special challenges the nature of partial clocks creates for information confinement.

[2]This fact—that the BPO is transitive but this notion of "perceivability" is not—will cause problems when consider information confinement (in Section 3.2).

3.2.1. Fault Tolerance

A natural question to ask when considering a distributed system that consists of a physically distributed collection of machines is: what happens when one of them goes awry? In our distributed systems model we have several elements:

- physical processors
- communication links between processors
- processes running on processors

Physical machines can fail (either gracefully or maliciously); processes can be downright malevolent; processes go into suspension while their machine is down, or when they move operation to a different machine; communication links can deliver messages out of order, or garbled, or not at all.

(In the remainder of this paper, we make the simplifying assumptions that each process resides on its on processor, and that the network never corrupts messages.)

We would like our distributed partial clocks to maintain some kind of reasonable performance in the face of such troubles. We can imagine the standard spectra measuring severity of individual failures and number of such failures, with a family of implementations that achieve increasing levels of performance on these spectra, probably by trading off against simplicity and efficiency, and by balancing the various types of robustness.

However, a new issue is exactly what we should regard as "reasonable performance." The functions we wish our clocks to calculate capture distributed, global properties. Even though events a and b might occur in the immediate proximity of a process P_i (e.g., in the weakest E_i), the individual arcs in the BPO graph that cause $a \longrightarrow b$ to hold might be distributed throughout the entire system. We could require the nonfaulty processes to calculate the BPO correctly on their perceivable events; less strongly, we could restrict these events to those belonging to nonfaulty processes (so we absolve nonfaulty P_i from any confusion that a message from a faulty process causes). Some of our work already suggests even weaker fault tolerance: requiring nonfaulty processes only to calculate the nontransitive restriction[3] of the BPO to the set of nonfaulty processes. Each of these cases partitions the set of processes, and hence the set of events, into nonfaulty and faulty categories, but only specifies how events in the former should be handled. How nonfaulty processes should deal with bad events raises another set of research questions.

3.2.2. Information Confinement

To illustrate another set of security issues, we now consider an especially naive implementation of partial clocks. Suppose a distributed system explicitly maintains the BPO graph.

[3]Recall the definition in Section 2.2.

After initialization, each process starts building a linear chain of its internal events. When sending a message, a process sends along its chain; when receiving a message, a process incorporates the graph information contained into its own graph. Consequently, whenever a process executes an event, it knows the entire BPO subgraph induced by taking all the ancestors of that event. This implementation allows processes to calculate the T and $\prec$ relations. However, even aside from questions of efficiency and fault tolerance, this implementation would be unsatisfactory in two crucial areas: reasons of security policy and reasons of innate causality may render it undesirable or impossible for a process to know the complete history behind every event.

Confinement by policy. Recall in Section 3.1 we offered a weakest definition of the events perceivable by a process: E_i, consisting of the events internal to process P_i and the send events of messages received by process P_i. In many real instances of distributed systems we may want to enforce an *information confinement* rule such as "process P_i can know nothing of the global BPO graph except its transitive restriction to E_i, unless authorization is explicitly granted in some way."

For example, consider distributed workstations in a university environment. Just because Alice sends a message to Bob does not mean Bob has the right to know everything Alice has been doing. We need to consider confinement from the future as well: professors Bob and Carla may need to have a lengthy discussion of student Alice's proposal—but naturally Alice should not be privy to this discussion, or even to the fact that "a lengthy discussion of my proposal is going on."

We formalize these concepts by introducing two new terms:

- *forward confinement*: keeping private information about a process from leaking to processes it influences in the BPO
- *backward confinement*: keeping private information about a process from leaking to processes that have influenced it

Enforcing principles of forward and backward information confinement raises some interesting implementation challenges. Let a be a send event at process P_1, and let events b at P_2 and c at P_3 be in the future of a (that is, $a \longrightarrow b$ and $a \longrightarrow c$) and suppose processes P_2 and P_3 need to know different details of the history of a in order to timestamp b and c, respectively. Forward confinement requires that P_1 not transmit this information with a. But backward confinement requires that P_2 and P_3 cannot just query P_1!

Confinement by structure. Confinement principles are just that—principles we impose for reasons external to the basic problem of tracing causality. However, some common system mechanisms create information barriers that fundamentally affect this basic problem. Suppose student Alice sends an anonymous suggestion to the suggestion box maintained by Professor Bob for his class, who acts on this suggestion. Bob's actions depend on Alice's

suggestion—but he cannot know whose action this suggestion is. Further, the suggestion is not completely anonymous, for in her later interactions with Bob, Alice knows that Bob's actions follow from her actions. Greif [7] calls this the phenomenon of hidden causality, and gives a more fundamental example: the relation between V and P operations on a binary semaphore.

How to resolve the problem of hidden causality in a distributed partial clock is another research issue we intend to explore. We may need to extend the BPO formalism to make it sufficiently rich to express all these nuances.

4. The SVT Protocol

4.1. Overview

The central issue in building a secure distributed partial clock toolkit is how to keep track of the partial order. Essentially, our BPO is a dynamically changing directed acyclic graph whose behavior meets the following criteria:

- **Monotonicity.** As [real] time progresses, edges and nodes are added. In the basic problem, nothing is deleted.
- **Distribution.** New nodes originate from individual processes within a distributed system; new edges from either individual processes or (in the case of message transmission) from pairs of processes.

Our toolkit needs to allow individual processes to answer connectivity queries about this graph, and hence must maintain this graph, at least in some virtual form. The distributed nature of the DAG forces processes to require nonlocal information in order to answer these queries. The issue of how and when this information should propagate—piggybacked on system messages, or transmitted only when requested by a query—delineates one axis of possible implementation approaches.

In this section we outline a starting point for our implementation work: *signed vector timestamps* (SVTs). This approach falls at the "piggyback" end of this axis. The SVT protocol extends Lamport event counters to provide an implementation of distributed partial clocks that is moderately robust against Byzantine failure. We conjecture that this may be the best protection possible if we disallow any special underlying computational structure.

However, this initial approach offers two principal drawbacks: ineffectiveness at enforcing forward confinement, and computational inefficiency in certain scenarios. Analyzing these drawbacks suggests several new directions for implementation research.

We begin by discussing Lamport clocks (Section 4.2), then extend them to vectors (Section 4.3), and then turn to SVTs: the protocol, its problems, and the new research avenues suggested (Sections 4.4 and 4.5).

4.2. Lamport Clocks

Lamport [11] discusses the issue of determining the BPO and presents an elegant partial solution using local event counters. Timestamps sent along with every message keep the local counters roughly synchronized, and capture a total order[4] consistent with the BPO.

Formally, each process P_i maintains a local scalar clock C_i. Process P_i marks each event a that occurs there and each message m it sends with a timestamp $C(a)$ (or $C(m)$), which reflects the current value of the clock C_i. This current value changes with each event a at P_i; the type of event determines the change.

a is internal	$C(a) \leftarrow C_i$ $C_i \leftarrow C_i + 1$
a is sending of message m	$C(a) \leftarrow C_i$ $C(m) \leftarrow C_i$ $C_i \leftarrow C_i + 1$
a is reception of m	$C_i \leftarrow \max\{C_i, C(m) + 1\}$ $C(a) \leftarrow C_i$ $C_i \leftarrow C_i + 1$

These timestamps order events consistently with the BPO:

Theorem 1 *For all events a, b, if $a \longrightarrow b$ then $C(a) < C(b)$.*

However, this method has two principal drawbacks—it only produces a total order (the converse to Theorem 1 does not hold), and it is egregiously unsecure, as each process's clock is essentially world-writable. For example, suppose process P_i has $C_i = s$ and receives a message m from process P_j with $C(m) = t \gg s$. Ostensibly, the timestamp t testifies that at least $t - s$ events have occurred in the outside world since P_i last received a message. But P_i cannot distinguish this presumed scenario from one where malicious process[5] P_j arbitrarily inflates the timestamp. After all, such maliciousness offers advantages:[6]

- If P_i lacks a "sensibility check" on its timestamps but plans to interact with process P_k that does, then P_j's action causes P_k to erroneously identify P_i as faulty.
- If processes store timestamps as a fixed-length word with maximum value N, then P_j could use $t = N - 1$ and cause P_i to roll over, either making P_i appear faulty or causing dangerous anachronism.

[4] Strictly speaking, it produces a partial order, as events at two processes could receive the same value timestamp. But we can easily linearize this order by choosing a linear order on the processes and using that order to break ties.

[5] We oversimplify here—consider that P_j itself may only be the last link in a chain of honest processes unwittingly passing on bogus information introduced by the malicious process.

[6] Again, actual scenarios may be even more complex: P_i may be just a link in a chain to reach the intended victim process.

- If processes store timestamps as unbounded values, P_j could still increase by orders of magnitude the number of words P_i uses for its clock. This both slows down P_i's dealings with its neighbors, and allows P_j to observe the spread of its influence—a violation of backward confinement.
- If P_j interacts with most processes fairly regularly, then it can render the entire clock system effectively useless by blowing up every timestamp with each message.

4.3. Extending Lamport Clocks to Vectors

Our SVT implementation extends Lamport counters by making timestamps *vectors* instead of *scalars*, and incorporating *digital signatures.* These extensions rectify the cited drawbacks.

In the *vector timestamp* protocol, processes maintain a vector indicating their "knowledge horizon"—the most recent event they (syntactically) know about at each other process. (Technically, we should note that this structure is not so much a vector but an indexed set: the length need not be fixed, nor the indices known *a priori.* This raises some interesting research questions regarding what to do with lost or missing members.) The SVT protocol extends this by using public key decryption to authenticate these timestamps.

The vector timestamp protocol exactly captures the BPO. The SVT protocol even allows the set of honest processes—no matter how few—to calculate the nontransitive restriction of the BPO despite any action whatsoever by malicious processes. The concept of using *dependency vectors* without authentication surfaces in earlier research (e.g., [23], [15], [5], [6]), but this paper is the first to consider these vectors as an implementation for a general purpose, *secure* partial clock toolkit.

The remainder of this section presents the basic protocol, and Sections 4.4 and 4.5 add authentication.

4.3.1. The Vector Timestamp Protocol

We begin by discussing the basic protocol, without authentication. Let n be the number of processes. Each process P_i maintains a local clock C_i, an event counter. Each process also maintains an n-element vector V_i to keep track of the most recent event it knows about at every other process. We will use the notation $V_i(j)$ to refer to the jth component of vector V_i—this component reflects process P_i's most current knowledge of process P_j. We can dispense with C_i altogether, and just store the value as $V_i(i)$. Let each component of each V_i be zero initially.

Each process will timestamp its events and outgoing messages with an n-element vector. To follow our previous notation strictly, we should denote these timestamps by $V(a)$; however, to make component indexing easier, we will use subscripting instead: V_a is the timestamp on event a, V_m on message m. The following table outlines how processes obtain these timestamp vectors and update their own vectors. Let event a occur on process P_i.

a is internal	$V_i(i) \leftarrow V_i(i) + 1$ $V_a \leftarrow V_i$
a is sending of message m	$V_i(i) \leftarrow V_i(i) + 1$ $V_a \leftarrow V_i$ $V_i(i) \leftarrow V_i(i) + 1$ $V_m \leftarrow V_i$
a is reception of m	$\forall j \neq i \;\; V_i(j) \leftarrow \max\{V_i(j), V_m(j)\}$ $V_i(i) \leftarrow V_i(i) + 1$ $V_a \leftarrow V_i$

The reason for the two increments in send events may not be intuitively clear. We increment the local component before sending a message so that the receiving process can treat all components equally when maximizing. We increment again so that the subsequent event at the sending process will not precede the receive event.

We define a natural ordering on the timestamp vectors.

Definition 2 *For vectors V, W, we say that $V \prec W$ when $\forall i \; V(i) \leq W(i)$ and $\exists i \; V(i) < W(i)$.*

This ordering exactly captures the BPO.

Theorem 3 *For all events a, b, $a \longrightarrow b$ iff $V_a \prec V_b$.*

4.3.2. Security Problems

Consider the timestamp vector V_i on process P_i. It is true that the components $V_i(j)$ are world-writable (for $i \neq j$) in the sense that a party sending P_i a message can force these components arbitrarily high. If P_k has $V_k(j) = 42$ (for k, j, i distinct), then P_k can send a message to P_i and know that afterward, $V_i(j) \geq 42$. If P_k is malicious, then it can render the vector V_i effectively useless.

But assume for the moment that everyone is honest. Let 0 be the initial value of all vector components. Process P_k can change a component of its vector in only two ways: it can increment its own component

$$V_k(k) \leftarrow V_k(k) + 1$$

or it can copy other components from incoming messages

$$V_k(j) \leftarrow \max\{V_m(j), V_k(j)\}$$

The vector on a message is just a copy of the vector at the sending process. Hence we can observe:

Theorem 4 *Let a be an event on process P_i, and let $j \neq i$. Then $V_a(j)$ is either 0 or is a copy of $V_m(j)$, where m is a message sent by event b at process P_j, and $b \longrightarrow a$.*

So processes now have some means of detecting when someone is sending them bogus information in a message's timestamp: they know that each nonzero component j of the timestamp should have been originally generated by process P_j.

4.4. SVT: Adding Signatures to the Vectors

By adding signatures to the vector timestamp scheme, we can add tolerance against Byzantine faults—arbitrary behavior by arbitrary numbers of processes.

Let us assume a public key decryption scheme, where for any x each process P_i can generate a signature $\mathcal{E}_i(x)$ such that

- any process P_j can, given x,i, and y, quickly determine whether $y = \mathcal{E}_i(x)$
- for $j \neq i$, any finite set X, and any $x \notin X$, no process P_j can calculate $\mathcal{E}_i(x)$, even if it has an oracle for $\mathcal{E}_i$ on X.

We directly extend the basic vector timestamp protocol to produce the secure protocol SVT. Namely, we just include and check signatures.

Every vector V will now have two fields in each component—the actual value $V(i)$, and the signature $V(i)'$. When a process P_i sends a message m, it sets

$$V_i(i)' \leftarrow \mathcal{E}_i(\ V_i(i)\ )$$

and then assigns $V_m \leftarrow V_i$. When a process P_i receives a message m, it first checks the signatures

$$\forall j \quad V_i(j)' = \mathcal{E}_j(\ V_i(j)\ )$$

before accepting it. If P_i decides to copy a component from the incoming message

$$V_i(j) \leftarrow V_m(j)$$

then P_i copies the signature as well

$$V_i(j)' \leftarrow V_m(j)'$$

Let H be the set of honest processes. The SVT protocol allows honest processes to correctly calculate the nontransitive restriction of the BPO.

Theorem 5 *If $a \xrightarrow{H} b$ then $V_a \prec V_b$.*

In the other direction, we can show something a bit stronger.[7]

Theorem 6 *Let events a and b occur at processes P_i and P_j. Let $i \in H$, and let V_b have proper signatures. If $V_a \prec V_b$ then $a \longrightarrow b$.*

A nice thing to observe about SVT is that honest processes do not need to know which other processes are honest.

4.5. Problems with the SVT Implementation

The SVT protocol has several drawbacks. For one thing, its tolerance of Byzantine failure is not ideal—the "reasonable performance" it achieves falls short of what we would have desired. We suspect that this behavior may be inherent for this style of implementation. Another problem is that the amount of information that SVT timestamps contain violates forward confinement and, in certain situations, might be rather inefficient.

4.5.1. Lost Influence

In Section 3 we state that a central goal of this work is to discover a protocol by which an honest process P_i can determine the BPO among its perceivable events E_i. The SVT protocol does not achieve this goal. It is true that in SVT, a malicious process cannot overwrite the clock values of other processes, and cannot generate arbitrarily large values in timestamp components corresponding to honest processes. However, the protocol does permit spoofing (in the sense of Herlihy and Tygar [9]). During the course of system operation, a process will receive many timestamp pairs $x, \mathcal{E}_i(x)$ for many of the i. A process is supposed to use the largest x it has received in each component, but it can use any other one it wants to.

For example, suppose Alice and the Bank are honest, but Carla is pretty nasty. Suppose Alice deposits \$10 in her previously empty bank account, and then gives Carla a check for \$10. Carla can roll back all her timestamps and quickly cash the check—and the Bank would believe that Alice's request depends on Carla's, and thus will execute Carla's first, getting Alice into trouble.

The problem remains that any dealings with dishonest or faulty processes will be suspect. We conjecture[8] that this behavior is inherent for a large family of implementations: any protocol built around the following assumptions will risk losing chains of influence through malicious processes.

---

[7] Actually, the question of whether Theorem 6 is stronger than the converse of Theorem 5 is not answered so easily: we could interpret proving the latter as being able to distinguish $a \xrightarrow{H} b$ from $a \longrightarrow b$, which broaches the awkward topic of honest processes identifying the dishonest ones. Research questions remain here.

[8] Since the preparation of the original document in 1991, we have formalized and proved this conjecture. The proof will appear in the first author's thesis. [21]

- the processes themselves do all the computation—nothing is hidden or unconscious
- no honest process has a right to know anything about the internal events of any other process

4.5.2. Confinement and Efficiency

Since SVT timestamps are real data packets which entirely determine event ordering, the SVT implementation easily enforces backward information confinement. A process examining a timestamp does not need to bother anyone else. However, a cursory inspection of the protocol reveals a a fundamental violation of forward confinement: the fact that processes must pass on the most recent timestamp components from everyone in the system.

If the distribution of messages is fairly uniform, then SVT is reasonably efficient. But the real world contains highly non-uniform scenarios. For example, consider a system consisting of clusters of workstations at various universities. Most of the communication takes place within each cluster, so the system graph has two fairly densely connected components, with only a few edge between the components. If we have n processes and only $\delta \ll n$ messages across this cut, then we're transmitting much extra data—$\Omega(\delta n)$ when we really only need $O(\delta^2)$.

One can argue similarly that much of the timestamp information in a tightly coupled cluster is irrelevant, as everyone knows everything already. This situation is troublesome because of redundant data, rather than unnecessary data. Some fairly straightforward methods exist to reduce this waste—consider that process P_1 can obtain from the timestamps it exchanges with process P_2 a good lower bound for each component in P_2's internal vector, and only needs to transmit the components that exceed this bound.[9]

5. Future Work

The traditional way to regard time is as a linear order on the events in a system—for any pair of distinct events e_1, e_2, one must have happened before the other. By deliberately leaving unordered events that did not influence each other, the BPO opens the door for more general classes of temporal orderings.

Besides being of theoretical interest (e.g., Pratt [17]), these alternative time models have some exciting implications for asynchronous distributed systems. Partial orders in form or another lie at the heart of many application problems.[10] For example:

- **Tracking concurrency.** In terms of the partial orders, the *distributed snapshot* prob-

[9] After the 1991 document, we discovered that Singhal and Kshemkalyani [19] had previously examined some optimization techniques for vector timestamp protocols.

[10] E.g., [1],[2], [3],[4], [8], [10], [16]. See [20] or [21] for a more thorough overview.

lem reduces to finding a maximal set of mutually concurrent events.

- **Tracking forward influence.** The problem of *rollback* requires determining the future of an event: if event e_1 is to be undone, then all events e_2 with $e_1 \longrightarrow e_2$ must be undone. Protocols based on linear time orders only detect a superset of what e_1 influenced; protocols based on partial orders give the set exactly.

- **Tracking reverse influence.** The problem of *orphan detection* requires determining, given event e_1, if any aborts preceded it. Protocols based on linear time orders only detect a superset of what influenced e_1; protocols based on partial orders give the set exactly.

In his thesis proposal [20], the first author argues that solving such application problems requires first solving the problem of maintain partial order information, and hence these solutions to these application problems will automatically inherit the security problems of partial order clocks. Hence developing a theory of partial order time and encapsulating its clock primitives and security issues into a single package will provide a framework for building secure protocols for these general application problems. Forthcoming publications will expand on this research.

6. References

1. Birman and Joseph. "Exploiting Virtual Synchrony in Distributed Systems." *Eleventh Symposium on Operating Systems Principles.* 123-138. 1987.

2. Birman and Joseph. "Reliable Communication in the Presence of Failures." *ACM Transactions on Computer Systems,* 5: 47-76. February 1987.

3. Chandy. *The Essence of Distributed Snapshots.* Caltech CS TR 89-5. March 1989.

4. Chandy and Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems." *ACM Transactions on Computer Systems.* 3: 63-75. February 1985.

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

6. Fidge. "Logical Time in Distributed Computing Systems." *IEEE Computer.* 24 (8):28-33. August 1991.

7. Greif. *Semantics of Communicating Parallel Processes.* Ph.D. thesis, MIT, 1975.

8. Herlihy, Lynch, Merritt and Weihl. *On the Correctness of Orphan Elimination Algorithms.* MIT LCS TM-329. 1987.

9. Herlihy and Tygar. *How to Make Replicated Data Secure.* CMU-CS-87-143. August 1987.

10. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing.* Ph.D. thesis, Rice University, 1989.

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

12. Lamport and Melliar-Smith. "Byzantine Clock Synchronization." *Third ACM Symposium on Principles of Distributed Computing.* 1984.

13. Marzullo and Owicki. "Maintaining the Time in a Distributed System." *Second ACM Symposium on Principles of Distributed Computing.* 1983.

14. Mattern. "Algorithms for Distributed Termination Detection." *Distributed Computing.* 2: 161-175. 1987.

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

16. Peterson, Bucholz and Schlichting. "Preserving and Using Context Information in Interprocess Communication." *ACM Transactions on Computer Systems.* 7: 217-246. August 1989.

17. Pratt. "Modeling Concurrency with Partial Orders." *International Journal of Parallel Programming.* 15 (1): 33-71. 1986.

18. Reiter and Gong. "Preventing Denial and Forgery of Causal Relationships in Distributed Systems." *1993 IEEE Symposium on Research in Security and Privacy.* (To appear.)

19. Singhal and Kshemkalyani. *An Efficient Implementation of Vector Clocks.* Ohio State TR OSU-CISRC-11/90-TR34. November 1990.

20. Smith. *Secure Clocks for Partial Order Time.* Thesis proposal, School of Computer Science, Carnegie Mellon University. October 30, 1991.

21. Smith. *Secure Clocks For Partial Order Time.* Ph.D. thesis, School of Computer Science, Carnegie Mellon University. (In preparation.)

22. Srikanth and Toueg. "Optimal Clock Synchronization." *Journal of the ACM.* 34 (3): 626-645. July 1987.

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