Virtual Time

DAVID R. JEFFERSON
University of Southern California

Virtual time is a new paradigm for organizing and synchronizing distributed systems which can be applied to such problems as distributed discrete event simulation and distributed database concurrency control. Virtual time provides a flexible abstraction of real time in much the same way that virtual memory provides an abstraction of real memory. It is implemented using the Time Warp mechanism, a synchronization protocol distinguished by its reliance on lookahead-rollback, and by its implementation of rollback via antimessages.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism*; D.4.1 [**Operating Systems**]: Process Management—*synchronization*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*

General Terms: Performance, Reliability, Theory, Verification

Additional Key Words and Phrases: Concurrency control, simulation, Time Warp

1. INTRODUCTION

In this paper we propose a new paradigm for distributed computation, called *virtual time*, and an implementation for it, called the *Time Warp mechanism*. The virtual time paradigm is a method of organizing distributed systems by imposing on them a temporal coordinate system more computationally meaningful than real time, and defining all user-visible notions of synchronization and timing in terms of it. The Time Warp mechanism implements virtual time, and does so in a manner that is strongly analogous to the way that paging or segmentation mechanisms implement virtual memory. Thus, the well-developed theory and terminology of virtual memory acts as a guide to understanding virtual time. Two major applications for virtual time are as a synchronization mechanism for distributed simulation and as a concurrency control mechanism for databases.

Most distributed systems (including all those based on locks, semaphores, monitors, mailboxes, rendezvous, etc., and the usual mechanisms of flow control) use some kind of *block-resume* mechanism to keep processes synchronized. In addition, some systems divide a computation into discrete atomic actions, called transactions, and allow *abortion-retry*, essentially a form of rollback where rollback is only possible to the beginning of the current transaction. In contrast, the distinguishing feature of Time Warp is that it relies on general *lookahead-rollback* as its fundamental synchronization mechanism. Each process executes

Author's current address: Dept. of Computer Science, UCLA, Los Angeles, CA 90024.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1985 ACM 0164-0925-185/0700-0404 $00.75

without regard to whether there are synchronization conflicts with other processes. Whenever a conflict is discovered after the fact, the offending process(es) are rolled back to the time just before the conflict, no matter how far back that is, and then executed forward again along a revised path. Both the detection of synchronization conflicts and the rollback mechanism for resolving them are entirely transparent to the user. We are not, in this paper, recommending rollback as a programming tool for explicit use, though that may be reasonable to consider elsewhere.

General rollback has traditionally been used for fault tolerance, especially in sequential environments; but as far as we know it has never been used for synchronization purposes. This is perhaps because at first glance distributed rollback seems almost hopelessly complex to implement (consider its interactions with I/O, run-time error handling, creation and destruction of objects, etc.), and because even if implementable, it would seem expensive at run time compared to alternative synchronization mechanisms. To justify the value of virtual time and Time Warp, we argue later that (1) distributed rollback has a natural and elegant implementation; (2) whenever rollback occurs, other rollback-free implementations would require blocking for an amount of real time equal to that spent on wasted computation; and (3) that rollback will usually occur relatively infrequently. Argument (3) rests on temporal locality assumptions about the dynamic behavior of programs that are analogous to the spatial locality assumptions underlying virtual memory systems. These assumptions have yet to be tested empirically.

In the next section we describe the concept of virtual time and the semantics of virtual time systems. In Section 3 we compare virtual time to the work of other theorists in the field of distributed systems. Section 4 describes the Time Warp mechanism, both its local and its global parts. In Section 5 we give three examples of paradigms that can be seen in a unified framework when viewed as virtual time systems. Finally, Section 6 gives the extended comparison between virtual time and virtual memory that has been a central focus of this research.

2. VIRTUAL TIME

A *virtual time system* is a distributed system executing in coordination with an imaginary *virtual clock* that ticks *virtual time*. Virtual time itself is a global, one-dimensional, temporal coordinate system imposed on a distributed computation; it is used to measure computational progress and to define synchronization. It may or may not have a connection with real time. We assume that virtual times are real values (with a positive infinite value *+inf*), totally ordered as usual by the relation $<$. From a programmer's semantic point of view, the global virtual clock always progresses forward (or at least never backward) at an unpredictable rate with respect to real time. But, from the implementer's point of view, there are many local virtual clocks, loosely synchronized, and while all of the virtual clocks tend to go forward toward higher virtual times, they occasionally jump backward.

We envision systems of many (maybe thousands) of small processes all executing concurrently on a computer with many processors. It is useful to consider each process as occupying a *point* in *virtual space*, with its unique name acting

as its spatial coordinate. Every primitive action (changing a variable, sending a message, etc.) thus has both a virtual time coordinate and a virtual space coordinate, and the set of all actions (side effects) that take place at the same virtual place x and virtual time t is referred to as the *event* at (x, t).

Processes communicate only by exchanging messages, and any process is free at any time to send a message to any process (including itself) that it can name. There is no concept of a *channel* between two processes, and no need for any open/close protocol.

All messages are stamped with four values: the name of the *sender*, the *virtual send time*, the name of the *receiver*, and the *virtual receive time*. The virtual send time is the virtual time at the moment the message is sent. Likewise, the virtual receive time of a message is the virtual time when the message must be received. We also say, equivalently, that a message is stamped with the coordinates of the sending and receiving events. A message is thus the transfer of information from one event (point) in virtual space-time to another, like a photon in physics. All four coordinate stamps are considered to be part of the information in the message, and may thus be read by the receiver.

Virtual time systems are subject to two fundamental semantic rules:

Rule 1. The virtual send time of each message must be less than its virtual receive time.

Rule 2. The virtual time of each event in a process must be less than the virtual time of the next event at that process.

These rules are exactly Lamport's Clock Conditions [17], and embody our desire that the arrow of causality, or the direction of information transfer, always be pointed in the direction of increasing virtual time. They imply that all messages output from any one process are sent in order of virtual send time (but not necessarily virtual receive time), and that all messages input to any one process are read in order of virtual receive time (but not necessarily virtual send time). Although dynamic creation and destruction of processes is perfectly compatible with virtual time, we exclude it from consideration in this paper because it would force a lengthy digression.

An event at (x, t) is an ordinary deterministic sequential computation conducted entirely within process x, and involves zero or more of the following operations:

(1) x may receive any number of messages stamped with receiver x and virtual receive time t, and read their contents;
(2) it may read its virtual clock;
(3) it may update its state variables;
(4) it may send any number of messages, all of which will automatically be stamped with sender x and virtual send time t.

We say that event A *causes* event B (or is one of the causes) if there exists any sequence of events $A = E_0, E_1, \ldots, E_n = B$ such that for each pair E_i and E_{i+1} of adjacent events either (a) E_i and E_{i+1} are both in the same process (at the same virtual place) and the virtual time of E_i is less than that of E_{i+1}, or (b) event E_i sends a message to be received at event E_{i+1}. Note that this is identical

to the relation *happens before* in [17]. With this relation in mind, the major constraint on the implementation of virtual time can be stated in extremely simple terms:

> *If an event A causes event B, then the execution of A and B must be scheduled in real time so that A is completed before B starts.*

Note that even though event A has an earlier virtual time than event B, we need not actually perform A before B if there is no causal chain from A to B. We may achieve better performance by scheduling A concurrently with B, or even after it. If A and B have exactly the same virtual time coordinate, then there is no restriction on their scheduling, and they may be arbitrarily interleaved. If they are distinct events, they will have different virtual space coordinates (i.e., will occur in different processes), and neither will be a cause of the other, so any interleaving will be undetectable by any test that the programmer can perform. Two events at the same virtual time thus act as parallel components of a single atomic operation, indivisible with respect to events at other virtual times. All events at virtual times earlier than t appear operationally to complete before events at time t start, and all events at later virtual times appear to start only after events at time t are complete. No process can, by any computation or exchange of messages, deduce otherwise.

Virtual time systems are not all isomorphic; they may vary in any of several dimensions. For example, the virtual time scale may be either discrete or continuous (although here we assume continuous, i.e., that the virtual time coordinates are true real numbers, not discrete approximations). Virtual times may be only partially ordered (though we assume here that they are totally ordered). Virtual time may be related to real time or be independent of it. (We give examples of both.) Virtual times may be visible to programmers and manipulated explicitly as values, or they may be hidden and manipulated implicitly according to some system-defined discipline. (The same choices exist for process names and virtual memory addresses, the spatial counterparts of virtual times.) In addition, virtual time systems differ in how virtual times are assigned to messages. For example, virtual times associated with events may be explicitly calculated by user programs or they may be assigned by fixed default rules. (Again, we give examples of both.) Although each set of choices defines a different kind of virtual time system, all are similar enough that a unified approach to the theory and implementation is appropriate.

3. COMPARISON TO OTHER WORK INVOLVING ARTIFICIAL TIME SCALES

Recently, there have been a number of proposals published for synchronizing distributed systems using artificial time scales. We now briefly contrast three of them with the virtual time paradigm.

3.1 Lamport's Work

Lamport [17] seems to have been among the first to recognize that real-time temporal order, simultaneity, and causality between events in a distributed system bear a strong resemblance to the same concepts in special relativity. In

particular, he showed that the real-time temporal relationships *happens before* and *happens after* that are operationally definable within a distributed system form only a partial order instead of a total order, and that distinct concurrent events are incomparable under that partial order. He further showed that it is always possible to effectively extend this partial order to a total order by defining a system of artificial clocks, one clock for each process, that labels each event with a unique value from a totally ordered set in a manner consistent with the partial order. The important point for our purposes is that he gives an algorithm for accomplishing this, starting from a particular execution of a distributed system and yielding an assignment of totally ordered clock values to the events of that execution.

With virtual time we do the reverse: we *assume* that every event is labeled with a clock value from a totally ordered virtual time scale in a manner obeying Lamport's Clock Conditions (Rules 1 and 2), and we show how to unfold a fast concurrent execution (i.e., a wide and shallow partial ordering that is consistent with the total ordering). The Time Warp mechanism (which we describe in Section 4) is thus an inverse of Lamport's algorithm.

One of the virtues of adopting the virtual time paradigm is that we can reason correctly about the relations *before* and *after* in virtual time, using ordinary Newtonian intuition. The more difficult "relativistic" reasoning that Lamport shows is necessary to understand *before* and *after* in real time is unnecessary for virtual time. In a very practical sense virtual time is easier to understand and reason about than real time.

3.2 Reed's Work

In his study of distributed systems [24], Reed invented the notion of *pseudotime*, which bears a strong resemblance to virtual time, but has a very different motivation and implementation. Reed is primarily interested in implementing distributed atomic actions, and his work has been used as the basis for the design of multiversion timestamp order mechanisms for concurrency control in distributed databases.

With virtual time, atomic actions are not the focus; instead, the relations *before* and *after* are primary. For the same reasons that we can define equality in terms of "greater" and "less," but not vice-versa, we can define atomicity in terms of before and after, but not the other way around.

This difference can be illustrated by the following example. Two actions occurring at different pseudotimes may be committed in either order. Reed's mechanism manages execution so that the action with the earlier pseudotime will probably be committed earlier, but this is only a heuristic. With bad luck or bad timing the action with the earlier pseudotime may have to be aborted and retried later with a later pseudotime after the other action is complete. But with virtual time there is no possibility of a retry with a new timestamp; the virtual times of events are *fixed*, and all events *must* appear to be executed in virtual time order. We can express this contrast as follows: by assigning pseudotimes we cannot with certainty specify the order in which two conflicting atomic actions are to be executed, even if we want to; but by assigning virtual times to two events we always specify their order if they conflict, even if we do not want to.

One other difference is worth noting. Because Reed's mechanism uses abortion and retry for synchronization, and there is no limit to the number of times an action may have to be retried, starvation is an issue. There is no deadlock, starvation, or any other corresponding hazard when virtual time is implemented by the Time Warp mechanism.

3.3 Schneider's Work

Schneider has done a more general study of synchronization [27], in which he presents a mechanism for implementing essentially any synchronization protocol in a distributed environment. His technique is based on broadcasting all synchronization-related messages ("phase-transition messages") to every process in the system, with every process in turn broadcasting its acknowledgment to all other processes, thus making all processes aware of every synchronization action in the system. All such messages and acknowledgments are timestamped with values from a valid clock system such a Lamport's, so that all processes agree both on what synchronization messages have been broadcast and on the logical order in which they were broadcast. Broadcasting all synchronization messages and acknowledgments is logically equivalent to keeping synchronization information in a globally accessible shared memory.

Schneider shows, under the assumption of reliable, order-preserving message delivery, that each process may make synchronization decisions locally (e.g., whether to proceed or to block) based on the set of "fully acknowledged" messages it has received. A message is considered fully acknowledged at process P if P has received it and also copies of the acknowledgments to it from every other process in the system. The importance of recognizing when a message m is fully acknowledged is that the receiver is then guaranteed that it will never again receive a message or acknowledgment with a timestamp earlier than that of m.

Schneider's mechanism is similar to the Time Warp mechanism in that it *does* assign global temporal coordinates to *some* of the actions in a distributed system, namely the synchronization actions. But where the Time Warp mechanism is extremely optimistic, making synchronization decisions on a provisional basis and rolling back when they turn out to be wrong, Schneider's mechanism is extremely conservative, waiting to make such decisions until such time as it can be proved that they cannot be wrong.[1] One disadvantage of Schneider's mechanism is that it seems to be limited to systems with only a few processes; it does not scale upward smoothly to thousands of processes because of the prohibitive amount of message and acknowledgment processing inherent in it. The Time Warp mechanism seems to have no such barrier to scale-up to thousands of processes.

4. THE TIME WARP MECHANISM

From now on we refer to the virtual receive time of a message as its *timestamp*. For correct implementation of virtual time, it is necessary and sufficient that *at each process* messages are handled in timestamp order. It is generally undesirable for an implementation to require that all processes progress through virtual time

[1] This observation is due to J. C. Brown.

at the same rate with respect to real time, since that would essentially sequentialize the execution of the entire system. At any given moment some processes are allowed to be ahead in virtual time according to their local virtual clocks while others lag behind.

It is not obvious how incoming messages at each process can be processed in timestamp order because they will not generally arrive in timestamp order, and, since we assume virtual times are real numbers, it is impossible for a process, on the basis of local information alone, to block and wait for the message with the "next" timestamp. No matter which one is presumed to be "next" it is always possible that another message with an earlier timestamp will arrive later. Thus, even when the message with the "next" timestamp does arrive, it cannot be recognized as such. This is the central problem in implementing virtual time that is solved by the Time Warp mechanism.

The Time Warp mechanism is defined without reference to any underlying computer architecture, and can run efficiently on many multiple processor systems, from tightly coupled multiprocessors such as C.mmp [31], Cm* [30], or the N.Y.U. Ultracomputer [28], to a medium-coupled system with no shared memory such as the Caltech Hypercube [5, 10], to a local area network connected by Ethernet [19]. We assume that message communication is reliable, but we do not assume that messages are delivered in the order sent; in fact, such a protocol would be wasteful because messages are not generally processed exactly in sending order.

Time Warp [12–14, 29] has two major parts, the *local control mechanism*, concerned with making sure that events are executed and messages received in correct order (providing a "weakly correct" implementation of virtual time), and the *global control mechanism*, concerned with global issues such as space management, flow control, I/O, error handling, and termination detection (all contributing to its "strong correctness"). We discuss these in turn.

4.1 The Local Control Mechanism

Although abstractly there is a single global standard of virtual time, there is no global virtual clock variable in the implementation; instead, each process has its own *local virtual clock* variable. The local virtual clock of a process does not change during an event at that process; it changes only between events, and then only to the value in the timestamp of the next message from the input queue. At any moment some local virtual clocks will be ahead of others, but this fact is invisible to the processes themselves because they can read only their own virtual clock. Whenever a message is sent, its virtual send time is copied from the sender's virtual clock. The receiver and virtual receive time fields may be assigned by any one of a variety of application-specific conventions to be discussed later.

Each process has a single input queue in which all arriving messages are stored in order of increasing virtual receive time. Ideally, the execution of a process is simply a cycle in which it receives messages and executes events in increasing virtual time order. This ideal execution proceeds as long as no message ever arrives with a virtual receive time in the "past." But this is bound to happen occasionally because of variation in the computation rates of different processes or because of transmission delays in the network. Whatever the reasons for the

late arrival of a message with a low timestamp, the semantics of virtual time demands that incoming messages be received by each process strictly in timestamp order. The only way to accomplish this is for the receiver to roll back to an earlier virtual time, canceling all intermediate side effects, and then to execute forward again, this time receiving the late message in its proper sequence. Whenever a process has processed all input messages in its input queue, its virtual clock is set to *+inf*, and the process is said, by convention, to *terminate.* However, it is not destroyed because the arrival of a new message later may cause it to roll back and *unterminate.*

Because it is impossible to wait for the "next" message, each process executes continuously, processing in increasing virtual receive time order those messages that have already arrived. All of its execution is provisional, however, because it is constantly gambling that no message will arrive with a virtual receive time less than the one stamped on the message it is now processing. As long as it wins this bet, execution proceeds smoothly. But whenever the bet is lost the process pays a performance penalty by rolling back to the virtual time when it "should" have received the late message. The situation is quite similar to the gamble that paging mechanisms take in the implementation of virtual memory: they are constantly betting with every memory reference that no page fault will occur. Execution is smooth as long as the bet is won, but comparatively expensive disk or drum operations are necessary when it is lost.

We might describe the situation differently by saying that each process is constantly doing a "lookahead," processing "future" messages from its input queue. In any kind of lookahead scheme there are certain comparatively infrequent contingencies that require the undoing of some work already accomplished, and in this case the contingency is the arrival of a late message with an early timestamp. But lookahead generally is successful when the overhead of occasional undoing is outweighed by improved performance the rest of the time.[2]

The name "Time Warp" derives from the fact that the virtual clocks of different processes need not agree, and the fact that they go both forward and backward in time. Over a lengthy computation, each process may roll back many times while generally progressing forward. The fact that virtual clocks are sometimes set back does not violate our stated intention that "the global virtual clock always progresses forward (or at least never backward)" because rollback is completely transparent to the process being rolled back. Programmers can write correct software without paying any attention to late-arriving messages, and even with no knowledge of the possiblity of rollback, just as they can write without any attention to, or knowledge of, the possibility of page faults in a virtual memory system.

In many practical situations there will be more processes than processors, so only a subset of the processes can run at any one moment. The natural scheduling rule is always to execute those processes whose local virtual clocks are farthest behind. On a shared-memory multiprocessor, this means always running the n farthest behind processes, where n is the number of processors in use. On a network, it means always running, on each processor, the farthest behind process

[2] I am indebted to Tim Standish at the University of California at Irvine for this characterization.

residing on that processor. From now on we assume that one of these scheduling rules is in effect, and that on networks there is no dynamic reassignment of processes to processors.

Rollback in a distributed environment is complicated by the fact that the process in question may have sent any number of messages to other processes, causing side effects in them and leading them to send still more messages to still more processes, and so on. Some of those messages may have requested output or some other irreversible action (dispense money, launch missile). Some of them may be physically in transit and therefore out of the system's control for arbitrary durations. The paths followed by these direct and indirect messages from process to process may not form a tree, but may converge or even cycle, leading to worries about infinite loops or deadlock. Nevertheless, all such messages, direct or indirect, in transit or not, causing I/O or not, must be effectively "unsent" and their side effects, if any, reversed. The Time Warp rollback mechanism is able to accomplish all this quite efficiently, and without stopping any part of the system.

4.2 Antimessages and the Rollback Mechanism

To understand the rollback mechanism, we must describe more of the structure of processes and messages. In Figure 1 we see the structure of a process named *A*. The runtime representation of a process is composed of

(1) A *process name* (virtual space coordinate), which must be unique in the system.
(2) A *local virtual clock* (virtual time coordinate), which in the figure reads 162, indicating that the message with virtual receive time 162 is being processed.
(3) A *state*, which in general is the entire data space of the process, including its execution stack, its *own* variables, and its program counter. We show here only two *own* variables to represent the whole state.
(4) A *state queue*, containing saved copies of the process's recent states. In order to make rollback possible, the Time Warp mechanism must, from time to time, save the state of a process. We assume for simplicity that this is done after every event at each object. It is possible to reduce or increase this frequency, and a good strategy is to save a process's state every time it uses *n* seconds of processor time. As we shall see when we discuss the global control mechanism, it is not necessary to retain states from all the way back to the beginning of virtual time, but there must be at least one saved state that is older than *global virtual time* (GVT).
(5) An *input queue*, containing all recently arrived messages sorted in order of virtual receive time. Some of these messages have already been processed because their virtual receive times are less than 162. Nevertheless, they are not deleted from the queue because it may be necessary to roll back and reprocess them. Other messages with virtual receive times greater than 162 have not yet been processed, or else they have been processed and "unprocessed" an equal number of times. Only incoming messages whose virtual send times are greater than or equal to GVT must be retained.
(6) An *output queue*, containing negative copies of the messages the process has recently sent, kept in virtual send time order. They are needed in case of

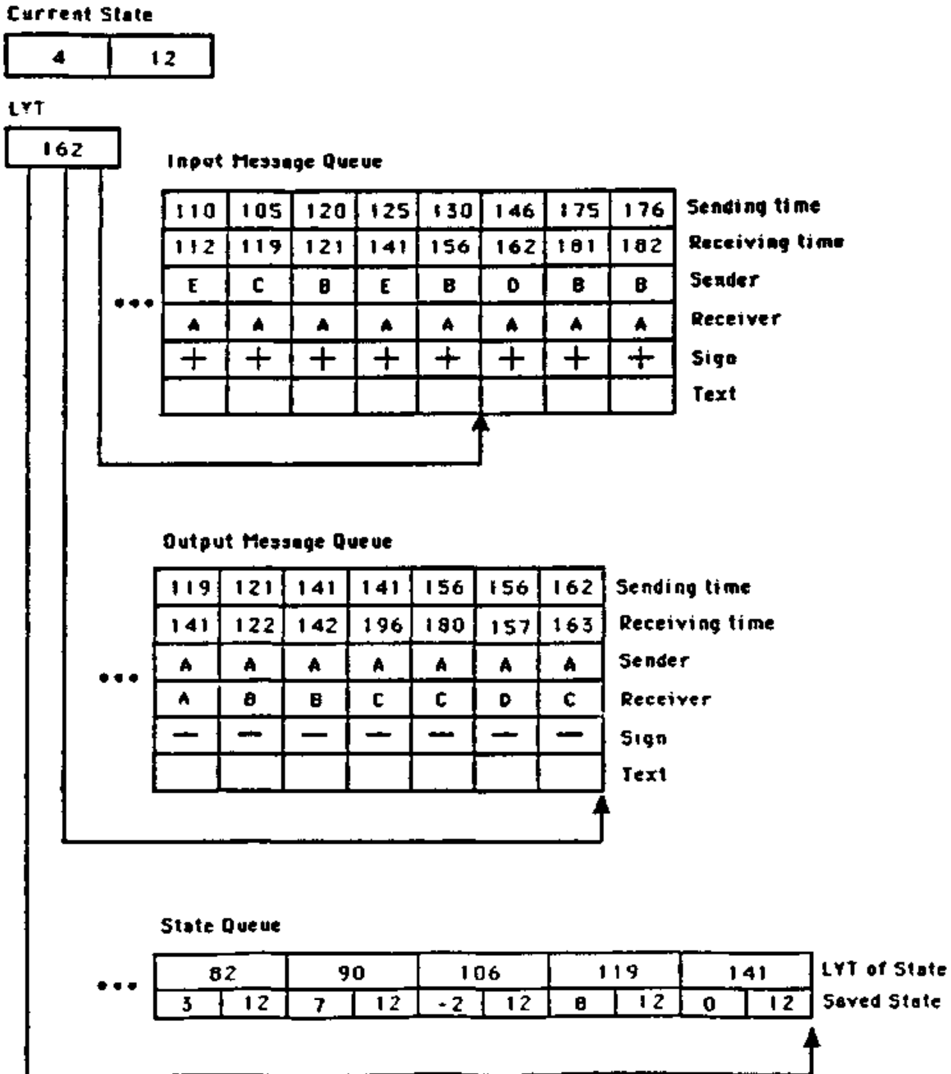

Fig. 1. A Time Warp process.

rollback, in order to "unsend" them. Again, only those messages whose virtual send times are greater than or equal to GVT must be retained.

For every message there exists an *antimessage* that is exactly like it in format and content except in one field, its *sign*. Two messages that are identical except for opposite signs are called antimessages of one another. All messages sent explicitly by user programs have a positive (+) sign; their antimessages have a negative (−) sign. Whenever a process sends a message, what actually happens is that a faithful copy of the message is transmitted to the receiver's input queue, and a negative copy, the antimessage, is retained in the sender's output queue for use in case the sender rolls back.

The manipulation of negative messages is symmetric to that of positive messages. What makes antimessages so useful is the queueing discipline defined for them. Whenever a message and its antimessage occur in the same queue, they immediately *annihilate* one another. Thus, the result of enqueueing a message may be to shorten the queue by one message rather than lengthen it by one. It does not matter which message, negative or positive, arrives at the queue first; if and when the second one arrives, the annihilation happens. In general, messages and antimessages are created in pairs and annihilated in pairs, and at any moment the algebraic sum of all messages in a Time Warp system is zero. It is no doubt unnecessary to point out that this annihilation discipline is reminiscent of the behavior of particles and antiparticles in physics.

Ordinarily, when a message arrives at the input queue of a process with a timestamp greater than the virtual clock time of its destination process, it is simply enqueued (by the interrupt routine), and the running process continues. But consider the situation that arises when a message with virtual receive time 135 arrives at an object whose virtual clock reads 162, as in Figure 2. The work done by this process since virtual time 135 may be incorrect and must be undone by rollback.

The first step in the rollback mechanism is simply to search the state queue for the last state of *A* saved before time 135, and then to restore it. We also restore 135 as the value of *A*'s local virtual clock. After this, we can discard from the state queue all states saved after time 135 and start *A* executing forward again.

However, we still must correct for the fact that between time 135 and 162, *A* sent several messages to other processes that must now be "unsent." The Time Warp mechanism accomplishes this through an extremely simple device.

To "unsend" a message it suffices simply to transmit its antimessage.

Time Warp thus transmits to their receivers all of those messages in *A*'s output queue that have virtual send times between 135 and 162, leaving *A* with no record that any of those messages ever existed. Process *A*, and all of its queues, are now in states they could have been in if the message with timestamp 135 had arrived in its proper order. The negative messages are on their way to their destinations, chasing the original positive ones. A negative message causes a rollback at its destination if its virtual receive time is less than the receiver's virtual time, just as a positive message would. Depending on timing, there are several possible sequences of events, all of which end up with the receiver in a state that it could reach if neither message ever existed. Here are the possibilities:

(1) If the original (positive) message has arrived, but has not yet been processed, its virtual receive time must be greater than the value in the receiver's virtual clock. The negative message, having the same virtual receive time, will be enqueued and will not cause a rollback. It will, however, cause an annihilation, leaving the receiver with no record that either message ever existed, exactly as desired.

Current State

5	12

LVT

135

Input Message Queue

...										
	110	105	120	132	125	130	146	175	176	Sending time
	112	119	121	135	141	156	162	181	182	Receiving time
	E	C	B	E	E	B	D	B	B	Sender
	A	A	A	A	A	A	A	A	A	Receiver
	+	+	+	+	+	+	+	+	+	Sign
										Text

Output Message Queue

...			
	119	121	Sending time
	141	122	Receiving time
	A	A	Sender
	A	B	Receiver
	—	—	Sign
			Text

State Queue

...									
	82		90		106		119		LVT of State
	3	12	7	12	-2	12	8	12	Saved State

Fig. 2. Rollback to time 135.

(2) Another possibility is that the original positive message has a virtual receive time that is now in the present or past with respect to the receiver's virtual clock, and it (and maybe others with still later virtual receive times) may have already been partially or completely processed, causing side

effects on the receiver's state and the sending of more messages to a third set of processes. In this circumstance, the negative message will also arrive in the receiver's virtual past and cause it to roll back to the virtual time when the positive one was received. It will, of course, also annihilate with the positive one, so that when the receiver starts executing forward again, the situation will again be as though neither message had ever existed. As part of this secondary rollback, more antimessages may be sent to the third set of processes, and the same actions we are describing will proceed there, *but at a later virtual time.*

(3) There is another case as well, namely, that the negative message arrives at the destination *before the positive one.* (Remember, we do not assume order preservation in the communication medium.) In this case it is enqueued as usual, and will eventually be annihilated when the positive message finally arrives. If the negative message is actually processed before it is annihilated by the positive message, the receiver may take any action at all (e.g., a no-op). Any action taken will anyway eventually be rolled back when the positive message arrives. An obvious optimization is for the negative message to be skipped in the input queue and treated as a no-op; then, when the positive message arrives, we can allow it to annihilate with the negative message, but inhibit any rollback.

This antimessage protocol is extremely robust, and works correctly under all possible circumstances. The levels of indirection may be to any depth, and there may even be circularity in the graph of antimessage paths with no ill effects. The rollback process need not be atomic, and indeed many interacting rollbacks may be going on simultaneously with no special synchronization. There is no possibility of deadlock (simply because there is no blocking). There is also no possibility of the "domino effect" (i.e., a cascading of rollbacks far into the past); the worst case is that *all* processes in the system roll back to the same virtual time as the original one did, and then proceed forward again.

There are a number of arguments suggesting that in a realistic system rollback is not as costly on the average as one might fear. We conjecture that most programs obey a *temporal locality principle*; namely, that most messages arrive in the virtual future at their destination, not causing any rollback at all, and that those that arrive in the virtual past tend strongly to arrive in the *recent* past, so that few events are rolled back. Many systems operate in a pattern where each event involves one input message and one output message. Hence, the number of antimessages *directly* sent in any one rollback is approximately the number of events rolled back. Even when a rollback causes antimessages to be sent, we can expect that most of those antimessages will not cause additional rollbacks because they each have virtual receive times greater than that of the message that caused the original rollback. Generally, the higher the virtual receive time of a message, the less likely it is to cause rollback. The extent to which the temporal locality principle applies is obviously application-dependent, and can only be verified empirically.

The "cost" of virtual time synchronization is only the cost of the rollback and antimessage overhead. The time taken for computation that is subsequently

rolled back (i.e., "lookahead" computation that is therefore "wasted") should not be considered a cost when comparing to alternative synchronization mechanisms. The only alternative to lookahead/rollback is for the process to be blocked (i.e., doing *nothing*) for the same length of real time as the lookahead computation, which is just as much of a "waste."

Several studies of the performance of Time Warp are in progress. For analytical approaches, see [3, 15, 18]. One empirical study is reported in [28]. Each study indicates that speedup of discrete event simulations is possible using Time Warp. A more ambitious implementation on the Caltech Hypercube at the Jet Propulsion Laboratory is underway [14], and more definitive performance statistics should soon be available.

4.3 The Global Control Mechanism

The local control part of the Time Warp mechanism leaves unresolved a number of critical issues. How can we be sure that amidst all of the rollback activity the system makes progress globally? How can global termination be detected? How can errors and I/O be handled in the face of rollback? How can we avoid running out of memory when the local control mechanism calls for saving two copies of all messages and several copies of processes' states? The global control mechanism resolves all of these issues, and several others.

The central concept of the global control mechanism is *global virtual time* (*GVT*). Global virtual time is a property of an instantaneous global snapshot of the system at real time *r*, and is defined as follows:

Definition. GVT at real time *r* is the minimum of (1) all virtual times in all virtual clocks at time *r*, and (2) of the virtual send times of all messages that have been sent but have not yet been processed at time *r*.

GVT is defined in terms of virtual *send* time of unprocessed messages, instead of the virtual *receive* time, because of the flow control protocol discussed later. Messages that have not been processed include those that are in transit or are in the future part of the receiver's input queue. It is easily shown by induction on the number of message communication acts (sends, arrivals, and receipts) that *GVT never decreases*, despite the fact that individual local virtual clocks roll back frequently. As such, GVT serves as a floor for the virtual times to which any process can ever again roll back. In fact, if every event completes normally, and if messages are delivered reliably (as we assume), and if the scheduler does not indefinitely postpone execution of the farthest-behind processes (as we also assume), and if there is sufficient memory, then *GVT must eventually increase*.

These properties make it appropriate to consider GVT as the virtual clock for the system as a whole, and to use it as the measure of system *progress* (i.e., a measure of how much of the system's activity is final and complete). GVT can thus be viewed as a moving *commitment horizon*: any event with virtual time less than GVT cannot be rolled back, and may be irrevocably committed with safety.

The definition of GVT, given as it is in terms of an instantaneous snapshot of distributed system with messages in transit, is not entirely operational. It is generally impossible for the Time Warp mechanism to know at real time *r* exactly

what GVT at r is. Of course, user processes need no access to GVT, but the Time Warp mechanism uses it for the global control of the system. Fortunately, GVT can be characterized more operationally as being less than or equal to the minimum of (a) all virtual times in all virtual clocks in the snapshot, (b) all virtual send times of messages that have been sent but not yet acknowledged (and may therefore be in transit at the moment of the snapshot), and (c) all virtual send times of messages in input queues that have not yet been processed by the receiving process. This characterization leads to a fast, distributed GVT-estimation algorithm that takes $O(d)$ time, where d is the delay required for one broadcast to all processors in the system. The algorithm runs concurrently with the main computation and returns a value that is between the true GVT at the moment the algorithm starts and the true GVT at the moment of its completion. It thus gives a slightly out-of-date value for GVT, which is fundamentally the best we can do without synchronizing the entire system. One such algorithm is given in [26].

During the execution of a virtual time system, the Time Warp mechanism must estimate GVT every so often. The actual frequency is a trade-off: high frequency produces faster response time and better space utilization (because of more frequent storage reclamation, to be discussed later), but it also uses processor time and network bandwidth, and thus slows progress. This is similar to the trade-off we are familiar with in time-slicing operating systems when we adjust the length of the time quantum.

4.3.1 *Use of GVT for Memory Management and Flow Control.* One of the attractive features of the Time Warp mechanism is that it is possible to give simple, natural algorithms for managing memory. In addition to the memory used for code and the current data of processes (which the programmer is responsible for managing), there are four additional kinds of memory overhead to be managed.

(1) Old states in the state queues.
(2) Messages stored in output queues.
(3) "Past" messages (in input queues) that have already been processed.
(4) "Future" messages (in input queues) that have not yet been received.

The first three classes of storage, used only to support rollback, are all managed similarly. Any message in an input or output queue whose virtual receive time is less than GVT can be discarded, as it is impossible to roll back to a virtual time when it might be either re-received or canceled with an antimessage. Similarly, for each process all but one saved state older than GVT can be discarded. We refer generally to the process of destroying information older than GVT as *fossil collection.*

Managing the fourth class of storage, that containing unreceived messages, is essentially the flow control problem common to all distributed systems. In most environments, where the only synchronization tool is process blocking, flow control protocols act as valves to limit the flow of messages from sender to receiver. The receiver must be careful never to accept too many messages, because every message it accepts responsibility for must be buffered until it is processed and discarded. But because we assume every message is stamped with the virtual

coordinates of the sending and receiving events, and because rollback is possible, the flow control problem has more structure in the context of virtual time than it usually does, and a new approach is warranted.

If a receiver's memory is full of input messages, the Time Warp mechanism may be able to recover space by *returning* an unreceived message to the process that sent it. The original sender must then roll back to the state it was in when it sent the message, and resend the message as it executes forward again. When it is necessary to send back a message, the natural choice is the unreceived message having the latest virtual send time (regardless of where it came from). Returning a message to the sender is the message analog of rolling back a process; the two operations are the obverse and reverse of the same coin. We do not have space here to give a better description of this kind of protocol or the arguments in its favor; details will be published separately.

4.3.2 *Normal Termination Detection.* Termination detection in distributed systems has been an active field of research for some time now [9, 11]. With Time Warp the detection of termination is just one of several global issues handled in terms of GVT. Recall that whenever a process runs out of messages it terminates, and its local virtual clock is set to *+inf*. This is the only circumstance in which a virtual clock can read *+inf*. Therefore, whenever GVT reaches *+inf*, all local virtual clocks must read *+inf* and no messages can be in transit. No process can ever again unterminate by rolling back to a finite virtual time, and so whenever the GVT calculation returns *+inf*, Time Warp signals termination.

4.3.3 *Error Handling.* When a process commits a run-time error we assume its state must be marked *error*. This should not necessarily cause termination of the entire computation because the erring process might roll back and "unerr." An error is only "committed" if it is impossible for the process to roll back to a virtual time at or before the error.

A process in an *error* state may be stopped or it may be allowed to keep executing, but all successive states are also marked *error*. If and when an *error* state is fossil-collected (because its virtual time is older than GVT), then no rollback will ever undo the error. Only then should the error be considered committed and reported to some policy software and/or to the user.

4.3.4 *Input and Output.* When a process sends a command to an output device, or any other external agent, it is important that the physical output activity not be committed immediately, because the sending process may roll back and cancel the output request. Output can only be physically performed when GVT exceeds the virtual receive time of the message containing the command. After that point, no antimessage for the command can ever be generated, and the output can be safely committed (in timestamp order, of course).

4.3.5 *Snapshots and Crash Recovery.* The problem of taking a consistent, global snapshot that is useful for continuation in the event of a crash arises with all distributed systems [6, 25]. A snapshot at a single instant of real time is, of course, impossible to implement. But, in a virtual time system, a snapshot of all processes (and the relevant messages), at a particular virtual time, forms a natural and meaningful snapshot of the system that is easily implementable. A

full snapshot of the system at virtual time t can be constructed by a procedure in which each process snapshots itself as it passes virtual time t in the forward direction, and "unsnapshots" itself whenever it rolls back over virtual time t. Whenever GVT exceeds t, the snapshot is complete and valid.

5. EXAMPLES OF VIRTUAL TIME SYSTEMS

In the next three subsections we give a variety of examples of distributed systems that can be viewed as virtual time systems.

5.1 Example 1. Distributed Discrete Event Simulation

The most extensively studied example is distributed discrete event simulation, in which every process represents an object in the simulation, and virtual time is identified with simulation time. The fundamental operation in discrete event simulation is for one process to schedule an event for execution by another at a later simulation time. In a virtual time system we emulate this simply by having the first process send an *event message* to the second process, with its virtual receive time equal to the event's scheduled simulation time. The requirement that each process must receive messages in virtual receive time order is equivalent to the simulation requirement that events be executed in simulation time order. Simulation is clearly one of the most "general" applications of the virtual time paradigm because the virtual times (simulation times) of events are completely under the control of the user, and because it makes use of almost all of the degrees of freedom allowed in the definition of a virtual time system.

Any mechanism for general distributed discrete event simulation can be used as an implementation of virtual time. Chandy and Misra in [7] give a simulation method based on blocking and distributed deadlock detection. Other distributed simulation methods are described in [4, 8, 21, 22, 23, 32].

5.2 Example 2. Distributed Database Concurrency Control

In a distributed database the fundamental synchronization problem is to make distributed transactions appear to be atomic with respect to other transactions. To accomplish these effects in a virtual time system it is only necessary to do two things. First, the entire database system, including all transaction software, most management software, and all of the "data," must be cast as a collection of processes communicating by message. In particular, data items (records, tables, etc.) must be viewed as processes that respond to *read* and *write* messages. Second, the system must ensure that each transaction process executes within a band of virtual time that does not overlap with the bands allocated to other transactions. This can be done simply by using the real time of a transaction's initiation as the high-order bits of its virtual time band, with the place of initiation as the middle order bits to break ties. (It does not matter if the real time clocks are not perfectly synchronized.) The apparent indivisibility of transactions follows directly. A full description of virtual time used as the basis of database concurrency control can be found in [16].

This assignment of virtual time bands to transactions guarantees not only that they are atomic, but also that they are apparently executed (committed) strictly in virtual time order (i.e., in order of their initiation). This is a stronger scheduling constraint than the usual serializability criterion, but it should not have a serious deleterious effect on the throughput of the database system because the Time Warp mechanism is not constrained to actually execute transactions in virtual time order; it need only commit them in that order. It may, however, have a deleterious effect on response time.

One benefit to the virtual time approach is that neither deadlock nor starvation is possible, and there is no need to add any extra software to prevent or detect them. It is also easily possible to define transactions that are themselves concurrent and have nested atomic subtransactions. We need only allocate to the subtransactions the nonoverlapping sub-bands of the virtual time band allocated to the main transaction.

Virtual time used for database concurrency control is quite different from that used for distributed simulation. First, virtual time is derived from real time in the database case, whereas it is independent in the simulation case. Second, virtual time values in a simulation are actually manipulated as data and assigned to the receive time fields of messages by user software. By contrast, in database systems, the virtual time values will typically be "hidden" from most levels of DBMS software and from users.

In many respects the Time Warp mechanism applied to database concurrency control is similar to *multiple-version concurrency control mechanisms* [1, 2, 20], in that it maintains several successive versions of each data item (in the data item's state queue) and has the ability to satisfy a *read* request that is timestamped earlier than the "current" version of the data by accessing (rolling back to) a saved earlier version. But when a multiple-version mechanism is faced with a *write* request that is timestamped earlier than the current version of the data, there is often no choice but to abort the entire transaction that the write request is part of (and possibly some other transactions in progress as well), and to restart it with a later timestamp. In situations where there is a high probability of collision, transaction abortion can lead to much wasted computation and to the possibility of starvation. The Time Warp mechanism, however, never aborts a transaction. It may roll back *parts* of several transactions when a collision occurs, but the amount of the computation unwound during a collision is limited to the part that would be causally affected if requests were handled out of timestamp order, which is often very much less than the entire transaction.

5.3 Example 3. Virtual Circuit Communication

One of the main functions of some network communication protocols is to provide a "virtual circuit" facility, a buffering and synchronization mechanism that delivers messages to the receiver in the same order they were sent. This effect can be implemented using a virtual time system by defining the virtual receive time of a message to be the real time of its sending. Processing messages in virtual receive time order is then equivalent to processing them in sending order. Implementing virtual circuit communication as a virtual time system may

not offer any particular performance benefits over other (simpler) implementations, but it does show something of the breadth of the virtual time concept. There may be important practical benefits in environments where the ability to checkpoint a distributed system is required.

6. EXTENDED ANALOGY TO VIRTUAL MEMORY

One of the most interesting things about the virtual time paradigm is that there is an extended analogy between it and another phenomenon already well understood: virtual memory. In fact, the term "virtual time" is intended to evoke thoughts of virtual memory, and the vocabulary and theory of virtual memory can, by systematically substituting temporal for spatial terms and vice-versa, be a fairly reliable guide to virtual time. This extended analogy between the two may lend some credibility to our otherwise unorthodox approach to distributed computation. We present the comparison as a sequence of parallel concepts in which space and time play almost symmetric roles.

(1) We consider a page in virtual memory to be analogous to an event in virtual time. The virtual address of a page is its spatial coordinate; the virtual time of an event is its temporal coordinate.

(2) We consider a page that is resident in main memory at time t to be analogous to an event with a virtual time in the future of process x (i.e., having a virtual time greater than the virtual clock of process x); a page out of memory at time t is analogous to an event in the present or past of process x.

(3) Accessing a page that is resident in main memory is comparatively inexpensive, but accessing a page out of memory causes a very expensive page fault. Similarly, sending a message that arrives in the virtual future of the receiving process is comparatively inexpensive, while sending a message into its virtual past causes a very expensive *time fault*, that is, rollback.

(4) Under a virtual memory system it is only cost-effective in time to execute programs that obey the spatial locality principle, so that most memory accesses are to pages already resident in memory, and page faults are relatively rare. Likewise, under a virtual time system, it is only cost-effective to run programs that obey the temporal locality principle, that is, most messages arrive in the virtual future of the destination processes so that time faults are relatively rare. (The cost of a page fault is on the order of 10,000 times the cost of a simple memory access. While no empirical measurements have been made, it is hard to imagine an implementation in which a message causing a time fault costs more than 100 times that of a message not causing one. Thus, we can expect to be able to tolerate a considerably higher fraction of time faults than page faults.)

(5) The term "memory mapping" refers to the translation of virtual addresses to real addresses. We could use the term *time mapping* to refer to the mapping of virtual times to real times (i.e., deciding when in real time an event with a particular virtual time is to be executed). There is, however, already a term for that: "scheduling." Note that the same virtual address may be mapped to different real addresses at different times, and similarly, the same virtual time may be mapped to (scheduled at) different real times at different places.

(6) The only acceptable memory maps are the one-to-one functions because they preserve *distinctness*, mapping distinct virtual addresses (pages) to distinct

real addresses (page frames). At any given moment, some virtual addresses may be unmapped because they refer to pages not in memory. Similarly, the only acceptable time maps are the strictly increasing functions because they preserve *order*, mapping distinct virtual times into distinct real times, earlier virtual times into earlier real times, and later virtual times into later real times. At any given place, some events may be unmapped (not yet scheduled) because they are in the local future.

(7) For a process running under virtual memory, the page fault rate can usually be reduced by increasing the number of pages it is permitted to have in main memory. Similarly, the time fault rate of a process running under virtual time can usually be reduced by increasing the number of events that are in its near future. It is important to understand that increasing the number of events in the near future of a process means slowing it down somehow, perhaps by inserting artificial delays.

(8) If a process can have enough pages in memory, its page fault rate can be reduced to zero; in general, this is undesirable because it leads to inefficient use of main memory. Similarly, if a process is slowed down sufficiently (so that it becomes the farthest behind process in the system), its time fault rate can be reduced to zero; this too is undesirable because it then becomes the bottleneck process, artificially holding back the progress of GVT, thereby leading to inefficient use of real time.

The analogy between virtual time and virtual memory can be extended further in order to discover temporal analogies to such virtual memory concepts as working set and pointer-as-virtual-address, and it is very thought provoking to do so. The success of the analogy suggests that virtual time can provide the same kind of clean, efficiently implementable abstraction of the time resource in a distributed environment that virtual memory provides for space resources. This, at least, is the goal, and work is now in progress to demonstrate it empirically.

ACKNOWLEDGMENTS

I thank Henry Sowizral for his coinvention of the Time Warp mechanism, Ami Motro for clarifying the issues in database concurrency control, and Orna Berry, Rick Carlson, Youan Chang, Anat Gafni, Arthur Goldberg, and Andrej Witkowski for their insight and criticism. I also thank Phil Klahr and the Rand Corporation for providing the impetus to study the distributed simulation problem, and the Jet Propulsion Laboratory for aiding in the current Time Warp implementation effort.

REFERENCES

1. BERNSTEIN, P. A., AND GOODMAN, N. Concurrency control in distributed database systems. *ACM Comput. Surv. 13*, 2 (June 1981).
2. BERNSTEIN, P. A., AND GOODMAN, N. A sophisticate's introduction to distributed database concurrency control. In *Proceedings of the 8th International Conference on Very Large Databases* (Sept. 1982).
3. BERRY, O., AND JEFFERSON, D. R. Critical path analysis of distributed simulation. In *1985 Society for Computer Simulation Multiconference* (San Diego, Calif., Jan. 1985).

4. BRYANT, R. E. Simulation of packet communication architecture computer systems. Ph.D. dissertation, M.I.T., Nov. 1977.
5. CALTECH. *Annual Report 1983–1984 and Recent Documentation.* Caltech Concurrent Computation Project, Jet Propulsion Laboratory, Pasadena, Calif., Aug. 30, 1984.
6. CHANDY, K. M., AND LAMPORT, L. Distributed snapshots: Determining global states of distributed systems. (To appear, *ACM Trans. Comput. Syst.*).
7. CHANDY, K. M., AND MISRA, J. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM 24*, 4 (Apr. 1981), 198–206.
8. CHANDY, K. M., AND MISRA, J. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. Softw. Eng. SE-5*, 5 (Sept. 1979), 440–452.
9. DIJKSTRA, E. W., AND SCHOLTEN, C. S. Termination detection in diffusing computations. *Inf. Process. Lett. 11*, 1 (Aug. 29, 1980).
10. FOX, G. C., AND OTTO, S. W. Algorithms for concurrent processors. *Phys. Today* (May, 1984), 50.
11. FRANCEZ, N. Distributed termination. *ACM Trans. Program. Lang. Syst. 2*, 1 (Jan. 1980), 42–55.
12. JEFFERSON, D. R., AND SOWIZRAL, H. A. Fast concurrent simulation using the Time Warp mechanism, part I: Local control. Rand Note N-1906AF, the Rand Corp., Santa Monica, Calif., Dec. 1982.
13. JEFFERSON, D. R., AND SOWIZRAL, H. A. Fast concurrent simulation using the Time Warp mechanism. In *Proceedings of the SCS Distributed Simulation Conference* (San Diego, Calif., Jan. 1985).
14. JEFFERSON, D. R., ET AL. Implementation of Time Warp on the Caltech Hypercube. In *1985 Society for Computer Simulation Multiconference* (San Diego, Calif., Jan. 1985).
15. JEFFERSON, D. R., AND WITKOWSKI, A. An approach to performance analysis of timestamp-driven synchronization mechanisms. In *Proceedings of the 3rd ACM Annual Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 1984), ACM, New York.
16. JEFFERSON, D. R., AND MOTRO, A. The Time Warp mechanism for database concurrency control. U.S.C. Tech. Rep., Dept. of Computer Science, Univ. of Southern California, Los Angeles, June 1983.
17. LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.
18. LAVENBERG, S., MUNTZ, R., AND SAMADI, B. Performance analysis of a rollback method for distributed simulation. Dept. of Computer Science, U.C.L.A., 1982.
19. METCALFE, R. M., AND BOGGS, D. R. Ethernet: Distributed packet switching for local computer networks. *Commun. ACM 19*, 7 (July, 1976), 395–404.
20. PAPADIMITRIOU, C. H., AND KANELLAKIS, P. C. On concurrency control by multiple versions. In *ACM Conference on Principles of Database Systems* (PODS), 1982, ACM, New York.
21. PEACOCK, J. K., WONG, J. W., AND MANNING, E. G. A distributed approach to queueing network simulation. In *1979 Winter Simulation Conference*, IEEE, New York, 1979, 399–406.
22. PEACOCK, J. K., MANNING, E. G., AND WONG, J. W. Synchronization of distributed simulation using broadcast algorithms. *Comput. Networks 4*, 1 (Feb. 1980), 3–10.
23. PEACOCK, J. K., WONG, J. W., AND MANNING, E. G. Distributed simulation using a network of processors. *Comput. Networks 3* , 1 (Feb. 1979), 44–56.
24. REED, D. P. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst. 1*, 1 (Feb. 1983), 3–23.
25. RUSSELL, D. L. State restoration in systems of communicating processes. *IEEE Trans. Softw. Eng. SE-6*, 2 (Mar. 1980), 183–194.
26. SAMADI, B. Distributed simulation: Performance and analysis. Ph.D. dissertation, Dept. of Computer Science, UCLA, Los Angeles, 1985.
27. SCHNEIDER, F. B. Synchronization in distributed programs. *ACM Trans. Program. Lang. Syst. 4*, 2 (Apr. 1982), 179–195.
28. SCHWARTZ, J. T. Ultracomputers. *ACM Trans. Program. Lang. Syst. 2*, 4 (Oct. 1980), 484–521.
29. SOWIZRAL, H. A. The Time Warp simulation system and its performance. In *1985 Society for Computer Simulation Multiconference* (San Diego, Calif., Jan. 1985).
30. SWAN, R., FULLER, S., AND SIEWIOREK, D. CM*—A modular multimicrocomputer. In *Proceed-*

ings of the 1977 National Computer Conference (Apr. 1981), AFIPS Press, Baltimore, Md., 198–206.

31. WULF, W. A., LEVIN, R., AND HARBISON, S. P. *Hydra/C.mmp: An Experimental Computer System*. McGraw-Hill, New York, 1981.
32. WYATT, D., SHEPPARD, S., AND YOUNG, R. An experiment in microprocessor-based distributed digital simulation. In *Proceedings of the 1983 Winter Simulation Conference* (Arlington, Va., Dec. 1983), S. Roberts, J. Banks, and B. Schmeiser, Eds.

Received June 1983; revised January 1985; accepted February 1985