Optimistic Distributed Simulation Based on Transitive Dependency Tracking

Om P. Damani
Dept. of Computer Sci.
Uni. of Texas at Austin
damani@cs.utexas.edu

Yi-Min Wang
AT&T Labs-Research
Murray Hill, NJ
ymwang@research.att.com

Vijay K. Garg*
Dept. of Elect. & Comp. Eng
Uni. of Texas at Austin
garg@ece.utexas.edu

Abstract

In traditional optimistic distributed simulation protocols, a logical process(LP) receiving a straggler rolls back and sends out anti-messages. Receiver of an anti-message may also roll back and send out more anti-messages. So a single straggler may result in a large number of anti-messages and multiple rollbacks of some LPs. In our protocol, an LP receiving a straggler broadcasts its rollback. On receiving this announcement, other LPs may roll back but they do not announce their rollbacks. So each LP rolls back at most once in response to each straggler. Anti-messages are not used. This eliminates the need for output queues and results in simple memory management. It also eliminates the problem of cascading rollbacks and echoing, and results in faster simulation. All this is achieved by a scheme for maintaining transitive dependency information. The cost incurred includes the tagging of each message with extra dependency information and the increased processing time upon receiving a message. We also present the similarities between the two areas of distributed simulation and distributed recovery. We show how the solutions for one area can be applied to the other area.

1 Introduction

We modify the time warp algorithm to quickly stop the spread of erroneous computation. Our scheme does not require output queues and anti-messages. This results in less memory overhead and simple memory management algorithms. It also eliminates the problem of cascading rollbacks and echoing [15], resulting in faster simulation. We use aggressive cancellation [7].

Our protocol is an adaptation of a similar protocol for the problem of distributed recovery [4, 21]. We illustrate the main concept behind this scheme with the help of Figure 1. In the figure, horizontal arrows show the direction of the simulation time. Messages are shown by the inter-process directed arrows. Circles represent states. State transition is caused by acting on the message associated with the incoming arrow. For example, the state transition of $P1$ from $s10$ to $s11$ happened when $P1$ acted on $m0$. In the time warp scheme, when a logical process (LP) $P2$ receives a straggler (i.e., a message which schedules an event in $P2$'s past) it rolls back the state $s20$ and sends an anti-message corresponding to message $m2$. On receiving this anti-message, $P1$ rolls back state $s10$ and sends an anti-message corresponding to $m1$. It then acts on the next message in its message queue, which happens to be $m0$. On receiving the anti-message for $m1$, $P0$ rolls back s00 and sends an anti-message for $m0$. On receiving this anti-message, $P1$ rolls back $s11$.

In our scheme, transitive dependency information is maintained with all states and messages. After rolling back $s20$ due to a straggler, $P2$ broadcasts that $s20$ has been rolled back. On receiving this announcement, $P1$ rolls back $s10$ as it finds that $s10$ is transitively dependent on $s20$. $P1$ also finds that $m0$ is transitively dependent on $s20$ and discards it. Similarly $P0$ rolls back $s00$ on receiving the broadcast. We see that $P1$ was able to discard $m0$ faster compared to the previous scheme. Even $P0$ would likely receive the broadcast faster than receiving the anti-message for $m1$ as that can be sent only after $P1$ has rolled back $s10$. Therefore, simulation should proceed faster. As explained later, we use incarnation number to distinguish between two states with the same timestamp, one of which is committed and the other is rolled back.

We only need the LP that receives a straggler to broadcast the timestamp of the straggler. Every other LP can determine whether they need to roll back or not by comparing their local dependency information

*supported in part by the NSF Grants CCR-9520540 and ECS-9414780, a TRW faculty assistantship award, a General Motors Fellowship, and an IBM grant.

with the broadcast timestamp. Other LPs that roll back in response to a rollback announcement do not send any announcement or anti-messages. Hence, each LP rolls back at most once in response to a straggler, and the problem of multiple rollbacks is avoided. Several schemes have been proposed to minimize the

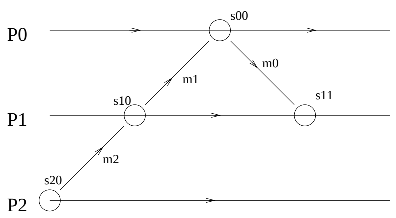

Figure 1: A Distributed Simulation.

spread of erroneous computations. A survey of these schemes can be found in [7]. The Filter protocol by Prakash and Subramanian [17] is most closely related to our work. It maintain a list of *assumptions* with each message, which describe the class of straggler events that could cause this message to be canceled. It maintains one assumption per channel, whereas our protocol can be viewed as maintaining one assumption per LP. In the worst case, Filter tags each message with $O(n^2)$ integers whereas our protocol tags $O(n)$ integers, where n is the number of LPs in the system. Since for some applications even $O(n)$-tagging may not be acceptable, we also describe techniques to further reduce this overhead. If a subset of LPs interact mostly with each other, then, for most of the time, the tag size of their messages will be bounded by the size of the subset.

The paper is organized as follows. Section 2 describes the basic model of simulation; Section 3 introduces the happen before relation between states and the simulation vector which serves as the basis of our optimistic simulation protocol; Section 4 describes the protocol and gives a correctness proof; Section 5 presents optimization techniques to reduce the overhead of the protocol; Section 6 compares distributed simulation with distributed recovery.

2 Model of Simulation

We consider event-driven optimistic simulation. The execution of an LP consists of a sequence of states where each state transition is caused by the execution of an event. If there are multiple events scheduled at the same time, it can execute those events in an arbitrary order. In addition to causing a state transition, executing an event may also schedule new events for other LPs (or the local LP) by sending messages. When LP $P1$ acts on a message from $P2$, $P1$ becomes dependent on $P2$. This dependency relation is transitive.

The arrival of a straggler causes an LP to roll back. A state that is rolled back, or is transitively dependent on a rolled back state is called an *orphan state*. A message sent from an orphan state is called an *orphan message*. For correctness of a simulation, all orphan states must be rolled back and all orphan messages must be discarded.

An example of a distributed simulation is shown in Figure 2. Numbers shown in parentheses are either the virtual times of states or the virtual times of scheduled events carried by messages. Solid lines indicate useful computations, while dashed lines indicate rolled back computations. In Figure 2(a), $s00$ schedules an event for $P1$ at time 5 by sending message $m0$. $P1$ optimistically executes this event, resulting in a state transition from $s10$ to $s11$, and schedules an event for $P2$ at time 7 by sending message $m1$. Then $P1$ receives message $m2$ which schedules an event at time 2 and is detected as a straggler. Execution after the arrival of this straggler is shown in Figure 2(b). $P1$ rolls back, restores $s10$, takes actions needed for maintaining the correctness of the simulation (to be described later) and restarts from state $r10$. Then it broadcasts a rollback announcement (shown by dotted arrows), acts on $m2$, and then acts on $m0$. Upon receiving the rollback announcement from $P1$, $P2$ realizes that it is dependent on a rolled back state and so it also rolls back, restores state $s20$, takes actions needed, and restarts from state $r20$. Finally, the orphan message $m1$ is discarded by $P2$.

3 Dependency Tracking

From here on, i,j refer to LP numbers; k refers to incarnation number; s,u,w,x refer to states; P_i refers to logical process i; $s.p$ refers to the number associated with the LP to which s belongs, that is, $s.p = i \Rightarrow s \in P_i$; m refers to a message and e refers to an event.

3.1 Happen Before Relation

Lamport defined the *happen before*($\rightarrow$) relation between events in a rollback-free distributed computation [12]. To take rollbacks into account, we extend this relation. As in [4, 21], we define it for the states. For any two states s and u, $s \rightarrow u$ is the transitive closure of the relation defined by the following three conditions:

1. $s.p = u.p$ and s immediately precedes u.
2. $s.p = u.p$ and s is the state restored after a rollback and u is the state after $P_{u.p}$ has taken the

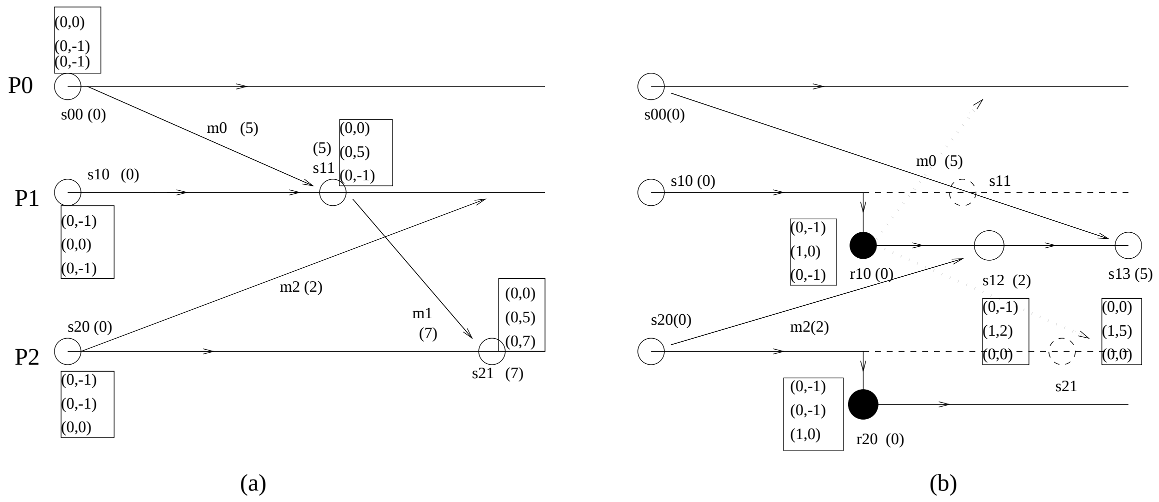

Figure 2: Using Simulation Vector for Distributed Simulation. (a) Pre-straggler computation. (b) Post-straggler computation.

actions needed to maintain the correctness of simulation despite the rollbacks. For example, in Figure 2(b), $s20 \rightarrow r20$.

3. s is the sender of a message m and u is the receiver's state after the event scheduled by m is executed.

For example, in Figure 2(a), $s10 \rightarrow s11$ and $s00 \rightarrow s21$, and in Figure 2(b) $s11 \not\rightarrow r10$. Saying s happened before u is equivalent to saying that u is *transitively dependent* on s.

For our protocol, "actions needed to maintain the correctness of simulation" include broadcasting a rollback announcement and incrementing the incarnation number. For other protocols, the actions may be different. For example, in time warp, these actions include the sending of anti-messages. Our definition of happen before is independent of such actions. The terms "rollback announcements" and "tokens" will be used interchangeably. Tokens do not contribute to the happen before relation. So if u receives a token from s, u does not become transitively dependent on s due to this token.

3.2 Simulation Vector

A *vector clock* is a vector of size n where n is the number of processes in the system [16]. Each vector entry is a timestamp that usually counts the number of send and receive events of a process. In the context of distributed simulation, we modify and extend the notion of vector clock, and define a *Simulation Vector (SV)* as follows. To maintain dependency in the presence of rollbacks, we extend each entry to contain both a timestamp and an *incarnation number* [19]. The timestamp in the i^{th} entry of the SV of P_i corresponds to the virtual time of P_i. The timestamp in the j^{th} entry corresponds to the virtual time of the latest state of P_j on which P_i depends. The incarnation number in the i^{th} entry is equal to the number of times P_i has rolled back. The incarnation number in the j^{th} entry is equal to the highest incarnation number of P_j on which P_i depends. Let entry en be a tuple (incarnation v, timestamp t). We define a lexicographical ordering between entries as follows: $en_1 < en_2 \equiv (v_1 < v_2) \vee [(v_1 = v_2) \wedge (t_1 < t_2)]$.

Simulation vectors are used to maintain transitive dependency information. Suppose P_i schedules an event e for P_j at time t by sending a message m. P_i attaches its current SV to m. By "virtual time of m", we mean the scheduled time of the event e. If m is neither an orphan nor a straggler, it is kept in the incoming queue by P_j. When the event corresponding to m is executed, P_j updates its SV with m's SV by taking the componentwise lexicographical maximum. Then P_j updates its virtual time (denoted by the j^{th} timestamp in its SV) to the virtual time of m. A formal description of the SV protocol is given in Figure 3. Examples of SV are shown in Figure 2 where the SV of each state is shown in the box near it.

The SV has properties similar to a vector clock. It can be used to detect the transitive dependencies between states. The following theorem shows the relationship between virtual time and SV.

Theorem 1 *The timestamp in the i^{th} entry of P_i's SV corresponds to the virtual time of P_i.*

```
LP P_i :
type entry = (int inc, int ts)
              /* incarnation, timestamp */
var sv:       array [0..n-1] of entry
Initialize :
  ∀ j : sv[j].inc = 0 ; sv[j].ts = -1 ;
  sv[i].ts = 0 ;
Send_message(m) :
  m.sv = sv ;
  m.ts = time at which m should be executed ;
  send (m.data, m.ts, m.sv) ;
 Execute_message (m.data, m.ts, m.sv) :
  /* P_i executes event scheduled by m */
  ∀ j: sv[j] = max(sv[j],m.sv[j]) ;
  sv[i].ts = m.ts ;
 Rollback :
  /* State s is restored. So, sv = s.sv */
  sv[i].inc + + ;
```

Figure 3: Formal description of the Simulation Vector protocol

Proof. By Induction. The above claim is true for the initial state of P_i. While executing a message, the virtual time of the P_i is correctly set. After a rollback, virtual time of the restored state remains unchanged. ∎

Let $s.sv$ denote the SV of $P_{s.p}$ in state s. We define the ordering between two SV's c and d as follows.

$$c \leq d \equiv (\forall i : c[i] \leq d[i]).$$

In P_i's SV, the j^{th} timestamp denotes the maximum virtual time of P_j on which P_i depends. This timestamp should not be greater than P_i's own virtual time. Lemma 1 formalizes the above notion.

Lemma 1 *The timestamp in the i^{th} entry of the SV of a state of P_i has the highest value among all the timestamps in this SV.*

Proof. By induction. The lemma is true for the initial state of P_i. Assume that state s of P_j sent a message m to P_i. State u of P_i executed m, resulting in state w. By induction hypothesis, $s.sv[j].ts$ and the $u.sv[i].ts$ are the highest timestamps in their SV's. So the maximum of these two timestamps is greater than all the timestamps in $w.sv$ after the *max* operation in *Execute_message*. Now $m.ts$, the virtual time of message m, is not less than the virtual time of the state s sending the message. It is also not less than the virtual time of the state u acting on the message, otherwise, it would have caused a rollback. So by theorem 1, $m.ts$ is not less than the maximum of $s.sv[j].ts$ and the $u.sv[i].ts$. Hence setting the $w.sv[i].ts$ to $m.ts$ preserves the above property. All other routines do not change the timestamps. ∎

The following two lemmas give the relationship between the SV and the happen before relation.

Lemma 2 *If s happens before u, then $s.sv$ is less than or equal to $u.sv$.*

Proof. By induction. Consider any two states s and u such that s happens before u by applying one of the three rules in the definition of happen before. In case of rule 1, state s is changed to state u by acting on a message m. The update of the SV by taking the maximum in the routine *Execute_message* maintains the above property. Now consider the next action in which $u.sv[u.p].ts$ is set to $m.ts$. Since virtual time of m cannot be less than the virtual time of state s executing it, this operation also maintains the above property.

In case of rule 2, in routine *Rollback*, the update of the SV by incrementing the incarnation number preserves the above property. The case of rule 3 is similar to that of the rule 1. Let state w change to state u by acting on the message m sent by state s. By lemma 1, in m's SV, $s.p^{th}$ timestamp is not less than the $u.p^{th}$ timestamp. Also the virtual time of m is not less than the $s.p^{th}$ timestamp in its SV. Hence setting the i^{th} timestamp to the virtual time of m, after taking *max*, preserves the above property. ∎

The following lemma shows that LPs acquire timestamps by becoming dependent on other LPs. This property is later used to detect orphans. This lemma states that if j^{th} timestamp in state w's SV is not minus one (an impossible virtual time) then w must be dependent on a state u of P_j, where the virtual time of u is $w.sv[j].ts$.

Lemma 3 $\forall w, j : j \neq w.p : (w.sv[j].ts = -1) \vee (\exists u : (u.p = j) \wedge (u \rightarrow w) \wedge (u.sv[j] = w.sv[j]))$.

Proof. By induction. *Initialize* trivially satisfies the above property. In *Execute_message*, let x be the state that sends m and let state s change to state w by acting on m. By induction hypothesis, x and s satisfy the lemma.

In taking maximum, let the j^{th} entry from x is selected. If j is $x.p$ then x itself plays the role of u. Else, by induction hypothesis, $(x.sv[j].ts = -1) \vee (\exists u : (u.p = j) \wedge (u \rightarrow x) \wedge u.sv[j] = x.sv[j])$. Hence either $w.sv[j].ts$ is -1 or by transitivity, u happens before w.

The same argument also applies to the case where the j^{th} entry comes from s.

In case of *Rollback*, let s be the state restored and let w be the state resulting from s by taking the actions needed for the correct simulation. By induction hypothesis, s satisfies the lemma. Now $s.sv$ and $w.sv$ differ only in $w.p^{th}$ entry and all states that happened before s also happened before w. Hence w satisfies the lemma. ∎

LP P_i :

type entry = (int inc, int ts)
var sv : array[0..n-1] of entry; /* simulation vector */
 iet : array[0..n-1] of set of entry;
 /* incarnation end table */
 $token$: entry; /* rollback announcement */
Initialize :
 $\forall\ j$: $sv[j].inc$ = 0 ; $sv[j].ts$ = -1 ;
 $sv[i].ts$ = 0 ;
 $\forall\ j$: $iet[j] = \{\}$; /* empty set */
Receive_message(m) :
 if $\exists j, t : ((m.sv[j].inc, t) \in iet[j]) \wedge (t < m.sv[j].ts)$
 then discard m ;
 else if $m.ts < sv[i].ts$ **then**
 /* m is a straggler */
 $token = (sv[i].inc, m.ts)$;
 Broadcast($token$) ;
 /* P_i receives its own broadcast and rolls back. */
 Block till all LPs acknowledge broadcast ;
Execute_message :
 m = messages with the lowest value of $m.ts$;
 $\forall\ j$: $sv[j] = \max(sv[j], m.sv[j])$;
 $sv[i].ts = m.ts$;
 Act on the event scheduled by m ;
Receive_token(v, t) from P_j :
 Send acknowledgement ;
 $iet[j] = iet[j] \cup \{(v, t)\}$;
 $\forall m \in input\_queue$:
 if $(m.sv[j].inc = v) \wedge (t < m.sv[j].ts)$
 then discard m ;
 if $(sv[j].inc = v) \wedge (t < sv[j].ts)$
 then Rollback($j, (v, t)$) ;
Rollback($j, (v, t)$) :
 Save the iet ;
 Restore the latest state s such that
 $sv[j] \le (v, t)$ ...(C1)
 Discard the states that follow s ;
 Restore the saved iet ; $sv[i].inc++$;

Figure 4: Our protocol for distributed simulation

4 The Protocol

Our protocol for distributed simulation is shown in Figure 4. To keep the presentation and correctness proof clear, optimization techniques for reducing overhead are not included in this protocol. They are described in the next section. Besides a simulation vector, each LP P_i also maintains an *incarnation end table* (iet). The j^{th} component of iet is a set of entries of the form (k, ts), where ts is the timestamp of the straggler that caused the rollback of the k^{th} incarnation of P_j. All states of the k^{th} incarnation of P_j with timestamp greater than ts have been rolled back. The iet allows an LP to detect orphan messages.

When P_i is ready for the next event, it acts on the message with the lowest virtual time. As explained in Section 3, P_i updates its SV and the internal state, and possibly schedules events for itself and for the other LPs by sending messages.

Upon receiving a message m, P_i discards m if m is an orphan. This is the case when, for some j, P_i's iet and the j^{th} entry of m's SV indicate that m is dependent on a rolled back state of P_j. If P_i detects that m is a straggler with virtual time t, it broadcasts a token containing t and its current incarnation number k. It rolls back all states with virtual time *greater than* t and increments its incarnation number, as shown in *Rollback*. Thus, the token basically indicates that all states of incarnation k with virtual time greater than t are orphans. States dependent on any of these orphan states are also orphans.

When an LP receives a token containing virtual time t from P_j, it rolls back all states with the j^{th} timestamp greater than t, discards all orphan messages in its input queue, and increments its incarnation number. It does not broadcast a token, which is an important property of our protocol. This works because transitive dependencies are maintained. Suppose state w of P_i is dependent on a rolled back state u of P_j. Then any state x dependent on w must also be dependent on u. So x can be detected as an orphan state when the token from P_j arrives at $P_{x.p}$, without the need of an additional token from P_i. The argument for the detection of orphan messages is similar.

We require an LP to block its execution after broadcasting a token until it receives acknowledgments from all the other LPs. This ensures that a token for a lower incarnation of P_j reaches all LPs before they can become dependent on any higher incarnation of P_j. This greatly simplifies the design because, when a dependency entry is overwritten by an entry from a higher incarnation in the lexicographical maximum operation, it is guaranteed that no future rollback can occur due to the overwritten entry (as the corresponding token must have arrived). While blocked, an LP acknowledges the received broadcasts.

4.1 Proof of Correctness

Suppose state u of P_j is rolled back due to the arrival of a straggler. The simulation is correct if all the states that are dependent on u are also rolled back. The following theorem proves that our protocol correctly implements the simulation.

Theorem 2 *A state is rolled back due to either a straggler or a token. A state is rolled back due to a token if and only if it is dependent on a state that has been rolled back due to a straggler.*

Proof. The routine *Rollback* is called from two places: *Receive_message* and *Receive_token*. States that are rolled back in a call from *Receive_message* are rolled back due to a straggler. Suppose P_j receives a straggler. Let u be one of the states of P_j that are rolled back due to this straggler. In the call from routine *Receive_token*, any state w not satisfying condition (C1) is rolled back. Since the virtual time of u is greater than the virtual time of the straggler, by Lemma 2, any state w dependent on u will not satisfy condition (C1). In the future, no state can become dependent on u because any message causing such dependency is discarded: if it arrives after the token, it is discarded by the first test in the routine *Receive_message*; if it arrives before the token, it is discarded by the first test in the routine *Receive_token*. So all orphan states are rolled back.

From Lemma 3, for any state w not satisfying condition (C1) and thus rolled back, there exists a state u which is rolled back due to the straggler, and $u \rightarrow w$. That means no state is unnecessarily rolled back. ■

5 Reducing the Overhead

For systems with a large number of LP's, the overhead of SV and the delay due to the blocking can be substantial. In this section, we describe several optimization techniques for reducing the overhead and blocking.

5.1 Reducing the blocking

For simplicity, the protocol description in Figure 4 increments the incarnation number upon a rollback due to a token (although it does not broadcast another token). We next argue that the protocol works even if the incarnation number is not incremented. This modification then allows an optimization to reduce the blocking. We use the example in Figure 2(b) to illustrate this modification. Suppose P_2 executes an event and makes a state transition from $r20$ to $s22$ with virtual time 7 (not shown in the figure). If $P2$ does not increment its incarnation number on rolling back due to the token from $P1$, then $s22$ will have $(0, 7)$ as the 3rd entry of its SV, which is the same as $s21$'s 3rd entry in Figure 2(a). Now suppose the 3rd entry of a state w of another LP $P3$ is $(0, 7)$. How does $P3$ decide whether w is dependent on $s21$ which is rolled back or $s22$ which is not rolled back? The answer is that, if w is dependent on $s21$, then it is also dependent on $s11$. Therefore, its orphan status will be identified by its 2nd entry, without relying on the 3rd entry.

The above modification ensures that, for every new incarnation, a token is broadcast and so every LP will have an *iet* entry for it. This allows the following optimization technique for reducing the blocking. Suppose P_i receives a straggler and broadcasts a token. Instead of requiring P_i to block until it receives all acknowledgements, we allow P_i to continue its execution in the new incarnation. One problem that needs to be solved is that dependencies on the new incarnation of P_i may reach an LP P_j (through a chain of messages) before the corresponding token does. If P_j has a dependency entry on any rolled back state of the old incarnation then it should be identified as an orphan when the token arrives. Overwriting the old entry with the new entry via the lexicographical maximum operation results in undetected orphans and hence incorrect simulation. The solution is to force P_j to block for the token before acquiring any dependency on the new incarnation. We conjecture that this blocking at the token receiver's side would be a improvement over the original blocking at the token sender's side if the number of LPs (and hence acknowledgements) is large.

5.2 Reducing the size of simulation vectors

The *Global Virtual Time(GVT)* is the virtual time at a given point in simulation such that no state with virtual time less than GVT will ever be rolled back. It is the minimum of the virtual times of all LPs and all the messages in transit at the given instant. Several algorithms have been developed for computing GVT [2, 20]. To reduce the size of simulation vectors, any entry that has a timestamp less than the GVT can be set to NULL, and NULL entries need not be transmitted with the message. This does not affect the correctness of simulation because: (1) the virtual time of any message must be greater than or equal to the GVT, and so timestamps less than the GVT are never useful for detecting stragglers; (2) the virtual time contained in any token must be greater than or equal to the GVT, and so timestamps less than the GVT are never useful for detecting orphans. Since most of the SV entries are initialized to -1 (see Figure 3) which must be less than the GVT, this optimization allows a simulation to start with very small vectors, and is particularly effective if there is high locality in message activities.

Following [21], we can also use a K-optimistic protocol. In this scheme, an LP is allowed to act on a message only if that will not result in more than K non-NULL entries in its SV. Otherwise it blocks. This ensures that an LP can be rolled back by at most K other LPs. In this sense optimistic protocols are N-optimistic and pessimistic protocols are 0-optimistic.

Another approach to reducing the size of simulation vectors is to divide the LPs into clusters. Several designs are possible. If the interaction inside a cluster is optimistic while inter-cluster messages are sent conservatively [18], independent SV's can be used inside each cluster, involving only the LPs in the cluster. If intra-cluster execution is sequential while inter-cluster execution is optimistic [1], SV's can be used for inter-cluster messages with one entry per cluster. Similarly one can devise a scheme where inter-cluster and intra-cluster executions are both optimistic but employ different simulation vectors. This can be further generalized to a hierarchy of clusters and simulation vectors. In general, however, inter-cluster simulation vectors introduce false dependencies [14] which may result in unnecessary rollbacks. So there is a trade-off between the size of simulation vectors and unnecessary rollbacks. But it does not affect the correctness of the simulation.

6 Distributed Simulation and Distributed Recovery

The problem of failure recovery in distributed systems [6] is very similar to the problem of distributed simulation. Upon a failure, a process typically restores its last checkpoint and starts execution from there. However, process states that were lost upon the failure may create orphans and cause the system state to become inconsistent. A *consistent* system state is one where the send of a message must be recorded if its receive is recorded [6]. In *pessimistic logging* [6], every message is logged before the receiver acts on it. When a process fails, it restores its last checkpoint and replays the logged messages in the original order. This ensures that the pre-failure state is recreated and no other process needs to be rolled back. But the synchronization between message logging and message processing reduces the speed of computation. In *optimistic logging* [19], messages are stored in a volatile memory buffer and logged asynchronously to the stable storage. Since the content of volatile memory is lost upon a failure, some of the messages are no longer available for replay after the failure. Thus, some of the process states are *lost* in the failure. States in other processes that are dependent on these lost states then become *orphan states*. Any optimistic logging protocol must roll back all orphan states in order to bring the system back to a consistent state.

There are many parallels between the issues in distributed recovery and distributed simulation. A survey of different approaches to distributed recovery can be found in [6]. In Table 1, we list the equivalent terms from these two domains. References are omitted for those terms that are widely used. The equivalence is exact in many cases, but only approximate in other cases.

Stragglers trigger rollbacks in distributed simulation, while failures trigger rollbacks in distributed recovery. Conservative simulation [7] ensures that the current state will never need to roll back. Similarly, pessimistic logging [6] ensures that the current state is always recoverable after a failure. In other words, although a rollback does occur, the rolled back states can always be reconstructed.

The time warp optimistic approach [10] inspired the seminal work on optimistic message logging [19]. The optimistic protocol presented in this paper is based on the optimistic recovery protocol presented in [4, 21]. In the simulation scheme by Dickens and Reynolds [5], any results of an optimistically processed event are not sent to other processes until they become *definite* [3]. In the recovery scheme by Jalote [11], any messages originating from an *unstable* state interval are not sent to other processes until the interval becomes *stable* [6]. Both schemes confine the loss of computation, either due to a straggler or a failure, to the local process.

Distributed Simulation	Distributed Recovery
Logical Process	Recovery Unit [19]
Virtual Time	State Interval Index
Sim. Vector (this paper)	Trans. Dep. Vector [19]
Straggler	Failure
Anti-Message	Rollback Announcement
Fossil Collection [10]	Garbage Collection [6]
Global Virtual Time [2]	Global Recovery Line [6]
Conservative Schemes	Pessimistic Schemes
Optimistic Schemes	Optimistic Schemes
Causality Error	Orphan Detection
Cascading Rollback [15]	Domino Effect [6]
Echoing [15]	Livelock [6]
Conditional Event [3]	Unstable State [6]
Definite Event [3]	Stable State [6]

Table 1: Parallel terms from Distributed Simula-

tion and Recovery

Conservative and optimistic simulations are combined in [1, 18] by dividing LPs into clusters and having different schemes for inter-cluster and intra-cluster executions. In distributed recovery, the paper by Lowry et al. [14] describes an idea similar to the conservative time windows in the simulation literature [7].

Now we list some of the main differences between the two areas. While the arrival of a straggler can be prevented, the occurrence of a failure cannot. But pessimistic logging can cancel the effect of a failure through message logging and replaying. The arrival of a straggler in optimistic simulation does not cause any loss of information, while the occurrence of a failure in optimistic logging causes volatile message logs to be lost. So some recovery protocols have to deal with "lost in-transit message" problem [6] which is not present in distributed simulation protocols. Incoming messages from different channels can be processed in an arbitrary order, while event messages in distributed simulation must be executed in the order of increasing timestamps. Due to these differences, some of the protocols presented in one area may not be applicable to the other area.

Distributed recovery can potentially benefit from the advances in distributed simulation in the areas of memory management [13], analytical modeling to determine checkpoint frequency [8], checkpointing mechanisms [22], and time constrained systems [9]. Similarly, research work on coordinated checkpointing, optimal checkpoint garbage collection, and dependency tracking [6] can potentially be applied to distributed simulation.

References

[1] H. Avril and C. Tropper. Clustered Time Warp and Logic Simulation. *Proc. 9th Workshop on Parallel and Distributed Simulation*, 112-119, 1995.

[2] S. Bellenot. Global Virtual Time Algorithms. *Proc. Multiconference on Distributed Simulation*, 122-127, 1990.

[3] K. M. Chandy and R. Sherman. The Conditional Event Approach to Distributed Simulation. *Proc. SCS Multiconference on Distributed Simulation*, 93-99, 1989.

[4] O. P. Damani and V. K. Garg. How to Recover Efficiently and Asynchronously when Optimism Fails. *Proc. 16th IEEE Intl. Conf. Distributed Computing Systems*, 108-115, 1996.

[5] P. M. Dickens and P. F. Reynolds Jr. SRADS with Local Rollback. *Proc. SCS Multiconference on Dist. Simulation*, 161-164, 1990.

[6] E. N. Elnozahy, D. B. Johnson and Y. M. Wang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *Tech. Rep. No. CMU-CS-96-181, Dept. of Computer Science, Carnegie Mellon University*, ftp://ftp.cs.cmu.edu/user/mootaz/papers/S.ps, 1996.

[7] R. Fujimoto. Parallel Discrete Event Simulation. *Comm. ACM*, 33(10), 30-53, Oct. 1990.

[8] J. Fleischmann and P. A. Wilsey. Comparative Analysis of Periodic State Saving Techniques in Time Warp Simulators. *Proc. 9th Workshop on Parall. and Dist. Simulation.*, 50-58, 1995.

[9] K. Ghosh, R. M. Fujimoto, and K. Schwan. Time Warp Simulation in Time Constrained Systems. *Proc. 7th Workshop on Parallel and Distributed Simulation*, 163-166, 1993.

[10] D. R. Jefferson. Virtual Time. *ACM Trans. Prog. Lang. and Sys.*, 7(3), 404-425, 1985.

[11] P. Jalote. Fault Tolerant Processes. *Distributed Computing*, 3, 187-195, 1989.

[12] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, vol. 21, no. 7, 558-565, 1978.

[13] Y. B. Lin. Memory Management Algorithms for Optimistic Parallel Simulation. *Proc. 6th Workshop on Parallel and Distributed Simulation*, 43-52, 1992.

[14] A. Lowry, J. R. Russel and A. P. Goldberg. Optimistic Failure Recovery for Very Large Networks. *Proc. Proc. 10th IEEE Symp. on Reliable Distributed Systems*, 66-75, 1991.

[15] B. D. Lubachevsky, A. Schwartz, and A. Weiss. Rollback Sometimes Works ... if Filtered. *Proc. 1989 Winter Simulation Conference*, 630-639, 1989.

[16] F. Mattern. Virtual Time and Global States of Distributed Systems. *Parallel and Distributed Algorithms: Proc. of the Intl. Workshop on Parallel and Distributed Algorithms, Elsevier Science Publishers B. V.(North Holland)*, 215-226, 1989.

[17] A. Prakash and R. Subramanian. An Efficient Optimistic Distributed Simulation Scheme Based on Conditional Knowledge. *Proc. 6th Workshop on Parallel and Dist. Simulation*, 85-94, 1992.

[18] H. Rajaei, R. Ayani, and L. E. Thorelli. The Local Time Warp Approach to Parallel Simulation. *Proc. 7th Workshop on Parallel and Distributed Simulation*, 119-126, 1993.

[19] R. E. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Trans. on Computer Systems*, 204-226, August 1985.

[20] A. I. Tomlinson and V. K. Garg. An Algorithm for Minimally Latent Global Virtual Time. *Proc. 7th Workshop on Parallel and Distributed Simulation*, 35-42, 1993.

[21] Y. M. Wang, O. P. Damani, and V. K. Garg. Distributed Recovery with K-Optimistic Logging. To appear in *Proc. the 17th IEEE Intl. Conf. Distributed Computing Systems*, 1997.

[22] D. West and K. Panesar. Automatic Incremental State Saving. *Proc. 10th Workshop on Parallel and Distributed Simulation*, 78-85, 1996.