JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 16, 243-269 (2000)

Design Issues for Optimistic Distributed Discrete Event Simulation

YI-BING LIN
Department of Computer Science and Information Engineering
National Chiao Tung University
Hsinchu, Taiwan 300, R.O.C.
E-mail: liny@csie.nctu.edu.tw

Simulation is a powerful tool for studying the dynamics of a system. However, simulation is time-consuming. Thus, it is natural to attempt to use multiple processors to speed up the simulation process. Many protocols have been proposed to perform discrete event simulation in multi-processor environments. Most of these distributed discrete event simulation protocols are either *conservative* or *optimistic*. The most common optimistic distributed simulation protocol is called *Time Warp*. Several issues must be considered when designing a Time Warp simulation; examples are reducing the state saving overhead and designing the global control mechanism (i. e., global virtual time computation, memory management, distributed termination, and fault tolerance). This paper addresses these issues. We propose a heuristic to select the checkpoint interval to reduce the state saving overhead, generalize a previously proposed global virtual time computation algorithm, and present new algorithms for memory management, distributed termination, and fault tolerance. The main contribution of this paper is to provide guidelines for designing an efficient Time Warp simulation.

Keywords: discrete event simulation, distributed systems, fault tolerance, memory management, time warp

1. INTRODUCTION

A discrete event simulation consists of a series of events, along with times when they occur. Execution of any event can give rise to any number of events with later timestamps. All of this is straightforward in implementation if there is a centralized system with one event queue: we just execute the earliest not-yet-executed event next.

Since simulation is time-consuming, it is natural to attempt to use multiple processors to speed up the simulation process. In *distributed discrete event simulation* (or *distributed simulation*), the simulated system is partitioned into a set of sub-systems that are simulated by a set of processes that communicate by sending/receiving timestamped messages. The scheduling of an event for a sub-system at time t is simulated by sending a message with timestamp t to the corresponding process. The global event list and global clock of a sequential simulation do not exist in the distributed counterpart. Each process has its own input message queue and local clock. To correctly simulate a sub-system, the corresponding process must execute arriving messages in their timestamp order, as opposed to their real-time arrival order. To satisfy this causality constraint, a synchronization mechanism is

Received July 27, 1998; accepted November 29, 1999.
Communicated by Chyi-Nan Chen.

required. One of the most common synchronization protocols for distributed simulation is called *Time Warp* [12] (Different approaches to distributed simulation are discussed elsewhere [8, 15, 19, 22, 28, 31, 32].)

The Time Warp protocol takes an optimistic approach in which a process executes every message[1] as soon as it arrives. If a message with an earlier timestamp subsequently arrives (called a *straggler*), the process must roll back its state to the time of the straggler and re-execute from that point. To support rollback, several data structures are maintained in a process:

- *Input queue*: the set of all messages which have recently arrived. These messages are sorted in their timestamp order. Some of them may have been processed.
- *Local clock*: the timestamp of the message being processed. If all the messages in the input queue have been processed, then the local clock is set to ∞.
- *Output queue*: the set of negative copies (i.e., antimessages) of the positive messages the process has recently sent. An *antimessage* of a message m is exactly like m in format and content except in one field: its sign. Two messages that are identical except for opposite signs are said to be antimessages of one another.
- *State queue*: copies of the process's recent states.

When a message arrives at a process p_i with a timestamp no less than the local clock, it is inserted in the input queue. Process p_i executes messages in the input queue in their timestamp order. Let $ts(m)$ be the timestamp of a message m. Suppose that the scheduling of a message m is due to the execution of another message m_0. Then the *send time* of m (denoted as $ts'(m)$) is defined as the timestamp of m_0. In other words, $ts'(m) = ts(m_0)$. Since the execution of an event always schedules events with later timestamps, we have $ts'(m) = ts(m_0) < ts(m)$. When m is executed, the following steps are performed: (i) The local clock advances to $ts(m)$. (ii) Message m is processed. If any message m' is scheduled to another process p_j during the execution, the antimessage of m' is also created. The positive message is sent to p_j, and the negative copy is inserted in the output queue in send time order. (iii) The new process state after the execution is added in the state queue. (Step (iii) is not necessarily performed for every event execution. However, the state of the process must be saved regularly.)

If a straggler subsequently arrives, the process must roll back its state to the time of the straggler and re-execute from that point. Consider the example shown in Fig. 1. A horizontal line represents the progress of a process in simulation time, and a dashed arrow represents sending a message. When p_i's local clock is τ, it receives a message m with timestamp $\tau' < \tau$. Thus, p_i is rolled back to the timestamp τ'. During the roll back computation, p_i may have sent messages to other processes (cf. message f in Fig. 1). These output messages are potentially *false messages* because their timestamps are greater than the local clock. (A false message does not exist in the sequential simulation. That is, the message does not have any effect on the simulation and must be cancelled if it is created in Time Warp simulation.) The cancellation of a potential false message f sent from p_i to p_j is done by sending the corresponding negative message $\overline{f}$ (which is stored in the output queue of p_i

[1] A message consists of six fields: *send time*, *timestamp* (or *receive time*), *sender, receiver, sign*, and *text*. We will elaborate on these fields later.

when f is sent). Once p_j receives $\overline{f}$, it discards f and any effect caused by f. One may assume that all output messages f generated during the roll back computation are false, and negative messages $\overline{f}$ are immediately sent to cancel these output messages at the time the rollback occurs. This is called *aggressive cancellation*. On the other hand, one may assume that all messages sent during the roll back computation are *true* (i.e., not false), and are not cancelled at the time the rollback occurs. After the rollback, new messages m_1, m_2, ... will be generated. Negative messages $\overline{f}$ need only be sent for potential false messages that are not regenerated (i.e., $f \neq m_i$). Depending on the application, lazy cancellation may outperform aggressive cancellation or vice versa. Guidelines for designing the rollback mechanism can be found elsewhere [21, 30].

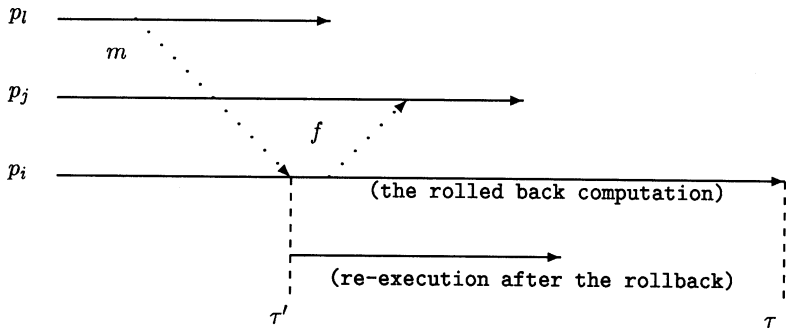


Fig.1. Effect of Rollback. A horizontal line represents the progress of a process in simulation time, and a dashed arrow represents sending a message.

Besides the rollback (local control) mechanism, a global control mechanism for Time Warp is required. The central concept of the global control mechanism is *global virtual time* (GVT). Let an *unprocessed* message be a message in the input queue of a process that has not yet been executed. GVT is defined as follows.

Definition 1: GVT at time t (denoted as GVT(t)) is the minimum of (i) the values of all local clocks at time t, (ii) the timestamps of all unprocessed messages, and (iii) the send times[2] of all transient messages.

At any real time, there exists a global virtual time GVT such that all executed messages with timestamps earlier than GVT will not be rolled back. Based on GVT, the global control mechanism addresses several critical issues, such as garbage collection, distributed termination, and fault tolerance.

This paper concentrates on reducing the state saving overhead and the design issues for the Time Warp global control mechanism. The paper is organized as follows. Section 2 derives a heuristic used to select the checkpoint interval to reduce the state saving overhead. Section 3 generalizes a previously proposed GVT computation algorithm. Sections 4-6 present new algorithms for memory management, distributed termination, and fault tolerance.

[2] In some studies [8, 18], the timestamps of unprocessed messages are considered in computing GVT, instead of their send times. This paper follows the original definition of GVT given by Jefferson [12].

2. REDUCING THE STATE SAVING OVERHEAD

In a Time Warp simulation, the state of each process must be saved regularly (regardless of whether or not rollbacks actually occur). Lin and Lazowska [20] have indicated that the performance of Time Warp is dominated by the efficiency of state saving. Thus, it is important to reduce the state saving overhead.

There are two approaches to reducing the state saving overhead. One approach is to accelerate the state saving process. Fujimoto et al. [9] developed special-purpose hardware to support fast state saving. A complementary approach is to reduce the frequency of state saving. This section pursues the second approach. We give a heuristic to select the checkpoint interval to reduce the state saving overhead and report on the confirmation of our results in an experimental study conducted by Preiss et al. [29].

From a model similar to that in [11, 35, 37], we derive bounds for the optimal checkpoint interval χ_{opt}. Note that our derivations are based on several simplifying assumptions. Thus, the term "optimal" means the best possible choice of the parameter χ_{opt}, subject to our assumptions.

Consider a process p in a Time Warp simulation. We assume that there is no message preemption, and that a state saving operation occurs atomically with the completion of the checkpointed event. We refer to the interval between two consecutive rollbacks of p as a *computation cycle*. Suppose every χ message executions are followed by a state saving event (i.e., the execution of the χth event is *checkpointed*). χ is called the *checkpoint interval*. Consider the ith computation cycle of length $R_{\chi,i}$ as shown in Fig.2.

In this figure, a solid circle represents a state saving event. After $R_{\chi,i}$ events have been executed, a straggler arrives, which undoes $\beta_{\chi,i}$ events. However, it is necessary to roll back to the first checkpoint (but not include the checkpoint itself) prior to the timestamp of the straggler, and an extra $\gamma_{\chi,i}$ event must be re-executed to restore the current process state. For convenience of analysis, the executions of these events are assumed to be in the i + 1st computation cycle. A computation cycle i consists of three parts:

- rollback and restoring the current process state by re-executing $\gamma_{\chi,i-1}$ events (cf. Fig. 2),
- forward executions (the executions of $\alpha_{\chi,i}$ events in Fig. 2),
- and periodic checkpointing (cf. the solid circles in Fig. 2).

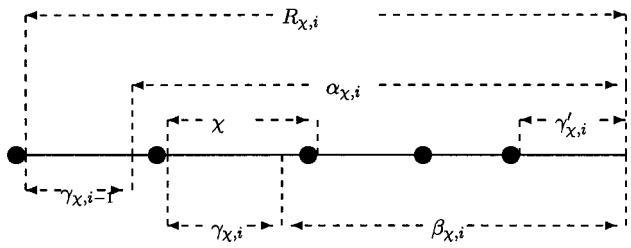


Fig. 2. The ith computation cycle.

Thus, $R_{\chi,i}$ can be expressed as

$$R_{\chi,i} = \alpha_{\chi,i} + \gamma_{\chi,i-1}, \quad \text{for } i \geq 1 \text{ and } \gamma_{\chi,0} = 0. \tag{1}$$

Note that $\alpha_{\chi,i}$ may be negative (i.e., a straggler message arrives before all $\gamma_{\chi,i-1}$ messages have been executed), but $\alpha_{\chi,i} + \gamma_{\chi,i-1}$ is always positive. In our model, the state saving overhead $\Delta_{\chi,i}$ for the ith computation cycle is defined as the overhead needed to re-execute the $\gamma_{\chi,i-1}$ events in the previous computation cycle plus the overhead of the state saving operations done among $\gamma_{\chi,i-1} + \alpha_{\chi,i}$ events. Let δ_s be the overhead of saving a process state, assumed to be a constant. Let $\delta_{i,j}$ be the execution time of the jth event ($j \le R_{\chi,i}$) in the ith computation cycle. Assume that the process state is checkpointed before the first event is executed, then the state saving overhead, $\Delta_{\chi,i}$, of the ith computation cycle is

$$\Delta_{\chi,i} = \sum_{j=1}^{\gamma_{\chi,i-1}} \delta_{i,j} + \begin{cases} \left\lceil \dfrac{\alpha_{\chi,i}}{\chi} \right\rceil \delta_s, & i = 1 \\ \left\lceil \dfrac{\alpha_{\chi,i} + \gamma_{\chi,i-1}}{\chi} \right\rceil \delta_s, & \text{otherwise.} \end{cases} \tag{2}$$

Equation (2) consists of two components. The first component represents the overhead for restoring the current process state (when $i = 1$, $\gamma_{\chi,i-1} = 0$ and the component does not exist). The second component represents the overhead for periodic checkpointing. This equation holds whether $\alpha_{\chi,i} \ge \beta_{\chi,i}$ or $\alpha_{\chi,i} < \beta_{\chi,i}$ ($\alpha_{\chi,1} \ge \beta_{\chi,1}$, according to the definition of a Time Warp simulation). Let k_χ be the number of rollbacks that occur in a process when the checkpoint interval is χ. Let $\Delta_\chi = E\left[\sum_{i=1}^{k_\chi} \Delta_{\chi,i}\right]$. Then Δ_χ can be expressed as $\Delta_\chi = \Delta_\chi^C + \Delta_\chi^U$, where Δ_χ^C is the time devoted to periodic checkpointing that Δ_χ^U is the time devoted to re-executing the extra undone events needed to restore the current process state. Our goal is to choose an optimal χ value which minimizes the net effect Δ_χ. We first derive a lower bound Δ_χ^- and an upper bound Δ_χ^+ for Δ_χ with a fixed checkpoint interval. Let C_χ be the number of checkpoints required in a process. From (2),

$$C_\chi = \left\lceil \frac{\alpha_{\chi,1}}{\chi} \right\rceil + \sum_{i=2}^{k_\chi} \left\lfloor \frac{\alpha_{\chi,i} + \gamma_{\chi,i-1}}{\chi} \right\rfloor .$$

Assume that the behavior of the system is not affected by the checkpoint interval; that is, for all χ, the process states are the same at the end of a computation cycle. This assumption is reasonable when the system being simulated is homogeneous. However, it does not reflect the real world in general. Thus, our solution should be considered as a heuristic for checkpoint interval selection. From this assumption, we have

$$k_\chi = k_1 = k, \quad \text{and for all } i \qquad \alpha_{\chi,i} = \alpha_{1,i} = \alpha_i \qquad \text{and} \quad \beta_{\chi,i} = \beta_{1,i} = \beta_i. \tag{3}$$

Let N_χ be the number of events (including the roll back events) executed in a process when the checkpoint interval is χ. Since $\gamma_{1,i} = 0$, (1) and (3) imply that $\alpha_{\chi,i} > 0$, and

$$N_\chi = N_1 + \sum_{j=1}^{k} \gamma_{x,j} . \tag{4}$$

It is not possible to derive Δ_χ without knowing the distributions of α_i and β_i. Lacking knowledge of these distributions, we will instead derive bounds for Δ_χ.

To derive a lower bound for Δ_x, consider the ith computation cycle in Fig. 2. Execution of every x events requires a checkpoint, except for the last $\gamma'_{\chi,i}$ events. (The process is checkpointed at the beginning; then, every x event executions are followed by a checkpointing operation.) In other words, the execution of the last $\gamma'_{\chi,i}$ events does not incur checkpointing overhead. Thus, the best case occurs when $\gamma'_{\chi,i} = \chi - 1$. Then, C_x is bounded below as

$$C_\chi \geq \frac{N_\chi - (\chi-1)k}{\chi} = \frac{N_\chi + k}{\chi} - k. \tag{5}$$

If we assume that $\gamma_{x,i}$ is uniformly distributed in [0, x − 1], then a tighter bound can be obtained. Let $\overline{\alpha} = \frac{N_1}{k}$, and let δ be the expected event execution time. Then from (2), (4) and (5), a lower bound Δ^-_χ for Δ_x is derived as follows:

$$\begin{aligned}\Delta_\chi &\geq \frac{(\chi-1)k\delta}{2} + \left[\frac{N_1 + \frac{1}{2}(\chi-1)k + k}{\chi} - k\right]\delta_s \\ &= \frac{k}{2}\left[(\chi-1)\delta + \left(\frac{2\overline{\alpha}+1}{\chi} - 1\right)\delta_s\right] = \Delta^-_\chi.\end{aligned} \tag{6}$$

The checkpoint interval that minimizes (6) is

$$\chi^+ = \left\lfloor \sqrt{\frac{(2\overline{\alpha}+1)\delta_s}{\delta}} \right\rfloor \tag{7}$$

It is interesting to note that (7) is almost identical to Young's result [37] under the proper interpretation of the parameters. To derive an upper bound for Δ_x, let $\gamma'_{\chi,i} = 0$ and $\gamma_{x,i} = x - 1$. Then the number of checkpoints required in a process is bounded above by

$$C_\chi \leq \frac{N_\chi}{\chi} \leq \frac{N_1 + (\chi-1)k}{\chi}. \tag{8}$$

From (2) and (8), an upper bound Δ^+_χ for Δ_n is derived as:

$$\begin{aligned}\Delta_\chi &\leq k(\chi-1)\delta + \frac{N_1 + (\chi-1)k}{\chi}\delta_s \\ &= k\left[(\chi-1)\delta + \frac{\overline{\alpha} + (\chi-1)}{\chi}\delta_s\right] = \Delta^+_\chi.\end{aligned} \tag{9}$$

The checkpoint interval that minimizes (9) is

$$\chi^- = \left\lfloor \sqrt{\frac{(\overline{\alpha}-1)\delta_s}{\delta}} \right\rfloor.$$

Fig. 3 plots the curves for Δ^-_χ and Δ^+_χ. The functions for both Δ^+_χ and Δ^-_χ are of the form $f(x) = f_1(x)f_2(x)$, where $f_1(\chi) = \frac{1}{\chi}$ and $f_2(x) = ax^2 + bx + c$.

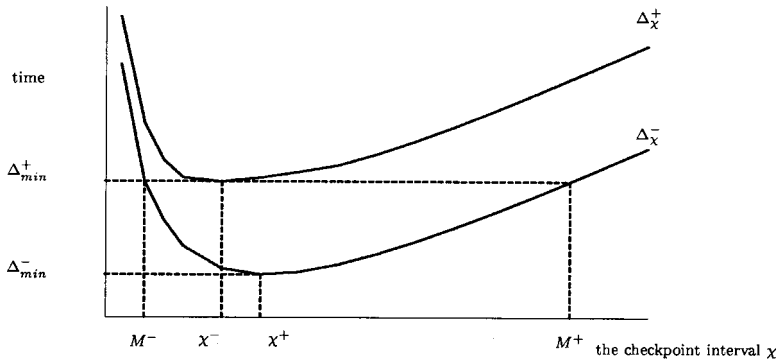


Fig. 3. Bounds for Δ_x.

Since $f_2(x)$ is a parabola, the effect of $f_1(x)$ makes $f(x)$ decrease quickly as x increases before the minimum is reached (in Fig. 3, the minima are $\Delta^+_{\min} = \Delta^+_{\chi-}$ and $\Delta^-_{\min} = \Delta^-_{\chi+}$), and increase slowly as x increases after the minimum is reached. From Fig. 3, the optimal checkpoint interval x_{opt} is bounded by $M^- \le x_{opt} \le M^+$, where M^- and M^+ are the roots of the equation $\Delta^-_\chi = \Delta^+_{\min}$. Note that $M^- \le x^- \le x^+ \le M^+$ and $M^+ - x^+ \ge x^+ - M^-$, and

$$M^+ \to \chi^+, M^- \to \chi^-, \text{ as } \Delta^+_\chi + \to \Delta^-_\chi. \tag{10}$$

Since the derivations for M^- and M^+ are more complicated than the derivations for x^- and x^+, it is more practical to determine x_{opt} in terms of x^- and x^+. As indicated by (9), this is a good approximation when $\Delta^+_\chi \to \Delta^-_\chi$.

We make the following observations.

- If the execution times of events are random variables, then Δ_x is affected by the mean of the execution times but is insensitive to the distribution of the execution times. This is derived from the strong law of large numbers.
- For a fixed x, the state saving overhead Δ_x decreases as $\overline{\alpha}$ increases. This implies that reducing the state saving overhead is important for simulation with small $\overline{\alpha}$.
- Erring on the side of a x value that is too large will degrade performance less than erring on the side of a x value that is too small.
- A large x should be chosen if (i) $\frac{\delta_s}{\delta}$ is large, and/or (ii) $\overline{\alpha}$ is large. Intuitively, if δ_s is small (compared with δ), then Δ^C_χ only has an insignificant contribution to Δ_x. Thus, a small x should be chosen to minimize Δ^U_χ. For a large $\overline{\alpha}$, Δ^C_χ has a more significant effect on Δ_x than Δ^U_χ does. Thus, a large x should be chosen to reduce Δ^C_χ. When $\overline{\alpha} = \infty$ (i.e., no rollback occurs at process p), no state saving is required, and $x = \infty$ should be selected.

Preiss et al. [29] have conducted experiments to study our simple heuristic. In their experiments, the execution times of simulation were measured instead of the state saving overhead Δ_x. Interestingly, the curves for the execution times have the same shapes as do the curves in Fig. 3. The experiments show that for the *round-robin* processor scheduling

policy [3] and for both aggressive cancellation and lazy cancellation, the optimal checkpoint intervals fall in the interval $[x^-, x^+]$ or are slightly higher. For the *smallest timestamp first* policy, [4] the optimal checkpoint intervals are larger than x^+. Although M^+ was not known in the experiments, we believe that the optimal checkpoint intervals all fall in $[x^+, M^+]$. The experiments also indicated that $\Delta_{x_1} \simeq \Delta_{x_2}$ for $x_1, x_2 \in [x^+, x_{opt}]$. Thus, this experimental study concluded that x^+ is a good predictor for the optimal checkpoint interval. We note that in these experiments, the parameters $\overline{\alpha}$, δ_s and δ were obtained after the simulation run finished. A large issue is the fact that these parameters are seldom known ahead of time and are a function of the application characteristics. One way to obtain the values for the parameters is to compute these parameters during the simulation and update the values dynamically. Several issues about the use of our simple heuristic for reducing the state saving overhead are being investigated at the Jet Propulsion Laboratory [2, 33]. Bellenot's experiments indicated that our heuristic is less accurate as the number of processors available increases. The reason is simple. The derivation of our heuristic assumes that processors are not idle during the simulation. (This assumption is generally true when the simulated system is large.) As the number of processors increases, the number of idle processors also increases, which may invalidate our assumption. Thus, we conclude that the heuristic is useful when the size of the simulated system is large.

3. GLOBAL VIRTUAL TIME COMPUTATION

Time Warp requires a global control mechanism to address several critical problems such as memory management, distributed termination and fault tolerance. The central concept behind the global control mechanism is based on GVT. Since GVT is smaller than the timestamp of every unprocessed message in the system, we have the following theorem.

Theorem 1 [12]**:** At time *t*, no event with a timestamp earlier than GVT(t) can be rolled back, and such events may be irrevocably committed with safety.

Jefferson [12] showed that GVT is a non-decreasing function of time, which guarantees global progress of the Time Warp simulation.

In a shared memory multiprocessor environment, GVT can be easily computed [7]. On the other hand, GVT cannot be easily obtained in a fully distributed environment where messages might not be delivered in the order they are sent. (In such an environment, the transient messages in the system cannot be directly accessed.) In most approaches [3, 18, 27, 34], the task of finding GVT involves all the processes in the system. One of the processes, called the *coordinator*, is assigned to initiate the task. The coordinator broadcasts a STARTGVT message to all processes. Then every process computes its local minimum (to be defined), and reports this value to the coordinator. When all the local minima are received, the coordinator computes the minimum among these local minima, and GVT is found.

---

[3] In the round-robin policy, the processes that are ready to execute (i.e., the processes that have messages to process) are allowed to process messages in round-robin fashion one at a time.

[4] In the smallest timestamp first policy [23] or the *minimum message timestamp policy* [29], a process with smaller timestamp event (i.e., the next event to be executed in the process has a smaller timestamp) has higher priority for execution.

In a GVT algorithm, *control messages* (such as STARTGVT) are sent to perform GVT computation. These messages are distinguished from *data messages*, the normal messages sent in Time Warp simulation. Most GVT algorithms are based on Samadi's approach [34]. In this approach, when process q receives a data message from p, it needs to send an acknowledgement back to p. We have proposed a GVT algorithm [18] that does not require acknowledgements for data messages. Thus, this algorithm can eliminate about 50% of the message sending during the simulation. However, the algorithm assumes that the set of processes that send messages to a process is pre-defined. This section removes this restriction.

Consider a pair of processes p and q. A message has a sequence number i if it is the ith message (denoted as m_i) sent from a process p to a process q. Consider an example of a send time histogram of messages sent from p to q in the time interval $[0, t]$ as shown in Fig. 4. Note that a message with a larger sequence number may have a smaller send time due to a rollback in p. Let a *valley* be a message m_i such that $i = 1$ or $ts'(m_i) < ts'(m_{i-1})$, $i > 2$. (Thus, m_1 is always a valley.) In Fig. 4, the set of valleys is $\{m_1, m_{21}, m_{50}, m_{72}\}$. The message with the smallest send time is among these valleys. Thus, to find the minimal send time of messages in transit, we only need to consider those valleys in transit.

To obtain the minimal timestamp of transient messages, a set $V_p(q)$ is maintained in process p to record the send time information of valleys that have been sent from p to q. If we represent a valley m as a (sn, ts') pair, where sn and ts' are the sequence number and send time of m, respectively, then the set $V_p(q)$ for the example in Fig. 4 is

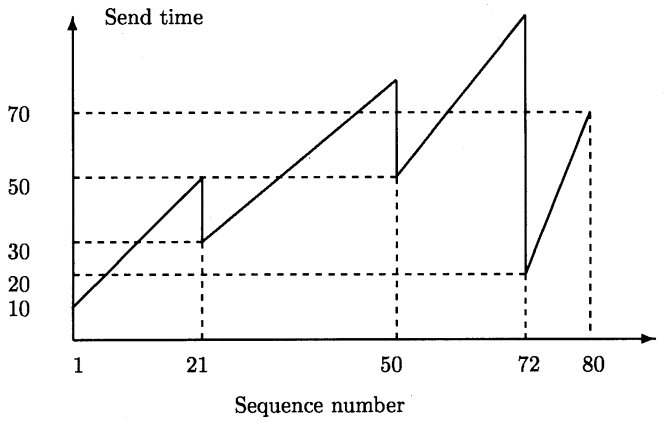


Fig. 4. Send time as a function of sequence number.

$$V_p(q) = \{(1,10), (21,30), (50,50), (72,20), (80,70)\}.$$

When process p sends a message m to process p, it checks if m is a valley. If it is, $V_p(q)$ is updated to reflect the sending of m.

In process q, a set $SN_q(p)$ is used to record the (sequence number) ranges of messages which are sent from p and have been received by q. An element (i.e., a range) in $SN_q(p)$ is of the form (sn_f, sn_l), where sn_f is the sequence number of the first message in the range, and sn_l is the sequence number of the last message in the range. We note that the sequence number holes in $SN_q(p)$ are due to the non-FIFO communication property in the distributed environment. If the messages received by q are those with sequence numbers 1, 2, ..., 20, 32, 33, ..., 67, 69, 70, ..., 108, and 120, 121, ..., 131, then

$$SN_q(p) = \{(1,20), (32,67), (69,108), (120,131)\}.$$

When process q receives a message m from process p, $SN_q(p)$ is updated to reflect receipt of m.

Now, we will describe GVT computation. Consider a process p. Let the *input set* of p (denoted as $S_I(p)$) be the set of processes that send messages to p, and let the *output set* of p (denoted as $S_O(p)$) be the set of processes that receive messages from p. Two types of *control messages* (STARTGVT and TYPE1) are sent in the GVT computation. The coordinator p_0 initiates GVT computation by broadcasting a message STARTGVT to every process. When process p receives the message STARTGVT, it enters the following phase.

Phase 1. In this phase, p (i) computes lm_p, the minimum of p's local clock and the minimum send times of all unprocessed messages in p's input queue, and (ii) sends a message msg = (TYPE1, $sn_{q,p}$) to every process $q \in S_I(p)$, where

$$sn_{q,p} = \min_{(sn_f, sn_l) \in SN_q(p)} sn_l + 1$$

is the smallest sequence number of the transient message sent from q to p.
When p receives the first TYPE1 message from another process, it enters the following phase.

Phase 2. In this phase, p waits to receive a message msg = (TYPE1, $sn_{p,q}$) from every $q \in S_O(p)$. Using $sn_{p,q}$, p locates the smallest send time $\tau_{p,q}$ of the transient messages from p to q by searching the set $V_p(q)$. That is,

$$\tau_{p,q} = \min_{(sn,ts') \in V_p(q), sn \geq sn_{p,q}} ts'.$$

After p has received all the TYPE1 messages, it computes $\tau_p = \min_{q \in S_0(p)} \tau_{p,q}$.
After p has completed both Phase 1 and Phase 2, it reports to the coordinator the *local minimum*

$$LM_p = \min(lm_p, \tau_p).$$

After the coordinator has received all LM_p, it computes

$$GVT = \min_{\forall p} LM_p \tag{11}$$

The time when a process p enters Phase 1 may affect the correctness of the GVT computation. Suppose that process p enters Phase 1 at time t_p. Due to the effect of rollback, if $t_p \neq t_q$, it is possible that $\max_{\forall p} LM_p > GVT(t_M)$, where $t_M = \max_{\forall p} t_p$ (i.e., (11) does not compute a lower bound for GVT). In Appendix A, we show that this problem is avoided if every process enters Phase 1 before Phase 2.

Theorem 2: Suppose that process p enters Phase 1 at time t_p, and $t_M = \max_{\forall p} t_p$. If every process enters Phase 1 before Phase 2, then $\min_{\forall p} LM_p \leq GVT(t_M)$.

Thus, the GVT algorithm for a process p is described as follows:

- If p receives the STARTGVT message before any TYPE1 message, it enters Phase 1. After p has sent all the TYPE1 messages, it enters Phase 2.
- If p receives any TYPE1 message *msg* before the STARTGVT message, then p enters Phase 1. After p completes Phase 1, it enters Phase 2 and processes *msg*. (In Phase 2, p ignores the arrival of the late message STARTGVT.)

In the above algorithm, the sets $S_I(p)$ and $S_O(p)$ are pre-defined. In a practical Time Warp simulation, both sets may change from time to time. To accommodate this situation, we assume that $S_I(p)$ and $S_O(p)$ are updated dynamically, i.e., $S_I(p) \leftarrow S_I(p) \cup \{q\}$ ($S_O(p) \leftarrow S_O(p) \cup \{q\}$) when p receives (sends) the first data message from (to) q. With the dynamic input/output sets, the algorithm just described may fail in the following scenario: Suppose that process q completes Phase 1 before it receives the first data message from p. Then p may expect to receive a TYPE1 message from q and never exit from Phase 2. This problem is solved by introducing a new control messages of type TYPE2: When p enters Phase 1, it also sends a TYPE2 message to every process $q \in S_O(p)$. If (i) q has already completed Phase 1 when it receives a TYPE2 message from p, and if (ii) q did not send a TYPE1 message to p during Phase 1, then q sends *msg* = (TYPE1, 0) to p. (The message *msg* tells p that q did not receive any message from p when q entered Phase 1.) Thus, the complete algorithm is described as follows.

A process p enters the GVT computation mode when it receives the first message *msg* of type STARTGVT, TYPE1 or TYPE2. Upon receipt of *msg*, the following procedure is executed:

$LM_p = lm_p$, $S_{GVT,I}(p) = S_I(p)$, $S_{GVT,O}(p) = S_O(p)$;
for all $q \in S_{GVT,I}(p)$ **do send** a message (TYPE1, lm_p, $sn_{q,p}$) to q;
for all $q' \in S_{GVT,O}(p)$ **do send** a message (TYPE2) to q';

Then p enters (modified) Phase 2 and handles the message *msg*: If *msg* = (TYPE1, $sn_{p,q'}$), then compute $\tau_{p,q'}$ and

$LM_p = \min(LM_p, \tau_{p,q'})$.

If *msg* = (TYPE2) is from $q' \notin S_{GVT,I}(p)$, then p sends a message (TYPE1, 0) to q'. After p has received all the TYPE1 messages from processes in $S_{GVT,O}$, the value LM_p is sent to p_0. (Note that p may continue to receive TYPE2 messages before it receives the computed GVT from p_0). The input (output) set of a process p considered in the GVT computation is $S_{GVT,I}(p)$ ($S_{GVT,O}(p)$), the input (output) set when p enters Phase 1. This is required because the local minimum of process p is computed at the time when it enters Phase 1.

4. MEMORY MANAGEMENT

A parallel simulation may consume much more storage than space a sequential simulation no matter which parallel simulation protocol is used [13, 24]. Since extra memory is required to store the histories of processes, memory management for Time Warp is more critical than that for conservative simulation protocols, such as the Chandy-Misra approach.

Basically, there are two approaches to reducing the memory consumption of Time Warp: reducing the state saving frequency as described in section 2 and *fossil collection*. Fossil collection is described as follows. Jefferson showed that:

Theorem 3: Let $\tau < GVT(t)$. After time t, the following objects in a process p_i are obsolete and can be deleted:

- the messages with timestamps no later than τ in the input queue;
- the copies of the process state with timestamps earlier than τ except for the one with the largest timestamp no later than τ (where the timestamp of a process state x, denoted as $ts(x)$, is the local clock of p_i when its process state is x);
- the messages in the output queue with send times no later than τ.

Based on Theorem 3, fossil collection reclaims obsolete objects after GVT is computed.

The frequency of GVT computation is basically determined by fossil collection: If a low frequency is chosen, a process may exhaust memory before the next fossil collection is performed. On the other hand, a high frequency may result in heavy overhead of GVT computation and fossil collection, and thus reduce the progress of the simulation. Some Time Warp implementations [14] periodically perform fossil collection on at fixed time intervals.

Unfortunately, even if we reduce the state saving frequency and perform fossil collection frequently, Time Warp may still consume much more storage space than a sequential simulation. Thus, it is important to design a memory management algorithm for Time Warp such that the space complexity of Time Warp is $O(M_s)$, where M_s is the amount of storage consumed in sequential simulation. (A memory management algorithm for parallel simulation is called *optimal* if the amount of memory consumed by the algorithm is of the same order as the corresponding sequential simulation.) Jefferson proposed the first optimal memory management algorithm, called *cancelback* [13]. In this protocol, when the Time Warp simulation runs out of memory, objects (i.e., input messages, states, or output messages) with send times later than GVT are cancelled to make more room. The cancelled objects will be reproduced later. This section proposes another optimal protocol, called the *artificial rollback protocol*. The basic idea behind this protocol is to roll back "possibly-correct computa-

tions" if necessary, to obtain more free memory space. Since the mechanism for the artificial rollback protocol is the same as that for the rollback mechanism, this protocol can be easily implemented.

We first introduce a special type of rollback that does not exist in normal Time Warp execution.

Definition 2: Without receiving a straggler, a process p may (on purpose) roll back its computation to a timestamp τ ($\tau \geq GVT$) earlier than its local clock. Process p is said to *artificially* roll back to timestamp τ.

We impose the restriction $\tau \geq GVT$ for two reasons. First, artificial rollback should have the same properties as normal rollback, and no process can normally roll back to $\tau < GVT$. Second, if fossil collection is performed at GVT, a process cannot be artificially rolled back to a simulation time earlier than GVT because the earlier parts of process histories may already have been discarded. We will show that artificial rollback does not affect the correctness of Time Warp:

Theorem 4: Let S be a Time Warp simulation consisting of n processes $p_1, p_2, ..., p_n$. Let ck_i be the local clock of p_i. Repeat the same simulation except that process p_i artificially rolls back to a timestamp $\tau < ck_i$ at time t and re-executes. Then the new Time Warp simulation is equivalent to S.

Proof: Let S_s be the sequential counterpart of S. Consider another sequential simulation S'_s, which is identical to S_s except that a new process p_{n+1} is added. Process p_{n+1} does not communicate with other processes except that it schedules an event e with timestamp τ to p_i. When the event occurs, p_i does nothing. Thus, S_s and S'_s are equivalent in terms of the behaviors of $p_1, ..., p_n$. Consider S', the Time Warp implementation of S'_s. From the above discussion, S' is the same as S if we ignore p_{n+1}. Suppose that p_{n+1} sends a message (corresponding to the occurrence of event e) to p_i at time 0, and that the message sending delay for that message is t. (Note that assuming arbitrary message sending delay does not affect the correctness of Time Warp.) In effect, this is the same as an artificial rollback to p_i. Thus, artificial rollbacks do not change the results of a Time Warp simulation. □

From Theorem 4, a process may roll back to an earlier simulation time $\tau \geq$ GVT, and the memory used in the rolled back computation can be reclaimed. Later on, the rolled back computation (if it is correct) will be re-executed, which will produce the same results as the original Time Warp simulation.

Consider a shared memory environment. It is easy to prove that if we compute GVT, perform fossil collection, and roll back all processes to GVT, then the amount of memory used in Time Warp is of the same order as the amount of memory used in the sequential counterpart at simulation time GVT (cf. Theorem 6 in section 6).

With the artificial rollback protocol, Time Warp is able to reduce the amount of memory space used in parallel simulation (while other simulation approaches, such as the Chandy-Misra protocol cannot). However, progress of the simulation may be degraded. The trade-off between time and space is still an open question.

5. DISTRIBUTED TERMINATION

In most Time Warp implementations, termination detection is handled in terms of GVT: When a process p has processed all messages in its input queue, its local clock is set to ∞. When GVT reaches ∞, all the local clocks must be ∞, and no messages can be in transit. Thus, GVT is computed periodically to see if the simulation terminates (i.e., if GVT = ∞). To efficiently detect distributed termination, we should compute GVT infrequently at the beginning of the simulation. As time passes, the frequency should increase. However, such a selection of frequency may conflict with the needs of fossil collection. To simplify the selection of the GVT computation frequency, an algorithm that automatically detects distributed termination without GVT computation might be attractive.

The distributed termination (DT) problem is non-trivial because no process has complete knowledge of the global state. Many algorithms [5, 6, 16, 17, 26] have been designed to detect distributed termination. Most of these algorithms deliver the control messages (i.e., the messages sent for DT detection) in pre-defined paths. Thus, in the view of these DT detection algorithms, the processes are connected in some fashion. Some algorithms connect the processes as a ring [16]. Others organize the processes as a tree (either dynamically [17] or statically [36]). In general, the "logical topologies" used in these DT detection algorithms may not match physical processor connections. For example, it is not efficient to implement a ring algorithm on a tree architecture, or vice versa. Even if the algorithm and the architecture match initially, inefficiency may be caused by process migration.

In this section, we propose a simple DT detection algorithm. In this algorithm, every process reports its termination to a coordinator which announces the termination of distribution. In other words, we have a "star" logical topology, and the shortest path is always chosen to deliver control messages between a process and the coordinator. One may argue that the coordinator may become a bottleneck. In Time Warp simulation, we expect that the number of control messages sent in DT detection is small compared with the number of data messages sent. Thus, if we have a dedicated coordinator process for DT detection, it will not be a bottleneck.

The sets $V_p(q)$ and $SN_p(q)$ described in section 3 are used in our DT detection algorithm. This algorithm is based on the *principle of message counting*: If all processes are idle and the number of messages sent in the system is equal to the number of messages received, then the distributed computation has terminated.

Definition 3: Let $S_I(p_i)$ be the input set of a process p_i. Process p_i satisfies the *local termination condition* if and only if its local clock $ck_i = \infty$, and for all processes $p_j \in S_I(p_i)$, $|SN_{p_i}(p_j)| = 1$ (i.e., there is only one range in $SN_{p_i}(p_j)$).

Note that if $|SN_{p_i}(p_j)| \neq 1$, then at least one data message sent from p_j to p_i is in transit, and p_i will be re-activated after the message is received. Let $SN_{p_i}(p_j) = \{(sn_f, sn_l)\}$, and let $r_{i,j} = sn_l$ for all $j \in S_I(p_i)$ and $s_{i,k} = V_{p_i}(p_k).sn$ for all $k \in S_O(p_i)$. If the local termination condition is satisfied, then $r_{i,j}$ ($s_{i,j}$) is the number of messages p_i received from (sent to) p_j. When this condition is detected, p_i sends a message M_i = (IDLE, S_i, R_i) to the coordinator p_0, where

$$S_i = \{s_{i,j} | j \in S_O(p_i)\} \text{ and } R_i = \{r_{i,j} | j \in S_I(p_i)\}.$$

The coordinator maintains two sets $S' = \{s'_{i,j} | \ \forall i,j\}$ and $R' = \{r'_{i,j} | \forall i,j\}$. Initially, $s'_{ij} = r'_{i,j} = 0$ for all i,j. When p_0 receives a message M_i = (IDLE, S_i, R_i), the following code is executed to update S' (R' is updated in the similar way):

for all $s_{i,j} \in S_i$ **do if** $s_{i,j} > s'_{i,j}$ **then** $s'_{i,j} = s_{i,j}$.

Now we will describe a distributed termination condition. When p_0 satisfies this condition, the distributed computation must have terminated.

Definition 4: Process p_0 satisfies the *distributed termination* (DT) *condition* if and only if (i) it receives at least one idle message from every process and (ii) $s'_{i,j} = r'_{j,i}$ for all i,j.

In Appendix B we prove the following theorem.

Theorem 5: The distributed computation terminates if the coordinator satisfies the DT condition.

Thus, every time the coordinator p_0 receives an IDLE message, it tests the DT condition. If the condition is satisfied, then p_0 announces distributed termination. It is apparent that this algorithm is optimal in time complexity (if we ignore message contention): Suppose that a process p_i is idle forever after time t_{p_i}. Let $*t_{p_i}$ be the message sending delay of the IDLE message p_i sends at time t_{p_i}. Then p_0 reports distributed termination at time $\max_{\forall i}(t_{pi} + \Delta t_{pi})$. The above algorithm is based on "process-to-process" message counting; i.e., we need to check if $s'_{i,j} = r'_{j,i}$ for all possible pairs (i, j). In fact, we only need to count the number of messages $s_{i,j}$ sent from p_i to every process $p_j \in S_O(p_i)$ on a process-to-process basis, and count the total number r_i of messages received by process p_i (or vice versa). The DT condition tested by p_0 is now $\sum_{\forall j} s'_{j,i} = r'_i$. Note that if we only count the total numbers of messages sent and received by a process, then the algorithm may detect false termination. Directions for optimizing our algorithm are given in [25].

To conclude, using a simple DT detection algorithm that does not rely upon GVT computation means that optimization of fossil collection is not affected by DT detection.

6. FAULT TOLERANCE

In a distributed system, there are two kinds of failures: *process failures* and *communication failures*. Both types of failures are usually detected by *timeout*. Recovery of a communication failure only involves two parties and can be done locally. On the other hand, recovery of a process failure may involve more than two processes and require global information of the system. In a distributed Time Warp simulation, if a process fails at time t and is recovered at time t', then the computations of all the other processes during $[t, t']$ are usually incorrect. In other words, a process must re-execute from the state at time t even though it is not a failed process; i.e., recovery for one-process failure is the same as that for an all-process failure in a Time Warp simulation. Thus, to recover process failures, a distributed snapshot is required. This paper concentrates on process failures and ignores communication failures. (We assume that either the communication system is reliable or that a lost message is recovered locally, and the only effect is that the message experiences longer message sending delay. A lost message can be easily detected by using data structure $SN_p(q)$ in a timeout scheme which avoids acknowledging every message.) Henceforth, "failure" means "process failure."

The distributed snapshot algorithm proposed by Chandy and Lamport [4] cannot be used in Time Warp because the algorithm requires the FIFO communication property. (In [1], a fault tolerance protocol with FIFO communication property was proposed for Time Warp simulation.) The difficulty in taking a distributed snapshot in a Time Warp system with the non-FIFO communication property is similar to that for GVT computation; that is, it is usually not possible to record the current state of the system. Fortunately, since the history of a process is saved, we can, based on GVT, record an earlier "legal" state of the system. We first describe the information saved in a snapshot of a sequential simulation. Then we show how to obtain this information in the corresponding Time Warp simulation. The basic idea was proposed by Jefferson [12]. This section gives concrete descriptions and proofs.

Without loss of generality, assume that all messages executed in the simulation have different timestamps. Here, the timestamp is used to distinguish the execution order of the messages. For messages with the same timestamps, some execution order still exists. Thus, this assumption does not restrict the results presented in this section.

Consider a simulation S of n processes. Let e be an event occurring in process p_i. Consider S_s, the sequential implementation of S. Let $x_{s,i}(\tau)$ be the process state of p_i in S_s after e (where $ts(e) = \tau$) is executed. Let

$$\Psi_{p_i} = \{e_1, ..., e_j, ..., e_k\}$$

be the set of events executed at process p_i, where $ts(e_1) < ts(e_2) < ... < ts(e_k)$. Let $\Psi_{pi} = \bigcup_{1 \le i \le n} \Psi_{p_i}$. Then the sequence of states of p_i is $x_{s,i}(\tau_1), x_{s,i}(\tau_2), ..., x_{s,i}(\tau_k)$, where $\tau_j = ts(e_j)$, and the set of p_i's states is

$$X_{s,i} = \{x_{s,i}(\tau_1), x_{s,i}(\tau_2), ..., x_{s,i}(\tau_k)\}.$$

At simulation time τ (after all events with timestamps no less than τ are executed), the event queue of S_s is

$$\Psi(\tau) = \{e \in \Psi | ts'(e) =: \tau, ts(e) > \tau\} \tag{12}$$

and the set of process states is

$$X_s(\tau) = \bigcup_i \{\sup(X_{s,i,} \tau)\}, \tag{13}$$

where $(X_{s,i}, \tau) = x$, a process state in $X_{s,i}$ such that $ts(x) =: \tau$ and for all $x' \in X_{s,i}$, if $ts(x') < \tau$ then $ts(x') < ts(x)$. Note that (12) and (13) are the information to be saved in the snapshot of S_s at simulation time τ. That is, $[\Psi(\tau), X_s(\tau)]$ is a legal state of the sequential simulation, and starting from this state, a correct simulation result can be produced.

Consider S_{tw}, the Time Warp implementation of S. For all i, $1 \le i \le n$, let $t_i \ge t$, and $\tau < GVT(t)$. Let I_{tw,i,t_i} (O_{tw,i,t_i}) be the set of events (i.e., messages) in the input (output) queue of p_i at time t_i, and let X_{tw,i,t_i} be the copies of states in the process state queue of p_i at time t_i. Then from Theorem 1, the events in the set

$$O_{tw,i,t_i}(\tau) = \{e \in O_{tw,i,t_i} | ts'(e) \le \tau\}$$

are never sent for cancellation in rollbacks, and the messages in the set

$$I_{tw,i,t_i}(\tau) = \{e \in I_{tw,i,t_i} | ts'(e) \leq \tau\}$$

and the states in the set

$$X_{tw,i,t_i}(\tau) = \{x \in X_{tw,i,t_i} | ts(x) \leq \tau\}$$

are never cancelled. This implies that $I_{tw,i,t_i}(\tau) \subseteq \Psi_{p_i}$ and that $X_{tw,i,t_i}(\tau) \subseteq X_{s,i}$. More precisely,

$$I_{tw,i,t_i}(\tau) = \{e \in \Psi_{p_i} | ts'(e) \leq \tau\} \text{ and } X_{tw,i,t_i}(\tau) = \{x \in X_{s,i} | ts(x) \leq \tau\}. \quad (14)$$

Definition 5: Let $\tau < GVT(t_0)$. A Time Warp simulation is artificially rolled back to τ at time $t \geq t_0$ if and only if all processes are artificially rolled back to τ (i.e., the executions of events with timestamps later than τ are rolled back) at time t_i, $t_0 \leq t_i \leq t$, and if during $[t_i, t]$, p_i only executes the negative messages (if any).

If a Time Warp simulation is artificially rolled back to τ at time t, then all negative messages with send times later than τ are sent to annihilate the corresponding positive messages; in other words, after the artificial rollback, all objects have send times no later than τ. Thus, for every process p_i, we have

$$I_{tw,i,t} = I_{tw,i,t_i}(\tau), \qquad X_{tw,i,t} = X_{tw,i,t_i}(\tau), \qquad \text{and } O_{tw,i,t} = O_{tw,i,t_i}(\tau). \quad (15)$$

Suppose that fossil collection with timestamp τ is performed after the artificial rollback and is completed at time $t^+ > t$. Then from Theorem 3, Definition 5, and Equations (14) and (15), for a process p_i, we have

$$\begin{aligned} &I_{tw,i,t^+} = I_{tw,i,t} - \{e \in I_{tw,i,t} | ts(e) =: \tau\} = \{e \in \Psi_{p_i} | ts'(e) \leq \tau, ts(e) > \tau\}, \\ &X_{tw,i,t^+} = X_{tw,i,t} - (X_{tw,i,t} - \{\sup(X_{tw,i,t}, \tau)\}) = \{\sup(X_{s,i}, \tau)\}, \\ &O_{tw,i,t^+} = O_{tw,i,t} - \{e \in O_{tw,i,t} | ts'(e) \leq \tau\} = \emptyset. \end{aligned} \quad (16)$$

Thus, we have the following theorem:

Theorem 6: Let $\tau < GVT(t_0)$. Suppose that a Time Warp is artificially rolled back to τ at time $t \geq t_0$, and that a fossil collection with timestamp τ is performed and is completed at time $t^+ > t$; then

$$\bigcup_i I_{tw,i,t+} = \Psi(\tau), \quad \bigcup_i X_{tw,i,t+} = X_s(\tau), \text{ and } \bigcup_i O_{tw,i,t+} = \varnothing.$$

Theorem 6 states that at any time, we can obtain a legal state of the sequential simulation from Time Warp. Thus, a distributed snapshot of Time Warp can be taken using the following steps:

Step 1: Compute GVT at time t. Let τ be the largest timestamp smaller than $GVT(t)$.

Step 2: At time $t_i \geq t$, process p_i stores the following information into stable storage:

- a subset of the input queue, $I_{ds,i}(\tau) = \{e \in I_{tw,i,t_i} | ts'(e) < \tau, ts(e) \geq \tau\}$;
- a process state $x_{ds,i}(\tau) = \sup(X_{tw,i,t_i}, \tau)$

We note that:

- For all $t_1, t_2 \geq t$ and $\tau < GVT(t)$,

  $$I_{tw,i,t_1}(\tau) = I_{tw,i,t_2}(\tau), \text{ and } X_{tw,i,t_1}(\tau) = X_{tw,i,t_2}(\tau)$$

  (from (14)). This implies that the processes do not need to take local snapshots at the same time. The local snapshots are consistent if they are taken with respect to the same simulation time τ, and if the local snapshots are taken at time $t_i \geq t$.
- From Theorem 6, there is no need to save the output queues in a distributed snapshot. In other words, the amount of storage required to save the distributed snapshot is the same as the snapshot for a sequential simulation, and the distributed snapshot taken in the above procedure is the same as the sequential snapshot taken at simulation time τ.

This section has shown that a distributed snapshot of a Time Warp simulation can be easily taken. To our knowledge, there is no simple way to address the fault tolerance issue for conservative protocols, such as Chandy-Misra.

7. SUMMARY

This paper has addressed several important issues in designing a distributed Time Warp simulation. We have proposed a heuristic to select the checkpoint interval in order to reduce the state saving overhead. We have generalized a previously proposed GVT algorithm by allowing dynamic communication topologies. We have proposed a new algorithm for memory management called artificial rollback, which ensures that Time Warp only consumes the same amount of memory as does the corresponding sequential simulation. The idea is to roll back uncommitted computation to make more memory space in order to complete computation in the critical path. Based on the message conservation law, we have presented a distributed termination detection algorithm which does not require periodic computation of GVT. Finally, using GVT, we have addressed the fault tolerance issue by presenting a simple and efficient distributed snapshot algorithm.

ACKNOWLEDGEMENT

We would like to thank Jon Agre, Brian Coan, Friedemann Mattern, Peter Reiher, and Abel Weinrib for their valuable comments.

REFERENCES

1. J. R. Agre, A. Johnson and S. Vopova, "Recovering from process failures in the time warp mechanism," in *Proceedings of 8th Symposium on Reliable Distributed System*, 1989, pp. 24-30.

2. S. Bellenot, "State skipping performance with the time warp operating system," submitted to--for publication by Private Communication, 1991.
3. S. Bellenot, "Global virtual time algorithms," in *Proceedings of 1990 SCS Multiconference on Distributed Simulation*, 1990, pp. 122-130.
4. K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Programming Languages and Systems*, Vol. 3, No. 1, 1985, pp. 63-75.
5. E. W. Dijkstra, W. H. J. Feijen and A. J. M. Van Gasteren, "Derivation of a termination detection algorithm for distributed computations," *Information Processing Letters*, Vol. 16, 1983, pp. 217-219.
6. N. Francez and M. Rodeh, "Achieving distributed termination without freezing," *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 3, 1982, pp. 287-292.
7. R. M. Fujimoto, "Time warp on a shared memory multiprocessor," in *Proceedings of 1989 International Conference on Parallel Processing*, Vol. III, 1989, pp. 242-249.
8. R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, Vol. 33, No. 10, 1990, pp. 31-53.
9. R. M. Fujiimoto, J.-J. Tsai and G. Gopalakrishnan, "Design and performance of special purpose hardware for time warp," in *Proceedings of 15th Annual International Symposium on Computer Architecture*, 1988, pp. 401-408.
10. A. Gafni, "Rollback mechanisms for optimistic distributed simulation," in *Proceedings of 1988 SCS Multiconference on Distributed Simulation*, 1988, pp. 61-67.
11. E. Gelenbe, "On the optimum checkpoint interval," *Journal of ACM*, Vol. 26, No. 2, 1979, pp. 259-270.
12. D. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, 1985, pp. 404-425.
13. D. Jefferson, "Virtual time II: The cancelback protocol for storage management in time warp," in *Proceedings of 9th Annual ACM Symposium on Principles of Distributed Computing*, 1990, pp. 75-90.
14. D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. D. Loreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J. Wedel, H.Younger and S. Bellenot, "Distributed simulation and the time warp operating system," in *Proceedings of 11th ACM Symposium on Operating Systems Principles*, 1987, pp. 77-93.
15. F. J. Kaudel, "A literature survey on distributed discrete event simulation," Simuletter, Vol. 18, No. 2, 1987, pp. 11-21.
16. D. Kumar, "A class of termination detection algorithms for distributed computations," *5th Conference on Foundations of Software Technology and Theoretical Computer Science*, 1985, pp. 73-100.
17. T.-H. Lai, "Termination detection for dynamically distributed systems with non-first-in-first-out communication," *Journal of Parallel and Distributed Computing*, Vol. 3, No. 4, 1986, pp. 577-599.
18. Y.-B. Lin and E. D. Lazowska, "Determining the global virtual time in a distributed simulation," in *Proceedings of International Conference on Parallel Processing*, Vol. III, 1990, pp. 201-209.
19. Y.-B. Lin and E. D. Lazowska, "Exploiting lookahead in parallel simulation," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 4, 1990, pp. 457-469.
20. Y.-B. Lin and E. D. Lazowska, "Optimality considerations for time warp parallel simulation,

" in *Proceedings of 1990 SCS Multiconference on Distributed Simulation*, 1990, pp. 29-34.
21. Y.-B. Lin and E. D. Lazowska, "A study of time warp rollback mechanisms," *ACM Transactions on Modeling and Computer Simulation*, Vol. 1, No. 1, 1991, pp. 51-72.
22. Y.-B. Lin and E. D. Lazowska, "A time-division algorithm for parallel simulation," *ACM Transactions on Modeling and Computer Simulation*, Vol. 1, No. 1, 1991, pp. 73-83.
23. Y.-B. Lin and E. D. Lazowska, "Processor scheduling for time warp parallel simulation," *Workshop on Parallel and Distributed Simulation*, 1991, pp. 50-58.
24. Y.-B. Lin, E. D. Lazowska and J.-L. Baer, "Conservative parallel simulation for systems with no lookahead," in *Proceedings of 1990 SCS Multiconference on Distributed Simulation*, 1990, pp. 144-149.
25. Y.-B. Lin, E. D. Lazowska and P. Blanc, "Termination detection for distributed computation," in preparation, 1990.
26. F. Mattern, "Algorithms for distributed termination detection," *Distributed Computing*, Vol. 2, 1987, pp. 161-175.
27. F. Mattern, "Efficient distributed snapshots and global virtual time algorithms for non-FIFO systems," Technical Report, Department of Computer Science, University of Kaiserslautern, Fed. Rep. Germany, 1990.
28. J. Misra, "Distributed discrete-event simulation," *Computing Surveys*, Vol. 18, No. 1, 1986, pp. 39-65.
29. B. Preiss, I. D. MacIntyre and W. M. Loucks, "On the trade-off between time and space in optimistic parallel discrete-event simulation," Technical Report, Department of Electrical and Computer Engineering, University of Waterloo, 1990.
30. P. Reiher, R. Fujimoto, S. Bellenot and D. Jefferson, "Cancellation strategies in optimistic execution systems," in *Proceedings of 1990 SCS Multiconference on Distributed Simulation*, 1990, pp. 112-121.
31. P. F. Reynolds, "Heterogenous distributed simulation," in *Proceedings of 1988 Winter Simulation Conference*, 1988, pp. 206-209.
32. R. Righter and J. C. Walrand, "Distributed simulation of discrete event systems," in *Proceedings of the IEEE*, Vol. 77, No. 1, 1989, pp. 8-24.
33. P. Reiher, *Private Communication*, 1991.
34. B. Samadi, "Distributed simulation, algorithms and performance analysis," Ph.D. thesis, Computer Science Department, University of California, Los Angeles, 1985.
35. A. N. Tantawi and M. Ruschitzka, "Performance analysis of checkpointing strategies," *ACM Transactions on Computer Systems*, Vol. 2, No. 2, 1984, pp. 123-144.
36. R. Topor, "Termination detection for distributed computations," *Information Processing Letters*, Vol. 18, 1984, pp. 33-36.
37. J. W. Young, "A first order approximation to the optimum checkpoint interval," *Communications of the ACM*, Vol. 17, No. 17, 1974, pp. 530-531.

APPENDIX

A: PROOFS FOR THE GVT ALGORITHM

This appendix proves Theorem 2.

Let P be the set of processes. Consider the time t_p at which the process p enters Phase 1. Then the value lm_p is computed at time t_p (and is denoted as $lm_p(t_p)$). Let $t_M = \max_{p \in P} t_p$. Let $TR_{p,q}(t)$ be the set of transient messages sent from p to q at time t, and let $TR_q(t) = \bigcup_{p \in S_I(q)} TR_{p,q}(t)$. Let $M_{p,q}(t_1, t_2)$ be the set of messages sent from p to q [5] in the time interval $[t_1, t_2]$. By convention, $M_{p,q}(t_1, t_2) = \emptyset$ if $t_1 > t_2$.) In Phase 2, if p receives a message msg = (TYPE1, $sn_{p,q}$) at time $t_{p,q}$, then $t_{p,q} > t_q$ and

$$\tau_{p,q} = \min_{m \in TR_{p,q}(t_q) \cup M_{p,q}(t_q, t_{p,q})} ts(m). \tag{17}$$

Let GVT_1 be the GVT computed in our algorithm; then

$$GVT_1 = \min_{p \in P} \left[lm_p(p), \min_{q \in S_0(p)} \left(\min_{m \in TR_{p,q}(t_q) \cup M_{p,q}(t_q, t_{p,q})} ts(m) \right) \right]. \tag{18}$$

Now we ignore the messages in $M_{p,q}(t_q, t_{p,q})$ in (18) and consider a new variable GVT_2, where

$$\begin{aligned} GVT_2 &= \min_{p \in P} \left[lm_p(t_p), \min_{q \in S_0(p)} \left(\min_{m \in TR_{p,q}(t_q)} ts(m) \right) \right] \\ &= \min_{p \in P} \left[lm_p(t_p), \min_{q' \in S_I(p)} \left(\min_{m \in TR_{q',p}(t_p)} ts(m) \right) \right] \\ &= \min_{p \in P} \left(lm_p(t_p), \min_{m \in TR_p(t_p)} ts(m) \right). \end{aligned} \tag{19}$$

Let $LM'_p(t_p) = \min \left(lm_p(t_p), \min_{m \in TR_p(t_p)} ts(m) \right)$; then (19) is re-written as

$$GVT_2 = \min_{p \in P} LM'_p(t_p).$$

We first derive a condition when $GVT_2 > GVT(t_M)$. From the condition, we show that

$$\begin{aligned} GVT(t_M) &\geq \min \left[GVT_1, \min_{p,q \in P, p \neq q} \left(\min_{m \in M_{p,q}(t_q, t_p)} ts(m) \right) \right] \\ &= \min_{p,q \in P, p \neq q} \left(LM'_p(t_p), \min_{m \in M_{p,q}(t_q, t_p)} ts(m) \right). \end{aligned} \tag{20}$$

[5] These messages may not be received by q before time t_2.

Based on (20), we derive a condition which ensures $GVT(t_M) \geq GVT_1$.

Definition 6: A process p is called a *Type A* process if it has rolled back to a simulation time $ts \leq GVT(t_M)$ in the time interval $(t_p, t_M]$.

Definition 7: A process p is called a *Type B* process if (i) it rolls back a Type A process q, (ii) $t_p > t_q$, and (iii) the straggler m that rolls back q is sent from p in the time interval $(t_q, t_p]$.

Corollary 1: Let m be a straggler sent from a Type B process. Then $ts(m) \leq GVT(t_M)$.

Proof: Directly from Definitions 6 and 7. □

Lemma 1: If $GVT(t_M) < GVT_2$, then there exists a Type A process.

Proof: Cf. Lemma 2, [34]. □

Lemma 2: Suppose a Type A process p is rolled back by a message m sent from a process q. Then q is either a Type A process or a Type B process.

Proof: We prove by contradiction. Assume that q is neither a Type A process nor a Type B process. From Definition 6, p is rolled back by a message m after time t_p. From Corollary 1 , $ts(m) = GVT(t_M)$. Suppose that m is sent at time t. There are two possibilities:

A. $t_q \leq t_p$: There are two sub-cases.

A.1. $t > t_q$ (cf. Fig. 5(a)): q must have been rolled back in time interval $[t_q, t]$; otherwise, $ts(m) > LM'_q(t_q) > GVT(t_M)$, which contradicts the fact that $ts(m) \leq GVT(t_M)$. This implies that q is a Type A process, a contradiction.

A.2. $t < t_q$ (cf. Fig. 5(b)): Since p does not receive m before $t_p > t$, m is a transient message at time t_p. In other words, $m \nprec TR_p(t_p)$. Thus,

$$LM'_p(t_p) = \min\left[lm_p(t_p), \min_{m \in TR_p(t_p)} ts(m)\right] \leq ts(m) \leq GVT(t_M) < GVT_2.$$

This contradicts the fact that $GVT_2 \leq LM'_p(t_p)$.

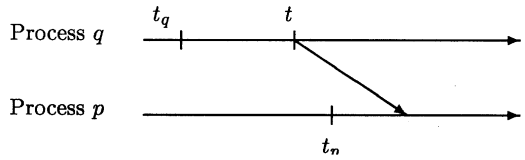
B. $t_q > t_p$: There are two sub-cases.

B.1. $t > t_q$ (cf. Fig. 5(c)): Similar to A.1, q is a Type A process, a contradiction.

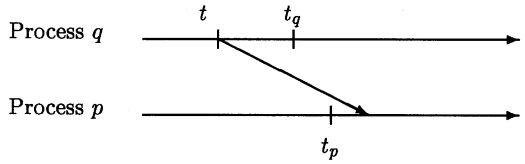
B.2. $t < t_q$(cf. Fig. 5(d)): q is a Type B process (Definition 7), a contradiction. □

Definition 8: A *rollback propagation* is defined as a set of rollbacks occurring in a set of processes $P_r \subseteq P$ (P_r, the set of processes that involve in a rollback), where for every rollback Rb_p occurring at the process $p \in P_r$, there exists a rollback Rb_q occurring at $q \in P_r$ such that either Rb_p is caused by Rb_q (and Rb_p is called the successor of Rb_q) or vice versa.

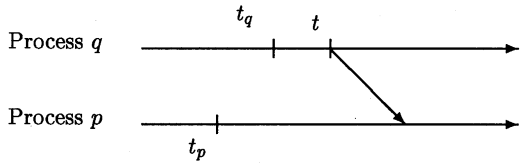
Definition 9: Consider a rollback propagation and its process set P_r. A rollback Rb_p occurring at $p \in P_r$ is called the *root* of the rollback propagation if there exists no rollback Rb_q, $q \in P_r$ such that Rb_p is caused by Rb_q.



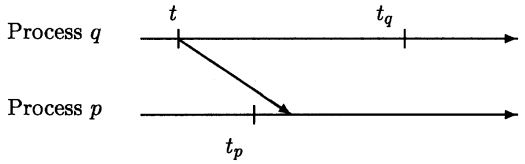
(a) Case A.1.



(b) Case A.2.



(c) Case B.1.



(d) Case B.2.

Fig. 5. Four cases in Lemma 2.

Lemma 3: Consider a rollback propagation and its process set $P_r = \{p_1, p_2, ..., p_n\}$, where $Rb_{p_{i+1}}$ is the successor of Rb_{p_i}, $1 \le i \le n-1$. (Without loss of generality, we assume that each rollback causes at most one other rollback.) Let m_i be the straggler sent from p_i to p_{i+1} (i.e., $Rb_{p_{i+1}}$ is caused by the arrival of m_i). Then $ts(m_i) \le ts(m_{i+1})$, $1 \le i \le n-1$.

Proof: Directly from the definition of a rollback and Definition 8. □

Definition 10: Consider a rollback propagation and its process set P_r. The propagation is called a *Type A rollback propagation* if all processes in P_r are Type A processes.

Lemma 4: If $GVT(t_M) < GVT_2$, then there exists a Type B process.

Proof: From Lemma 2 and an inductive argument (omitted), it is apparent that (i) there exists a Type A rollback propagation, and (ii) the root of a Type A rollback propagation is caused by a message sent from a Type B process. Then from Lemma 4, a Type B process exists. □

Theorem 7: $GVT(t_M) \geq \min_{p,q \in P, p \neq q} \left(LM'_q(t_q), \min_{m \in M_{p,q}(t_q, t_p)} ts(m) \right)$.

Proof: If no Type A process exists, then

$$GVT(t_M) \geq \min_{p \in P} LM'_p(t_p) \geq \min_{p,q \in P, p \neq q} \left(LM'_p(t_p), \min_{m \in M_{p,q}(t_q, t_p)} ts(m) \right)$$

(Lemma 1) If there exists a Type A process, then there exists a Type B process p (Lemmas 1 and 4), such that p sends a straggler m' to a Type A process q in the time interval $(t_q, t_p]$ (Definition 7) such that $ts(m') \leq GVT(t_M)$ (Corollary 1). In other words, $m' \in M_{p,q}(t_q, t_p)$ and $\min_{p,q \in P, p \neq q} \left(\min_{m \in M_{p,q}(t_q, t_p)} ts(m) \right) \leq ts(m') \leq GVT(t_M)$. This implies that

$$GVT(t_M) \geq \min_{p,q \in P, p \neq q} \left(LM'_p(t_p), \min_{m \in M_{p,q}(t_q, t_p)} ts(m) \right)$$

Now Theorem 2 can be proven as follows. If all processes p enter Phase 1 before Phase 2, then p receives a TYPE1 message at time $t_{p,q} > t_p$. From (18) and Theorem 7, we have

$$\begin{aligned} GVT_1 &= \min_{p \in P} \left[lm_p(p), \min_{q \in S_0(p)} \left(\min_{m \in TR_{p,q}(t_q) \cup M_{p,q}(t_q, t_{p,q})} ts(m) \right) \right] \\ &\leq \min_{p \in P} \left[lm_p(p), \min_{q \in S_0(p)} \left(\min_{m \in TR_{p,q}(t_q) \cup M_{p,q}(t_q, t_p)} ts(m) \right) \right] \\ &= \min_{p,q \in P, p \neq q} \left(LM'_p(t_p), \min_{m \in M_{p,q}(t_q, t_p)} ts(m) \right) \\ &\leq GVT(t_M). \end{aligned}$$

B: CORRECTNESS OF THE DT DETECTION ALGORITHM

This appendix proves Theorem 5; that is, we show the correctness of our DT detection algorithm. In this appendix, the term "messages" means data messages or the messages sent in the Time Warp simulation, and the term "reports" means idle messages sent for DT detection. Let $t_s(m)$ be the (real) time when a message/report m is sent. Let $t_r(m)$ be the (real) time when a message/report m is received. We assume that $t_r(m) > t_s(m)$ (i.e., non-zero message sending delay). Let $M_{i,k}$ be the kth report sent from p_i.

Definition 11: $M_i(t) = M_{i,k}$ is called the *last effective report* of process p_i at time t if and only if $t_r(M_{i,k}) < t$ and for all k', $t_r(M_{i,k'}) \in [t_r(M_{i,k}), t]$, we have $k' < k$.

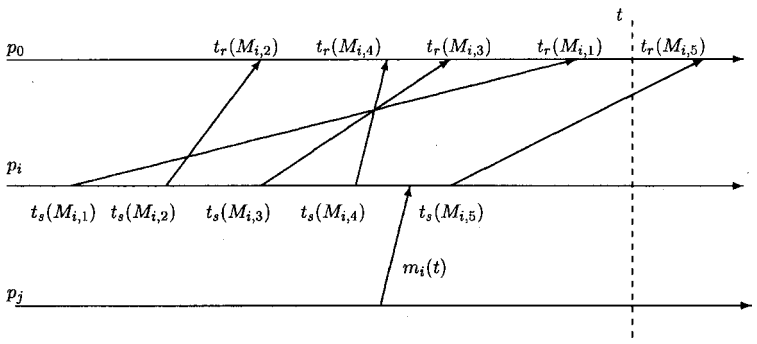


Fig. 6. An example of the last effective report.

Definition 12: The first message which re-activates process p_i after p_i sends $M_i(t)$ is denoted as $m_i(t)$.

In Fig. 6, $M_i(t) = M_{i,4}$ is the last effective report of p_i at time t, and $m_i(t)$ re-activates p_i after $M_i(t)$ is sent.

Lemma 5: Suppose that p_0 satisfies the DT condition at time t_T, and for all $p_i \in P$, let $M_i(t_T)$ be the last effective report of p_i at time t_T. Then p_i is idle in the time interval $[t_s(M_i(t_T)), t_T]$.

Proof: We prove by contradiction. Suppose that p_i is active in $(t_s(M_i(t_T)), t_T]$. Then p_i receives (and is re-activated by) a message $m_i(t_T)$ after time $t_s(M_i(t_T))$. Suppose that $m_i(t_T)$ is sent by p_j. Let $m_i(t_T)$ be the lth message sent from p_j to p_i. Since $r'_{i,j}(t_T) = r_{i,j}(t_s(M_i(t_T))) < l$ and $s'_{j,i}(t_T) = r'_{i,j}(t_T)$, we have $l > s'_{j,i}(t_T)$. This implies that $t_s(m_i(t_T)) > t_s(M_j(t_T))$, that p_j must be re-activated by a message $m_j(t_T)$ in the interval $(t_s(M_j(t_T)), t_s(m_i(t_T)))$, and that

$$t_r(m_j(t_T)) < t_s(m_i(t_T)) < t_r(m_i(t_T))$$

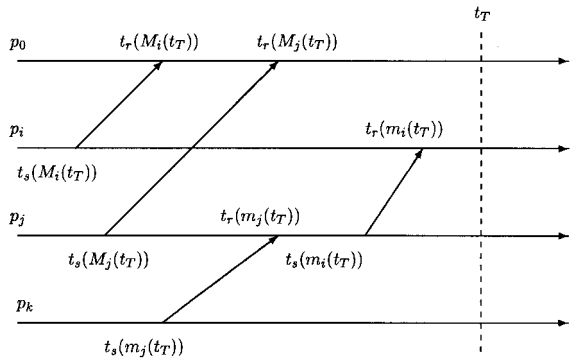


Fig. 7. The timing diagram for Lemma 5.

(cf. Fig. 7) Thus, for every active process p_{i_l}, there always exists another active process $p_{i_{l+1}}$ such that $t_r(m_{i_{l+1}}(t_T)) < t_r(m_{i_l}(t_T))$. In other words, we can find an infinitive sequence $i_1, i_2, ..., i_l, ...$ such that

$$t_r(m_{i_1}(t_T)) > t_r(m_{i_2}(t_T)) > ... > t_r(m_{i_l}(t_T)) > ...$$

Since $|P|$ is finite, we have $p_{i_l} = p_{i_n}$ for some $n < l$. Without loss of generality, let $n = 1$. Then we have

$$t_r(m_{i_1}(t_T)) > t_r(m_{i_2}(t_T)) > ... > t_r(m_{i_{l-1}}(t_T)) > t_r(m_{i_l}(t_T)) = t_r(m_{i_1}(t_T)),$$

a contradiction. □

Lemma 6: If p_0 satisfies the DT condition at t_T, then there is no data message in transit.

Proof: We prove by contradiction. Suppose that a message sent from p_i to p_j is in transit at time t_T. $p_i(p_j)$ is idle in the intervals $[t_s(M_i(t_T)), t_T]([t_s(M_j(t_T)), t_T])$, where $M_i(t_T)(M_j(t_T))$ is the last effective report at t_T (From Lemma 5). Thus, $s_{i,j}(t_T) = s'_{i,j}(t_T)$ and $r_{j,i}(t_T) = r'_{j,i}(t_T)$. Since the transient message must be sent before $t_s(M_i(t_T))$ (Lemma 5, and p_j has not received the message at time t_T), we have $r_{j,i}(t_T) < s_{i,j}(t_T)$ (cf. Fig. 8). This implies that $r'_{j,i}(t_T) < s'_{i,j}(t_T)$, which contradicts the DT condition.

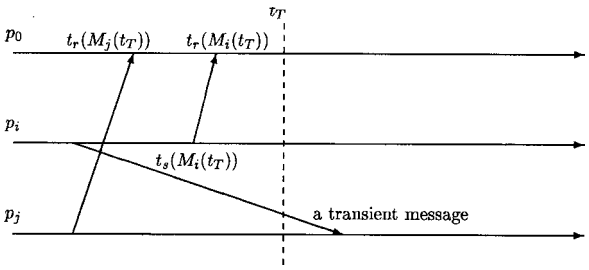


Fig. 8. The timing diagram for Lemma 6.

Thus, Theorem 5, (i.e., the system terminates when p_0 satisfies the DT condition) is a direct consequence of Lemmas 5 and 6. It is apparent that this algorithm is optimal in time complexity:

Theorem 8: Suppose that a process p_i is idle forever after time t_{p_i}. Let Δt_{p_i} be the message sending delay of the report p_i sends at time t_{p_i}. Then p_0 reports distributed termination at time $\max_{\forall i}(t_{pi} + \Delta t_{pi})$.

From Theorem 5, our algorithm satisfies the *safety* property (that is, no false termination is detected). From Theorem 8, our algorithm satisfies the *liveness* property (that is, after the system terminates, the termination is detected in finite time).

Yi-Bing Lin (林一平) received his BSEE degree from National Cheng Kung University in 1983, and his Ph.D. degree in Computer Science from the University of Washington in 1990. From 1990 to 1995, he was with the Applied Research Area at Bell Communications Research (Bellcore), Morristown, NJ. In 1995, he was appointed as a professor of Department of Computer Science and Information Engineering (CSIE), National Chiao Tung University (NCTU). In 1996, he was appointed as Deputy Director of Microelectronics and Information Systems Research Center, NCTU. Since 1997, he has been elected as Chairman of CSIE, NCTU. His current research interests include design and analysis of personal communications services network, mobile computing, distributed simulation, and performance modeling.

Dr. Lin is an associate editor of IEEE Network, an editor of IEEE J-SAC: Wireless Series, an editor of IEEE Personal Communications Magazine, an editor of Computer Networks, an area editor of ACM Mobile Computing and Communication Review, a columnist of ACM Simulation Digest, an editor of International Journal of Communications Systems, an editor of ACM/Baltzer Wireless Networks, an editor of Computer Simulation Modeling and Analysis, an editor of Journal of Information Science and Engineering, Program Chair for the 8th Workshop on Distributed and Parallel Simulation, General Chair for the 9th Workshop on Distributed and Parallel Simulation, Program Chair for the 2nd International Mobile Computing Conference, a guest editor for the ACM/Baltzer MONET special issue on Personal Communications, a guest editor for IEEE Transactions on Computers special issue on Mobile Computing, and a Guest Editor for IEEE Communications Magazine special issue on Active, Programmable, and Mobile Code Networking. Lin received 1997 Outstanding Research Award from National Science Council, R.O.C., and Outstanding Youth Electrical Engineer Award from CIEE, R.O.C.