AN ALGORITHM FOR MINIMALLY LATENT GLOBAL VIRTUAL TIME [*]

Alexander I. Tomlinson Vijay K. Garg

Department of Electrical and Computer Engineering
The University of Texas at Austin, Austin, Texas 78712

This paper appears in:
Proceedings of the 7th Workshop on Parallel and Distrubuted Simulation, San Diego, CA, May 1993.

Abstract

Global virtual time (GVT) is used in distributed simulations to reclaim memory, commit output, detect termination, and handle errors. It is a global function that is computed many times during the course of a simulation. A small GVT latency (delay between its occurrence and detection) allows for more efficient use of resources. We present an algorithm which minimizes the latency, and we prove its correctness. The algorithm is unique in that a target virtual time (TVT) is predetermined by an initiator who then detects when GVT $\geq$ TVT. This approach eliminates the avalanche effect because the collection phase is spread out over time, and it allows for regular and timely GVT updates. The algorithm does not require messages to be acknowledged, which significantly reduces the message overhead of the simulation. One possible application is with interactive simulators, where regular and timely updates would produce output that is up to date and appears smooth.

1 Introduction

Discrete event simulation is used to analyze the behavior of systems which are too complex for mathematical models. As a result of this complexity, simulations are extremely computation intensive. Distributed simulation attempts to speed up a simulation by sharing the computational load among several computers.

A simulation consists of a set of processes interacting with each other through events. Each process has a *local virtual time* (LVT) which indicates its simulation time. Each event has a destination process and a *timestamp*. The destination process is responsible for simulating the event; the timestamp indicates at what simulation time the event occurs. Each process executes events in order of timestamps and also produces events for other processes. A process' LVT is equal to the timestamp of the last event executed.

Within a distributed computing system events are implemented as messages. Due to variable process execution times and message delays, processes will often receive events with a timestamp that is less than the process' LVT (indicating that one or more previous events have been executed out of order). This situation is referred to as a *causality error*. Events that result in causality errors are referred to as *stragglers*.

Distributed simulation implementations can be broadly categorized [Rey88] by their method of dealing with causality errors. Conservative systems [CM81] prevent their occurrence while optimistic systems [Jef85, Jef87] recover from their occurrence. A survey of both optimistic and pessimistic schemes appears in [Fuj90]. This paper addresses concerns regarding optimistic systems only.

Time Warp [Jef85] is a synchronization mechanism that solves the causality problem for optimistic systems. In Time Warp, a process that receives a straggler event is rolled back to a LVT before the timestamp of the straggler event. In order to roll back, a process must periodically checkpoint its internal state as well as save copies of all events it executed and generated. This leads to high memory requirements. One of the functions of global virtual time is to enable the reclaiming of memory.

Global virtual time is an example of a distributed monotonic function. Tel[Tel91] provides an excellent overview of distributed monotonic computations and describes several algorithms for computing them.

Time Warp uses the notion of global virtual time to monitor the progress of the simulation. At any point during the simulation, there exists a simulation time, exact_GVT, such that no process will ever roll back to a LVT less than exact_GVT. In terms of processes and events, exact_GVT is the minimum element in the set that includes: the LVT of every process and the

[*]Research supported in part by NSF Grant CCR 9110605, Navy Grant N00039-88-C-0082, TRW faculty assistantship award, IBM Agreement 153, and a Microelectronics Computer Development Fellowship.

timestamp of every event (message) in transit. In a distributed simulation, it is impossible to determine an exact value for exact_GVT without halting the simulation. However, it is possible to determine a lower bound on exact_GVT without halting the simulation. In this paper, "GVT" refers to a value that is a lower bound estimate of exact_GVT, while "global virtual time" refers to the concept.

Since global virtual time is monotonic[Jef85] and the simulation will never roll back to a simulation time less than GVT, GVT is a measure of system progress. The simulation is correct up to GVT. Time Warp repeatedly calculates GVT during the simulation in order to detect normal termination, reclaim storage, commit output events, and handle errors[Jef85]. Timely global virtual time updates reduce the memory requirements and system output delays. However, there is a tradeoff since calculation of GVT takes computing time away from the application.

We use the term *timestamp* to refer to the event attribute which is used in the calculation of GVT. The parameter may be the event's send time or the receive time; the choice depends on implementation issues such as flow control and message exchange.

This paper introduces an algorithm for minimally latent global virtual time. We define the *latency* of a GVT calculation as the delay between the occurrence of $exact\_GVT = x$ and the detection of $GVT \geq x$. The latency of a global virtual time algorithm is the worst case delay in terms of:

t_p = worst case time required to execute a send or receive operation.

t_m = worst case message transmission delay

The parameter t_p allows us to model the fact that sending N messages costs the sender more than sending 1 message.

In our algorithm, an initiator (any process can be the initiator) schedules a special event at all processes to be executed at a *target virtual time* (TVT). When a process executes this event it sends a report to the initiator. The initiator can then determine when GVT becomes greater than or equal to TVT. Each process submits its report when its local virtual time equals TVT.

This algorithm has two important characteristics: 1) minimal latency, and 2) specification of TVT which will be the next computed value of GVT. The algorithm is initiated by any process when it needs an updated value of GVT. At the very least, a process should initiate the algorithm when it runs low on memory. This provides a feedback mechanism for deciding when to compute GVT.

One possible application is interactive simulation, where the output delay should be minimized and the simulation-time interval between successive output commits should regular. The output delay is minimized due to the minimal latency, and the regular output commits can be achieved by setting the target times at regular virtual time intervals.

Section 2 of this paper summarizes related work on global virtual time algorithms. Section 3 presents a few definitions and describes our notation. The algorithm is presented in section 4, and proof of its correctness appears in section 5. In Section 6 we discuss the latency and message overhead of our algorithm and compare to other algorithms. We conclude the paper in section 7.

2 Related Work

In this section we briefly describe other global virtual time algorithms. Since the same approach is used in each algorithm, we first describe the approach and then describe the individual algorithms. Most global virtual time algorithms make the assumption that all executed events are acknowledged[BL89] (our algorithm does not assume this).

Suppose we could take an instantaneous global snapshot at some real time RT. Then, using the assumption of acknowledged events, calculation of GVT is easily accomplished as follows. At RT each process determines a virtual time vt that is equal to the minimum of: the timestamps of all its unacknowledged events and its its local virtual time. It then forwards vt to an initiator process which calculates GVT to be the minimum vt received from all processes.

In reality, instantaneous global snapshots are not possible, thus we cannot determine an exact value for RT. (We assume that a global clock is not available). However, we can set upper and lower bounds on RT at each process by creating intervals that overlap in real time [GW92]. Each process can then forward information to the initiator as it leaves the interval. This is the approach taken in most global virtual time algorithms. We will first describe how GVT can be determined from a set of intervals, and then review how different algorithms create the intervals.

Let $[start_i, stop_i]$ denote an interval at process i. The set of intervals $\{[start_i, stop_i] \mid i \in \text{LP}\}$ must share a common real time, RT. Each process knows that RT occurs within the interval, but does not know exactly where it occurs within the interval. Figure 1 shows an example.

Each process i calculates a virtual time vt_i and forwards it to the initiator at $stop_i$. Figure 2 shows exactly how vt_i is calculated. The initiator collects

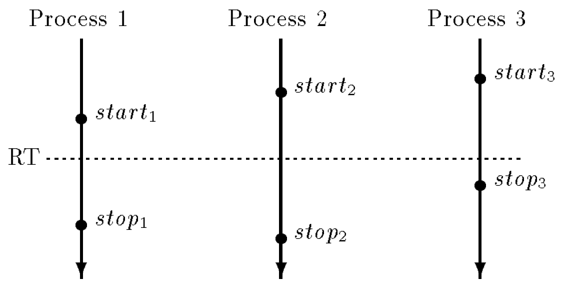

Figure 1: Virtual time intervals overlapping in real time.

$$\begin{aligned}
E1_i &:= \{e.ts \mid e \text{ was sent from } i \text{ before } start_i \ \wedge \\ &\qquad e \text{ has not been acknowledged } \} \\
E2_i &:= \{e.ts \mid e \text{ was sent from } i \text{ between } start_i \\ &\qquad \text{and } stop_i \ \} \\
vt_i &:= min(E1_i \cup E2_i \cup \{LVT_i \ at \ stop_i\})
\end{aligned}$$

Figure 2: Calculating vt_i at process i. (For an event e, $e.ts$ refers to its timestamp.)

the values of all vt_i, determines the minimum, and asserts that the new GVT equals this minimum value. Note that if $start_i = stop_i$ then the above algorithm is equivalent to the instantaneous global snapshot version.

Using this approach, calculation of GVT involves four phases. The START and STOP phases to generate $start_i$ and $stop_i$ at each process, the COLLECT phase to collect the information needed to calculate GVT, and the NOTIFY phase to notify all processes of the updated GVT. The STOP and COLLECT phases can often be combined. Similarly, the NOTIFY phase from a previous round can be combined with the START phase of the next round. We note here that the latency of this approach is the time required to execute the STOP and COLLECT phases.

Bellenot [Bel90] describes the algorithm used in the JPL Mark III Hypercube implementation of Time Warp (we refer to this algorithm as the *broadcast* algorithm). This algorithm uses broadcasts to create the overlapping intervals. An initiator broadcasts a START message to all processors. After all processes respond with a START_ACK message, the initiator broadcasts a STOP message and waits for STOP_ACK messages. Each processor defines defines $start_i$ to be the time at which START was received, and $stop_i$ to be the time at which STOP was received. Each process calculates vt_i immediately after receiving STOP and forwards the result to the initiator as part of the STOP_ACK. The initiator collects vt_i from all processes and calculates GVT to be the minimum of all vt_i.

Bellenot [Bel90] improves on the Mark III algorithm by using a binary tree forwarding mechanism (we refer to this algorithm as the *tree* algorithm). Instead of broadcasting START and STOP, they are propagated throughout the system using binary trees. Two binary trees are embedded in the process graph and their leaves are connected as shown in figure 3. The root of the left tree sends START to both children, and the children propagate the message. When the right root receives START from both children, it sends STOP along with vt_i to both children, who propagate the message. The minimum vt_i can be calculated as STOP propagates to the left root. The left root determines GVT when it receives STOP from both children.

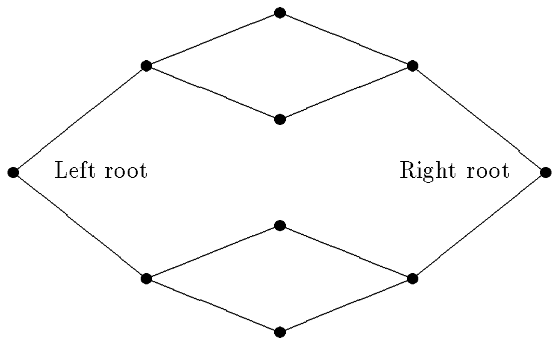

Figure 3: Message paths for the Bellenot's tree algorithm.

Another variant is to impose a ring on the processes and pass a token around the ring[Bel90] (we refer to this as the *token* algorithm). The same token implements the three phases for three different rounds. When a process receives a token, it defines $start_i$ for round n, $stop_i$ for round $n-1$, and propagates GVT for round $n-2$.

There are algorithms that do not use the overlapping intervals technique. Concepcion describes a hierarchical approach with tokens in [CK91]. He defines an intersecting hierarchy of token rings. Only half the nodes in the system are used for simulation; the other half are used for GVT calculation and other system duties. The lowest level tokens interact with the simulation processes. These tokens carry the minimum vt_i for all the nodes on it's ring. The upper level tokens propagate vt_i up to the highest level, at which point GVT is calculated.

3 Notation

In this section we define the components of an event timestamp and a process. We describe the algorithm for one initiator and one active round only. The reader's attention is directed to the new component: rollback counts. A process must maintain its rollback count and include it as part of the timestamp of every message it generates. The rollback count is defined later in this section.

LP	=	Set of Local Processes
IP	=	Initiator Process, IP $\in$ LP
GVT	=	Global Virtual Time
TVT	=	Target Virtual Time
TARGET	=	Target Event

```
Each event $e$ contains {
    $e.sndr$  =  process that sent event
    $e.rcvr$  =  process to receive event
    $e.ts$    =  timestamp
    $e.rb$    =  rollback count
}
```

Each process must maintain a set of event queues. This overhead already exists in optimistic distributed simulation systems such as Time Warp. Thus it is not an overhead of the algorithm; we just make use of the information.

A *relevant rollback* at process i is defined to be one which resets LVT_i from a value greater or equal TVT to a value less than TVT. The *rollback count*, rb_i, for process i is a non-negative integer value equal to the number of *relevant rollbacks* experienced by i. An event e is a *relevant event* if and only if $GVT \leq e.ts < TVT$. Note that an event is relevant if and only if it can cause a relevant rollback. Relevant events are the events that we keep track of for a given round of the algorithm.

As a final comment, we wish to stress that this algorithm *detects* when GVT $\geq$ TVT as opposed to *calculating* a value for GVT.

4 Algorithm

This algorithm assumes that messages are not acknowledged, channels are reliable, and message delivery is not ordered (non-FIFO). For simplicity, we describe the algorithm for the case where there is only one initiator. Upon initiating a round, the initiator must wait until that round is completed before initiating another. It is easy to extend the algorithm to handle multiple rounds concurrently (at most N concurrent rounds would be needed, where N is the number of processes in the simulation).

When the initiator process, IP, begins to run low on memory, it sets TVT equal to its LVT and initiates a round by initializing its local data and sending a TARGET event to each process. Choosing the target virtual time equal to LVT ensures that exact_GVT $\leq$ TVT. The timestamp of TARGET is set to TVT, thus each process will execute TARGET when its local virtual time equals TVT. IP then waits for the processes to respond to the TARGET events.

IP: To initiate a round

```
for each i $\in$ LP do {
  $\mathrm{Nsent}_i$ := 0
  $\mathrm{Nexec}_i$ := 0
  $\mathrm{Rb}_i$ := $-1$
  $\mathrm{RV}_i$ := 0
  send TARGET to $i$ with timestamp TVT
}
```

Now let us consider what happens at a process $p \in$ LP that receives a TARGET event. When the TARGET event is received, it is placed in the pendingQ and executed in turn. However, a TARGET event is not really executed, it just triggers the algorithm at p. When it comes time to execute the TARGET event, p gathers some local information and sends a REPORT message to IP. If multiple events have a timestamp of TVT, the TARGET event should be the first one executed (this is not necessary, but it decreases the latency of the calculation).

Note that if TARGET arrives at p when its local virtual time is greater than or equal to TVT, then p may roll back. However, TARGET will not effect $state_i$, thus a better idea is to "execute" TARGET without rolling back (similar to lazy reevaluation[RFBJ90]).

LP: To execute TARGET event

```
nexec := size of
    { e $\in$ executedQ | (GVT $\leq$ e.ts $<$ TVT) }
nsent := size of { e $\in$ outQ | (GVT $\leq$ e.ts $<$ TVT) }
for each $i \in$ LP do
  vclk[$i$] := max ({0} $\cup$ {e.rb | e $\in$ executedQ $\wedge$
                    e.sndr = $i$ $\wedge$ (GVT $\leq$ e.ts $<$ TVT)})
REPORT := <nsent, nexec, rb, vclk>
send REPORT to IP
CutMark /* For proof: this defines part of VCUT */
```

If p incurs a relevant rollback, then p's roll back count is incremented. During a relevant rollback the TARGET event is not canceled; instead it is moved back to the pendingQ. As p executes forward after the rollback, the TARGET event will be replayed and another REPORT message will be generated. The new REPORT message will contain the new rollback count. Outgoing events will also include the new rollback count, which will be used at the receivers to construct the rollback vectors that are included in their REPORT messages.

The REPORT message contains information that IP uses to determine when GVT $\geq$ TVT. It contains four fields. *Nexec* is the number of relevant events that p has executed. *Nsent* is the number of relevant events p has generated. The REPORT message also contains p's rollback count and a rollback vector.

The *rollback vector* is a vector clock [Mat89] built from the rollback counts included in each event's timestamp. If, at process p, $vclk[i] = n$ then p received a relevant event from i after i's n^{th} relevant rollback. Note that the rollback vector is produced from the event queues; the only tag appended to messages is the integer valued rollback count *rb*.

Now we consider IP to see how it handles incoming REPORT messages. IP collects reports from all processes and uses the information contained within to determine when GVT $\geq$ TVT.

IP evaluates the expression TVT_Validated, which is true if and only if GVT $\geq$ TVT. Due to rollback, a process may send more than one report, however, TVT_Validated forces IP to wait for the correct set of REPORT messages. Once TVT_Validated becomes true, no more REPORT messages will be received.

IP keeps track of incoming reports in local data structures. When a report is received from p, the *rb*, *nsent*, and *nexec* parameters from the report are copied directly to Rb_p, $Nsent_p$, and $Nexec_p$. The *vclk* vector parameter is used to update the rollback vector RV. If process p's $vclk[i] = n$, then i sent to p an event e such that GVT $\leq e.ts <$ TVT, and i incurred n relevant rollbacks prior to generating e. This means that i must submit a report *after* it's n^{th} relevant rollback in order for TVT to be validated. RV_i contains the rollback count needed from i; in this case it would be n (unless RV_i was already greater than n from a different report that was received out of order).

IP: To receive REPORT=<nsent,nexec,rb,vclk> from p

```
if ($\mathrm{Rb}_p$ < rb) {
  $\mathrm{Rb}_p$ := rb
  $\mathrm{Nsent}_p$ := nsent
  $\mathrm{Nexec}_p$ := nexec
  for each $i \in$ LP
    $\mathrm{RV}_i$ := max(vclk[$i$],$\mathrm{RV}_i$)
  if (TVT_Validated) then
    Assert GVT = TVT
} else
  /* report received our of order, ignore it          */
```

IP: To evaluate TVT_Validated

$$\text{TVT\_Validated} \Leftrightarrow (\forall i \in \text{LP} : \text{Rb}_i \geq \text{RV}_i) \wedge \left(\textstyle\sum_{i\in \text{LP}} \text{Nsent}_i = \sum_{i\in \text{LP}} \text{Nexec}_i\right)$$

TVT_Validated is true only when every process has submitted an up to date report and the sum of all relevant events generated equals the sum of all relevant events executed. We prove the correctness of this algorithm in the next section.

5 Proof

We prove that TVT_Validated is TRUE if and only if exact_GVT $\geq$ TVT. First we develop the notion of a virtual cut, then define a few notational conveniences, followed by some lemmas and finally the proof.

Let VCUT be a cut in virtual time (similar to Mattern's consistent cuts [Mat89]) defined by the occurrence of CutMark in the procedure LP:To_execute_TARGET_event. VCUT corresponds to the most recent time each process submitted a report. We say VCUT is *complete* if and only if each process has submitted at least one REPORT message.

Messages can cross VCUT in both the forward and backward directions. A message m travels *backward* across VCUT if and only if its sender has not submitted a REPORT message since sending m, and its receiver has submitted a REPORT message since receiving m. A message m travels *forward* across VCUT if and only if its sender has submitted a REPORT message since sending m, and its receiver has not submitted a REPORT message since receiving m. Note that we are only concerned with relevant messages that cross VCUT.

For notational convenience we define the following:

$$\begin{aligned}
\text{NS} &\triangleq \textstyle\sum_{i\in \text{LP}} \text{Nsent}_i \\
\text{NX} &\triangleq \textstyle\sum_{i\in \text{LP}} \text{Nexec}_i \\
\text{S} &\triangleq \{\text{event } e \mid \exists j : e \in \text{outQ}_j \wedge \\
&\qquad\qquad \text{GVT} \leq e.ts < TVT\ \} \\
\text{X} &\triangleq \{\text{event } e \mid \exists j : e \in \text{executedQ}_j \wedge \\
&\qquad\qquad \text{GVT} \leq e.ts < TVT\ \}
\end{aligned}$$

NS is the sum of the *nsent* parameters of the REPORT messages, which is equal to the total number of relevant messages generated before VCUT. *NX* is the sum of the *nexec* parameters of the REPORT messages, which is equal to the total number of relevant messages executed before VCUT. S is the set of relevant events generated; X is the set of relevant events executed. Note that NS and NX correspond to VCUT, while S and X correspond to a real time. If, as in the unattainable instantaneous snapshot algorithm, CutMark_i occurred at real time RT for all i, then NS and NX would equal the cardinality of S and X. In fact, part of the proof establishes that this is true if TVT_Validated is true.

To avoid confusion, we point out that Rb_i is a variable in IP, while rb_i is a variable at process $i \in LP$. Our goal is to show that we detect GVT $\geq$ TVT if and only if it occurs:

$$\begin{array}{ll} \textit{detect}: & (\forall i \in \text{LP} : \text{Rb}_i \geq \text{RV}_i) \wedge (\text{NS} = \text{NX}) \\ \Leftrightarrow & \\ \textit{occur}: & (\forall i : LVT_i \geq TVT) \wedge (S = X) \end{array}$$

Lemma 1 *If* VCUT *is incomplete then* detect *is false.*

Proof: If VCUT is incomplete, then there exists a process i that has not submitted a REPORT message. Thus $\text{Rb}_i = -1$ (its initial value), which implies that $\text{Rb}_i < \text{RV}_i$. Therefore, *detect* is false. ■

Lemma 2 *If a relevant message* m *crosses a complete VCUT, then* detect *is false.*

Proof: If m travels backward across VCUT, then *m.sndr*'s rollback count shows up in *m.rcvr*'s REPORT message as $\text{vclk}_{m.sndr}$, which in turn shows up as $\text{RV}_{m.sndr}$. This value is greater than $\text{Rb}_{m.sndr}$ since m travels backwards. Thus $\text{Rb}_{m.sndr} < \text{RV}_{m.sndr}$. Therefore *detect* is false for this case.

On the other hand, if m travels forward across VCUT, then consider two cases.

Case One: No relevant message travels backward across VCUT. Then m creates a surplus of relevant messages sent forward across VCUT. Thus NS $\neq$ NX, and *detect* is false for case 1.

Case Two: There is a relevant message m' that travels backward across VCUT. Then m' causes *detect* to be false, as shown above. ■

Theorem 1 *detect* $\Rightarrow$ *occur*

Proof: We will prove the contrapositive: $\neg occur \Rightarrow \neg detect$. There are two cases to consider.

Case 1: $\exists i : LVT_i < TVT$.
If i is in its 0th incarnation ($\text{rb}_i = 0$), then VCUT is not complete, thus by lemma 1 *detect* is false. If $\text{rb}_i = n$, for $n > 0$, then there must exist a message that crosses VCUT. Thus by lemma 2, *detect* is false.

Case 2: $S \neq X$.
Then $X \subset S$ (since by definition, $X \subseteq S$, i.e. events must be created before they can be executed). If VCUT is not complete then *detect* is false. If VCUT is complete, then, since $X \subset S$, there must exist a message that crosses VCUT. Thus by lemma 2, *detect* is false. ■

Theorem 2 *occur* $\Rightarrow$ *detect*

Proof: We assume *occur* and show *detect*. If *occur* holds, then no process will rollback over TVT. Thus no process will incur any more relevant rollbacks, and each processes has submitted its last REPORT message. This implies that, for all i, Rb_i is greater than or equal to every rollback count value i included in the timestamps of its outgoing events. Thus, for all i, $\text{Rb}_i \geq \text{RV}_i$.

During the interval that begins when exact_GVT exceeds TVT and ends when i sends its last REPORT message, nsent and nexec at i will not have changed. Thus NS = | S |, and NX = | X |. Since S = X, then NS = NX.

Thus for all i, $\text{Rb}_i \geq \text{RV}_i$ and NS = NX. Therefore *detect* is true. ■

6 Discussion

6.1 Latency

We compare the latency of our algorithm with the latencies of the broadcast, tree, and token algorithms described in section 2. We use the definition of latency presented in section 1 and assume that there are N processes in the system.

Recall that the broadcast, tree and token algorithms all use the same strategy of creating overlapping intervals (figure 1). The latency of an algorithm using this strategy is the time required to terminate these intervals and gather information. As mentioned in section 2, this is the time required to execute the STOP and the COLLECT phases.

The broadcast algorithm terminates the intervals with a 1 to N broadcast followed by an N to 1 collection. The initiator must send and receive $2(N-1)$ messages, which contributes $\mathcal{O}(t_p N)$ to the latency. The message path has length two, which contributes $\mathcal{O}(t_m + t_p)$. Thus the latency is $\mathcal{O}(t_m + t_p N)$. Similar calculations result in a latency of $\mathcal{O}((t_p + t_m) \log N)$ for the tree algorithm and a latency of $\mathcal{O}((t_p + t_m)N)$ for the token algorithm.

Our algorithm has latency $t_m + 2t_p = \mathcal{O}(t_m + t_p)$ since the message path has length one. As soon as a process determines that it is possible that GVT is greater than TVT, it sends a message directly to the initiator. It is clear that $t_m + 2t_p$ is the minimum latency of any GVT algorithm on a distributed system since this is the time needed to send one message between two processes (send + transit + receive $= t_p + t_m + t_p$). Figure 4 summarizes the latencies of these algorithms.

Algorithm	GVT Latency
Ours	$t_m + t_p$
Broadcast	$t_m + t_p N$
Tree	$(t_m + t_p) \log N$
Token	$(t_m + t_p) N$

Figure 4: Comparison of asymptotic latency.

6.2 Message Overhead

The broadcast, tree, and token algorithms each require that messages are acknowledged; our algorithm does not require messages to be acknowledged. The message overhead of these algorithms is shown in figure 5 in terms of M,N, and R. M is the total number of message acknowledgments; N is the number of processes, and R is the total number of relevant rollbacks.

Figure 5 shows two components of the overhead: fixed (per round), and variable (per simulation).

Algorithm	Fixed (per round)	Variable (per simulation)
Ours	N	R
Broadcast	$4(N-1)$	M
Tree	$3N$ *	M
Token	$2N$	M

Figure 5: Comparison of message overhead.
(* indicates an approximation)

The fixed overhead is incurred for each round of the algorithm. For our algorithm this overhead is N since each process sends at least one report to the initiator. The fixed overhead for the other algorithms depends on how they create the overlapping intervals. For example, the broadcast approach requires $N-1$ messages for each of the four phases.

The variable overhead is incurred once during the entire simulation. We characterize this overhead on a per simulation basis because there is no way to determine how much of this overhead will occur for a given round. Each of the other algorithms require message acknowledgments, thus the variable overhead per simulation is M.

Our algorithm does not require message acknowledgments, but does require one message for each relevant event. Since a rollback can only occur after a message is received, we know that $R \leq M$. In the analysis of a large simulation, [WH$^+$89] find that the number or rollbacks is about one fourth of the number of messages. Since not every rollback will be a relevant rollback, R would be less than $M/4$ (in that example). Thus the overhead of our algorithm is significantly less than the any of the broadcast, tree, or token algorithms.

7 Conclusion

We have presented and proved the correctness of an algorithm for determining global virtual time which is based on a novel idea. The algorithm enables an initiator process to decide upon a virtual time, TVT, that will be the next computed value of GVT. The condition "GVT $\geq$ TVT" is then detected with minimal latency (the delay between the actual occurrence of the condition and its detection). This algorithm has two important and unique characteristics: 1) minimal latency, and 2) specification of TVT which will be the next computed value of GVT.

Contrary to previous algorithms, this algorithm does not require messages to be acknowledged. This results in a large savings on the number of messages required in the overall simulation. In addition, the fixed overhead per GVT calculation is less than other well known algorithms.

This algorithm eliminates a problem with many previous global virtual time algorithms: the avalanche of messages during the collection phase. In our algorithm, the collection phase is spread out over time, thereby eliminating the avalanche problem.

One possible application is with interactive simulators where it is desirable to get global virtual time updates at regular virtual time intervals with minimal latency. These characteristics allow output to appear continuous and up to date.

8 Acknowledgments

We would like to thank the referees of this paper for providing insightful comments. We would also like to thank Steven Glicker and David Brant who reviewed earlier versions of this paper.

References

[Bel90] Steven Bellenot. Global virtual time algorithms. In *Proc. of the SCS MultiConference on Distributed Simulation*, volume 22(1), pages 122–127. Society for Computer Simulation, January 1990.

[BL89] S. Bellenot and M. Di Loreto. Tools for measuring the performance and diagnosing the behavior of distributed simulations using time warp. In *Proc. of the SCS MultiConference on Distributed Simulation*, volume 21(2), pages 145–150. Society for Computer Simulation, March 1989.

[CK91] Arturo I. Concepcion and Scott G. Kelly. Computing global virtual time using the multi-level token passing algorithm. In *Proc. of the SCS MultiConference on Distributed Simulation*, volume 23(1), pages 63–68. Society for Computer Simulation, January 1991.

[CM81] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(11), April 1981.

[Fuj90] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10), October 1990.

[GW92] V.K. Garg and B. Waldecker. Detection of unstable predicates in distributed programs. In *Proc. of 12th Conference on the Foundations of Software Technology & Theoretical Computer Science*, pages 253–264. Springer Verlag, December 1992. Lecture Notes in Computer Science 652.

[Jef85] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3), July 1985.

[Jef87] David Jefferson. The time warp operating system. In *Eleventh Symposium on Operating System Principles*, pages 77–93, November 1987. vol 21, no 5.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proc. of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V. (North-Holland), 1989.

[Rey88] Paul Reynolds. A spectrum of options for parallel simulation protocols. In *Winter Simulation Conference Proceedings*, pages 325–332, December 1988.

[RFBJ90] P Reiher, R. Fujimoto, S. Bellenot, and D. Jefferson. Cancellation strategies in optimistic execution systems. In *Proc. of the SCS MultiConference on Distributed Simulation*, volume 22(1), pages 112–121. Society for Computer Simulation, January 1990.

[Tel91] Gerard Tel. *Topics in Distributed Algorithms.* Cambridge University Press, Cambridge, England, 1991.

[WH+89] Frederick Wieland, Lawrence Hawley, et al. Distributed combat simulation and time warp: The model and its performance. In *Proc. of the SCS MultiConference on Distributed Simulation*, volume 21(2), pages 14–20. Society for Computer Simulation, March 1989.

Alexander I. Tomlinson is currently enrolled in the PhD program in the Department of Electrical and Computer Engineering at The University of Texas at Austin. His research focus is distributed systems. In the past Alex has worked at Sandia National Laboratories, Hewlett-Packard Company, and Digital Telephone Systems (a subsidiary of Harris Corporation). Alex received his BS in Computer Engineering from Carnegie Mellon University in 1986 where he was a University Scholar and earned an NCAA Academic All American award for his role on the varsity soccer team. He received his MS from UT Austin in 1992 with the support of a University Fellowship.

Vijay K. Garg received his Bachelor of Technology degree in computer engineering from the Indian Institute of Technology, Kanpur, in 1984. He continued his education at the University of California, Berkeley where he received his MS in 1985 and Ph.D. in 1988 in Electrical Engineering and Computer Science. He is currently an assistant professor in the Department of Electrical and Computer Engineering at the University of Texas, Austin. His research interests are in the areas of distributed systems and supervisory control of discrete event systems.