Fault-Tolerant Distributed Simulation

Om P. Damani — Dept. of Comp. Sc.

Vijay K. Garg* — Dept. of Elec. and Comp. Eng.

University of Texas at Austin, Austin, TX, 78712

http://maple.ece.utexas.edu/

Abstract

We present the first discussion of the issues involved in fault-tolerant distributed simulation. We integrate an existing optimistic fault-tolerance scheme with an existing optimistic distributed simulation scheme. We make use of the novel insight that a failure can be modeled as a straggler event with the receive time equal to the virtual time of the last checkpoint saved on stable storage. This results in saving of implementation efforts, as well as reduced overheads. We define stable global virtual time (SGVT), as the virtual time such that no state with a lower timestamp will ever be rolled back. We make a simple change in existing GVT algorithms to compute SGVT. Our use of transitive dependency tracking eliminates antimessages. We club LPs into clusters to minimize stable storage access time.

1 Introduction

In a distributed simulation, the failure of any LP causes the entire computation to halt. As the number of LPs taking part in a simulation increases, so does the probability that one of them will crash during the simulation. Simply restarting the failed LP may leave the simulation in an inconsistent state [5]. So far, the only recourse in such a situation has been to restart the entire

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

system. However, restarting the system is unacceptable for simulations that run for hours or days. Clearly, some form of fault-tolerance is required to minimize the wasted computation.

A process may crash due to a bug in the application code, simulator code, or operating system code. Even when all the code is correct, the code being run with a distributed simulator may have been written for a sequential simulator [15]. In such scenarios, it is difficult to find and correct the source of the crash. Even when the bug lies entirely in the application code, the user of an application may not be the developer of the code. So the user may be unable (or unwilling) to debug the application. The situation will be hopeless, if every time the system is restarted, the same bug were to lead to the same crash. Luckily, experiments with different kind of software systems have shown that most of the bugs encountered in practice are transient [9, 19]. For example, a process may crash if it does not check for a 'null pointer' value on returning from a memory allocation routine. When the process is restarted, more memory may be available, thereby avoiding the crash. Crashes should especially be transient in the optimistic simulation, where a different message ordering or a different process scheduling results in a different execution. Therefore, restarting the failed process is a viable option, provided steps are taken to ensure that the resulting system state is consistent: for example, by rolling back the other processes.

In addition to software failures, we should also be able to tolerate hardware failures. Hardware failures may be in the form of processor malfunctioning, power failure or someone tripping over the connecting wires. We need to restart the failed processes on same or different processor.

We assume that processes fail by simply crashing and they do not send out any erroneous messages or do some other harm. A process loses all its volatile memory in a failure. To reduce the amount of wasted computation, it periodically writes its checkpoints to stable storage. After a failure, it is restarted from its last stable checkpoint. We model a failure as a straggler event with a timestamp equal to the timestamp of the highest checkpoint saved on stable storage. In this model, computation lost due to a failure can be treated in the same way as a computation rolled back due to a straggler.

Note that a naive application of the above idea to Time Warp does not work. In Time Warp, when an LP receives a straggler, it sends out antimessages for the messages in its output queue. But the output queue is lost in a failure. In fact, all volatile storage is lost in a failure. Hence to tolerate failures, we need a simulation scheme that does not use any state information of the rolled back computation.

In [6], we presented one such simulation scheme. We integrate that scheme with the optimistic fault-tolerant method presented in [21] to come up with a low overhead, fault-tolerant, optimistic, distributed simulation protocol. The contributions of this paper are following:

- Development of a formal framework for fault-tolerant distributed simulation.
- Modeling of a *failure* as a straggler to facilitate low overhead fault-tolerance.
- Integration of the optimistic simulation scheme in [6] with the optimistic fault-tolerance scheme in [21].
- Definition of *Stable GVT*: Global Virtual Time in failure prone systems.
- Identification of the issues involved in the design of a fault-tolerant simulation.

There has not been much discussion in simulation literature about fault-tolerance. In the seminal paper on Virtual Time [11], Jefferson mentions that processes may co-ordinate to take a consistent snapshot and save it on stable storage. When any process fails, the entire system may roll back to the last snapshot. In contrast, our method rolls back only those states that are dependent on a lost state, and thus minimizes the wasted computation. In [1], a replication based strategy is presented to provide fault-tolerance. Our scheme has much lower overhead than that of the replication.

Our scheme does not require any synchronization. Checkpoints are saved to stable storage asynchronously, i.e., while computation is in progress. Any failure during the saving of checkpoints is indistinguishable from a scenario, where no checkpoints were being saved at that time. Similar treatment of failure and straggler coupled with asynchronous stable storage operations results in reduced overhead for the fault-tolerance.

2 Background in Fault-Tolerance

There are many well-known techniques for providing fault-tolerance to distributed applications. They can be classified into two broad categories: replication based [4] and checkpointing based [7]. Replication based techniques consume extra resources and have synchronization overhead to maintain consistencies between replicas. In checkpointing based techniques, checkpoints are saved on stable storage, so that a failed process can be restarted from its last stable checkpoint. We only consider the checkpointing based schemes. Checkpointing schemes require synchronization between processes, or else they suffer from the *domino effect*, where all the processes may rollback to their initial state [16]. Note that checkpoints need not be immediately saved on stable storage. Applications with high checkpointing activity may take many checkpoints before writing them to stable storage.

To avoid both synchronization and domino effect, some schemes also save the received messages on stable storage. This is called *message logging*. After a failure, a process restores its last checkpoint and replays the logged messages. It may inform other processes about its failure and may also request some information from other processes. This method is similar to 'periodic checkpointing' and 'coast forward' mechanism used in simulation [13]. Message logging schemes can be divided into three categories [7]: pessimistic, optimistic, and causal. Pessimistic and causal schemes recreate the pre-failure computation. Pessimistic logging requires that each message be synchronously saved in stable storage, before a process acts on it. This is unacceptable in distributed simulation where message activity is very high. Causal logging piggybacks the processing order of messages on each outgoing message. This will also result in high overhead due to high message activity in simulation.

In optimistic schemes, messages are saved to stable storage asynchronously. As a result, processing order of messages may be lost in a failure. So the execution after a failure may be different from the pre-failure execution, resulting in *lost* states. States dependent on a lost state are called *orphan* states. Correctness of computation requires that these orphan states be rolled back. It is no coincidence that this reminds one of optimistic simulation. The seminal work on optimistic recovery by Strom and Yemini [20] was inspired by the Time Warp mechanism.

In an optimistic scheme, a process may fail without logging any of its received messages since its last checkpoint. This implies that messages need not be logged independent of checkpoints. To reduce the cost of accessing stable storage, messages can be logged only when checkpoints are being written to stable storage. This makes optimistic schemes well suited for distributed simulation, where message activity is very high.

3 Model of Simulation

We consider an event-driven optimistic simulation. The execution of an LP consists of a sequence of states where the state transition is caused by the execution of an event. In addition to causing a state transition, executing an event may also schedule new events for other LPs (or the local LP) by sending messages. When LP $P1$ acts on a message from $P2$, $P1$ becomes dependent on $P2$. This dependency relation is transitive.

An LP periodically saves its checkpoints on stable storage. After a failure, an LP restores its last checkpoint from stable storage and starts executing from there. The resulting execution may be different from the pre-failure execution. States that are not re-executed in post-failure computation are called *lost* states.

The arrival of a straggler causes an LP to roll back. A state that is lost, rolled back, or transitively dependent on a lost or rolled back state is called an *orphan state*. We denote transitive dependency by the *happened before* ($\rightarrow$) relation, which we define later in this section. As stated earlier, we model a failure as a straggler event with a timestamp equal to the timestamp of the highest checkpoint saved on stable storage. We formally define *orphan* states as follows:

$straggled(s) \equiv$ state s has been rolled back due to a straggler

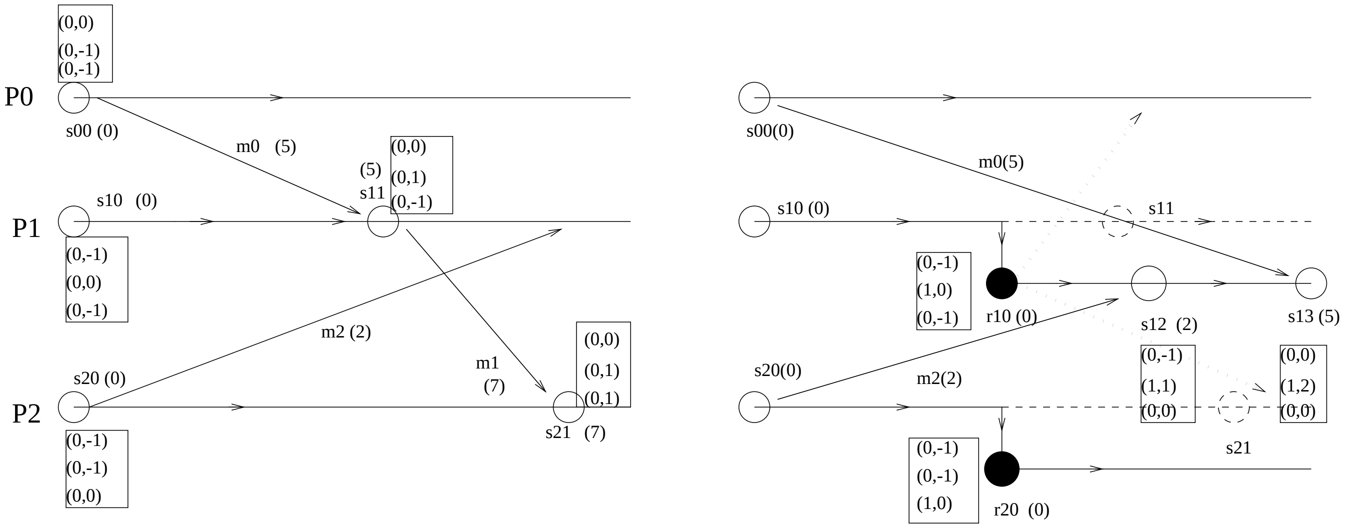
$stable(P) \equiv$ timestamp of last stable checkpoint of LP P

$failure(P) \equiv$ event scheduled for P at time $stable(P)$

$orphan(s) \equiv straggled(s) \vee \exists u : (orphan(u) \wedge u \rightarrow s\ )$

Note that in this model lost states are treated as straggled states. A message sent from an orphan state is called an *orphan message*. For correctness of a simulation, all orphan states must be rolled back and all orphan messages must be discarded. To distinguish the computation before and after the rollback, we say that an LP starts a new *incarnation* after each rollback.

An example of a distributed simulation is shown in Figure 1. Numbers shown in parentheses are either the virtual times of states or the virtual times of scheduled events carried by the messages. Solid lines indicate useful computation, while dashed lines indicate rolled back computation. In Figure 1(a), $s00$ schedules an event for $P1$ at time 5 by sending message $m0$. $P1$ optimistically executes this event, resulting in the state transition from $s10$ to $s11$, and schedules an event for $P2$ at time 7 by sending message $m1$. Then $P1$ receives message $m2$ which schedules an event at time 2 and is detected as a straggler. Execution after the arrival of this straggler is shown



(a) (b)

Figure 1: A Distributed Simulation. (a) Pre-straggler(failure) computation. (b) Post-straggler(failure) computation.

in Figure 1(b). $P1$ rolls back by restoring the state $s10$. It then takes the actions needed for maintaining the correctness of the simulation, which, for our scheme, consists of broadcasting a rollback announcement (shown by dotted arrows). It restarts from $r10$, acts on $m2$, and then acts on $m0$. Upon receiving the rollback announcement from $P1$, $P2$ realizes that it is dependent on a rolled back state and so it also rolls back, restores state $s20$, takes actions needed, and restarts from state $r20$. Finally, the orphan message $m1$ is discarded by $P2$. In [6] we have shown that by tracking transitive dependency, only the LP receiving the straggler needs to send a rollback announcement. All other LPs, on receiving the announcement roll back their orphan states and discard the received orphan messages. Other LPs do not need to send rollback announcement while rolling back in response to a rollback. Simulation proceeds correctly, without requiring antimessages. Note that we assume aggressive cancellation [18].

Now we describe a simulation in a failure-prone system. Let us look at Figure 1 again. Assume that the system has performed the computation shown in Figure 1(a), but $P1$ has not yet received the message $m2$. Let $P1$ fail before it receives the message $m2$. It loses its volatile memory, which includes the message $m0$. Now Figure 1(b) shows the post-failure computation. $P1$ restores its last stable checkpoint $s10$. It then broadcasts a failure-announcement. On receiving this announcement, $P0$ and $P2$ resend the messages $m0$ and $m2$ as they might have been lost in the failure. $P2$ also realizes that it is dependent on a lost state and rolls back, restores state $s20$, takes actions needed, and restarts from state r20. $P1$ on the other hand processes $m0$ and $m2$ in the correct order. This shows how we handle a failure and a straggler in the same way.

In order to track transitive dependency in presence of rollback, in [6], we extended the *happened before*($\rightarrow$) relation defined by Lamport [12]. Intuitively, state s happens before state u, if, in the simulation diagram, there exists a directed path from s to u consisting of solid or dashed arrows. Failure or rollback announcements, denoted by dotted arrows, do not contribute to the happened before relation. For example, in Figure 1(a), $s10 \rightarrow s11$ and $s00 \rightarrow s21$, and in Figure 1(b) $s11 \not\rightarrow r10$. Saying s happened before u is equivalent to saying that u is *transitively dependent* on s.

4 The Fault-Tolerant Protocol

For fault-tolerance, checkpoints need to be saved on the stable storage. The overhead of separately accessing stable storage for the checkpoint of each LP is unacceptable. Therefore we club LPs into clusters and take checkpoint of entire clusters. The idea of clustering has already been used in [17] and [2]. In [17], inter-cluster execution is conservative, whereas intra-cluster execution is optimistic. In [2], inter-cluster execution is optimistic, whereas intra-cluster execution is sequential. We assume inter-cluster execution to be optimistic. Our scheme works with both conservative and optimistic policy for intra-cluster execution. For purpose of exposition, we assume that intra-cluster execution is sequential. Details of intra-cluster execution can be found in [2].

Since the simulation inside a cluster is sequential, the state of a cluster at a given virtual time is well-defined. This state includes the input messages in all the LPs input queues. The state also includes the cluster output queue, which is described later. Clusters periodically save their state on stable storage. For discussion we use the term 'state of a cluster'. For implementation, states of only those LPs need be saved on stable storage, that have changed since the last state saving operation. From here on, we refer to intra-cluster messages as 'internal' messages and inter-cluster messages as 'external' messages.

From here on, i,j refer to cluster numbers; v refers to incarnation number; s,u refer to states; P_i refers to cluster i ; m refers to a message and e refers to an event.

```
Cluster P_i :

type   entry =    (int inc,   int sii) ;          /* incarnation, state interval index */
var    cvt : real;                                /* cluster virtual time */
       dv : array[0..n-1] of entry;               /* dependency vector */
       iet : array[0..n-1] of set of entry;       /* incarnation end table */
       token :    entry;                          /* rollback announcement */
       CIQ :      set of pointers ;               /* cluster input queue for external messages */
       COQ :      set of messages ;               /* cluster output queue for external messages */


Initialize :
    cvt = 0 ;
    ∀ j : dv[j].inc = 0 ; dv[j].sii = -1 ;
    dv[i].sii = 0 ;
    ∀ j : iet[j] = {} ; /* empty set */
    token = (0,-1) ; CIQ = {} ; COQ = {} ;
```

Figure 2: Data Structures used by a cluster

4.1 Data Structures

The data structures used by a cluster are shown in figure 2. We describe the main data structures below:

- **Dependency Vector**: To track dependency information, we use a *Dependency Vector(DV)* [6]. It has n entries, where n is the number of clusters in the system. Each entry contains an *incarnation number* and a *state interval index*. A state interval is the sequence of states between two events scheduled by the external messages. The index in the i^{th} entry of the DV of P_i corresponds to the number of external messages that P_i has acted on. The index in the j^{th} entry corresponds to the index of the latest state of P_j on which P_i depends. The incarnation number in the i^{th} entry is equal to the number of times P_i has rolled back. The incarnation number in the j^{th} entry is equal to the highest incarnation number of P_j on which P_i depends. Let entry en be a tuple (incarnation v, index t). We define a lexicographical ordering between entries as follows:

$en_1 < en_2 \equiv (v_1 < v_2) \vee [(v_1 = v_2) \wedge (t_1 < t_2)]$.

Dependency vectors are used to maintain transitive dependency information. Suppose P_i schedules an event e for P_j by sending a message m. P_i attaches its current DV to m. If m is neither an orphan nor a straggler, it is kept in the incoming queue by P_j. When the event corresponding to m is executed, P_j updates its DV with m's DV by taking the componentwise lexicographical maximum, as shown in the routine *Execute_message* in figure 3. An example of DV is shown in figure 1 where DV of each state is shown in a box near it.

In [6], entries in DV include the virtual time instead of the state interval index. This method does not work in the presence of failures. Let P receive two messages with the same scheduling time. Let P fail after scheduling one of the events. We need to distinguish between the states resulting from these two events. Hence we replace the timestamp in each DV entry with a state interval index.

In general, DV of the sending state needs to be attached with each message to correctly track transitive dependencies and detect orphans. But we reduce this overhead by making the observation that with clustering, DV needs to be attached with inter-cluster messages only. For intra-cluster messages, it suffices to track send time as the receiver's DV is always greater than or equal to the sender's DV.

- **Incarnation End Table**: Besides a dependency vector, each cluster also maintains an *incarnation end table*(iet). The j^{th} component of iet is a set of entries of the form (v, sii), where all states of the v^{th} incarnation of P_j with indices greater than sii have been rolled back. The iet allows a cluster to detect orphan messages.

- **Cluster Input Queue**: Both external and internal messages are kept in the destination LP's input queue. A cluster keeps track of the external messages by keeping pointers to them in cluster input queue(CIQ).

- **Cluster Output Queue**: There is one output queue per cluster called Cluster Output Queue(COQ). Only inter-cluster messages are kept in COQ. intra-cluster messages are not saved in any per LP output queue.

4.2 Protocol Description

The formal description of our protocol is given in figures 2 to 6. In addition to the receive time, internal messages also carry the send time, which is the same as the local virtual time (lvt) of the sending LP. External messages carry DV of the sender. Actions taken by an LP are shown in figure 3.

```
Logical Process LP : /* belongs to cluster P_i */
var lvt : real ;   /* local virtual time */
    input_queue : set of message ; /* keeps input messages */

Execute_message(m) :
    lvt = cvt = m.receive_time ; /* cvt is cluster virtual time */
    if m is an external message then
        ∀ j: dv[j] = max(dv[j], m.dv[j]) ; /* dv is cluster dependency vector */
        dv[i].sii++ ;
    Act on the event scheduled by m ;
Send_message(m)
    if intra_cluster(m) then
        put (m, lp.lvt, m.receive_time) in destination LP's input queue ;
    if inter_cluster(m) then
        send(m, dv, m.receive_time);
Rollback(ts)
    Restore the latest state s such that s.lvt ≤ ts ;
    Discard the states that follow s ;
    ∀m ∈ input_queue
        if m.send_time > ts then discard m; /* m is orphan */
```

Figure 3: Actions taken by an LP

Actions taken by a cluster on receiving an external message are shown in the figure 4. Upon receiving an external message m, P_i discards m if m is an orphan. This is the case when, for some j, P_i's iet and the j^{th} entry of m's DV indicate that m is dependent on a rolled back state

Receive_message(m) :

if $\exists j, t : ((m.dv[j].inc, t) \in iet[j]) \wedge (t < m.dv[j].sii)$

then discard m ; return ; /* m is orphan */

Copy m in input queue of the destination LP;

In the CIQ, add a pointer to m;

if $m.receive\_time < cvt$ **then**

Let s be the latest cluster checkpoint such that $s.cvt \leq m.receive\_time$;

$token = (s.dv[i])$; /* m is a straggler */

Broadcast($token$); /* P_i receives its own broadcast and rolls back. */

Block till all clusters acknowledge ;

Figure 4: Actions taken by a cluster on receiving an external message

of P_j. A straggler for any LP in the cluster is also considered a straggler for the entire cluster. If P_i detects that m is a straggler, it broadcasts a token containing the i^{th} entry (v, sii) of the DV of its highest checkpoint s such that the virtual time of s is no greater than the receive time of m. The token basically indicates that all states of incarnation v with index greater than sii are orphans. States dependent on any of these orphan states are also orphans. For simplicity of presentation, we show P_i rolling back in figure 5(after it receives its broadcast). In practice, it will roll back immediately.

Steps taken by a cluster on receiving a token are shown in figure 5. On receiving a token (v, sii) from P_j, P_i acknowledges the token and enters it in its incarnation end table. It discards all the orphan messages in the cluster input queue. A message is an orphan if it is dependent on a rolled back state of P_j. If the cluster is orphan then it restores the maximal non-orphan state. This is done by rolling back all orphan LPs and discarding the orphan messages in each LPs input queue. Note that in routine **Rollback** in figure 3, an LP does not have to rollback if it is not orphan. After the rollback, incarnation number is incremented. It does not broadcast a token, which is an important property of this protocol. Note that internal orphans are detected using a separate mechanism, as compared to external orphans. If the current state of a cluster is not orphan, then no LP and consequently no internal message can be orphan. Routine **Rollback** is not called for any LP. All external orphans are discarded using CIQ.

```
Receive_token(v, sii) from P_j :
    Send acknowledgment ;
    iet[j] = iet[j] ∪ {(v, sii)} ;
    ∀mp ∈ CIQ : /* mp : message pointer */
        if (*mp.dv[j].inc = v) ∧ (sii < *mp.dv[j].sii)
            then discard *mp ; /* orphan *mp */
    if (dv[j].inc = v) ∧ (sii < dv[j].sii) then
            /* Following actions are taken only when cluster is orphan */
        Let s be the latest checkpoint such that s.dv[j] ≤ (v, sii) ;
            /* s is the highest non-orphan state */
        ∀lp ∈ cluster: lp.Rollback(s.cvt) ;
        dv = s.dv ; dv[i].inc++ ;
```

Figure 5: Actions taken by a cluster on receiving a token

```
Restart /* after failure */
    Restore last checkpoint s from stable storage;
    Broadcast(s.dv[i]) ; /* incarnation and index of this checkpoint */
    Block till all clusters acknowledge.
```

Figure 6: Actions taken by a cluster on restarting after a failure

Steps taken by a cluster on restarting after a failure are shown in figure 6. The failure is handled in the same way as a straggler. After the failure, the cluster is restarted from its last checkpoint on stable storage. It broadcasts a token containing the incarnation number and the index of the restored state. Other clusters react to this token in the same way as they do to the token due to a straggler.

There is one important difference between a failure and a straggler, which we have not shown in figure 5 for clarity. In a failure, a cluster loses its volatile state, i.e., its *iet* and all the messages that it has received but not acted on till its last stable checkpoint. So on learning about the failure, other clusters must resend messages to the failed cluster. Of these messages, the failed cluster should replay only those messages, which it did not act on before the last checkpoint. For this purpose, we need each sender to put a sequence number on outgoing messages on a per cluster basis. We assume FIFO message delivery order. Each cluster keeps only the expected sequence number of next message to be received from every other cluster. Now sender needs to resend only those messages whose sequence number is greater than the sequence number of the last message received till the checkpoint. These sequence numbers are broadcast along with the failure announcement. Note that the above scheme can handle an arbitrary number of concurrent failures. When a processor fails, all clusters on that processor need to be restarted as if each one of them have failed independently.

A rolled back state is called *rollback inconsistent* with the states that occur after the corresponding rollback [15]. For example, in figure 1, $s11$ is rollback inconsistent with $s12$. Allowing a state to become dependent on two rollback inconsistent states have been identified by Nicol and Liu as a potential source of crash [15]. The next theorem shows that the our protocol avoids this problem.

Theorem 1 *A state cannot become dependent on two rollback inconsistent states.*

Proof: After a rollback, a process blocks till it receives acknowledgment of its rollback announcement from all processes. Therefore, all processes receive the rollback announcement before becoming dependent on a post-rollback state. Now, as per the routine **Receive_token** in figure 5, any state dependent on a rolled back state is rolled back on receiving the corresponding token. Hence no state can become dependent on two rollback inconsistent states. ∎

4.3 Correctness Proof

The following lemma states that Dependency Vectors correctly track transitive dependency information.

Lemma 1 *If s happens before u, then $s.dv \leq u.dv$.*

Proof. As in [6], the proof follows by the induction on the length of the path from s to u.
■

The next theorem proves that our protocol correctly completes simulation in the presence of failures.

Theorem 2 *At the end of the simulation, the following conditions are true:*

- *All LPs have received all the messages that they would have received in a sequential simulation.*
- *All LPs have processed all the messages in the increasing order of their receive time.*
- *All orphan states have been rolled back.*
- *All orphan messages have been discarded.*

Proof: To simplify the presentation, we assume that each cluster contains only a single LP. The proof can easily be extended to multiple LPs. First note that in the presence of reliable delivery, actions taken after receiving two tokens commute with each other and also the actions taken after receiving a token commute with a failure. Therefore, f concurrent failures are not different from the case where f processes fail one after another, such that between failures each process receives each others failure announcement and takes no other action. Hence, we only consider the single non-concurrent failure case.

We have modeled the failure as a straggler event. However, the failure also results in loss of the volatile state. We do not use any of the lost information even if the failure were truly a straggler event. The only information we need is the received messages and the incarnation end table entries. This information is collected from the other processes. The only tricky case is when the sender itself has failed and it cannot resend some message as the state that sent that message is *lost*. This is harmless because according to the protocol, messages sent from a lost state are also *orphan* and they anyway need to be discarded upon their receive.

So our only remaining proof obligation is to show that in absence of failures, our protocol handles the straggler messages and orphan states correctly. This proof follows directly from the proof of theorem 2 in [6]. The heart of the proof is that lemma 1 assures us that all orphan states are detected upon the arrival of the corresponding token. ■

4.4 Stable Global Virtual Time (SGVT)

Global Virtual Time(GVT) is defined as the virtual time such that no state prior to GVT will ever be rolled back [3]. Traditional methods for computing GVT fail in presence of failures. A failure of a cluster may result in the restoration of a state with the virtual time less than GVT.

It is interesting to note that our modeling of failure as a straggler event can directly be used in the standard GVT algorithm to come up with a value, which we call SGVT, such that no state with virtual time less than SGVT will ever be rolled back. Since failure is treated as a straggler, so a potential failure of process P can be treated as an unacknowledged message with time-stamp $stable(P)$.

GVT is approximated by taking the minimum of receive times of all unacknowledged messages and all the local virtual times of each process. We note that GVT can be approximated by the minimum of receive times of all unacknowledged messages and all the unprocessed messages in the input queue of a process, which for a process P, we denote by $unacked(P)$ and $unprocessed(P)$ respectively. Now we can define SGVT as follows:

$$T_i \equiv \min\{stable(P_i), unacked(P_i), unprocessed(P_i)\}$$
$$SGVT \equiv \min\{\forall i : T_i\}$$

Theorem 3 *No state with virtual time less than SGVT can ever be rolled back.*

Proof: Every rollback has a first cause in a straggler or a failure. A failure cannot restore a state with a timestamp less than the global minimum of $stable$. A straggler cannot have a timestamp less than the global minimum of $unacked$ and $unprocessed$. Hence the result follows.

■

In addition to being useful for fossil collection and output commit, the SGVT has another interesting application. DV is used by a process to determine whether it needs to roll back due to a rollback of another process. We make the observation that only those entries need to be kept in the DV whose associated states have virtual time greater than SGVT. Dependency on a state with virtual time less than SGVT need not be tracked because the corresponding state will never be rolled back. This results in the reduction of the overhead associated with the DV. In fact, DV starts with only one entry (process's own entry). As processes interact with one-another, size of DV starts increasing. However SGVT also keeps on increasing. So we expect the number of entries in DV to be sufficiently small.

4.5 Overheads

Our scheme incurs the following overheads for providing fault-tolerance:

- **Accessing stable storage**: We need to periodically save checkpoints on stable storage. This seems a necessary cost in absence of redundant resources like those used for replication. We save checkpoints asynchronously. So computation is not blocked when stable storage is being accessed.

- **Dependency information**: We tag a DV with each inter-cluster message. We expect the number of inter-cluster messages to be much smaller than the total number of messages. The size of DV is $O(n)$ entries, where n is the number of clusters in the system. But as explained in section 4.4, we expect the number of entries to be much smaller in practice.

- **Cluster Output queue**: Only inter-cluster messages are saved in the output queue. So we expect this overhead to be much smaller than that for Time Warp.

- **Clustered rollback**: Rollback of a single LP means rollback of the entire cluster. This slows down the simulation. But each cluster rolls back at most once in response to each straggler or failure. There is no possibility of avalanche of antimessages or echoing [14]. This compensates for the slowdown owing to the clustered rollback.

5 Failure Related Implementation Issues

So far we have discussed the modifications required in simulation schemes when failures can occur. Now we discuss some general issues that any distributed computation must address if it wants to survive failures:

- **Failure Detection:** In theory, it is impossible to distinguish a failed process from a very slow process [8]. In practice, many failure detectors have been built that work well for most practical situations [10]. Most of these detectors use a timeout mechanism.

- **Stable Storage:** Stable storage must be available across failures. In a multi-processor environment this is easy, as other processors can access the disk even if one of the processors fails. In a networking environment, the local disk may be inaccessible when the corresponding processor fails. So a network server must be used to make checkpoints stable.

- **Process Identity:** When a failed process is restarted, it may have a different port number or IP address. So location independent identifiers shall be used for the purpose of inter-process communication. These logical identifiers must be efficiently mapped to their physical values.
- **Environment Variables:** If a process is restarted on a different processor then some inconsistency may arise due to mismatch of the values of environment variables in pre- and post-failure computation. Logging and resetting of environment variables is required.

6 Conclusion

We have presented the motivation for and design of a fault-tolerant distributed simulation algorithm. By modeling a failure as a straggler, we are able to achieve low cost fault-tolerance.

References

[1] D. Agrawal and J. R. Agre. Replicated Objects in Time Warp Simulations. *Proc. Winter Simulation Conf.*, 657-664, 1992.

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

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

[4] K. P. Birman. *Building Secure and Reliable Network Applications*, Greenwich, CT: Manning Publications Co., 1996.

[5] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. on Computer Systems*, 3(1): 63-75, Feb. 1985.

[6] O. P. Damani, Y. M. Wang and V. K. Garg. Optimistic Distributed Simulation Based on Transitive Dependency Tracking. *Proc. 11th Workshop on Parallel and Distributed Simulation*, 90-97, 1997.

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

[8] M. J. Fischer, N. Lynch and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2), 374-382, April 1985.

[9] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* San Mateo, CA: Morgan Kaufmann Publishers, 117-119, 1993.

[10] Y. Huang and C. Kintala. Software Implemented Fault Tolerance: Technologies and Experience. *Proc. 23rd Intl. Symposium on Fault-Tolerant Computing,* 2-9, 1993.

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

[12] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), 558-565, 1978.

[13] Y. B. Lin, B. R. Preiss, W. M. Loucks, and E. D. Lazowska. Selecting the Checkpoint Interval in Time Warp Simulation. *Proc. 7th Workshop on Parallel and Distributed Simulation*, 3-10, 1993.

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

[15] D. M. Nicol and X. Liu. The Dark Side of Risk (What your mother never told you about Time Warp). *Proc. 11th Workshop on Parallel and Distributed Simulation*, 188-195, 1997.

[16] B. Randell. System Structure for Software Fault Tolerance. *IEEE Trans. on Software Eng.*, SE-1(2), 220-232, June 1975.

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

[18] P. Reiher, R. Fujimoto, S. Bellenot, and D. Jefferson. Cancellation Strategies in Optimistic Execution Systems. *Proc. SCS Multiconference on Distributed Simulation*, 112-121, 1990.

[19] G. Suri, Y. Huang, Y. M. Wang, W. K. Fuchs and C. Kintala. An Implementation and Performance Measurement of the Progressive Retry Technique. *Proc. IEEE International Computer Performance and Dependability Symposium*, 41-48, 1995.

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

[21] Y. M. Wang, O. P. Damani, and V. K. Garg. Distributed Recovery with K-Optimistic Logging. *Proc. 17th Intl. Conf. Distributed Computing Systems*, 60-67, 1997.