Proceedings of the 1991 Winter Simulation Conference
Barry L. Nelson, W. David Kelton, Gordon M. Clark (eds.)

FAULT-TOLERANT DISTRIBUTED SIMULATION

Bojan Grošelj

Department of Computer Science
University of Maryland
College Park, Maryland 20742

ABSTRACT

We consider two kinds of problems that may appear in a distributed discrete-event simulation: (1) the appearance of deadlocks due to unknown lookaheads and (2) processor crashes. Both of the problems are solved by taking global snapshots of the distributed simulation efficiently. We prove that the recorded global states of the distributed simulation are consistent and can thus be used by the deadlock breaking algorithm and for checkpointing.

1 INTRODUCTION

The model of the distributed simulation is Cocktail Party Simulation (COPS), where each processor accommodates a cluster of logical processes or simulated objects (Grošelj 1990, Grošelj and Tropper 1990). COPS is a conservative distributed simulation. We assume that the logical processes communicate with each other utilizing the edges of a known process graph. Therefore, if logical processes p_i and p_j are mapped to two different processors, Π_i and Π_j, then the two processors exchange event (i.e. simulation) messages with one another on behalf of processes p_i and p_j. There may or may not be a physical connection between the two processors. However, we assume that there is a communication channel between Π_i and Π_j and a communication channel between Π_i and Π_j. If the communication is only from p_i to p_j, then there exists only channel (Π_i, Π_j). We will use the notion of processor graph $G(V, E)$, where V is the set of processors and E is the set of directed communication channels among the processors.

Let there be N processors, numbered with distinct integers from 1 to N. Channels are assumed to be FIFO queues that can hold up to N messages. The messages are delivered in the order they were sent. Channels are assumed to be error-free. A message takes finite time to get to the head of the channel.

It will be assumed that the processor graph is strongly connected, i.e., there is a directed path from every processor to every other processor in G. If the original processor graph is not strongly connected, it is easy to make it strongly connected. For example, a ring of N dummy logical processes $(d_1, d_2, ..., d_N)$ can be added to the system, such that logical process d_i is mapped to processor Π_i, for $1 \leq i \leq N$. The importance of the last assumption will be demonstrated later in this work.

Several authors have considered the problem of deadlocks in distributed simulation (Chandy and Misra 1981, Grošelj and Tropper 1990). Deadlocks appear because of the cycles of logical processes for which the lookahead is unknown and has to be considered zero. By lookahead we mean the sum of the smallest service time and propagation delay associated with an event and a pair of logical processes. Most of the conservative algorithms for distributed simulation work only with a positive lookahead (Chandy and Misra 1979, Lubachevsky 1988). It has been demonstrated that the more accurate the lookahead, the faster the distributed simulation (Grošelj 1990). Especially good results can be obtained if service times and propagation delays are known beforehand (Nicol 1988). Unfortunately, in many simulations this is not the case. In this work we are concerned with a simulation that might have isolated cycles of logical processes with small, possibly zero lookahead. Therefore, the simulation might deadlock.

It has been shown (Grošelj and Tropper 1990), that there exists an efficient algorithm, the Time-of-Next-Event (TNE) algorithm, for breaking deadlocks that are associated with logical processes mapped onto the same processor. In the same work, an algorithm that breaks global deadlocks was presented as well. Both algorithms were implemented in COPS. The experiments show that the performance of the global deadlock-breaking algorithm deteriorates if there are many logical processes for which the lookahead is unknown.

We will show that it is possible to take global snapshots efficiently and run the TNE algorithm using the global state of the simulation as the input. The method also enables each processor to store local snapshots in stable storage (e.g., disk). Since a local snapshot is a part of a consistent global state, all processors can restart the computation after a crash from the global state using the stored local state. By performing the simulation in phases, it is possible to find the last consistent global state. We show that each processor has to maintain not more than $N+1$ local snapshots if the processor graph is strongly connected.

In Section 2 we present the global state recording algorithm. Section 3 deals with its correctness proof. The use of the global state for deadlock breaking and checkpointing is presented in Section 4. Discussion follows in Section 5.

2 GLOBAL STATE RECORDING IN COPS

The main problem in recording a global state of a distributed system is efficiency. Additional messages (i.e., markers) are normally used (Chandy and Lamport 1985). A termination algorithm is also needed in order to detect the end of the recording. By executing COPS in phases we can eliminate most of the overhead needed for global state recording.

The problem is reduced to defining the computation phases in an efficient manner. In other words, the message passing for synchronization of phases should be minimized. Fortunately, in COPS, the phases are defined naturally. During each phase, each processor is required to send precisely one message on every output channel. A processor starts a new phase after receiving messages on all of its input channels. The message is defined as one or several event or null messages lumped together.

The simulation program for each processor Π_i is as follows:

```
$ph := -1$;                    /* initial phase number */
while ($not\_finished$) do
  if ($ph \geq 0$)
    accept all in-msgs for phase $ph$;
  record local state $(s_i, ph)$;
  $ph := ph + 1 \bmod M$;
  simulate for period $P$ saving out-msgs in mboxes;
  for every output channel do
    deliver mbox $m_j$ to processor $\Pi_j$ as $(m_j, ph)$;
endwhile;
```

The line "simulate for period P..." includes scheduling all of the logical processes for simulation, running the local TNE algorithm and processing the incoming messages. Outgoing messages are concatenated in mailboxes corresponding to different processors. If there is no outgoing simulation message, a null message, containing the value computed by the local TNE algorithm, is inserted.

Note that a processor might receive input messages labeled with a higher phase number than the local variable ph. These input messages are stored in the input buffer and are not a part of the current local state.

3 CORRECTNESS PROOF

We prove that the global state recorded by the described procedure is consistent and that the simulation makes progress.

Let (S,k) represent the global state recorded in phase k and let (s_i,k) represent the local state of processor Π_i recorded in phase k.

First, let us define a consistent global state. During the computation, that starts in the initial state, each process finds itself in a different state. The transition from one state to another is caused by an atomic action, event e. The state of the channel is represented by the corresponding message queue. This state changes as a new message is appended to the channel (i.e., sent) or if the message at the head of the channel is deleted (i.e., received). A recorded global state (S,k) represents a history of events up to phase k. It divides all of the events in the entire computation in two sets, $PAST_k$ and $FUTURE_k$.

Following Chandy and Lamport (1985), and Bracha and Toueg (1987), we define partial relation $BEFORE$ on events: $e_i \; BEFORE \; e_j$ if

(1) both e_i and e_j occurred at the same processor, and e_i occurred before e_j, or

(2) e_i is the sending of a message and e_j is the receipt of this message.

Definition 1. A global state (S,k) is consistent iff $\langle \forall e_i, e_j : (e_i \in PAST_k) \wedge (e_j \in FUTURE_k) :: e_i \; BEFORE \; e_j \rangle$.

For simplicity first assume $M = \infty$.

Lemma 1. Let e_i be the sending of a message and e_j be the receipt of the same message. Then, $\langle \exists k : k \geq -1 :: ((S,k) \; BEFORE \; e_i \; BEFORE \; (S,k+1)) \wedge ((S,k) \; BEFORE \; e_j \; BEFORE \; (S,k+1)) \rangle$.

Proof. Each message is labeled at the sending processor with the number of the next snapshot. The message is recorded at the receiver in the local snapshot that has the same number as the message label. Hence if message label is $(k+1)$, then $((S,k)$ *BEFORE* e_i *BEFORE* $(S,k+1)) \wedge ((S,k)$ *BEFORE* e_j *BEFORE* $(S,k+1))$ holds. □

Lemma 2. $\langle \forall k : k \geq -1 :: \bigcup_{i=1}^{N}(s_i,k) = (S,k)$ is a consistent global state $\rangle$.

Proof. We use induction. For $k=-1$, $(S,-1)$ is the initial state and is by definition consistent. Let (S,k) be consistent. We have to prove, that $(S,k+1)$ is consistent as well. Since the time and phase numbers on the same processor are monotonically increasing, the *BEFORE* relation is maintained. We have to consider only those events that represent message sending and receiving. Let e_i represent message sending and let e_j represent receiving of the same message. From the consistency of (S,k), and Lemma 1, there exist only the following two cases:

(a) $(e_i \in PAST_k) \wedge (e_j \in PAST_k)$
(b) $(e_i \in FUTURE_k) \wedge (e_j \in FUTURE_k)$

Snapshot $(S,k+1)$ preserves the consistency in case (a). Case (b) may or may not be maintained. It remains to analyze the situation in which case (b) is not maintained. The following three cases do not maintain (b):

(b1) $(e_i \in PAST_{k+1}) \wedge (e_j \in PAST_{k+1})$
(b2) $(e_i \in PAST_{k+1}) \wedge (e_j \in FUTURE_{k+1})$
(b3) $(e_j \in PAST_{k+1}) \wedge (e_i \in FUTURE_{k+1})$

From Lemma 1, only case (b1) is possible, thus preserving the consistency of the global snapshot. □

From the two lemmas it follows that in the recorded global state (S,k) all of the messages are recorded in the local state of the receiver. The input channels are recorded as empty.

Next, we prove that at any moment, the maximum lag between two phase numbers is bounded. Let ph_i and ph_j be the current phase numbers at processor Π_i and Π_j, respectively. Then,

Lemma 3. $\langle \forall i,j : 1 \leq i,j \leq N :: |ph_j - ph_i| < N \rangle$ is an invariant.

Proof. Initially, the invariant holds since all phase numbers are -1. Consider two processors Π_i and Π_j connected by channel (Π_i, Π_j). Let $ph_i = k$. Then, the largest label of a message sent from Π_i to Π_j is k. This message can cause Π_j to increment ph_j to at most $k+1$. The invariant follows by induction and the fact that the processor graph is strongly connected. □

Next we bound the phase numbers.

Lemma 4. $M \geq 2N-2$.

Proof. Let the highest newly generated phase number be ph_h, and the current lowest phase number in the system be ph_l. M has to be large enough such that ph_i *BEFORE* ph_h for all current i's except for $i=h$. Let us first consider the case equivalent to the unbounded phase numbers, namely $ph_h - ph_l > 0$. By Lemma 3, $ph_h - ph_l < N$ and *BEFORE* relation can be easily established. If $ph_h - ph_l < 0$, then a wrap-around of phase numbers must have occurred. Assume that there exists the following rule:

$$ph_i - ph_j \geq N \Rightarrow ph_i \text{ } BEFORE \text{ } ph_j$$

By this rule,

$$ph_l - ph_h \geq N \Rightarrow ph_l \text{ } BEFORE \text{ } ph_h$$

and

$$\langle \forall ph_i : ph_l \leq ph_i \leq M :: ph_i \text{ } BEFORE \text{ } ph_h \rangle$$

From Lemma 3, there are at most N different phase numbers at a time. The smallest M that can accommodate all of the different phase numbers plus the gap between ph_h and ph_l is $2N-2$. □

In order to use common arithmetic operations and relation "<" on phase numbers, we adopt the following rule: if $ph_i - ph_j \geq N$ then change ph_i into $M+1+ph_i$.

Let $last_i$ be the phase number of the last snapshot at Π_i. Then,

Lemma 5. $\langle \forall i,j : 1 \leq i,j \leq N :: |last_j - last_i| \leq N \rangle$ is an invariant.

Proof. It follows from Lemma 3 and the invariant $(last_i = ph_i) \vee (last_i = ph_i - 1)$. □

Next, we define the *last* consistent global state $(S, last)$.

Definition 2. $(S, last) = (\bigcup_{i=1}^{N} s_i, \min_i(last_i))$

The last consistent global state is important for checkpointing. It is the state at which a computation has to be restarted after a processor crash. The computation at the processor with the largest $last_i$ has to be restarted from the snapshot numbered with the smallest $last_j$ in the system. It follows, from Lemma 5, that each processor has to store at least $N+1$ latest local snapshots.

Next, we prove that the simulation program makes progress.

Lemma 6. $\langle \forall i : 1 \leq i \leq N ::$
$(ph_i = k) \longmapsto (ph_i = k+1 \bmod M) \vee \neg not\_finished_i \rangle$.

The meaning of the lemma is that for every processor the phase number is incremented in finite time or the program terminates.

Proof. We assume that each line of the program is executed in finite time. The line "simulate for period P..." guarantees that every output channel receives a message since null-messages are generated for every output channel. Therefore, it remains to prove that if $ph_i = k$, Π_i eventually receives messages labeled with $ph = k + 1$ on all input channels. Initially, $ph = -1$ and is incremented to $ph = 0$ at every processor. All processors send output messages labeled with $ph = 0$. Hence all processors can accept these messages and can increment ph to 1. Lemma 6 follows by induction. □

4 DEADLOCK BREAKING AND CHECK-POINTING

This section describes the implementation of the deadlock breaking algorithm and the mechanism for crash recovery.

4.1 Deadlock Breaking

As already mentioned, there exists an algorithm (TNE) that breaks all simulation deadlocks if the global state of the system is known (Grošelj and Tropper 1990).

After recording the local state, each processor can send it to the host processor. The global TNE algorithm is executed at the host after it receives all of the local snapshots numbered with the same phase number. After the computation, the host sends to all processors the estimates on future event times for the incoming communication channels. These values are then used by the local TNE algorithm. Note that the global snapshot need not be taken during each phase. It can be taken every kth phase instead.

4.2 Crash Recovery

The last consistent state is found by following Lemma 2 and Definition 2. Crash recovery is initiated by the processor that crashed and recovered. It initiates the computation to find the last local snapshot with the smallest phase number. The computation consists of a broadcast by the recovered processor and collection of all $last_i$'s, followed by the broadcast of the smallest $last_i$. The computation can also be implemented as a two phase diffusing computation (Dijkstra and Scholten 1980). In the first phase, the smallest $last_i$ is found, and in the second phase it is propagated to every processor.

After knowing the smallest $last_i$, each processor can restart the simulation from the recorded local state, numbered with the smallest $last_i$. The computation resumes after the line "record local state...".

5 DISCUSSION

We have shown that the global state of the distributed simulation can be recorded efficiently. This enables efficient deadlock breaking and crash recovery.

We assumed that the processor graph is strongly connected. This is not a necessary condition for global state recording. It limits the number of saved states. It also limits the number of messages in the system and thus serves as a flow control mechanism.

Consider the following example. The processor graph in Figure 1 is not strongly connected. Since Π_1 does not get any input messages, it is not synchronized with Π_2 and Π_3. Therefore, it can increment ph_1 on its own, and flood Π_2 and Π_3 with messages. This phenomenon is enhanced if Π_2 and Π_3 are slowed down because of a deadlock.

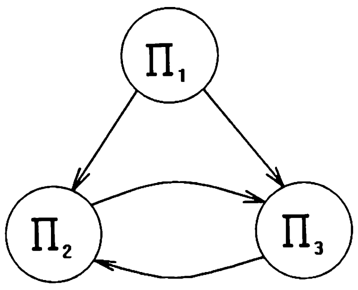

Figure 1: Π_1 Is Not Synchronized With Π_2 And Π_3

The deadlock breaking algorithm, using global snapshots, has been implemented in COPS. The performance tests on a 32-node Intel iPSC/2 hypercube show that the overhead is relatively small. In other words, in a distributed simulation with a good lookahead the *safe* simulation was never more than 10% slower than the *unsafe* simulation. The *safe* simulation was also tested on a worst-case example – a simulation of a hypercube with zero lookahead. The simulation did not deadlock in spite of zero lookahead and many cycles in the communication graph.

We were fortunate enough to implement the first version of COPS on a network of SUN workstations using the Cosmic Environment (Seitz and Su 1988). Since the system would not run without a flow control, we were forced to invent one of our own. That is how the idea of fault-tolerant COPS was formed.

ACKNOWLEDGMENTS

I am thankful to David Nicol and to Intel Corporation for allowing me to use their machines. The support from the National Science Foundation (Grant CCR-8909098) is greatly acknowledged.

REFERENCES

Bracha, G. and S. Toueg. 1987. Distributed deadlock detection. *Distributed Computing* **2**: 127–138.

Chandy, K.M. and L. Lamport. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems* **3**: 63–75.

Chandy, K.M. and J. Misra. 1979. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering* **5**: 440–452.

Chandy, K.M. and J. Misra. 1981. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM* **24**: 198–206.

Dijkstra, E.W. and C.S. Scholten. 1980. Termination detection for diffusing computations. *Information Processing Letters* **11**: 1–11.

Grošelj B. 1990. Cocktail Party Simulation and its performance on a network of workstations. In *Proceedings of the 1990 SCS Western Multiconference*, ed. D. Nicol, 70–73. SCS Simulation Series **22**, San Diego, California.

Grošelj, B. and C. Tropper. 1990. The distributed simulation of clustered processes. *Distributed Computing* **4**: 111–121.

Lubachevsky, B. 1988. Bounded lag distributed discrete event simulation. In *Proceedings of the 1988 Distributed Simulation Conference*, eds. B. Unger and D. Jefferson, 183–191. SCS Simulation Series **19**, San Diego, California.

Nicol, D.M. 1988. Parallel discrete-event simulation of FCFS stochastic queuing networks. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming, Environments, Applications, and Languages* **23**, 9, 124–137. Yale University, New Haven, Connecticut.

Seitz, C.L., J. Seizovic, and W.K. Su. 1988. The C programmer's abbreviated guide to multicomputer programming. Technical Report CS-TR-88-1, Caltech, Pasadena, California.

AUTHOR BIOGRAPHY

BOJAN GROŠELJ is a Visiting Assistant Professor in the Department of Computer Science at the University of Maryland, College Park. His research interests are distributed computing, algorithms, and program proving. He is Guest Editor for the International Journal of Computer Simulation.