Proceedings of the 1992 Winter Simulation Conference
ed. J. J. Swain, D. Goldsman, R. C. Crain, and J. R. Wilson

REPLICATED OBJECTS IN TIME WARP SIMULATIONS

Divyakant Agrawal

Department of Computer Science
University of California
Santa Barbara, CA 93106, U.S.A.

Jonathan R. Agre

Science Center
Rockwell International Corporation
Thousand Oaks, CA 91360, U.S.A.

ABSTRACT

The simulation objects in a Time Warp distributed, discrete event simulation system can be replicated in order to gain improvements in both performance and fault tolerance. Performance can be increased by having copies of frequently used objects at multiple sites, reducing the message traffic and delays. This represents a tradeoff since there is overhead involved in maintaining the consistency of the various copies. In general, there are advantages if the ratio of state modifying operations versus state observing operations is small. However, the replicated object can also be used as the basis of a new technique for increasing performance of fault tolerant systems called Optimistic Fault Tolerance (OFT). OFT uses Time Warp as part of the consistency maintenance scheme allowing the simulation to mask various forms of processor failures and continue operations in the case of a subset of processor failures.

1 INTRODUCTION

The notion of *Virtual Time* for synchronizing distributed applications was first introduced by Jefferson and Sowizral (1985) for controlling event sequencing in distributed, discrete event simulation (Misra 1986, Fujimoto 1990). Unlike real time, Virtual Time is not a monotonically increasing, single-valued function, but rather it allows different portions of a computation to advance and retract their own relative times, dynamically preserving the precedence constraints. The *Time Warp Mechanism* (TWM) (Jefferson 1985, Berry 1986) is an implementation of virtual time in a distributed environment, in which the asynchronous components are synchronized using partial *rollback* instead of the conventional method of blocking (Chandy and Misra 1979). One of the drawbacks of the TWM is that it assumes a completely reliable and failure-free environment.

Object replication is used in distributed systems to increase availability and fault-tolerance. By storing multiple copies of objects at several sites in the system, there is an increased likelihood of objects remaining available and accessible to users despite site and communication failures. However, complex and expensive synchronization mechanisms (Gifford 1979, Thomas 1979, Davcev and Burkard 1985, El Abbadi, Skeen and Cristian 1985, Paris and Long 1988, Jajodia and Mutchler 1990) are needed to maintain the consistency and integrity of objects. In this paper, we describe how object replication can be used to improve performance and fault-tolerance of simulations based on the Time Warp Mechanism.

System performance can be enhanced using strategically located replicas and allowing objects to access replicas locally. If an object is being sent messages from several other objects on differing nodes and if the frequency of reads is much greater than writes, then it may be beneficial to replicate the object on multiple nodes. It is felt that an optimistic processing mode, such as Time Warp, has an inherent advantage over other consistency control schemes because the replicas do not need to be explicitly synchronized in order to achieve consistency. Rather, the replicas can execute independently and use the rollback capability of Time Warp to maintain consistency in an efficient manner. A scheme for performance improvement was investigated by Goldberg (1992) who used virtual time to synchronize transactions in a database. However, the scheme was not appropriate for simulation, since it allowed the receiver (rather than the source) to set the event time. We have modified that scheme for distributed simulation

and extended it for fault tolerant applications.

2 THE TIME WARP MECHANISM

In this section we describe the Time Warp mechanism (Jefferson 1985) and some of its properties. The TWM controls a set of N objects distributed over a computer network. The objects communicate with each other by transmitting time-stamped messages over separate logical communication channels. We assume that the logical communication channel between two objects is completely reliable and preserves the order of messages. Each process executes optimistically without regard to synchronization conflicts with other objects. A synchronization conflict in the TWM arises when an out-of-order message is received, resulting in the rollback of some TWM objects. Each object, O_i, in the TWM maintains the following data-structures required for implementing rollback:

1. An input message queue, InQ_i.
2. An output message queue, $OutQ_i$.
3. A local state queue, $StateQ_i$.
4. A logical clock that maintains the local virtual time, LVT_i.

A message in the TWM conforms to the following format:

m = msgType(src, SVT, dest, RVT)

where msgType is one of $\{+, -\}$, src is the source object of m, SVT is the source virtual time at which m was sent, dest is the destination object of m, and RVT is the receive virtual time at which m must be processed at the destination. A data field that specifies the contents of the message is also part of the message, but is not relevant to the discussion in this paper. Messages received as input at an object O_i with the type field "+" are stored in InQ_i in the increasing order of RVT of the input messages. Copies of the messages that are sent by O_i are stored in $OutQ_i$ with the type field "−" and are stored in increasing order of the SVT of the output messages. The virtual time at O_i, denoted as LVT_i, is used as an index in the InQ_i, $OutQ_i$, and $StateQ_i$ to determine the current state of the process. The index in InQ_i also indicates the extent to which the input messages have been processed at O_i (since processed messages are not immediately removed).

A TWM object O_i executes by processing the input messages in InQ_i according to the scheduling rule of smallest RVT next. When a message m is processed, LVT_i is set to the receive virtual time of the message, $RVT(m)$, and we say that an event has occurred at O_i at time $LVT_i = RVT(m)$. This event may alter the local state of O_i and a copy of the object state is saved in $StateQ_i$. In addition, the event may cause one or more messages to be sent to other objects in the system. These positive messages have the RVT set to the intended virtual time to be processed at the destination and the SVT set to the LVT_i. After sending a positive message m', O_i places a negative counterpart of m', called the antimessage $\overline{m'}$ in $OutQ_i$. When all of the messages in the InQ_i have been processed, LVT_i is advanced to $+\infty$.

The execution of TWM objects is expected to happen as described above most of the time. However, occasionally an object O_i may receive an input message m_{oo} such that $RVT(m_{oo})$ is earlier than the current value of LVT_i. The message m_{oo} is termed an *out-of-order* message and causes O_i to roll back. O_i is rolled back to the event time prior to $RVT(m_{oo})$ and all events since that time must be undone. This is accomplished by discarding the states of O_i from $StateQ_i$ since $RVT(m_{oo})$ and removing and transmitting all antimessages in $OutQ_i$ with SVT greater than or equal to $RVT(m_{oo})$ to their intended recipients. O_i restores its internal state from $StateQ_i$ to the state with the largest time earlier than $RVT(m_{oo})$. The object then continues its forward execution by processing input messages in InQ_i starting from m_{oo}.

When an antimessage $\overline{m}$ is received at O_i, InQ_i is searched for the corresponding m. If m is in the unprocessed part of InQ_i, i.e., $RVT(m)$ is later than the current LVT_i, then m and $\overline{m}$ annihilate each other. The effect is as if m was never sent. On the other hand, if m is already processed at O_i, O_i must be rolled back to undo the effect of m. This is identical to the processing of an out-of-order message, except for the distinction that the positive message is withdrawn from the processed part of InQ_i and then annihilated.

As a Time Warp computation executes a distributed computation, the notion of a system time is defined and used to control storage, detect termination and handle other critical events such as I/O.

Definition 1 The *global virtual time* at real time t, $GVT(t)$, is defined as the minimum of the LVT_i of all objects O_i and the RVT of any unprocessed messages (including those in transit) (Fujimoto 1990)

At any real time t, it can be shown that no object will roll back to a virtual time earlier than $GVT(t)$. Similarly, it can be shown that GVT is a nondecreasing function of real time and progress in a computation is thus defined as an increase in GVT. Under certain reasonable assumptions a Time Warp computation is guaranteed to make progress (Jefferson 1985). In a distributed environment, only a lower bound of GVT can be computed.

3 TWM WITH REPLICATED OBJECTS

3.1 Overview

We now consider a model of the time warp mechanism where objects are replicated. In particular, the system consists of N objects and each object may be replicated at several sites. Furthermore, we assume that event messages can be classified into two categories:

1. *Read* messages or state-observing events.
2. *Write* messages or state-modifying events.

The state of an application object at simulation time t is defined to be the values of its instance variables at that time. Automatic modification to the local virtual time that occurs when a message is executed is not considered as a modification to the state since the application has no access to the previous time. In addition, the output message queue is not considered part of the state. Hence, it is possible for a Read message to generate output to any object in the system.

When objects are replicated in the TWM, inconsistencies could arise due to two reasons. The first type of inconsistency could arise due to the out-of-order messages that result from the optimistic or look-ahead execution mode of the TWM. The second type of inconsistency arises since replicas of an object may be inconsistent with each other during processing of write messages. For example, a read message at different replicas of the same object could return different results if a write has been received and processed at one, but not the other. In the following, we describe the extensions to the TWM event execution model to eliminate such inconsistencies.

As in the original TWM, the J^{th} replica, O_i^J of object O_i maintains the three data-structures: InQ_i^J, $OutQ_i^J$, and $StateQ_i^J$. In addition, LVT_i^J is the local virtual time at replica O_i^J. In order to deal with replication, we extend the message format as follows:

m = msgType(src, SVT, destList, RVT, primaries)

where the msgType is now { $+R$,$+W$,$-R$,$-W$ } to express the new categories of read and write. The field destList is now a list of replicas where the message is destined and the field primaries are members in destList who are responsible for generating the output for m (if any). All other members in destList will process m but will not generate any output.

Based on the above model, we now describe how a simulation proceeds in the TWM with replicated objects. A *read* message to an object O_i is sent to the closest (perhaps local) replica O_i^J of object O_i. That is, the field destList contains the address of replica O_i^J and the field primary is also set to O_i^J. Replica O_i^J processes the *read* message in the same way as objects process messages in the original TWM.

A *write* message to an object O_i is sent to all the replicas of O_i. That is, the field destList contains the addresses of all replicas $O_i^1, \ldots, O_i^K$ of object O_i. In addition, primary in the write message is set to the address of the closest (perhaps local) replica of O_i, say O_i^J. The *write* message is also processed at each replica in the same manner as objects process messages in the original TWM. The main distinction is that any output resulting from the processing of the *write* message is discarded at replicas that are not mentioned as primary. Thus, the output messages are sent only from the primary replica. As in the original TWM, a simulation starts through a special message from the application to one of the objects in the system. This message can be either a state-modifying message or a read message and it has the same format as described above.

Consider an object O_i with several replicas. Note that the contents of input and output queues at the replicas of O_i may not be identical. In particular, the input queues of replicas will contain all the write messages destined to O_i but will not contain all the read messages sent to O_i. Thus, the set of write messages in the input queue will be the same at each replica and the union of read messages to all replicas will contain all read messages sent to O_i. Furthermore, the relative order of all the write messages will be identical at each replica indicating that the state

queues of each replica will be identical. Similarly, the union of the output queues of all replicas will contain the messages that are sent from O_i to other objects. Thus, the rollback mechanism of the TWM can be used to correct the processing of an out-of-order message as well as to withdraw incorrect replies of read messages.

In particular, the reply of a read message from a replica may be incorrect if it has not received all prior write messages. However, later when the write message arrives at this replica, the replica will rollback and incorporate the effects of the write message and will then send the correct reply corresponding to the read message. We note that the computation of GVT will function correctly with this replication scheme and will not advance GVT to a new time until all replicas have processed the event at that time.

3.2 Implementation Issues

It is desirable to implement the replicated object management in a Time Warp kernel system so that many of the details are hidden from the application. In general, the application should transmit normal event messages without regard to the underlying replica numbers and locations. Message delivery to the replicated objects and the consistency maintenance should be handled by the Time Warp kernel. An implementation scheme for replicated objects is described for the Time Warp Object Oriented Distributed Simulation (TWOODS) kernel system (Tinker and Agre 1989, Agre and Tinker 1991). In this implementation, the only interface between the application and the specification of the replica numbers and distribution occurs at object creation time.

Unlike most Time Warp systems, TWOODS supports dynamic object creation so that the application can schedule an object to be created at any time during the simulation. The kernel maintains an *objectCreator* object on each node which is responsible for object creation. For the non-replicated case, an application object requests the creation of another object by sending a create event-message of the form defined earlier to the *objectCreator* at a specified node, where the data field specifies the type of object to be created. The create message immediately returns an *objId* that is a globally unique reference of the form: *[src, seq, dest]*, where *src* is the source node, *seq* is the number of objects previously requested to be created by the source node and *dest* is the node on which the object is to be created. The application objects are referred to by their *objId's* and in application event messages the source and destination fields are given as *objId's*.

The *objectCreator* maintains several standard tables including an *Object Location Table* and a *Communications Port Table*. In order to deliver an event message, the destination node is decoded from the *objId*. For objects that reside on a given node, the Object Location Table supplies the translation between the *objId* and the local address of that object. If an object resides on a foreign node, the destination node is used to select a port from the Communications Port Table on which to transmit the event message.

In order to support replicated objects, the user specifies the number of replicas of each object and their locations as part of a modified create message of the following form:

$objId$ = create(src, SVT, destList, RVT, primaries)

which immediately returns an *objId* tag for the new object using a primary node as the *dest* field of the *objId*. The *objectCreator* must also maintain a *Replica Table* consisting of entries of the form: *[objId, { destList }]*. We note that the *[src,seq]* pair of the *objId* is sufficient to uniquely identify the replicated object set. For the purposes of this paper, the distribution of replicas is assumed to be static throughout an execution of the simulation (although it will operate in the case of objects that are created and deleted during the course of the run).

The distribution of the Replica Table is broadcast-based and assumes that all nodes will maintain an identical copy of the complete replica table. When a create message is given to the kernel by the application, the kernel broadcasts the message to all of the *objectCreators* on all nodes. This supplies the data necessary for each node to construct an entry in its Replica Table. When a create message is processed by an *objectCreator* at a node in the *destList*, the object is created and the appropriate entry in its object table will be made. Hence it is possible to determine locally if an object is a replica and where the original replica and all of its copies reside.

The application generates event messages to an object in the conventional manner using the *objId* as the single destination and does not specify replicas. We assume that the kernel, using predetermined rules, will select the most efficient replica to serve as the primary object. In general, the choice is a function

of the communication delay and the processor loading. When the application generates an event message, the kernel will use the destination *objId* field to check the replica table and identify a replica set. The kernel will then make a determination of which replica will handle the request. For a read message, it will forward the message to that object, also designating it as the primary. When a write message is generated, the kernel identifies the primary object and then forwards the message to all of the replicas. All recipients of the message will execute it, but only the primary will generate output messages. Antimessages are designated as read or write according to their positive counterparts and are handled in the same fashion as event messages.

A variation of this scheme for systems with very large numbers of objects and replicas is possible in which each replica table contains only entries for objects that are replicated on that node. However, this is accomplished at the expense of an additional message hop delay for forwarding of replica copies.

A second variation removes the restriction that the source object determine whether the message is a state modifying message (i.e., read or a write). In this case, the application sends the message to the replica given by the *objId*. When the event message is executed, the object determines if the state has been modified by the message through a priori knowledge or direct comparison of the new and old states. If so, the message is tagged as a write and the kernel is notified to deliver copies of the message to all the other replicas, with the original object as the primary. An antimessage for a write message must also be replicated.

In the second variation it is possible for a message that is rolled back to change from a read message to a write message and vice versa. Hence, the rollback mechanism must be modified to remember whether the previous execution resulted in a change. If a message is reexecuted and remains the same, then no action is required of the kernel. If a message that was previously a read is reexecuted and becomes a write, then the kernel must replicate the message. If a write message is reexecuted and becomes a read, then it is necessary to retract the replicas. We note that each replica will have executed the entire set of state modifying messages earlier than a given GVT, yielding correct results. The overhead involves the possibly substantial delay in waiting for the message to be executed in the simulation, and then an additional message delay to send the replica copies for the write messages. This delay would also have an impact on the replicas since they would receive their messages at a later time, possibly causing more rollback.

3.3 Discussion

A surveillance application consisting of a large number of sensor objects monitoring moving targets (Agre and VoPava 1992) was enhanced using replicated objects. There are a small number of server objects, called environment sector managers, which maintain lists of targets in their area. Sensors periodically query the environment managers, which return information on the states of their target objects. The state of the environment is modified relatively infrequently by the introduction or departure of targets. These environment managers became bottlenecks due to the large number of sensor queries that needed to be processed. The performance penalty in the simulation was large since the environment manager tended to sequentialize the sensor requests, reducing the potential parallelism. Since the TWOODS kernel does not yet support replicated objects, we modified the application and replicated the environment manager at several nodes resulting in a significant decrease in execution time for cases with large numbers of sensors.

In general, the time warp mechanism with replicated objects described above improves performance of systems in which the communication delay between different sites is significant and the ratio of read accesses over write accesses is high. In such an environment, it would be particularly beneficial to identify those objects that are frequently accessed for reads and replicate them at several sites. Thus, read accesses to such objects can be handled by sending read messages to their local replicas. A write access, on the other hand, will result in a larger overhead since the write message must be sent to all replicas of replicated objects. However, this overhead can be reduced significantly if the underlying system provides a multicast communication facility.

Finally, we consider the issue of GVT computation. For the purpose of GVT computation, each replica is treated as an individual object and GVT then can be computed as in the original TWM. In the above mechanism a read message is sent to only one replica of an object, and, if this message has not been processed at

that replica, GVT cannot be allowed to be advanced beyond the RVT of this message. This mode of GVT computation impacts the fault-tolerance in the system as will be discussed in the next section.

4 OPTIMISTIC FAULT TOLERANCE

4.1 Overview

Another motivation for considering replicated objects is for increased fault tolerance. This may be more important than improving performance in certain applications. A new technique for improving fault tolerance, called Optimistic Fault Tolerance (OFT), that employs the Time Warp Mechanism for synchronizing redundant objects is described. The OFT assumes that objects are fault free most of the time and can proceed optimistically. Only when faults are detected is it necessary to perform recovery. The capabilities of Time Warp to rollback and retract events are then used to efficiently correct many forms of intermittent and permanent faults.

Clearly, requiring that all replicas participate in the computation of GVT as described in the previous section, results in reducing the overall fault-tolerance of the GVT computation. Consider for example a system with N objects and each object is available with probability p. Thus at the time when GVT is computed, the probability that the computation will be successful is p^N. However, considering a triple modular redundant (TMR) system where each object is replicated at three sites, this probability becomes p^{3N}, making it less likely to compute GVT at any time. Note, however, the overall ability of the simulation to terminate in the presence of faults is potentially very high. In the original TWM, if a process fails, the simulation fails. On the other hand, in the TWM with TMR, all three copies of an object have to fail simultaneously for the simulation to fail. Otherwise, the state of the failed replica can be constructed from the redundant copies using a recovery protocol similar to that described in Agrawal and Agre (1992).

Reduced fault-tolerance of the GVT computation discussed in section 3 arises because of the following two reasons:

1. A read message r is sent to only one replica and unless r is processed, GVT cannot go beyond the RVT of r. Hence, the LVT of each replica must be consulted for GVT computation.
2. Similarly, a write message w generates output only at the replica designated as primary by w. Since GVT cannot be advanced beyond the virtual time of output messages in transit, once again, the status of each replica is needed for GVT computation.

Thus, even though the earlier approach improves the overall performance and fault-tolerance of the system, it has some disadvantages. In particular, we would like to instill enough redundancy in the states of the replicas so that not all replicas are needed to compute GVT at any time. In particular, if read messages are sent to multiple replicas of an object and write messages generate output at multiple replicas, then only a subset of replicas are needed to compute GVT. In the following, we generalize this approach to tolerate up to $K-1$ failures of replicas when there are K replicas in a group.

In the proposed optimistic fault-tolerant scheme, we use the following model of the system. We assume that processes are *fail-stop*, i.e., if a process fails it will not generate further messages and will cease to participate in further computation. However, this implies that it is possible for a process to fail leaving erroneous messages in the system and replicas in inconsistent states. We will assume that it is possible to detect such failures. As before each object may be represented by multiple replicas of that object. Furthermore, all messages are assumed to be writes so that every replica receives a copy of each message. In addition, output is generated at all replicas by assigning each as a primary. Since multiple messages may represent a singleton entry in the queues, the structure of input and output queues is modified so that the entry in a queue is a message suite instead of singleton messages. The unique identifier of a message suite is $\langle SoureObjectId, SVT \rangle$.

As a result of message processing, several replicas will send the reply. All messages from the same replica group with identical SVT are enqueued in the same message suite of the input queue. The key element of this scheme is that forward processing at an object proceeds optimistically as soon as any one of the messages in a message suite is received at the object. However, if the majority of messages in a message suite changes, then the object will rollback, retracting the previously executed member of the suite and execute the majority. Otherwise, rollback and annihilation execute as in normal TWM.

The processing of messages is performed as follows. When a message m arrives at replica O_i^J it is stored in an appropriate message suite. During its normal forward processing, O_i^J processes the next message suite by setting its LVT to $RVT(MsgSuite)$. $RVT(MsgSuite)$ is set to the dominant RVT value in the message suite, i.e., most messages in $MsgSuite$ agree upon RVT. If the receipt of m causes $RVT(MsgSuite)$ containing m to change and LVT is larger than $RVT(MsgSuite)$, O_i^J is rolled back. After rollback, O_i^J continues its forward processing.

The GVT computation proceeds by consulting each object for its *reliable LVT*. The reliable LVT is defined to be the largest time for which a majority of replica messages in the suite and every earlier suite have been processed. By this we mean that if a suite receives from a 3-replica set, then a majority of two messages is sufficient. The reliable LVT is always less than or equal to the object's current LVT, guaranteeing a lower bound on the actual GVT. If the GVT computation discovers that a node has failed, then every other node will update its replica set to reflect the loss of replicas (necessary to compute the sufficient majority). The new GVT and new replica configuration is sent to all operational replicas. The replicas use the new configuration to discard messages from failed replicas with RVT larger than the new value of GVT. It can be easily shown that as long as one replica has survived, the GVT computation as well as the TWM simulation can continue. We note that the communication traffic has been greatly increased in this approach in order to achieve this level of fault tolerance. For example, if there are K replicas of each object then we have increased the message traffic by a factor of K^2.

4.2 Discussion

In a TMR OFT system, the availability of GVT computation increases significantly from a non-replicated case. Let the system be composed of N objects on N nodes each with availability p. In this case, the availability or fault-tolerance of the GVT computation as explained before is p^N. On the other hand, when each object is represented by three replicas on $3N$ different nodes, at least one from each replica set is sufficient to compute GVT. The probability that at least one replica is available at any time is: $1-(1-p)^3$. Hence, the availability of GVT computation increases from p^N to $(1-(1-p)^3)^N$. For a numerical value of $N = 10$ and $p = 0.90$, the availability of a GVT computation is 0.35 in the non-replicated TWM whereas it is 0.99 in the OFT.

A more equitable comparison using an equal number of nodes is shown by considering the following example of a simulation consisting of three objects A,B, and C. Suppose we have 3 replicas of each object and 9 nodes, so that each replica is on a separate node. A traditional approach would be to execute the simulation as 3 independent, non-communicating runs, i.e., $[A_1, B_1, C_1]$, $[A_2, B_2, C_2]$, and $[A_3, B_3, C_3]$. The simulation is successful if one or more runs completes. One or more failures in each run would result in an unsuccessful simulation. In contrast, the TMR OFT scheme would communicate among the replicas in order to maintain consistency and would "switch-out" the failed nodes. Availability is increased, however, the cost is a 3-fold increase in message traffic. Assuming that each node is available with probability p, the availability of the three independent runs is: $1-(1-p^3)^3$. The availability of the TMR OFT is: $(1-(1-p)^3)^3$. For $p = .9$, the independent runs has an availability of .98 while the TMR OFT has an availability of .997.

5 CONCLUSIONS

Optimistic processing of replicated objects in a distributed, discrete-event simulation system has been implemented by augmenting the standard features of Time Warp such as time stamped messages and rollback. Replicated objects can be used to improve the performance and/or the fault tolerance of the simulation, depending on the needs of the application. Further research is needed to quantify the benefits of replicated objects and to determine the parameters of simulation applications that determine whether an application can be enhanced by object replication.

There are several extensions to the optimistic fault tolerance (OFT) technique that are being pursued. The fault model described in this paper is fail-stop, however, the scheme will mask many other types of faults. Extensions to other classes of faults such as intermittent or Byzantine faults is feasible. In addition, a combination of replication and a recovery scheme such as in Agrawal and Agre (1992) could resurrect failed objects on other nodes. Similarly, nodes could fail and then later recover or be replaced by spares, thus further increasing the fault tolerance of the sys-

tem. Lastly, although this scheme has been presented for simulations, many other applications that use active objects (i.e., those that are both clients and servers) could benefit from this technique.

ACKNOWLEDGEMENTS

Dr. Agrawal was supported by the NSF under grant numbers IRI-9004998 and IRI-9117094.

REFERENCES

Agrawal, D., and J. R. Agre. 1992. Recovering from Process Failures in the Time Warp Mechanism. *IEEE Transactions on Computers.* To Appear.

Agre, J. R., and P. A. Tinker. 1991. Useful extensions to a time warp simulation system. In *Proceedings of the SCS Multiconference on Parallel and Distributed Simulation*, pages 78–85, Anaheim, CA.

Agre, J. R., and S. M. VoPava. 1992. Distributed end-to-end simulation of a strategic attack/defense engagement model (sadem). In *Proceedings of the Summer Computer Simulation Conference*, pages 914–919, Reno, NV.

Berry, O. 1986. *Performance Evaluation of the Time Warp Distributed Simulation Mechanism.* PhD thesis, University of Southern California.

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

Davcev, D., and W. Burkhard. 1985. Consistency and Recovery Control for Replicated Files. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 87–96.

El Abbadi, A., D. Skeen, and F. Cristian. 1985. An efficient fault-tolerant protocol for replicated data management. In *Proceedings of the 4th ACM Symposium on Principles of Database Systems*, pages 215–228.

Fujimoto, R. M. 1990. Parallel discrete event simulation. *Communications of the ACM*, 33(7):30–53.

Gifford, D. K. 1979. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 150–159.

Goldberg, A. P. 1992. Virtual time synchronization of replicated processes. In *Proceedings of the SCS Multiconference on Parallel and Distributed Simulation*, pages 107–116, Newport Beach, CA.

Jajodia, S., and D. Mutchler. 1990. Dynamic Voting Algorithms for Maintaining the Consistency of a Replicated Database. *ACM Transactions on Database Systems*, 15(2):230–280.

Jefferson, D. R. 1985. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425.

Jefferson, D. R., and H. A. Sowizral. 1985. Fast concurrent simulation using the time warp mechanism. *SCS Simulation*, 15(2):63–69.

Misra, J. 1986. Distributed-discrete event simulation. *ACM Computing Surveys*, 18(1):39–65.

Paris, J. F., and D. E. Long. 1988. Efficient Dynamic Voting Algorithms. In *Proceedings of the Fourth IEEE International Conference on Data Engineering*, pages 268–275.

Thomas, R. H. 1979. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transaction on Database Systems*, 4(2):180–209.

Tinker, P. A., and J. R. Agre. 1989. Object handling, messaging, and state manipulation in a time warp system. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 79–84, Tampa, FL.

AUTHOR BIOGRAPHIES

DIVYAKANT AGRAWAL is currently an Assistant Professor in the Department of Computer Science at the University of California at Santa Barbara. He received his B.E. degree from Birla Institute of Technology and Science, Pilani, India in 1980. He received his M.S. and Ph.D. degrees in computer science from SUNY at Stony Brook in 1984 and 1987. His research interests include design of algorithms for concurrent and fault-tolerant systems.

JONATHAN R. AGRE is currently working at the Rockwell International Science Center in the Software Systems Department on simulation technologies. He received his B.S. in Mathematics and Computer Science in 1975, the M. S. and the Ph. D. in Computer Science in 1977 and 1981, all from the University of Maryland. His research interests include distributed and parallel simulation and fault tolerant systems.