Proceedings of the 2003 Winter Simulation Conference
S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, eds.

DISTRIBUTED SIMULATION SYSTEMS

Richard M. Fujimoto

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332, U.S.A.

ABSTRACT

An overview of technologies concerned with distributing the execution of simulation programs across multiple processors is presented. Here, particular emphasis is placed on discrete event simulations. The High Level Architecture (HLA) developed by the Department of Defense in the United States is first described to provide a concrete example of a contemporary approach to distributed simulation. The remainder of this paper is focused on time management, a central issue concerning the synchronization of computations on different processors. Time management algorithms broadly fall into two categories, termed conservative and optimistic synchronization. A survey of both conservative and optimistic algorithms is presented focusing on fundamental principles and mechanisms. Finally, time management in the HLA is discussed as a means to illustrate how this standard supports both approaches to synchronization.

1 INTRODUCTION

Here, the term *distributed simulation* refers to distributing the execution of a single "run" of a simulation program across multiple processors. This encompasses several different dimensions. One dimension concerns the motivation for distributing the execution. One paradigm, often referred to as *parallel simulation*, concerns the execution of the simulation on a tightly coupled computer system, e.g., a supercomputer or a shared memory multiprocessor. Here, the principal reason for distributing the execution is to reduce the length of time to execute the simulation. In principal, by distributing the execution of a computation across N processors, one can complete the computation up to N times faster than if it were executed on a single processor. Another reason for distributing the execution in this fashion is to enable larger simulations to be executed than could be executed on a single computer. When confined to a single computer system, there may not be enough memory to perform the simulation. Distributing the execution across multiple machines allows the memory of many computer systems to be utilized.

A second, increasingly important motivation for distributed simulation concerns the desire to integrate several different simulators into a single simulation environment. One example where this paradigm is frequently used is in military training. Tank simulators, flight simulators, computer generated forces, and a variety of other models may be used to create a distributed virtual environment into which personnel are embedded to train for hypothetical scenarios and situations. Another emerging area of increasing importance is infrastructure simulations where simulators of different subsystems in a modern society are combined to explore dependencies among subsystems. For example, simulations of transportation systems may be combined with simulations of electrical power distribution systems, computer and communication infrastructures, and economic models to assess the economic impact of natural or human-caused disasters. In both these domains (military and infrastructure simulations) it is far more economical to link existing simulators to create distributed simulation environments than to create new models within the context of a single tool or piece of software. The High Level Architecture (HLA) developed by the U.S. Department of Defense defines an approach to integrate, or *federate*, separate, autonomous simulators into a single, distributed simulation system.

Another dimension that differentiates distributed simulation paradigms is the geographical extent over which the simulation executes. Often distributed simulations are executed over broad geographic areas. This is particularly useful when personnel and/or resources (e.g., databases or specialized facilities) are included in the distributed simulation exercise. Distributed execution eliminates the need for these personnel and resources to be physically co-located, representing an enormous cost savings. Distributed simulations operating over the Internet have create an enormous market for the electronic gaming industry. At the opposite extreme, high performance simulations may execute on multiprocessor computers confined to a single cabinet or machine room. Close proximity is necessary to reduce the delay for inter-processor communications that might otherwise severely degrade performance. These

high performance simulations often require too much communication between processors, making geographically distributed execution too inefficient. Historically, the term distributed simulation has often been used to refer to geographically distributed simulations, while parallel simulation traditionally referred to simulations executed on a tightly coupled parallel computer, however, with new computing paradigms such as clusters of workstations and grid computing, this distinction has become less clear, so we use the single term distributed simulation here to refer to all categories of distributed execution.

Two widely-used architectures for distributed simulation are the *client-server* and the *peer-to-peer* approaches. As its name implies, the client-server approach involves executing the distributed simulation on one or more server computers (which may be several computers connected by a local area network) to which clients (e.g., users) can "log in" from remote sites. The bulk of the simulation computation is executed on the server machines. This approach is typically used in distributed simulations used for multiplayer gaming. Centralized management of the simulation computation greatly simplifies management of the distributed simulation system, and facilitates monitoring of the system, e.g., to detect cheating. On the other hand, peer-to-peer systems have no such servers, and the simulation is distributed across many machines, perhaps interconnected by a wide area network. The peer-to-peer approach is often used in distributed simulations used for defense.

The remainder of this paper is organized as follows. First, an historical view of distributed simulation technology and how it has evolved over the last twenty to thirty years is briefly presented. The High Level Architecture is presented to introduce aspects of a contemporary approach to distributed simulation. The remainder of this paper focuses on the synchronization problem, and time management algorithms that have been developed to address this issue. A more detailed, comprehensive treatment of these topics is presented in (Fujimoto 2000). Certain sections of this tutorial borrow material presented in (Fujimoto 2001).

2 HISTORICAL PERSPECTIVE

Distributed simulation technology has developed largely independently in at least three separate communities: the high performance computing community, the defense community, and the Internet/gaming industry. Each of these is briefly discussed next.

Distributed simulation in the high performance computing community originated in the late 1970's and early 1980's, focusing on synchronization algorithms (now referred to as time management). Synchronization algorithm were designed, for the most part, in order for the distributed execution to produce exactly the same results as a sequential execution of the simulation program, except (hopefully) more quickly. Initial algorithms utilized what is know referred to as a conservative paradigm, meaning blocking mechanisms were used to ensure no synchronization errors (out of order event processing) occurred. Initial algorithms date back to the late 1970's with seminal work by (Chandy and Misra 1978), and (Bryant 1977), among others, who are credited with first formulating the synchronization problem and developing the first solutions. These algorithms are among a class of algorithms that are today referred to as conservative synchronization techniques. In the early 1980's seminal work by Jefferson and Sowizral developed the Time Warp algorithm (Jefferson 1985). Time Warp is important because it defined fundamental constructs widely used in a class of algorithms termed optimistic synchronization. Conservative and optimistic synchronization techniques form the core of a large body of work concerning parallel discrete event simulation techniques, and much of the subsequent work in the field is based on this initial research.

The defense community's work in distributed simulation systems date back to the SIMNET (SIMulator NETworking) project. While the high performance computing community was largely concerned with reducing execution time, the defense community was concerned with integrating separate training simulations in order to facilitate interoperability and software reuse. The SIMNET project (1983 to 1990) demonstrated the viability of using distributed simulations to create virtual worlds for training military personnel for engagements (Miller and Thorpe 1995). This lead to the development of a set of standards for interconnecting simulators known as the Distributed Interactive Simulation (DIS) standards (IEEE Std 1278.1-1995 1995). The 1990's also saw the development of the Aggregate Level Simulation Protocol (ALSP) that applied the SIMNET concept of interoperability and model reuse to wargame simulations (Wilson and Weatherly 1994). ALSP and DIS have since been replaced by the High Level Architecture whose scope spans the broad range of defense simulations, including simulations for training, analysis, and test and evaluation of equipment and components.

A third track of research and development efforts arose from the Internet and computer gaming industry. Some of the work in this area can be traced back to a role-playing game called dungeons and dragons and textual fantasy computer games such as Adventure developed in the 1970's. These soon gave way to Multi-User Dungeon (MUD) games in the 1980's. Important additions such as sophisticated computer graphics helped created the video game industry that is flourishing today. Distributed, multi-user gaming is sometimes characterized as the "killer application" where distributed simulation technology may have the greatest economic and social impact.

3 THE HIGH LEVEL ARCHITECTURE

The High Level Architecture (HLA) was developed in the mid 1990's. It is intended to promote reuse and interoperation of simulations. The HLA effort was based on the premise that no one simulation could satisfy all uses and applications for the Defense community. The intent of the HLA is to provide a structure that supports reuse of different simulations, ultimately reducing the cost and time required to create a synthetic environment for a new purpose. An introduction to the HLA is presented in (Kuhl, Weatherly et al. 1999).

Though developed in the context of defense application, the HLA was intended to have applicability across a broad range of simulation application areas, including education and training, analysis, engineering and even entertainment, at a variety of levels of resolution. These widely differing application areas indicate the variety of requirements that were considered in the development and evolution of the HLA.

The HLA does not prescribe a specific implementation, nor does it mandate the use of any particular set of software or programming language. It was envisioned that as new technological advances become available, new and different implementations would be possible within the framework of the HLA.

An HLA federation consists of a collection of interacting simulations, termed federates. A federate may be a computer simulation, a manned simulator, a supporting utility (such as a viewer or data collector), or an interface to a live player or instrumented facility. All object representation stays within the federates. The HLA imposes no constraints on what is represented in the federates or how it is represented, but it does require that all federates incorporate specified capabilities to allow the objects in the simulation to interact with objects in other simulations through the exchange of data.

Data exchange and a variety of other services are realized by software called the Runtime Infrastructure (RTI). The RTI is, in effect, a distributed operating system for the federation. The RTI provides a general set of services that support the simulations in carrying out these federate-to-federate interactions and federation management support functions. All interactions among the federates go through the RTI.

The RTI software itself and the algorithms and protocols that is uses are not defined by the HLA standard. Rather, it is the *interface* to the RTI services that are standardized. The HLA runtime interface specification provides a standard way for federates to interact with the RTI, to invoke the RTI services to support runtime interactions among federates and to respond to requests from the RTI. This interface is implementation independent and is independent of the specific object models and data exchange requirements of any federation.

The HLA is formally defined by three components: the HLA rules, the Object Model Template (OMT), and. the interface specification. Each of these is briefly described next.

3.1 HLA Rules

The HLA rules summarize the key principles behind the HLA (IEEE Std 1516-2000 2000). The rules are divided into two groups: federation and federate rules. Federations are required to define a Federation Object Model (FOM) specified in the Object Model Template (OMT) format. The FOM characterizes the information (objects) that are visible by more than one federate. During the execution of the federation, all object representation must reside within the federates (not the RTI). Only one federate may update the attribute(s) of any instance of an object at any given time. This federate is termed the owner of the attribute, and ownership may transfer from one federate to another during the execution of the federation via the ownership management services defined in the Interface Specification. All information exchanges among the federates takes place via the RTI using the services defined in the HLA interface specification.

Additional rules apply to individual federates. Under the HLA, each federate must document their public information in a Simulation Object Model (SOM) using the OMT. Based on the information included in their SOM, federates must import and export information, transfer object attribute ownership, updates attributes and utilize the time management services of the RTI when managing local time.

3.2 Object Models

HLA object models are descriptions of the essential sharable elements of the federation in 'object' terms. The HLA is directed towards interoperability; hence in the HLA, object models are intended to focus on description of the critical aspects of simulations and federations, which are shared across a federation. The HLA puts no constraints on the content of the object models. The HLA does require that each federate and federation document its object model using a standard object model template (IEEE Std 1516.2-2000 2000). These templates are intended to be the means for open information sharing across the community to facilitate reuse of simulations.

As mentioned earlier, the HLA specifies two types of object models: the HLA Federation Object Model (FOM) and the HLA Simulation Object Model (SOM). The HLA FOM describes the set of objects, attributes and interactions, which are shared across a federation. The HLA SOM describes the simulation (federate) in terms of the types of objects, attributes and interactions it can offer to future federations. The SOM is distinct from internal design information; rather it provides information on the capabilities of a simulation to exchange information as part of a federa-

tion. The SOM is essentially a contract by the simulation defining the types of information it can make available in future federations. The availability of the SOM facilitates the assessment of the appropriateness of the federate for participation in a federation.

While the HLA does not define the contents of a SOM or FOM, it does require that a common documentation approach be used. Both the HLA FOM and SOM are documented using a standard form called the HLA Object Model Template (OMT).

3.3 The Interface Specification

The HLA interface specification describes the runtime services provided to the federates by the RTI, and by the federates to the RTI (IEEE Std 1516.3-2000 2000). There are six classes of services. *Federation management* services offer basic functions required to create and operate a federation. *Declaration management* services support efficient management of data exchange through the information provided by federates defining the data they will provide and will require during a federation execution. *Object management* services provide creation, deletion, identification and other services at the object level. *Ownership management* services supports the dynamic transfer of ownership of object/attributes during an execution. *Time management* services support synchronization of runtime simulation data exchange. Finally*, data distribution management* services support the efficient routing of data among federates during the course of a federation execution. The HLA interface specification defines the way these services are accessed, both functionally and in an application programmer's interface (API).

4 TIME MANAGEMENT

Time management is concerned with ensuring that the execution of the distributed simulation is properly synchronized. This is particularly important for simulations used for analysis (as opposed training where errors that are not perceptible to humans participating in the exercise may be acceptable). Time management not only ensures that events are processed in a correct order, but also helps to ensure that repeated executions of a simulation with the same inputs produce exactly the same results. Currently, time management techniques such as those described here are typically not used in training simulations, where incorrect event orderings and non-repeatable simulation executions can usually be tolerated.

Time management algorithms usually assume the simulation consists of a collection of *logical processes* (LPs) that communicate by exchanging timestamped messages or events. In the context of the HLA, each federate can be viewed as a single LP. The goal of the synchronization mechanism is to ensure that each LP processes events in timestamp order. This requirement is referred to as the *local causality constraint*. Ignoring events containing exactly the same time stamp, it can be shown that if each LP adheres to the local causality constraint, execution of the simulation program on a parallel computer will produce exactly the same results as an execution on a sequential computer where all events are processed in time stamp order. This property also helps to ensure that the execution of the simulation is repeatable; one need only ensure the computation associated with each event is repeatable.

Synchronization is particularly interesting for the case of discrete event simulations. In this case, each LP can be viewed as a sequential discrete event simulator. This means each LP maintains local state information corresponding to the entities it is simulating and a list of time stamped events that have been scheduled for this LP, but have not yet been processed. This *pending event list* includes local events that the LP has scheduled for itself as well as events that have been scheduled for this LP by other LPs. The main processing loop of the LP repeatedly removes the smallest time stamped event from the pending event list and processes it. Thus, the computation performed by an LP can be viewed as a sequence of event computations. Processing an event means zero or more state variables within the LP may be modified, and the LP may schedule additional events for itself or other LPs. Each LP maintains a simulation time clock that indicates the time stamp of the most recent event processed by the LP. Any event scheduled by an LP must have a time stamp at least as large as the LP's simulation time clock when the event was scheduled.

The time management algorithm must ensure that each LP processes events in time stamp order. This is non-trivial because each LP does not a priori know what events will later be received from other LPs. For example, suppose the next unprocessed event stored in the pending event list has time stamp 10. Can the LP process this event? How does the LP know it will not later receive an event from another LP with time stamp less than 10? This question captures the essence of the synchronization problem.

Much research has been completed to attack this problem. Time management algorithms can be classified as being either *conservative* or *optimistic*. Briefly, conservative algorithms take precautions to avoid the possibility of processing events out of time stamp order, i.e., the execution mechanism avoids synchronization errors. In the aforementioned example where the next unprocessed event has a time stamp of 10, the LP must first ensure it will not later receive any additional events with time stamp less than 10 before it can process this event. On the other hand, optimistic algorithms use a detection and recovery approach. Events are allowed to be processed out of time stamp order, however, a separate mechanism is provided to recover from such errors. Each of these are described next.

4.1 Conservative Time Management

The first synchronization algorithms were based on conservative approaches. The principal task of any conservative protocol is to determine when it is "safe" to process an event. An event is said to be safe when can one guarantee no event containing a smaller time stamp will be later received by this LP. Conservative approaches do not allow an LP to process an event until it has been guaranteed to be safe.

At the heart of most conservative synchronization algorithms is the computation for each LP of a Lower Bound on the Time Stamp (LBTS) of future messages that may later be received by that LP. This allows the mechanism to determine which events are safe to process. For example, if the synchronization algorithm has determined that the LBTS value for an LP is 12, then all events with time stamp less than 12 are safe, and may be processed. Conversely, all events with time stamp larger than 12 cannot be safely processed. Whether or not events with time stamp equal to 12 can be safely processed depends on specifics of the algorithm, and the rules concerning the order that events with the same time stamp (called simultaneous events) are processed. Processing of simultaneous events is a complex subject matter that is beyond the scope of the current discussion, but is discussed in detail in (Jha and Bagrodia 2000). The discussion here assumes that each event has a unique time stamp. It is straightforward to introduce tie breaking fields in the time stamp to ensure uniqueness (Mehl 1992).

4.1.1 Null Messages and Deadlock Avoidance

The algorithms described in (Bryant 1977; Chandy and Misra 1978) were among the first synchronization algorithms that were developed. They assume the topology indicating which LPs send messages to which others is fixed and known prior to execution. It is assumed each LP sends messages with non-decreasing time stamps, and the communication network ensures that messages are received in the same order that they were sent. This guarantees that messages on each incoming link of an LP arrive in timestamp order. This implies that the timestamp of the last message received on a link is a lower bound on the timestamp of any subsequent message that will later be received on that link. Thus, the LBTS value for an LP is simply the minimum among the LBTS values of its incoming links.

Messages arriving on each incoming link are stored in first-in-first-out order, which is also timestamp order because of the above restrictions. Local events scheduled within the LP can be handled by having a queue within each LP that holds messages sent by an LP to itself. Each link has a clock that is equal to the timestamp of the message at the front of that link's queue if the queue contains a message, or the timestamp of the last received message if the queue is empty. The process repeatedly selects the link with the smallest clock and, if there is a message in that link's queue, processes it. If the selected queue is empty, the process blocks. The LP never blocks on the queue containing messages it schedules for itself, however. This protocol guarantees that each process will only process events in non-decreasing timestamp order.

Although this approach ensures the local causality constraint is never violated, it is prone to deadlock. A cycle of empty links with small link clock values (e.g., smaller than any unprocessed message in the simulator) can occur, resulting in each process waiting for the next process in the cycle. If there are relatively few unprocessed messages compared to the number of links in the network, or if the unprocessed events become clustered in one portion of the network, deadlock may occur very frequently.

Null messages are used to avoid deadlock. A null message with timestamp T_{null} sent from LP_A to LP_B is a promise by LP_A that it will not later send a message to LP_B carrying a timestamp smaller than T_{null}. Null messages do not correspond to any activity in the simulated system; they are defined purely for avoiding deadlock situations. Processes send null messages on each outgoing link after processing each event. A null message provides the receiver with additional information that may be used to determine that other events are safe to process.

Null messages are processed by each LP just like ordinary non-null messages, except no activity is simulated by the processing of a null message. In particular, processing a null message advances the simulation clock of the LP to the time stamp of the null message. However, no state variables are modified and no non-null messages are sent as the result of processing a null message.

How does a process determine the timestamps of the null messages it sends? The clock value of each incoming link provides a lower bound on the timestamp of the next event that will be removed from that link's buffer. When coupled with knowledge of the simulation performed by the process, this bound can be used to determine a lower bound on the timestamp of the next *outgoing* message on each output link. For example, if a queue server has a minimum service time of *T*, then the timestamp of any future departure event must be at least *T* units of simulated time larger than any arrival event that will be received in the future.

Whenever a process finishes processing a null or non-null message, it sends a new null message on each outgoing link. The receiver of the null message can then compute new bounds on its outgoing links, send this information on to its neighbors, and so on. It can be shown that this algorithm avoids deadlock (Chandy and Misra 1978).

The null message algorithm introduced a key property utilized by virtually all conservative synchronization algorithms: lookahead. If an LP is at simulation time T, and it

can guarantee that any message it will send in the future will have a time stamp of at least T+L regardless of what messages it may later receive, the LP is said to have a lookahead of L. As we just saw, lookahead is used to generate the time stamps of null messages. One constraint of the null message algorithm is it requires that no cycle among LPs exist containing zero lookahead, i.e., it is impossible for a sequence of messages to traverse the cycle, with each message scheduling a new message with the same time stamp.

4.1.2 Exploiting Next Event Timestamp Information

The main drawback with the null message algorithm is it may generate an excessive number of null messages. Consider a simulation containing two LPs. Suppose both are blocked, each has reached simulation time 100, and each has a lookahead equal to 1. Suppose the next unprocessed event in the simulation has a time stamp of 200. The null message algorithm will result in null messages exchanged between the LPs with time stamp 101, 102, 103, and so on. This will continue until the LPs advance to simulation time 200, when the event with time stamp 200 can now be processed. A hundred null messages must be sent and processed between the two LPs before the non-null message can be processed. This is clearly very inefficient. The problem becomes even more severe if there are many LPs.

The principal problem is the algorithm uses only the current simulation time of each LP and lookahead to predict the minimum time stamp of messages it could generate in the future. To solve this problem, we observe that the key piece of information that is required is the time stamp of the next unprocessed event within each LP. If the LPs could collectively recognize that this event has time stamp 200, all of the LPs could immediately advance from simulation time 100 to time 200. Thus, the time of the next event across the entire simulation provides critical information that avoids the "time creeping" problem in the null message algorithm. This idea is exploited in more advanced synchronization algorithms.

Another problem with the null message algorithm concerns the case where each LP can send messages to many other LPs. In the worst case, the LP topology is fully connected meaning each LP could send a message to any other. In this case, each LP must broadcast a null message to every other LP after processing each event. This also results in an excessive number of null messages.

One early approach to solving these problems is an alternate algorithm that allows the computation to deadlock, but then detects and breaks it (Chandy and Misra 1981). The deadlock can be broken by observing that the message(s) containing the smallest timestamp is (are) always safe to process. Alternatively, one may use a distributed computation to compute lower bound information (not unlike the distributed computation using null messages described above) to enlarge the set of safe messages.

Many other approaches have been developed. Some protocols use a synchronous execution where the computation cycles between (i) determining which events are "safe'" to process, and (ii) processing those events. It is clear that the key step is determining the events that are safe to process each cycle. Each LP must determine a lower bound on the time stamp (LBTS) of messages it might later receive from other LPs. This can be determined from a snapshot of the distributed computation as the minimum among:

- the simulation time of the next event within the LP if it is blocked, or the current time of the LP if it is not blocked, plus the LP's lookahead and
- the time stamp of any transient messages, i.e., any message that has been sent but has not yet been received at its destination.

A barrier synchronization can be used to obtain the snapshot. Transient messages can be "flushed" out of the system in order to account for their time stamps. If first-in-first-out communication channels are used, null messages can be sent through the channels to flush the channels, though as noted earlier, this may result in many null messages. Alternatively, each LP can maintain a counter of the number of messages it has sent, and the number if has received. When the sum of the send and receive counters across all of the LPs are the same, and each LP has reached the barrier point, it is guaranteed that there are no more transient messages in the system. In practice, summing the counters can be combined with the computation for computing the global minimum value (Mattern 1993).

To determine which events are safe, the distance between LPs is sometimes used (Ayani 1989; Lubachevsky 1989; Cai and Turner 1990). This "distance" is the minimum amount of simulation time that must elapse for an event in one LP to directly or indirectly affect another LP, and can be used by an LP to determine bounds on the timestamp of future events it might receive from other LPs. This assumes it is known which LPs send messages to which other LPs. Other techniques focus on maximizing exploitation of lookahead, e.g., see (Meyer and Bagrodia 1999; Xiao, Unger et al. 1999). Full elaboration of these and other technique is beyond the scope of the present discussion.

Another thread of research in synchronization algorithms concerns relaxing ordering constraints in order to improve performance. Some approaches amount to simply ignoring out of order event processing (Sokol and Stucky 1990; Rao, Thondugulam et al. 1998). Use of time intervals, rather than precise time stamps, to encode uncertainty of temporal information in order to improve the performance of time management algorithms have also been proposed (Fujimoto 1999) (Beraldi and Nigro 2000). Use of causal order rather than time stamp order for distributed simulation applications has also been studied (Lee, Cai et al. 2001).

4.2 Optimistic Time Management

In contrast to conservative approaches that avoid violations of the local causality constraint, optimistic methods allow violations to occur, but are able to detect and recover from them. Optimistic approaches offer two important advantages over conservative techniques. First, they can exploit greater degrees of parallelism. If two events *might* affect each other, but the computations are such that they actually don't, optimistic mechanisms can process the events concurrently, while conservative methods must sequentialize execution. Second, conservative mechanism generally rely on application specific information (e.g., distance between objects) in order to determine which events are safe to process. While optimistic mechanisms can execute more efficiently if they exploit such information, they are less reliant on such information for correct execution. This allows the synchronization mechanism to be more transparent to the application program than conservative approaches, simplifying software development. On the other hand, optimistic methods may require more overhead computations than conservative approaches, leading to certain performance degradations.

The Time Warp mechanism (Jefferson 1985) is the most well known optimistic method. When an LP receives an event with timestamp smaller than one or more events it has already processed, it rolls back and reprocesses those events in timestamp order. Rolling back an event involves restoring the state of the LP to that which existed prior to processing the event (checkpoints are taken for this purpose), and "unsending" messages sent by the rolled back events. An elegant mechanism called anti-messages is provided to "unsend" messages.

An anti-message is a duplicate copy of a previously sent message. Whenever an anti-message and its matching (positive) message are both stored in the same queue, the two are deleted (annihilated). To "unsend" a message, a process need only send the corresponding anti-message. If the matching positive message has already been processed, the receiver process is rolled back, possibly producing additional anti-messages. Using this recursive procedure all effects of the erroneous message will eventually be erased.

Two problems remain to be solved before the above approach can be viewed as a viable synchronization mechanism. First, certain computations, e.g., I/O operations, cannot be rolled back. Second, the computation will continually consume more and more memory resources because a history (e.g., checkpoints) must be retained, even if no rollbacks occur; some mechanism is required to reclaim the memory used for this history information. Both problems are solved by global virtual time (GVT). GVT is a lower bound on the timestamp of any future rollback. GVT is computed by observing that rollbacks are caused by messages arriving "in the past." Therefore, the smallest timestamp among unprocessed and partially processed messages gives a value for GVT. Once GVT has been computed, I/O operations occurring at simulated times older than GVT can be committed, and storage older than GVT (except one state vector for each LP) can be reclaimed.

GVT computations are essentially the same as LBTS computations used in conservative algorithms. This is because rollbacks result from receiving a message or anti-message in the LP's past. Thus, GVT amounts to computing a lower bound on the time stamp of future messages (or anti-messages) that may later be received.

Several algorithms for computing GVT (LBTS) have been developed, e.g., see (Samadi 1985; Mattern 1993), among others. Asynchronous algorithms compute GVT "in background" while the simulation computation is proceeding, introducing the difficulty that different processes must report their local minimum at different points in time. A second problem is one must account for transient messages in the computation, i.e., messages that have been sent but not yet received. Mattern describes an elegant solution to these problems using consistent cuts of the computation and message counters, discussed earlier (Mattern 1993).

A pure Time Warp system can suffer from overly optimistic execution, i.e., some LPs may advance too far ahead of others leading to excessive memory utilization and long rollbacks. Many other optimistic algorithms have been proposed to address these problems. Most attempt to limit the amount of optimism. An early technique involves using a sliding window of simulated time (Sokol and Stucky 1990). The window is defined as [GVT, GVT+W] where W is a user defined parameter. Only events with time stamp within this interval are eligible for processing. Another approach delays message sends until it is guaranteed that the send will not be later rolled back, i.e., until GVT advances to the simulation time at which the event was scheduled. This eliminates the need for anti-messages and avoids cascaded rollbacks, i.e., a rollback resulting in the generation of additional rollbacks (Dickens and Reynolds 1990; Steinman 1992). An approach that also a local rollback mechanism to avoid anti-messages using a concept called lookback (somewhat analogous to lookahead in conservative synchronization protocols) is described in (Chen and Szymanski 2002; Chen and Szymanski 2003). A technique called direct cancellation is sometimes used to rapidly cancel incorrect messages, thereby helping to reduce overly optimistic execution (Fujimoto 1989; Zhang and Tropper 2001).

Another problem with optimistic synchronization concerns the amount of memory that may be required to store history information. Several techniques have been developed to address this problem. For example, one can roll back computations to reclaim memory resources (Jefferson 1990; Lin and Preiss 1991). State saving can be performed infrequently rather than after each event (Lin, Preiss et al. 1993; Palaniswamy and Wilsey 1993). The memory used

by some state vectors can be reclaimed even though their time stamp is larger than GVT (Preiss and Loucks 1995).

Early approaches to controlling Time Warp execution used user-defined parameters that had to be tuned to optimize performance. Later work has focused on adaptive approaches where the simulation executive automatically monitors the execution and adjusts control parameters to maximize performance. Examples of such adaptive control mechanisms are described in (Ferscha 1995; Das and Fujimoto 1997), among others.

Practical implementation of optimistic algorithms requires that one must be able to roll back all operations, or be able to postpone them until GVT advances past the simulation time of the operation. Care must be taken to ensure operations such as memory allocation and deallocation are handled properly, e.g., one must be able to roll back these operations. Also, one must be able to roll back execution errors. This can be problematic in certain situations, e.g., if an optimistic execution causes portions of the internal state of the Time Warp executive to be overwritten (Nicol and Liu 1997).

Another approach to optimistic execution involves the use of reverse computation techniques rather than rollback (Carothers, Perumalla et al. 1999). Undoing an event computation is accomplished by executing the inverse computation, e.g., to undo incrementing a state variable, the variable is instead decremented. The advantage of this technique is it avoids state saving, which may be both time consuming and require a large amount of memory. In (Carothers, Perumalla et al. 1999) a reverse compiler is described to automatically generate inverse computations.

Synchronization is a well-studied area of research in the distributed simulation field. There is no clear consensus concerning whether optimistic or conservative synchronization perform better; indeed, the optimal approach usually depends on the application. In general, if the application has good lookahead characteristics and programming the application to exploit this lookahead is not overly burdensome, conservative approaches are the method of choice. Indeed, much research has been devoted to improving the lookahead of simulation applications, e.g., see (Deelman, Bagrodia et al. 2001). Otherwise, optimistic synchronization offers greater promise. Disadvantages of optimistic synchronization include the potentially large amount of memory that may be required, and the complexity of optimistic simulation executives. Techniques to reduce memory utilization further aggravate the complexity issue.

4.3 Time Management in the HLA

The HLA provides a set of services to support time management. A principal consideration in defining these services was the observation that different federates may use different local time management mechanisms and have different requirements for message ordering and delay. Two major categories emerged. One class of simulations were designed to created virtual environments for training and test and evaluation (e.g., hardware-in-the-loop) applications. The execution of these simulations is paced by wallclock time, and synchronization algorithms to guarantee time stamp ordering of events are typically not used. Achieving low, predictable delays to transmit messages are important. A second class of simulations are those that require synchronization algorithms, in part to ensure proper ordering of events, and in part as a means to ensure that executions are repeatable, i.e., multiple executions of the same simulation with the same inputs yield exactly the same results. These simulations may use event stepped or time stepped execution mechanisms locally. It was envision that some federates may be executing on a parallel processor, and may be using conservative or optimistic synchronization mechanisms within their federate. The HLA time management services were designed to accommodate this wide variety of applications.

There are two principal elements of the HLA time management services: message ordering, and time advance mechanisms. The HLA supports two types of ordering: receive ordered communication, and time stamp order. With receive ordered communication, no guarantees are provided by the RTI concerning the order that messages are delivered to a federate; they are essentially delivered in the order that they are received. This minimizes the latency to transmit messages through the RTI, and is the ordering typically used for real-time training exercises and test and evaluation applications. With time stamp ordering, each message is assigned by the sender a time stamp, and messages are delivered to the federate in time stamp order. In some situations the RTI may need to buffer the message in order to guarantee that it won't later receive a message with a smaller time stamp before delivering it to the federate. Thus, the latency for transmitting messages may be larger when using time stamp ordering. Time stamp order is normally used for analysis applications which are often not paced by wallclock time where correct ordering of events and repeatable execution are important.

The HLA time advance mechanisms are realized by a set of services for advancing simulation (or logical) time. A protocol is defined where federates request a time advance, and the RTI issues a Time Advance Grant when the request can be honored. The RTI ensures that a federate is not advanced to simulation time T, until it can guarantee that no time stamp ordered messages with later arrive with time stamp less than T.

Both time stamp ordering and the time advance mechanisms rely on computation of a lower bound on the time stamp (LBTS) of messages that will later arrive for a federate. To compute LBTS values, federates provide the following information: its current simulation time, a single lookahead value for the federate (L), and guarantees concerning the generation of future events. Regarding the lat-

ter, when a federate invokes the Time Advance Request(T) service to request its simulation time be advanced to T, it makes an unconditional guarantee that no messages will later be sent with time stamp less than T+L. This service is typically used by time stepped federates. As noted earlier, use of only unconditional guarantees leads to the lookahead creep problem. To address this issue, federates also provide conditional guarantees. Specifically, when a federate invokes the service Next Event Request (T), it conditionally guarantees that no future messages will be sent with time stamp less than T, *provided the federate does not receive additional messages with time stamp less than T*. This service is typically used by event driven federates, where T is specified as the time of the next local event within the federate.

The HLA time management services define additional services to support optimistic execution. Optimistic execution requires that the federate must be able to process events even though messages with a smaller time stamp may later arrive. For this purpose, the Flush Queue service is defined that delivers all available time stamp ordered messages to the federate. In addition, some mechanism is required to implement anti-messages in Time Warp. This is accomplished through the Retract service. When a federate invokes Retract, it cancels a previously sent message. If the message has already been delivered, the retraction request is forwarded to the receiving federate, who must then cancel the original event. Finally, the Flush Queue service includes specification of time stamp information and advances a federates simulation time much like the Next Event Request service. This is used to advance Global Virtual Time for the federation. It should be noted that it is the federate's responsibility to implement its own rollback, e.g., using a state saving/restoration or a reverse execution mechanism.

While the HLA was originally developed to combine *different* simulators, other work has explored using HLA as an approach to parallelize sequential simulations. The central idea is to use HLA to federate a simulation with itself. Early work using this approach, though not in the context of HLA, is described in (Nicol and Heidelberger 1996) for queueing network simulations. Recent work using HLA to parallelize a commercial air traffic control simulation is described in (Bodoh and Wieland 2003). This concept has also been applied to parallelizing existing sequential simulators of communication networks (Bononi, D'Angelo et al. 2003; Perumalla, Park et al. 2003). Other work, also aimed at simulating communication networks, parallelizes sequential simulations using a fixed point computation paradigm (Szymanski, Liu et al. 2003). Self-federated HLA-based distributed simulations for supply chain analysis is described in (Turner, Cai et al. 2000).

5 CONCLUSIONS

Beginning with research and development efforts in the 1970's, research in distributed simulation systems has matured over the years. Much of the early research in this area was motivated purely by performance considerations. As processor speeds have continued to increase at an exponential pace, performance alone has become less of a motivating factor in recent years. For many problems such as simulation of large-scale networks such as the Internet, performance remains a principal motivating objective, however, much interest in this technology today stems from the promises of cost savings resulting from model reuse. Standards such as IEEE 1516 for the High Level Architecture demonstrate the widespread interest in use of distributed simulation technology for this purpose.

What is the future for the technology? It is interesting to speculate. One potential path is to focus on applications. High performance computing remains a niche market that targets a handful of important, computation intensive applications. For more broader impacts in society, one must look to the entertainment and gaming industry, where distributed simulation technology has seen the most widespread deployment, and impact in society. Another view is to observe that software is often driven by advances in hardware technology, and look to emerging computing platforms to define the direction the technology will turn. In this light, ubiquitous computing stands out as an emerging area where distributed simulation may be headed. For example, execution of distributed simulations on handheld computers necessitates examination of power consumption because battery life is a major constraint in such systems. Grid computing is still another emerging approach where distributed simulations may emerge and have an impact.

ACKNOWLEDGMENTS

The author gratefully acknowledges support for distributed simulation research from the National Science Foundation under grants EIA-0219976 and ECS-0225447.

REFERENCES

Ayani, R. (1989). A Parallel Simulation Scheme Based on the Distance Between Objects. *Proceedings of the SCS Multiconference on Distributed Simulation*, Society for Computer Simulation. **21:** 113-118.

Beraldi, R. and L. Nigro (2000). Exploiting Temporal Uncertainty in Time Warp Simulations. *Proceedings of the 4th Workshop on Distributed Simulation and Real-Time Applications***:** 39-46.

Bodoh, D. J. and F. Wieland (2003). Performance Experiments with the High Level Architecture and the Total Airport and Airspace Model (TAAM). *Proceedings of*

the 17th Workshop on Parallel and Distributed Simulation: 31-39.

Bononi, L., G. D'Angelo, et al. (2003). HLA-Based Adaptive Distributed Simulation of Wireless Mobile Systems. *Proceedings of the 17th Workshop on Parallel and Distributed Simulation*: 40-49.

Bryant, R. E. (1977). Simulation of packet communications architecture computer systems. *MIT-LCS-TR-188*.

Cai, W. and S. J. Turner (1990). An Algorithm for Distributed Discrete-Event Simulation -- the "Carrier Null Message" Approach. *Proceedings of the SCS Multiconference on Distributed Simulation*, SCS Simulation Series. **22:** 3-8.

Carothers, C. D., K. Perumalla, et al. (1999). "Efficient Optimistic Parallel Simulation Using Reverse Computation." *ACM Transactions on Modeling and Computer Simulation* **9**(3).

Chandy, K. M. and J. Misra (1978). "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs." *IEEE Transactions on Software Engineering* **SE-5**(5): 440-452.

Chandy, K. M. and J. Misra (1981). "Asynchronous Distributed Simulation via a Sequence of Parallel Computations." *Communications of the ACM* **24**(4): 198-205.

Chen, G. and B. K. Szymanski (2002). Lookback: A New Way of Exploiting Parallelism in Discrete Event Simulation. *Proceedings of the 16th Workshop on Parallel and Distributed Simulation*: 153-162.

Chen, G. and B. K. Szymanski (2003). Four Types of Lookback. *Proceedings of the 17th Workshop on Parallel and Distributed Simulation*: 3-10.

Das, S. R. and R. M. Fujimoto (1997). "Adaptive Memory Management and Optimism Control in Time Warp." *ACM Transactions on Modeling and Computer Simulation* **7**(2): 239-271.

Deelman, E., R. Bagrodia, et al. (2001). Improving Lookahead in Parallel Discrete Event Simulations of Large-Scale Applications using Compiler Analysis. *Proceedings of the 15th Workshop on Parallel and Distributed Simulation*: 5-13.

Dickens, P. M. and J. Reynolds, P. F. (1990). SRADS With Local Rollback. *Proceedings of the SCS Multiconference on Distributed Simulation*. **22:** 161-164.

Ferscha, A. (1995). Probabilistic Adaptive Direct Optimism Control iin Time Warp. *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*: 120-129.

Fujimoto, R. M. (1989). "Time Warp on a Shared Memory Multiprocessor." *Transactions of the Society for Computer Simulation* **6**(3): 211-239.

Fujimoto, R. M. (1999). Exploiting Temporal Uncertainty in Parallel and Distributed Simulations. *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*: 46-53.

Fujimoto, R. M. (2000). *Parallel and Distributed Simulation Systems*, Wiley Interscience.

Fujimoto, R. M. (2001). Parallel and Distributed Simulation. *Proceedings of the Winter Simulation Conference*.

IEEE Std 1278.1-1995 (1995). *IEEE Standard for Distributed Interactive Simulation -- Application Protocols*. New York, NY, Institute of Electrical and Electronics Engineers, Inc.

IEEE Std 1516-2000 (2000). *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) -- Framework and Rules*. New York, NY, Institute of Electrical and Electronics Engineers, Inc.

IEEE Std 1516.2-2000 (2000). *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) -- Object Model Template (OMT) Specification*. New York, NY, Institute of Electrical and Electronics Engineers, Inc.

IEEE Std 1516.3-2000 (2000). *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) -- Interface Specification*. New York, NY, Institute of Electrical and Electronics Engineers, Inc.

Jefferson, D. (1985). "Virtual Time." *ACM Transactions on Programming Languages and Systems* **7**(3): 404-425.

Jefferson, D. R. (1990). Virtual Time II: Storage Management in distributed Simulation. *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*: 75-89.

Jha, V. and R. Bagrodia (2000). "Simultaneous Events and Lookahead in Simulation Protocols." *ACM Transactions on Modeling and Computer Simulation* **10**(3): 241-267.

Kuhl, F., R. Weatherly, et al. (1999). *Creating Computer Simulation Systems: An Introduction to the High Level Architecture for Simulation*, Prentice Hall.

Lee, B.-S., W. Cai, et al. (2001). A Causality Based Time Management Mechanism for Federated Simulations. *Proceedings of the 15th Workshop on Parallel and Distributed Simulation*: 83-90.

Lin, Y.-B. and B. R. Preiss (1991). "Optimal Memory Management for Time Warp Parallel Simulation." *ACM Transactions on Modeling and Computer Simulation* **1**(4).

Lin, Y.-B., B. R. Preiss, et al. (1993). Selecting the Checkpoint Interval in Time Warp Simulations. *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*: 3-10.

Lubachevsky, B. D. (1989). "Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks." *Communications of the ACM* **32**(1): 111-123.

Mattern, F. (1993). "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation." *Journal of Parallel and Distributed Computing* **18**(4): 423-434.

Mehl, H. (1992). A Deterministic Tie-Breaking Scheme for Sequential and Distributed Simulation. *Proceedings of the Workshop on Parallel and Distributed Simulation*, Society for Computer Simulation. **24:** 199-200.

Meyer, R. A. and R. L. Bagrodia (1999). Path Lookahead: A Data Flow View of PDES Models. *Proceedings of the 13th Workshop on Parallel and Distributed Simulation***:** 12-19.

Miller, D. C. and J. A. Thorpe (1995). "SIMNET: The Advent of Simulator Networking." *Proceedings of the IEEE* **83**(8): 1114-1123.

Nicol, D. and P. Heidelberger (1996). "Parallel Execution for Serial Simulators." *ACM Transactions on Modeling and Computer Simulation* **6**(3): 210-242.

Nicol, D. M. and X. Liu (1997). The Dark Side of Risk. *Proceedings of the 11th Workshop on Parallel and Distributed Simulation***:** 188-195.

Palaniswamy, A. C. and P. A. Wilsey (1993). An Analytical Comparison of Periodic Checkpointing and Incremental State Saving. *Proceedings of the 7th Workshop on Parallel and Distributed Simulation***:** 127-134.

Perumalla, K. S., A. Park, et al. (2003). Scalable RTI-Based Parallel Simulation of Networks. *Proceedings of the 17th Workshop on Parallel and Distributed Simulation***:** 97-104.

Preiss, B. R. and W. M. Loucks (1995). Memory Management Techniques for Time Warp on a Distributed Memory Machine. *Proceedings of the 9th Workshop on Parallel and Distributed Simulation***:** 30-39.

Rao, D. M., N. V. Thondugulam, et al. (1998). Unsynchronized Parallel Discrete Event Simulation. *Proceedings of the Winter Simulation Conference***:** 1563-1570.

Samadi, B. (1985). Distributed Simulation, Algorithms and Performance Analysis. *Computer Science Department*. Los Angeles, California, University of California, Los Angeles.

Sokol, L. M. and B. K. Stucky (1990). MTW: Experimental Results for a Constrained Optimistic Scheduling Paradigm. *Proceedings of the SCS Multiconference on Distributed Simulation*. **22:** 169-173.

Steinman, J. S. (1992). "SPEEDES: A Multiple-Synchronization Environment for Parallel Discrete Event Simulation." *International Journal on Computer Simulation*: 251-286.

Szymanski, B. K., Y. Liu, et al. (2003). Parallel Network Simulation under Distributed Genesis. *Proceedings of the 17th Workshop on Parallel and Distributed Simulation***:** 61-68.

Turner, S. J., W. T. Cai, et al. (2000). Adapting a Supply-Chain Simulation for HLA. *Proceedings of the 4th IEEE Workshop on Distributed Simulation and Real-Time Applications***:** 71-78.

Wilson, A. L. and R. M. Weatherly (1994). The Aggregate Level Simulation Protocol: An Evolving System. *Proceedings of the 1994 Winter Simulation Conference***:** 781-787.

Xiao, Z., B. Unger, et al. (1999). Scheduling Critical Channels in Conservative Parallel Simulation. *Proceedings of the 13th Workshop on Parallel and Distributed Simulation***:** 20-28.

Zhang, J. L. and C. Tropper (2001). The Dependence List in Time Warp. *Proceedings of the 15th Workshop on Parallel and Distributed Simulation***:** 35-45.

AUTHOR BIOGRAPHY

RICHARD M. FUJIMOTO is a professor with the College of Computing at the Georgia Institute of Technology. He received the Ph.D. and M.S. degrees from the University of California (Berkeley) in 1980 and 1983 (Computer Science and Electrical Engineering) and B.S. degrees from the University of Illinois (Urbana) in 1977 and 1978 (Computer Science and Computer Engineering). He has been an active researcher in the parallel and distributed simulation community since 1985 and has published numerous papers and a book entitled on this subject. He served as the technical lead in defining the time management services for the DoD High Level Architecture (HLA). Fujimoto is Co-Editor-in-Chief for Simulation: Transactions of the Society for Modeling and Simulation International as well as an area editor for ACM Transactions on Modeling and Computer Simulation. He also served leadership roles in the organization of several conferences in the distributed simulation field..He can be contacted by e-mail at `<fujimoto@cc.gatech.edu>`