Parallel and Distributed Simulation Systems and the High Level Architecture

Kalyan S. Perumalla, Ph.D.
Oak Ridge National Laboratory
Oak Ridge, Tennessee
perumallaks@ornl.gov

ABSTRACT

This tutorial introduces the fundamental principles and algorithms underlying parallel/distributed simulation, along with an overview of synchronization methods and their implementation. The manifestation of these principles and methods in the Time Management services of the High Level Architecture (HLA) are described. Important systems issues such as computation and communication overheads are highlighted. This tutorial is designed to help the audience gain a detailed understanding of the concepts, terminology and application of parallel/distributed simulation methods, especially in the context of the HLA.

ABOUT THE AUTHOR

Kalyan S. Perumalla is a senior researcher in the modeling and simulation group at the Oak Ridge National Laboratory. Prior to his current position, he served as a research faculty member in the College of Computing, Georgia Tech, and a member of the modeling and simulation center (MSREC) at Georgia Tech. He has over 8 years of research and development experience in the area of parallel and distributed simulation systems, including high-performance runtime infrastructures and large-scale simulation, and has published widely on these topics. He co-developed the Federated Simulations Development Kit (FDK), a widely-disseminated high-performance runtime infrastructure for HLA-like distributed simulator federations. He has also built several additional research prototype systems and tools (e.g., for distributed debugging, network modeling, interoperable simulations and parallel optimization), most of which are in use by researchers worldwide. He received his Ph.D. in Computer Science from Georgia Tech in 1999. Dr. Perumalla has served as co-principal investigator on multiple federally-funded projects on scalable parallel/distributed discrete event simulation systems. He delivered a similar tutorial at last year's I/ITSEC (Dec 2004).

Parallel and Distributed Simulation Systems and the High Level Architecture

Kalyan S. Perumalla, Ph.D.
Oak Ridge National Laboratory
Oak Ridge, Tennessee
perumallaks@ornl.gov

INTRODUCTION

Parallel and Distributed Simulation (PADS) systems are, at their core, concerned with time-synchronized simulations running on multiple inter-connected processors. The US Department of Defense (DoD) High Level Architecture (HLA) includes support for time-synchronized parallel/distributed simulations, built on fundamental PADS concepts.

There are four main goals of this tutorial: (1) understand the fundamental PADS concepts relevant to the HLA (2) understand the usage of basic Time Management (TM) services in the HLA (3) appreciate algorithmic and performance aspects behind some of the HLA TM implementation, and (4) gain exposure to typical usage of relatively advanced TM services.

The rest of this document is organized as follows. A brief overview of the HLA is provided for completeness. This is followed by an introduction to some of the fundamental concepts in PADS that underlie the synchronization service frameworks in HLA. The most commonly used HLA TM services are then described, along with templates of their intended/typical usage. This is followed by a brief introduction to some implementation approaches inside RTI software for runtime support of TM services. Exposure to RTI implementations is intended to provide an idea of the complexity and performance implications of RTI TM services to the simulation developers. Finally, some of the more advanced features of TM services are described, such as retractions, simultaneity and optimistic simulation.

HLA OVERVIEW

The US DoD HLA is based on a "system of systems" approach to integrating separate, individual simulation systems. It is based under the premise that no single simulation could satisfy all user needs. Instead, interoperability of multiple systems is envisioned as a way to integrate as well as reuse different DoD simulations.

Although primarily designed for defense simulations, the architecture is general enough for use in other domains as well.

Architecture

In the HLA, an integrated execution of simulations is called a federation. Individual simulators participating in a federation are called federates. Federates can be of different types: pure software simulators such as computer generated forces, human-in-the-loop simulators such as virtual simulators, or live components such as instrumented weapon systems.

The HLA consists of *Rules* that federates must adhere to for proper interaction during execution. It also defines an *Object Model Template* (OMT), a format for specifying the set of common objects used by the federation. A third component called the *IFSpec* provides interface to a runtime infrastructure (*RTI*) that ties the federates together during execution.

Interface

The IFSpec is further divided into multiple categories of services, such as federation management, declaration management, object management, ownership management, time management and data distribution management.

The primary focus of this tutorial is in time management (TM) services.

Typical Usage

In a typical usage template of the *IFSpec* services, each federate invokes the services roughly in the following sequence:

(1) Initialize federation – create & join federation execution (Federation Management)
(2) Declare objects – publish & subscribe to object classes (Declaration Management)
(3) Exchange information at runtime – update/reflect attribute values, send/receive interactions (Object Management); synchronization (Time Management); affect object ownership (Ownership Management); change interest regions (Data Distribution Management)

(4) Terminate – resign & destroy federation execution (Federation Management).

The rest of this tutorial focuses on time management services.

FUNDAMENTAL PADS CONCEPTS

A significant amount of literature exists in the parallel and distributed simulation (PADS) research community, which has previously explored issues in time synchronized simulations[1-5]. The HLA TM has, in large part, been built on insights from PADS research. Thus, the fundamental concepts in HLA TM are common with those in PADS.

Notions of Time

In simulation, there are generally three distinct notions of time. The first is the *physical time*, which is the time in the physical system that is being modeled (e.g., 10-11pm on January 1990). The second is the *simulation time*, which is a representation of the physical time for the purposes of simulation (e.g., number of seconds since 10pm of January 1990, represented in floating point values in the range [0..3600] corresponding to the simulated time period of the physical time). Finally, the *wallclock time* is the elapsed real time during execution of the simulation, as measured by a hardware clock (e.g., number of milliseconds of computer time during execution). For each, the notions of time axis and time instant can be defined – *time axis* is the totally ordered set of *time instant*s along the corresponding timeline. In particular, for simulation time, the time line is called the *federation time axis* (common across all federates), and the *federate time* is a specific federate's current time instant along the federation time axis up to which the federate has advanced its simulation.

Execution Pacing

In general, there is a one-to-one mapping from physical time to simulation time. In contrast, there may or may not exist a specific relationship between simulation time and wallclock time. The mode of simulation execution determines this particular relationship. In an *as-fast-as-possible* execution, the simulation time is advanced as fast as computing speed can allow, unrelated to wallclock time. In *real-time* execution, on the other hand, advances in simulation time are performed in lockstep with wallclock time – one unit of simulation time is advanced exactly in one same unit of wallclock time. A variation of real-time execution is *scaled real-time* execution, in which simulation time period is some constant factor times an equivalent wallclock time period.

Events and Event Orderings

An event is an indication of an update to simulation system state at a specific simulation time instant. Thus each event specifies a timestamp. When events are exchanged among federates, their delivery at the receiving federates needs to be carefully coordinated at runtime. In general, multiple different types of delivery ordering systems can be defined. Two such orderings, employed by the HLA, are (1) *receive-order* (2) *timestamp-order*. Other types[6], such as causal order[7], could also be useful in certain cases, but they are not as commonly used.

In receive-ordered delivery (RO), events from other federates are delivered to the receiving federate as and when the events arrive at the receiving federate. In contrast, in timestamp-ordered delivery (TSO), events are guaranteed to be delivered in non-decreasing order of their timestamps. Typically, since RO delivers events right away, RO events incur lower delivery delay/latency from the moment they are sent by a federate to the moment the destination federate(s) receives them. TSO events on the other hand undergo runtime checks and buffering until their non-decreasing timestamp order can be ascertained and guaranteed, and hence TSO events incur relatively higher latency. However, a significant difference arises with respect to modeling accuracy afforded by RO and TSO. RO cannot always preserve "before and after" relationships, while TSO does guarantee preservation of such relationships. Similarly, with TSO, all federates see the exact same ordering of events, whereas with RO, identical ordering among events cannot be guaranteed across federates. Federation execution can be made repeatable with TSO from one execution to the next, while RO cannot ensure such repeatability.

Timestamp-Ordered (TSO) Processing

The rationale behind timestamp-ordered processing is that it permits the models to be accurately simulated, such that events are processed in the same order as their corresponding actions in they physical system. To enable such processing order, a simple local rule is that a federate whose simulation time is at T should not receive events with timestamps less than T. Hence, advances of federate's current time have to be coordinated and controlled carefully to prevent events appearing in federate's "past"

The HLA's TM services thus address two important components: (1) overall event processing order by each federate (2) synchronized event delivery to each federate.

Interoperability Challenge
While enabling event processing order and synchronized event delivery, all in a single encompassing standard framework, the HLA needs to accommodate a large variety of individual types of simulators.

In general, there is a plethora of different simulator types – event-stepped vs. time-stepped, sequential vs. parallel, real-time vs. as-fast-as-possible, conservative vs. optimistic, etc. An HLA federation might include any combination of any of the simulator types. Moreover, the exact combination of the types is not known *a priori* to the HLA RTI, and hence the interface as well as the implementation must be sufficiently general to accommodate any/all of them. The HLA TM interface does an amazing job of accommodating any arbitrary combinations of, and any number of instances of, different types of simulators, all in one core, seamless interface.

BASIC HLA TIME MANAGEMENT SERVICES

The HLA's Time Management services are highly parameterized to support the wide variety of timestamp-ordered synchronization requirements of HLA federates. The federates first declare their roles, along with their key concurrency parameters. They then utilize the RTI's TM services to control simulation time advances and timestamp-ordered event delivery.

Federate Roles
In order to participate in the time managed portion of the federation, a federate must declare such intent, by setting its *time regulating* and *time constrained* flags. A federate whose time regulating flag is turned on acts in a role in which it can send TSO messages and hence can prevent other federates from advancing their simulation time. A federate whose time constrained flag is turned on acts in a role in which it can receive TSO messages and hence can be constrained by other time regulating federates. Note that a federate can be both time regulating and time constrained at the same time (typically used for analytic simulations[8]). An example of a time regulating-only federate is a message source. An example of a time constrained-only federate is a Stealth Display. Federates such as training simulators are examples of those that are neither time regulating nor time constrained (such federates effectively turn off time synchronization for their events).

Request-Update-Grant Scheme
The HLA TM interface provides a particular request-update-grant scheme to realize timestamp-ordered event delivery and federation-synchronized simulation time advances. In this scheme, federates explicitly request the RTI for permission to advance to a certain simulation time. There are fundamentally three basic types of such requests – Time Advance Request (TAR), Next Event Request (NER) and Flush Queue Request (FQR). TAR is typically used by time-stepped federates, NER by event-driven federates and FQR by optimistic event-driven federates or other advanced federates. At a later time, as and when the RTI deems fit, the RTI invokes a Time Advance Grant (TAG) callback on the federate to notify the federate permission to advance to a given time specified in the TAG. The RTI bases its decision to issue a TAG on many factors, including a distributed computation of a safe time called Lower Bound on Time Stamp (LBTS), as will be explained in a later section.

In this section, we will focus on TAR and NER services only, and will cover FQR in a later section.

Time Advance Request
A Time Advance Request (TAR) is typically used by time-stepped federates. A time-stepped federate is one that performs its processing in fixed increments of simulation time, irrespective of timestamps of events it receives. Upon completing processing until a simulation time T, the federate is ready to advance to a time $T+dt$, where dt is determined independent of any incoming future events.

The federate invokes *TimeAdvanceRequest(T)* to request the RTI's permission to advance its simulation time to T. This informs the RTI to deliver all those events destined for this federate whose timestamps are less than T. Once the RTI can guarantee the delivery of all such events, it issues a *TimeAdvanceGrant(T)* to advance the federate's simulation time to T. In the interim period between a *TimeAdvanceRequest(T)* and *TimeAdvanceGrant(T)*, the federate continually invokes the *tick()* method to grant computation cycles to the RTI. Note that the granted time is always equal to the requested time.

Next Event Request
A Next Event Request (NER) is typically used by discrete event federates. This service is intended to facilitate each federate in processing all its events in timestamp-order, irrespective of whether the events were generated locally or received from other federates.

Suppose T_E is the earliest timestamp of events locally present for processing in the federate. In the most common use of NER, the federate invokes *NextEventRequest(T_E)* when T_E is less than the most recently granted time, T_{TAG}, obtained via a *TimeAdvanceGrant(T_{TAG})* from the RTI. The RTI then works to ensure one of two things: either (1) all events eventually generated & destined to this federate by other federates necessarily have timestamps greater than or equal to T_E, or (2) finds and delivers to this federate one or more events whose timestamps are less than T_E. In the former case, the RTI issues a *TimeAdvanceGrant(T_E)*. In the latter case, it first delivers all appropriate events/interactions via *ReflectAttributeValues()* and then issues a *TimeAdvanceGrant(T_{RAV})*. Where T_{RAV} is the timestamp value of delivered events, $T_{RAV} < T_E$.

Lookahead

A fundamental problem with TAR-based and NER-based federations is concerned with the concept of lookahead. For simplicity, let us focus on NER. In the absence of the concept of lookahead, suppose any federate that is processing an event with timestamp *T* can generate another event, whose timestamp is also equal to *T*, to another federate. Moreover, this new event could be destined to any or all federates. In such a scenario, in order to ensure timestamp-ordered processing, it is clear that there is little concurrency among federates. Only the event with the globally minimum timestamp in the entire system can be processed at its federate, while all the rest of the federates necessarily have to stay idle. In other words, only one federate gets a grant to its requested NER time, while the rest of the federates are "idling" in *tick()*. Essentially, this degenerates to sequential execution, albeit with multiple federates. Clearly, this is undesirable in interest of runtime performance. It becomes desirable to uncover concurrency among federates to avoid such serialization. The concept of lookahead is defined to resolve this problem.

Lookahead is defined as the minimum increment in simulation time between an event and any new events generated during processing of that event. When this lookahead is greater than zero at all federates, the federation can experience concurrency. If the lookahead is zero for any federate (i.e., a federate can generate events with zero delay), then the entire federation suffers from serial execution (discounting unrelated events with equal timestamps at different federates).

In simulation models, it is possible to extract lookahead by examining the minimum time for interactions to occur among entities. For example, speed-of-light delays could be used to compute minimum propagation delays across radio/satellite entities. In other models, it might be difficult to extract non-zero lookahead. Lookahead extraction is a topic of much research, and unfortunately remains a challenge in its generality[9, 10].

HLA TM provides a *SetLookahead()* service for the federates to specify zero or positive lookahead on a federate-by-federate basis. The minimum lookahead among all federates is then used by the RTI for enabling concurrency.

INSIDE RTI – TM IMPLEMENTATION APPROACHES

It is usually sufficient for federate developers to be conversant with the HLA TM services as far as usage of its interface is concerned. However, it is helpful to also be familiar with some implementation approaches taken by RTI vendors in implementing the TM interface primitives. In particular, it is useful to be aware of distributed computation factors, such as message and computation performance overheads, incurred by the federate (and the federation) when certain primitives are used in a certain way. One example of this has already been covered in the previous section, namely, the performance effects of specifying zero vs. positive lookahead. Here, we will examine additional computational effects of TM services.

Centralized vs. Distributed

Like most distributed computation problems, TM services can generally be implemented in two ways: *centralized approach* and *fully distributed approach*.

In a centralized approach, a single, designated computer acts as the RTI's "time management gateway" for the entire federation. All time-synchronized operations are routed by the RTI through this gateway. For example, time advance requests and grants are coordinated by the gateway. Since the gateway holds information about the state of all federates, the gateway can decide on and satisfy most requests in a straightforward fashion. The centralized approach affords great simplicity of design as well as ease of debugging and testing, and hence employed by some commercial RTI vendors. The drawbacks of this approach include existence of a single point of failure, and potentially higher runtime overhead. The latency of services could also increase due to the need to contact the gateway for most operations.

In a fully distributed approach, every federate's computer node directly undertakes a synchronization role in a peer-to-peer fashion. Fully distributed algorithms are then employed to realize time-synchronized event delivery and processing. This approach is used successfully in some high-performance RTI implementations[11]. While faster execution is an advantage (as also a more uniform synchronization load across all federate nodes), the drawback is significant increase in implementation complexity.

Computing LBTS

A fundamental role of a TM implementation is in computing a quantity known as Lower Bound on incoming Time Stamps (LBTS). At each federate, the LBTS value specifies a guarantee on the least timestamp on any future incoming event. In other words, no event will ever arrive at that federate with a timestamp smaller than LBTS. Once this global value is known, it is rather straightforward to locally serve TM requests, such as TAR, NER and FQR. In order to compute the LBTS value at each federate, a distributed algorithm is required that exchanges messages to coordinate the LBTS computation without deadlocks, live-locks or undue performance degradation. Several such algorithms have been proposed in PADS literature[12]. A close cousin to the LBTS computation is GVT computation in optimistic simulation[13]. Another closely related work in general distributed processing is that of distributed "flush barrier" algorithms[14]. Analogous to these algorithms, several variants exist for LBTS computation.

One such algorithm is based on global asynchronous distributed reductions. In this algorithm, the minimum local (conditional) guarantee on timestamps of events that could be generated is taken at each federate, and a global reduction algorithm is used to find the minimum of all the local minima. This can be performed fairly quickly and scalably, in $\log(N_p)$ steps, where N_p is the number of federates, using a butterfly pattern of communication[15]. Assuming there are no events in transit across federates, the minimum of the minima gives a tight lower bound on LBTS.

Transient Messages

What if there are some events that are in transit in the network while the global minimum of local minima is being computed? This is called the transient event problem, in which some events could become potentially unaccounted for if they are not considered into the global algorithm. There exist several schemes by which transient events can be accounted for, albeit at the cost of either additional messages being sent/received and/or additional time spent blocking while waiting for all transient events to reach their destinations. A popular one is called the Mattern's algorithm[16] in which distributed consistent cuts are used to mark and recognize events belonging to distributed different snapshots.

It is clear that the larger the lookahead, the fewer the number of LBTS computations that need to be performed.

Serving Requests

The RTI internally maintains a priority queue of TSO events, ordered by their timestamps. When a federate invokes *TAR(T)*, the RTI first examines if LBTS is greater than *T*. If so, the request is trivially satisfied – the RTI delivers all events from its TSO queue whose timestamps are less than or equal to *T*, and then issues a *TAG(T)*. If *T* is greater than LBTS, then the RTI initiates a new distributed LBTS computation (if one is not already in progress). The lesser of *T* and minimum timestamp in TSO queue is used as this federate's contribution in the LBTS computation. The operation is similar for *NER(T)* invocations as well, except that the TAG time could be smaller than *T* if events with timestamps earlier than *T* are delivered.

ADVANCED HLA TIME MANAGEMENT SERVICES

In addition to supporting basic integration of conservative federates, the HLA TM services include some additional primitives to integrate federates that use advanced simulation methods, such as Time Warp-style optimistic simulation[17].

In general, time synchronization approaches are categorized as conservative or optimistic. In a conservative execution, synchronization errors are always avoided by avoiding the possibility of events arriving at a federate after the federate's local simulation time has been advanced to beyond the new events' timestamps. Conservative algorithms tradeoff idle time for achieving such safety by blocking event processing. Optimistic algorithms on the other hand, do not block, but instead process events without regard to potential order violations. If and when timestamp order violation is detected, a rollback mechanism is used to recover from erroneous computation. Conservative federates are easier to implement, but rely on existence of (large) positive lookahead values. Optimistic federates, on the other hand, are more complex to implement, but are resilient to zero

lookahead (i.e., work well even under low statically-determinable concurrency).

Retractions

In simulations, models are sometimes written to un-schedule previously generated events. For example, although a *move* event is scheduled on an entity at *T*, it might have to be retracted later if the entity gets destroyed after the event is scheduled but prior to *T*. Such event retractions are called user-level retractions. Typically, user-level retractions are enabled as follows. When an event is scheduled, the system returns a handle to that event. Later, if and when that event needs to be retracted, a *retract* primitive is invoked to which the event handle is given. The system then un-schedules that event. The HLA RTI provides such a framework using event handles and retraction primitive. Interestingly, the same service is also used for "system-level" retraction in optimistic simulations, as described next.

Optimistic Time Management

As mentioned previously, the HLA supports conservative federates as well as optimistic federates, as well as their arbitrary combinations. Optimistic federates differ from their conservative counterparts in that they do not discard events after processing them. Instead they keep the events around, and also maintain copies of simulation states before modifying them as part of event processing. Since optimistic federates do not rely on lookahead, they execute their events without blocking for safety. In particular, they use the *FlushQueueRequest(T)* service of the RTI to force the RTI to deliver events from its TSO queue even if LBTS has not progressed past *T*. The difference between FQR and NER is that FQR does not guarantee that it has delivered all events with timestamp less than *T*. Thus, the federate will have to rollback[18] its computation if/when it later receives events whose timestamp is less than T. There are two main parts to such rollback: (1) undo local computation by restoring the state prior to erroneous event processing (2) undo all events erroneously sent to other federates. The first part is typically federate-specific, and hence the HLA does not provide a standard service for it. The second part is realized by using the event retraction service described previously. When an optimistic federate receives a retraction request, it performs an event annihilation procedure canceling the original event.

Note that the HLA RTI shields conservative federates from optimistic events by holding on to optimistic events in RTI TSO queues until such a time that LBTS sweeps past their timestamps. If the optimistically scheduled events happen to get retracted by their sending federates, those events will get annihilated within the RTI's TSO queues without ever getting delivered to the (conservative) destination federate.

Simultaneity

The notion of simultaneity arises when the timestamps of two or more events are exactly equal[19]. There are many ways in which simultaneous events could be generated. One simple way is when an event generates another event with zero delay in a zero-lookahead federation[20]. Another way is when two federates generate two otherwise unrelated events whose timestamps happen to be exactly the same (e.g., due to pure coincidence with random number generators)[21]. The former way is rather more difficult to resolve than the latter: zero-lookahead federations present problems with interface definition with TAR and NER. The latter can be resolved using tie-breaker fields so two timestamps cannot be coincidentally equal[21].

Given a *TAR(T)* or *NER(T)*, how can the RTI guarantee that all events with timestamps less than or equal to *T* have all been delivered before a *TAG(T)* is issued? Clearly, with zero lookahead, any event with time *T* can generate another event with same time *T*. This presents a semantic problem with the definition of TAR and NER.

In order to resolve this impasse due to simultaneity of events, new variants of TAR and NER are provided – Time Advance Request Available (*TARA*) and Next Event Request Available (*NERA*). A *TARA(T)* invocation allows the RTI to issue *TAG(T)* with the proviso that more events with timestamp equal to *T* might be delivered later. Similarly, an *NER(T)* invocation by a federate allows the RTI to safely issue a *TAG(T)* to the federate with the proviso that not all events with timestamp strictly equal to *T* have been accounted for.

QUESTIONS

1. *What are the two common types of event ordering most commonly used in HLA federations? Which among the two incurs lesser runtime overhead? Which ensures better modeling accuracy?*

2. *Describe what is meant by lookahead of a federate? Explain why it is generally desirable to have the value of lookahead to be as large as possible.*

3. *List the three principal RTI primitives that a federate can invoke to request synchronization of events and simulation time with the rest of the federation.*

REFERENCES

[1] R. Bagrodia, K. M. Chandy, and W. T. Liao, "A Unifying Framework for Distributed Simulation," *ACM Transactions on Modeling and Computer Simulation*, vol. 1, pp. 348-385, 1991.
[2] K. M. Chandy and R. Sherman, "Space, Time, and Simulation," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 21: SCS Simulation Series, 1989, pp. 53-57.
[3] R. M. Fujimoto, "Parallel Discrete Event Simulation," *Communications of the ACM*, vol. 33, pp. 30-53, 1990.
[4] J. Misra, "Distributed Discrete Event Simulation," *ACM Computing Surveys*, vol. 18, pp. 39-65, 1986.
[5] P. F. Reynolds, Jr., "A Spectrum of Options for Parallel Simulation," in *Proceedings of the 1988 Winter Simulation Conference*, 1988, pp. 325-332.
[6] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, pp. 558-565, 1978.
[7] B.-S. Lee, W. Cai, and J. Zhou, "A Causality Based Time Management Mechanism for Federated Simulations," in *Proceedings of the 15th Workshop on Parallel and Distributed Simulation*, 2001, pp. 83-90.
[8] A. L. Wilson and R. M. Weatherly, "The Aggregate Level Simulation Protocol: An Evolving System," in *Proceedings of the 1994 Winter Simulation Conference*, 1994, pp. 781-787.
[9] B. A. Cota and R. G. Sargent, "A Framework for Automatic Lookahead Computation in conservative Distributed Simulations," in *Proceedins of the SCS Multiconference on Distributed Simulation*, vol. 22: SCS Simulation Series, 1990, pp. 56-59.
[10] E. Deelman, R. Bagrodia, R. Sakellariou, and V. Adve, "Improving Lookahead in Parallel Discrete Event Simulations of Large-Scale Applications using Compiler Analysis," in *Proceedings of the 15th Workshop on Parallel and Distributed Simulation*, 2001, pp. 5-13.
[11] R. M. Fujimoto, T. McLean, K. Perumalla, and I. Tacic, "Design of High Performance RTI Software," presented at Workshop on Distributed Interactive Simulations and Real-Time Applications, 2000.
[12] K. Perumalla and R. Fujimoto, "Virtual Time Synchronization over Unreliable Network Transport," presented at Workshop on Parallel and Distributed Simulation, 2001.
[13] S. Bellenot, "Global Virtual Time Algorithms," in *Proceedings of the SCS Multiconference on Distributed Simulation*: Society for Computer Simulation, 1990, pp. 122-127.
[14] M. Ahuja, "Flush Primitives for Asynchronous Distributed Systems," *Information Processing Letters*, pp. 5-12, 1990.
[15] D. E. Brooks, "The Butterfly Barrier," *The International Journal of Parallel Programming*, vol. 14, pp. 295-307, 1986.
[16] F. Mattern, "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation," *Journal of Parallel and Distributed Computing*, vol. 18, pp. 423-434, 1993.
[17] D. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 404-425, 1985.
[18] A. Gafni, "Rollback Mechanisms for Optimistic Distributed Simulation Systems," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 19, *SCS Simulation Series*, 1988, pp. 61-67.
[19] V. Jha and R. Bagrodia, "Simultaneous Events and Lookahead in Simulation Protocols," University of California, Los Angeles 1996.
[20] R. M. Fujimoto, "Zero Lookahead and Repeatability in the High Level Architecture," in *Proceedings of the Spring Simulation Interoperability Workshop*: Paper 046, 1997.
[21] H. Mehl, "A Deterministic Tie-Breaking Scheme for Sequential and Distributed Simulation," in *Proceedings of the Workshop on Parallel and Distributed Simulation*, vol. 24, M. Abrams and P. Reynolds, Jr., Eds.: Society for Computer Simulation, 1992, pp. 199-200.