Design of High Performance RTI Software

Richard Fujimoto, Thom McLean
Kalyan Perumalla, and Ivan Tacic
College Of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
{fujimoto,mclean,kalyan,ivant}@cc.gatech.edu

Abstract

This paper describes the implementation of RTI-Kit, a modular software package to realize runtime infrastructure (RTI) software for distributed simulations such as those for the High Level Architecture. RTI-Kit software spans a wide variety of computing platforms, ranging from tightly coupled machines such as shared memory multiprocessors and cluster computers to distributed workstations connected via a local area or wide area network. The time management, data distribution management, and underlying algorithms and software are described.

Keywords: High Level Architecture, runtime infrastructure, time management, data distribution management

1. Introduction

Composing autonomous simulators and/or simulation components has become an accepted paradigm to realize parallel/distributed simulation systems. For example, this is the approach used in the High Level Architecture (HLA) that has become the standard technical architecture for modeling and simulation in the U.S. Department of Defense [1]. Such systems require runtime infrastructure (RTI) software to provide services to support interconnecting simulations as well as to manage the distributed simulation execution. One component of the HLA, the Interface Specification (IFSpec) [2], defines the set of services that are used by individual simulations to interact with each other.

A distributed simulation in the HLA is referred to as a *federation*. Each simulator is referred to as a *federate*. This paper is concerned with the implementation of runtime infrastructure (RTI) software. Here, we are particularly concerned with implementation of the services defined in version 1.3 of the HLA IFSpec.

The HLA spans a broad range of applications with diverse computation and communication requirements. We are concerned with realizing RTI software that can span a broad range of computing platforms with widely varying cost and performance characteristics. The RTI software must execute efficiently on tightly coupled machines such as shared memory multiprocessors or workstation clusters using high-speed interconnects. At the same time, the same software should be configurable to realize distributed simulations interconnected over local or wide area networks.

2. Related Work

To date, most work on HLA RTI software has focused on networked workstations using well-established communication protocols such as UDP and/or TCP. While such implementations are sufficient for large portions of the M&S community, many applications require higher communication performance than can be obtained utilizing these interconnection technologies. Shared memory multiprocessors and cluster computing platforms offer high performance alternatives.

A few systems have been adapted for use in high performance computing platforms. Early versions of the RTI-Kit software described here for cluster and shared memory multiprocessors are described in [3, 4]. An implementation of RTI version 1.3 (dubbed the DMSO RTI) for shared memory multiprocessors was developed by the MIT Lincoln Laboratory [5, 6] Adaptation of the SPEEDES framework to realize an HLA RTI is described in [7].

3. RTI-Kit

RTI-Kit is a collection of libraries designed to support development of Run-Time Infrastructures (RTIs) for parallel and distributed simulation systems. Each library can be used separately, or together with other RTI-Kit libraries, depending on what functionality is required. These libraries can be embedded into existing RTIs, e.g., to add new functionality or to enhance performance by exploiting the capabilities of a high performance interconnect. For example, RTI-Kit

software was successfully embedded into an HLA RTI developed in the United Kingdom [3, 8]. Alternatively, the libraries can be used in the development of new RTIs.

This "library-of-libraries" approach to RTI development offers several important advantages. First, it enhances the modularity of the RTI software because each library within RTI-Kit is designed as a stand alone component that can be used in isolation of other modules. Modularity enhances maintainability of the software, and facilitates optimization of specific components (e.g., time management algorithms) while minimizing the impact of these changes on other parts of the RTI. This design approach facilitates technology transfer to other RTI development projects because utilizing RTI-Kit software is not an "all or nothing" proposition; one can extract modules such as the time management while ignoring other libraries.

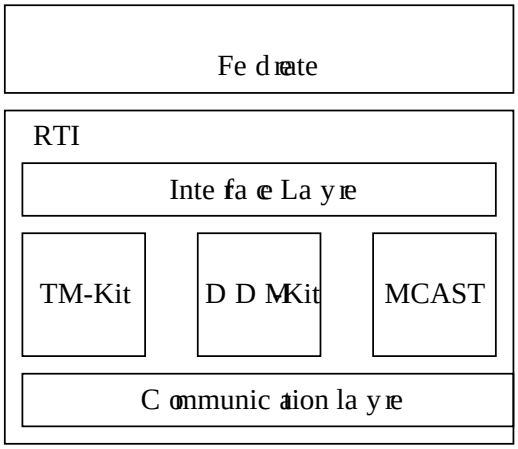
Multiple implementations of the RTI-Kit software have been realized targeting different platforms. Specifically, the current implementation can be configured to execute over shared memory multiprocessors such as the SGI Origin, cluster computers such as workstations interconnected via a low latency Myrinet switch [9], to workstations interconnected over local or wide area networks using standard network protocols such as IP.

The architecture for RTI software constructed using RTI-Kit is shown in Figure 1. At the lowest level is the communication layer that provides basic message passing primitives. Communication services are defined in a module called FM-Lib. This communication layer software acts as a multiplexer to route messages to the appropriate module. The current implementation of FM-Lib implements reliable point-to-point communication. It uses an API based on the Illinois Fast Messages (FM) software [10] for its basic communication services, and provides only slightly enhanced services beyond those of FM.

Above the communication layer are modules that implement key functions required by the RTI. These modules form the heart of the RTI-Kit software. Specifically, *TM-Kit* is a library that implements distributed algorithms for realizing time management services. Similarly, *DDM-Kit* implements functionality required for data distribution management services. *MCAST* is a library that implements group communication services. Other libraries, not shown in Figure 1, provide other utilities such as software for buffer and queue management.

Finally, the interface layer utilizes the primitive operations defined by these modules to implement a specific Application Program Interface (API) such as the HLA Interface Specification. The current RTI-Kit distribution includes an implementation of a subset of the HLA IFSpec (version 1.3).

The RTI-Kit architecture is designed to minimise the number of software layers that must be traversed by distributed simulation services. For example, TM-Kit does not utilise the MCAST library for communication, but rather directly accesses the low-level primitives provided in FM-Lib. This is important in cluster computing environments because low level communications are on the order of a few microseconds latency for short messages, compared to hundreds of microseconds or more when using conventional networking software such as TCP/IP. Thus, if not carefully controlled, overheads introduced by RTI software could severely degrade performance in cluster environments, whereas such overheads would be insignificant in traditional networking environments where the time required for basic communication services is very high. Measurements indicate the overheads introduced by RTI-Kit are small; a federation of optimistic sequential simulators



based on the Georgia Tech Time Warp (GTW) software interconnected via RTI-Kit was observed to yield performance comparable to the native, parallel, GTW implementation [11].

Figure 1. RTI architecture using RTI-Kit.

4. Time Management

There are two principal components to the HLA time management (TM) services. First, a time stamp ordered (TSO) message delivery service guarantees that successive messages delivered to each federate have non-decreasing time stamps. Second, the time management services manage simulation time (termed logical time in the HLA) advances of each federate.

Federates must explicitly request that their logical time be advanced by invoking an IFSpec service such as *Next Event Request, Time Advance Request, or Flush Queue Request* (see Figure 2). The RTI only grants the advance via the *Time Advance Grant* service (callback) when it can guarantee that no TSO messages will later be delivered with a time stamp smaller than the granted advance time. In this way the RTI ensures federates never receives messages with time stamp less than the federate's current logical time. See [12] for additional details on the time management services.

In the HLA, time management is distinct from sending and receiving messages (events). Services such as *Update Attribute Values* and *Reflect Attribute Values* are used to send and receive messages, respectively.

To facilitate the development of time management services, a separate module called TM-Kit was developed in RTI-Kit. This same TM-Kit module can be utilized to implement and experiment with different implementations of the HLA TM services.

4.1 Time Types

Logical time values in TM-Kit are defined as an abstract data type called **TM_Time**. Like the HLA, this data type may be defined arbitrarily; it can be as simple as an integer or as complex as a tuple of values that includes priorities and other fields to break ties. In addition, comparison and other operators on this data type must be defined. In order to maximize performance, the current implementation of TM-Kit implements operations on time types using macros. Thus, the time type and associated macros must be defined when TM-Kit is compiled. In the case of federates using C++, a simulation time class can be defined as a wrapper around this **TM_Time** data type.

4.2 TM-Kit API

TM-Kit provides primitives for computing a lower bound on the time stamp (LBTS) of future messages that could later be received by a federate. The RTI TM software uses these primitives to both control time advances as well as regulate event delivery. In this sense, TM-Kit can be viewed as simply a distributed LBTS calculator over which services such as RTI TM are easily implemented. See [13] for an in depth discussion of algorithms to compute LBTS.

TM-Kit itself does not directly handle time stamped messages. Instead, the interface layer software built over TM-Kit is responsible for dealing with message queuing and timestamp ordered delivery. The TM-Kit merely requires that it be informed of very simple information such as how many TSO messages are sent or received over the network by the RTI between two successive LBTS computations. This carefully designed demarcation of responsibility permits TM-Kit to be easily imported into other RTI implementations.

The central procedures in the TM-Kit API are described next (see Figure 2):

- **TM_StartLBTS:** The RTI in any processor can call this procedure to initiate a new LBTS computation. If two different processors simultaneously and independently invoke this primitive, the resulting two computations are automatically merged, and only one new LBTS computation is actually started.
- **LBTS_Started:** This procedure is a callback indicating another processor has initiated a new LBTS computation. TM-Kit invokes this callback to retrieve logical time information from this federate for this new LBTS computation. Specifically, the federate must provide the minimum time stamp of any future message it might produce, assuming no additional TSO messages are later delivered to the federate.

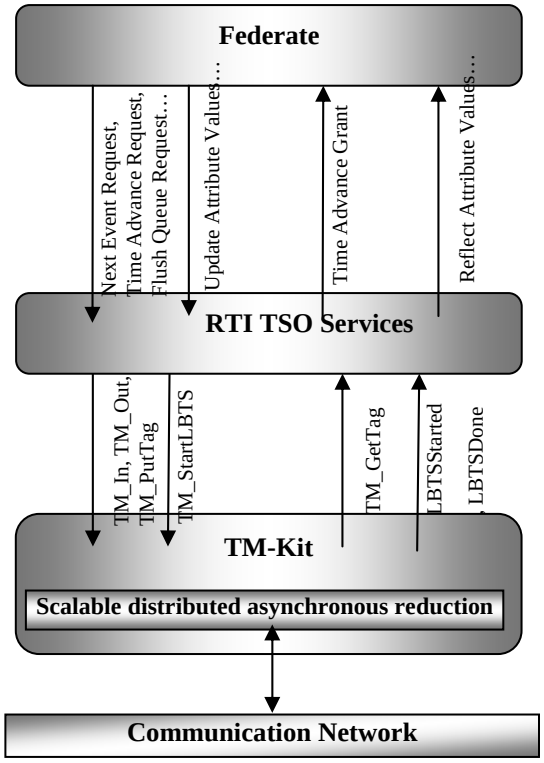


Figure 2. TM-Kit interface and implementation.

- **LBTS_Done:** This procedure is a second callback

that the TM-Kit invokes to indicate that an LBTS computation has completed. The newly computed LBTS value is passed as an argument.

- **TM_In** and **TM_Out:** These two procedures form the mechanism by which information about transient messages is indicated to the TM-Kit. Transient messages are those that have been sent, but have not yet been received while the LBTS computation is taking place. **TM_Out** must be called whenever a TSO message is sent, and **TM_In** must be called whenever one is received. This information is sufficient for TM-Kit to take transient messages into account to correctly compute the LBTS.
- **TM_PutTag** and **TM_GetTag:** These procedures provide a means for the TM-Kit software to piggyback and retrieve important control information in event messages. **TM_PutTag** is called prior to sending a message in order to place time management information in the message. **TM_GetTag** is called at the destination to extract the time management information from a received message.

Different approaches may be used to initiate new LBTS computations. For example, each processor might asynchronously start a new computation whenever it needs a new LBTS value to be computed; as discussed earlier, the TM-Kit software automatically merges multiple, simultaneous initiations of new LBTS computations by different processors into a single LBTS computation. Alternatively, a central controller could be used to periodically start a new LBTS computation at fixed intervals of wallclock time, or using some other criteria.

4.3 TM-Kit Implementation

The heart of the LBTS software in the TM-Kit is a scalable, distributed, asynchronous reduction engine. Each LBTS computation is realized as a series of reduction operations. Each reduction operation is aimed at computing the reduction of processor values along a consistent distributed snap shot. The value at each processor *i* is a pair $<L_i,M_i>$, where L_i is the local conditional lower bound on future timestamps that can be generated by processor *i*, and M_i is the difference between the counts of total sent and received messages at processor *i* since the previous LBTS computation. The L_i values are reduced with the minimum operator, while the M_i values are reduced using the addition operator. The LBTS computation terminates successfully when the sum of all M_i becomes zero. All processors receive the resulting LBTS value as the minimum among all L_i.

The distributed reduction engine employed here differs from other work such as [14] in that our algorithm is general-purpose in nature, and not tied to any specific type of communication network. In particular, it is designed to work efficiently over shared-memory, local area and wide area networks. Broadcast communication is never employed in the reduction algorithm, and hence the reduction engine exhibits high scalability, while retaining optimal logarithmic time complexity. Also, no barriers are used in the computation, and the algorithm operates completely asynchronously.

The reduction engine itself is a module that is independent of TM-Kit, and hence can be reused for other purposes as well. The software for both the reduction engine as well as the TM-Kit software is compact. The reduction engine consists of approximately 1000 lines of code, while the TM-Kit consists of an additional 500 lines. The architecture of this software is carefully designed to accommodate adaptive and hierarchical approaches to LBTS computation for heterogeneous communication platforms.

4.4 Distributed Reduction

In the distributed algorithm employed by the reduction engine, each processor *i* executes an ordered sequence of actions, $S_i=<a^1_i,\ldots,a^m_i>$, called its schedule. (The number of actions in the schedule can be different for different processors). Each action $a=s_j$ (or $a=r_j$) corresponds to a *send* to (or *receive* from) another processor *j*. The reduction proceeds as follows: each processor *i* attempts to process as many actions as possible in its schedule S_i in its specified order. If an action is a *receive* action, $a=r_j$, and processor *j* has not yet sent its value to processor *i* then the schedule execution blocks at this *receive* action until such time that the value is received from processor j. When the value is received, it is immediately reduced with the processor's current reduction value. Thus, values received from other processors are reduced in the order in which their corresponding *receive* actions appear in the schedule. A *send* action, $a=s_j$, in the schedule is processed by sending a value *v* to processor j, where *v* is equal to the (partially reduced) value obtained by reducing all received values from the beginning of the schedule until this *send* action. The global reduction completes when all the processors successfully complete the execution of their schedules.

The schedules are carefully designed in such a way that all processors compute precisely the same final reduced value by the end of all schedule executions. Several different schedules holding this property are possible, corresponding to different communication

patterns for reduction (e.g. “all-to-all”, “star” and “butterfly”). In particular, we have implemented a variant of the butterfly communication pattern which guarantees important scalability properties: ensuring optimal logarithmic complexity for the time to complete the reduction, while also limiting to logarithmic complexity the number of message sends and receives performed by any single processor.

The convenient abstraction of a schedule, coupled with the customizable distributed reduction algorithm, allows one to easily vary and experiment with different communication alternatives on different communication platforms (e.g, Ethernet LAN, TCP wide-area networks and shared memory), with few modifications to the software.

4.5 LBTS Computation

TM-Kit's LBTS computation is built over the distributed reduction software. Each LBTS computation involves one or more reduction phases. Each reduction phase is called a trial, which computes a snapshot across all processors of their individual conditional lowerbounds on timestamps of future messages they can generate. These snapshots may not correspond to a consistent global snapshot because of transient messages that might not have been accounted for in the snapshot. A count of the total number of messages sent and received at each processor is included in the reduction. Thus, as part of the reduced value, all the processors obtain information on the number of outstanding (transient) messages, which signals to them either that the snapshot is in fact consistent (if the number of outstanding messages is zero), or that they need to retry the reduction. The LBTS computation ends successfully when the last reduction phase indicates a consistent snap shot.

It might initially appear as though multiple reduction phases can be inefficient. However, it should be noted that in a network with ordered delivery (e.g., TCP, Myrinet, shared memory) successive reductions increase the probability that all transient messages will be flushed and delivered before the later reduction completes, leading to rapid algorithm convergence.

5. Data Distribution Management

Data Distribution Management (DDM) services are used to specify the routing of data among federates. In the HLA, DDM is based on an n-dimensional coordinate system called a routing space. For example, a two-dimensional routing space might represent the play box in a virtual environment. A rectangular update region can be associated with each update message generated by a federate. Federates express interests via rectangular subscription regions. If the update region associated with a message overlaps with a federate's subscription region, the message is routed to that subscribing federate. For example, in Figure 3 updates using update region U are routed to federates subscribing to region S_1 but not to federates subscribing to region S_2.

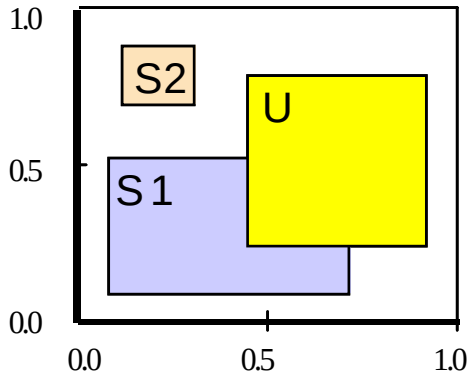


Figure 3. Two-dimensional routing space with subscription regions S_1 and S_2 and update region U.

5.1 Implementation Approaches

DDM-Kit uses multicast services (implemented in the MCAST library) to realize communications among federates. MCAST provides standard group communication services (join, leave, and send messages to groups). A central problem in realizing the DDM services concerns the definition and composition of the multicast groups. Subscription regions must be mapped to groups to which the federate must join. Update regions associated with a message are mapped to one or more groups to which the message must be sent.

Two well-known approaches to realizing DDM are to form groups based on (1) grids and (2) update regions. As will be seen momentarily, the grid-based approach provides a simple means to match update and subscription regions, but tends to utilize a large number of multicast groups, and can result in duplicate or extra messages that must be filtered at the receiver. The update region approach avoids these drawbacks, but at the cost of greater complexity (and runtime overhead) to match update and subscription regions. DDM-Kit uses a variation on the update region approach using grid cells to reduce matching overhead. Each of these are described next.

5.1.1 Region-Based Groups

In the regions based approach a multicast group is defined for each update region [15]. Updates are

simply sent to the group associated with the update region. A federate subscribes to the group if one or more of its subscription regions overlap with the update region.

When a subscription region changes, the new subscription region must be matched against all other update regions in order to determine those that overlap with the new subscription region. The federate must then subscribe to the groups with overlapping update regions. Similarly, when an update region changes, the new update region must be matched against all subscription regions to determine the new composition of the update region's group. This requires examining all subscription/update regions in use by the federation. Thus it does not scale well as the number of regions becomes large.

5.1.2 Grid-Based Groups

In the grid-based approach the routing space is partitioned into non-overlapping grid cells, and a multicast group is defined for each cell [13, 16]. A federate subscribes to the group associated with each cell that partially or fully overlaps with its subscription regions. An update operation is realized by sending an update message to the groups corresponding to the cells that partially or fully overlap with the associated update region.

A federate may have multiple subscription regions overlapping a specific grid cell. To avoid multiple subscriptions to the group, each grid cell can maintain a subscription count array with an entry for each federate that indicates the number of subscription regions for that federate that overlap this cell. The federate leaves the group if this count becomes zero during a subscription region change. Similarly, the federate will join the group if its count becomes non-zero.

The grid-based approach eliminates the need to explicitly match update and subscription regions. While grid partitioning eliminates the matching overhead, a large number of groups is needed if a fine grid structure is defined; a coarse grid leads to imprecise filtering, negating some of the benefits of DDM. In addition, the grid scheme has other shortcomings:

- *Duplicate messages* may occur. For example, if a subscription and update region both overlap with the same two cells, two identical copies of the message will be sent to the subscribing federate over different multicast groups. These must be filtered at the receiver, incurring additional overhead.
- *Extra messages* may occur. This is a direct result of discretizing the routing space into grid cells. Subscription and update regions may overlap with the same grid cell, but may not overlap with each other. In this case, a message will be sent to the subscribing federate, even though its subscription region does not overlap with the update region. These unwanted messages will also have to be filtered at the destination.

There is a tradeoff between the number of duplicate and extra messages as the grid cell size changes. Smaller grid cells will generally result in fewer extra messages, but more duplicates, and vice versa.

5.1.3 Region-Based Groups with Grids

DDM-Kit uses a variation on the region-based approach that uses grid cells to reduce matching overhead. A multicast group is defined for each update region, eliminating the duplicate and extra message problem of the grid scheme. However, grid partitioning is used to match update and subscription regions, improving the scalability of the pure update-region based approach.

Grids can be used to improve the efficiency of region changes. Logically, when a subscription region changes, one need only consider those update regions overlapping the grid cells covering the old and new subscription regions to determine the new composition of multicast groups. Similarly, when an update region changes, one need only consider those subscription regions that overlap the grid cells of the old/new update region to determine the new composition of the group.

DDM-Kit uses a variation on this approach to manage group membership. Recall the pure grid-based approach used subscription counts to track the number of times a federate is subscribed to a grid cell. DDM-Kit uses a similar concept, but for update regions, to trigger group join and leave requests. Specifically, a *subscription strength* array is defined for each update region, with one entry per federate. The entry for a federate indicates the "strength" of that federate's subscription to the update region (group). One unit of strength corresponds to one subscription region for the federate overlapping with the update region in exactly one grid cell. The strength of a subscription region is the number of grid cells in which the subscription region overlaps with the update region. The total strength of the federate's subscription to an update region is the sum of the strengths of each of the federate's subscription regions. For example, if the federate has two subscription regions, and one overlaps the update region in one cell, and the second overlaps it in two cells, the strength of the federate's

subscription to the update region is three. The federate remains joined to the update region's multicast group so long as it has a subscription strength of at least one. The DDM-Kit software keeps the strength arrays updated as regions come and go and are modified. It issues a join request if the federate's subscription strength becomes non-zero, and issues a leave request if the strength becomes zero. This approach is easily extended to consider classes and attributes, as required in the HLA DDM services.

Finally, the various data structures that are required to implement DDM may be centralized, or distributed among the processors participating in the federation execution [15]. Further, the data structures may be replicated to enable fast lookup, at the expense of additional communication to keep the multiple copies consistent. The current implementation of DDM-Kit uses a replicated copy of the data structures in each processor. Alternate implementation approaches are under investigation.

5.2 Time Managed DDM

The HLA DDM services are defined to operate independent of the time management services. In particular, changes to subscriptions and update regions are not synchronized with logical time. DDM-Kit does provide support for time managed DDM, however.

Without time managed DDM, missed and/or extra messages may occur:

- *Missed messages.* If a federate is added to a multicast group at logical time T after an update with a time stamp greater than T has been sent to the group, the federate will not receive a message it should have received.
- *Extra messages.* If a federate leaves a group at logical time T after an update with a time stamp greater than T was sent to the group, the federate will receive a message that it had not expected to receive.

This problem is discussed in detail in [17]. Briefly, one solution to this problem is to provide a message log to avoid missed messages. Updates are logged as they are issued. When a change in group membership indicates that a previously issued update should have been sent to a federate but in fact was not, an update is retrieved from the log and sent. On the other hand, extra messages can be avoided by performing extra filtering by the federates receiving the updates.

6. RTI Implementations

This section explains the implementation of an RTI using RTI-Kit. A specific example is given, based on the HLA IFSpec definition of the Time Advance Request service.

6.1 Basic RTI Functionality

As shown in Figure 1, an RTI implementation can be thought of as an interface to RTI-Kit functionality. An RTI implementation presents services to the federate according to a specific paradigm for simulation execution management and exchange of data. Each RTI implementation must manage whatever global and local state information is required for its paradigm.

Typically, an RTI will have state variables which include time management information (such as local time, result of the most recent LBTS computation, lookahead, the state of any federate requests to advance time), communication information (such as the multicast groups, and group membership, and the mapping of groups to message types, as mentioned in section 5) and the state of execution management processes (such as pause/resume, save/restore, join/resign). The RTI must also have a means for delivering messages and other information to the federate. In the case of an HLA federate, this is done using callback functions. Therefore, the RTI must have a means of registering callback functions.

6.2 TAR Implementation

As an example, let us explore how one might implement the Time Advance Request (TAR) function using RTI-Kit primitives. A federate invokes TAR when it is ready to 1) receive messages up to a specific time, and 2) advance its clock to that time. The expected behavior of the RTI is to deliver messages up to the requested time, and issue a Time Advance Grant (TAG) when no more messages with timestamps less than or equal to the requested time will be delivered. As with other current HLA RTI implementations we will expect the federate to use a “tick()” method to pass control to the RTI. It is in tick() that federate callbacks are issued.

Upon receiving a TAR invocation, the RTI records that a TAR is pending and notes the requested time. Then the RTI computes the local minimum timestamp (by adding the requested time to the lookahead), and initiates an LBTS computation (**TM_StartLBTS**) specifying that time value. In initiating the LBTS computation, the RTI also indicates the routine to be executed when the LBTS computation is complete (**LBTS_Done**). After the LBTS computation has been started, the RTI returns from the TAR method. Other federates’ RTI implementations will receive the LBTS start-up message, and have an **LBTS_Started** callback invoked. This is the first step in the TAR process,

where all RTI instances have calculated a local minimum timestamp, and are participating in an LBTS computation.

Typically, once a federate invokes TAR, it will tick() the RTI until a TAG is issued. While the federate is waiting for LBTS to be advanced to the requested time, receive-order and "safe" timestamp-order messages can be delivered. Message delivery is conducted as follows. Each time the federate invokes tick(), the RTI-Kit modules, including TM-Kit, must be "ticked." This allows the messages to be pulled off the wire, and permits the continued processing of LBTS computations. Each message is dispatched to its appropriate handler. RTI-Kit provides efficient FIFO and heap implementations for buffering receive-order and timestamp order messages. After the RTI-Kit has been ticked, the messages on the FIFO queue can be delivered. If an LBTS computation was completed during **TM-Tick**, the **LBTS_Done** callback will pass the new value of LBTS. If LBTS is greater than the timestamps of any messages in the TSO heap, then those messages can also be delivered, in order. Once the messages have been delivered, the tick() call returns control to the federate. Message delivery, from within the tick() call, is the second step in the TAR process.

The federate will continue to tick the RTI, until the value of LBTS is greater than the requested time. At this point (after delivering the pending messages) the RTI will update the local time, note that a TAR is no longer pending, and invoke the TAG callback. This completes the TAR process.

Of course, the TAR process is one common method for advancing time in a conservative simulation. Because many RTIs use similar paradigms for advancing federate time, RTI-Kit includes a module called RTI-Core which simplifies RTI implementation. The RTI-Core module provides basic sets of services for dealing with conservative and optimistic time management interfaces, as well as event retraction.

6.3 Exploring DesignTrade-offs

One important feature of a modular RTI design is the ability to explore design trade-offs. The overhead of a particular interface design may lead one to choose a modified, or partial implementation. This may produce a more efficient execution for the target federation. This is a reasonable trade-off, even in an HLA execution environment, considering that freely available compliant RTIs exist, and the principle reason for choosing a different implementation would either be for 1) performance or 2) federation specific architectural considerations.

An example of this type of trade-off is evident when considering the flexibility in configuring object attribute updates. The HLA IF specification allows for the ownership, transport and ordering of every attribute of every object to be individually set. While this could be a powerful tool for customizing the communications configuration of a federation execution, there is a significant overhead associated with checking each attribute in an attribute handle-value pair set (AHVPS). In federations where ownership is static, and transport is never altered from the default, a significant simplification is possible. This fact was exploited in the design of an RTI-Kit-based AHVPS class. The design assumes that a new AHVPS (or Parameter HVPS) will eventually be sent as an object attribute update or an interaction message. The AHVPS constructor allocates memory for the entire message, marshalling the AHVPS data into the appropriate slot. This eliminates the need to copy any data during an UpdateObjectAttributeValue() or SendInteraction() call. Such an implementation would not be efficient if attribute updates cannot be assumed to be atomic.

7. Conclusion

RTI-Kit provides a software base for research and development of distributed simulation systems. Although it was designed with the High Level Architecture in mind, the software is applicable to many other classes of parallel and/or distributed simulation systems. The modular design approach makes RTI-Kit will suited for experimental research in federated simulation systems.

RTI-Kit is currently distributed as part of the Federated Distributed Simulation Tool Kit (FDK) package. It is being used in a variety of educational and research projects such as research in DDM, use of high bandwidth and active networks for distributed simulations, and federated simulations for modeling telecommunication networks.

8. References

1. Kuhl, F., R. Weatherly, and J. Dahmann, *Creating Computer Simulation Systems: An Introduction to the High Level Architecture for Simulation*. 1999: Prentice Hall.
2. Defense Modeling and Simulation Office, *High Level Architecture Interface Specification, Version 1.3*, . 1998: Washington D.C.
3. Fujimoto, R.M. and P. Hoare, *HLA RTI Performance in High Speed LAN Environments*, in *Proceedings of the Fall Simulation Interoperability Workshop*. 1998: Orlando, FL.
4. Ferenci, S. and R.M. Fujimoto, *RTI Performance on Shared Memory and Message Passing Architectures*, in

Proceedings of the 1999 Sprin Simulation Interoperability Workshop. 1999: Orlando, FL.

5. Boswell, S.B., *et al.*, *Communication Experiments with RTI 1.3*, . 1999, MIT Lincol Laboratory: Lexington, MA.
6. Christensen, P.J., D.J. Van Hook, and M.W. H, *HLA RTI Shared Memory Communication*, in *Proceedings of the 1999 Spring Simulation Ineroperability Workshop*. 1999: Orlando, FL. p. Paper 99S-SIW-090.
7. Steinman, J.S., *et al.*, *Design of the HPC-RTI for the High Level Architecture*, in *Proceedings of the Fall Simulation Interoperability Workshop*. 1999: Orlando, FL. p. Paper 99F-SIW-071.
8. Hoare, P., G. Magee, and I. Moody, *The Development of a Prototype HLA Runtime Infrastructure (RTI-Lite) Using CORBA*, in *Proceedings of the 1997 Summer Computer Simulation Conference*. 1997. p. 573-578.
9. Boden, N., *et al.*, *Myrinet: A Gigabit Per Second Local Area Network.* IEEE Micro, 1995. **15**(1): p. 29-36.
10. Pakin, S., *et al.*, *Fast Message (FM) 2.0 Users Documentation*, . 1997, Department of Computer Science, University of Illinois: Urbana, IL.
11. Ferenci, S.L., K.S. Perumalla, and R.M. Fujimoto, *An Approach for Federating Parallel Simulators*, in *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*. 2000, IEEE Computer Society. p. 63-70.
12. Fujimoto, R.M., *Time Management in the High Level Architecture.* Simulation, 1998. **71**(6): p. 388-400.
13. Fujimoto, R.M., *Parallel and Distributed Simulation Systems*. 2000: Wiley Interscience.
14. Srinivasan, S., *et al.*, *Implementation of Reductions in Support of PDES on a Network of Workstation*, in *Proceedins of the 12th Workshop on Parallel and Distributed Simulation*. 1998. p. 116-123.
15. Van Hook, D.J. and J.O. Calvin, *Data Distribution Management in RTI 1.3*, in *Proceedings of the Spring Simulation Interoperability Workshop*. 1998: Orlando, FL. p. paper 98S-SIW-206.
16. Van Hook, D.J., S.J. Rak, and J.O. Calvin, *Approaches to Relevance Filtering*, in *Proceedings of the 11th DIS Workshop on Standards for the Interoperability of Distributed Simulations*. 1994: Orlando, FL.
17. Tacic, I. and R.M. Fujimoto, *Synchronized Data Distribution Management in Distributed Simulations*, in *Proceedings of the Workshop on Parallel and Distributed Simulation*. 1998.