Rationale and Design of the MÄK Real-Time RTI

Douglas D. Wood
Len Granowetter
MÄK Technologies
185 Alewife Brook Parkway
Cambridge, MA 02138
617-876-8085
dwood@MAK.com, lengrano@MAK.com

Keywords:
HLA, RTI

ABSTRACT: *The development of the MÄK Real-Time RTI primarily came about in response to the difficulties of working with the existing RTI implementations in a development environment. These difficulties were evident both internally with our own development as we transitioned our COTS products to HLA and externally as our customers sought support in their transition efforts. In addition, the real-time virtual simulation community's concern for RTI performance indicated the need for an RTI that optimized the basic requirements of real-time simulation. The development of the MÄK Real-Time RTI focused on the subset of the HLA Interface Specification that meet those requirements.*

The design of the MÄK Real-Time RTI is based on simplicity and efficiency. At the same time, it does not neglect the use of data abstractions that promote extension and adaptation (essential for continued development of the remaining services). It also minimizes the amount of handshaking and synchronization that occurs between the RTI components. In fact, the MÄK Real-Time RTI supports a configuration that does not require a centralized RTI Executive process. This configuration is especially useful in a development and debugging environment.

In addition to addressing the rationale and design, this paper will discuss some of the implementation, operation, and performance issues and point out areas of future work.

1 Introduction

The MAK Real-Time RTI implements a subset of the HLA Interface Specification that supports the basic requirements of the real-time virtual simulation environment. The implemented services along with the associated support services are: Federation Management (including synchronization points; excluding save/restore), Declaration Management, Object Management, and Ownership Management. Both the best effort and reliable transport types are supported, but only receive order is supported. At this time, Data Distribution Management (DDM), Time Management, and the Management Object Model services are not supported.

This paper discusses the motivation for the development of the MAK RTI, the basic components in its design, and a few pertinent issues.

2 Rationale

The motivation for the development of the MÄK Real-Time RTI came from external as well as internal factors. MÄK was an established vendor of COTS products for the DIS community. It was imperative that MÄK become an early adopter of HLA if its products were going to be viable as the community made the transition from DIS to HLA. As we worked to develop the HLA versions of our products, we learned that the existing RTIs were not particularly well suited to use in the development and debugging phases of a project. Certain aspects involving the operation of the RTI that were minor or secondary in the context of a "real" exercise significantly impacted productivity in a development environment. In addition to these factors, we knew that the performance of the RTI was a major concern within the real-time simulation community. We saw a need for an RTI that focused on optimizing the basic requirements of the real-time virtual simulation environment.

2.1 Facilitate Development And Debugging

During development and debugging, the typical "federation execution" consists of just one or two federates, and runs for only a few seconds - just long enough to see whether your last feature addition or bug fix works - before one changes some more code and iterates through again. The developer is constantly restarting federates and federation executions, and federates under development often exit unexpectedly or crash. We found that we were spending a lot of our development time dealing with RTI operation issues such as:

- starting and stopping the rtiexec (centralized RTI process for creating and destroying federations)
- changing RTI Initialization Data (RID) files
- starting and stopping the fedexec (centralized process for managing a federation)
- waiting for the fedexec window to come up
- waiting for the RTI to complete its handshaking during joins and resigns
- manually removing dead federates

These issues became a significant hindrance to productivity during the development of the new versions of our products under HLA. Based on this early experience, we saw the need for a low-overhead, easy-to-use RTI that could be used during the development, debugging, and support of the new HLA versions of our tools.

2.2 Provide RTI Support

MÄK's customers rely on us to handle the details of HLA communications. We are often the primary source of support when they encounter connectivity problems as they first start to run their applications under HLA. Due to the black-box nature of existing RTIs at the time, we often found it difficult to help them trace the source of their problems. We saw a need for an RTI that allowed users to gain some more insight into what was happening inside of the RTI, for example, by allowing them to inspect the data messages that were being exchanged by Local RTI Components (LRCs).

Along the same lines, some of our customers were concerned about how we were going to help them make the transition to HLA when their simulations were built on platforms not supported by existing RTIs. If there was no RTI available to them, their applications might need to be retired, meaning a loss of customers for us. Therefore, we saw the need to develop a very portable RTI.

2.3 Improve RTI Performance

Performance concerns were a factor in many potential users delaying their HLA transition. We believed that we could speed community acceptance of HLA, and help drive a smooth transition, by insuring that a real-time RTI was available. After all, more HLA users meant more potential customers of our HLA tools. Existing RTIs were developed under a very comprehensive set of requirements. They needed to support a wide variety of simulation domains - not just the real-time community - and there was a great emphasis on the daunting task of getting all of the different RTI service groups to work together correctly. We believed that by focusing on the subset of services that were needed by the real-time simulation community, and by making performance of these services a primary concern, we could develop an RTI that was lean and efficient.

3 Design

The MÄK RTI is constructed from relatively basic and direct components. The infrastructure supports the required functionality with as little overhead and indirection as possible. It also minimizes the amount of handshaking and synchronization that has to happen among LRCs, particularly during startup and shutdown. At the same time, the design does not neglect the use of data abstractions that promote extension and adaptation. This latter point is essential, as continued development must be able to incorporate the remaining RTI services into the existing infrastructure as well as improve, diversify, and extend the implementation of the existing components.

3.1 General Architecture

On the surface, the architecture is composed of the familiar local RTI component (LRC), an RTI Executive, and one or more Federation Executives (see Figure 1). However, unlike other implementations, the RTI Executive and Federation Executive are optional parts of the MÄK RTI – unnecessary in a lightweight mode designed to support the development and debugging stages of an HLA project. In this lightweight mode, all communication between federates is peer-to-peer.

The LRC is embedded within each federate and performs its functions within the federate process space. The LRC handles the processing of the RTI Ambassador service calls from the federate and the manufacturing of Federate Ambassador calls in reaction to other federates and the RTI itself.

The RTI Executive and Federation Executive, if they are used, exist in a separate process. The RTI Executive handles any centralized processing required across federations. The RTI Executive creates one or more Federation Executive objects that handle any centralized processing within a federation. The Federation Executives are not separate processes; they are C++ objects that exist within the RTI Executive process.

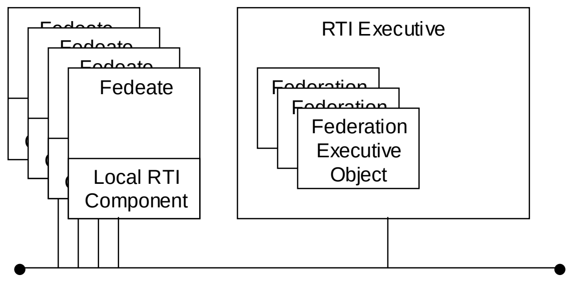


Figure 1 RTI Major Components

3.2 Local RTI Component

The core infrastructure of the LRC is composed of just a few categories of components. The **exercise connections** handle the interprocess communication. The **service managers** handle the processing of RTI service calls within the LRC. The **data messages** package the RTI service call arguments for transmission between RTI components. Several more support components are used in the construction of these core components. Of course, there are implementations for the RTI Ambassador and the various RTI basic data structures defined by the API (e.g., attribute handle set, attribute handle value pair set, etc.).

Exercise connections provide the interprocess communication for the RTI. The exercise connection class provides a layer of abstraction that hides the details of how interprocess communication is implemented, giving us the flexibility to change implementations without affecting a lot of code. The current implementation of best effort and reliable connections are built using sockets that support UDP and TCP connections respectively, however the architecture will allow us to adapt to shared memory or other implementations easily.

The RTI sends and receives data using three exercise connection references: an *internal connection* - where RTI bookkeeping messages like subscribe, join, and synchronization point messages are sent, a *best effort connection* – where best effort FOM data is sent, and a *reliable connection* – where reliable FOM data is sent. These three references map to just two and possibly only one actual exercise connection object. A typical configuration used during debugging maps all three references to a single UDP exercise connection. The internal and reliable references can be mapped to a TCP exercise connection when support for the reliable transport mode is required.

Each of the RTI service calls are handled by one of three service manager objects, a connection manager, an interaction manager, and an object manager. The RTI performs practically all of its processing within one of these three objects. The connection manager handles the federation management service calls. The declaration and object management service calls are divided between the interaction and object managers. The object manager handles the ownership management service calls.

The object manager maintains a list of objects and associated state information for each object that is registered and discovered by the federate. The objects and their state information support the processing of object management and ownership management services. The objects maintain information such as the actual class of the object versus the class discovered to the federate, the state of attribute acquisitions and divestitures, the set of owned attributes, etc. It also encapsulates some of the object management and ownership management processing, taking the implementation to where the state information is located. The objects do not maintain the attribute values or any simulation state data.

Data messages package the necessary RTI service call arguments for transmission to the other RTI components. The data messages define accessor and mutator methods in order to set and inspect the individual fields; although, a constructor is defined that will fill out the message in one step as it is created. The message classes provide methods for serializing their data (and initializing from serialized data) to facilitate transmission to the LRCs and RTI Executive through various communication mechanisms. Byte-swapping of RTI bookkeeping information like object IDs, attribute handles, etc. takes place here, so that a receiving LRC on a different platform can correctly interpret it. Data marshalling of FOM data is the federate's responsibility, and it is sent as-is.

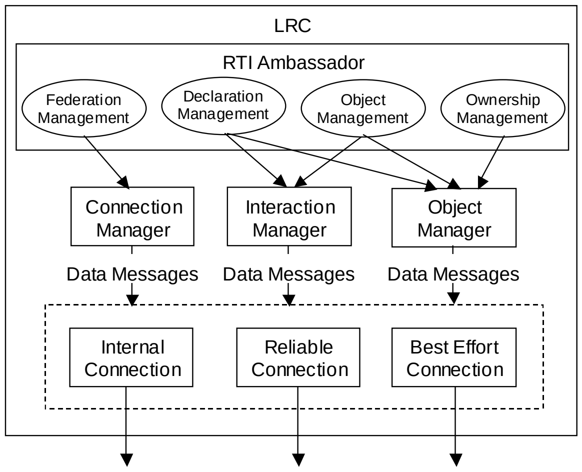


Figure 2 LRC RTI Ambassador Processing

3.3 LRC Processing

The general purpose of an LRC is to process the RTI Ambassador service calls invoked by the federate and to assemble Federate Ambassador service calls to be invoked on the federate. In processing RTI ambassador service calls, the service managers are consumers of RTI service call arguments and producers of data messages. The data messages are transmitted to external RTI components via the exercise connections. The execution path through these objects is depicted in Figure 2. In the reverse direction, the service managers are consumers of data messages and producers of service call arguments. The data messages come from external RTI components through the exercise connections. These data messages are processed by the service managers to produce the service call arguments that are then passed to the federate via the Federate Ambassador. Therefore, the depiction of the execution path is nearly the same only the flow of data is reversed and the Federate Ambassador lies outside of the LRC. However, the mechanism for carrying out the execution is a little more complex in that it occurs because of the federate invoking the RTI Ambassador tick() method.

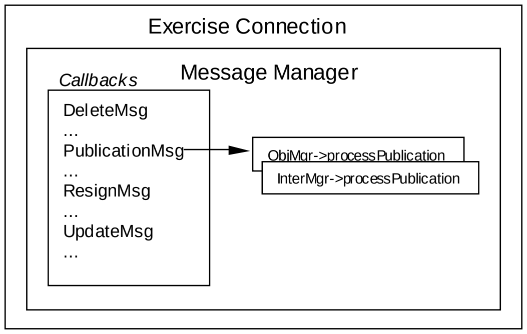


Figure 3 Exercise Connection Callbacks

The processing of incoming messages by the LRC is carried out through an internal callback mechanism (see Figure 3). The service managers register callbacks with the exercise connections. During the RTI Ambassador tick(), the connections receive external messages to process. The connections, through a *message manager*, determine the message kind and invoke the registered callback functions passing to them the message and the manager that registered the function. The callback function processes the message and if necessary invokes the appropriate Federate Ambassador service call.

In addition to processing the RTI services directly, the RTI Ambassador tick method invokes a tick method on both the object and interaction service managers. These service manager tick methods handle the creation and transmission of advisory messages as a side effect of any previous declaration and object management service calls, but there is no other bookkeeping or other processing necessary here.

3.4 RTI Executive

Unlike the LRC, the RTI Executive is not a required component of an RTI implementation. It is possible to implement an RTI where all of its processing occurs within the LRCs. However, certain services of the HLA specification require some kind of coordinated control or arbitration. While it is possible to distribute these global tasks across the LRCs, such schemes are in general either less efficient, much more difficult to implement, or both. For example, one approach could be to give the responsibility for some global task (like generating unique federate IDs) to an LRC of a particular federate (i.e., the federate creating the federation). However, this task would have to be seamlessly transferred to another LRC if the responsible federate resigns. The overhead and complexity involved in this kind of approach was considered overly burdensome relative to the added benefit. The familiar centralized approach was followed instead.

The RTI Executive process manages the creation and destruction of federations. A Federation Executive object is created for each federation. Each Federation Executive object manages requests for federation joins, resigns, and synchronization points. The RTI Executive uses the same type of exercise connections as the LRCs and these are shared with each of the Federation Executives. The executive process executes a simple infinite loop that waits for and processes data messages received by the exercise connections. The processing of each data message (via callbacks) usually results in the transmission of one or more data messages in response.

3.5 Reliable Transport

The current implementation of reliable transport uses a TCP forwarder, which receives reliable traffic from each federate, and forwards it to other federates. This approach was chosen over alternatives for simplicity. It allows all necessary connections to be set up with a minimum of handshaking, and it means that the RTI can easily recover from federate failures. Each LRC reads the IP address of the TCP forwarder from the RID file, and creates just a single TCP connection to the forwarder. LRCs never need to know the IP addresses of the other federates, and never get socket errors when a remote federate dies unexpectedly.

In the current implementation, the TCP forwarder is a part of the RTI Executive (as a result support for reliable transport requires the use of an RTI Executive). However, the TCP Forwarder is a self-contained class with no knowledge of other RTI Executive classes. The RTI Executive uses the same type of exercise connections as the LRCs. Therefore, it connects to the TCP forwarder in exactly the same way as the LRCs – through a TCP connection – even though the forwarder is part of the RTI Executive process. This decoupling of the forwarder functionality from other RTI Executive functionality leaves open the possibility of separating the forwarding functionality into its own process (or replacing the forwarding approach altogether).

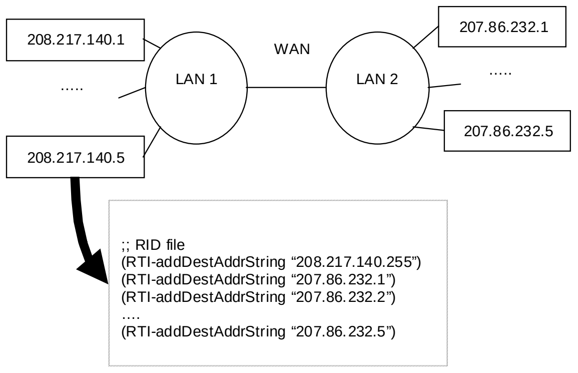
The TCP forwarder approach for reliable transport involves more latency than an approach where each federate sends directly to each of the other federates. However, it moves the computation burden of sending multiple copies of each packet out of the real-time application, and into the forwarder process that has little else to do. In typical real-time exercises, latency-critical data is sent using best-effort transport anyway.

We are currently planning to implement a more robust reliable transport scheme that will give users more flexibility to make the tradeoff decisions between latency, CPU usage, bandwidth, etc. In addition, we plan to provide a plug-in API for exercise connections (see section 6), which would allow users to explore additional communication models.

3.6 WAN Communications

Implementation of RTI communications in a wide area network (WAN) requires a method of routing the RTI messages across the local area network (LAN) boundaries. For the current reliable transport implementation using a TCP exercise connection, no additional configuration procedures are required. Each federate is given the IP address of the RTI Executive and the LRCs make a TCP connection to the TCP Forwarder in the RTI Executive, just as it would in a LAN. The TCP Forwarder uses these TCP connections to distribute the data to each federate; the underlying TCP protocol handles the routing of messages across the WAN.

Best effort communication in a WAN environment is supported using an abstraction we call a group socket exercise connection. The group socket exercise connection can be initialized to send to any number of IP addresses. When a message is sent via the group socket, it sends a copy to each of its destinations. The RID file contains an entry that designates the required destinations. A RID file is configured for each LAN such that one entry designates the broadcast (or multicast) address for the LAN and one entry for each remote federate.



```
;; RID file
(RTI-addDestAddrString "208.217.140.255")
(RTI-addDestAddrString "207.86.232.1")
(RTI-addDestAddrString "207.86.232.2")
....
(RTI-addDestAddrString "207.86.232.5")
```

Figure 4 WAN Best Effort Communications

4 RTI Sans Executive

Section 3.4 alluded to an RTI implementation without the need for a centralized process like the RTI Executive. In fact, an underlying design goal was to identify and minimize the functionality of the RTI Executive. Among the implemented services, there are surprisingly few places where this centralized processing is required. Often, the functionality involved is not critical for supporting a federation execution. This insight points to a solution where no RTI Executive is necessary. The solution is not to move the centralized processing into the LRCs, but rather to eliminate it altogether, with the caveat of relying on a few assumptions. The result is a configuration where the LRCs communicate in a truly peer-to-peer fashion.

One responsibility of the RTI Executive is that it keeps track of which federates have achieved each synchronization point. It needs this information so that it can alert all of the LRCs when the federation is synchronized. However, if the federation is not using synchronization points, this centralized processing is not required.

Similarly, an RTI Executive coordinates the task of managing federation executions to support the service calls for create, destroy, join, and resign. It must keep track of the federation executions that exist and whether they have federates currently joined. This accounting of federation executions and joined federates serves just two main purposes, other than the accounting itself.

The first purpose is to ensure that the federate receives a unique federate handle. In small federation executions, each federate LRC can randomly generate its own federate handle and face only the tiniest possibility that two federates choose the same handle.

The second purpose is to validate the preconditions of the create, destroy, join and resign service calls so that the federate LRC can throw the proper exceptions if necessary (e.g., FederationExecutionAlreadyExists). An exception is critical when it indicates that the post conditions of the service call cannot be satisfied. In many cases, the post conditions (e.g., a federation execution exists) are established whether there is an exception thrown or not. Federates in these cases are typically ignoring these exceptions anyway and failing to throw them would rarely even be noticed. The exceptions supported by the RTI Executive all fall under this category.

The support for reliable transport uses a TCP forwarder (see section 3.5). The TCP forwarder is created within the RTI Executive. However, if an RTI Executive is not used the reliable exercise connection defaults to use a UDP exercise connection implementation. Given a lightly loaded network, data sent via UDP will typically be delivered with few if any collisions.

In summary, the RTI Executive is only necessary when the federation uses synchronization points, when the RTI must specifically enforce rules about throwing exceptions that are for the most part ignored (or can be avoided), to absolutely guarantee that no two federates will choose the same federate handle, or when reliable transport must be guaranteed. Failing to guarantee these requirements means that the MÄK RTI when used without the RTI Executive is not technically compliant with the HLA interface specification. However, during development and debugging, when the vast majority of federation executions occur, this is usually not critical. Our tool developers find that they rarely need to use the RTI Executive at all.

5 Implementation, Operation And Performance Issues

5.1 Implementation Issues

The MÄK RTI supports multiple connections per process without restrictions on concurrent access. The lack of this feature in existing RTIs created major hurdles in the development of our multi-threaded applications (e.g., PVD, VR-Forces). This feature is essential if HLA is going to truly support a component-to-component communication architecture.

The MÄK RTI creates object handles that are globally unique across the federation, as the current IEEE

1516.1 specification requires. While this characteristic was not part of the HLA draft 1.3 specification and thus could not be relied on, it was useful during debugging to identify objects updates across federates without having to lookup the name. In addition, the default name provided during object registration is simply a string representation of the handle. Again, this makes it easier to identify objects when debugging.

The memory footprint of the MÄK RTI is small. The library and dll for the Windows platform is just 83kb/450kb for the MÄK RTI versus 284kb/1.8mb for DMSO RTI 1.3v7 and 2.2mb/4.5mb for the DMSO RTI NG (not including ACE 1mb and TOA 2.5mb). Granted, the MÄK RTI is not a full implementation and the remaining services may be considerably more complex. However, there are applications and platforms that exist where memory is still a premium. The MÄK RTI was ported to VxWorks in an environment that had a limited amount of memory and no virtual memory. The small footprint of the MÄK RTI made this port feasible.

5.2 DMSO RTI Compatibility

The MÄK RTI implementation was specifically tailored to be link compatible with the DMSO RTI implementations. While only a subset of the services is supported, a stub is provided for all the service calls defined in the RTI API. Federates that restrict their service calls to the subset supported by the MÄK RTI can switch between the implementations without recompiling.

5.3 Ease Of Use

The MÄK RTI has many features that make it easy to use especially in a development environment. As described in section 4, a federation can be run without an RTI Executive process. This feature is especially useful when debugging a single federate that has the capability to create its own federation. The federate can be executed independent of any other processes. Even with an RTI executive, the MÄK RTI has very quick startup and join times. The creation of the RTI Ambassador requires just a single connection to the RTI executive and no additional connections are required when the federate joins. In addition, there are no Federation Executive processes since these are just C++ objects within the RTI Executive process.

The RID file used for RTI configuration is small and can be omitted if there is no need to override the defaults. The RID and FED files are sought in the RTI configuration environment variable ($RTI_CONFIG) or they can be found in the working directory.

Because current RTI implementations are essentially black boxes, it is often difficult to determine the nature of problems involving the use of the RTI. An RTI dump utility was created that allows users to inspect the messages that the RTI sends over the network. By seeing the messages sent through the RTI, the user can determine which federates are sending what messages (including hex dump of attribute/parameter data) and when they are sent. This type of utility greatly enhances the investigation of problems involving the use of the RTI.

5.4 Latency And RTI Processing

The end-to-end latency of a typical best-effort update through the MÄK RTI is not that much more than sending a packet over Ethernet at around 1.1 – 1.5 ms. The latency for the reliable transport is approximately 2.0 ms end-to-end. These measurements were taken over our normal corporate LAN with our development machines and included processing by our VR-Link middleware.

The efficiency and low latency in the MÄK RTI can be attributed to its direct and simple design. The path from an RTI Ambassador service call to transmitting a data message over the network (and vice versa) traverses through a thin RTI infrastructure with no intervening third party libraries. The best effort communication is peer-to-peer. In addition to a minimal latency time, the MÄK RTI consumes very little processing time for any other ancillary activities. Other than directly processing the service calls, the servicing of the RTI advisories is the only other processing performed by the LRC. This low overhead minimizes the impact on the federate.

5.5 Message Format

The format for transmitting data messages between RTI components is compact. The fact that the RTI does not rely on third party tools for data distribution means that it can control the size of the messages (no extra headers, padding, etc.). All packets have a common header that is just eight bytes (see Figure 5), excluding any TCP or UDP protocol headers. Updates contain an additional fixed header of eight bytes (object id, class handle, object name length and user tag length), a variable header containing the object name and user tag, plus four bytes of overhead per attribute (handle, length). The RTI generated names are typically six bytes, but federates may supply longer names. Interaction messages have a similar format with a fixed header of four bytes, plus the user tag, plus four bytes of overhead per parameter.

A typical real-time virtual simulation update containing five separate TPSI attributes (location, orientation, velocity, acceleration, angular acceleration) has 72 bytes of data, plus approximately 66 bytes of overhead (depending on object name and user tag length). Total packet size of approximately 134 bytes (based on name and tag length) is less than the 144 bytes needed for a DIS Entity State PDU. The bandwidth efficiency of sending the same TSPI information can be improved if a single attribute is used to package the TSPI data, reducing the packet size by 20 bytes.

Packet Size	16-bit Unsigned Integer
Federate Handle	16-bit Unsigned Integer
Message Kind	Octet
Execution Name	3 Octets (1st 3 characters)

Figure 5 Data Message Header

6 Future Work

Development of the MÄK RTI will continue to add support for the remaining HLA interface specification services. The objective is to become a certified RTI implementation and certification requires a complete implementation. Ongoing development is adding support for the DDM services. Among the remaining set of services, these are the most beneficial for the real-time virtual simulation environment.

Another key feature that is under development is adding support for a plugin API. The plugin API will allow users to adapt, extend, or simply replace certain aspects of the RTI. One candidate for this feature is the exercise connection abstraction. For instance, implementing a shared memory exercise connection would allow a federation to operate via shared memory simply by reconfiguring the RTI. Creating a plugin API for DDM would allow for different implementations that take advantage of the communication infrastructure or federation topology. Encryption, compression, or other formatting techniques could be inserted by extending the existing implementation of data messages (exercise connections could also incorporate encryption or compression). Support for RTI logging could be inserted in the RTI and Federate ambassador interface. These are just a few of the possibilities.

While there was a concerted effort to optimize the efficiency and performance of the initial set of services, some improvements will be made in the current MÄK RTI implementation. For example, the TCP forwarder implementation for reliable transport could be improved as well as the support for WAN communications.

7 Conclusions

The MÄK RTI follows a simple and direct design. It satisfies the basic requirements of many real-time virtual simulations. The implementation optimizes latency and bandwidth within the constraints of the architecture. Some comparisons have been made between the MÄK RTI and the DMSO RTI. To be fair, the MÄK RTI is only a subset versus a complete implementation by the DMSO RTI. The implementation of the additional services is bound to affect the performance to some degree. However, the implementation of the MÄK RTI demonstrates that HLA can be lean and efficient. It is up to the RTI implementers to ensure that the level of performance for core services is not sacrificed by unnecessary complexity. At the very least, the introduction of additional services should impact the core services only when those services are employed.

Author Biographies

DOUGLAS D. WOOD is a Senior Software Scientist at MÄK Technologies. Mr. Wood is leading the development of an optimized HLA environment; a major part of which is enhancing and optimizing the MÄK Real-Time RTI. Over the past 12 years, he has performed research and software development in distributed simulation, including computer generated forces, electronic warfare protocols, emergency management training, and DIS-HLA translation. Mr. Wood received an MS and BS in Computer Science from the University of Central Florida. His interests are in simulation and distributed systems.

LEN GRANOWETER is the Director of Product Development at MÄK Technologies, responsible for MÄK's COTS product development group. He was the chief architect of the MÄK Real-Time RTI during its initial development. He has worked on the VR-Link HLA/DIS toolkit and other MÄK simulation and visualization products for over 7 years, serving as the lead products engineer for several years during the HLA transition. Mr. Granowetter holds a Bachelor of Science degree in Computer Science and Engineering from the Massachusetts Institute of Technology.