2003-12

The extensible run-time infrastructure (XRTI) : an experimental implementation of proposed improvements to the high level architecture

Kapolka, Andrzej

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

THE EXTENSIBLE RUN-TIME INFRASTRUCTURE (XRTI): AN EXPERIMENTAL IMPLEMENTATION OF PROPOSED IMPROVEMENTS TO THE HIGH LEVEL ARCHITECTURE

by

Andrzej Kapolka

December 2003

Thesis Advisor:	Michael Zyda
Co-Advisor:	Bret Michael

This thesis is done in cooperation with the MOVES Institute.
Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** December 2003	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: The Extensible Run-Time Infrastructure (XRTI): An Experimental Implementation of Proposed Improvements to the High Level Architecture			**5. FUNDING NUMBERS**
6. AUTHOR(S) Andrzej Kapolka			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release, distribution unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

The establishment of a large-scale network of persistent shared virtual worlds depends on the presence of a robust standard for communicating state information between the applications that host and provide access to those worlds. The High Level Architecture (HLA) can serve as the basis for such a standard, but not before several of its shortcomings are resolved. First, it must be made easier to use. Second, it must specify a standardizable message protocol. Third, it must support dynamic object model extension and composition. Finally, its authors must provide an open-source, freely redistributable run-time infrastructure.

This thesis documents the creation of the Extensible Run-Time Infrastructure (XRTI), an experimental platform that addresses the above requirements while retaining full backwards compatibility with the existing HLA standard. To increase ease-of-use, the XRTI provides a proxy compiler that generates customized sets of Java™ source files based on the contents of arbitrary Federation Object Model Document Data (FDDs). To encourage message protocol standardization, the XRTI uses a novel bootstrapping methodology to define its low-level interactions in terms of an HLA object model. The XRTI supports the dynamic composition and extension of such object models through its Reflection Object Model (ROM), and this thesis demonstrates that ability by depicting the integration of the XRTI into NPSNET-V, a dynamically extensible platform for virtual environment applications.

14. SUBJECT TERMS HIGH LEVEL ARCHITECTURE, HLA, RUN-TIME INFRASTRUCTURE, RTI, NPSNET, NETWORKED VIRTUAL ENVIRONMENTS, NETWORK PROTOCOLS, MIDDLEWARE, OPEN-SOURCE, JAVA, DYNAMIC EXTENSIBILITY, CODE GENERATION, INTEROPERABILITY, DISTRIBUTED SIMULATION			**15. NUMBER OF PAGES** 133
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

THE EXTENSIBLE RUN-TIME INFRASTRUCTURE (XRTI): AN EXPERIMENTAL IMPLEMENTATION OF PROPOSED IMPROVEMENTS TO THE HIGH LEVEL ARCHITECTURE

Andrzej Kapolka
Research Associate, Naval Postgraduate School
B.S., University of California at Santa Cruz, December 2000

Submitted in partial fulfillment of the
requirements for the degree of

**MASTER OF SCIENCE IN
MODELING, VIRTUAL ENVIRONMENTS AND SIMULATION**

from the

**NAVAL POSTGRADUATE SCHOOL
December 2003**

Author: Andrzej Kapolka

Approved by: Michael Zyda
Thesis Advisor

Bret Michael
Co-Advisor

Rudy Darken
Chairman, MOVES Curriculum Committee

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The establishment of a large-scale network of persistent shared virtual worlds depends on the presence of a robust standard for communicating state information between the applications that host and provide access to those worlds. The High Level Architecture (HLA) can serve as the basis for such a standard, but not before several of its shortcomings are resolved. First, it must be made easier to use. Second, it must specify a standardizable message protocol. Third, it must support dynamic object model extension and composition. Finally, its authors must provide an open-source, freely redistributable run-time infrastructure.

This thesis documents the creation of the Extensible Run-Time Infrastructure (XRTI), an experimental platform that addresses the above requirements while retaining full backwards compatibility with the existing HLA standard. To increase ease-of-use, the XRTI provides a proxy compiler that generates customized sets of Java™ source files based on the contents of arbitrary Federation Object Model Document Data (FDDs). To encourage message protocol standardization, the XRTI uses a novel bootstrapping methodology to define its low-level interactions in terms of an HLA object model. The XRTI supports the dynamic composition and extension of such object models through its Reflection Object Model (ROM), and this thesis demonstrates that ability by depicting the integration of the XRTI into NPSNET-V, a dynamically extensible platform for virtual environment applications.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

The author would like to acknowledge the help of his thesis advisors, Mike Zyda and Bret Michael. Particular thanks are due to Professor Zyda for giving the author the opportunity to complete his master's degree while working as a researcher and staff member at the MOVES Institute. In his time at MOVES, the author has had the pleasure of working closely with a number of dedicated and talented students: Major William D. Fischer, USA; LT James Harney, USN; LCDR Ernesto J. Salles, USN; LTJG Ekrem Serin, Turkish Navy; Major David B. Washington, USA; and LT Michael S. Wathen, USN. To these students, as well as to Don McGregor, Mike Capps, and the entire faculty and staff of the MOVES Institute, the author extends his deepest thanks. Finally, the author would like to thank his family—Gerry, Daphne, Basia, and Marek Kapolka—for their love and support.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. OVERVIEW

Virtual environment researchers, science-fiction authors, simulation developers, and video game players share a common dream: the existence of an always-on, globally connected, infinitely expandable network of virtual worlds. Ideally, this network would subsume all types of virtual environments currently available without inheriting their limitations. The worlds of such a network could be used for social interaction, but they would share neither the superficial level of interactivity nor the impermanence of three-dimensional chat worlds such as Adobe Atmosphere [Adobe 03]. They could be used for gaming, but they would not be subject to the limited scalability of first-person shooting games such as America's Army: Operations [America's Army 03], nor to the monopolistic control exercised by the hosts of massively multiplayer online role-playing games such as EverQuest [Sony 03]. The worlds could support large-scale, long-running simulations without requiring the enormous number of man-hours expended by the military simulation community in constructing and maintaining heterogeneous High Level Architecture (HLA) federations such as the one built for Millennium Challenge 2002 [USJFCOM 03]. They would form the perfect setting for virtual environment research, for they would be as robust and enduring as current research environments are brittle and short-lived.

To those who would attempt it, the establishment of such a network presents an intimidating challenge: the creation of a set of standards that would formalize certain aspects of virtual environment applications without limiting their overall functionality. These standards would have to be general enough to apply to all existing types of environments, flexible enough to accommodate new types of environments, evolvable enough to allow for extension and improvement over time, and simple and well-specified enough to encourage widespread adoption. The standards on which the World Wide Web is based—the Hyper-Text Transport Protocol (HTTP) and the Hyper-Text Markup Language (HTML)—provide an excellent example of how to achieve the kind of universality and flexibility required. As with those associated with the World Wide Web,

a generalized set of standards for virtual environments is likely to include at least two categories: formats for static data representation, like HTML, and protocols for dynamic data interchange, like HTTP. This thesis concerns the development of a standard of the second category: a mechanism for transmitting dynamic state information between virtual environment applications.

The most promising basis for such a standard is the HLA, a standardized middleware interface specifically designed for distributed simulations [IEEE 1516, Kuhl 99]. The HLA is well specified and fully generalized, but it is limited in a number of respects. First, it is difficult to use. In order to make their applications communicate with others using the HLA, developers must obtain handles for object attributes and interaction parameters, register their ability to receive certain types of interactions and to discover certain types of objects, manually encode and decode variables to and from byte arrays, track shared object state, and respond to management requests. Second, the HLA does not specify a common message protocol. This means that run-time infrastructures (RTIs: implementations of the HLA interface standard) developed by different organizations—and even different versions of the same RTI—cannot typically interoperate. Third, once a federation execution has begun, the HLA does not support modification of the object model associated with that federation. It is therefore impossible to introduce new classes of objects or interactions into an active federation. Finally, there are no open-source RTIs. Developers cannot simply download an RTI for free and modify it to meet their specific needs, nor can they easily include an RTI with their open-source applications.

This thesis, therefore, introduces the Extensible Run-Time Infrastructure (XRTI): an open-source, freely redistributable RTI with experimental extensions that address the concerns above. To increase ease-of-use, the XRTI includes a proxy compiler that generates sets of Java™ source files based on the contents of arbitrary FOM Document Data (FDDs). Developers may use these autogenerated proxy classes as an intuitive, type-safe means to interact with the federation. To encourage message protocol standardization, the XRTI uses a bootstrapping methodology to define its low-level interactions in terms of an HLA object model. The encoding schemes specified by the

HLA standard provide the foundation for the Bootstrap Object Model (BOM), which defines the XRTI's message protocol much as the Management Object Model (MOM) describes the HLA's service interfaces. Likewise, the XRTI's Reflection Object Model (ROM) represents the object model of an active federation. By manipulating the ROM, federates may introduce new object and interaction classes without interrupting the federation execution. This thesis demonstrates that ability by incorporating support for XRTI networking into NPSNET-V, a component-based virtual environment platform [Capps 00]. Once the integration is complete, this thesis compares the performance of the XRTI to that of two widely available closed-source RTIs in the context of a typical NPSNET-V application.

B. BACKGROUND

NPSNET-V is a dynamically extensible platform for shared virtual worlds. Applications hosted within the NPSNET-V framework consist of hierarchies of dynamically loaded, loosely coupled modules rooted at an invariant microkernel [Kapolka 02]. Modules may be added to or removed from the framework at any time, and loaded modules may be hot-swapped—that is, seamlessly upgraded or otherwise replaced with new modules. The NPSNET-V distribution includes modules that provide system-level functionality, such as resource management, as well as modules specific to virtual environment applications, such as modules that represent virtual entities. NPSNET-V's entity model is based on the Model-View-Controller design pattern, which requires that the model, or the abstract state of an entity, be separated from its views, the modules that present the entity's state to the user, and from its controllers, the modules that manipulate the entity's state. View modules included with NPSNET-V include those that depict entity state textually and those that represent entities graphically in two or three dimensions. NPSNET-V's controller modules support the modification of entity state in response to user input, according to simulated physical behavior, and in reflection of updates transmitted by other applications over the network.

The extensible nature of NPSNET-V incurs a unique requirement upon the network controller modules that transfer entity state between applications. Because new types of entities may be loaded at any time and because those entities may require the

transfer of new types of state data, the communications mechanisms used by NPSNET-V must themselves support dynamic extension. The first version of NPSNET-V used a set of composable micro-protocols to achieve this ability. Each micro-protocol module transmitted or interpreted a single entity state element: for example, an entity's position or its animation parameters. Creating and maintaining these modules required a significant amount of developer effort, however, and the micro-protocol strategy, which was tightly integrated into the NPSNET-V kernel, prevented NPSNET-V from networking with applications that used standards such as the Distributed Interactive Simulation (DIS) protocol [IEEE 1278]. The next version of NPSNET-V included a set of DIS controller modules in order to allow NPSNET-V to interoperate with preexisting applications. Unfortunately, the applicability of the DIS protocol is limited to a narrow application domain—that of platform-level military simulations—and while the DIS protocol does support limited extensibility, the extension mechanism is primitive and awkward. Thesis student Ekrem Serin integrated support for his Cross-Format Schema Protocol (XFSP) [Serin 03] into NPSNET-V, allowing developers to define custom binary protocols using XML Schema, but because of the XFSP support library's inefficient approach to data representation, manipulation, and encoding, use of the XFSP significantly reduced the performance of the applications in which it was tested. As a demonstration of NPSNET-V's versatility, its developers also created rudimentary HLA controller modules that relied on the Real-Time Platform Reference Federation Object Model (RPR-FOM), a federation object model (FOM) that describes an ontology equivalent to that of the DIS protocol. As its name suggests, the RPR-FOM serves as a reference model for platform-level simulations that developers may extend as needed. Unfortunately for the purposes of NPSNET-V, however, this extension may not occur while the simulation is running.

C. PROBLEM STATEMENT

At present, there exists no communications mechanism for networked virtual environments that is generalizable enough to support any kind of environment; usable enough to encourage widespread adoption; standardizable enough to allow universal interoperability, or the ability to facilitate communication between any group of virtual

environment applications; and adaptable enough to permit run-time extension of its ontology. The HLA, while fully generalized, is difficult to use, provides no standardized message protocol, and does not permit modification of its object models during the course of a federation execution. Also, there are currently no open-source HLA RTIs available for widespread use.

D. PREVIOUS WORK

In the brief history of networked virtual environments, developers have used many different techniques to share state information between applications. These techniques can be divided into four categories: heavyweight fixed protocols, composable micro-protocols, generic protocols, and middleware solutions.

1. Heavyweight Fixed Protocols

The most basic means of sharing state information involves the definition and use of a fixed network protocol that provides explicit, byte-by-byte descriptions of every type of message that may be exchanged between participating applications. The Multicast VRML Interchange Protocol (MVIP), for instance, is a simple fixed protocol that defines eight types of messages [Robinson 00]. MVIP is specifically tailored to the requirements of applications based on Internet Protocol (IP) multicast technology and the Virtual Reality Modeling Language (VRML). Similarly, the DIS protocol is designed for use in platform-level military simulations: combat scenarios involving warships, tanks, and fighter planes. In general, the usefulness of heavyweight fixed protocols is limited to the application domains for which they were designed.

2. Composable Micro-Protocols

When a single fixed protocol does not address all of an application's communication requirements, the developers of the application may choose to use multiple protocols simultaneously. The composable micro-protocol strategy takes this approach to its logical extreme: the usage of a unique protocol for each separable aspect of communication. Such micro-protocols may be combined in stack or graph arrangements in order to allow application developers to define complex messaging behavior in terms of simple building blocks. For example, the output of a protocol

module that generates packets for transmission may be connected to the input of a filter module that compresses the packets. The output of that module may in turn be connected to the input of a filter that encrypts the compressed packets, and finally to an endpoint module that transmits the compressed, encrypted packets to a multicast channel. As demonstrated by the TreacleWell system [Oliveira 02], the composable micro-protocol strategy allows dynamically extensible applications to extend their communication languages by inserting new modules into their micro-protocol frameworks. This form of extensibility is difficult to standardize, however, as doing so would depend not only on the standardization of the micro-protocols themselves, but also on the manner by which they were combined and configured.

3. Generic Protocols

An alternate approach to achieving extensible communication is the usage of generic protocols, or meta-protocols, that allow developers to specify protocol syntax at run-time. The Dynamic Behavior Protocol (DBP), for instance, relies on documents conforming to an ad-hoc Extensible Markup Language (XML) format to describe packets in terms of a number of named, typed fields [Fischer 01]. The Cross-Format Schema Protocol (XFSP) provides a significantly improved encoding mechanism based on XML Schema [Serin 03]. Using either DBP or XFSP, applications may effect protocol extension by modifying and redistributing the protocol document. Supporting in-band protocol modifications requires the definition of a management layer to supplement the basic protocol encodings. If established, standards for encoding algorithms and management functionality would likely be complex, preventing most application developers from implementing them from scratch.

4. Middleware Solutions

Middleware solutions provide application programming interface (API) level standards for communication. Recognizing that any sufficiently complicated networking mechanism is best provided in the form of a third-party library, middleware standards ensure that applications may interact with such libraries interchangeably. The Common

Object Request Broker Architecture (CORBA) [Bolton 02] and the HLA are two middleware standards particularly relevant to networked virtual environments.

a. CORBA

CORBA provides applications with the ability to invoke object methods across network boundaries in a language and platform independent manner. Developers use Interface Definition Language (IDL) documents to describe the publicly accessible methods of CORBA-aware objects. From these documents, IDL compilers generate stub and skeleton source files that map native language features to Object Request Broker (ORB) interactions. ORBs—implementations of the CORBA interface standard—use the Internet Inter-ORB Protocol (IIOP) to send and receive encoded requests and responses over the Internet. In addition to IDL and IIOP, the CORBA standard formalizes a number of services, such as a naming service and an event service, that simplify the development of CORBA-based distributed systems.

Researchers have successfully developed virtual environment systems based on CORBA [Louis Dit Picard 01, Wilson 01], although traditional CORBA is not an ideal technology for that purpose. Traditional CORBA interactions are based on a connection-oriented request-response model: a single application locates a single object at a single remote host, transmits an encoded method call, and waits for a response containing the return value. Real-time and multicast CORBA improve on that model by supporting the connectionless publish-subscribe paradigm favored by developers of networked virtual environments and distributed simulations.

b. HLA

The HLA is a middleware solution that is specifically designed to meet the needs of the distributed simulation community. It is based on a publish-subscribe model: network participants, or federates, send all messages to the federation, or network, at large, and receive only the messages specified by their subscription parameters. There are two basic types of messages: interactions and object attribute updates. Interactions are roughly equivalent to global method calls; they contain sets of named, typed parameters, and are not associated with objects. Object attribute updates convey changed

state. The HLA allows networked applications to define FOMs that represent the nature of their shared state in terms of a number of object classes with named, typed attributes. Each attribute may be owned by only one federate at any given time, and only the owner of an attribute may update its value. The RTI is the middleware implementation. To ensure consistency, RTIs must support the federate interface described in the HLA standard [IEEE 1516.1].

The RTI interface is thorough, but it is not easy for application developers to use. In order to send an object attribute update, for instance, an application must first obtain a handle to the object, then obtain a handle to the attribute, then encode the attribute value into a byte array, then create a mapping object and use it to map the attribute handle to the encoded value, and finally invoke the RTI's `updateAttributeValues` method with the object handle and the mapping object as parameters. Many developers have attempted to increase the HLA's ease-of-use, either by creating proxy compilers that generate source code based on the contents of FDDs [Cox 98, Hunt 99] or by building object-oriented frameworks on top of the RTI [Cazard 02, Dumond 01], but these approaches have not been standardized.

Another shortcoming of the HLA standard is its lack of a specification for RTI interoperability. RTIs provided by different vendors, and even different versions of the same RTI, cannot typically exchange information with one another. By preventing simulation developers from easily connecting their simulations to others over the Internet, this leads to fragmentation of the HLA community. The most commonly recommended solution to this problem is the establishment of a common message protocol, and indeed researchers have designed such protocols and recommended them for standardization [Mullally 03, Myjak 99]. Others, however, have objected that the requirement to use such a protocol would severely limit the freedom of implementation currently enjoyed by RTI developers [Granowetter 03]. Another approach to RTI interoperability requires the use of bridges: applications that connect to two or more RTIs and translate messages between them. Enabling interoperation between N RTIs requires $O(N^2)$ bridges in the worst case (creating a bridge from every RTI to every other RTI) or O(N) bridges in the best case (creating N front-ends and N back-ends to an intermediate bridge). This

requirement, along with the extra effort required on the part of the end user to find and install the necessary bridges, clearly eliminates HLA-with-bridging as a candidate technology for allowing universal interoperability between virtual environment applications.

Researchers have successfully used the HLA in networked virtual environment applications, but not without some difficulty [Blümel 02, Brassé 00]. Developers have even integrated HLA support into a dynamically extensible virtual environment framework, but not in such a way as to allow dynamically extensible messaging [Liles 98]. The problem is that once a federation execution is in progress, the HLA does not allow federates to modify the FOM in order to introduce new object or interaction classes. For applications based on platforms such as NPSNET-V, where newly loaded software modules may require changes to the communication ontology, this restriction prevents the HLA from acting as a complete solution to the state transference problem.

Aside from its technical limitations, one of the principal impediments to widespread acceptance of the HLA is the lack of a free, open-source RTI [Givens 00]. Military simulation centers may not balk at the prospect of licensing their RTIs from commercial companies, but virtual environment researchers and game developers tend to prefer software that they can freely examine, modify, and redistribute. If an open-source RTI were available, it could serve as a reference implementation of the standard and provide a test bed for experimental extensions and enhancements.

E. OBJECTIVES

The objectives of the XRTI project are to create and demonstrate a set of extensions to the HLA standard that resolve several of its shortcomings; to provide NPSNET-V with a middleware-based networking solution that is easy-to-use, standardizable, and amenable to dynamic extension; and to deliver a freely redistributable open-source RTI to the modeling and simulation community.

F. SCOPE

The thesis project encompasses the design and implementation of the XRTI, its integration into NPSNET-V, and a series of tests comparing its performance to that of two closed-source RTIs within the context of a typical NPSNET-V application. In order to limit the complexity of the initial version of the XRTI, the author has narrowed the thesis scope by omitting support for several HLA services—ownership management, time management, and data distribution management—and by restricting the XRTI to operation in a client-server network topology. These limitations do not affect the immediate usefulness of the XRTI for NPSNET-V applications, however, because current virtual worlds based on NPSNET-V operate in real-time, and are small and short-lived.

The products of thesis project include the HLA extension interfaces and the modified object model template (OMT) DTD; the proxy compiler; the XRTI Ambassador; the XRTI Executive; the Bootstrap, Meta-Federation, and Reflection Object Models; the NPSNET-V XRTI controller modules; and the thesis document. The HLA extension interfaces define recommended additions to the standard HLA RTI interfaces, while the modified OMT DTD describes recommended changes to the OMT XML format. The proxy compiler reads FDDs in XML form and generates sets of Java™ source files that the developer may use to interact with the RTI in an intuitive, type-safe manner. The XRTI Ambassador is the central class of the XRTI: the class through which federates (or their proxy classes) perform fundamental operations such as sending interactions and updating attribute values. The XRTI Executive is the server application to which federates must connect when participating in federation executions. The Bootstrap, Meta-Federation, and Reflection Object Models are the object models used for low-level communication, multi-federation management, and object model representation, respectively, much as the Management Object Model defined in the HLA standard is the object model used for management of a single federation. The NPSNET-V XRTI controller modules provide support for transferring entity state between NPSNET-V applications using the XRTI. The thesis document provides a complete description of the design, implementation, integration, and testing of the XRTI.

G. EXPERIMENT

1. Hypothesis

The hypothesis of the thesis experiment states that the XRTI can provide a small virtual world with a communications framework that maintains a level of performance comparable to that of commercially available RTIs. For this thesis, the relevant measures of performance are graphical frame rate, central processing unit (CPU) usage, communication latency, and network traffic volume.

2. Test Method

To test the hypothesis, the author executes three trials within a simple environment installed on several computers connected by a local area network (LAN). In each trial, a small number of entities follow sets of scripted paths. The first and second trials use NPSNET-V's existing HLA controller modules in conjunction with different closed-source, commercial RTIs. The third trial uses the XRTI and the XRTI controller modules. For each trial, the author uses specially developed code to determine the average frame rate and communication latency, and external tools to measure CPU usage and network traffic volume.

H. THESIS ORGANIZATION

The thesis document contains the following chapters:

- Chapter I: Introduction. Provides an overview of the thesis project, a summary of the project's background, a statement of the problem that the thesis addresses, a description of the related work that the thesis builds upon, an explanation of the thesis objectives, a definition of the thesis project's scope, a description of the thesis experiment, a list of the deliverables associated with the project, and a chapter-by-chapter outline of the thesis document.

- Chapter II: High-Level Design. Explains the high-level strategy behind the XRTI's design; describes the technologies selected by the author to implement and test the XRTI; explains the XRTI's conformance to and divergence from

the HLA standard; documents the XRTI's networking model, bootstrapping methodology, and usage of unique integer handles; and provides a coarse-grained overview of the XRTI's object models and software components.

- Chapter III: Low-Level Design and Implementation: Object Models. Documents the creation of the Bootstrap, Meta-Federation, and Reflection Object Models, providing tabular object model descriptions conforming to the format used by the authors of the IEEE 1516.1 standard to describe the MOM.
- Chapter IV: Low-Level Design and Implementation: Software Components. Depicts the design and implementation of the proxy compiler, XRTI Ambassador, and XRTI Executive.
- Chapter V: Integration into NPSNET-V. Provides a brief overview of the NPSNET-V platform and documents the creation of a set of NPSNET-V controller modules that use the XRTI to share entity state information between networked virtual environment applications.
- Chapter VI: Testing. Describes the simple test applications used to verify the XRTI's functionality and documents the thesis experiment, which compares the performance of the XRTI to that of two widely available closed-source RTIs in the context of a typical NPSNET-V shared virtual world.
- Chapter VII: Conclusion. Summarizes the results of the project and compares them to the requirements stated in the introduction, notes the shortcomings of the XRTI and lists opportunities for future enhancement, and provides the reader with the information necessary to obtain the XRTI software from the World Wide Web.

II. HIGH-LEVEL DESIGN

A. DESIGN STRATEGY

The XRTI's design reflects a consistent strategy that emphasizes simplicity, versatility, and maintainability. The first part of the strategy requires that the XRTI use HLA constructs, as well as its own enhancements to the HLA, to as great an extent as possible in the implementation of its underlying mechanisms. This requirement ensures that the XRTI's design minimizes redundancy, credibly validates the HLA standard, and allows developers familiar with the external interface of the XRTI to understand its internals as well. The second part of the strategy dictates that the XRTI should begin as a minimal working prototype, but that its development should spiral outwards over time to incorporate new features as XRTI users request them. The third and final part of the strategy states that as this expansion occurs, the XRTI should support a growing number of profiles, or modes of operation tailored to specific sets of requirements. The initial XRTI prototype documented in this thesis can be thought of as the first profile: a simple client-server configuration suitable for small-scale, Internet-based shared environments. Future profiles will support peer-to-peer and hybrid topologies as well as advanced HLA services such as time management and data distribution management.

B. IMPLEMENTATION PLATFORM

The XRTI is written in Java, built using the Ant tool, and documented using the Javadoc utility. Using the Java Virtual Machine (JVM) as the XRTI's execution platform ensures that it will run unmodified under almost any operating system (OS). The Java language also provides a rich API that covers all categories of functionality required by the XRTI, including threading, data processing, and networking. The Ant tool complements the Java environment by allowing developers to create cross-platform build mechanisms. Unlike typical make files, which must execute OS-specific binaries to perform special tasks, Ant scripts can invoke custom tasks implemented as Java classes. Ant also includes a number of built-in tasks for common operations such as compiling Java classes, manipulating files and directories, and running the Javadoc utility. Javadoc

generates HTML documentation based on specially formatted comments embedded in Java source files. This is particularly useful for the XRTI, because it allows automatic documentation of the classes and interfaces generated by the proxy compiler.

C. STANDARDS COMPLIANCE

Aside from its omission of support for ownership management, time management, and data distribution management, the XRTI is fully compliant with the IEEE 1516, 1516.1, and 1516.2 standards. This means that it follows the rules listed in IEEE 1516, that it provides the Java RTI interface specified in IEEE 1516.1, and that it accepts object models conforming to the template described by IEEE 1516.2. Future versions of the XRTI will support the full HLA standard, and thus will be candidates for official certification.

D. STANDARDS EXTENSION

The XRTI also supports several proposed extensions to the HLA standard. The first is a modified OMT that allows multiple inheritance relationships between object and interaction classes. Multiple inheritance is crucial to dynamically extensible applications because it allows federates to merge classes occupying different regions of the inheritance hierarchy. Consider an object class, `Tank`, and another class, `RadarVisible`, that is not an ancestor of `Tank`. A federate wishes to create a new class: a radar-visible tank. In a traditional HLA federation, this change would require a "FOM-fest" in which the federation is taken offline, the FOM rewritten so that all `Tank`s are `RadarVisible`, the federates updated to reflect the change, and the federation execution restarted. In an XRTI federation, the federate can create the new class without interrupting the federation execution, but only if the two classes can be tied together without changing their ancestry.

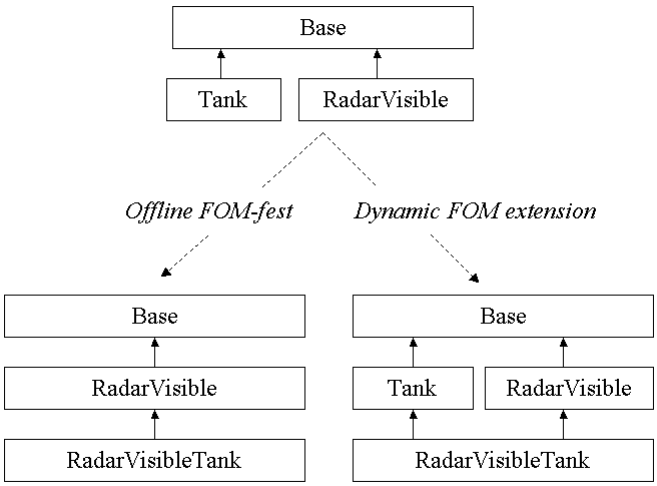

Figure 1. Two ways to merge separate classes. On the left, an offline FOM-fest restructures the inheritance hierarchy. On the right, dynamic FOM extension combines the existing classes using multiple inheritance.

The necessary changes to the OMT can best be expressed as revisions to the document type definition (DTD) contained in section C.2 of IEEE 1516.2. In order to indicate that the DTD defines a revised OMT, the fixed value of the *DTDversion* attribute of the *objectModel* element changes from "1516.2" to "1516.2ex." The *objectClass* and *interactionClass* elements gain new attributes, *parents* and *parentsNotes*, both #IMPLIED and of type NMTOKENS. The value of the *parents* attribute, if present, contains the names of all parents of the object or interaction class other than the direct parent—that is, the parent implicitly specified by nesting *objectClass* or *interactionClass* elements. The *parentsNotes* attribute allows object model authors to link the value of the *parents* attribute with notes defined at the bottom of the FDD.

The second extension adds a single method to the RTI interface in order to allow federates to extend and compose FOMs during the course of a federation execution. In the standard HLA model, the FOM of the execution is fixed when the first federate calls the `createFederationExecution` method, specifying the name of the execution and the location of the FDD as parameters. The XRTI adds a new method, `mergeFDD`, which any federate can invoke at any time (except during a save or restore operation) to

merge the contents of another FDD into the execution's FOM. This extension to the HLA standard not only allows federates to introduce new classes of objects and interactions into their executions, but also encourages the use of lightweight, composable object models. Traditionally, all HLA FOMs must include the complete Management Object Model (MOM), a special object model defined by the HLA standard. This makes even the simplest FDDs thousands of lines long. Under the XRTI, federates can specify an FDD containing only the MOM as the initial FDD, then merge smaller FDDs into the FOM as needed. In fact, since the MOM is a mandatory component of every FOM, the XRTI automatically merges it into the FDD of every execution created.

In order to indicate its status as an extension method, the XRTI defines `mergeFDD` in a new interface, `hla.rti.extensions.RTIambassadorEx`, which inherits from `hla.rti.RTIambassador`, the RTI ambassador interface defined in IEEE 1516.1. The documented signature of the method is as follows.

```
/**
 * Merges the object model contained in the specified federation
 * description document with the current federation object model.
 *
 * @param fdd the location of the federation description document
 * @exception CouldNotOpenFDD if the federation description
 * document could not be opened
 * @exception ErrorReadingFDD if an error occurred while reading
 * the federation description document
 * @exception RTIinternalError if an internal error occurred in
 * the run-time infrastructure
 */
public void mergeFDD(URL fdd)
            throws CouldNotOpenFDD,
                   ErrorReadingFDD,
                   FederateNotExecutionMember,
                   SaveInProgress,
                   RestoreInProgress,
                   RTIinternalError;
```

Figure 2. The signature of the `mergeFDD` method.

The other proposed extensions take the form of standardizable object models which, like the MOM, are accessible to federates but are also used by the XRTI itself. The first of these is the Bootstrap Object Model (BOM), which describes the message protocol payload in terms of HLA data types and defines the most basic interactions, such as `HLAupdateAttributeValues`. The second is the Meta-Federation Object Model (MFOM). Upon initialization, the XRTI Ambassador automatically joins a meta-

federation that consists of all participants in the communication channel that its configuration has instructed it to use. That meta-federation's FOM is the MFOM, which contains interactions like `HLAcreateFederationExecution`. The third special object model is the Reflection Object Model (ROM), which contains object classes such as `HLAobjectClass` and `HLAdataType`. The XRTI uses the constructs contained in the ROM to represent the FOMs of federation executions. When a federate merges a new FDD into the FOM of its execution, the XRTI creates and/or modifies ROM-defined objects to reflect and announce the change.

E. NETWORKING CONSIDERATIONS

Because the XRTI is a distributed system, as opposed to a standalone application or middleware library, its design must include the manner in which its components connect and communicate over the network. The two primary aspects of networking to consider are those of topology and message channels.

1. Topology

Perhaps the most important consideration in any distributed environment is that of the network topology, which determines the connections between components. Most distributed simulation topologies fall into one of three categories: client-server, peer-to-peer, or a hybrid of client-server and peer-to-peer [Singhal 99]. In client-server topologies, a server component accepts messages from and distributes updates to a group of clients that can only communicate with each other through the server. This arrangement is well-suited to HLA simulations, because the functionality that the HLA expects of the RTI—coordinating global save and restore operations, filtering messages based on subscription parameters—suggests the use of a central point of control and mediation. However, pure client-server topologies suffer from limited scalability, because handling a flood of messages from a large number of clients can easily overwhelm a server, and non-optimal latency, because the updates sent by each client must travel first to the server, then to the other clients. For these reasons, RTIs often employ hybrid topologies in which federates rely on a central server, known as an executive, to act as a persistent control node and a means of distributing messages

reliably, but also use IP multicast groups to transmit messages directly between themselves. Because IP multicast offers low latency and high scalability at the cost of guaranteed message delivery, this approach works well for a common class of simulations: those that require frequent unreliable state updates and occasional reliable management operations.

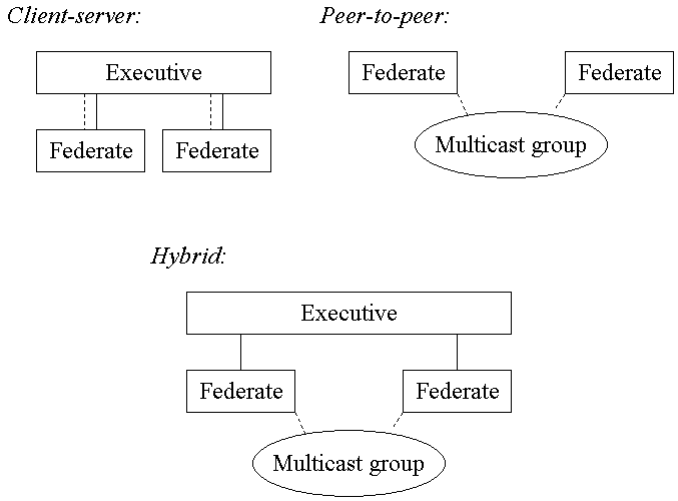

Figure 3. Example RTI network topologies. Solid lines represent reliable message channels; dashed lines represent low-latency, unreliable channels. The hybrid topology uses IP multicast for unreliable state updates and the executive for reliable management operations and central federation control.

Unfortunately, because Internet routers typically do not allow IP multicast traffic to escape the local area network (LAN), it is impractical to deploy multicast-based applications on the Internet. The first version of the XRTI therefore uses a pure client-server topology in which federates only communicate with one another through the XRTI Executive, a server application. It is important, however, that future versions of the XRTI be able to support different topologies. For example, if IP multicast gains Internet-wide acceptance, the XRTI must grow to accommodate peer-to-peer and/or hybrid topologies based on multicast groups in order to allow more scalable federations. Supporting different networking configurations in the same communications standard suggests the development of profiles, or selectable modes of communication. The HLA can define each profile in part through the constructs defined by that profile's BOM. A

BOM that supports peer-to-peer topologies based on IP multicast, for instance, must implement its own reliable multicast protocol by adding sequence numbers to messages and defining retransmission request interactions. The initial BOM, which represents a client-server profile, simplifies matters by assuming that a reliable message channel exists between each federate and the executive.

2. Message Channels

The nature of the message channels through which components communicate is the other networking consideration that the XRTI's design must address. The standard Internet channels are well-known and well-defined, and include unicast Transmission Control Protocol (TCP) connections and User Datagram Protocol (UDP) channels in both unicast and multicast configurations. TCP offers reliable, in-order message delivery at the cost of increased latency, whereas UDP offers low-latency messaging without guaranteeing delivery or order of receipt. In the first version of the XRTI, federates maintain two channels to the executive: a TCP channel for reliable communication and a UDP channel for unreliable messaging. Future versions of the XRTI must be able to support channels other than the standard TCP and UDP options, however, just as they must support different topologies. For example, simulation developers may wish to use overlay multicast, which mimics the functionality of IP multicast by performing application-layer routing, to build large-scale federations in a multicast-hostile Internet. To do this, they must be able to have the XRTI use message channels provided by their custom libraries instead of its default TCP or UDP channels. The XRTI supports this ability by allowing applications to supply channels as Java classes derived from an abstract base class, much as the Java API allows developers to implement custom protocol handlers by extending `URLStreamHandler` [Maso 00].

F. BOOTSTRAPPING METHODOLOGY

The XRTI's bootstrapping methodology allows it to define most of its low-level messaging and management operations in terms of HLA encodings and constructs. The bootstrap process begins with the format of the messages transmitted between XRTI components over the network. XRTI messages consist of a 32-bit protocol identifier

whose value is the same for each version of the XRTI; a 32-bit version number whose value—a 16-bit major version followed by a 16-bit minor version—is equal to the version number of the BOM; and a payload whose format is described by the BOM's *HLAbootstrapInteractionPayload* fixed record type. The following table describes the layout of these fields.

Field offset (bytes)	Description	Contents
0	Protocol identifier	0xFEEDAFED
4	Version number	0x00010000
8	Message payload	*HLAbootstrapInteractionPayload*

Table 1. XRTI message format.

As the name of the *HLAbootstrapInteractionPayload* type suggests, each XRTI message represents an HLA interaction. Small, simple interactions can be encoded directly as messages, while larger, more complex interactions must be transmitted within wrapper interactions, such as *HLAinteractionFragment*, defined in the BOM. Similarly, operations such as updating attribute values must be expressed in terms of the BOM's basic interaction classes. Along with the BOM, the MOM, MFOM and ROM each represent different aspects of the XRTI's internal communication ontology. In order for the XRTI to take advantage of the proxy compiler's ability to generate Java code based on FDDs, the XRTI's compilation consists of two phases. In the first phase, Ant compiles the proxy compiler. Then Ant uses the proxy compiler to generate Java proxies corresponding to the contents of the BOM, MOM, MFOM, and ROM. In the second compilation phase, Ant compiles the autogenerated proxies along with the rest of the XRTI.

G. HANDLES

The HLA standard does not explicitly specify a format for the handles that identify object and interaction classes, object instances, and federates, among other things, in the context of RTI messaging. The MOM treats all handles as opaque byte arrays, and the API defined by IEEE 1516.1 requires that RTIs represent handles as type-safe objects that do not expose their underlying representations. However, the RTI

ambassador specification includes a method, *normalizeFederateHandle*, that requires the ability to convert federate handles into a normalized representation: a 64-bit integer. For simplicity and consistency, the XRTI therefore uses 64-bit integers for all handles. The XRTI Executive generates unique handles and assigns them to federates as necessary. Handle values can serve more than one role; for instance, the value of an object class handle is simply the value of the object instance handle for that class's reflective *HLAobjectClass* instance.

H. OBJECT MODELS

1. Bootstrap Object Model

The BOM contains the most basic, fundamental elements of the XRTI's communication ontology. Its *HLAbootstrapInteractionPayload* data type defines the format of the XRTI message payload. The BOM also defines the *HLAinteractionFragment* interaction class in order to allow clients to transmit oversized interactions as sequences of interaction fragments. The *HLArequestHandles* interaction allows federates to obtain blocks of unique handles from the XRTI Executive, which must respond using *HLAreportHandles*, for use as object instance identifiers. Once they possess these identifiers, federates can use the BOM's *HLAregisterObjectInstance* interaction to announce the creation of new shared objects. The corresponding *HLAdeleteObjectInstance* interaction is defined in the MOM, as are publication and subscription services such as *HLApublishObjectClassAttributes* and *HLAsubscribeObjectClassAttributes*. To announce changes in the state of the attributes that they have published, federates must use the BOM's *HLAupdateAttributeValues* family of interactions. This family includes variants for attributes with both reliable and best-effort transportation: *HLAupdateAttributeValuesReliable* and *HLAupdateAttributeValuesBestEffort*. These interaction classes and the others defined in the BOM derive from an organizational base class, *HLAbootstrap*.

2. Meta-Federation Object Model

The MFOM supplements the MOM by providing a management layer outside of any single federation execution. Federates join the meta-federation implicitly when they

initialize their XRTI Ambassadors. The MFOM contains the *HLAcreateFederationExecution* interaction class, which federates use to create federation executions; the *HLAjoinFederationExecution* class, with which federates join federation executions; and the *HLAdestroyFederationExecution* class, which federates use to destroy federation executions. To leave a federation execution, federates must invoke the *HLAleaveFederationExecution* interaction, which is defined by the MOM. Instances of the MFOM's *HLAfederationExecution* object class represent active federation executions. All of the MFOM's interaction classes are children of its *HLAmetaFederation* interaction class. Similarly, the *HLAmetaFederation* object class acts as a base class for *HLAfederationExecution*.

3. Reflection Object Model

The ROM is the most straightforward of the XRTI's three core object models. It describes a number of classes based directly on elements of the Object Model Template (OMT). Where the OMT defines an *objectClass* element, for instance, the ROM defines an *HLAobjectClass* object class. That object class contains as its attributes the same information that the OMT requires *objectClass* elements to contain: the name of the object class, its parent classes, its English-language semantics, and its defined attributes. Within the XRTI, instances of ROM-defined classes like *HLAobjectClass* and *HLAinteractionClass* represent living, mutable aspects of the FOM. When an XRTI client modifies the *attributes* attribute of an *HLAobjectClass*, for example, it is changing the list of attributes supported by instances of the described class. For indirect FOM extension, the ROM defines the *HLAmergeFDD* interaction class. The primary object classes of the ROM are children of a common base class, *HLAreflection*, and include *HLAobjectClass*, *HLAinteractionClass*, *HLAdimension*, *HLAsynchronization*, *HLAtransportation*, and *HLAdataType*. The *HLAmergeFDD* interaction class inherits from the base interaction class *HLAreflection*.

I. SOFTWARE COMPONENTS

1. Proxy Compiler

The proxy compiler, which is both a standalone Java application and an Ant task, converts XML FDDs into sets of Java source files representing proxy classes and interfaces. The compiler begins by mapping simple types to appropriate Java equivalents. For example, the *HLAboolean* type maps naturally to the Java *boolean* type. The compiler then generates Java classes for enumerated types, fixed record types, and variant record types. Each class includes constructors, accessors, mutators, serialization and deserialization methods, and debugging methods as necessary. Next, the compiler creates a proxy ambassador class and an interaction listener interface. Clients use instances of the proxy ambassador class to send and receive the interactions described in the FDD, as well as to create and manage proxy objects. For each object class defined in the FDD, the proxy compiler must generate three Java source files: an object interface, a listener interface, and a proxy class. This is necessary because of the XRTI's support for multiple inheritance relationships between object classes. Java classes may implement an arbitrary number of interfaces, but they may have only one direct parent class. The proxy compiler therefore derives each child proxy class from the proxy class of the direct parent (identified by the *objectClass* element that encloses the child element in the FDD) and ensures that the child proxy implements the object interfaces of its other parents. This strategy allows clients to use Java's *instanceof* operator to identify object proxy types in terms of the classes defined in the FDD.

2. XRTI Ambassador

Each federate participating in an XRTI federation execution must maintain an instance of the XRTI Ambassador as its interface to the federation. The methods that the XRTI Ambassador must support are strictly defined by the IEEE 1516 standard, with the exception of the XRTI's new *mergeFDD* method. Typical XRTI federates, however, invoke few of those methods directly. Instead, they interact with their autogenerated proxy classes, which in turn manipulate the cumbersome interface of the XRTI Ambassador. Upon initialization, the XRTI Ambassador creates a connection to the

XRTI Executive and implicitly joins the meta-federation. At that point, the federate can call the ambassador's *createFederationExecution*, *destroyFederationExecution*, and *joinFederationExecution* methods. Creating a federation execution involves uploading an FDD to the XRTI Executive so that the executive can create the set of reflection objects that represent the FOM. Once the federate has joined a federation, it can use the *mergeFDD* method to add the contents of other FDDs to the FOM. The federate should also report its publication and subscription information to the executive, allowing the executive to filter the interactions and object state changes that it relays to the federate. If the federate has objects of its own, it can use the *registerObjectInstance* method to announce their presence to the executive, the *updateAttributeValues* method to announce changes in their state, and the *deleteObjectInstance* method to signal their removal. The federate can also use the *sendInteraction* method to transmit published interactions. Message transmission is asynchronous; as soon as the XRTI Ambassador receives a message from the XRTI Executive, it relays it to the federate through the FederateAmbassador interface. When the federate must leave the federation execution, it calls the ambassador's *resignFederationExecution* method. After resigning, the federate is returned to the meta-federation. It is only when the *finalizeRTI* method is called that the ambassador severs its link to the XRTI Executive.

3. XRTI Executive

Each federation execution hosted by the current version of the XRTI must be managed by an instance of the XRTI Executive, a standalone application that acts as a central server. After initialization, the XRTI Executive waits for incoming connections from remote instances of the XRTI Ambassador. For each connection created, the XRTI Executive spawns a new thread to handle communication with the remote client. If the client requests a block of handles, the executive responds with a block of the requested size. If the client requests the creation of a federation execution, the executive creates the execution and initializes the reflection objects that represent its FOM based on the initial FDD provided by the federate. In addition to these reflection objects, the executive must maintain other types of information over the lifetime of the federation execution. The executive must keep track of each joined federate's subscription information, as well as

the object instances that each federate has created. The executive must also respond to the service requests defined in the MOM, such as *HLArequestPublications* and *HLArequestSubscriptions*. The executive should be able to handle any number of federation executions simultaneously, within the constraints of available bandwidth and processing power.

THIS PAGE INTENTIONALLY LEFT BLANK

III. LOW-LEVEL DESIGN AND IMPLEMENTATION: OBJECT MODELS

A. OBJECT MODEL TABLES

In order to completely and precisely describe the object models associated with the XRTI, this chapter includes tabular representations based on those used by the authors of IEEE 1516.1 to present the Management Object Model (MOM). Each set of object model tables begins with object and/or interaction class structure tables that depict inheritance relationships between classes. The *P*, *S*, *PS*, or *N* flag that follows each class name indicates that federates can publish, subscribe, publish and subscribe, or neither publish nor subscribe, respectively, instances of the class. The next tables describe object attributes and interaction parameters. Attribute tables list the name, data type, update type, update condition, ownership transfer capability, publication and subscription capability, available dimensions, transportation, and order type of each attribute of every object class. Update types include *Static*, indicating that the attribute value does not change after initialization; *Conditional*, indicating that the attribute value changes on the specified update condition; and *Periodic*, indicating that the attribute's owner updates its value at regular intervals. An ownership transfer capability is one of *D*, *A*, *DA*, or *N*, respectively indicating that federates can divest, acquire, divest and acquire, or neither divest nor acquire, ownership of the attribute. Transportation is either *HLAreliable* or *HLAbestEffort*, and order is either *Receive* or *TimeStamp*. Parameter tables list the name, data type, available dimensions, transportation, and order type of each interaction parameter.

Because the XRTI relies on the MOM as well as the object models described in this chapter, the tables that follow reference types contained in the MOM, such as *HLAopaqueData*, without defining them. The set of tables that describe other data types begins with the simple data type table, which lists the name, representation, units, resolution, accuracy, and semantics of each simple data type. Next comes the enumerated data type table, which lists the name, representation, and semantics of each enumerated type, as well as the name and values of each one of that type's enumerators.

The array data type table similarly lists the name, element type, cardinality, encoding, and semantics of each array type. The cardinality can either be a fixed integer or *Dynamic*, indicating an array of variable length. The *HLAvariableArray* encoding is defined in the HLA standard, allowing federates to exchange encoded array data reliably. The fixed record data type table follows the array table, and contains the name, fields, encoding, and semantics of each fixed record type. Each field has a name, a data type, and its own semantics. Like *HLAvariableArray*, the *HLAfixedRecord* encoding defines an exact mapping from a live data structure to its serialized representation. The final set of object model tables provide semantics for each object class, interaction class, object class attribute, and interaction class parameter.

B. BOOTSTRAP OBJECT MODEL

The BOM contains the most basic of the XRTI's communication constructs. It defines the *HLAbootstrapInteractionPayload* fixed record type, for instance, which represents the payload of each message sent between distributed XRTI components. *HLAbootstrapInteractionPayload* contains four fields: the handle of the federation execution with which the message is associated, an uninterpreted user-supplied tag, the handle of the interaction class of which the message represents an instance, and a list of pairings between interaction parameter handles and their associated values.

The BOM also defines a number of fundamental interaction classes. The first of these is *HLAinteractionFragment*, which XRTI components use to send large interactions through best-effort channels that limit message sizes. Each interaction fragment contains four parameters: an interaction number that distinguishes fragments of one interaction from those of others, the total size of the fragmented interaction, the offset of the fragment, and the contents of the fragment. To send a fragmented interaction, XRTI components encode the interaction into a byte array and transmit a series of *HLAinteractionFragment* messages, each containing a portion of the array with a length underneath the message channel's size threshold. Upon receiving an interaction fragment, XRTI components create a buffer of the indicated size and begin to collect subsequent fragments. When the entire buffer is filled, the component decodes and

processes the interaction. If the component receives a fragment with a different interaction number before the buffer is filled, it discards all previously received fragments and starts over.

The *HLArequestHandles* and *HLAreportHandles* interaction classes allow federates to acquire blocks of contiguous unique identifiers from the XRTI Executive. Federates send the *HLArequestHandles* interaction to obtain a block of handles, using the *blockSize* parameter to indicate the number of handles desired. The XRTI Executive responds with the *HLAreportHandles* interaction, which relays the first handle of the block and the total number of handles acquired. Once the federate has acquired a series of handles, it can use them in interactions such as *HLAregisterObjectInstance*. *HLAregisterObjectInstance* announces the presence of a new shared object to the XRTI Executive, taking as its parameters the name of the object, its instance handle, and its class handle.

Federates use the *HLArequestAttributeValueUpdate* interaction to request that other federates transmit updated attribute values. The parameters of *HLArequestAttributeValueUpdate* are the instance handle of the object of interest and a list of handles identifying the attribute values desired. To announce updated attribute values, federates use either *HLAupdateAttributeValuesReliable* or *HLAupdateAttributeValuesBestEffort,* depending on the transportation of the attributes being updated. Both classes include as parameters the instance handle of the object with which the attributes are associated and a list of attribute handle/value pairs.

HLAinteractionRoot (N)	HLAbootstrap (N)	HLAinteractionFragment (PS)	
		HLArequestHandles (PS)	
		HLAreportHandles (PS)	
		HLAregisterObjectInstance (PS)	
		HLArequestAttributeValueUpdate (PS)	
		HLAupdateAttributeValues (N)	HLAupdateAttributeValuesReliable (PS)
			HLAupdateAttributeValuesBestEffort (PS)

Table 2. BOM interaction class structure table.

Interaction		Parameter	Datatype	Available dimensions	Transportation	Order
HLAinteractionRoot	HLAbootstrap	NA	NA	NA	HLAreliable	Receive
HLAbootstrap	HLAinteractionFragment	interactionNumber	HLAinteraction-SequenceNumber	NA	HLAreliable	Receive
		interactionSize	HLAbufferSize			
		fragmentOffset	HLAbufferOffset			
		fragmentContents	HLAopaqueData			
	HLArequestHandles	blockSize	HLAhandleBlockSize	NA	HLAreliable	Receive
	HLAreportHandles	blockStart	HLAnormalizedHandle	NA	HLAreliable	Receive
		blockSize	HLAhandleBlockSize			
	HLAregisterObjectInstance	objectName	HLAunicodeString	NA	HLAreliable	Receive
		objectInstanceHandle	HLAnormalizedHandle			
		objectClassHandle	HLAnormalizedHandle			
	HLArequest-AttributeValueUpdate	objectInstanceHandle	HLAnormalizedHandle	NA	HLAreliable	Receive
		attributeHandleList	HLAattributeHandleList			
	HLAupdateAttributeValues	objectInstanceHandle	HLAnormalizedHandle	NA	HLAreliable	Receive
		attributeHandleValuePairList	HLAattributeHandle-ValuePairList			
HLAupdate-AttributeValues	HLAupdateAttribute-ValuesBestEffort	NA	NA	NA	HLAbestEffort	Receive
	HLAupdateAttribute-ValuesReliable	NA	NA	NA	HLAreliable	Receive

Table 3. BOM parameter table.

Name	Representation	Units	Resolution	Accuracy	Semantics
HLAnormalizedHandle	HLAinteger64BE		NA	NA	A normalized handle.
HLAhandleBlockSize	HLAinteger64BE	handles	NA	NA	The size of a block of handles.
HLAinteractionSequenceNumber	HLAinteger32BE		NA	NA	An interaction sequence number.
HLAbufferSize	HLAinteger32BE	bytes	NA	NA	A data buffer size.
HLAbufferOffset	HLAinteger32BE	bytes	NA	NA	An data buffer offset.

Table 4. BOM simple datatype table.

Name	Element Type	Cardinality	Encoding	Semantics
HLAparameterHandle-ValuePairList	HLAparameterHandle-ValuePair	Dynamic	HLAvariableArray	A list of parameter handle/value pairs.
HLAattributeHandle-ValuePairList	HLAattributeHandle-ValuePair	Dynamic	HLAvariableArray	A list of attribute handle/value pairs.
HLAattributeHandleList	HLAnormalizedHandle	Dynamic	HLAvariableArray	A list of attribute handles.

Table 5. BOM array datatype table.

Record name	Field			Encoding	Semantics
	Name	Type	Semantics		
HLAbootstrapInteractionPayload	federationExecutionHandle	HLAnormalizedHandle	Federation execution handle.	HLAfixedRecord	The payload of the bootstrap interaction.
	userSuppliedTag	HLAopaqueData	User-supplied tag associated with the interaction.		
	interactionClassHandle	HLAnormalizedHandle	Interaction class handle.		
	parameterHandleValuePairList	HLAparameterHandleValuePairList	List of parameter handle/value pairs.		
HLAparameterHandleValuePair	parameterHandle	HLAnormalizedHandle	Parameter handle.	HLAfixedRecord	Pairs a parameter handle with an encoded value.
	value	HLAopaqueData	Encoded value.		
HLAattributeHandleValuePair	attributeHandle	HLAnormalizedHandle	Attribute handle.	HLAfixedRecord	Pairs an attribute handle with an encoded value.
	value	HLAopaqueData	Encoded value.		

Table 6. BOM fixed record datatype table.

Interaction		Definition
HLAbootstrap		Root class of BOM interactions.
HLAbootstrap	HLAinteractionFragment	A piece of a fragmented interaction.
	HLArequestHandles	Requests a block of handles.
	HLAreportHandles	Reports a block of handles.
	HLAregisterObjectInstance	Registers a new object instance.
	HLArequestAttributeValueUpdate	Requests an attribute value update.
	HLAupdateAttributeValues	Updates a set of attribute values.
HLAupdateAttributeValues	HLAupdateAttributeValuesBestEffort	Updates a set of attribute values with best-effort transportation and receive ordering.
	HLAupdateAttributeValuesReliable	Updates a set of attribute values with reliable transportation and receive ordering.

Table 7. BOM interaction class definitions table.

Class		Parameter	Definition
HLAbootstrap	HLAinteractionFragment	interactionNumber	The sequence number of the fragmented interaction.
		interactionSize	The size of the fragmented interaction.
		fragmentOffset	The offset of this fragment in the buffer.
		fragmentContents	The contents of the fragment.
	HLArequestHandles	blockSize	The number of handles desired.
	HLAreportHandles	blockStart	The first handle in the block.
		blockSize	The number of handles in the block.
	HLAregisterObjectInstance	objectName	The name of the object.
		objectInstanceHandle	The object instance handle
		objectClassHandle	The object class handle.
	HLArequestAttributeValueUpdate	objectInstanceHandle	Object instance handle.
		attributeHandleList	List of attribute handles.
	HLAupdateAttributeValues	objectInstanceHandle	Object instance handle.
		attributeHandleValuePairList	List of attribute handle/value pairs.

Table 8. BOM parameter definitions table.

C. META-FEDERATION OBJECT MODEL

The MFOM defines the object and interaction classes of the meta-federation: the federation execution that federates join implicitly by initializing their XRTI Ambassador instances and connecting to the XRTI Executive. The meta-federation acts as a sort of lobby, allowing federates to browse all federation executions maintained by the XRTI Executive, and to create, join, and destroy executions. Instances of the *HLAfederationExecution* object class, maintained by the executive, represent active federation executions. Each execution must have a unique name, represented by the *name* attribute. To create executions, federates invoke the *HLAcreateFederationExecution* interaction, providing a name and an encoded FDD as parameters. Upon receiving *HLAcreateFederationExecution*, the executive creates a new execution with the specified name, initializes its reflection objects to reflect the content of the transmitted FDD, and creates a new instance of *HLAfederationExecution* in the meta-federation. At that point, federates can send *HLAjoinFederationExecution* to join the execution. *HLAjoinFederationExecution* takes as its parameters the name of the execution to join, the type of the federate, and the federate's handle. Federates use their federate handles primarily to identify themselves in the interactions defined by the MOM, such as *HLAresignFederationExecution*. After resigning from an execution, federates return to the meta-federation, where they can create and join other executions, or invoke *HLAdestroyFederationExecution* to destroy executions.

HLAobjectRoot (N)	HLAmetaFederation (N)	HLAfederationExecution (PS)

Table 9. MFOM object class structure table.

HLAinteractionRoot (N)	HLAmetaFederation (N)	HLAcreateFederationExecution (PS)
		HLAjoinFederationExecution (PS)
		HLAdestroyFederationExecution (PS)

Table 10. MFOM interaction class structure table.

Object	Attribute	Datatype	Update type	Update condition	T/A	P/S	Available dimensions	Transportation	Order
HLAfederationExecution	name	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive

Table 11. MFOM attribute table.

Interaction		Parameter	Datatype	Available dimensions	Transportation	Order
HLAinteractionRoot	HLAmetaFederation	NA	NA	NA	HLAreliable	Receive
HLAmetaFederation	HLAcreateFederationExecution	federationExecutionName	HLAunicodeString	NA	HLAreliable	Receive
		federationDescriptionDocument	HLAopaqueData			
	HLAdestroyFederationExecution	federationExecutionName	HLAunicodeString	NA	HLAreliable	Receive
	HLAjoinFederationExecution	federationExecutionName	HLAunicodeString	NA	HLAreliable	Receive
		federateType	HLAunicodeString			
		federateHandle	HLAnormalizedHandle			

Table 12. MFOM parameter table.

Name	Representation	Units	Resolution	Accuracy	Semantics
HLAnormalizedHandle	HLAinteger64BE	NA	NA	NA	A normalized handle.

Table 13. MFOM simple datatype table.

Object	Definition
HLAmetaFederation	This object class is the root class of all MFOM object classes.
HLAfederationExecution	Represents an HLA federation execution.

Table 14. MFOM object class definitions table.

Interaction		Definition
HLAmetaFederation		Root class of MFOM interactions.
HLAmetaFederation	HLAcreateFederationExecution	Creates a federation execution.
	HLAdestroyFederationExecution	Destroys a federation execution.
	HLAjoinFederationExecution	Joins a federation execution.

Table 15. MFOM interaction class definitions table.

Class	Attribute	Definition
HLAfederationExecution	name	The name of the execution.

Table 16. MFOM attribute definitions table.

Class		Parameter	Definition
HLAmetaFederation	HLAcreateFederationExecution	federationExecutionName	The name of the execution to create.
		federationDescriptionDocument	The encoded federation description document.
	HLAdestroyFederationExecution	federationExecutionName	The name of the execution to destroy.
	HLAjoinFederationExecution	federationExecutionName	The name of the execution to join.
		federateType	The type of the joining federate.
		federateHandle	The handle of the joining federate.

Table 17. MFOM parameter definitions table.

D. REFLECTION OBJECT MODEL

The ROM defines the object and interaction classes required for run-time communication ontology reflection and extension. Its object classes are based on elements of the OMT: object classes, attributes, interaction classes, parameters, dimensions, synchronizations, transportations, and data types. The XRTI Executive creates instances of these reflection objects to represent the contents of the original FDD specified as a parameter to the MFOM's *HLAcreateFederationExecution* interaction and any FDDs that federates have merged into the FOM via the ROM's *HLAmergeFDD* interaction. The values of the instance handles of *HLAobjectClass*, *HLAattribute*, *HLAinteractionClass*, *HLAparameter*, and *HLAdimension* instances are equal to the values of the corresponding object class handle, attribute handle, interaction class handle, parameter handle, and dimension handle, respectively. Instances of *HLAobjectClass* contain an attribute named *attributes* whose value is a list of handles identifying attributes associated with the class. Similarly, instances of *HLAinteractionClass* contain an attribute named *parameters*. Other attributes—for instance, the *parents* attribute—are strings that identify shared objects by name. When federates invoke the *HLAmergeFDD* interaction, they specify an encoded FDD as its parameter. It is the responsibility of the XRTI Executive to resolve any conflicts that may exist between the existing FOM and the new FDD.

HLAobjectRoot (N)	HLAreflection (N)	HLAobjectClass (PS)	
		HLAattribute (PS)	
		HLAinteractionClass (PS)	
		HLAparameter (PS)	
		HLAdimension (PS)	
		HLAsynchronization (PS)	
		HLAtransportation (PS)	
		HLAdataType(N)	HLAbasicRepresentation (PS)
			HLAsimpleDataType (PS)
			HLAenumeratedDataType (PS)
			HLAarrayDataType (PS)
			HLAfixedRecordDataType (PS)
			HLAvariantRecordDataType (PS)

Table 18. ROM object class structure table.

HLAinteractionRoot (N)	HLAreflection (N)	HLAmergeFDD (PS)

Table 19. ROM interaction class structure table.

Object	Attribute	Datatype	Update type	Update condition	T/A	P/S	Available dimensions	Transportation	Order
HLAreflection	name	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
HLAobjectClass	parents	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	sharing	HLAsharingType	Static	NA	N	PS	NA	HLAreliable	Receive
	attributes	HLAattributeHandleList	Static	NA	N	PS	NA	HLAreliable	Receive
	semantics	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
HLAattribute	dataType	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	updateType	HLAupdateType	Static	NA	N	PS	NA	HLAreliable	Receive
	updateCondition	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	ownership	HLAownershipType	Static	NA	N	PS	NA	HLAreliable	Receive
	sharing	HLAsharingType	Static	NA	N	PS	NA	HLAreliable	Receive
	dimensions	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	transportation	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	order	HLAorderType	Static	NA	N	PS	NA	HLAreliable	Receive
	semantics	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
HLAinteractionClass	parents	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	sharing	HLAsharingType	Static	NA	N	PS	NA	HLAreliable	Receive
	dimensions	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	transportation	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	order	HLAorderType	Static	NA	N	PS	NA	HLAreliable	Receive
	parameters	HLAparameterHandleList	Static	NA	N	PS	NA	HLAreliable	Receive
	semantics	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
HLAparameter	name	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	dataType	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	semantics	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	semantics	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive

Table 20. ROM attribute table.

Object	Attribute	Datatype	Update type	Update condition	T/A	P/S	Available dimensions	Transportation	Order
HLAdimension	dataType	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	upperBound	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	normalization	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	value	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
HLAsynchronization	label	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	dataType	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	capability	HLAcapabilityType	Static	NA	N	PS	NA	HLAreliable	Receive
	semantics	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
HLAtransportation	name	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	description	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
HLAbasicRepresentation	size	HLAbasicRepresentationSize	Static	NA	N	PS	NA	HLAreliable	Receive
	endian	HLAendianType	Static	NA	N	PS	NA	HLAreliable	Receive
	interpretation	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	encoding	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
HLAsimpleDataType	representation	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	units	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	resolution	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	accuracy	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	semantics	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
HLAenumeratedDataType	representation	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	enumerators	HLAenumeratorList	Static	NA	N	PS	NA	HLAreliable	Receive
	semantics	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive

Table 21. ROM attribute table (continued).

Object	Attribute	Datatype	Update type	Update condition	T/A	P/S	Available dimensions	Transportation	Order
HLAarrayDataType	dataType	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	cardinality	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	encoding	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	semantics	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
HLAfixedRecordDataType	fields	HLAfieldList	Static	NA	N	PS	NA	HLAreliable	Receive
	encoding	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	semantics	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
HLAvariantRecordDataType	discriminant	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	dataType	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	alternatives	HLAalternativeList	Static	NA	N	PS	NA	HLAreliable	Receive
	encoding	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	semantics	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive

Table 22. ROM attribute table (continued).

Interaction		Parameter	Datatype	Available dimensions	Transportation	Order
HLAinteractionRoot	HLAreflection	NA	NA	NA	HLAreliable	Receive
HLAreflection	HLAmergeFDD	federationDescriptionDocument	HLAopaqueData	NA	HLAreliable	Receive

Table 23. ROM parameter table.

Name	Representation	Units	Resolution	Accuracy	Semantics
HLAnormalizedHandle	HLAinteger64BE	N/A	N/A	N/A	A normalized handle.
HLAbasicRepresentationSize	HLAinteger32BE	bits	N/A	N/A	N/A

Table 24. ROM simple datatype table.

Name	Representation	Enumerator	Values	Semantics
HLAendianType	HLAinteger32BE	Big	0	Endian type to be used for describing basic representations.
		Little	1	
HLAorderType	HLAinteger32BE	Receive	0	Order type to be used for sending attributes or interactions.
		TimeStamp	1	
HLAownershipType	HLAinteger32BE	Divest	0	Ownership type to be used for sending attributes.
		Acquire	1	
		DivestAcquire	2	
		NoTransfer	3	
HLAsharingType	HLAinteger32BE	Publish	0	Sharing type to be used for sending attributes or interactions.
		Subscribe	1	
		PublishSubscribe	2	
		Neither	3	
HLAupdateType	HLAinteger32BE	Static	0	Update type to be used for sending attributes.
		Periodic	1	
		Conditional	2	
		NA	3	
HLAcapabilityType	HLAinteger32BE	Register	0	Capability type to be used for synchronizations.
		Achieve	1	
		RegisterAchieve	2	
		NoSynch	3	

Table 25. ROM enumerated datatype table.

Name	Element Type	Cardinality	Encoding	Semantics
HLAattributeHandleList	HLAnormalizedHandle	Dynamic	HLAvariableArray	A list of attribute handles.
HLAparameterHandleList	HLAnormalizedHandle	Dynamic	HLAvariableArray	A list of parameter handles.
HLAenumeratorList	HLAenumerator	Dynamic	HLAvariableArray	List of enumerated type enumerators.
HLAfieldList	HLAfield	Dynamic	HLAvariableArray	List of fixed record type fields.
HLAalternativeList	HLAalternative	Dynamic	HLAvariableArray	List of variant record type alternatives.

Table 26. ROM array datatype table.

Record name	Field			Encoding	Semantics
	Name	Type	Semantics		
HLAenumerator	name	HLAunicodeString	Enumerator name.	HLAfixedRecord	Enumerated type enumerator.
	values	HLAunicodeString	Enumerator values.		
HLAfield	name	HLAunicodeString	Field name.	HLAfixedRecord	Fixed record type field.
	dataType	HLAunicodeString	Field data type.		
	semantics	HLAunicodeString	Field semantics.		
HLAalternative	enumerator	HLAunicodeString	Alternative enumerator.	HLAfixedRecord	Variant record type alternative.
	name	HLAunicodeString	Alternative name.		
	dataType	HLAunicodeString	Alternative data type.		
	semantics	HLAunicodeString	Alternative semantics.		

Table 27. ROM fixed record datatype table.

Object	Definition
HLAreflection	This object class is the root class of all ROM object classes.
HLAobjectClass	Represents an HLA object class.
HLAattribute	Represents an HLA object attribute.
HLAinteractionClass	Represents an HLA interaction class.
HLAparameter	Represents an HLA interaction parameter.
HLAdimension	Represents an HLA dimension.
HLAsynchronization	Represents an HLA synchronization.
HLAtransportation	Represents an HLA transportation.
HLAdataType	Represents an HLA data type.
HLAbasicRepresentation	Represents an HLA basic representation.
HLAsimpleDataType	Represents an HLA simple data type.
HLAenumeratedDataType	Represents an HLA enumerated data type.
HLAarrayDataType	Represents an HLA array data type.
HLAfixedRecordDataType	Represents an HLA fixed record data type.
HLAvariantRecordDataType	Represents an HLA variant record data type.

Table 28. ROM object class definitions table.

Interaction		Definition
HLAreflection		Root class of ROM interactions.
HLAreflection	HLAmergeFDD	Merges the contents of a new federation description document into the FOM.

Table 29. ROM interaction class definitions table.

Class	Attribute	Definition
HLAreflection	name	The name of the reflection object.
HLAobjectClass	parents	The parents of the object class.
	sharing	The types of sharing permitted on the object.
	attributes	The object attributes.
	semantics	The object semantics.
HLAattribute	dataType	Attribute data type.
	updateType	Attribute update type.
	updateCondition	Attribute update condition.
	ownership	Attribute ownership.
	sharing	Attribute sharing.
	dimensions	Attribute dimensions.
	transportation	Attribute transportation.
	order	Attribute order.
	semantics	Attribute semantics.
HLAinteractionClass	parents	The parents of the interaction class.
	sharing	The types of sharing permitted on the interaction.
	dimensions	The dimensions of the interaction.
	transportation	The transportation of the interaction.
	order	The order of the interaction.
	parameters	The parameters of the interaction.
	semantics	The semantics of the interaction.
HLAparameter	name	Parameter name.
	dataType	Parameter data type.
	semantics	Parameter semantics.
HLAdimension	dataType	The data type of the dimension.
	upperBound	The upper bound of the dimension.
	normalization	The normalization of the dimension.
	value	The value of the dimension.
HLAsynchronization	label	The label of the synchronization.
	dataType	The data type of the synchronization.
	capability	The capability of the synchronization.
	semantics	The semantics of the synchronization.
HLAtransportation	name	The name of the transportation.
	description	The description of the transportation.
HLAbasicRepresentation	size	The size of the basic representation.
	endian	The byte ordering of the basic representation.
	interpretation	The interpretation of the basic representation.
	encoding	The encoding of the basic representation.

Table 30. ROM attribute definitions table.

]

Class	Attribute	Definition
HLAsimpleDataType	representation	The representation of the simple type.
	units	The units of the simple type.
	resolution	The resolution of the simple type.
	accuracy	The accuracy of the simple type.
	semantics	The semantics of the simple type.
HLAenumeratedDataType	representation	The representation of the enumerated type.
	enumerators	The enumerators of the enumerated type.
	semantics	The semantics of the enumerated type.
HLAarrayDataType	dataType	The element type of the array type.
	cardinality	The cardinality of the array type.
	encoding	The encoding of the array type.
	semantics	The semantics of the array type.
HLAfixedRecordDataType	fields	The fields of the fixed record type.
	encoding	The encoding of the fixed record type.
	semantics	The semantics of the fixed record type.
HLAvariantRecordDataType	discriminant	The discriminant of the variant record type.
	dataType	The data type of the variant record type.
	alternatives	The alternatives of the variant record type.
	encoding	The encoding of the variant record type.
	semantics	The semantics of the variant record type.

Table 31. ROM attribute definitions table (continued).

Class		Parameter	Definition
HLAreflection	HLAmergeFDD	federationDescriptionDocument	The encoded federation description document.

Table 32. ROM parameter definitions table.

IV. LOW-LEVEL DESIGN AND IMPLEMENTATION: SOFTWARE COMPONENTS

A. PROXY COMPILER

The proxy compiler converts XML FOM Document Data (FDDs) into sets of proxy source files. Developers can use these proxies with any RTI conforming to the IEEE 1516 HLA standard, but the XRTI makes special use of the proxy compiler to generate proxies corresponding to the Bootstrap Object Model (BOM), the Meta-Federation Object Model (MFOM), the Reflection Object Model (ROM), and the Management Object Model (MOM). The XRTI's other software components, the XRTI Ambassador and the XRTI Executive, use these autogenerated proxies as part of the bootstrap process. For example, the proxy compiler converts the BOM's *HLAbootstrapInteractionPayload* data type into a Java class, `HLAbootstrapInteractionPayload`, that components can use to read and write payloads easily and efficiently. For this reason, the proxy compiler is a fundamental and important component of the XRTI.

1. Type Mappings

One of the most basic tasks of the proxy compiler is that of mapping the types defined by the HLA standard to equivalent Java data types. The HLA's object model template (OMT) specification includes a number of basic representations, simple types, enumerated types, and array types that every object model must support. The following tables indicate the manner in which the proxy compiler maps these built-in types, and any additional derived types defined in the FDD, to Java types.

HLA basic representation	Java type
HLAinteger16BE	short
HLAinteger16LE	short
HLAinteger32BE	int
HLAinteger32LE	int
HLAinteger64BE	long
HLAinteger64LE	long
HLAfloat32BE	float
HLAfloat32LE	float
HLAfloat64BE	double
HLAfloat64LE	double
HLAoctetPairBE	short
HLAoctetPairLE	short
HLAoctet	byte
(all others)	byte[]

Table 33. Mappings between HLA basic representations and Java data types.

Basic representations include integer, floating point, and octet values in different sizes and endian configurations. The XRTI maps 16-bit, 32-bit, and 64-bit integer representations to Java's `short`, `int`, and `long` types, respectively. Java's `float` and `double` types are similarly equivalent to the HLA's 32-bit and 64-bit floating point representations. The endian configuration of each representation determines the manner in which the code generated by the proxy compiler serializes that representation's values. The proxy compiler maps octet pairs—uninterpreted pairs of bytes—to `short` values, and octets to `byte` values. When the proxy compiler encounters basic representations without an explicit mapping, it maps them to byte arrays. Byte arrays are the lowest-common-denominator representation; any kind of data can be expressed as a byte array.

HLA simple type	Java type
HLAASCIIchar	char
HLAunicodeChar	char
(all others)	(basic representation mapping)

Table 34. Mappings between HLA simple types and Java data types.

For most simple data types, the XRTI simply retrieves the type's basic representation and uses that to obtain a mapping. However, the *HLAASCIIchar* and *HLAunicodeChar* map to Java's `char` type.

HLA enumerated type	Java type
HLAboolean	boolean
(all others)	(HLA type name)

Table 35. Mappings between HLA enumerated types and Java data types.

The *HLAboolean* type is equivalent to Java's `boolean` primitive; the *HLAfalse* enumerator becomes `false` and the *HLAtrue* enumerator becomes `true`. For all other enumerated types, as well as for all fixed and variant record types, the proxy compiler generates a Java source file with the name of the defined type as described in section IV.3.a.

HLA array type	Java type
HLAASCIIstring	java.lang.String
HLAunicodeString	java.lang.String
(all others)	(result of HLA type mapping + "[]")

Table 36. Mappings between HLA array types and Java data types.

Both *HLAASCIIstring* and *HLAunicodeString* map to Java's string class, `java.lang.String`. The proxy compiler maps all other array types by first obtaining a mapping for the element type, then appending Java's array indicator: a pair of square brackets. An array of arrays of *HLAboolean* values would thus, for instance, map to `boolean[][]`.

2. Encoding Streams

The encoding and decoding sequences generated by the proxy compiler to marshal object instance attributes and interaction parameters depend on instances of `HLAEncodingInputStream` and `HLAEncodingOutputStream` to read and write basic HLA type values. To encode an autogenerated fixed record data type class, for instance, the proxy compiler generates a sequence that wraps a Java

`ByteArrayOutputStream` within an `HLAEncodingOutputStream` and invokes the class's `encode` method. The body of that method, itself generated by the proxy compiler, writes each field of the fixed record to the stream in sequence. If the field represents a simple data type whose basic represention is *HLAinteger32BE*, for instance, then the method invokes `writeHLAinteger32BE`, passing it the `int` value of the field. The corresponding little-endian version of the method, `writeHLAinteger32LE`, reverses the byte ordering of the value. The encoding stream keeps track of the alignment of each element written, padding its output as necessary to conform to the HLA's alignment rules. Each of the HLA types listed in the tables above has a `read` method in `HLAEncodingInputStream` and a `write` method in `HLAEncodingOutputStream`.

3. Parameters

The proxy compiler accepts a number of parameters: command line parameters when invoked as a standalone application, task parameters when invoked as an Ant task. The only mandatory parameter is the location of the FDD to read as input. Other parameters include the directory in which to place the output files, the package name to associate with the output files, the name of the proxy ambassador class, and the name of the interaction listener interface. The following table lists the names of the parameters along with their default values. For the default values of the *proxyambassadorname* and *interactionlistenername* parameters, the proxy compiler massages the value of the FDD's object model name attribute into a Java identifier prefix by capitalizing each word and eliminating all whitespace and punctuation characters.

Parameter name	Default value
fdd	None (required parameter)
targetdirectory	Current working directory
packageprefix	None (default package)
proxyambassadorname	Object model name + "ProxyAmbassador"
interactionlistenername	Object model name + "InteractionListener"

Table 37. Proxy compiler parameters.

4. Output Files

a. Data Types

For each enumerated data type other than *HLAboolean*, for each fixed record data type, and for each variant record data type, the proxy compiler generates a Java source file. Each source file represents a Java class with the same name as its equivalent HLA type. For encoding and decoding data types, each class has a static `decode` method that reads and returns an instance of the type from an `HLAEncodingInputStream`, and a non-static `encode` method that writes the instance to an `HLAEncodingOutputStream`. Each class also has a no-argument default constructor and a copy constructor. The copy constructor initializes the object to be a copy of the constructor parameter, which must be of the same type as the class. Proxy compiler classes use the no-argument constructors to initialize instances to reasonable default values. The value of an enumerated type, for instance, defaults to the type's first enumerator. For fixed record types, the no-argument constructor initializes simple fields to Java defaults (`false` for boolean fields, `0` for integer fields, etc.), array fields to zero-length arrays, and object fields to new instances initialized with default constructors. Variant record types default to their first alternative, and their default constructors initialize each alternative value as if it were a fixed record field. Autogenerated enumerated type, fixed record type, and variant record type classes also override Java's default `toString` method to return string representations of their state.

Enumerated type state consists of a single value that corresponds to one of the type's enumerators. That value's Java type comes from the enumerated type's *representation* attribute. For each enumerator, the autogenerated enumerated type class contains a static member that federates can use directly to represent known values. For instance, the `HLAorderType` class, generated by the proxy compiler to represent the ROM's *HLAorderType* enumerated type, contains two static members: `HLAorderType.Receive` and `HLAorderType.TimeStamp`. The values of these variables are those specified by the *values* attributes of the FDD's *enumerator* elements. To encode and decode instances of the type, the `encode` and `decode` methods simply write and read, respectively, the enumerated type's representation value. When the value

corresponds to one of the enumerators, the `toString` method returns the enumerator's name; when the value is unknown, `toString` returns the value itself. The `equals` method compares the value of the type to that of the parameter, returning `true` if the values are equal and `false` if they are not. The `hashCode` method returns the integer equivalent of the representation value.

Fixed record types contain a number of fields, each with its own name, type, and semantics. For each field, the proxy compiler creates a member variable whose name is equal to that of the field, whose type is the Java equivalent of the field's type, and whose Javadoc comment is the field's semantics. To encode and decode instances of the type, the `encode` and `decode` methods write and read each field variable in sequence. Fixed record classes include constructors that accept initial values for all fields as their parameters. Also, for each field, the proxy compiler creates a pair of `get` and `set` methods for obtaining and modifying, respectively, the value of the field variable. The `toString` method returns string representations of each field.

Variant record types contain a named, typed discriminant and a number of named, typed alternatives. The value of the discriminant determines which of the alternatives is valid. To encode and decode variant records, the `encode` and `decode` methods write and read first the value of the discriminant, then the value of the active alternative that the discriminant identifies. Each variant record class includes a constructor that allows federates to specify the initial value of the discriminant, and `get` and `set` methods for the discriminant and for all alternatives. The `toString` method returns a string representation of the discriminant and the active alternative.

b. Proxy Ambassador

The proxy compiler creates a proxy ambassador class for each FDD. All proxy ambassadors are direct children of the `ProxyAmbassador` class included in the XRTI's utilities package along with the proxy compiler. `ProxyAmbassador` implements the HLA's `FederateAmbassador` interface in order to intercept callbacks generated by the RTI. Federates and XRTI components use instances of `ProxyAmbassador` directly as a means to broadcast RTI callbacks to multiple federate

ambassadors. The `registerFederateAmbassador` and `deregisterFederateAmbassador` methods add and remove federate ambassadors from the proxy ambassador's internal broadcast list. Through these methods, federates can use any number of proxy ambassadors at once, and can add or remove proxy ambassadors at any time.

The subclasses of `ProxyAmbassador` created by the proxy compiler perform two major functions: managing object instance proxies and allowing federates to send and receive interactions. The constructor of each proxy ambassador takes a reference to the `RTIambassador` as its parameter and initializes the instance by obtaining and storing handles for all supported object classes, interaction classes, and interaction parameters, as well as by publishing and subscribing all supported interactions. For each object class, the proxy compiler creates a `new` method that creates and returns a new instance of the corresponding object instance proxy class. This is the method that federates must use to create new locally-owned object instances. To create new proxies in response to discovered objects, proxy ambassadors capture the `discoverObjectInstance` callback and use its parameters to create proxy objects. Proxy ambassadors maintain a mapping between object instance handles and proxy objects in order to delegate callbacks such as `reflectAttributeValues` and `provideAttributeValueUpdate` to the appropriate object instance proxies. Federates can obtain proxies according to their object instance handles using the `getObjectInstanceProxy` method, and they can retrieve a list of references to all of the ambassador's proxies through the `getObjectInstanceProxies` method. The `ProxyAmbassador` class also allows objects implementing the `ProxyAmbassadorListener` interface to receive notifications concerning the creation and destruction of proxy objects by invoking the `addProxyAmbassadorListener` method. The corresponding `removeProxyAmbassadorListener` method removes the object from the proxy ambassador's notification list.

For each interaction class defined in the FDD, the proxy compiler creates a `send` method in the proxy ambassador that allows federates to send an interaction of

the described type with a single call. That method takes as its parameters Java versions of the parameters associated with the interaction class and all of its superclasses, as well as a user-supplied tag. The proxy compiler uses the semantics of the interaction as the method's Javadoc description. The body of the method populates an HLA `ParameterHandleValueMap` with the handles and encoded values of the interaction parameters, and calls the RTI ambassador's `sendInteraction` method with the interaction handle, the parameter handle value map, and the user-supplied tag. The proxy ambassador's `receiveInteraction` method checks the handles of received interactions against the handles of known interaction classes, decoding any interpretable interactions and sending them to objects that have registered as interaction listeners using the `addInteractionListener` method. That method and its counterpart, `removeInteractionListener`, take as their parameters instances of autogenerated listener interfaces containing methods corresponding to each interaction class.

c. Interfaces

For each FDD, the proxy compiler creates an interaction listener interface and a number of object instance and object instance listener interfaces. The interaction listener interface contains a `receive` method prototype for each interaction class with a Javadoc comment containing the class's semantics. The method's parameters include Java versions of the interaction parameters and a user-supplied tag, as above, as well as indications of the order and transportation types with which the interaction arrived. Object instance interfaces extend the interfaces generated for all of their superclasses, taking advantage of Java's support for multiple interface inheritance. For each attribute associated with the object class, the object instance interface includes a pair of `set` and `get` method prototypes. The `set` method accepts a Java version of the attribute's defined type as well as a user-supplied tag to associate with the update. The `get` method takes no parameters, and simply returns the Java value associated with the attribute. Object instance interfaces also include `addXListener` and `removeXListener` (where `X` is the name of the object class) method prototypes. These methods take as their parameters instances of the listener interface generated for the object class. These listener interfaces contain, for each attribute, an `xUpdated` (where `x` is the name of the

attribute) method prototype with a set of parameters that includes the source of the event (an instance of the corresponding object instance interface), the old value of the attribute, the new value of the attribute, the user-supplied tag associated with the update, the order type associated with the update, and the transportation type associated with the update.

d. Object Instance Proxies

The proxy compiler creates an object instance proxy class for each object class defined in the FDD. Each object instance proxy class extends the proxy of its corresponding class's direct parent (or the `ObjectInstanceProxy` utility class, if unparented) and implements the object instance interfaces created for that class and all of its superclasses. Instance proxy fields include a reference to the `RTIambassador`; the object instance handle; the object class handle; the name of the object; the auto-flush-disabled flag; the deleted flag; sets of listeners for attributes associated with each implemented interface; and the handle, dirty flag, and value of each attribute. Every instance proxy class includes three constructors: one for proxies created to represent discovered objects, one for locally created proxies with unspecified names, and one for locally created proxies with explicitly specified names. All of the constructors initialize the attributes by retrieving their handles, settings their values to valid defaults, and publishing and subscribing them through the RTI ambassador. The constructors for locally-owned objects also register the instances using the RTI ambassador's `registerObjectInstance` method.

The proxy ambassador calls the `reflectAttributeValues` and `provideAttributeValueUpdate` methods of the object instance proxy in response to callbacks from the RTI. For `reflectAttributeValues`, the object instance proxy iterates through the `AttributeHandleValueMap` provided, decoding the values of known attributes, storing them in their corresponding value fields, and notifying any object instance listeners registered for the class with which the attribute is associated. For `provideAttributeValueUpdate`, the object instance proxy identifies known attributes in the `AttributeHandleSet` provided, setting their associated dirty flags and calling the autogenerated `flushAttributeValues` method, which encodes the values of all dirty attributes, pairs them with their handles in an

`AttributeHandleValueMap`, clears their dirty flags, and calls the RTI ambassador's `updateAttributeValues` method. Federates can also invoke the `flushAttributeValues` method in a super-flush mode, causing it to update all of the object instance proxy's attribute values, whether or not they are dirty. For each attribute, the proxy compiler generates `set` and `get` methods conforming to the prototypes contained in the object instance interface. The `set` method modifies the attribute value, sets the attribute's dirty flag, and, if the auto-flush-disabled flag is not set, calls `flushAttributeValues` to notify the RTI of the update. The `get` method simply returns the attribute value.

Because Java objects cannot be explicitly deleted, each object instance proxy contains a `deleted` flag that indicates whether or not the proxy's corresponding object instance has been removed from the federation execution. For locally owned objects, calling the `delete` method sets the `deleted` flag and invokes the RTI ambassador's `deleteObjectInstance` method. Remotely owned objects become deleted when the proxy ambassador receives a `removeObjectInstance` callback, causing it to remove the object instance proxy from its internal list and to notify its listeners of the proxy's destruction.

B. XRTI AMBASSADOR

The XRTI Ambassador is the class that federates must instantiate and use in order to participate in XRTI federation executions. It conforms to the RTI ambassador interface defined by IEEE 1516.1 and implements the `mergeFDD` method described in section II.D of this thesis. To join a federation execution, federates must first invoke the ambassador's `initializeRTI` method, passing it a `java.util.Properties` object containing configuration properties expressed as mappings between string-valued names and textual values. After initializing the XRTI Ambassador, federates can create executions using the `createFederationExecution` method, specifying the execution name and the initial FDD, and join them using the `joinFederationExecution` method, passing the federate type, the execution name, the federate ambassador, and the mobile federate services. Once joined to an execution, federates can publish and subscribe object and interaction classes, send and receive

interactions, create and destroy object instances, update object attributes, and receive notifications in the form of federate ambassador callbacks when remotely owned objects are created, destroyed, and updated. When federates are ready to leave the execution, they call the `resignFederationExecution` method, and they can optionally invoke `destroyFederationExecution` to remove the execution if they are the last federate to leave.

1. Message Channels

Communication between the XRTI Ambassador and the XRTI Executive occurs through an abstract message channel interface that allows federates to plug in new types of channels at run time. Both the ambassador and the executive interpret the `message.channel.factory` configuration property, when specified, as the name of a concrete descendant of the abstract `MessageChannelFactory` class. `MessageChannelFactory` includes two method prototypes: `newChannelToExecutive` and `newFederateChannelAcceptor`. Both methods take as their parameter a `Properties` object containing configuration options. For channels created by the XRTI Ambassador, this is the same object passed by the federate to `initializeRTI`. The `newChannelToExecutive` method returns an instance of `MessageChannel`, the abstract superclass of all objects that transmit and receive messages with selectably reliable transportation. `MessageChannel` defines several method prototypes: `getMaximumPacketSize`, which returns the channel's packet size limit in bytes; `sendPacket`, which transmits the contents of its `DatagramPacket` parameter through the channel; `receivePacket`, which blocks until a packet is available, then places the received packet in a user-specified `DatagramPacket`; `getInputStream`, which returns the `java.io.InputStream` corresponding to the channel's reliable input; `getOutputStream`, which returns the `java.io.OutputStream` for reliable output; `close`, which closes the channel; and `isClosed`, which checks the channel's closed state.

If the `Properties` object passed to the XRTI Ambassador does not contain a mapping for the `message.channel.factory` option, the ambassador uses an

instance of `DefaultMessageChannelFactory`. This default class creates instances of `InternetMessageChannel` in response to `newChannelToExecutive` method calls, and instances of `InternetMessageChannelAcceptor` in response to `newFederateChannelAcceptor` calls. Internet message channels contain two components: a User Datagram Protocol (UDP) component for packet-based, best-effort messaging, and a Transmission Control Protocol (TCP) component for stream-based, reliable messaging. `DefaultMessageChannelFactory` interprets two configuration properties, `executive.host` and `executive.port`, in creating new message channels and channel acceptors. The `executive.host` property specifies the host name of the computer where the the XRTI Executive is running, and the `executive.port` property specifies the executive's port number. When a new channel to the executive is initialized, that channel opens any available UDP port, creates a TCP connection to the XRTI Executive, and writes its UDP port number though that connection. The executive, having accepting the connection, likewise opens a new UDP port and sends its port number through the TCP stream. After both sides have received each other's UDP port number, they can exchange datagram packets and streamed messages until the channel is closed.

2. Message Flow

After creating a channel to the XRTI Executive using the `MessageChannelFactory`, the XRTI Ambassador creates two threads for incoming messages: one for messages received in packets and one for messages received through the channel's input stream. When either of these threads receives a message, it passes it to the private `interpretReceivedMessage` method for dispatch. That method ensures that the message's protocol identifier and version number are equal to the expected values: 0xFEEDAFED and the BOM version number, respectively. It then uses the static `decode` method of the autogenerated `HLAbootstrapInteractionPayload` class to convert the encoded message into a directly interpretable object, verifies that the payload's federation execution handle is that of the joined execution, extracts the payload's interaction handle and parameters, and

invokes the `receiveInteraction` method of a `ProxyAmbassador` instance created upon initialization. That instance of `ProxyAmbassador` contains the `FederateAmbassador` passed to `joinFederationExecution` as well as proxy ambassadors for the BOM, MFOM, ROM, and MOM.

The XRTI Ambassador listens to both BOM and MOM interactions by implementing the autogenerated `BootstrapInteractionListener` and `ManagementInteractionListener` interfaces and registering with the BOM and MOM proxy ambassadors. When an *HLAinteractionFragment* is received, therefore, the BOM proxy ambassador calls the XRTI Ambassador's `receiveHLAinteractionFragment` method with the parameters of the interaction. The body of that method composes received fragments and sends completely formed interactions to the `interpretReceivedMessage` method. The `receiveHLAregisterObjectInstance` method invokes the proxy ambassador's `discoverObjectInstance` method, announcing the discovery of a new object instance. Similarly, `receiveHLArequestAttributeValueUpdate` invokes `provideAttributeValueUpdate` in the proxy ambassador, `receiveHLAupdateAttributeValuesBestEffort` and `receiveHLAupdateAttributeValuesReliable` invoke `reflectAttributeValues`, and `receiveHLAdeleteObjectInstance` invokes `removeObjectInstance`.

The XRTI Ambassador converts most interactions sent through its `sendInteraction` method directly into messages, prepending the protocol identifier and the BOM version number to an encoded instance of `HLAbootstrapInteractionFragment` and sending the resulting message through the `MessageChannel`. For best-effort interactions that exceed the channel's maximum packet size, however, the ambassador divides the interaction into a series of fragments and makes repeated calls to the BOM proxy ambassador's `sendHLAinteractionFragment` method, which in turn calls `sendInteraction` to send the fragments.

3. Obtaining Handles

After connecting to and establishing a message flow with the XRTI Executive in the `initializeRTI` method, but before returning control to the federate, the XRTI Ambassador must obtain a block of unique identifiers for it to use as handles. The interactions required for this task, and all other interactions associated with the meta-federation, have the number `0` as their federation execution handle. To acquire the handles, the XRTI Ambassador first sends an *HLArequestHandles* interaction through the BOM proxy ambassador. After requesting a block of 2^{32} handles, the XRTI Ambassador waits for the `receiveHLAreportHandles` callback and uses the result of that callback as the initial value for a handle counter. Every time the XRTI Ambassador needs a new handle for the `joinFederationExecution` or `registerObjectInstance` service, it uses the counter's current value, then increments the counter.

4. Service Mappings

Many of the services provided by the XRTI Ambassador correspond directly to interactions defined by the BOM, MFOM, ROM, or MOM. The following table maps the name of each service to its corresponding interaction class and the object model in which the class is defined. When federates invoke each service, the XRTI Ambassador transforms the method's parameters into those required by the interaction class and invokes the autogenerated `send` method that corresponds to that class in the appropriate object model proxy ambassador. These interactions are received and interpreted by the XRTI Executive.

Service name	Interaction class	Object model
createFederationExecution	HLAcreateFederationExecution	MFOM
destroyFederationExecution	HLAdestroyFederationExecution	MFOM
joinFederationExecution	HLAjoinFederationExecution	MFOM
resignFederationExecution	HLAresignFederationExecution	MOM
mergeFDD	HLAmergeFDD	ROM
publishObjectClassAttributes	HLApublishObjectClassAttributes	MOM
unpublishObjectClass	HLAunpublishObject-ClassAttributes	MOM
unpublishObjectClassAttributes	HLAunpublishObject-ClassAttributes	MOM
publishInteractionClass	HLApublishInteractionClass	MOM
unpublishInteractionClass	HLAunpublishInteractionClass	MOM
subscribeObjectClassAttributes	HLAsubscribeObject-ClassAttributes	MOM
subscribeObjectClassAttributes-Passively	HLAsubscribeObject-ClassAttributes	MOM
unsubscribeObjectClass	HLAunsubscribeObjectClass-Attributes	MOM
unsubscribeObjectClassAttributes	HLAunsubscribeObjectClass-Attributes	MOM
subscribeInteractionClass	HLAsubscribeInteractionClass	MOM
subscribeInteractionClassPassively	HLAsubscribeInteractionClass	MOM
unsubscribeInteractionClass	HLAunsubscribeInteractionClass	MOM
registerObjectInstance	HLAregisterObjectInstance	BOM
updateAttributeValues	HLAupdateAttributeValues	BOM
deleteObjectInstance	HLAdeleteObjectInstance	MOM
localDeleteObjectInstance	HLAlocalDeleteObjectInstance	MOM
changeAttributeTransportationType	HLAchangeAttribute-TransportationType	MOM
changeInteraction-TransportationType	HLAchangeInteraction-TransportationType	MOM
requestAttributeValueUpdate	HLArequestAttributeValueUpdate	BOM

Table 38. Service mappings.

5. Descriptor Manager

The XRTI Ambassador's descriptor manager tracks names, handles, and other information regarding object classes, attributes, interaction classes, parameters, object instances, dimensions, and regions. The ambassador initializes the descriptor manager by

populating it with the information contained in the BOM, MFOM, ROM, and MOM. For each of these base object models, the descriptor manager assigns handles to features sequentially, starting with `1`. Thus the handle of the BOM's first interaction class is `1`, the handle of its second interaction class is `2`, and so on, and the handle of the MFOM's first object class is one greater than the handle of the BOM's last interaction parameter. Assigning handles in this manner ensures that any two XRTI components with the same basic FDDs will use the same handles for features defined in the BOM, MFOM, ROM, and MOM. For federation-specific features, the descriptor manager uses the ROM proxy ambassador created by the XRTI Ambassador to obtain information concerning reflection objects managed by the XRTI Executive. When the descriptor manager detects the presence of a new object instance proxy, it creates and integrates a descriptor for the corresponding feature. That descriptor in turn listens to the instance proxy in order to reflect changes made to the reflection object. The XRTI Ambassador uses information maintained by the descriptor manager to respond to `getObjectClassHandle`, `getObjectClassName`, `getAttributeHandle`, `getAttributeName`, `getInteractionClassHandle`, `getInteractionClassName`, `getParameterHandle`, `getParameterName`, `getObjectInstanceHandle`, `getObjectInstanceName`, `getDimensionHandle`, `getDimensionName`, `getDimensionUpperBound`, `getAvailableDimensionsForClassAttribute`, `getKnownObjectClassHandle`, and `getAvailableDimensionsForInteractionClass`.

C. XRTI EXECUTIVE

The XRTI Executive is a standalone application that hosts federation executions, tracking their participants and object models and controlling the distribution of information between federates. Users starting the executive on the command line can provide configuration properties equivalent to those required by the XRTI Ambassador's `initializeRTI` method using the `-C` and `-configuration` arguments. Arguments of the form `-C<name>=<value>` set single configuration properties, whereas arguments of the form `-configuration <filename>` load configuration

files containing multiple property mappings. As with the XRTI Ambassador, the `message.channel.factory` property determines the subclass of `MessageChannelFactory` that the XRTI Executive uses, and the executive passes all configuration properties to the `newFederateChannelAcceptor` method. After starting up, the XRTI Executive begins to accept connections from remote federates, creating an `ExecutiveClientAmbassador` instance for each `MessageChannel` returned by the `MessageChannelAcceptor`. As clients establish new federation executions, the executive creates instances of `FederationExecutionAmbassador` to manage them.

1. Message Channel Acceptors

The base class `MessageChannelAcceptor` provides an abstraction layer for server objects, much as `MessageChannel` acts as an abstraction layer for communication channels. After creating a `MessageChannelAcceptor` by invoking the `newFederateChannelAcceptor` method of `MessageChannelFactory`, the XRTI Executive uses the acceptor's `acceptMessageChannel` method to wait for and return the next incoming connection, as represented by a `MessageChannel`. The acceptor's `close` method releases its resources.

Instances of `InternetMessageChannelAcceptor` returned by the `DefaultMessageChannelFactory` class open TCP server sockets on the port identified by the `executive.port` configuration property. When the executive calls `acceptMessageChannel`, they block until a connection is made and return an `InternetMessageChannel` representing the connected socket.

2. Executive Client Ambassador

The `ExecutiveClientAmbassador` class is a protected subclass of `XRTIAmbassador` whose instances represent the remote counterparts of federates' XRTI Ambassadors. After being initialized with a reference to the owning `XRTIExecutive` and the `MessageChannel` connected to the remote federate, the executive client ambassador receives and processes interactions sent by the federate, contacting the XRTI Executive and instances of

`FederationExecutionAmbassador` when necessary. For instance, on receiving a `receiveHLArequestHandles` invocation from the BOM proxy ambassador, the executive client ambassador calls the XRTI Executive's protected `acquireHandles` method to reserve a block of handles from the executive's global list, then sends an `HLAreportHandles` interaction containing the block parameters to the remote federate. In addition to BOM and MOM interactions, the executive client ambassador subscribes to MFOM and ROM interactions by implementing the `MetaFederationInteractionListener` and `ReflectionInteractionListener` interfaces and registering with the MFOM and ROM proxy ambassadors.

For the BOM's *HLAregisterObjectInstance*, *HLArequestAttributeValueUpdate*, *HLAupdateAttributeValuesBestEffort*, and *HLAupdateAttributeValuesReliable* interactions, the executive client ambassador relays the information to the federation execution ambassador corresponding to the joined execution. It does likewise for the MOM's *HLAdeleteObjectInstance* interaction. For the MOM's publication and subscription interactions, the executive client ambassador simply stores the handles of the published and subscribed interaction classes and object class attributes, and uses them to filter the interactions and attribute updates that the federation execution ambassador relays from other federates. The MFOM's *HLAcreateFederationExecution*, *HLAdestroyFederationExecution*, *HLAjoinFederationExecution*, the MOM's *HLAresignFederationExecution*, and the ROM's *HLAmergeFDD* all cause the executive client ambassador to contact its owning XRTI Executive in order to create, destroy, register with, deregister with, or update federation execution ambassadors.

3. Federation Execution Ambassador

Like `ExecutiveClientAmbassador`, `FederationExecutionAmbassador` is a specialized subclass of `XRTIAmbassador`. The XRTI Executive creates one `FederationExecutionAmbassador` for each federation execution and uses it to maintain that execution's management and reflection objects as well as to broadcast interactions and attribute value updates sent by federates. One of the federation

execution ambassador's most important methods is `createReflectionObjects`, which takes a parsed FDD document as its parameter and uses it to create or update a set of reflection objects corresponding to the features described by the FDD's object model: object classes and their attributes, interaction classes and their parameters, dimensions, synchronizations, transportations, and data types. These reflection objects, created using the ROM proxy ambassador and maintained as proxies, represent a dynamic version of the federation object model that federates can extend using the `mergeFDD` method of their `XRTIAmbassador` instances. The XRTI Executive calls each federation execution ambassador's `createReflectionObjects` method once when it creates the execution, and again whenever an `ExecutiveClientAmbassador` receives an *HLAmergeFDD* request.

Federation execution ambassadors maintain lists of `ExecutiveClientAmbassador` instances corresponding to federates joined to their executions. When a federate transmits an interaction, an attribute value update, or an object creation or deletion event, its executive client ambassador forwards the message to the federation execution ambassador, which broadcasts it to all registered client ambassadors. The client ambassadors then determine whether to forward the message to their connected federates according to their subscription parameters. Execution ambassadors similarly track the names, handles, classes, and owners of all registered objects, so that they can notify late joiners and late subscribers of the objects' presence and forward attribute update requests directly to the objects' owners.

THIS PAGE INTENTIONALLY LEFT BLANK

V. INTEGRATION INTO NPSNET-V

A. PLATFORM OVERVIEW

NPSNET-V is a component-based, dynamically extensible platform for networked virtual environment applications: clients, servers, peers, and standalone products [NPSNET 03, *NPSNET-V*]. Like other virtual environment toolkits, NPSNET-V includes modules for graphical rendering, user input, networking, and related tasks. Unlike traditional toolkits, however, whose features are either restricted to those included by their authors at the time of compilation, or which support limited run-time extension through plug-in interfaces, NPSNET-V allows applications to extend or upgrade virtually any aspect of its functionality at run-time. The vision behind dynamically extensible architectures such as NPSNET-V is one of near-infinite reconfigurability. As an example, consider a shared virtual world based on the NPSNET-V platform in which each piece of the environment—every building, every vehicle, and every humanoid—is the product of an independent code module. Developers insert new elements into the environment by writing new code modules, uploading them to publicly accessible Web servers, and instantiating them in the shared world. When users visit the world, their clients automatically download the code modules corresponding to the world's contents and plug them into the client-side application environment. Once activated, the newly downloaded modules become integral parts of the client, performing duties such as rendering entities onscreen, calculating physical forces, transmitting and receiving network updates, and processing mouse or keyboard input. As the world evolves, or as the user enters new regions of the world, the client upgrades existing modules, incorporates new modules, and unloads redundant modules as necessary to reflect the state of the shared world. Taken to the extreme, this scenario reaches the point at which every part of the client application—aside from a minimal microkernel—is downloaded and integrated dynamically. Such is the ideal underlying NPSNET-V's design.

1. Component Framework

Achieving that ideal, however, requires that NPSNET-V be more than a traditional toolkit or library for developers to link into their existing applications. Instead, NPSNET-V provides a framework into which application developers must place components, or modules: the atoms of the dynamic extension process. A typical application might combine modules supplied with the NPSNET-V distribution with third-party modules and modules written specifically for the application by its developer. Each module is an instance of a Java class that inherits from `org.npsnet.v.kernel.Module`, an abstract base class included in the NPSNET-V kernel package. That package also includes the `ModuleContainer` and `Kernel` classes. Modules whose classes are derived from `ModuleContainer` can contain other modules, much as directories contain files in hierarchical file systems. The `Kernel`, which is itself a `ModuleContainer`, acts as the equivalent of a root directory. Each `Module` has a local name which distinguishes it from other modules in the same container and a global path that identifies its position in the module hierarchy. For instance, the absolute path `/moduleContainer/module1` identifies a module with the local name `module1` contained within a module container named `moduleContainer` that is in turn contained within the kernel. As in file systems, relative paths such as `../module1` identify modules relative to a local context.

a. Module Life Cycle

After being instantiated as an object, each module undergoes a life cycle whose phase transitions are controlled by the module's container. After setting the module's local name and other properties, the container calls the module's `init` method to initialize the module. If the module implements the `Startable` interface, the container also invokes the module's `start` method, causing the module to create new threads of execution or otherwise activate itself. In cases where modules must be hot-swapped—that is, replaced with newer versions of themselves while they are still running—the container coordinates a hand-off operation that involves calling the `replace` method of the new module and the `retire` method of the old module, allowing both modules to play an active role in the replacement process. The `stop`

method causes modules to destroy any threads of execution that they have created, and the `destroy` method returns modules to their pre-initialized state.

b. Interface Layer

In order for modules to communicate with one another directly using Java method calls, they must share a common set of base classes or interfaces. The classes included in the kernel package are common to all modules, but their methods are limited to those necessary for system management. For application-specific interactions, NPSNET-V provides an extensible interface layer that consists of all interfaces whose archives are listed as dependencies by loaded modules. Because interfaces, unlike modules, cannot be unloaded or replaced once they become part of the system, module developers must exercise discipline in defining and publishing new interfaces. If, for example, a published interface called `ExampleInterface` must be modified to include a new method, its authors can only do so by extending it to create a new interface, `ExampleInterfaceEx` or `ExampleInterface2`. The NPSNET-V distribution includes two interface archives, `properties.jar` and `services.jar`, whose contents represent the platform's basic interface library. Property interfaces, which are those that inherit from `org.npsnet.v.kernel.PropertyBearer`, and service interfaces, which derive from `org.npsnet.v.kernel.ServiceProvider`, perform special roles in the NPSNET-V platform. For instance, the `getPropertyBearers` method of `ModuleContainer` allows modules to retrieve lists of all modules within the container that bear a given property. Similarly, the `getServiceProvider` method returns the provider of a specific service, first checking the local container, then the parent container, and so on up the module tree until a provider is found. Service providers, then, are modules that provide specific kinds of functionality to entire application subgraphs.

c. Configuration and Serialization

NPSNET-V's configuration and serialization mechanism allows applications to record module hierarchies as version-safe XML documents suitable for long-term storage and reactivation and for transmission between heterogeneous instances

of the NPSNET-V environment. NPSNET-V supports two types of serialized representations: configurations, which represent the states of individual modules, and prototypes, which represent the modules themselves. Prototypes, which include modules' configurations as well as their class and dependency information, can be used to stamp out multiple copies of modules, whereas configurations can only be applied to existing modules. The default implementations of `Module`'s `getConfiguration` and `applyConfiguration` methods use the extensible `PropertySerializationProvider` and `ConfigurationElementInterpretationProvider` services, respectively, to generate and interpret XML configuration elements corresponding to known properties. For instance, the default property serialization provider generates a `Transform` element for modules implementing the `org.npsnet.v.properties.model.Transformable` property with `translation`, `rotation`, and `scale` attributes. For prototypes, each module stores a number of objects that represent run-time dependencies, such as the module's reliance on a particular service. The `getPrototype` method of the `Module` class serializes these dependencies into XML elements, and `ModuleContainer`'s `createModule` method, which creates a new module based on a prototype, turns the dependency elements back into objects and resolves them before instantiating the described module.

d. Bootstrapping and Extension

The typical course of execution for an NPSNET-V application begins with a standard bootstrapping process initiated by the kernel. Unless directed otherwise, after instantiating itself as a module in its static `main` method, the kernel loads a bootstrap configuration file included with the NPSNET-V distribution. This file contains a single configuration element that instructs the kernel to load the `org.npsnet.v.resource.StandardResourceManager` module from the `resource.jar` file. Upon initialization, the resource manager scans the local NPSNET-V archive directory and creates a list of all available resources, including their names, versions, and any metadata associated with them through their Java archive (jar) manifests. The resource manager also attempts to establish connections with one or more

Lightweight Directory Access Protocol (LDAP) resource servers. The entries in these servers' databases correspond to resources that developers have published for public use. When a module requests a resource by name, the resource manager searches its local database as well as any connected LDAP servers and returns the location of the latest version. After loading the bootstrap configuration, the NPSNET-V kernel processes any `-configuration` or `-prototype` arguments on its command line. These arguments cause the kernel to apply configurations and instantiate prototypes, respectively, in order to extend the framework. To load the NPSNET-V console, for instance, one uses `-prototype resource:///org/npsnet/v/applications/console.xml` to load the latest version of the console prototype. After processing the command line arguments, the kernel relinquishes control to the modules that it has loaded. If an application must load other modules during the course of execution, it does so through the API provided by the `Module` and `ModuleContainer` classes. Any module can, for instance, extend the framework at any time by loading a prototype through the `createModule` method of its container.

2. Entity Model

The NPSNET-V entity model, which represents virtual worlds and entities in terms of modules in the framework, is based on the Model-View-Controller (MVC) design pattern [Gamma 95]. The MVC pattern requires that the abstract state, or model, of each entity be stored within a module that is separate from the modules used to depict that state to the user, known as views, and from the modules used to manipulate that state, called controllers. Entities within the NPSNET-V platform consist of a single model, any number of loosely coupled views and controllers, and a scaffold module that creates and destroys views, controllers, and sub-scaffolds according to changes in the state of the application framework, to the information stored within the model, or to the scaffold's own internal policies.

a. Models

Models are the central modules of NPSNET-V entities. They contain not only the entities' physical states, such as their positions and orientations, but also serialized prototypes of views and controllers. The prototype of an entity model, therefore, is a completely self-contained representation of a virtual entity, suitable for instantiation in any context. The entity model classes contained in the NPSNET-V distribution include generic entity, object, and world models suitable for direct use or as base classes for extension; camera models; environment models; and terrain models. Like other modules, entity models can be arranged in containment hierarchies. World models, for instance, can contain entity models, and those entity models can contain other entity models. The containment relationships between models affect the nature and scope of model operations much as scene graph structures determine the rendering behavior of high-level graphics APIs.

b. Views

View modules present the state of their target models to the user, typically by interfacing with just such a graphics library. The hierarchy of view modules tends to reflect that of the views' corresponding models, so that a world model is associated with an independent core view module, and the children of that world model are associated with views contained in the view core. The view core creates and maintains the principal interface to the graphics library, while the child views manage only the resources required by their model representations. In the case of scene graph APIs, this means that the view core creates the root of the scene graph and the child views create and control nodes underneath that root. When the models change, they issue events that their views interpret and use to modify their nodes. For immediate mode APIs, view cores request once per frame that each child view read the state of its target model and render itself accordingly. The NPSNET-V distribution includes modules for Java 3D, Java 2D, OpenGL, and text views.

c. Controllers

Controllers manipulate the state of their target models according to user input, network updates, simulated physical interactions, or other stimuli. Like views, controllers reside in hierarchies that reflect those of their target models, with controller cores managing device interfaces and child controllers interpreting data specific to their targets. The NPSNET-V distribution includes mouse and keyboard controllers; physics controllers; and network controllers for DIS, HLA, XFSP, and XRTI communication. The network controller modules are unique in that they must create ghost entities to represent remote masters controlled by other clients on the network. When network controller cores discover a new entity, they use the `org.npsnet.v.services.system.EntityTypeMappingProvider` service to obtain an NPSNET-V entity model prototype suitable for representing the entity locally. The cores then add the entity to the framework by instantiating the prototype in their target containers.

d. Scaffolds

When a new entity model appears within a model container, the scaffold of that container creates a sub-scaffold module to manage the new entity's views and controllers. Scaffold modules create, update, and destroy view and controller modules according to the prototypes stored within their target models, the views and controllers associated with the model's parent, and their own configurable policies. The view and controller prototypes associated with each model include mode filters: boolean expressions that act on sets of mode flags. For instance, prototypes bearing the filter `debug&(!ghost)` apply only to models in debug mode that do not represent ghost entities. The default view and controller policies will instantiate a prototype only if an appropriate context exists and the mode filter of the prototype agrees with the mode flags of the target.

3. Application Structure

a. Test Applications

The NPSNET-V distribution supplies a number of test applications in the form of configurations to be applied to the kernel. Each test provides an independent demonstration of a particular module or category of functionality. The `org/npsnet/v/applications/tests/hla_networking.xml` test, for instance, acts as a simple test of HLA networking. It loads two sub-configurations—`hla_networking_a.xml` and `hla_networking_b.xml`—that act as independent clients running side-by-side in separate windows. After loading a series of service providers, each client creates a simple world model containing a camera, a light, and a teapot object. It then creates a scaffold module for the world model, instantiates a Java 3D view within the scaffold, and opens a window for rendering. To add interactivity to the scene, each client creates a physics controller, a user interface controller, and an HLA controller for the world. Each teapot's scaffold module then loads a simple physics controller that integrates the teapot's position and orientation over time according to its velocity and acceleration; a mouse controller that allows the user to move the teapot by clicking on it and dragging the mouse pointer; and an HLA controller that transmits updates concerning the teapot's state to the RTI. Once both clients have loaded, each discovers the other's teapot entity and creates a corresponding ghost entity. The final result is one of two onscreen windows, each with two teapots, where moving a teapot in one window causes its ghost in the other window to move in an identical fashion.

b. Browser Environment

In addition to its test applications, the NPSNET-V distribution includes an extensible browser environment [Kapolka 03]. This environment allows its users to browse existing shared virtual worlds, build worlds of their own, publish their worlds to LDAP resource servers, and host their worlds within the NPSNET-V framework. Its principal interface closely resembles a standard Web browser, including navigation buttons, an address bar, a bookmark menu, and a preference manager. The browser's

collection of content handlers allows it to display images, VRML models, HTML pages, and other kinds of data in addition to NPSNET-V configurations and prototypes. Entering a URL beginning with `module:/` into the address bar causes the browser to display a tree view of modules loaded within the NPSNET-V framework, allowing users to add, remove, or otherwise manipulate loaded modules. Similarly, URLs beginning with `resource:/` display sections of the resource tree. From there, users can view resource metadata and select resources to display. When a user selects a world prototype, the browser loads the world into the framework, creates a default view for it, and shows that view in the browser window. The user can then create new entities within the world, manipulate them using controllers or context menus, and publish the world by uploading its prototype to a Web server and publishing its metadata to a resource directory. The browser environment also includes HTTP and Telnet server modules that let users host worlds within the environment. The HTTP prototype server, for instance, allows remote clients to instantiate prototypes using the HTTP PUT method, to retrieve prototypes with the GET method, to modify modules using the POST method, and to remove modules using the DELETE method. The Telnet console server allows users to log into the NPSNET-V console from remote sites in order to observe and manipulate the state of the framework. Together, these modules provide the means to establish unattended world server sites.

B. HLA CONTROLLERS

NPSNET-V's HLA controller modules exchange entity state using the older, 1.3 version of the HLA standard as opposed to the IEEE 1516 version. Most commercial RTIs conform to this older standard, which differs from the newer one in several minor ways. Its data files, for instance, use an S-expression syntax as opposed to an XML dialect. Also, it uses integers rather than type-safe objects for feature handles. NPSNET-V's HLA controllers rely on the Realtime Platform Reference Federation Object Model (RPR-FOM), and have been tested with both the DMSO RTI-NG 1.3v6 [DMSO 03, *RTI*] and the MÄK RTI v2.0.3 [MÄK 03].

1. HLAControllerCore

The `HLAControllerCore` module is the root of the HLA controller hierarchy, and as such it is responsible for creating and managing the interface to the RTI and for creating entities in response to the discovery of new object instances. In its constructor, the controller core creates an `RTIambassador` instance. In its `init` method, it attempts to create a federation execution with the name "NPSNET-V" using a federation description document that contains the RPR-FOM. If the RTI succeeds in creating the execution, or if the execution already exists, the controller core then joins the execution (passing a reference to itself as federate ambassador in order to receive callbacks from the RTI), fetches the handles of the features that it must use, and publishes and subscribes to the interaction classes and object class attributes that it must generate and interpret. When the RTI notifies the controller core of a new object instance through the callback interface, the controller core waits until it has received the object's entity type, then consults the `EntityTypeMappingProvider` service to obtain an appropriate ghost prototype. After instantiating the ghost, the controller core waits for the appearance of the ghost's `HLAController` module, then configures that controller with information received from the RTI.

2. HLAController

The `HLAController` property interface allows the controller core to interact with its children, HLA controllers that correspond to master and ghost entities within the world. For master entities, the controller core calls the `registerObjectInstance` method to request that the controller register its corresponding object instance with the RTI ambassador, then calls `getObjectHandle` to retrieve the handle of that instance. For ghost entities, the controller core passes the handle of the discovered object instance to the controller's `setObjectHandle` method. When the controller core receives attribute value updates, update requests, or object instance removal messages, it forwards the message to the appropriate `HLAController` based on its internal list of instance handle mappings. Once per second, the controller core also calls the `generateHeartbeat` method of all of its contained controllers, causing them to emit their complete state for the benefit of late-joining federates.

3. HLAPlatformController

The `HLAPlatformController` module is an implementation of the `HLAController` interface for platform entities. It reads the state of its target entity through the `Transformable`, `Inertial`, and `Accelerable` property interfaces, and transmits that state through the *WorldLocation*, *Orientation*, *VelocityVector*, *AngularVelocityVector*, and *AccelerationVector* attributes of the RPR-FOM *BaseEntity* object class. In order to minimize network traffic, `HLAPlatformController` uses a dead-reckoning algorithm to extrapolate position and orientation over time. On the sending side, the master controller extrapolates the position of the model over time and only transmits an update when the entity deviates from its predicted course. On the receiving side, the ghost controller extrapolates the position of the entity based on the last set of values received. When the ghost controller receives a new set of values, rather than abruptly changing the position of its target model, it causes the model to converge smoothly upon the updated course. This behavior helps to preserve the user's sense of realism and immersion by avoiding sudden, incongruous movements.

C. XRTI CONTROLLERS

NPSNET-V's XRTI controllers closely resemble its HLA controllers, but they differ in that they use the XRTI's utility and proxy classes in addition to the HLA API defined by IEEE 1516.1. Before it compiles the XRTI controllers, NPSNET-V's build script uses the XRTI's `CompileProxiesTask` to generate a set of proxies corresponding to the contents of the RPR-FOM. From the 11,281 line long RPR-FOM FDD, the proxy compiler generates 176 classes with 21,499 non-comment source statements and 2,049 lines of Javadoc comments. The XRTI controllers use these autogenerated proxy classes to interact with the federation execution in terms of the RPR-FOM.

1. XRTIControllerCore

The `XRTIControllerCore` creates an instance of `XRTIAmbassador` in its constructor and, in its `init` method, uses that ambassador to create and join a federation execution. As with `HLAControllerCore`, the XRTI controller core attempts to create

an execution with the name “NPSNET-V” and the RPR-FOM FDD. After creating an instance of `ProxyAmbassador` and passing a reference to it to the `XRTIAmbassador`’s `joinFederationExecution` method, the `XRTIControllerCore` creates an instance of `RPRProxyAmbassador` and registers it with the proxy ambassador in order to send and receive interactions and attributes defined in the RPR-FOM. Having registered as a `ProxyAmbassadorListener` with the `RPRProxyAmbassador`, the `XRTIControllerCore` receives a notification whenever that ambassador creates a new object instance proxy in response to the discovery of a remote object instance. When that happens, the controller core adds itself as a listener to the proxy in order to determine its entity type. Once it has received the required type information, it uses the `EntityTypeMappingProvider` to obtain an appropriate corresponding prototype, then instantiates that prototype within its target container.

2. XRTIController

When the prototype’s XRTI controller module appears, the controller core uses the `setRTIAmbassador` and `setProxyAmbassador` methods defined in the `XRTIController` property interface to set the controller’s references to the `XRTIAmbassador` and the `RPRProxyAmbassador`, respectively. If the controller corresponds to a remote ghost, then the controller core uses the `setObjectInstanceProxy` method to set the controller’s proxy reference to the proxy that was automatically created by the `RPRProxyAmbassador`. Otherwise, if the controller targets a locally owned entity model, then the controller core calls the `createObjectInstanceProxy` method to request that the controller use its reference to the `RPRProxyAmbassador` to create an appropriate proxy object. Like `HLAController`, `XRTIController` has a `generateHeartbeat` method that the controller core calls once per second. XRTI controllers can emit their complete state easily by invoking the `flushAttributeValues` methods of their object instance proxies with the `superFlush` parameter set to `true`.

3. XRTIPlatformController

Like `HLAPlatformController`, `XRTIPlatformController` is an XRTI controller for platform entities that uses dead-reckoning with smoothing to limit bandwidth usage while retaining first-order continuity of motion. For local master entities, `XRTIPlatformController` compares the position of its target to the remotely predicted position, calling the `emitEntityState` method when the difference exceeds a set threshold. That method updates the position of the object instance proxy by calling its `setWorldLocation`, `setOrientation`, `setVelocityVector`, `setAngularVelocityVector`, and `setAccelerationVector` methods, then calling its `flushAttributeValues` method to transmit a message containing all of the updated values. For remote ghost entities, the XRTI platform controller adds itself as a `BaseEntityListener` to its object instance proxy, updating its predicted position whenever it receives a state change notification through its `WorldLocationUpdated`, `OrientationUpdated`, `VelocityVectorUpdated`, `AngularVelocityVectorUpdated`, or `AccelerationVectorUpdated` callback methods.

D. INTEGRATION SUMMARY

Integrating the XRTI into NPSNET-V, a dynamically extensible component architecture for networked virtual environments, requires the presence of a set of XRTI controller modules. These modules, which are closely based on NPSNET-V's existing HLA controller modules, use constructs defined in the RPR-FOM to exchange state data concerning the virtual world and entity models that they control. NPSNET-V's Ant build script invokes the XRTI proxy compiler in order to turn an XML FDD into a complete set of RPR-FOM proxy classes. The XRTI controller modules use these autogenerated proxies along with the XRTI library to communicate with other federates. For platform entities, the controller modules use dead-reckoning with first-order smoothing to transmit position, velocity, and acceleration attributes. With the development of the XRTI prototype and the integration process complete, the next step is to test the XRTI both independently and within the context of an NPSNET-V environment.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. TESTING

A. TEST APPLICATIONS

1. HelloWorld

To demonstrate its basic functionality, the XRTI includes a test application, `org.npsnet.xrti.tests.HelloWorld`, modeled after the one included with the DMSO RTI. Like DMSO's, the XRTI's `HelloWorld` represents a very simple distributed simulation of population growth. The application uses a minimal object model, included as `HelloWorldObjectModel.xml`, to communicate information concerning a number of countries. That object model contains one object class, *Country*, with two attributes: *name*, a string, and *population*, a double-precision floating point variable. It also contains an interaction class, *Communication*, with a single parameter: a string-valued *message*. The XRTI's build script compiles `HelloWorldObjectModel.xml` into a set of proxies for the `HelloWorld` application to use.

The application itself takes three parameters: the name of a country, the country's initial population, and the number of time steps to simulate. On startup, `HelloWorld` creates and initializes an instance of `XRTIAmbassador`, attempts to create a federation execution called "HelloWorld" with `HelloWorldObjectModel.xml` as its FDD, creates a `ProxyAmbassador` instance and passes it to the RTI ambassador's `joinFederationExecution` method, and creates and registers an instance of `HelloWorldProxyAmbassador` in order to transmit and receive messages associated with the HelloWorld object model. After adding itself to that proxy ambassador as an interaction listener, `HelloWorld` uses the ambassador's `newCountry` method to create a new instance of `CountryProxy` and thus a new locally-owned object instance. After calling the proxy's `setName` and `setPopulation` methods in order to set the country's name and initial population to the values passed on the command line, `HelloWorld` enters its main simulation loop. For the number of time steps requested, `HelloWorld` increases the country's

population by one percent, prints out the names and populations of all country proxies tracked by the `HelloWorldProxyAmbassador`, and waits for a half-second before performing the next iteration. At every tenth time step, `HelloWorld` also invokes the `sendCommunication` method of `HelloWorldProxyAmbassor`, transmitting the message "Hello from [country name]" to all of the other federates. After completing the requested number of time steps, `HelloWorld` resigns from the federation, deletes the execution if it is the last federate to leave, and finalizes its RTI ambassador.

Below is a sample of the output generated by an instance of `HelloWorld` invoked with the command line parameters `USA 10 500`, run in conjunction with another instance whose command line parameters were `UK 10 500`.

```
XRTIAmbassador: Connected to executive
XRTIAmbassador: My handles start at 8589934592
XRTIAmbassador: Created execution HelloWorld
XRTIAmbassador: Joined execution HelloWorld (4294967296)
USA 10.1

Sent communication: Hello from USA

USA 10.201

USA 10.30301

USA 10.4060401

USA 10.510100501

USA 10.615201506010001

Got communication: Hello from UK
UK 10.1
USA 10.721353521070101

UK 10.201
USA 10.828567056280802
```

Figure 4. Sample output from the `HelloWorld` test application.

2. HelloWorldEx

The supplementary `HelloWorldEx` test application demonstrates the XRTI's support for extensible object models. It is nearly identical to the `HelloWorld` application, but its object model includes two new classes: *EconomicCountry*, which extends *Country* to include a double-valued *grossDomesticProduct* attribute, and *PrioritizedCommunication*, which extends *Communication* with the double-valued

priority parameter. `HelloWorldEx` takes the initial gross domestic product (GDP) as another command line parameter, to be included between the initial population and the number of time steps to execute. After joining the "HelloWorld" federation execution, instances of `HelloWorldEx` merge the extended `HelloWorldObjectModelEx.xml` object model into that execution's FOM. When they do so, they gain the ability to create instances of *EconomicCountry* and to send *PrioritizedCommunication* messages, even if the execution was originally established by an instance of `HelloWorld` using the unextended `HelloWorldObjectModel.xml` as its FDD. After performing the merge, instances of `HelloWorldEx` create *EconomicCountry* objects and simulate the growth of both their populations and their GDPs over time. Instead of *Communication* interactions, `HelloWorldEx` instances send *PrioritizedCommunication* interactions with random priority values at every tenth time step.

The sample output below results from running an instance of the `HelloWorldEx` application with parameters `USA 10 100 500` immediately after starting an instance of the `HelloWorld` application with parameters `UK 10 500`.

```
XRTIAmbassador: Connected to executive
XRTIAmbassador: My handles start at 17179869184
XRTIAmbassador: Created execution HelloWorld
XRTIAmbassador: Joined execution HelloWorld (4294967296)
USA 10.1 101.0

Sent communication with priority 0.01: Hello from USA

UK 10.828567056280802
USA 10.201 102.01

UK 10.93685272684361
USA 10.30301 103.0301

UK 11.046221254112046
USA 10.4060401 104.060401

Got communication: Hello from UK
UK 11.156683466653167
USA 10.510100501 105.10100501
```

Figure 5. Sample output from the `HelloWorldEx` test application.

B. THESIS EXPERIMENT

1. Overview

The purpose of the thesis experiment is to verify that the XRTI can provide a small-scale networked virtual world with a communications infrastructure whose performance is comparable to that of two commercial RTIs. The test world consists of a very simple NPSNET-V configuration, similar to the HLA networking test described in section V.A.3.a, that contains a shark swimming in a circle around the origin. The experiment involves invoking two instances of the client on separate machines, where each client creates a shark entity of its own as well as a Java 3D window that displays a top-down view of both sharks. For each of the three RTIs—the XRTI and the two commercial RTIs—the experiment involves recording over a fixed interval four performance metrics: average frame rate, average interaction latency, CPU usage, and network traffic volume. To capture the first two metrics, the experiment requires slight changes to the NPSNET-V codebase. For the second two, third party software tools are sufficient. The hypothesis of the thesis experiment states that the XRTI provides a level of performance that is equal to or better than that of the commercial RTIs in all four metrics.

2. Setup

a. Hardware

The hardware required for the thesis experiment consists of two computers connected by a 100 Mbps local area network (LAN). Computer A has a 1 gigahertz Intel Pentium III processor, 512 megabytes of system memory, and a GeForce FX 5900 Ultra graphics card with 256 megabytes of video memory. Computer B has a 2 gigahertz Intel Pentium IV processor, 512 megabytes of system memory, and a GeForce GTS graphics card with 64 megabytes of video memory.

b. Software

The software involved in the thesis experiment consists of the Microsoft Windows operating system, the Java Runtime Environment (JRE) by Sun Microsystems,

a version of NPSNET-V modified to measure average frame rate and interaction latency, the shark world configuration, the RTIs to be tested, and the testing utilities. Both Computer A and Computer B have Windows 2000 and the 1.4.1 version of the JRE installed.

The version of NPSNET-V used in the thesis experiment contains a modified version of the `J3DViewCore` module that reports on request its average frame rate over an interval of time. When the user presses the 'f' key, the `J3DViewport` class associated with the Java 3D view core module records the current time and clears the value of its frame count variable. Each time the viewport renders a frame, it increments the frame count. When the user presses the 'f' key again, the viewport computes the average frame rate by dividing the number of frames rendered by the amount of time elapsed, then prints the result to the standard output device.

The versions of `HLAControllerCore` and `XRTIControllerCore` used in the thesis experiment measure average interaction latency over time by exchanging special ping interactions with other federates. The versions of the RPR-FOM used by these modules contain two extra interaction classes, *ReliablePing* and *BestEffortPing*, each with two long-valued parameters: *PingID* and *PingTime*. Once per second, each controller core transmits both a reliable ping and a best-effort ping. Both pings include as their parameters a randomly generated identifier and the current time in nanoseconds as generated by the high-resolution `J3DTimer` class. After sending each ping, the controller cores record the ping's identifier. When the cores receive pings from other federates, they check the pings' identifiers against their recorded lists. Pings with identifiers that are not on the list are retransmitted exactly as received. Retransmitted pings with identifiers that are on the list—that is, pings that have traveled to another federate and back—cause the cores to increment their reliable or best-effort ping counts and to add the pings' round-trip intervals to their accumulated reliable or best-effort ping times. When the user presses the 'f' key, the controller cores compute the average round-trip times for reliable and best-effort pings by dividing the total number of pings in each category by their corresponding time totals. They then print the resulting metrics to the standard output device.

The thesis experiment requires two versions of the shark world configuration: one for the commercial HLA RTIs and one for the XRTI. Like the HLA test configuration described in section V.A.3.a, both versions load a series of service providers; a simple world model with a light, a camera, and an entity; a scaffold for the world model; a Java 3D view core with a single onscreen window; a physics controller core; and a user interface controller core. Rather than a user-controlled teapot, however, the entity in the thesis experiment configuration is a shark whose `SplinePathController` causes it to swim endlessly in a circle about the origin. The world's camera points straight down, and resides at a sufficient distance from the origin to ensure that the shark is always visible in the Java 3D window. The HLA version of the shark world configuration contains an `HLAControllerCore`, also, and the XRTI version contains an `XRTIControllerCore`.

The commercial RTIs involved in the thesis experiment are the DMSO RTI-NG 1.3v6 and the MÄK RTI v2.0.3. Both RTIs require the use of dynamically linked libraries (DLLs); the DMSO RTI also requires the presence of an RTI executive. Only the DMSO RTI includes Java bindings, but since the MÄK RTI is link-compatible with the DMSO RTI, one can easily use DMSO's bindings to interface with it. The version of the XRTI used in the experiment is the initial prototype version described in earlier sections of this thesis.

The Java Memory Profiler (JMP) [Olofsson 03] and NetWorx [SoftPerfect 03] round out the experiment's list of required software. The JMP is a profiling utility for Java that measures, among other things, the relative amount of time spent in methods of interest by continually sampling the call stack of the Java Virtual Machine. NetWorx, in its Speedometer mode, measures the average transfer rate, maximum transfer rate, and total data transferred over the network in both incoming and outgoing directions.

c. Procedure

The experimental procedure involves setting up two clients running the shark world configuration in NPSNET-V and recording the performance characteristics of one client over a five-minute interval using each of the three RTIs. To ensure that

bursts of LAN traffic do not interfere with the experiment, the tests are conducted at night, when network usage is at a minimum. Computer A runs the untracked client as well as the necessary servers: the DMSO RTI executive and the XRTI Executive. Computer B runs the tracked client along with the performance measurement utilities. Because using the JMP significantly slows the Java Virtual Machine, each CPU usage metric comes from a run that is separate from the one used to gather the other metrics.

The steps of the procedure are as follows.

- Running the DMSO RTI executive on Computer A and NetWorx on Computer B, activate the HLA shark world configuration on both computers. On Computer B, press 'f' and begin the NetWorx speedometer, wait five minutes, press 'f' again, and record the results: the average frame rate and interaction latency as reported by NPSNET-V, and the average incoming and outgoing transfer rates as reported by NetWorx. Shut down the shark world on Computer B.
- Restart the shark world on Computer B with JMP active. Wait five minutes, then record the amount of time spent in the representative `emitEntityState` method of `HLAPlatformController`. Shut down the shark world on both computers, and stop the RTI executive on Computer A.
- Replace the DMSO DLLs on both computers with the corresponding MÄK DLLs and repeat steps 1 and 2.
- Running the XRTI Executive on Computer A, using the XRTI shark world configuration instead of the HLA one, and tracking `XRTIPlatformController`'s `emitEntityState` method instead of `HLAPlatformController`'s, repeat steps 1 and 2.

The resulting data consists of three sets of performance measurements: one for the DMSO RTI, one for the MÄK RTI, and one for the XRTI. Each set contains the average frame rate in frames per second, the average best-effort and reliable round-trip interaction latencies in milliseconds, the average incoming and outgoing network

transfer rates in kilobytes per second, and the representative CPU usage in terms of the number of seconds spent in the `emitEntityState` method.

3. Hypothesis

The hypothesis of the thesis experiment states that the XRTI matches or exceeds the performance of the DMSO and MÄK RTIs in terms of all measurements to within a five percent margin of experimental error. The reasoning behind this hypothesis is that although the XRTI's internal implementation and the implementation of its autogenerated proxy classes involve routing messages through several layers of indirection, and although Java libraries are typically slower than their native code counterparts, these performance impediments are not severe enough to limit the XRTI's efficiency to the point where it is slower than commercially available RTIs. Also, because both NPSNET-V and the XRTI are implemented in pure Java, whereas the DMSO and MÄK RTIs rely on Java bindings to native code, the XRTI has a performance advantage in that it avoids the inefficiency of transmitting messages through the Java Native Interface (JNI).

4. Results

The results of the thesis experiment follow, first as a table containing all recorded data, and then as a set of four graphs depicting each category of measurements for purposes of visual comparison.

	DMSO	MÄK	XRTI
Average frame rate (FPS)	42.4	48.1	49.3
Average best-effort interaction latency (ms)	32.8	17.3	6.5
Average reliable interaction latency (ms)	38.7	17.3	5.7
Average incoming network transfer rate (KB/s)	25.8	16.4	8.1
Average outgoing network transfer rate (KB/s)	16.1	11.0	15.0
Time spent in `emitEntityState` method (s)	6.0	3.0	3.0

Table 39. Results of the thesis experiment. The four rows of the data represent the different performance metrics; the three columns represent the RTIs tested.

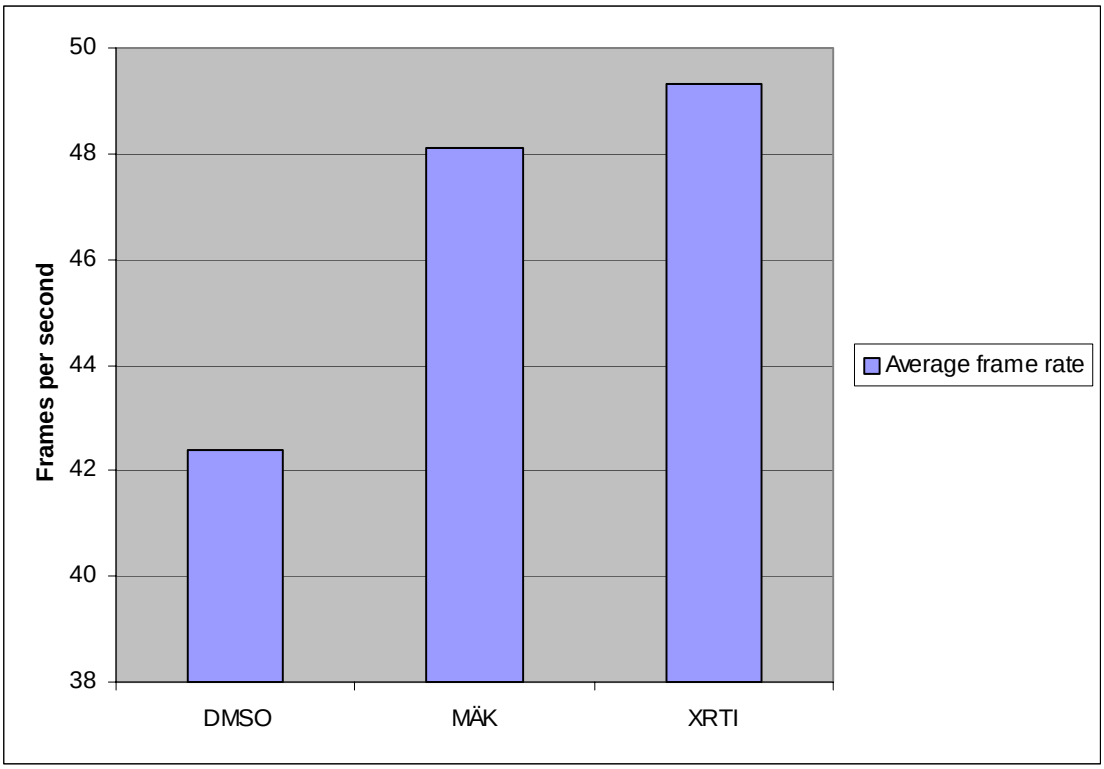

Figure 6. Graph of average frame rates.

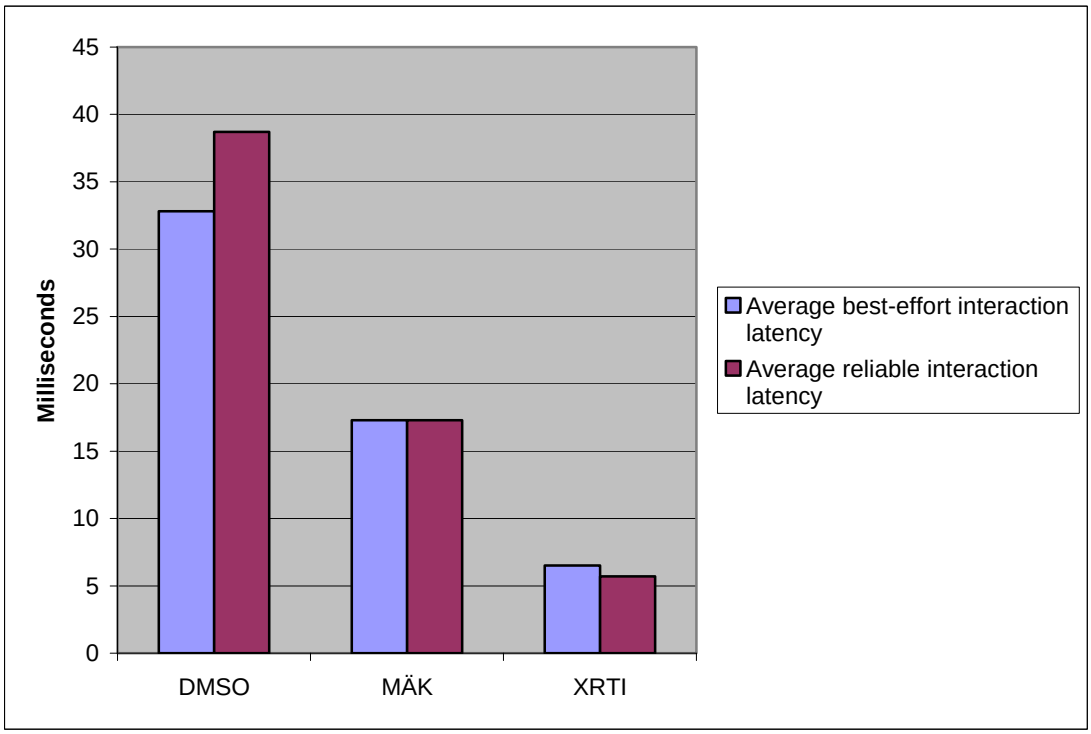

Figure 7. Graph of average interaction latencies.

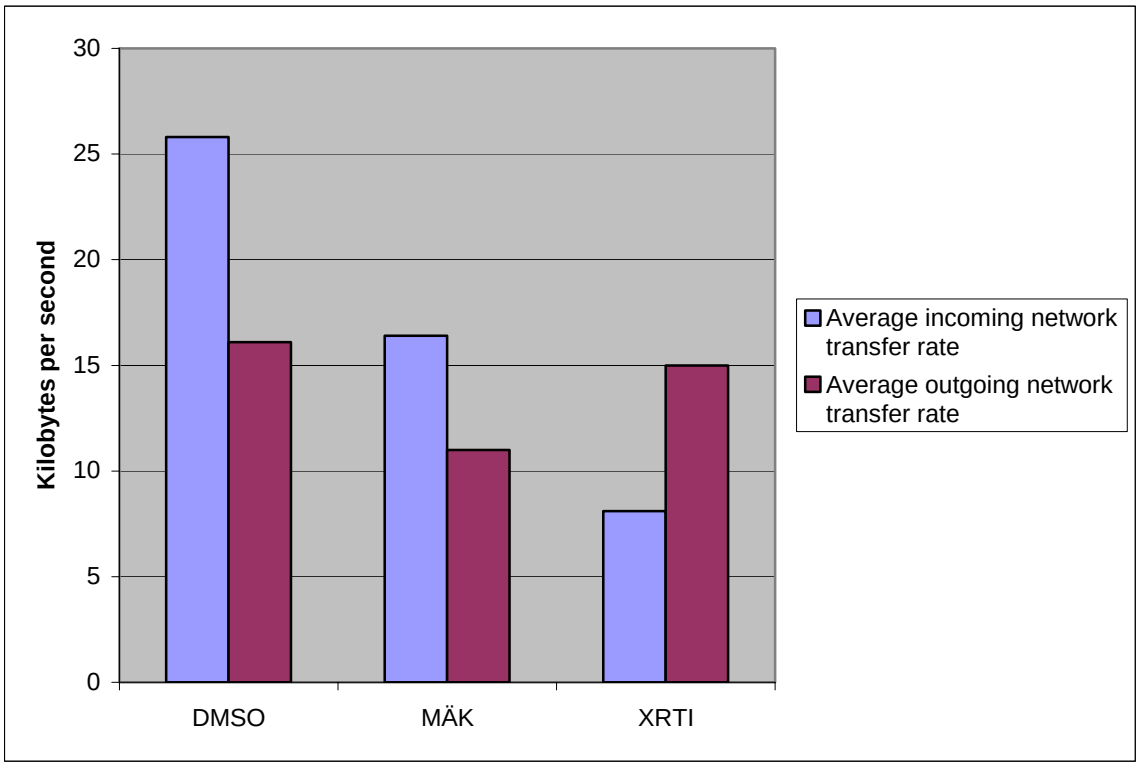

Figure 8. Graph of average network transfer rates.

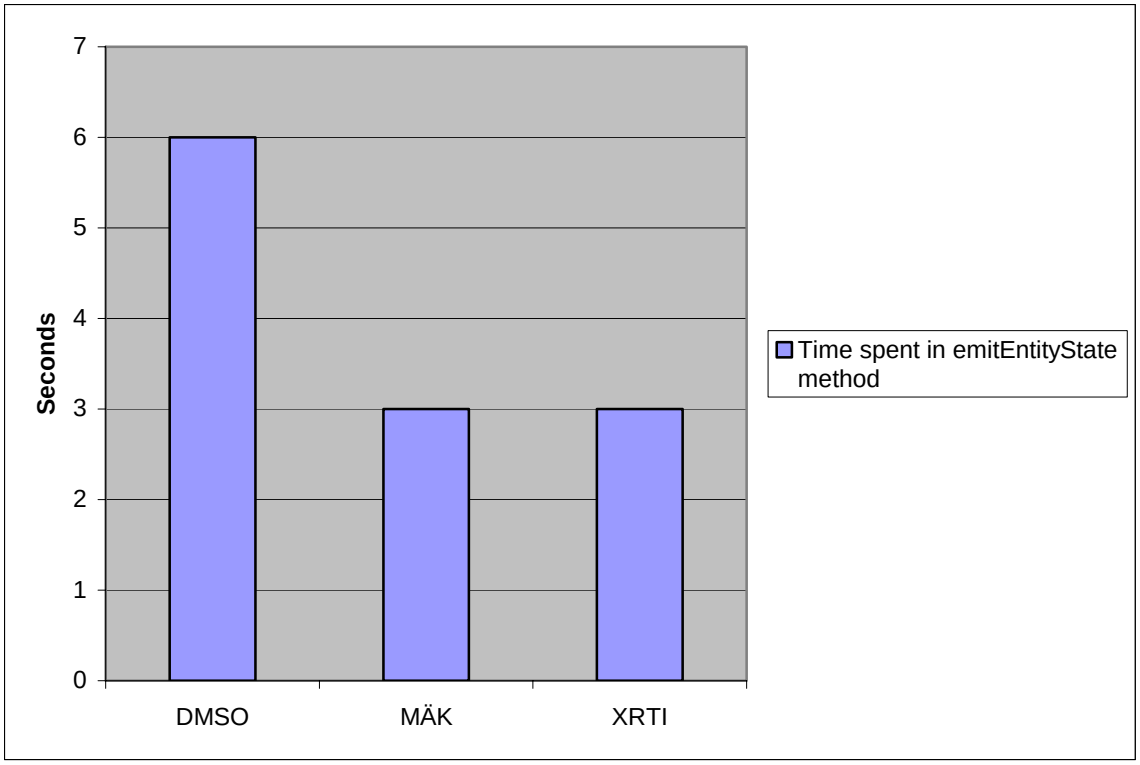

Figure 9. Graph of times spent in `emitEntityState` method.

5. Analysis

With one noteworthy exception, the results of the thesis experiment agree strongly with the hypothesis that the XRTI provides a level of performance equal to or exceeding the levels provided by commercial RTIs. In general, the MÄK RTI performs significantly better than the DMSO RTI, and the XRTI performs slightly better than the MÄK RTI. This is best demonstrated by comparing the frame rates of the three RTIs. The most dramatic difference in performance lies in the interaction latencies, where the MÄK RTI experiences about half the delay of the DMSO RTI, and the XRTI experiences about half the delay of the MÄK RTI. Possible causes for this may include the added latency involved in transmitting messages from Java code to native code using the JNI, or fundamental differences in the underlying network architectures of the RTIs. Comparing the times spent in the `emitEntityState` method suggests that the DMSO RTI's attribute update methods may involve some delay in contrast to the faster MÄK and XRTI update methods.

The discrepancy in the results lies in the network transfer rates, where the XRTI's average outgoing transfer rate is quite high—significantly higher than that of the MÄK RTI and nearly twice as high as its own incoming transfer rate. The most likely explanation for this behavior is that the faster CPU speed of Computer B, a 2 GHz machine as opposed to Computer A's 1 GHz, combined with the relative efficiency of the XRTI, allows Computer B to send out network updates at a significantly higher rate than that managed by Computer A. For future tests, using computers with identical feature sets should provide more consistent results.

VII. CONCLUSION

A. PROJECT SUMMARY

The Extensible Run-Time Infrastructure, or XRTI, is an open-source implementation of the IEEE 1516 High Level Architecture (HLA) standard with features designed to enhance ease-of-use, promote RTI interoperability, and enable dynamic object model extension and composition. The thesis project described by this document encompasses the design, implementation, and testing of the initial XRTI prototype, as well as the XRTI's integration into NPSNET-V, a dynamically extensible platform for networked virtual environments. In the context of a typical NPSNET-V environment, the performance of the XRTI matches or exceeds the performance of commercial RTIs. This fact, combined with the XRTI's open-source status, its comparative ease-of-use, and its support for dynamic extension, make the XRTI a better choice than commercial RTIs for networking between NPSNET-V instances.

The motivation behind the XRTI's development is the need for a standardizable communication mechanism for exchanging state data between the software components that support large-scale, long-running shared virtual worlds. Such a mechanism must be generalized enough to support any kind of environment, flexible enough to incorporate new kinds of data on the fly, efficient enough to meet the high performance requirements of interactive applications, and straightforward enough to encourage widespread acceptance. The HLA, a middleware standard for distributed simulations, is the most likely basis for such a mechanism. It is fully generalized, and its status as a middleware interface allows federate developers to concentrate on the data to be transferred as opposed to the network protocols that transport the data. Unlike the Common Object Request Broker Architecture (CORBA) and other generalized middleware interfaces, the HLA also includes features specifically useful to simulations, such as region-based filtering. However, the HLA cannot in itself solve the problem of providing universal interoperability, allowing all virtual environment applications to communicate with one another. Its lack of a common network protocol, the awkwardness of its interface, its

inability to extend live object models, and the fact that there are no open-source RTIs significantly limit the HLA's usefulness to virtual world developers.

The XRTI, therefore, represents an attempt to resolve the HLA's shortcomings in order to create a candidate standard for promoting interoperability between all networked virtual environment applications, including applications such as NPSNET-V whose communication ontologies change when they load new code modules. To conceal the HLA's awkward interface, the XRTI includes a proxy compiler that converts HLA FOM Document Data (FDDs) into sets of easy-to-use, type-safe Java proxy classes. To ease the process of defining a common message protocol for RTIs, the XRTI uses a novel bootstrapping technique to define its network protocols and other low-level communication elements in terms of HLA constructs and encodings. To allow federates to introduce new object and interaction classes without interrupting the federation execution, the XRTI provides a reflection mechanism that represents the federation object model (FOM) of each execution as a manipulable set of shared object instances. Finally, to ensure that federate developers always have access to an RTI that they can redistribute freely with their applications, to allow developers to examine the internal implementation of that RTI for diagnostic and educational purposes, and to provide a test bed for experimental RTI modifications and extensions, the XRTI is available for download on the World Wide Web under the very flexible Berkeley Systems Distribution (BSD) open-source license.

In order to test the XRTI, the Web-accessible distribution includes both standalone test applications and controller modules for NPSNET-V. NPSNET-V, a component-based platform for networked virtual environment applications, uniquely demonstrates a need for the XRTI's object model extension capability. Applications constructed on the NPSNET-V platform can dynamically add, remove, and replace parts of themselves at run-time in order to modify their functionality in response to changes in their simulated environments, upon the release of new versions of their code modules, or simply at the request of the user. When applications effect these changes, their networking modules must similarly adapt, extending their communication ontologies to include new constructs capable of representing new types of information. Unlike the

commercial RTIs used to test NPSNET-V's HLA controller modules, the XRTI provides explicit support for this kind of extensibility. In addition, a series of tests comparing the performance of the XRTI to that of the commercial RTIs shows that the XRTI's performance is quite competitive, and that even the XRTI prototype described in this thesis is able to meet the communication requirements of small shared virtual worlds.

B. FUTURE WORK

The initial XRTI prototype, however, represents only the beginning of what could be a complex and long-running project devoted to creating a full-featured and stable extensible RTI, to developing compelling demonstrations of the XRTI's functionality, and to augmenting the HLA standard with the extensions proposed in this thesis. Such a project would involve the investigation of many different areas, including both technical work and participation in the HLA community.

1. Widening Conversions

One of the first technical issues to investigate is that of widening conversions, which will allow federates to deal with unfamiliar constructs by representing them in terms of less specific, familiar ones. In object-oriented programming, if class `B` is a subclass (direct or otherwise) of class `A`, then the act of converting a `B` reference to an `A` reference is known as a widening cast, and the conversion of an `A` reference to a `B` reference is known as a narrowing cast. As the following Java code snippet demonstrates, widening casts can be performed implicitly, whereas narrowing casts must be explicitly stated, and can generate errors at run-time.

```
A a1 = new A(), a2;
B b1 = new B(), b2;

a2 = b1; // Widening cast, implicit

b2 = (B)a2; // Narrowing cast, explicit

b2 = (B)a1; // Illegal narrowing cast, throws
            // java.lang.ClassCastException
```

Figure 10. Widening and narrowing casts in Java. Class `B` is a subclass of class `A`.

For the XRTI, widening conversions will allow federates subscribed to object and interaction classes to receive interactions and object instance updates associated with derived classes. For an interaction class `A` and its subclass, `B`, the current version of the XRTI only reports `B` interactions to federates explicitly subscribed to `B`, not to federates subscribed only to `A`. Similarly, for an object class `A` and subclass `B`, federates only discover instances of `B` if they are subscribed to `B`. With support for widening conversions, the XRTI will report `B` interactions to all federates subscribed to interaction class `A`, and all federates subscribed to object class `A` will discover instances of `B` as well. Enabling this behavior is especially important for extensible applications, which must be able to interpret newly added constructs in terms of the constructs that they already understand. The widening process supports this by simply discarding the extra information unique to the new subclass. This form of representational flexibility is similar to that attributed to markup languages such as the Hyper-Text Markup Language (HTML) and the Extensible Markup Language (XML). Typically, when applications encounter elements or attributes that they do not understand in HTML or XML documents, they simply ignore them. This provides a level of forward compatibility, allowing applications to process data intended for later versions of themselves, as well as a certain robustness to heterogeneity, in that applications can choose to consume only the data that is relevant to their current configurations.

2. Extensible FOMs in NPSNET-V

As currently constructed, the NPSNET-V XRTI controller core does not provide an interface for its contained controller modules to extend the Realtime Platform Reference FOM (RPR-FOM) upon which it is based in order to introduce new constructs into active worlds. In order for NPSNET-V to take advantage of the XRTI's support for extensible object models, any of the XRTI controllers hosted by the controller core must be able to merge arbitrary FDDs into the execution's FOM and use proxies based on those FDDs to send new types of interactions and instantiate new kinds of object instances. Using NPSNET-V's resource identification system to track FDDs as named, versioned resources will prevent the XRTI controller core from having to upload an FDD to the XRTI Executive each time it loads a new sub-controller. Instead, the controller

core will remember the FDDs that it has already merged into the execution, and will only merge entirely new FDDs, or FDDs that represent newer versions of the ones already merged.

3. Supporting the Complete HLA Standard

The XRTI prototype does not, as it stands, represent a complete implementation of the HLA standard. Although it is usable for small-scale networked virtual environments of short duration, its lack of several critical services prevents it from supporting the complete range of distributed simulations for which the HLA is designed, as well as from supporting the large-scale, long-running networks of shared virtual worlds whose establishment is the XRTI project's long-term goal.

a. Ownership Management

The HLA's ownership management service ensures that at any given time, only one federate can own each attribute of each object instance, and that only an attribute's owner can update its value. Federates can transfer attribute ownership through a cooperative hand-off process, where both federates involved must agree to the transfer. Additionally, when a federate leaves the federation execution, rather than deleting the object instances that it has created, it can choose to divest ownership of their attributes to the first federate that claims it. For the XRTI, ownership management will be controlled by the XRTI Executive, which will have to keep track of the owner of each attribute of every object instance, acknowledging ownership transfers and rejecting attribute value updates sent by non-owners. The XRTI's proxy compiler will have to accommodate the ownership management service as well, supplying `divest` and `acquire` methods for each attribute and providing a means for federates to select either automatic or controlled attribute acquisition and divestiture.

b. Time Management

Time management is required to support simulations that do not operate in real-time. In discrete event simulation, for instance, the logical time of the simulation can advance in fixed time steps, or it can jump ahead varying lengths according to the timing of the events being simulated. Neither case requires the rate at which time

advances to bear any relation to wall-clock time, and thus the simulation is free to take full advantage of the processor time allocated to it, executing time steps or processing events as fast as possible. HLA federates must be synchronized by the RTI in order to achieve this ability in a distributed environment. Every federate subject to time management has both a logical time and a lookahead interval, and must choose whether it is to be time-constrained, time-regulating, both, or neither. Time-constrained federates must wait for notification before they can advance their logical times, and time-regulating federates can request time advances from the RTI. At any time, federates cannot send events with time stamps less than the value of their logical times plus their lookahead intervals. This allows the RTI to collect the events sent by each federate, sort them in time stamp order, and deliver them to the other federates without ever requiring a federate to process an event that occurred in its past.

In order to support time management, the XRTI Executive will have to track the logical times and lookahead intervals of all connected federates, processing time advance requests from the time-regulating ones and granting time advances to the time-constrained ones. Also, since time management involves associating time stamps with events such as interactions and attribute value updates, the proxy compiler will have to include additional versions of the `sendX` methods of each proxy ambassador class, the `receiveX` methods of each interaction listener interface, the `setX` methods of each object instance proxy class, and the `xUpdated` methods of each object instance listener interface, each with a time stamp parameter added to its original argument list.

c. Data Distribution Management

Data distribution management (DDM) allows each federate to filter the events that it receives according to the overlap between its region of interest and the events' regions of influence. Each region contains minimum and maximum extents in a number of logical dimensions, where each dimension may correspond to a spatial dimension, a time axis, a functional range, or some other type of dimension as defined by the FOM. In order for two regions to overlap, their extents must overlap in every dimension. Under DDM, federates specify subscription regions when they subscribe to interaction classes or object class attributes, and associate region sets with the events that

they send. The RTI considers the subscription parameters of each federate when it processes these events, notifying only the federates whose subscription regions overlap with the event regions. For the XRTI to support DDM, the XRTI Executive will have to track the subscription parameters of each federate and be able to accept events with associated region sets. As with time management, the proxy compiler will have to generate modified versions of the `sendX`, `receiveX`, `setX`, and `xUpdated` methods. For DDM, the extra methods must include region sets among their parameters.

d. Other Services

The other services missing from the XRTI prototype are event retraction, synchronization, and federation save/restore. Event retraction, in which the invocation of each event returns a retraction handle that the federate can use to retract the event later on, is useful for optimistic simulation. In this form of simulation, federates project their state ahead of the rest of the federation, but they must be prepared to roll that state back when they receive events whose time stamps lie within the interval that they have already simulated. The XRTI's role in this will be to relay retraction messages to the federation when such a federate, in the process of rolling its state back, must retract events that it has sent. Also, the methods generated by the proxy compiler will require additional variants; event-generation methods such as `sendX` and `setX` must return retraction handles, and event-reception methods such as `receiveX` and `xUpdated` must include retraction handle parameters.

The synchronization and save/restore services are similar to each other in that they require the cooperation of all federates. Synchronization operations allow all federates participating in an execution to reach a common milestone, or synchronization point. Once a single federate has requested synchronization by registering a synchronization point, the RTI notifies all federates that it is awaiting synchronization at that point. When each federate reaches the point, it notifies the RTI that it has done so. When the RTI has received such notifications from all joined federates, it sends a message to the federation indicating that it has reached the synchronization point. Save and restore operations occur almost identically; one federate requests that the state of the federation be saved under or restored from a particular label, and the RTI broadcasts the

save or restore request, waiting for responses from all federates before notifying the federation that the operation is complete. To coordinate these tasks, the XRTI Executive will have to track synchronization points and save/restore labels, remembering which federates have achieved which targets.

4. RTI Verification

Once the XRTI provides a complete implementation of the HLA standard, it will be a candidate for official DMSO verification [DMSO 03, *RTI Verification*]. The verification process, which ensures that RTIs conform exactly to the IEEE 1516 standard, consists of two levels. The first level requires RTI developers to run a series of tests on their RTIs, each exercising a particular aspect of the RTI's functionality. The titles of these tests are "Joining Federations," "Synchronization Points," "Save and Restore," "Object Discovery," "Attribute Divestiture and Acquisition," "Time Advancement," "Region Intersection," and "Management Object Model." When RTI developers are satisfied that their RTIs pass the first level of testing, they can submit their RTIs to DMSO's RTI Verification Facility for the second level. There, DMSO experts determine whether or not each RTI satisfies the requirements of the HLA standard, posting the results on the RTI Verification Status Board.

5. Proposing Extensions to the HLA Community

Along with achieving DMSO certification for the XRTI, the other goal of interacting with the HLA community is to propose the extensions introduced in this thesis as additions to the HLA standard. Specifically, if the extensibility afforded by the `mergeFDD` method and the interoperability that comes from having a common message protocol are to be made available to all applications based on the HLA, then the HLA standard must be extended to include a specification for the `mergeFDD` method and a message protocol definition. To assist this process, the XRTI will act as a proof of concept, demonstrating the feasibility and advantages of implementing the required changes, as well as a reference implementation, showing other RTI developers how to conform to the revised standard.

6. Porting/Binding to Other Languages

In order to allow federates written in languages other than Java to use it, the XRTI must include either bindings or complete ports to those languages. The IEEE 1516.1 specification, for instance, defines interfaces for C++ and Ada as well as for Java. In the simplest case, to support federates written in C++, it may be sufficient to create bindings that implement the C++ interface defined by IEEE 1516.1 simply by using the Java Native Interface (JNI) to invoke the XRTI's Java methods. Because using the JNI incurs a performance penalty, however, and because software written in native code tends to perform better than that written in Java, it may be desirable instead to recreate the XRTI in C++. In either case, if C++ federates are to use autogenerated proxies like their Java counterparts, then the XRTI must include a version of the proxy compiler that generates C++ source files and headers instead of Java classes. Providing support for languages other than C++, such as Ada, will require similar changes.

7. Integrating Additional Networking Profiles

The client-server networking profile supported by the XRTI prototype is the simplest profile to implement, but its efficiency and fault-tolerance are limited by its reliance on a central server, which acts as a bottleneck and a single point of failure. The hybrid and peer-to-peer profiles will offer several advantages over the client-server model.

a. Hybrid

The hybrid networking profile will combine the central control offered by the client-server model with the decentralized, unrestricted messaging associated with the peer-to-peer model. The hybrid profile will still require federates to connect to an instance of the XRTI Executive, but the majority of the federates' communication with each other will take place through peer-to-peer channels, such as IP multicast groups. Instead of routing all messages associated with the execution, the XRTI Executive will assign multicast groups or other communication channels to specific executions and regions as necessary, subdividing channels when they become too congested and combining them when their traffic dies down. Apart from this, the XRTI Executive will

still perform most of the roles that it performs in the client-server profile, which means that although it will no longer be a bottleneck, it will still be a single point of failure. When the XRTI Executive is shut down, federates will no longer be able to participate in federation executions.

b. Peer-to-Peer

The peer-to-peer networking profile will eliminate the need to run the XRTI Executive as a separate application by allowing any federate to act as an executive when necessary. When each federate initializes its RTI ambassador and thereby connects to a communication channel, its ambassador will query the channel to determine if other federates are online. If no federates respond, then the ambassador will establish itself as the channel's executive and await the arrival of other federates. The acting executive will perform the same duties that the XRTI Executive application performs in the client-server and hybrid profiles: allocating and populating new channels, coordinating save and restore operations, providing sets of unique identifiers, and so on. When the acting executive disconnects from the channel, it will have to transfer the executive role to another federate by using the HLA's ownership management service to divest ownership of all executive state. If the acting executive goes offline without warning, the other federates will have to detect its absence and promote a replacement automatically. This ability to replace the executive when necessary will prevent the peer-to-peer profile from having a single point of failure.

The peer-to-peer profile will be the most difficult one to implement, but it promises to be more versatile and useful than the client-server or hybrid profiles. Aside from the convenience of not having to run the XRTI Executive, and apart from the bottleneck reduction and improved fault-tolerance as compared to the client-server and hybrid profiles, the peer-to-peer profile more closely matches the vision of a fully distributed network of shared virtual worlds described in the introductory chapter of this thesis. Realizing that vision may involve implementing reliable multicast protocols on top of the XRTI's bootstrap object model, and it may involve using overlay multicast techniques to compensate for the limited availability of IP multicast. In the end,

however, for a virtual world network to be as successful as the Internet and the World Wide Web, it will have to imitate the Internet's decentralized nature as well as the Web's interoperability and extensibility.

C. OBTAINING THE XRTI

All of the software developed and documents written in support of the XRTI project are available on the World Wide Web through the XRTI Web site [NPSNET 03, *XRTI*]. That Web site also includes instructions for contributing to the XRTI's development as well as links to the sites of related projects, such as NPSNET-V.

1. Packaging

The XRTI distribution includes a Java archive, `xrti.jar`, that contains the compiled XRTI packaged as a Java extension. The extension manifest, included separately in the distribution as `xrti.mf`, contains the extension's name and version information, as well as a location from which applications can download the extension. Applications, applets, and other Java classes that use the XRTI can list this extension as a dependency in their jar manifests in order to instruct extension-aware applications such as NPSNET-V to download and link the XRTI before continuing to load the dependent archive.

2. Distribution

Each release of the XRTI distribution consists of two files: an archive containing the compiled, packaged XRTI along with all documentation and support files (`xrti-devkit.zip`), and an archive containing the XRTI's complete source code (`xrti-source.zip`). The first release, which contains the XRTI prototype described in this thesis, is available for download from a distribution center hosted by the SourceForge open-source community Web site [OSDN 03].

3. Licensing

SourceForge requires all of its hosted projects to subscribe to an open-source license. The XRTI's source code is available under the BSD license, which allows

anyone to modify and redistribute the XRTI code as they see fit, provided that they include a copy of its license file. The complete text of that file is as follows.

```
Copyright (c) 2003, The MOVES Institute
All rights reserved.

Redistribution and use in source and binary forms, with or without modification,
are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this
   list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

3. Neither the name of The MOVES Institute nor the names of its contributors may
   be used to endorse or promote products derived from this software without
   specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

Figure 11. The BSD license as included with the XRTI.

4. Development

SourceForge also hosts the Concurrent Versions System (CVS) repository where the XRTI's ongoing development occurs. Developers interested in contributing to the XRTI are encouraged to contact the author of this thesis in order to find development tasks that match their talents and to obtain permission to submit changes to the repository. It is the author's hope that, with the help of the open-source development community, the XRTI will grow from a research prototype into a complete and certified HLA RTI, and finally into a communications infrastructure capable of supporting an extensive, diverse, and lasting network of shared virtual worlds.

APPENDIX A. GLOSSARY

Attribute. A named, typed element of object state. HLA object models define the attributes associated with each object class. Object classes inherit their parents' attributes.

Bootstrap Object Model (BOM). A special FOM that describes the message protocol payload in terms of HLA data types, and describes basic interactions such as `HLAupdateAttributeValues`. A proposed extension to the HLA standard.

Communication ontology. A set of constructs used to share information between applications. Communication ontologies typically include the constructs' properties, their semantic associations, and any relationships that exist between the constructs. HLA object models represent communication ontologies.

Executive. A server that manages a number of federation executions. Acts as a central point of control and mediation between federates.

Federate. A federation participant. Federates communicate with one another through the RTI during the course of a federation execution.

Federation. A distributed simulation involving multiple federates.

Federation execution. A federation instance.

FOM Document Data (FDD). An XML document containing a federation object model and other federation parameters.

Federation Object Model (FOM). The object model associated with a federation. Defines the set of object and interaction classes that federates can instantiate during the course of a federation execution.

High Level Architecture (HLA). A standard for sharing information between federates during the course of a federation execution.

Interaction. An instance of an interaction class. Contains values for the parameters defined by that class. Represents an announcement made by a federate.

Interaction class. A type of interaction, as described in an object model. Interaction classes can have one or more parent classes and can define any number of parameters.

Management Object Model (MOM). A special FOM that describes management-level constructs and operations within the scope of a single federation execution. Defined by the HLA standard.

Message. A piece of information transmitted between RTI components. May represent, for instance, an interaction or an attribute value update.

Message protocol. An application-layer protocol used by the RTI to transmit and receive messages over the network.

Meta-Federation Object Model (MFOM). A special FOM that describes management-level constructs and operations outside the scope of any federation execution, such as the `HLAcreateFederationExecution` interaction. A proposed extension to the HLA standard.

Object. An instance of an object class. Contains values for the attributes defined by that class. Represents an entity in play.

Object class. A type of object, as described in an object model. Object classes can have one or more parent classes and can define any number of attributes.

Object model. A simulation schema. Describes a set of object and interaction classes.

Object Model Template (OMT). The format to which object models must conform.

Parameter. A named, typed interaction property. HLA object models define the parameters associated with each interaction class. Interaction classes inherit their parents' parameters.

Reflection Object Model (ROM). A special FOM that describes a set of constructs for representing the FOM itself. A proposed extension to the HLA standard.

Run-Time Infrastructure (RTI). The middleware interface through which federates communicate during the course of a federation execution.

LIST OF REFERENCES

[Adobe 03] Adobe Systems Incorporated. "Adobe Atmosphere." [http://www.adobe.com/products/atmosphere/main.html]. April 2003.

[America's Army 03] America's Army. "America's Army: Operations." [http://www.americasarmy.com/operations/index.php]. April 2003.

[Blümel 02] Blümel, E., Schenk, M., and Schumann, M. "Distributed Virtual Worlds with HLA?" *Proceedings of the 2002 Fall Simulation Interoperability Workshop*. Orlando, Florida. 8-13 September 2002.

[Bolton 02] Bolton, F. *Pure CORBA.* Sams Publishing, 2002.

[Brassé 00] Brassé, M. and Kuijpers, N. "Realizing a Platform for Collaborative Virtual Environments based on the High Level Architecture." *Proceedings of the 2000 Spring SISO Simulation Interoperability Workshop*. Orlando, Florida. 26-31 March 2000.

[Capps 00] Capps, M., McGregor, D., Brutzman, D., and Zyda, M. "NPSNET-V: A New Beginning for Dynamically Extensible Virtual Environments." *IEEE Computer Graphics and Applications*, *20*(5): 12-15 (2000).

[Cazard 02] Cazard, L. and Adelantado, M. "HLA Federates Design and Federation Management: Towards a Higher Level Object-Oriented Architecture Hiding the HLA Services." *Proceedings of the 2002 Spring SISO Simulation Interoperability Workshop*. Orlando, Florida. 10-15 March 2002.

[Cox 98] Cox, K. "A Framework-Based Approach to HLA Federate Development." *Proceedings of the 1998 Fall SISO Simulation Interoperability Workshop*. Orlando, Florida. September 14-18, 1998.

[DMSO 03, *RTI*] Defense Modeling and Simulation Office. "Runtime Infrastructure (RTI)." [https://www.dmso.mil/public/transition/hla/rti/]. September 2003.

[DMSO 03, *RTI Verification*] Defense Modeling and Simulation Office. "RTI Verification." [https://www.dmso.mil/public/transition/hla/verification/]. September 2003.

[Dumond 01] Dumond, R. "A FOM Flexible Federate Framework." *Proceedings of the 2001 Spring SISO Simulation Interoperability Workshop*. Orlando, Florida. 25-30 March 2001.

[Fischer 01] Fischer, W.D. *Enhancing Network Communication in NPSNET-V Virtual Environments Using XML-Described Dynamic Behavior (DBP) Protocols.* Master's Thesis. Naval Postgraduate School. Monterey, California. September 2001.

[Gamma 95] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Givens 00] Givens, B.R. "Positions for and against an Open-Source RTI." *Proceedings of the 2000 Spring SISO Simulation Interoperability Workshop*. Orlando, Florida. 26-31 March 2000.

[Granowetter 03] Granowetter, L. "RTI Interoperability Issues—API Standards, Wire Standards, and RTI Bridges." *Proceedings of the 2003 Spring SISO Simulation Interoperability Workshop.* Orlando, Florida. 30 March-4 April 2003.

[Hunt 99] Hunt, K. and Graham, J. "OMni™: A FOM and Language Independent Interface to the RTI." *Proceedings of the 1999 Spring SISO Simulation Interoperability Workshop*. Orlando, Florida. 14-19 March 1999.

[IEEE 1278] The Institute of Electrical and Electronics Engineers, Inc., IEEE 1278-1993. *IEEE Standard for Information Technology—Protocols for Distributed Interactive Simulation Applications.* 18 March 1993.

[IEEE 1516] The Institute of Electrical and Electronics Engineers, Inc., IEEE Std 1516-2000. *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)—Framework and Rules.* 11 December 2000.

[IEEE 1516.1] The Institute of Electrical and Electronics Engineers, Inc., IEEE Std 1516.1-2000. *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)—Federate Interface Specification.* 9 March 2001.

[IEEE 1516.2] The Institute of Electrical and Electronics Engineers, Inc., IEEE Std 1516.2-2000. *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)—Object Model Template (OMT) Specification.* 9 March 2001.

[Kapolka 02] Kapolka, A., McGregor, D., and Capps, M. "A Unified Component Framework for Dynamically Extensible Virtual Environments." *Proceedings of the Fourth ACM International Conference on Collaborative Virtual Environments*. 30 September-2 October 2002.

[Kapolka 03] Kapolka, A. "A Dynamically Extensible Platform for Browsing, Building, Publishing, and Hosting Shared Virtual Worlds." [http://www.npsnet.org/~npsnet/v/publications/presence2004.pdf]. September 2003.

[Kuhl 99] Kuhl, F., Weatherly, R., and Dahmann, J. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture.* Prentice Hall, 1999.

[Liles 98] Liles, S.W. *Dynamically Extending a Networked Virtual Environment Using Bamboo and the High Level Architecture.* Master's Thesis. Naval Postgraduate School. Monterey, California. September 1998.

[Louis Dit Picard 01] Louis Dit Picard, S., Degrande, S., and Gransart, C. "A CORBA-Based Platform as Communication Support for Synchronous Collaborative Virtual Environments." *Proceedings of the 2001 International Workshop on Multimedia Middleware.* Ottawa, Canada. 5 October 2001.

[MÄK 03] MÄK Technologies. "MÄK Technologies High Performance RTI." [http://www.mak.com/rti.htm]. September 2003.

[Maso 00] Maso, B. "A New Era for Java Protocol Handlers." [http://developer.java.sun.com/developer/onlineTraining/protocolhandlers]. May 2003.

[Mullally 03] Mullally, K., Hall, G., Gordon, D., Pemberton, B., and Peabody, C. "Open, Message-Based RTI Implementation—A Better, Faster, Cheaper Alternative to Proprietary, API-Based RTIs?" *Proceedings of the 2003 Spring SISO Simulation Interoperability Workshop.* Orlando, Florida. 30 March-4 April 2003.

[Myjak 99] Myjak, M.D., Sharp, S.T., and Briggs, K. "Javelin." *Proceedings of the 1999 Spring SISO Simulation Interoperability Workshop*. Orlando, Florida. 14-19 March 1999.

[NPSNET 03, *NPSNET-V*] The NPSNET Research Group. "NPSNET-V." [http://www.npsnet.org/~npsnet/v/]. September 2003.

[NPSNET 03, *XRTI*] The NPSNET Research Group. "XRTI: Extensible Run-Time Infrastructure." [http://www.npsnet.org/~npsnet/xrti/]. September 2003.

[Olofsson 03] Olofsson, R. "Java Memory Profiler." [http://www.khelekore.org/jmp/]. September 2003.

[Oliveira 02] Oliveira, M., Crowcroft, J., and Slater, M. "TreacleWell: Unraveling the Magic 'Black Box' of the Network." *Proceedings of the Sixth World Multiconference on Systemics, Cybernetics, and Informatics.* Orlando, Florida. 14-18 July 2002.

[OSDN 03] Open Source Development Network. "SourceForge.net." [http://sourceforge.net/]. September 2003.

[Robinson 00] Robinson, J.L., Stewart, J.A., and Labbe, I. "MVIP—Audio Enabled Multicast VNet." *Proceedings of the Fifth Symposium on Virtual Reality Modeling Language.* Monterey, California. 21-24 February 2000.

[Serin 03] Serin, E. *Design and Test of the Cross-Format Schema Protocol (XFSP) for Networked Virtual Environments.* Master's Thesis. Naval Postgraduate School. Monterey, California. March 2003.

[Singhal 99] Singhal, S., and Zyda, M. *Networked Virtual Environments: Design and Implementation.* Addison-Wesley, 1999.

[SoftPerfect 03] SoftPerfect Research, LLC. "NetWorx." [http://www.softperfect.com/products/networx/]. September 2003.

[Sony 03] Sony Computer Entertainment America Inc. "EverQuest." [http://everquest.station.sony.com]. April 2003.

[USJFCOM 03] United States Joint Forces Command. "USJFCOM: Millenium Challenge 2002." [http://www.jfcom.mil/about/experiments/mc02.htm]. April 2003.

[Wilson 01] Wilson, S., Sayers, H., and McNeill, M.D.J. "Using CORBA Middleware to Support the Development of Distributed Virtual Environment Applications." *Proceedings of the 9th WSCG International Conference in Central Europe on Computer Graphics, Visualization, and Computer Vision*. Plzen, Czech Republic. 5-9 February 2001.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Michael Zyda
   Naval Postgraduate School
   Monterey, California

4. Bret Michael
   Naval Postgraduate School
   Monterey, California

5. Andrzej Kapolka
   Naval Postgraduate School
   Monterey, California