SIP-RTI: A High Level Architecture, Runtime Infrastructure built on a SIP-enabled Conferencing Mechanism

by

Claude O. H. Van Ham,
M.S., BEng., PEng,

A Thesis Submitted to the Faculty of Graduate Studies and Research
In Partial Fulfillment of the Requirements For the Degree of

Master of Applied Science

Ottawa-Carleton Institute for Electrical and Computer Engineering

Faculty of Engineering

Department of Systems and Computer Engineering

Carleton University, Ottawa, Ontario

Your file Votre référence
ISBN: 978-0-494-18333-5
Our file Notre référence
ISBN: 978-0-494-18333-5

Abstract

The High Level Architecture (HLA) is a popular standard for distributed simulation interoperability. The specification defines a distributed runtime infrastructure (RTI) but not its implementation. Consequently, heterogeneous RTIs do not interoperate. Furthermore, the specification does not include an explicit session layer or security.

The SIP-RTI is a new model for RTI interoperability. A distributed conferencing mechanism, called the Conferencing Infrastructure (CI), is used below the RTI to provide generic communication services. RTI semantics are given in terms of the CI. The way in which the RTI uses the CI is known, therefore, other applications can interoperate with the RTI at the CI level. The CI uses the Session Initiation Protocol (SIP) to establish communication between CI nodes, which has the potential to solve other RTI issues as well. Ownership Management services (IEEE 1516) demonstrate the viability of the conferencing model to support HLA semantics. Interoperability between heterogeneous applications is demonstrated.

Dedicated in
memory of
of my father

Meinard L. Van Ham

Acknowledgements

Once again (this is my second graduate degree in which he has been a positive factor), I find myself indebted to Dr. Trevor Pearce who, during this research, acted not only as mentor but also as my advisor. His encouragement and advice were greatly appreciated. Trevor often went beyond the role of an advisor during my time at Carleton University and I thank him for his support. He remains one of the best teachers and engineers I know and I am happy to have been his student during this research.

Scott Holben collaborated with Trevor and myself in the development of some of the initial concepts on which this research is based and also provided guidance as I learned the Session Initial Protocol. I would like to thank him for this assistance.

I would also like to thank my mother and father for their encouragement and support in pursuing my own ideas and desires, one of which was this return to school.

This work was supported in part by similar grants from my advisor, the IBM Corporation and the Ontario Government in the form of an Ontario Graduate Scholarship in Science and Technology.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

A	- Available (floor)
API	- Application Programming Interface
BEEP	- Blocks Extensible Exchange Protocol
BOM	- Bootstrap Object Model (used in XRTI)
BOM	- Base Object Model (used by SISO)
CallerID	- Caller Identifier
CDATA	- Character Data (used in XML)
CI	- Conferencing Infrastructure
ConfID	- Conference Identifier
CORBA	- Common Object Request Broker Architecture
DCOM	- Distributed COM (DCOM)
DDM	- Data Distribution Management
DM	- Declaration management
DMSO	- Defense Modelling and Simulation Office
DoD	- Department of Defence
DTD	- Document Type Definition
F	- Floor
FDK	- Federated Simulations Development Kit
FIFO	- First In First Out
FloorID	- Floor Identifier
FOM	- Federation Object Model
HLA	- High Level Architecture
HTTP	- Hypertext Transfer Protocol
IEEE	- Institute of Electrical and Electronics Engineers
IETF	- Internet Engineering Task Force
IP	- Internet Protocol
IPSec	- IP Security
ISO/OSI	- International Standards Organization/Open Systems Interconnection
ITU	- International Telecommunication Union
JAIN	- Java API for Integrated Networks
L	- Listener
LAN	- Local Area Network
LI	- Listener Indicator
LRC	- Local RTI Component
M&S	- Modelling and Simulation
Mbps,	- Mega bits per second
MIME	- Multipurpose Internet Mail Extension
NIST	- National Institute of Standards and Technology
O	- Open (floor)
oA	- owned Available floor
OGSA	- Open Grid Services Architecture
OGSI	- Open Grid Services Infrastructure
OMT	- Object Model Template
ONERA	- Office National d'Études et de Recherches Aérospatiales

OOP	- Object Oriented Programming
ORB	- Object Request Broker
P2P	- Peer-to-Peer
PI	- Participation Indicator
PIL	- Participation Indicator List
PSTN	- Public Switched Telephone Network
QoS	- Quality of Service
RFC	- Request For Comments
RI	- Reference Implementation
RMI	- Remote Method Invocation
RPC	- Remote Procedure Call
RTI	- Runtime Infrastructure
S	- Speaker
SDP	- Session Description Protocol
SISO	- Simulation Interoperability Standards Organization
SI_F	- Speaker Indicator, Floor
SIP	- Session Initiation Protocol
SI_L	- Speaker Indicator, Listener
sLCC	- SIP Local Conferencing Infrastructure Component
sLRC	- SIP Local RTI Component
S/MIME	- Secure MIME
SOAP	- Simple Object Access Protocol
SCTP	- Stream Control Transport Protocol
TCP	- Transmission Control Protocol
TLS	- Transport Layer Security
uA	- unowned Available floor
UAC	- User Agent Client
UAS	- User Agent Server
UDP	- User Datagram Protocol
UML	- Unified Modelling Language
URI	- Universal Resource Indicator
VoIP	- Voice over IP
XML	- Extensible Markup Language
XRTI	- Extensible Run-Time Infrastructure
XMSF	- Extensible Modelling Simulation Framework

Chapter 1 Introduction

The main goal of the High Level Architecture (HLA) is the creation of simulations through the combination of other simulations [1]. Individual simulations, called *federates*, may be combined into a single, larger simulation, called a *federation*. The architecture allows such combinations to be created flexibly; disassociating federates so they can be used in other federations as well, thereby promoting reusability.

The HLA is defined by IEEE standard 1516-2000 ("IEEE 1516") in three volumes [2-4], the second of which defines the Runtime Infrastructure (RTI). The RTI provides services to federates which allow them to interact in a *federation execution* (a running federation). It is defined as an Application Programming Interface (API), not as an implementation. As long as the characteristics of the interface and the HLA rules are maintained, the RTI may be implemented in any way by its developers.

The flexibility afforded by the HLA's approach to defining the RTI and other aspects of the HLA specification have resulted in various shortcomings in current RTI implementations. Federations are often distributed across computers. Individual federates may run on disparate computers gaining access to RTI services through a locally implemented component of the RTI called a Local RTI Component (LRC). To support distributed design, RTIs must provide network communication capabilities, however, RTIs are generally not interoperable [5] due to differences in implementation. Therefore, federates running on one implementation typically cannot operate with federates running on another. In addition, IEEE 1516 provides no support for security, firewall traversal or the discovery of remotely located federation participants or services.

The Session Initiation Protocol (SIP) is a control protocol that supports the creation, control and termination of multipoint, multimedia communication sessions. It is becoming the protocol of choice in the implementation of Voice over IP (VoIP) networks, having largely supplanted the more complex International Telecommunication Union (ITU) H.323 protocols. While most frequently used in this arena, the SIP is a flexible protocol capable of supporting any application requiring an underlying communication infrastructure. Because it is stateful, the SIP supports firewall traversal, even using UDP.

It provides security mechanisms to protect its control messaging and to authenticate end points. In addition, the SIP uses separate control *signalling* channels between entities to establish application *session* communication channels. The signalling channels can be used to negotiate the use of any protocol over the session channels. As a result, existing security and quality of service (QoS) protocols can be set up and used between end points. Finally, because of its growing popularity in the telecommunications sector, the SIP is gaining increasingly widespread support. This has resulted in the rapid development of SIP extensions, while still maintaining the simplicity of the core specification (defined in [6]).

The thesis of this research is that a conference-based session layer (layer 5, ISO/OSI reference model) will provide a valuable step towards interoperability, not only of RTIs, but distributed systems in general. The primary goal of this research is the development and demonstration of a SIP-based, general-purpose distributed communication system, capable of supporting RTI communication and, eventually, interoperability. The use of the SIP in this work was inspired by [7] and by this protocol's various benefits. The HLA is used as an initial, more manageable, proof-of-concept of the underlying communication middleware and its potential to fulfill the larger vision of a general interoperability solution.

A fully distributed conferencing mechanism called a *Conferencing Infrastructure* (CI) has been designed and implemented. The CI implements a session layer communication system based on generic conferencing services. The CI uses TCP session channels, established using the SIP, to support communication among nodes participating in the CI. A conference-based model (referred to as the *crtiModel*) of HLA semantics is also developed. The crtiModel is implemented as a custom-built, IEEE 1516 compliant, RTI API middleware that uses the CI's services. Figure 1-1 depicts these components using a quasi-ISO/OSI reference model structure. As shown, when considered together, these two applications result in an RTI called the *SIP-RTI*. The SIP-RTI relies on common transport, network, data-link and physical layers and provides HLA services to a federation running above it. The current implementation supports HLA ownership management services as a test case. Because the system uses the SIP to establish session

channels between nodes, it provides the SIP-RTI with the benefits of the SIP and may lead to the solution of other RTI problems mentioned.

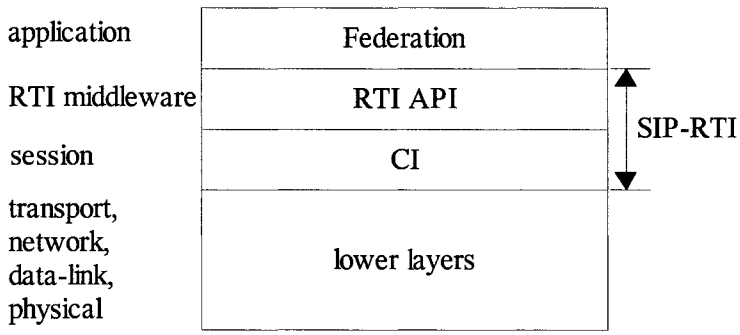

Figure 1-1 The CI-based SIP-RTI

The next chapter introduces the HLA and the SIP to the depth needed to understand this research. It also describes an abstract model of the RTI used in later discussions. Chapter 3 reviews the literature on current RTI implementations and issues including interoperability, firewall traversal and general security, discovery, and the lack of an open source RTI. It also presents current work on conferencing frameworks and conference control, and discusses issues relevant to distributed systems and applicable to the design of the CI. Chapter 4 analyzes the material presented in Chapter 3 focusing on the issue of RTI interoperability. As mentioned, this issue is not confined to the realm of the HLA; Chapter 4 also discusses the larger vision behind this research and scopes it to a goal manageable in a work of this size. Finally, Chapter 4 provides a concise statement of this dissertation's thesis, provides an overview of the SIP-RTI solution and identifies the contributions it makes. Chapters 5 and 6 describe the design of the two major components that comprise this system and the models on which they are based. The components are called the *SIP Local Conference Component* or *sLCC*, and the *SIP Local Runtime Infrastructure Component* or *sLRC*. Together they implement the local component of the SIP-RTI. A demonstration of the SIP-RTI and simple interoperability was developed in the form of three simple federates and a monitor application. The resulting air traffic controller demonstration is described in Chapter 7. Finally, Chapter 8 presents conclusions and recommendations for future work.

Chapter 2 Background Material

This chapter provides details of the HLA and the SIP that are relevant to the rest of this dissertation. An abstract RTI implementation model is also presented. The material in this chapter is largely based on [8].

2.1 An Overview of the High Level Architecture

The HLA was developed by a consortium of stakeholders (government, corporate and academic) as a result of a U.S. DoD desire to decrease the cost of modelling and simulation development. The approach is to use a flexible architecture capable of supporting the reuse and interoperability of individual software simulations, called federates, in larger simulations called federations.

A federation consists of three elements: 1) one or more federates, 2) the supporting RTI middleware, and 3) a Federation Object Model (FOM). Each federate is a stand-alone software application. To operate as part of a federation, a federate's design must adhere to the rules of the HLA as defined in [2]. The RTI provides various services to the federates, enforces the HLA rules and controls the federation execution. Finally, a federation's FOM is a unique document and is supplied as a parameter to the RTI on start up of a federation execution. It is created by the federation designers to convey simulation design decisions to the RTI and federates. The FOM is an XML document. Essentially, it is a database that describes the data that can be exchanged between federates in addition to other static information. The Object Model Template (OMT) Specification [4] describes the format and syntax of the FOM and other object models used in the HLA. Because of its importance within the HLA and to the SIP-RTI, the FOM is described further in section 2.1.1 below.

The HLA supports two types of data representations to share information between federates. Each is defined in the FOM using a class hierarchy scheme chosen by the designers to meet the needs of the simulation. An HLA *object*[1] class is typically used to

[1] Note the use of the term "object" in the HLA is not the same as in current object oriented software languages such as Java. Objects in the HLA are defined hierarchically such that sub-classes inherit the data attributes of their parents; however, HLA objects define no explicit behaviour (they have no methods). References [1, 3, 4, 9] provide further explanation of the distinctions between OOP and HLA objects.

define persistent data representing enduring simulation entities important to more than one federate. An object class may have one or more *attributes* which may be updated to represent the state of the entity as the federation execution progresses. The RTI keeps track of each object instance created (until deleted) identifying which federates own its attributes and which federates are to be notified of changes in attribute values. It does not, however, maintain a record of past or current attribute values. Each federate must maintain its own internal representation of the objects and attributes it is interested in. The second data type is the *interaction.* Interactions are normally used to convey transient event data to federates. Interactions are not addressed in this work and not considered further in this document.

The HLA defines service APIs on which federates can call RTI services and vice versa. RTI calls are exposed by an *RTIambassador.* Each federate is provided its own RTIambassador object by the RTI. The HLA also defines a callback API that each federate must implement as a *FederateAmbassador.* The RTI makes requests on a federate and passes it information using the federate's FederateAmbassador. These APIs are described in [3].

The RTI provides six distinct areas of service and a group of support services. Each is discussed briefly in the following paragraphs. Coverage is terse intending to give a very high-level understanding of RTI capabilities.

Federation management services support the creation of a federation execution and allow federates to join and leave a federation. Services are also provided to save federation state, set up synchronization points and carry out related chores.

Data sharing in the HLA is based on a publication-subscription model that decouples producers from consumers. **Declaration management** (DM) services allow federates to indicate their intent to produce (called *publish* in the HLA) and consume (*subscribe*) data. The RTI uses this information to route object instance attribute updates between federates using the object management services.

Object management services are used to *register* (create) new object instances and to issue attribute data (*update* the attribute), via callbacks on federates' FederateAmbassador in accordance with each federate's current attribute subscriptions.

Ownership management services support the transition of the ownership of instance attributes between federates. Before a federate can update an attribute, it must *own* it. The owner of an attribute is responsible for updating it. Ownership applies to instance attributes (that is, a federate owns a given attribute of a single object instance). An attribute can have only one owner at a time.

Time management services allow the definition of a common concept of logical time and can be used to manage the progress of a federate in accordance with this concept.

Data distribution management (DDM) services support finer control of instance attribute and interaction routing between federates based on abstract regions.

Finally, various **Support services** provide a multitude of functions such as setting of various federation-wide switches, obtaining federate, object class and instance handles or names from the RTI for use in other service calls and other support functions.

A deeper understanding of how attribute-based data sharing is handled in the HLA is required to understand the crtiModel and implementation. This requires further discussion of the FOM, the creation of object classes and the transfer of attribute ownership.

2.1.1 The Federation Object Model

The classes specified in the FOM describes the data that can be exchanged among federates. The FOM is passed into the RTI as an XML file. The RTI parses the FOM to obtain the names of the object classes and attributes. Federation developers must use these same names in the federates they develop for any class instances they wish to

involve in inter-federate communications. Appendix E provides an example of a FOM based on the demonstration federation described in Chapter 7. Object classes are defined in the FOM using an inheritance scheme. In this scheme all object classes are sub-classes of the HLAObjectRoot object class. In the HLA, each object sub-class inherits the attributes of its parent. Only one parent per sub-class is allowed. The HLAObjectRoot contains only the HLAprivilegeToDeleteObject attribute (the owner of this attribute, for any given object class instance, may delete that instance). As a result, all objects have at least this attribute. Typically, however, other attributes are defined and used to share simulation information.

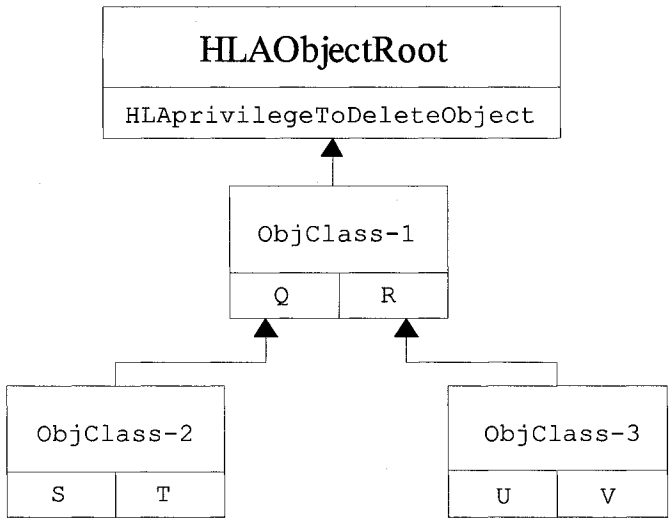

Figure 2-1 An Example Federation Object Class Hierarchy Diagram

The object class inheritance scheme can be depicted using an *Object Class Hierarchy Diagram*. An example is shown in figure 2-1 using a Unified Modelling Language (UML)-like symbology. Named rectangles represent HLA object classes. Each class's *declared* attributes, if any, are identified at the bottom the class rectangle. Solid arrows identify a class's superclass and indicate inherence by pointing to a class's parent; the sub-class inherits all of the parent's declared and inherited attributes. In figure 2-1 four classes are depicted. In addition to the HLAObjectRoot superclass, three user-defined classes are shown: ObjClass-1, ObjClass-2 and ObjClass-3. Each user-defined class

inherits HLAprivilegeToDelete. In addition, ObjClass-2 and ObjClass-3 inherit ObjClass-1's "Q" and "R" attributes.

2.1.2 Inter-federate Information Transfer

As indicated, the HLA uses a publish/subscribe mechanism to transfer information between federates using instance attributes or interactions. The following considers the use of attributes only.

Before a federate can instantiate an object class, the federate must declare its intent to publish at least one of the class's attributes. Note that the federate indicates its intent to publish specific attributes, not the entire object class. The federate may then begin registering object instances of that class.

In each new instance, the registering federate owns the attributes it has indicated it will publish. Attributes that have not been identified for publication by the registering federate, are *unowned.* Other federates that have declared the intent to publish the unowned attributes may request ownership of them. The owner of an instance attribute may give up ownership. These topics are discussed further in the next section.

Once published and registered, an instance's attributes may be updated by the owning federate. The RTI sends attribute updates to subscribing federates; they are said to *reflect* the new data. In order to receive attribute changes a federate must subscribe to that class attribute. Subscriptions are federation-wide; the subscriber will receive updates of all instance attributes of the same class and attribute. If a federate has subscribed at least one attribute of an object class, it is notified by the RTI of each new object instance of that class when the instance is registered; the federate is said to *discover* the new instance. Multiple federates may own attributes in an instance, but only the owner of a given attribute may update it.

An interesting concept in the HLA is that of object class type *promotion* on discovery. If a federate subscribes an attribute, it will discover all instances created of the attribute's object class. This is the case even if a child object is registered because the child will

have inherited the attribute as well. In this case, however, the instance will be discovered as the object class type that the federate subscribed rather than as the child's type (assuming the federate has not subscribed the child as well, in which case it will discover the instance as the child's object class type). Considering the figure 2-1 example again, if a federate, call it Federate A, has subscribed only attribute Q of ObjClass-1 and another federate, Federate B, registers an instance of ObjClass-3, the RTI will cause Federate A to discover the new instance as an instance of type ObjClass-1. This is referred to as promoting the instance. If Federate A subsequently subscribes ObjClass-3, all following ObjClass-3 instances will be discovered without promotion as ObjClass-3 objects. However, the original discovery will remain an ObjClass-1 type instance.

Note that no direct communication between federates takes place; all services are carried out by and through the RTI thus disassociating federates and supporting the reuse of a federate in different federations.

2.1.3 Transfer of Attribute Ownership

Federates may give up or receive ownership of an instance attribute using the related ownership management services. An owner may offer to *divest* ownership in which case the first federate to request it will be assigned ownership by the RTI. This is called the *push* model of ownership transfer. A federate may request to *acquire* ownership of a given instance attribute as well. Ownership will be transferred only if the owner agrees. This is referred to as the *pull* model. In neither case will the RTI force the removal or award of ownership.

The pull model is relevant to this research. The HLA defines various calls to support this approach including;

7.2[1] - Unconditional Attribute Ownership Divestiture
7.7 - Attribute Ownership Acquisition Notification†
7.8 - Attribute Ownership Acquisition
7.11 - Request Attribute Ownership Release†

[1] These numbers reflect the number of the call in [IEEE-2]. The dagger symbol (†) indicates a callback.

A federate may request (pull) ownership of an attribute by making an Attribute Ownership Acquisition call on the RTI identifying the attribute(s) it wishes to own. The requestor remains in an ownership request pending state until it receives ownership or cancels the request. If the attribute is unowned ownership is immediately granted and the requestor notified via an Attribute Ownership Acquisition Notification† callback. If the attribute is owned, the RTI will react to this request by sending a Request Attribute Ownership Release† callback to the current owner. If the current owner wishes to release the attribute it may[1] make an Unconditional Attribute Ownership Divestiture call on its LRC. This action will result in the RTI assigning ownership of the attribute to a requesting federate and the federate will, again, be notified via an Attribute Ownership Acquisition Notification†.

2.2 An Abstract RTI Implementation Model

Discussion of the crtiModel assumes a flexible, component-based, abstract representation of an RTI implementation as presented in [10]. The abstract representation is based on the use of LRCs, because in most HLA RTI designs individual nodes typically access the RTI through a locally instantiated RTI component. This generic model is shown in figure 2-2. In this model federates 'a' through 'n' participate in and comprise a single federation. Federates access HLA services using the RTI API, viewing the RTI as a black box. A federate has no knowledge of the how the RTI is implemented other than the services it offers. These services are made available to a federate via an RTIambassador which, in the model, is considered part of the LRC. The RTI is composed of one or more LRCs, which may be distributed. An LRC may support one or more federates (LRC_1, LRC_2, LRC_K), while others (e.g. LRC_3) may be used to provide specific support services to other LRCs.

[1] There are, in fact, other HLA calls that a federate may make in the scenario being described; however, this research only involves the calls identified.

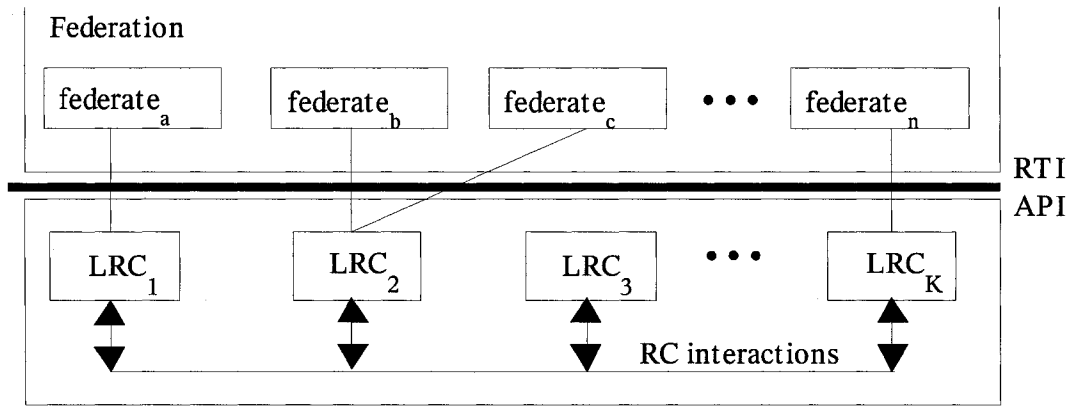

Figure 2-2 An Abstract RTI Model based on LRCs

The model supports a variety of implementations. For example, a centralized client/server implementation may use LRC_3 as the central server while the other LRCs act as distributed clients. A distributed implementation could use LRC_3 as the FedExec[1], while the others support distributed HLA algorithms.

2.3 An Overview of the Session Initiation Protocol

The Session Initiation Protocol is a session layer control protocol designed for use by applications to create, control, and terminate network-based communication sessions between two or more participants. It is intended to be used as an "out-of-band" control protocol; SIP signalling messages move in a different communication channel than the one transporting the media of the communication session being controlled. It is a robust, relatively simple and highly flexible protocol. Often used for VoIP, it can be used in any situation requiring the establishment of a communication session between two or more end points. Reference [6] defines the core protocol while other documents define extensions and more complex services based on its primitives (e.g. [11-13]). The protocol provides the following functions to support communication; determination of location of target end users and services or "discovery", determination if the target users wish to engage in the session, determination of session parameters (such as codec and media type), setup of the session and, finally, management of the session while it is active.

[1] RTI implementations often use a single, global process to support federates joining and leaving a federation execution. This process is called the FedExec.

Session Initiation Protocol end points are called *user agents*. They are usually implemented on a computer but may be implemented in cell phones, PSTN (public switched telephone network) gateways and other devices. An end point is composed of a *user agent client* (UAC) and a *user agent server* (UAS). The UAC sends *requests* such as invitations to join a session or announcement of the termination of a session to a UAS, which sends a *response* indicating its success or failure in carrying out the request. An end point operates in the mode appropriate for the current function it is carrying out. UAC and UAS behaviours are also implemented in SIP *proxy servers*, which are used in the network core to assist in routing requests, to find user endpoints and to provide other support such as billing services.

Successful requests invoke *methods* on the UAS. The SIP does not implement any specific services. It provides method primitives with which services, such as two-party communication sessions or conference call set-up, can be implemented. Reference [6] defines six methods in the SIP. Of these, INVITE is the most important and is used to initiate a communication session.

The Session Initiation Protocol uses an HTTP-like request-response transaction model. A *transaction* is comprised of one request and all responses to it. Transactions are relatively short lived; however, they can be used to set up a stateful peer-to-peer *dialog* between two user agents that lasts the duration of the session. Not all transactions must be part of a dialog. For example, OPTIONS, another core SIP method, does not establish a dialog, simply a short-lived transaction. Because dialogs maintain state, they may be routed through firewalls.

The Session Initiation Protocol uses text *messages* to carry requests and responses between UACs and UASs. TCP or UDP transport is typically used under the SIP to carry messages. In addition to other information, messages contain a header with fields to identify, the method they are requesting or response they are carrying, target and sender network addresses, and sequence information. Identification information allows the message to be identified as part of a specific transaction and, where relevant, as part of a

specific dialog. Messages may include further session setup (e.g. media type, codec identification etc.). Finally, the Session Description Protocol (SDP) [14] and Multipurpose Internet Mail Extension (MIME) types are supported and can be used to carry other information in SIP messages.

A UAS responds to a UAC request by sending one of a series of possible response codes in a response message. Response messages are similar to request messages but contain the appropriate response code signifying the result of the request.

The SIP provides various security services as well. The HTTP message authentication technique is used to allow UASs to request UACs authenticate themselves. Hop-by-hop message encryption using Transport Layer Security (TLS) or IP Security (IPSec) or end-to-end S/MIME-based (Secure MIME) tunnelling may be used to protect messages from manipulation, man-in-the-middle attacks, replay attacks and other security threats. Separate security techniques must be applied to protect media channels.

Chapter 3 A Literature Review of RTI Issues, Conferencing Architectures and Distributed Systems

The SIP-RTI involves research and implementation effort in three areas; the HLA and existing HLA RTI implementations, network-based conferencing systems, and distributed system design. This chapter begins by identifying some existing RTIs and describing them with respect to the main RTI issues addressed. This section of the chapter concentrates on RTI interoperability, as it is the focus of this work a although other RTI problems are discussed as well. Current conferencing frameworks and the use of the SIP in their design are then discussed. There is a large body of work in this area and, although a predominant approach seems to have emerged (centralized server), there are no universally accepted standards, and work remains before a generic approach will be agreed upon. In addition, while application conferencing is mentioned in the literature, the general thrust of existing research is in support of human conferencing (e.g. teleconferencing). Finally, distributed systems design is an extremely large research topic. As it is not the focus of this work but just an aspect of its implementation, it is addressed only briefly with respect to some of the issues faced in the realization of the CI's implementation.

3.1 Existing RTIs and RTI Design

In an effort to curtail the increasing costs of maintaining multiple M&S standards and the resulting myriad of simulations, the U.S. Department of Defence (DoD) directed the specification of the HLA and development of a reference implementation (RI). The Defense Modelling and Simulation Office (DMSO) [15] is the organization within DoD which undertook this task. After the release of several initial versions of the specifications and of developmental RTIs, this office released the DoD HLA 1.3 specification in March of 1998 [16]. In the same year DMSO also released the first complete RI of a HLA 1.3 RTI [17] referred to as RTI-1.3v1. It was released in the public domain (although not as open source), made freely available, and was supported by DMSO with further releases coming out over the next few years.

During this time the specification was commercialized and the IEEE 1516 series of specifications [2-4] were released in 2000. While there are still legacy users of the DMSO specified RTIs, DMSO advocates transition to the IEEE specification. (Similar in most regards, the IEEE specification diverges significantly from the 1.3 versions in the area of DDM.)

In 1999, RTI 1.3 NG (Next Generation) was released. This implementation differed from past ones in that it was developed by commercial entities, as opposed to by DMSO. It was still freely available and DMSO support continued. In August of 2002, however, DMSO discontinued support of both the RTI implementation and specification [18].

As a result of this history most RTI implementations used today are proprietary and produced by three companies. Pitch Technologies AB [19] in Sweden produces both 1.3 and IEEE 1516 compliant RTIs along with support tools. MÄK Technologies [20] in Cambridge, Massachusetts, produces competing products. Finally, Virtual Technology Corporation [21] in Alexandria, Virginia, produces predominantly RTI 1.3 NG products. As may be expected, these products are not open to the public, and information on their internal design is limited.

There are also some public RTIs available. Three efforts, in particular, are worth noting [22]. Researchers at the Georgia Institute of Technology in Atlanta developed a partial implementation of the 1.3 specification called the Federated Simulations Development Kit (FDK) [23-25]. Written in C++, its code is available to academic researchers as part of a university outreach program supported by DMSO. In fact, the FDK has been used as the basis of a Java development effort here at Carleton University [26]. The FDK does not support HLA ownership management or DDM services, concentrating rather on HLA time services (the leader of the FDK effort is Dr. R. Fujimoto, who also led the specification effort of the time management services for the HLA). The FDK provides a "toolkit" of modules, called the RTI Kit, from which users may select and build a custom RTI, selecting just those services desired.

The Extensible Run-Time Infrastructure (XRTI) [27] is another public RTI and the subject of a Masters thesis completed at the U.S. Naval Postgraduate School's Moves Institute. The XRTI was developed to solve various problems with the HLA in order that it might eventually be used to support large scale virtual environments. Reference [27] identifies and targets four problem areas in the HLA: 1) it is difficult to use, 2) it does not define a standardized inter-RTI messaging standard and thus RTIs based on it are not interoperable, 3) data types used to share information between simulation components (federates) may not be defined dynamically (i.e. while the simulation is running), and 4) there are no open source implementations available.

Taken in the order mentioned in the preceding paragraph, the XRTI solves these problems with 1) a compiler which automatically creates code stubs from the FOM and can be used by federate developers as a starting point, 2) by using a "novel bootstrapping methodology to define its low-level interactions in terms of an HLA object model" (discussed further later in this document), 3) by implementing a means to introduce new object and interaction classes into a running federation execution, and 4) by open sourcing the XRTI code.

The resulting RTI is implemented in Java and based on the IEEE 1516 specification. Where necessary this specification is extended to support the solutions just mentioned. The XRTI does not currently support ownership management, time management or DDM services, and, although [27] indicates an intent to continue its development further, the XRTI could not be found on the web, nor was its web site [28] available during the course of this writing.

Finally, an ongoing project at ONERA (the French Office National d'Études et de Recherches Aérospatiales) has resulted in the CERTI RTI [29-32]. This RTI is a full implementation of the DMSO 1.3 HLA specification in C++. It is based on a local architecture of communicating processes. Local RTI components are used to support a federate on a single machine. A global (central) RTI component supports communication amongst local components and implements distributed algorithms needed to support the

HLA. A high performance version of the CERTI RTI, called HP-CERTI, was under development in 2004 [33].

3.1.1 RTI Interoperability

RTIs from different vendors do not interoperate. It is not possible for federates running on different RTIs to be part of the same federation without some other form of support or translator between them. This is because the algorithms and communication mechanisms used by the different RTI developers in implementing the HLA services are different. This is possible because the HLA is an *API standard.* It does not specify how the API services it defines are to be implemented; rather, it views the RTI, the realization of this API, as a black box. A key requirement for interoperability between components in a distributed system is that they all use the same algorithms in implementing the services provided by the system [5].

Within the M&S community, this issue is almost as old as the HLA itself. In 1999, the Simulation Interoperability Standards Organization (SISO) [34] RTI Interoperability Study Group provided one of the first in-depth reports on the issue [35] and, although it presented detailed analysis of just what interoperability means and indicated its importance to the future of the HLA, the report made no concrete recommendation about how to achieve it.

Since then, other papers have raised the issue, discussing it from both a technical [5] and a commercial perspective [36], while others have proposed solutions. The suggested solutions generally fall into two categories: the use of bridges or gateways between heterogeneous RTIs, or the addition of a *wire standard* to the HLA specification.

A wire standard forces system components to use the same algorithms by defining the syntax and semantics of each inter-component message in a distributed system [5]. The work presented in [37], and the XRTI discussed earlier, are examples of wire standard solutions.

In [37] (and the related [38]) a specification is presented which defines message formats and content. These have been used to implement a limited HLA RTI based on the 1.3 HLA specifications. Ownership management, DDM and time management services are not supported. This effort led to the formation of another SISO study group tasked with investigating the usefulness of development of an "Open RTI Protocol" messaging standard. The final findings of this study group [39] were inconclusive.

In the XRTI, communication messages are encoded according to a new object model, called the Bootstrap Object Model (BOM). The BOM is based on the Object Model Template tables provided in [4] with the addition of new objects to identify messages that are not covered by the encodings found there. The result is a set of message encodings similar to those of [37]. Note that the Bootstrap Object Model used in the XRTI is not the same as the Base Object Model [40].

The other commonly advocated approach to the interoperability issue is the use of RTI bridges or gateways between different RTI implementations. In this solution, a bridge or gateway would be developed for each RTI-type pair used to support a federation. Limited implementations of this solution are commercially available from MÄK and publicly [41, 42], although the latter work does not support DDM or time management services. The development of a standard defining the requirements of a RTI gateway is suggested in [43].

Further approaches to RTI interoperability also exist. The use of Common Object Request Broker Architecture (CORBA) technology [44, 45] could eventually lead to an interoperable RTI as [10] suggests. The authors of [46] propose an approach to doing so using a central Object Request Broker. Because CORBA's Interface Definition Language can be compiled to any programming language, interoperability between RTI implementations designed to use this approach is achieved. While not targeting this objective specifically, [47] mentions the actual implementation of a CORBA-based RTI. A related approach is discussed in [9, 48]. In this effort, the software objects that

implement HLA objects are transferred between LRCs. The work does not seem to have been continued.

In [49] the author advocates a two-pronged approach to interoperability. The first prong is technical and advocates the development of RTIs that support dynamic-link-compatibility (a solution that, rather, supports interchangeability, not interoperability, as [5] points out). The second prong advocates a more precise HLA specification and more cooperation between RTI developers, federate developers and federation program managers, ostensibly resulting in RTIs that are similar internally (and more closely match the requirements of M&S programs).

One intriguing approach is the Extensible Modelling Simulation Framework (XMSF) [50, 51]. The object of the XMSF is to use existing Web-based technologies within its extensible framework to allow interoperability between RTIs over the web. This work would eventually allow users to assemble federation executions from existing federates running non-stop on the Web. An example implementation, based on [52], using the Simple Object Access Protocol (SOAP) [53] on top of the Blocks Extensible Exchange Protocol (BEEP) [54], has been developed. The SOAP uses XML to exchange data via remote procedure calls. The BEEP supports the definition of application specific network communication protocols on top of TCP/IP. In the XMSF exemplar, SOAP services are used to make remote invocations on a server-based RTI through a local (client) stub. The BEEP is used to define the necessary communication services. The BEEP also allows the definition of security services and, thus, this approach to interoperability might also lead to a solution of this RTI issue.

3.1.2 Other RTI Issues

Current RTI's have other issues as well, and weak support for security is a commonly mentioned one. The authors of [55] describe some of the concerns. They generally have to do with the protection of simulation data (interactions, objects and attribute data). The IEEE 1516 standard does not include security features and, as a result, few current implementations support secure transfer of simulation data or other security capabilities.

Approaches have been suggested including the use of the SIP [56] and web-based security services [50, 51]. The CERTI implementation provides security using a variety of secure protocols in its implementation of the RTI 1.3 specification [57] while [55] proposed the use of IPSec to protect all RTI inter-node communications.

Related to the issue of security is the fact that the HLA does not include capabilities to support firewall traversal if the UDP transport protocol is used [58]. No direct solution to this problem has been proposed other than the use of the SIP [56, 58]. An example of how this might be done is given in [56] while [58] discusses the topic further.

The discovery of federation services (such as the FedExec) and/or federation participants is another issue for RTIs. Reference [59] refers to this problem and presents a framework that might provide a solution. Based on the Grid services model [60, 61], specifically the Globus Kit GT3 (a toolkit supporting Grid services creation [62]), this solution uses a central "Index Service" which can be used in the dynamic discovery of not only HLA services, but encapsulated federates as well. This is an unrealized framework at this time and the lack of RTI discovery services remains an issue.

Finally, the requirement for an open source RTI has been debated for some time. The authors of [22] indicate the possibility that existing commercial companies may not be able to recover from the damage due to the competition of an open source RTI but that this is unlikely. On the other hand, [36] describes the threat to the HLA specification if it does not continue to evolve - its ultimate demise in the user community – a fate that will certainly damage commercial enterprises. The author of [63] presents various arguments for and against the development of an open-source RTI, ultimately concluding that the M&S community would benefit from the availability of such a package. This conclusion is based on arguments similar to those presented by [36]: the need for the HLA to evolve to survive and the expectation that RTI reliability and quality will increase from multiple, independent, developers reviewing and working on the same implementation.

3.2 Conferencing

A *conference* is a set of *members* or *participants* that share a common interest. A conferencing service or application provides *conference setup* services. These typically provide at least for the creation, destruction and discovery of conferences. A means to allow members to join and leave a conference and to define their privileges is also provided. Finally, some means to handle resource contention, often referred to as *floor control*, is also supplied. Together these services are referred to as *conference control* and are usually based on a *conferencing policy*; a set of rules which may define when a conference can be created, conference types, member types, when and what type of a member can access a resource etc[1]. In the work presented here, members represent a software application's interests (as opposed to a human's), namely a federate's interests, and conferences are used to communicate these interests and share related information.

There is a large body of work, particularly in the Internet and SIP communities, studying the challenges of implementing network-based conferencing. This research centers on the development of conferencing frameworks and includes research into architectures and conference control mechanisms. The SIP was, in fact, originally developed as a large-scale, multiparty conferencing protocol [64] and, therefore, is involved in much of this research. A protocol-agnostic, universal framework and related components, such as exemplified by [65], are the ultimate goal of this area of research.

Much of this research is being done by various working groups of the Internet Engineering Task Force (IETF) [66] and takes the form of Request For Comments (RFC) or Internet-Draft documents. As such, they represent works in progress. This research area is still quite new and without a universally applicable set of standards. The next two sub-sections present the main aspects of this work, particularly those focused on the use of the SIP in implementing conferencing mechanisms.

[1] All of the preceding definitions are based on similar definitions found in [67-69].

3.2.1 Conferencing Architectures

The authors of [64] define various conferencing models supported by SIP. The models are defined by both the architecture and control methodology used. Six are defined. These models have been further generalized to two: the multicast model and the mixing model [70] which, in turn, relate directly to the *loosely coupled* and *tightly coupled* models defined in [71]. This more recent summation of conferencing models adds a third: *fully distributed multiparty conferencing.*

In the loosely coupled model, there is no central point of control nor is any means of coordinating signalling (control) between participants used. Implementations based on this model typically use multicast groups to disseminate information. This model contrasts sharply with the tightly coupled model. The tightly coupled model is based on the use of a central point of control called the *focus*. All members have control links to the focus and it maintains control signalling to them. Typically, the focus also handles the mixing of media data, although this may be done using end-user mixing as well. In this latter case, mesh connectivity among participants is maintained by the participants in order to distribute the media data once mixed. Finally, in the fully distributed multiparty conferencing model, there is again, no central point of control; each member has a signalling channel to all other members and control is distributed amongst them. The result is a full mesh topology which is used both to handle control and media distribution.

While most current conferencing research is focused on the tightly coupled model [70, 72], interest is mounting in distributed architectures as a result of their fault tolerance and elimination of the communication bottleneck that a central server creates. The authors of [70] believe a *full-mesh conferencing* model is required to support smaller, more impromptu conferences. In this distributed model all conference members have the same control and capabilities, none are unique. New members join when invited by an existing member through the establishment of connections with all other members. Existing members are identified to the new member by the one who invited it. To handle scenarios in which a participant represents an end user with limited bandwidth (such as a home user with a dial up link) a hybrid version of the full mesh topology is suggested. It

is based on edge servers each of which supports multiple end users. In the hybrid version a full mesh would be implemented among the servers but not end users.

Common to all models is a need to provide conference control mechanisms in the implementation of the model. The next section considers this aspect of conferencing.

3.2.2 SIP-based Conference Control

Conference control is a core component of any conferencing framework. Conference control includes conference establishment and user management including the addition and deletion of members and manipulation of their rights and access to resources.

As the de facto standard Internet session control protocol [67] the SIP, along with related supporting protocols such as SDP and SOAP, is often suggested as the best method of implementing conference control mechanisms. The general approach of research in this area suggests the use of basic SIP methods and custom SIP extension methods to implement the control signalling required. Examples of this approach are given in [67, 70, 71, 73]. Reference [74] provides detailed SIP user agent call control flow diagrams for various conferencing services.

Typically, in these SIP-based solutions, a conference server supports one or more conferences, each identified with a separate conference URI. Depending on the model used, to join a conference potential participants might call INVITE on this URI or the server or another, third party, might call INVITE on them. A SIP negotiation process then takes place using SDP to negotiate the prospective participant's rights, after which it is authorized to join the conference. Other SIP methods are used in ongoing member maintenance (e.g. dynamically changing user rights) and floor control, while a separate server might be used to initially establish the conferences and conference URIs. The SIP is also used in the negotiation and set-up of media (session) channels according to the topology and mixing policies defined by the conference model being implemented.

While the preceding paragraph describes a generalized approach to implementing SIP-based conference control, the referenced papers present a multitude of variations within this basic methodology. However, in all cases the SIP is used as the fundamental control protocol and is ingrained into most, if not all, functions carried out by the conferencing application.

3.3 Distributed Systems

In the development of network-based services or applications, two general design approaches are possible: *distributed* or *centralized.* A distributed design approach is often selected because it increases fault tolerance and eliminates the bottlenecks that centralized servers can create.

A distributed system generally refers to a system that uses multiple, separate, stand alone, computer nodes working together to accomplish some common task and connected by a relatively slow (in comparison to CPU speeds and cluster communication techniques) communication network. Nodes in a distributed system are often referred to as loosely coupled [75]. In contrast, clusters of computers located in close proximity to one another and supported by dedicated high-speed links or a multiprocessor machine in which each processor shares common memory are typically referred to as a tightly coupled system. This research is concerned with loosely coupled distributed systems.

As [73] points out, maintaining a consistent view of distributed state is a classic hard problem in computer science. As distributed systems per se are not the focus of this research, this section will simply cover possible approaches to some of the design issues faced in implementing the CI. The selected approaches are identified and discussed in Chapter 4 but generally reflect two design philosophies; a desire to keep it simple and a desire to make it work.

3.3.1 Communication Between Nodes

A fundamental issue in designing distributed systems is how to implement inter-node communication over the network. The classic approaches to communication in loosely coupled systems are *message passing* or the use of remote procedure call (RPC) techniques. Recently, however, more sophisticated techniques have been developed to support loosely coupled systems including the use of distributed objects and remote method invocations (RMI) such as provided by the CORBA, Distributed COM (DCOM) and Java RMI systems [76, 77].

In general, message passing is more effective and efficient in smaller, simpler systems with small, simple command sets. As the command set increases in size and complexity the complexity of using the low level techniques required to implement message passing may outweigh the bandwidth and speed gains it provides. In these situations distributed object techniques are usually more suitable [76].

3.3.2 Asynchronous vs. Synchronous Operation

Communication techniques such as RPC and RMI are typically synchronous; the calling system blocks until the called system responds. Message passing, generally, results in asynchronous communication exchanges. Once the request is sent, the caller continues its operations until a response is received, as a separate message, from the callee. This has the benefit that the caller can continue operations while the callee processes the request. The callee may respond immediately or after some time. Message passing systems may also be used to implement synchronous systems, if callers wait for a response to a request before continuing their operations. In such cases the caller may be made to progress in "lock-step" with its peers.[1]

3.3.3 Peer-to-peer vs. Client-Server

In a client-server architecture one node acts as a server providing services to other nodes which act as its clients. Although this architecture may be used in distributed designs, strict client-server architectures are not distributed but, rather, centralized. In a fully

[1] The use of the terms client (caller) and server (callee) have been intentionally avoided in this discussion as the client-server architecture has not yet been addressed.

distributed system each node can carry out all or most of the functions that its peers can and is not subordinate to its peers. That is, each node can act as a server or a client as required. Such systems are called peer-to-peer (P2P) systems. Peer-to-peer systems are fully distributed and, as a result, provide the benefits of a distributed system mentioned earlier.

Chapter 4 Towards Solving the RTI Interoperability Problem: The SIP-RTI

The HLA RTI has various shortcomings. The fact that different RTI implementations do not interoperate is arguably the most important of these, certainly it is the one that has received the most attention. A solution to this problem, ideally, would be a generic interoperability solution capable of supporting distributed systems in general. The ideal generic solution is beyond the scope of a master's thesis. As a result, this research is constrained to the initial exploration of one promising approach, called the SIP-RTI, that may eventually lead to an ideal generic solution to RTI interoperability.

This chapter first carries out an analysis of the material presented in Chapter 3 identifying the specific problems with the HLA and current RTI implementations, and aspects of distributed conferencing that are addressed in this research. An ideal generic solution, the larger vision of this research is then described. The final part of this chapter states the thesis of this research, introduces the SIP-RTI solution, and presents the contributions resulting from its development.

4.1 Analysis of Work on Current RTI Issues, Conferencing Applications and Distributed Systems Design

The following sub-sections analyse current work attempting to solve the RTI shortcomings discussed in Chapter 3, and the use of a distributed application conferencing system.

4.1.1 Current Attempts to Solve the Interoperability Issue

In its use here, the term *interoperability* refers to the ability to provide communication between systems. *Functional interoperability* between systems may be implemented upon a common communication mechanism that provides this ability. General RTI interoperability requires that a common approach to inter-LRC communication be taken by each RTI. The HLA does not specify a common approach because it defines an API but not how this API must be implemented. As a result, heterogeneous RTIs are not interoperable.

Various approaches to this problem have been suggested, each attacks it by, essentially, providing commonality at some level in the ISO/OSI reference model. The most recommended [27] is the addition of a wire standard to the HLA specification. This approach defines a common communication mechanism below the RTI API at the network level. So far, such an approach has been unsuccessful mainly because, as [49] points out, it will limit flexibility in RTI design by dictating the algorithms that an RTI implementation must use in implementing the services called out in the HLA. The use of distributed object technology such as CORBA has also been investigated. This approach imposes commonality at an intermediate level in the reference model, but still below the RTI API. Unfortunately, these technologies provide no means to support the time services required by the HLA and thus are considered inadequate methods of implementing the HLA. Finally, the efforts identified in [50, 51] are based on existing Internet-centric protocols again providing commonality at an intermediate level. However, these technologies are not considered as capable, or as simple, as the use of the SIP [56].

RTI bridges offer a possible solution above the RTI API. A bridge federate connects to two RTIs and translates inter-LRC communications, effectively implementing a "common" communication mechanism between the RTIs. Both, industry ([20]) and public efforts, have tackled the interoperability issue with bridges. However, this solution has some serious problems. First, as [27] points out, with N different RTIs the number of bridges required to provide interoperability is between O(N), if an intermediate bridge is used, and $O(N^2)$, if a dedicated bridge is used between each RTI pair. In addition, Dingel et al [78] identify a multitude of other problems with the implementation of bridges including the possibility of deadlocked federates. Also, [49] states HLA functionality (aspects of or complete API services) is lost when a bridge is used between RTIs. All of this leads to the conclusion that this solution may be acceptable to provide interoperability for two or three RTI implementations but not universally.

4.1.2 Loss of the Session Layer

An explicit session layer is missing from the HLA specification. This also contributes to the lack of a common approach to RTI communication requirements. Therefore, again, RTI interoperability is not supported.

The ISO/OSI 7-layer interconnection reference model defines a distinct session layer that should act as an interface between disparate applications running above it and the communication layers implemented below it. As [79] states, the session layer:

> *"Deals with creating and managing sessions when one application process requests access to another application process (e.g., Microsoft Word importing a chart from Excel)".*

A 5-layer model, sometimes called the TCP model, which collapses the upper (application, presentation and session) layers into a single "process and applications" layer, often replaces the 7-layer model. As [79] points out, this model has resulted in various problems for the applications which follow it in their design. These issues include interoperability problems. Reference [5] indicates that a slightly different reference model has emerged into which the HLA fits nicely. It is shown in figure 4-1 and is, essentially, a 6-layer reference model with the primitive communication layers collapsed into a single Communication Layer.

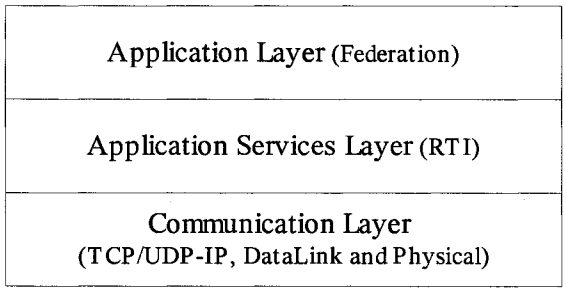

Figure 4-1 A 3-Layer Reference Model Applied to the HLA

(modified from [5] to show where HLA components fit in)

The result of this reduction of layers, in both models, is that a distinct session layer, which would have encouraged interoperability, has been eliminated and drawn into the layer or layers above it. In the case of figure 4-1, the definition of the HLA API results in

the session layer being implemented in the Application Services Layer. The lack of an independent definition or specification of this layer results in a developer-dependent implementation of how the lower communication layers are used by each application. This results in a lack of interoperability among different applications. In the case of the HLA, because its specification does not define the use of the lower layers, even among RTIs, a common approach to the required communications is not followed and the result is a lack of interoperability among RTIs.

4.1.3 RTI Security

The HLA defines no mechanism to protect inter-LRC communications, potentially exposing sensitive data when used in a distributed simulation. Related to this issue, firewall traversal, when the common UDP transport protocol is used for inter-LRC communication, is not supported.

Current solutions to providing security (e.g. [55, 57]) dictate the protocols used leaving little or no flexibility in their selection. The work in [50, 51] might provide solutions to both of these RTI problems, however, while it is yet to be applied to these issues, the SIP is considered a simpler and more capable approach to them [56].

4.1.4 The Problem of Discovery

Dynamic discovery of HLA participants (federates or services) is another issue not adequately addressed by the HLA specification. Few approaches have been proposed to this issue. The only one found in a review of the literature is discussed in [59].

4.1.5 Lack of an Open Source RTI

A final HLA problem identified in Chapter 3 is the lack of an IEEE 1516 open source RTI implementation. Various public RTI efforts have been mounted; none have resulted in a full IEEE 1516 RTI implementation. An open source RTI is one way to assist evolution of the HLA much as it did the evolution, and overwhelming popularity, of the Linux operating system. A truly open, interoperable RTI can only increase the possibilities for

industry and research while enhancing the future of the HLA and circumventing the possibility, suggested by [36], of its demise.

4.1.6 Lack of an Application Conferencing System

Although there is a large body of work done and ongoing in the area of conferencing, there is little consensus on the best approach to use, no standard architecture, and few actual implementations of the theories and frameworks discussed in Chapter 3. In addition, current work generally targets the human conferencing problem; application conferencing software has not yet emerged.

4.1.7 Distributed vs. Centralized System

Most current conferencing system work is based on a client-server, centralized paradigm. In this work, however, a fully distributed implementation is more desirable because the conferencing mechanism will be used to support the HLA, which is itself a distributed architecture and because of the benefits of fully distributed systems already mentioned (scalability, robustness, bottleneck elimination).

4.2 An Ideal Interoperability Solution

This work is motivated by the vision of a general interoperability solution. Ideally, the solution to the issues just discussed would be one that can be used with any set of distributed systems providing interoperability and security, not only to RTIs, but to other applications as well.

The ideal solution should provide a common communication mechanism, implemented via a distinct session layer and based on the semantics of a publicly specified model. The definition of a public model will require all applications that adhere to it be designed and implemented in accordance with common semantics thus promoting interoperation. In addition, the solution should provide some means for the flexible use of security protocols and the ability to traverse firewalls regardless of the transport protocol used.

4.3 The Thesis

The thesis of this work is:

> *"An RTI implemented above a SIP-based application conference takes a valuable step towards the ideal interoperability solution."*

4.4 The SIP-RTI Solution

The solution developed in this research is the SIP-RTI. The SIP-RTI is a SIP-enabled RTI comprised of two components. The first is the general-purpose, distributed communication system called the Conferencing Infrastructure or CI. The second component is a custom-built middleware package designed to use the services of the CI to create an IEEE 1516 RTI. These distributed components are implemented by local components instantiated on each node participating in a federation. The SIP Local Conferencing Infrastructure Component (sLCC) implements the local component of the CI while the SIP Local RTI Component (sLRC) uses the services provided by its underlying sLCC to establish, manipulate and maintain conference state in accordance with the crtiModel which is a conference-based model of the HLA semantics.

To establish a SIP-RTI node an sLRC is first instantiated by a federate. When the first call to the HLA "Create Federation" service is made on an sLRC it instantiates the node's sLCC. The sLRC then uses the sLCC's services to create a local copy of the conference hierarchy, called a *conference map*, which is used in the CI in accordance with the crtiModel and the FOM. This is described in detail when the crtiModel is presented in Chapter 6. The new sLCC then joins the CI by using the SIP to create signalling channels, which are then used to negotiate the setup of session channels with all participating sLCCs.

Figure 4-2 presents the stack implemented on each node running the SIP-RTI. The sLCCs use the JAIN-SIP stack [80] to set up signalling channels among themselves creating a mesh network. These channels are implemented as common TCP sockets-based connections. The SIP signalling channels maintain a SIP dialog for the duration of

the federation execution. In this implementation, these channels are used only in setting up the session channels. Future versions of the SIP-RTI may use the signalling channels to tear down session channels on completion of the federation execution and to support additional features such as security. The session channels are also set up as TCP sockets-based channels. The session channels are used during the federation execution to transfer CI state data and user data between sLCCs. The way in which the SIP is used to implement the SIP-RTI is referred to as *SIP-nominal* [7, 10]. This approach allows the extrication of the session layer services from the RTI implementation and results in a general-purpose communication system. The study of alternate approaches to using the SIP and alternate (non-SIP) control protocols would be valuable future research.

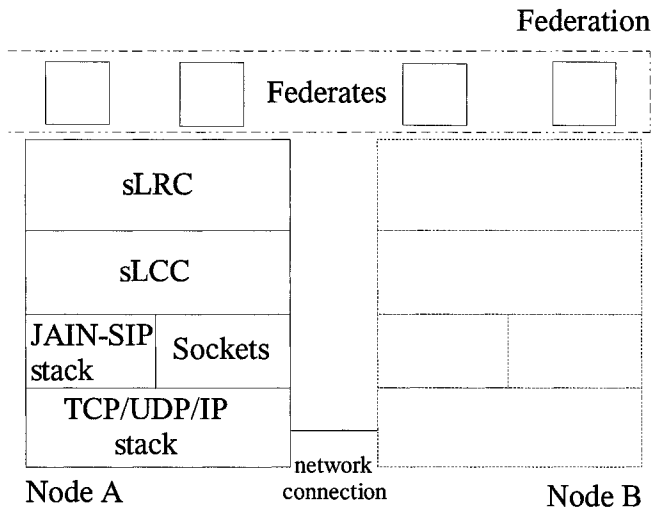

Figure 4-2 The SIP-RTI Components

The sLRC on each node provides its federates with the HLA services called out in [3]. Only those services necessary to support the aircraft demonstration federation have been implemented. Addition of the other services is left as future work. The sLRCs maintain no dynamic state information; the CI maintains all dynamic state. The sLRCs use the CI to implement the crtiModel thus implementing the algorithms required to provide the HLA services. Each node participating in the CI supports a single sLCC. Further, each node participating in the SIP-RTI supports a single sLRC. Finally, each sLRC may support multiple federates.

The SIP-RTI satisfies the thesis of this work due to the ability of the CI to provide communication among, not only RTIs, but other applications as well. In addition, because the CI uses the SIP to establish communication session channels among sLCCs, the SIP-RTI might eventually be used to provide solutions to other RTI issues as well as interoperability.

4.5 Contributions

The theoretical and technical contributions made by the SIP-RTI are described in the following sub-sections.

4.5.1 Contribution 1 – A Common Communication Model

The first contribution of the SIP-RTI is the definition, development and implementation of a common communication model and mechanism. The selected conferencing architecture allows the HLA semantics to be represented easily. In this model, communication takes place when *speakers* make *announcements* to *listeners* participating in the same conference. The implementation of the resulting conferencing model is the fully distributed Conferencing Infrastructure.

A significant characteristic of the CI is that it is application independent and can be used to support applications requiring a communication solution (even human-based conferences). The CI realizes the needs of an RTI, while providing an approach to the issue of universal interoperability as well! In addition, its development contributes a working implementation in the relatively unaddressed area of application conferencing.

The resulting system, has a "tightly coupled, fully distributed multiparty" topology. It is "fully distributed multiparty" because the CI's local components fully distribute state within the CI and are mesh connected. With the exception of how new nodes are added to an existing CI, all nodes act as both client and server. The system is "tightly coupled" because one entity, the CI, controls all participants (the sLRCs which act as surrogates for the federates). While similar to the unrealized hybrid solution suggested in [70], this is a unique solution to the conferencing problem. It has the benefits of both tightly coupled

systems (full knowledge of all relevant state in each node even although it allows participants to join and leave a conferencing session at any time) along with those of distributed design.

4.5.2 Contribution 2 – Reintroduction of the OSI Session Layer

The second contribution of the SIP-RTI is the reintroduction of a session layer definition and service. The CI is based on a simple conferencing model and is implemented at the OSI session layer level using the SIP (a session layer protocol) to establish communication among nodes (applications) participating in a conferencing session.

4.5.3 Contribution 3 – A Conferencing-based RTI

Another contribution of the SIP-RTI is the design and implementation of an RTI built to use the CI. The HLA semantics are modeled on the simple conferencing model. The result is the crtiModel, a new, intuitive way to describe this architecture. The crtiModel is implemented by the SIP-RTI. The SIP-RTI makes use of the CI's services to implement the subset of the HLA services required to support the proof-of-concept demonstration (see sub-section 4.3.1.5 below).

4.5.4 Contribution 4 – An Open Source RTI

It is hoped that the contribution of this initial development work on the SIP-RTI will eventually lead to a complete IEEE 1516 RTI. The SIP-RTI code will be released as open-source software and plans are underway to continue this project in the Systems and Computer Engineering Department of Carleton University. Furthermore, the CI component will be released as an easily removed subsystem. This will support future research into general application interoperability beyond the HLA-specific SIP-RTI. In addition, the CI's use of the SIP may support the development of solutions to the security, firewall traversal and discovery issues mentioned earlier, applicable again, not only to RTIs but to distributed systems in general.

4.5.5 Contribution 5 – Demonstration of the Developed Models and Implementations

The final contribution of this research is a demonstration that shows: the viability of the crtiModel to support HLA semantics, the functionality of the SIP-RTI, and the potential of the CI to support interoperability. Ownership management services are used in this demonstration. It employs an air traffic control-based federation that simulates the handoff of aircraft flight control between two air traffic controllers. In this simulation, ownership of an "AIRCRAFT" object class instance's "Controller" attribute is transferred to a second controller when the instance's "Position" attribute indicates the aircraft has entered the second controller's airspace. The SIP-RTI supports all of the IEEE 1516 Federation, Declaration, Object, Ownership management and Support services required by this demonstration. To demonstrate the CI's ability to support inter-application interoperability a simple monitor application is also implemented. Not part of the federation, this program connects to the CI and receives all position update announcements, thereby demonstrating interoperability between the RTI and monitor applications. The demonstration software is described in Chapter 8.

Chapter 5 The Conferencing Model and Design of the SIP Local Conferencing Infrastructure Component

While the intent of the sLCC is to provide the services required to support the crtiModel it was developed as a generic conferencing mechanism not just targeted at providing RTI specific capabilities. The result is a fully distributed system capable of supporting conferencing requirements in general.

After presenting the conferencing model, this chapter presents the requirements the sLCC must fulfill. A high level study of its design and actual implementation, identifying its significant sub-components (classes) and their functions, is then carried out. This portion of the chapter also describes inter-sLCC communication techniques. Because they support a distributed system, sLCCs must maintain both their own local state, and the subset of global state information that is of importance to them. The bulk of the chapter describes how this is done. The design is complete, however; because a full implementation would have taken an inordinate amount of time to realize, the complete design is not implemented in the current CI and is possible future work. Only those portions required to support the ownership transfer case study are implemented and discussed in this document[1]. In addition, to maintain the focus (and brevity) of this chapter the implementation details presented are discussed at a high level. Reference [81] provides a detailed description of the complete design while [82] provides further, lower level, details of the implementation. The chapter ends by discussing the synchronization of commands. This is required by some inter-sLCC commands to properly maintain global state.

5.1 The Conference Model

This section describes the conference model used in implementing the sLCC. It also expands slightly the definitions of conferencing terms given in Chapter 3 and presents further terms used in this chapter and in describing the crtiModel in Chapter 6.

[1]This constitutes implementation of about 2/5th of the design and is itself approximately 6500 lines of (commented) code!

A *conference* enables communication among multiple parties with common interests. Concurrent conferences may be created at the same level or as sub-conferences. Parties interested in a conference participate in it as *attendees*[1]. Parties interested in more than one conference may send attendees to each. Attendees may play the role of *speaker* (S) or *listener* (L). A speaker may *announce* information to listeners. To make an announcement the speaker must use a *floor* (F). There may be more than one floor in a conference and attending listeners hear all announcements from each. Announcements made in a given conference are not heard in its super or sub-conferences. Announcements provide the conference model's ability to support communication between attendees.

Floors may or may not be *owned* depending on their type. The model supports two types of floors; *available* (A) floors and *open* (O) floors. An open floor may be used by any speaker to make an announcement. The term "available" floor is meant to imply that these floors may be available for ownership transfer or acquisition. Available floors are subdivided further into two types: *owned available* (oA) floor and *unowned available* (uA) floor. An unowned available floor is a floor that has been created in an unowned state or has been divested by its owner; it is available for immediate acquisition by a user with a speaker in the conference. Ownership of an owned floor may be transferred to another attendee if the owner is willing to release it. The conference *moderator*, on behalf of the requesting attendee, negotiates release with the owner. In both cases the conference moderator ultimately grants ownership to the new owner.

5.2 sLCC Design Requirements

To realize the requirements of this conferencing model the sLCC provides the ability to:

1) create and destroy conferences,
2) place and remove attendees in/from a given conference,
3) place and remove floors in/from a given conference,
4) maintain identification of a floor's owner,
5) support floor status changes (ownership transfers), and
6) make and deliver announcements made in a conference to all affected listeners.

[1] An attendee is the same as a "member" or "participant"; terms used in the literature and in the definitions presented in Chapter 3 of this document.

With these simple capabilities, the sLCC may carry out all the functions required by the crtiModel and support a multi-federate federation execution running on a single node. To support a multi-node, distributed conferencing infrastructure, a means to share current state between sLCCs is provided. The approach chosen minimizes network traffic while sharing global state information necessary for an sLCC to make local decisions. Finally, a method to identify and connect to other sLCCs participating in the CI is provided. The following sections discuss how all of these services are implemented in an sLCC.

5.3 A High-Level View of the sLCC Implementation

The term *user* is applied to any application that calls the sLCC API. User applications use this API to create conferences, place attendees and floors in them, and apply the conferencing paradigm to carry out inter-user communication. The sLRC is a user of the sLCC. In this case, however, the sLRC acts a *user surrogate* representing multiple end users; the federates. Figure 5-1 provides a high-level view of an individual sLCC, identifying the components discussed below.

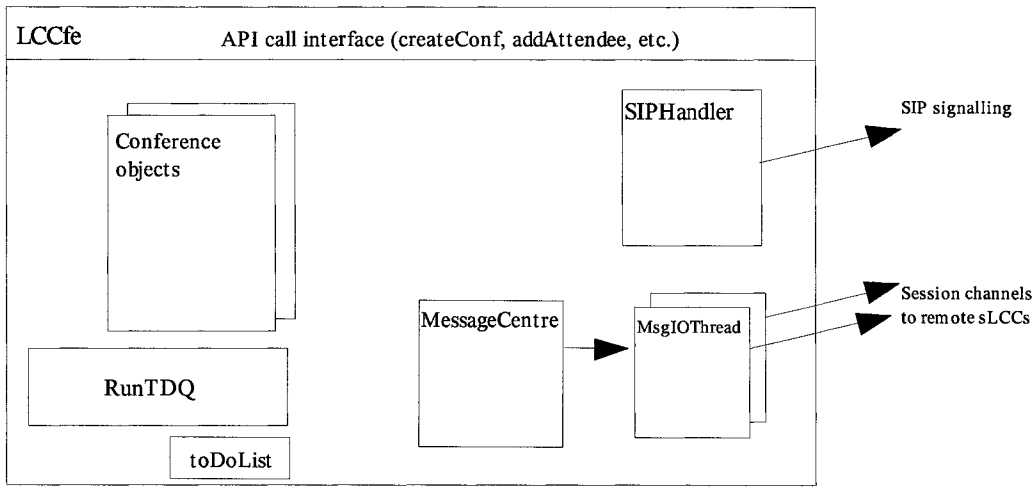

Figure 5-1 Main sLCC Components

The sLCC API is exposed to users by the LCCfe ("front end") object and defines calls which provide the six basic functions identified in section 5.2 and others needed to

support them[1]. Appendix A presents a listing and description of all API calls supported. Before a user may call any (other) sLCC service, however, it must request a "callerID". This ID is used in all further calls to the sLCC. The callerID identifies the caller as a unique and valid user and is used by the sLCC to identify its attendees and any floors they own.

The SIPHandler manages all SIP transactions for the sLCC. The MessageCentre implements a server, which allows other sLCCs to create application sessions with the sLCC once SIP negotiations have been completed by the SIPHandler. New application sessions are supported by separate "MsgIOThread" thread objects that implement socket-based input/output methods and are spun off by the MessageCentre's server.

The RunTDQ is also a separate thread. Along with the toDoList queue it handles the synchronization of some user calls on the sLCC API. They must be synchronized with responses from the other sLCCs to ensure proper state is maintained between all participating sLCCs. This is discussed further in section 6.6.

Conference objects handle all state changes, maintenance of local and global state information and other activity related to carrying out API calls. They, effectively, each implement a conference. All user conferences are subordinate to the ROOT conference, which is under the sLCC's control.

After presenting the terms that will be used, the following sections expand on these components and related design issues.

The conference map is the structure of parent and child sub-conferences created in each sLCC. In typical use, the conference map is static and the same on all sLCCs participating in the same CI. It is established in each sLCC by the user (using the sLCC API's conference creation service) before the sLCC is joined to the CI and before any further API activity takes place. (In the case of the sLRC, the structure is based on the

[1] As mentioned, the design of the sLCC is complete and described in [81], however, of the six basic requirements identified in section 6.2 the attendee and floor remove operations of requirements 2 and 3 are not supported in the current proof-of-concept implementation.

contents of a single XML file, the FOM, which each sLCC is provided.) The conference map is hierarchical. An example conference map is shown in figure 5-2. It shows a user parent conference: C:1 which has two sub-conferences, C:2 and C:3, one of which, C:3, has a single sub-conference, C:4. Although not accessible to users (and therefore shown as a dotted line) the conferencing structure begins with the ROOT conference, C:0, shown in figure 5-2 as encompassing all sLCCs and always present in each sLCC. Although three nodes are shown in figure 5-2 recall that each sLCC is, in fact, part of the overall CI, local representations of the conference map are exactly that, a local representation used by the sLCC to maintain state information. In general, in this document conference maps will depict a single node.

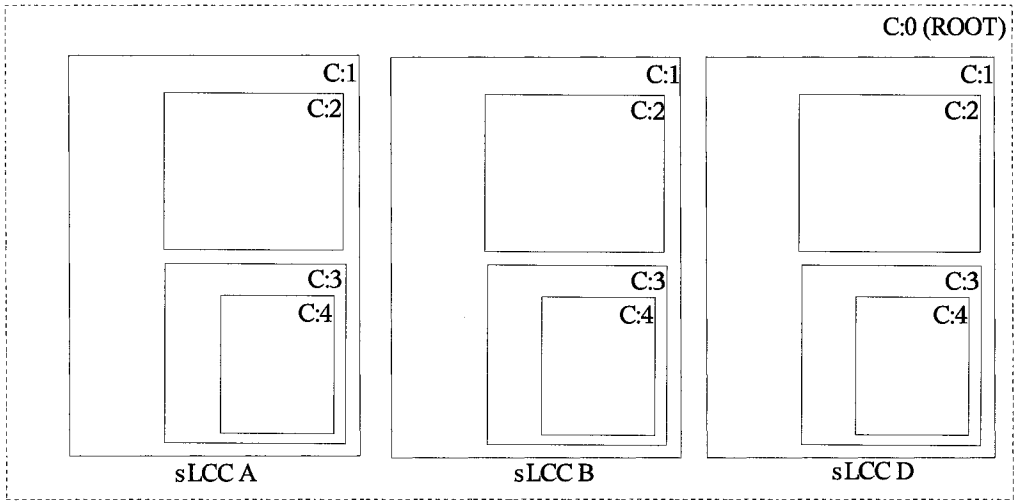

Figure 5-2 A Hypothetical Conference Map

All *joined* sLCCs *participate* in the ROOT conference and, therefore, are listed in each sLCC's ROOT conference participation list. Called a *participation indicator list*, or "PIL", this list is discussed further later. An sLCC may or may not participate in other conferences. Participation in a conference is established when the user makes a call on its sLCC that *places* an attendee or floor in the conference. For example, when a user directs the addition of a speaker attendee an 'S' is placed in the designated conference provided the API's rules were correctly followed by the user. Another example is the placement of a floor. These artefacts are called *items*. The sLCC maintains lists of all items placed in each conference. These lists represent the sLCC's local state. The sLCC also maintains

various *indicator* lists for each conference, such as the PIL. Indicators indicate aspects of remote sLCC's state (i.e. global state) important to the sLCC. Their use is discussed in section 5.5, which describes the maintenance of distributed state in the CI.

An sLCC has *responsibility* for all items it creates and places. While the responsibility for attendees remains with the sLCC that placed them, available floors are unique in that responsibility for them may be transferred to another sLCC through transfer of their ownership. In addition, users do not place open floors; each conference is logically created with an open floor by the sLCC. An open floor's ownership cannot be transferred and it remains the responsibility of the sLCC that placed it.

5.4 Messaging and the use of the SIP to Establish Inter-sLCC Communication

To implement a distributed system some means of communication among components that make up the system is needed. The sLCC uses XML-based messages and relatively simple message passing and queuing techniques to do this. Message passing is used largely because the message set is small enough to keep this approach simple. The following two subsections describe the messages used and how inter-sLCC communication links are established.

5.4.1 Inter-sLCC Messages

Two message types may be passed between sLCCs and are sent as XML streams. *Command* messages are used to pass state information and to negotiate floor ownership transfers. *Announcement* messages are used to send user announcements. On receipt, each message is parsed by the affected conference object which takes the required actions in the case of a command or, if an announcement, sends the announcement to each local user that has a listener in that conference.

All inter-sLCC messages are based on a single XML Data Document Type (DTD), the LCCMsg.dtd. This DTD is presented in Appendix B. It defines all the elements that can be used in both command and announcement messages. The use of XML and a DTD supports flexible message creation; all messages follow the same basic format and are easily created by inserting the elements needed to create an XML string adhering to the

DTD. A special class, the LCCMsgs class (part of the sLCC software, but not shown in figure 5-1), provides methods to the sLCC to create any of the messages required.

```
<?xml version=\"1.0\"?>
<!DOCTYPE message SYSTEM "LCCMsg.dtd">
        <message>
                <conference ConfID = "myConfNum"/>
                <reply TF = "reply"/>
            <commands>
                    <command name = "command1">
                    </command>
                </commands>
            </message>
```

Figure 5-3 A Single Command Message

Command messages may carry one or more commands embedded in them. Figure 5-3 presents an example of a single command message. As is apparent, the first element is the empty element "conference", it has one attribute; the *confID* of the affected conference ("ConfID"). (The sLCC uses confIDs to uniquely identify each conference in the conference map.) Only one is allowed per message. The next element is the empty element "reply". Again, only one reply element is allowed per message, and it has one attribute: "TF". This indicates if the message is an *initial command*, in which case "reply" is set to 'F', or a reply to an initial command, in which case it is set to 'T'. This is used in call synchronization activities. The next element, "commands", has a sub-element, "command", which may be repeated, and has one attribute, "name". This identifies the actual command carried by the message. If a command message carries multiple commands, the name attribute for each command element is parsed out by the sLCC and the appropriate actions taken. There are 4 basic commands. They are named after the indicators, which, as mentioned in the last section, a local machine places in its indicator lists to represent remote state. As is discussed further later, these indicators are actually sent between machines in command messages. An example is the *participation indicator* (PI) which tells remotes a local has just entered a conference for the first time; i.e. it is now participating in that conference. A command built with this command name attribute would be referred to as a *PI command.* There are also various *specialty*, non-indicator, commands implemented in the software. All commands necessary to this

discussion are explained as needed. These, and those not mentioned here, are covered in more detail in [81].

The command element has a further sub-element called "fids". If the command name attribute is "FID", one of the specialty commands, this element is used to pass a list of floor identifiers (floorIDs or FIDs) to remote sLCCs. (A FID is used by an sLCC to uniquely identify a floor.) This is necessary to support floor ownership transfers. Figure 5-4 shows a two command message with a FID list. The fids element's single attribute is "id_type_list". It is comprised of the FID and other information needed by sLCCs when a floor's ownership is transferred.

```
<?xml version=\"1.0\"?>
<!DOCTYPE message SYSTEM \"LCCMsg.dtd\">
        <message>
                <conference confID = "myConfNum"/>
                <reply yn = " reply"/>
                        <commands>
                                <command name = "command1">
                                </command>
                                <command name = "command2 ">
                                </command>
                                <command name = "FID">
                                        <fids id_type_list = "fidList"/>
                                </command>
                        </commands>
        </message>
```

Figure 5-4 A Two Command Message with a FID List

Finally, consider the makeup of an announcement message. An example is shown in figure 5-5. These messages are the same as command messages except that they contain an "announcements" element which may contain multiple "announcement" elements. Each announcement element has a single attribute, "the_announcement". The the_announcement attribute is a string of CDATA[1], which can contain any information the user system wishes. Its use in the sLRC is discussed in the next chapter.

[1] CDATA or "Character Data" is used in XML to escape XML control symbols in user element and attribute data.

```
<?xml version="1.0"?>
<!DOCTYPE message SYSTEM "LCCMsg.dtd">
        <message>
            <conference confID = "myConfNum"/>
            <reply yn = "reply"/>
            <announcements>
              <announcement the_announcement = "announcement">
            </announcements>
        </message>
```

Figure 5-5 An Announcement Message

Note that messages do not contain a "from" element or attribute to identify the sLCC the message is from. This is because this information can be obtained by identifying the MsgIOThread on which the message came in on, as there is a dedicated MsgIOThread for each remote sLCC an sLCC is connected to.

5.4.2 Implementing Inter-sLCC Message Channels

Before any messages can be sent, communication links must be established among all participating sLCCs. This is done using a *queen* sLCC or *qLCC*. The qLCC acts as a normal sLCC but also assists new sLCCs in connecting with other participants. The qLCC must be the first sLCC started in a new CI. All other sLCCs are started thereafter as *subordinate* sLCCs and are given the qLCC's identification on startup[1]. Figure 5-6 shows the process of establishing inter-sLCC connections. On initial startup each sLCC's SIPHandler, using a SIP *INVITE* request, contacts the qLCC via a newly created signalling channel (step 1 in fig 5-6). On receipt of this INVITE, the qLCC adds the new sLCC to its ROOT conference participation indicator list. The qLCC then sends it a SIP "OK" response (step 2) to which the new sLCC sends an "ACK" (step 3). This final message establishes a SIP dialog between the two. The qLCC then contacts the new sLCC's MessageCentre server (step 4) which instantiates a new MsgIOThread and establishes a session channel between the two over which all further communication takes place. The first message sent over the new session channel is from the qLCC to the new sLCC and is the specialty *ConnectList* command (step 5). The ConnectList command provides the new sLCC the identification of all other machines on which sLCCs are already participating in the CI. The new sLCC, using the same process, adds each

[1] This is the one area where the CI design veers from a pure fully distributed or P2P system.

machine identified to its ROOT conference participation indicator list and establishes an application session with each (step 6). In this case, however, when initially connected to subordinate sLCCs do not respond with a ConnectList command as the qLCC does; they merely set up the application session requested once the SIP INVITE-OK-ACK "negotiations" are complete.

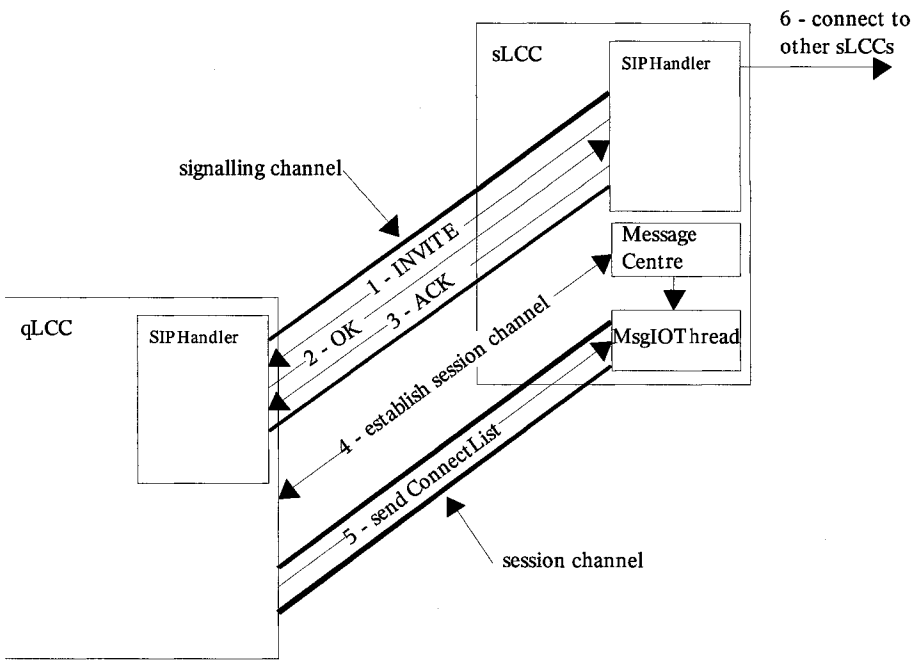

Figure 5-6 Establishing an Application Session Between sLCCs

Note that in either case, no real negotiation regarding session channel characteristics, security procedures, etc., takes place in this use of the SIP; a simple TCP session channel is always assumed and immediately established after the ACK is received. Although the SIP dialogues established over the signalling channel remain in place throughout the session, once the session channel is established, SIP signalling, and therefore the signalling channel, is no longer used. All further communication takes place as application communication over the session channel. However, future extensions of this software can use the dialogue to negotiate QoS, security and media formats, etc., to be used over the session channel, in dismantling the session and, possibly, for other functions such as firewall navigation. Such negotiations would be implemented using

SDP (Session Description Protocol) scripts embedded in the initial INVITE and final ACK messages and in other mid-session SIP signalling as required.

The overall process just described is referred to as *joining* the CI and results in the establishment of a mesh network among all participating sLCCs.

Internal sLCC activity and the inter-sLCC communication that takes place as the result of API call-generated events that affect sLCC state may now be discussed. These events generally result from the addition or removal of an item in a given C:X ("C:X" represents any conference "X" in the conference map). As is discussed in the next section, the state information that must be shared between sLCCs on any of these events depends on the item and what is being done with it (addition or removal) in a given C:X.

5.5 Implementation of Data Distribution in the sLCC – the Maintenance of State

Most information sharing takes place when an sLCC first enters, leaves or places its first of any item type in a conference. The basic approach is to send, or place locally, indicators[1], which inform an sLCC of the others' states. This allows the local to provide remotes with the appropriate information when an API call-generated event occurs on it. There are four indicator types:

PI - participation indicator – identifies an sLCC that is participating in C:X
LI - listener indicator – used by a local that has speakers to identify a remote sLCC that has at least one listener in C:X
SI_L- speaker indicator, listener – used by a local that has listeners to identify a remote sLCC that has at least one speaker, and
SI_F- speaker indicator, floor – used by a local that has floors to identify a remote sLCC that has at least one speaker.

The use of indicators allows minimization of the information that must be sent because an indicator can represent a multitude of same-type items. For example, only one LI is placed in a local with a speaker (to represent a remote that must receive an announcement made by one of its speakers in the affected conference) even if the remote has multiple

[1] When an indicator is *placed* in C:X in fact the CI-wide unique sLCCID (sLCC identifier) for which the indicator is placed is added to the appropriate indicator list in C:X; the term is the same as when applied to attendees and floors.

listeners. In this example, the "speaking" sLCC need only know the remote has at least one listener and need not maintain data on all its listeners.

The following paragraphs discuss how indictor lists are populated. Recall that the indicator acronyms ('PI', etc.) are also used in command messages. These messages indicate activity that has taken place in the sender and that may require the remote to place an appropriate indicator in the remote's global state representation. Depending on its own state, the remote may or may not carry out the placement. This is not always known by the sender prior to sending the message as will be seen as the discussion progresses

For each conference C:X the first event that must be handled is the placement of the first item. This first entry into a conference is an important step in establishing state data in both a local and any affected remotes. Not only does it signify the local's entry into the conference to other participating sLCCs, but it allows the local to establish the lists that it requires to identify remotes that must be sent state information resulting from its own future activities. All further activity in the conference, and any resulting state changes, use and build on these lists.

When an sLCC first enters a conference on behalf of one of its users (the result of an addAttendee() or addFloor() call), it must advise all remote sLCCs that are participating in the conference. However, as shall become apparent, until it enters a conference an sLCC does not know which remotes are participating in that conference, and, therefore, which remotes it must advise. To handle this situation the *initial participation* message that results from first entry into a conference is sent to all participants in the conference's parent's PIL. A rule in the use of the CI (rule 12 - see Appendix C for a listing of all rules) is that, before an sLCC may enter a sub-conference, it must enter the sub-conference's parent conference. Because of this rule, the sLCC will be aware of all remotes participating in the parent conference. Therefore, a parent conference PIL will contain a superset of all remotes participating in any of its sub-conferences.[1]

[1] Recall the ROOT conference's PIL identifies all nodes participating in the CI and is created when each node's sLCC joins the CI.

In describing the activity that takes place when a conference is first entered, a simple example is used to assist in the explanation. It is based on the conference map shown in figure 5-2. In the example sLCC A enters C:2 by placing a listener in it. It is assumed that sLCC B already has a speaker in C:2 while sLCC C has a listener. A sequence diagram (figure 5-7) is used to support the example and shows inter-sLRC messages and activity that takes place on each affected sLRC.

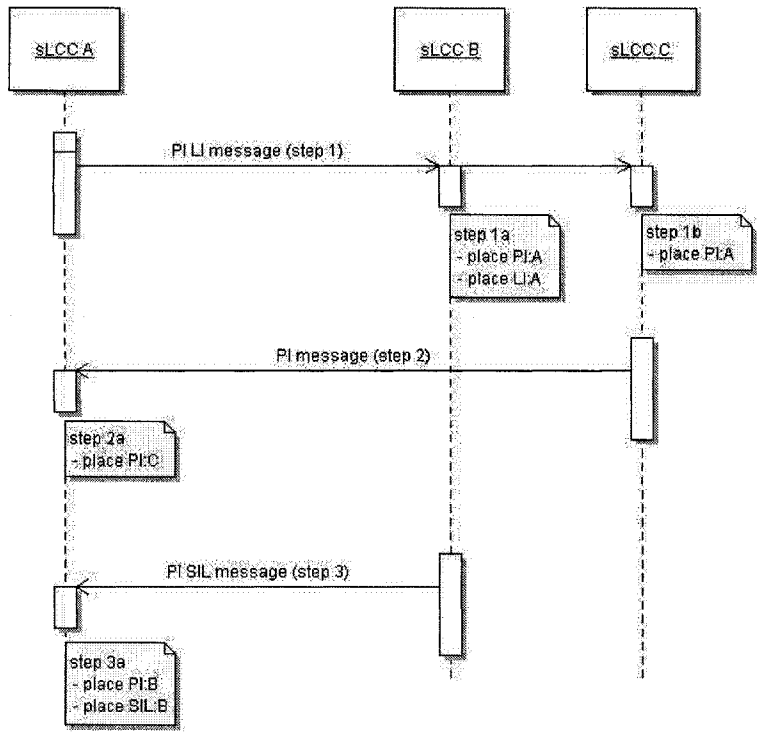

Figure 5-7 Sequence Diagram for Listener Placement Example

In general, on first entering a conference a two-command message is sent. The first command is a PI. This identifies the message as an initial participation message and indicates that the sLCC is now participating in C:X. As a result, remotes already participating in this conference will place the local in their PIL for this conference. The second command identifies the type of item that was placed. As a result of this command, remote sLCC's that are participating in C:X may change their state. In the example, because it is placing a listener, sLCC A sends an LI as the second command (step 1). Because sLCC B has a speaker in C:2 it places both a PI and a listener indicator

for sLCC A in its appropriate C:2 lists (step 1a). sLCC C simply places sLCC A in its PIL (step 1b).

In response to an initial participation message, remotes participating in C:X will reply with an *initial participation reply* message indicating they too are participating in the affected conference. This is necessary, as the local (sLCC A) will not be aware that they are participating since it will have ignored their initial participation messages sent earlier. (This is because non-participating sLCCs ignore initial participation messages they receive as these messages can have no effect on their state.) The remote's initial participation reply messages establish the local's PIL for C:X therefore the parent's PIL no longer needs to be used. In the initial participation reply message the remote will also send any state change information the local needs. In the example, because both sLCC B and C are participating in C:2, both send an initial participation reply message to sLCC A (steps 2 and 3). As a result, sLCC A places a PI for both (steps 2a and 3a). Further, because it has a speaker in C:2 sLCC B includes an SI_L in its response. This tells sLCC A to place an SI_L in its SI_L list for C:2 (step 3a). The SI_L will "remind" sLCC A to inform sLCC B when it (sLCC A) removes its last listener and, therefore, no longer needs to receive user announcements made in C:2.

There are many exchanges of this type possible. They are not all discussed in this document. The reader is referred to [81] for further details if desired.

The next event requiring information sharing occurs when the first instance of any item type is placed in C:X. (In this scenario the local and remotes participating in C:X already know of one another as a result of earlier initial participation message exchanges.) For example, when an sLCC places its first listener, all sLCCs that have speakers must be made aware of this and change their local state to indicate that they must now send this sLCC any user announcements made by their speakers. In these cases, the sLCC in which the initial event occurs uses its C:X PIL to identify the remotes to which to send this change. The remote sLCCs, on receiving such a message, will typically need to make some internal state change (in this example they will place an LI identifying the sLCC

that placed its first listener). Depending on the message sent and their internal state remotes may or may not reply with a response. Only in cases where a remote's state is affected is a response necessary (although, as discussed in section 5.6, an ACK may be needed in situations where call synchronization is necessary).

The placement of any more items of any type after placement of the first instance typically results in no transfer of information, except in the case of available floors. As discussed, available floors may be acquired by other speakers. Because of this, remote sLCCs that have speakers in the conference in which new available floors are placed must be advised of these additions. This makes the remote sLCCs aware of the floors' existence, their identity, and which sLCC is responsible for them. Thus, in this case, the sLCC in which a new available floor is placed, communicates this fact via an sLCC message sent to each sLCC with a speaker in the affected conference.

Changes in a floor's status (its type - uA or oA - and which sLCC is responsible for it) must also be communicated to all sLCCs with speakers in the affected conference. Again, this has to do with their ability to be acquired. This is explained further in the next section.

The removal of items may also result in the sharing of information. As these capabilities are not present in the current implementation they are not discussed here and the reader is, again, referred to [81] for details if desired.

5.5.1 Transferring Floor Ownership and Impact on sLCC State

Floor ownership transfers are negotiated, on behalf of users, by their sLCCs. Ownership is a local concept while responsibility is a global concept. Remote sLCCs are aware only of the sLCC that is responsible for a floor, and are not aware of the actual user that owns it. This allows the requestor's sLCC to identify the responsible sLCC. Only the responsible sLCC is aware of the floor's owner.

The sLCC maintains, in each conference, a FIFO queue of pending requests for floors in the conference for which it is responsible. When an ownership transfer request is received, it is added to the queue. If the floor is unowned, the request is immediately de-

queued and actioned; otherwise, on receipt of an ownership release, the sLCC finds the first request for the released floor and handles it as described below. A monitor protects all floor ownership changes. This is required as multiple, independent requests for a floor may be received by an sLCC, both from its own users and from remote sLCCs.

There may be more that one request for a floor in the queue. Each must be answered or remain pending on the responsible sLCC. If the result of handling the first request is that responsibility for the floor is transferred to a remote sLCC, the sLCC removes all outstanding requests for the floor and sends them to the newly responsible sLCC. The new sLCC will add each pending call to its floor request queue.

There are three floor ownership transfer situations to consider, and each is described in the following sub-sections.

5.5.1.1 Negotiating and Carrying out an oA to oA Floor Ownership Transfer

In this situation, a user wishes to acquire ownership of an available floor that is owned by another. Before an ownership change can be carried out by the sLCC or sLCCs[1] involved, the current owner must be contacted and divest ownership of the floor. The sLCC that supports the prospective owner contacts the responsible sLCC, which in turn, contacts the owning user requesting the floor's release. The owner may reply with a YES, NO or not reply at all. If the reply is YES or NO, the request is removed from the floor queue and the requesting sLCC is notified. If a YES, this sLCC (the one supporting the new owner) notifies the new owning user. The divesting sLCC notifies all interested sLCCs of the affected floor's change in status with a specialty *TX* (transfer) command message. Sending this message from the divesting sLCC helps reduce "not-mines" (see the next subsection). If the answer is NO, no further activity is carried out. If no reply is made, the request remains pending in the floor request queue. Once requested, a user is committed to accepting ownership of the requested floor; cancellations are not supported.

5.5.1.2 Carrying out a uA to oA Floor Ownership Transfer

Changing an available floor from unowned to owned is similar to an oA-oA change. While there is no current owner to query, the responsible sLCC is still asked for

[1] The current and prospective owners may both be supported by the same sLCC.

permission to carry out the change. This is done in order to eliminate races through the implementation of a first-come-first-served mechanism that ensures two (or more) sLCCs do not try to seize an unowned floor at the same time.

The first sLCC to request ownership of the unowned available floor will be granted it by the responsible sLCC, which will then announce this to all other sLCCs with a TX command message. Despite this announcement, requests from other sLCCs may still be received by the originally responsible sLCC. In this case, it replies with a *NOT_MINE* specialty command message. The requestor, on receipt of this message, must recheck its database, hopefully finding the TXed update now, and then re-send its request to the new owner.

5.5.1.3 Carrying out an oA to uA Floor Ownership Transfer

Changing a floor's status from owned to unowned is simple and may be handled internally by the responsible sLCC, because it will remain responsible for the floor. Again, all interested sLCCs are informed of the floor's status change with a TX message.

This completes discussion on the maintenance of sLCC local and global state. The final section describes the concurrent threads that incite activity in any given sLCC and the use of synchronization for some user calls.

5.6 Call Synchronization and Concurrency in the sLCC

At any point in time there may be multiple threads of control within an sLCC. The user software thread enters the sLCC with each call of the API. These calls are returned immediately; either by carrying out the function requested or, if that is not possible, placing the call in the toDoList queue (see figure 5-1) to be run later. Multiple remote sLCCs may also call the sLCC at any time via their dedicated session channels and, while sLCCs act independently and concurrently, deadlock is not entered when two sLCCs send, for example, a PI command at the same time. This is because sLCCs do not queue received commands. Received commands are handled concurrently as received from remote sLCCs. Finally, the internal RunTDQ thread runs user calls placed in the toDoList queue when no remote responses are outstanding as is now discussed.

To use the state maintenance techniques described in this chapter, an sLCC must know the impact, on global state, of certain local state changes it makes before it may proceed with further local state changes. For example, when an sLCC initially enters a conference by placing a unowned available floor it must know of all the sLCCs that are interested in this floor before it may process an ownership transfer request. If it does not, the global state will be corrupted as it will not know which sLCCs to advise of the change. As a result of this type of scenario, some *state affecting* user calls are handled synchronously if they result in the issuance of command messages. This is called *call synchronization.* In the sLCC a state affecting call is any user call that affects the state of the sLCC. This includes the placement of attendees and floors. Such commands are placed in the toDoList queue and run after the sLCC receives responses from other remotes to the preceding state affecting call, if any. The local simply counts all responses, ensuring it receives one for each sLCC that was sent a message related to the current call before running the next one in the toDoList queue[1]. Note again, that the sLCC must only wait if the call resulted in the issuance of an inter-sLCC command message. If it does not, there is no possibility of it affecting global state and therefore no possibility of a resulting global state change affecting the local.

Also, note that not all commands must be placed in the toDoList queue. For instance, user announcements are not acknowledged. Nor does this have major impact on network messaging load, as most messages require some form of response in any case. However, there are cases, notably when a remote is not participating in a given conference, where a simple "ack" is returned to provide the reply required by the sender of a state affecting command.

The sLCC also implements one *state dependent* call. It is the changeOwner() call used in floor ownership transfers. States dependent calls are dependent on the CI's global state to function properly and are, therefore, placed in the toDoList so all state affecting calls are

[1] The RunTDQ thread is normally in a wait mode. When it is signaled, it loads and runs calls in the toDoList queue until signaled to return to wait mode or it determines if this is necessary itself.

completed before they are run. However, once actioned, the sLCC does not require acknowledgment from the remote sLCCs; the local may continue on to the next call.

Chapter 6 The crtiModel and Design of the SIP Local RTI Component

This chapter presents the crtiModel and the design and implementation of an HLA LRC, the sLRC, which uses the sLCC to realize the semantics of the crtiModel. The resulting RTI, the SIP-RTI, provides federates with a partial implementation of the IEEE 1516 Java API ([3], Annex B) and supports the requirements of the demonstration federation introduced in Chapter 4. Appendix D provides a listing of the IEEE 1516 calls implemented by the sLRC.

After presenting the crtiModel, this chapter provides an overview of the sLRC, identifying and discussing the main components used in its implementation. It then describes how these components are used to implement the API calls supported. During these discussions important aspects of sLRC design, such as the naming conventions used and call synchronization, are introduced and discussed as necessary.

6.1 The crtiModel

The initial crtiModel addressed ownership transfer capabilities [10]. A later paper, [8], describes extensions to the crtiModel to support DDM services. The initial version is used in this research.

The crtiModel uses conferences to represent the federation, object classes and attributes. Speaker and listener attendees are used to represent federates and their interests and floors are used to represent object class instances. All conferences used in the crtiModel are created based on information in the FOM, although the first, which represents the federation, is not explicitly defined. It is called the *federation conference*; all federation RTI activity takes place in the federation conference.

Once the federation conference is created, a sub-conference is created within the federation conference for each object class defined by the FOM. These are called *object*

class conferences. The object class conference is the vehicle used to inform federates of new class instances. Federates that publish attributes of a given class send a speaker to the associated object class conference so they can announce new instances as they are registered. Federates that subscribe to an attribute in a given object class attend as listeners so they can discover new instances. Federates that do both attend the class conference with both, a speaker and listener. There is one floor in a class conference and it remains open so new instances can be announced by any federate.

An *attribute conference* is created within each object class conference for each attribute defined for that object class in the FOM. Attribute conferences are used in the definition of object class instances and to distribute updates as is discussed in the next paragraph. Once object class and attribute conferences have been created for all objects and attributes defined in the FOM, the federation's conference map is complete. Figure 6-1 depicts a conference map defined by a FOM which has a single object class 'X' with two attributes 'X.a' and 'X.b'[1]. In the diagram, conferences are represented as named enclosing boxes. Floors are depicted with a short, heavy, horizontal line. An open floor is identified by the word "open" above the line. Note that the inherited HLAprivilegeToDeleteObject attribute has an attribute conference defined as well.

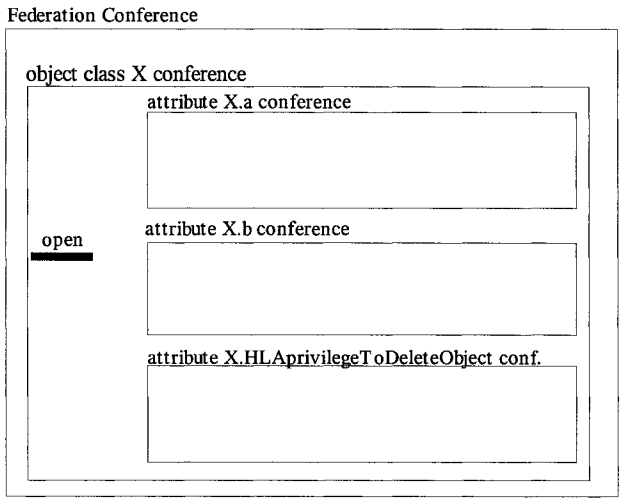

Figure 6-1 An Example Federation Conference Map

[1] For clarity, the CI's ROOT conference is not shown in this map. The ROOT conference would otherwise surround the federation conference in this depiction. Similarly, the HLAObjectRoot conference is not shown.

As mentioned, attribute conferences are used in the definition of object class instances and the update of instance attribute values. If a federate publishes an attribute, it sends a speaker attendee (represented by the name, in capitals, of the party it represents with a subscript to identify its type; "'S' for speaker, 'L' for listener) to the related attribute conference so it can announce updates. If the federate subscribes the attribute, it sends a listener to hear updates. In addition, an available floor is created in each of the object class's attribute conferences. Combined, these floors represent the new instance and are identified by placing the instance name under the floor. Instance floors are owned by the attendee that created the object instance if it had a speaker in the floor's attribute conference when the instance was created (i.e. the federate that registered the instance publishes that attribute). A light line to the attendee indicates this. Otherwise, they are unowned.

Only the federate whose attendee owns an instance attribute floor may update that instance attribute. When a federate updates an instance attribute, the update is announced in the attribute conference via the instance attribute floor. As a result, all subscribing (listening) federates receive the update. Figure 6-2 adds a single instance, "inst 1", to the earlier example. Both instance attributes are published and, therefore, owned by Federate

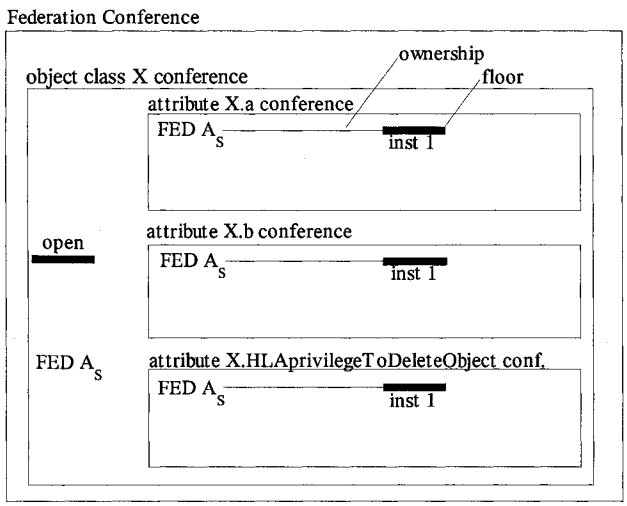

Figure 6-2 Example Federation with One Instance

A. Because Federate A created this instance, it also owns the HLAprivilegeTo-DeleteObject attribute. The instance floors created in the HLAprivilegeToDeleteObject attribute conference are not used for updates, as no values can be associated with this attribute in the HLA. Instead ownership of an instance floor in a HLAprivilegeToDelete-Object attribute conference is treated as a privilege token. All federates that register an instance of a class send a speaker to that class's HLAprivilegeToDeleteObject attribute conference.

The deletion of an object has repercussions throughout the associated class conference. Only the federate whose attendee owns the HLAprivilegeToDeleteObject floor for that object instance may delete the object. The deletion of the object is accomplished by deleting the associated HLAprivilegeToDeleteObject attribute floor. This, in turn, causes each of that object's related instance attribute floors to be deleted. To ensure all federates are informed, the deleting federate's object class conference speaker announces the deletion of the object class instance from the open class conference floor. For example, if the lone instance in figure 6-2 is deleted, the conference map would appear as shown in figure 6-3. Note the 'FED A_S' attendee has been removed from the HLAprivilegeTo-DeleteObject attribute conference. That attendee is no longer required because the instance no longer exists.

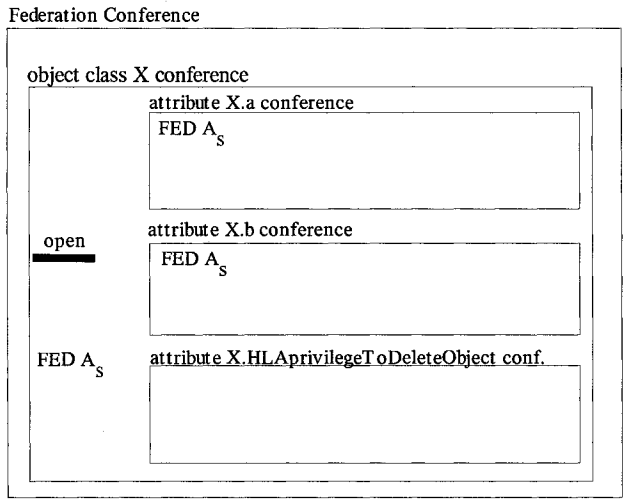

Figure 6-3 Example Federation after the Deletion of "inst 1"

To further clarify the crtiModel, figure 6-4 provides a fuller version of the previous example, adding two more federates and the RTI along with all federate's publish and subscribe status. The federation execution consists of three federates; A, B and C. Again, a single object class X with two attributes is defined: X.a and X.b. Federate A publishes attributes of object class X and, therefore, sends a speaker (FED A_S) to the object class X conference. This will allow it to announce new instances of X. Federates B and C subscribe this object class and, therefore, have listeners at the conference to hear new instance registrations. Federate A has registered one instance of class X. As a result, a floor (inst 1) has been created in each attribute conference. Federate A's attendees (FED A_S) own instance 1's floors in X.a and X.b because Federate A publishes both of these attributes. They are speakers so updates can be sent. Federate A also owns the HLAprivilegeToDeleteObject attribute for inst 1 because, when registered, Federate A was publishing at least one attribute of this object class. Finally, Federate B and C have listeners at attribute conference X.a as they both subscribe this attribute and Federate C has a listener at attribute conference X.b as it further subscribes this attribute.

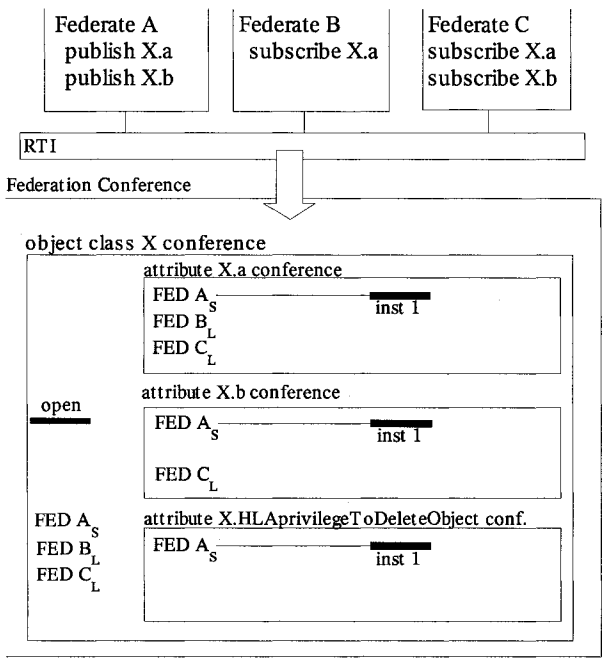

Figure 6-4 A More Complex Federation Conference Example

6.1.1 Object Attribute Ownership

In the crtiModel, ownership of an object's instance attribute floor corresponds to ownership of the instance attribute in the HLA. If a federate would like to acquire ownership of an attribute (the "pull" model of ownership acquisition) it must negotiate with the current owner. This is accomplished by sending a Request_Ownership message to the conference moderator. If the floor is unowned, ownership is immediately granted. If the floor is owned, the message is relayed to the owner and it is up to the owner to grant the request or not. If the owner does grant the request it advises the moderator which, in turn, advises the original requestor that it now owns the floor.

If a federate would like to divest ownership of an instance attribute, it notifies the conference moderator of this desire, by sending it a Divest_Ownership. The moderator then marks the floor as unowned.

If a federate resigns from a federation then all instance attribute floors held by the federate's attendees become unowned. If the resigning federate owned permission to delete objects, then the way in which the resign is requested will determine whether the objects are deleted (according the HLA specification). When the federation resigns, all of the federation's attendees leave the conference.

6.2 An Overview of the sLRC

The sLRC implements the semantics embodied in the crtiModel using the services of the sLCC, thus implementing the semantics of the HLA. The sLRC acts as a user surrogate; it is an intermediary between the federates it supports and the sLCC. It uses sLCC attendees to represent the federates. Because each sLRC can support multiple federates, a means to uniquely identify them in calls to the sLCC is required. This is provided by a sLCC generated CallerID object. The sLRC obtains a CI-wide unique callerID instance from the sLCC for each new federate that joins the federate execution and associates this ID with the federate.

Figure 6-5 provides an overview of the sLRC's design. The dotted line represents the Java package that contains all sLRC classes. The main component is the sLRC object[1], which is a singleton class. It implements the functionality required by the selected IEEE 1516 services, however, federates do not call it directly. Rather, they invoke RTI calls, and thus sLRC object activity, through HLA-specified RTI ambassador objects called *RTIAmbs* that present the IEEE 1516 API to the federates. An RTIAmb then calls the appropriate method on the sLRC object. Before a federate can access any of the HLA services it must obtain an RTIAmb with a call to getRTIAmbassador(). This is the sole public method exposed by the sLRC object. The first such call also results in instantiation of the singleton sLRC object.

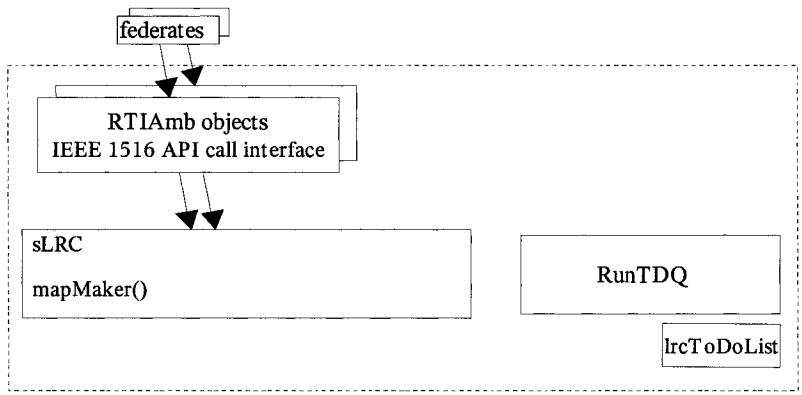

Figure 6-5 A High Level View of the sLRC

The RTIAmb also adds additional information to the calls it makes on the sLRC object. The added information allows the sLRC to determine which federate made the original call and find its callback interface when a callback is necessary. This is required because the API calls do not contain a federate identifier argument similar to the callerID used in calls on the sLCC.

As with the sLCC some sLRC API calls must be synchronized. If a call can be handled immediately by the sLRC object it is, otherwise it is placed in the lrcToDoList queue and executed by the sLRC's RunTDQ thread object when it can be run. In the case of the

[1] In the following discussion, if a distinction must be made between the sLRC as a whole and the sLRC object the term "sLRC object" will be used, other wise the term "sLRC" refers to the sLRC as a whole.

sLRC the reason for doing this is because some HLA calls have impact on the state of the sLCC and, therefore, the state of the SIP-RTI. For example, the creation of a new HLA object class instance results in the creation of new instance attributes. These must be placed in the CI as floors before the attributes they represent in the RTI's domain can be updated or their ownership transferred. Such calls (HLA Update Attributes and some HLA ownership transfer calls) are referred to, again, as state dependent calls. In the sLRC a state dependent call is any call made on the sLRC that is dependent on the state of the RTI (and therefore the state of the CI) to correctly perform its functions. Recall from the discussion on sLCC synchronization that certain calls made on the sLCC are state affecting; they affect the state of the sLCC and, possibly, the global state of the CI. The sLRC lrcToDoList/RunTDQ mechanism is set up so that no sLRC state dependent calls are run if previous sLCC state affecting calls have not been completed by the sLCC. This ensures all state necessary to run an sLRC state dependent call has been received by the sLCC. To allow the sLRC to make this determination the sLCC signals the sLRC via a callback when each state affecting call the sLRC makes on the sLCC is completed. The sLRC keeps a list of all state affecting calls it has made on the sLCC, removing each as it completes and does not run state dependent calls unless the list is empty. Note non-state dependent calls can be run at any time. Reference [83] identifies sLRC state and non-state dependent calls.

This raises another issue; the sLRC maintains no dynamic state information whatsoever. If the SIP-RTI (or any other user of the sLCC) needs information about its own state it must either save such data, redundantly, itself or request it from the sLCC. For example, if the sLRC needs to know if a federate is publishing a given attribute (i.e. if a given user it is a surrogate for has a speaker in the associated conference) the sLRC must query the sLCC. The sLCC's API provides services to obtain state information users may need, they are listed at the end of the sLCC API listing in Appendix A.

Another important aspect of the use of the sLCC is that one of its users must be responsible for building the sLCC's conference map. In the case of the sLRC this is done by the sLRC, the only direct user of the sLCC. This is done when the sLCC's LCCfe

object is instantiated when the HLA API Create Federation Execution service is called. The sLRC uses the FOM passed in to this call in its internal mapmaker() method to create the conference map, parsing out the object class and attribute names and creating the required conferences in the sLCC. All sLCCs must contain the same conference map and thus have the same FOM file available locally. Section 7.4.1 describes the SIP-RTI conference map further.

The design of the sLRC object is described in the following sections and sub-sections by explaining how the operations just discussed and those of the other HLA services supported are implemented. Additional design issues of importance are presented where necessary. Before this is done, however, a brief discussion of the name translations that the sLRC must carry out is necessary.

6.3 Names, Handles and IDs – the Naming Conventions Used in the sLRC

The sLRC must be able to handle and translate between three types of designators; those used by the federates and defined in the FOM, handles which it has created (as required by the HLA specification), and those used by the CI. Depending on the call, a federate may use FOM names or handles in calls to the sLRC. The sLRC must translate these designators to the equivalent sLCC conference and floor identifiers (confID and floorID) in any calls it makes to the sLCC. Similarly, federate handles must be translated to the associated callerIDs the sLCC uses to uniquely identify each user. Each sLRC maintains the three designator conversion lists shown in table 6-1 to assist in these conversions.

FOM Name	HLA/RTI Handle	LCC Identifier
ObjectClassName	ObjectClassHandle/ AttributeHandle	ConfID
not applicable	ObjectInstanceHandle	FloorIDList
not applicable	FederateHandle	CallerID

Table 6-1 Designator Conversion Lists Maintained by the LRC

The population of these lists is discussed in the relevant sections below.

6.4 Implementing HLA Services in the sLRC

Implementation of the sLRC API services is now described. This section is divided into the HLA service groupings, services the SIP-RTI implements are described in the appropriate section. The descriptions are presented in terms of the designator conversions lists just mentioned and are kept relatively high level. Detailed implementation is described in [83]. The manner in which the sLCC is used to implement these services is a direct result of the design of the crtiModel.

Note that, while not mentioned further in the sections below, before proceeding with a call the sLRC ensures the user is a valid user (has a valid callerID) and is obeying the IEEE 1516 HLA rules. For example, only federates that publish at least one attribute of an object class may register a new instance of this class. If these rules are not followed the appropriate exception is thrown and the call aborted.

6.4.1 Federation Management

The SIP-RTI supports HLA federation creation and joining services. The following paragraphs explain their implementation.

In typical use all federates call the Create Federation Execution service prior to attempting to join the federation. On the initial invocation of this call the sLRC object parses the FOM file passed in and creates the sLCC conference map, at the same time populating the ObjectClassName-ObjectClassHandle/AttributeHandle-ConfID list.

The SIP-RTI creates an essentially flat conference map. The ROOT conference is the parent of all conferences. Each object class defined in the FOM is implemented as a first level user sub-conference of the ROOT (the ROOT conference is not a user accessible conference) whether it is a sub-class or not. One second level user sub-conference is created in an object class conference for each of its attributes including inherited attributes. Appendixes E and F discuss this further and provide an example FOM, its representative object class hierarchy diagram and the resulting SIP-RTI conference map.

Once the conference map is completed the sLCC is instructed to connect to the qLCC and thus any other sLCCs already participating in the CI. In effect, each sLRC acts as the RTI FedExec, using the connection services of the qLCC (which it accesses through its sLCC) to connect to the other sLRCs. There is no separate RTI-level FedExec process.

Once the federation execution has been created, a federate may join it. The sLRC object does little more than indicate internally that the new federate is joined, obtain a callerID for it from the sLCC, issue the federate a federate handle and add all relevant information to its FederateHandle-CallerID list. The sLRC object also adds some of this information to the federate's RTIAmb to assist it with identifying the federate.

6.4.2 Declaration Management

Declaration management services supporting attribute publication and subscription are provided by the SIP-RTI. The sLRC, on receiving a request to publish a set of object class attributes, adds one "speaker" type attendee to each attribute conference affected. It uses the ObjectClassName-ObjectClassHandle/AttributeHandle-ConfID list to find the sLCC's confID of the conferences representing each attribute. It then requests the sLCC add a speaker for the calling federate to each. A speaker is also added to the object class conference and is used to announce new object class instances registered by the federate.

Attribute subscription is handled exactly as publishing is, except that listeners are placed in the subject conferences. Because it is not necessary to the demonstration of attribute ownership transfer, HLA object class promotion is not implemented, however, [83] describes a straightforward modification of the sLRC that will provide this capability.

6.4.3 Object Management

Services to register and update object instances are provided by the SIP-RTI. Both rely on the sLCC announcement mechanism. Once this mechanism is described each service's implementation is presented.

The CI's announcement concept supports communication between its users. The SIP-RTI implements an sLRC layer (figure 5-2) messaging system using sLCC announcements. All sLRC generated announcements are constructed with a type identifier and related information. There are two types of inter-sLRC messages: *REGISTERs* and *UPDATEs*. Each is constructed as a string containing its type and information relevant to its type. When an sLCC receives a remotely generated announcement it adds further information to it before passing it up to its sLRC; it adds the callerID of the local users (federates) with listeners in the affected conference. This allows the sLRC to determine which of its federates must receive the update or registration contained in the message. The sLCC then makes an announcement callback on the sLRC. In handling the announcement callback the sLRC first parses off the callerIDs identifying the associated federates. It then determines the type of announcement it has received and parses the relevant information. The sLRC then takes action relevant to the announcement type as described in the following paragraphs.

To register a new object instance the sLRC must create a new floor in each of the attribute conferences associated with the object class being registered. The ObjectClassName-ObjectClassHandle/AttributeHandle-ConfID list is used to determine the associated sLCC conferences and the sLCC is directed to add a floor to each. If the calling federate is publishing a given attribute (which is determined by querying the sLCC if the federate has a speaker in the attribute conference) it is made owner of the new floor. All other floors are created as unowned available floors.

The sLRC generates and returns an RTI-wide unique Object Instance Handle to federates registering a new object class instance. Each of the floorIDs representing the new instance and returned by the sLCC is stored along with the Object Instance Handle in the ObjectInstanceHandle-FloorID list.

Once the new floors are created, the sLRC uses its sLCC to send an announcement through the federate's speaker in the object class conference. The announcement contains information that identifies the new instance. All sLCCs that have subscribers (listeners)

in this conference receive the announcement and pass it up to their sLRCs. The announcement contains the floorIDs, which the remote sLRCs strip off and use to update their ObjectInstanceHandle-FloorID list. They also call each subscribing federate's "discover" callback on its FederateAmbassador using the information provided by the sLCC to identify the affected federates as was described above.

The sLRC carries out an HLA attribute update by making an UPDATE announcement in the affected attribute conferences. The affected conferences are determined from the ObjectClassName-ObjectClassHandle/AttributeHandle-ConfID list, which is searched based on the attribute handle(s) passed to the sLRC by the calling federate. The UPDATE announcement is received by all sLCCs supporting sLRCs that have federates subscribing the affected attribute conferences. Once the announcement is parsed by the receiving sLRC, it notifies all subscribers via an update callback on their FederateAmbassador.

6.4.4 Attribute Ownership Management

The SIP-RTI implements the pull methodology and supports the following four RTI calls:

7.2 - Unconditional Attribute Ownership Divestiture
7.7 - Attribute Ownership Acquisition Notification†
7.8 - Attribute Ownership Acquisition
7.11 - Request Attribute Ownership Release†

The HLA semantics of these calls are described in Chapter 2. The calls are asynchronous; the RTI and federates respond to the related calls or callbacks via a separate response. In addition, multiple requests may be made for any given attribute. As a result, there may be multiple outstanding requests for an attribute's ownership at any one time. As discussed in Chapter 5, the sLCC maintains a queue of pending floor requests; this supports these RTI requirements. Most activity related to attribute (floor) divestiture and acquisition occurs in the sLCC and is described in Chapter 5. The sLRC related activities are described in the following paragraphs.

When the Unconditional Attribute Ownership Divestiture service is requested, the sLRC indexes into the ObjectInstanceHandle-FloorID to obtain the floorID of the affected floor (s) and then directs the CI to release ownership of the identified floors.

The sLRC implements the Attribute Ownership Acquisition service by indexing into the ObjectInstanceHandle-FloorID list to find the floorIDs of the affected attributes and then asks the CI for ownership of the identified floors.

The sLCC determines the floor's responsible sLCC and, if it is not itself, sends it an acquisition request message (a specialty sLCC message). This sLCC identifies the callerID of the owner and contacts its sLRC passing it the callerID and floorID. This sLRC translates the callerID to the related federate and the floorID to the related instance attribute. It then contacts the federate with a Request Attribute Ownership Release† on its FederateAmbassador.

The current owner will respond either by calling Unconditional Attribute Ownership Divestiture or by doing nothing if it is unwilling to release it. If an Unconditional Attribute Ownership Divestiture is sent, the responsible sLRC notifies its sLCC that the associated floor is available. This is passed to the original sLCC, with another inter-sLCC specialty message, which results in its sLRC invoking the requestor's Attribute Ownership Acquisition Notification† callback. If the owner does not release the attribute, the request remains in the responsible sLCC's floor request queue until it is.

6.4.5 HLA Support Methods

Most of the HLA calls supported in this implementation require the federate use sLRC-given handles for object classes and attributes rather than FOM names for their arguments. The SIP-RTI generates a set of SIP-RTI-wide unique handles for each data type when the FOM is parsed and provides the HLA support service calls getObjectClassHandle() and getAttributeHandle() to allow federates to obtain them. These calls simply index into the associated designator conversion lists, using the FOM name the federate passes in, and return the associated handles.

Chapter 7 Testing and the Air Traffic Control Demonstration Federation

Both, the sLCC and the sLRC code, were subjected to two levels of testing during development. A demonstration federation, consisting of three federates and simulating the handoff of an aircraft between two air traffic controllers, was employed to display, and further test, the HLA ownership management services implemented. Finally, a simple monitor application, which connected directly to the demonstration federation CI, was used to demonstrate interoperability between different applications. After describing the development and test environment, the following sections describe the testing done and the demonstration federation and monitoring application.

7.1 Development and Test Environment

A three-node test and development network was used (figure 7-1). All implementation was done in JAVA using various releases of the Eclipse 3 workbench [84]. The SIP services were implemented using the JAIN SIP API v1.1 RI, which is a SIP API, based on the NIST SIP V1.2 parser and stack [80]. Development and test computers were IBM compatible desktops, of differing speeds, running Microsoft Windows 2000, XP Professional, and XP Home Edition. A 100BASE-T (Fast Ethernet), LAN was used. Note figure 8-1 shows the final configuration of each node after the sLCC, sLRC and demonstration federation code was developed and installed on each. The Hawaiian Island names are the machine names of each computer used in the network.

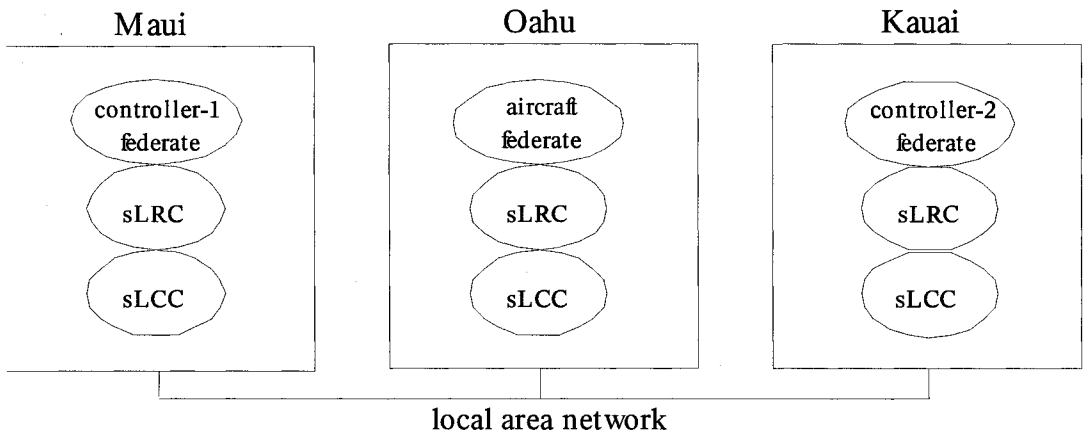

Figure 7-1 Development and Test Bed

To give an idea of the scope of the implementation project and the testing required, the statistics for the sLRC, sLCC and demonstration software are given in table 8-1.

Component	Package name	Lines of code[1]	Number of classes[2]	Number of exception classes[3]
sLCC	slcc	6357	19	26
sLRC	hla.rti	3920	23	29
demo	testFederate	628	1	0

Notes

[1] Lines of commented code. Does not include exception code.

[2] Not including exception classes

[3] Number of exception classes defined in package specifically for use by package software.

Table 7-1 Code Statistics

7.2 sLCC Testing

The sLCC was developed first. Testing was progressive; after the initial infrastructure was developed each API service was added and tested. Conference creation was the first call implemented. It was used to create a 5-conference conference map that was then used in all further testing. Initially two nodes were used in development. Once all services were completed, a two-stage test set was run. The first stage tested all non-ownership transfer related services. The second stage added ownership transfer calls and tested all variations of their use. Both nodes ran the same call sets. Once two node testing was complete the same set of tests were run on the full, three node test bed. This revealed some "concurrent access of shared data structures" errors, which were corrected.

Variations in join times were also tested. For example, all nodes were started together and at staggered intervals. This tested the distributed system's ability to support "late joiners".

7.3 sLRC Testing

Testing of the sLRC was similar to that of the sLCC. However, a much more complete conference map was used based on a FOM that had four object classes and, cumulatively, nine attributes. With the attribute inheritance involved, this resulted in the creation of a

23-conference conference map. Dummy federates which did not implement an actual simulation were used in this testing. Testing was, again, carried out incrementally during call development and final testing consisted of two stages; all non-ownership management calls were tested and verified first, after which ownership management calls were added. During sLRC testing sLCC state was monitored in addition to sLRC activity to verify correct behaviour in this more complex scenario. All tests were run between two nodes, then with multiple federates (first two, then three) on a single node and then using three separate nodes. (In addition, the demonstration federation was run across three nodes and with all federates on a single node.)

The final test was the development and running of the demonstration test program which is now described.

7.4 The Demo Federation

As the test case for this proof-of-concept implementation is HLA ownership transfer, a demonstration of the SIP-RTI's ability to support this capability was in order. Simulation of an air traffic controller scenario provided a suitable real-world situation. In this scenario an aircraft, commencing a flight in one air traffic controller's airspace, is handed off to a second air traffic controller responsible for a neighbouring airspace when the aircraft enters that airspace.

The scenario is modeled using a 3-federate federation. Each federate is run on a separate node on the network and, therefore, on separate sLCCs. The first federate, the "aircraft" federate, models an aircraft flying between two points while the other two federates, "controller-1" and "controller-2", model the air traffic controllers. The aircraft starts its flight in controller-1's airspace and flies a straight line into controller-2's airspace. The flight is 10 distance units long and divided into 11 positions, 0 to 10. Controller-1's "official" airspace extends over positions 0 to 5 while controller-2's is from 5 to 10. There is a two unit buffer zone from position 4 to 6 in which either air traffic controller may control the aircraft. The aircraft starts its flight at position 0 (controller-1's airspace).

There is only one object class in the simulation, the AIRCRAFT object. The FOM used in this demonstration is contained in Appendix E. It can be represented using the HLA object class hierarchy diagram shown in figure 7-2. The resulting CI conference map is provided in Appendix F.

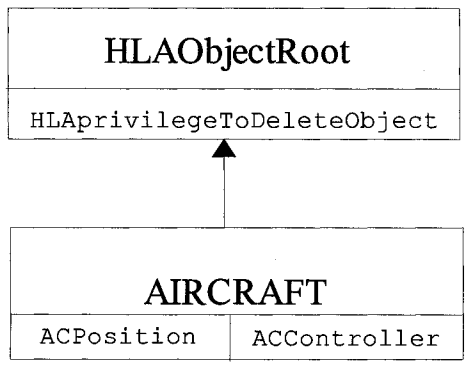

Figure 7-2 HLA Object Class Hierarchy Diagram of the Air Traffic Controller Federation

The AIRCRAFT object has two attributes (in addition to the inherited HLAprivilegeTo-DeleteObject attribute); "ACPosition" represents the aircraft's position, "ACController" represents the controller currently in control of the aircraft. Both attributes are implemented as Java Integer class objects by the federates.

The aircraft federation controls the progress of the aircraft's flight. It publishes ACPosition and then instantiates the single aircraft object instance. Because the aircraft federate does not publish the ACController attribute, when the AIRCRAFT object is instantiated this attribute is unowned. After a suitable delay to allow controller federates to join the federation (HLA synchronization points would typically be used for this, however, they are not implemented in the SIP-RTI), the aircraft federate begins incrementing ACPosition.

Both air traffic controller federates subscribe to ACPosition and ACController. They also publish ACController. When the AIRCRAFT object is instantiated controller-1 immediately requests ownership of the ACController attribute. As mentioned, this

attribute is created as unowned, however, it is the responsibility of the sLCC on the node on which the aircraft federate is running (Oahu in figure 7-1). Therefore, the sLCC on Maui, where controller-1 is running, must ask Oahu's sLCC for ownership of the unowned attribute. This is immediately granted and transfer completed in accordance with the procedures described in Chapters 5 and 6.

As the flight progresses the aircraft federate updates the ACPosition attribute. Both controllers receive these updates via update callbacks from their sLRCs. When this attribute reaches a value of 4 (i.e. the aircraft reaches position 4 in its flight plan) controller-2, running on Kauai, requests ownership of the ACController attribute. Controller-1 is programmed to release this attribute, if requested, if the aircraft is 4 or more units away. Therefore, on receipt of the request, it releases the attribute and the SIP-RTI transfers ownership to controller-2 which holds it until the termination of the flight and then releases it.

The demo was successfully run with all federates joined before the simulation began running and with late joining controllers (the aircraft federate cannot join "late" as it drives the simulation). In the latter cases, if the aircraft is in range of the late joining controller its ACController attribute will be acquired, if not, the controller simply echoes the position update info.

The scenario demonstrates all aspects of ownership transfer and all possible types of transfer including transfer of an owned floor between federates (oA-oA), transfer of an unowned floor to an owned status (uA-oA) and transfer of an owned floor to unowned status (oA-uA) between different sLRCs. It is considered a complete test of the sLRC and sLCC functionality provided and, as well, a successful proof of the application of a conferencing paradigm to distributed communication, the use of the SIP in establishing the required conferencing architecture and of a conference-based approach to implementing the HLA.

7.4.1 Demonstrating Interoperability between Differing Applications

A monitoring application, called "MonitorApp", was developed as a simple non-RTI application to demonstrate the ability of the CI to support interoperability between different types of applications. It demonstrates this by communicating through a common CI with the SIP-RTI that supports the air traffic controller demonstration. Figure 7-3 shows the configuration of the two demonstration applications. The SIP-RTI is implemented by the RTI/CI combination and supports the three federates implementing the air traffic controller simulation. The federates are represented by the ovals labeled "F". The monitor program is represented by the square labeled "Mon". A square is used to highlight the fact that it is a different application than the RTI. Note that the monitor and the RTI middleware connect directly to the CI. The monitor program attends conference 5 in the CI. This conference implements the ACPosition attribute conference. As a result, the monitor application receives all position updates in the form of an inter-sLRC Update message.

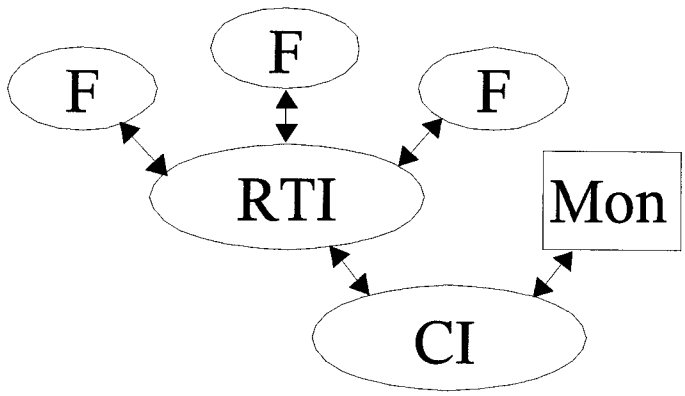

Figure 7-3 Demonstrating Interoperability

The monitor and RTI middleware communicate through the CI; it is the CI, and the application developer's knowledge of the conferencing model that it uses, that allows communication to occur between these dissimilar applications. Thus, within the definition of interoperability used in this work (i.e. communication) this simple test demonstrates the ability of the CI to provide interoperability, between heterogeneous applications designed to make use of the conferencing paradigm and session layer implemented by the CI. This verifies the viability of this approach as the basis of a

general purpose interoperability solution. More detailed tests, consisting of other CI-based applications and RTI's ported to use the CI would provide interesting future work and further establishment of the CI's ability to support interoperability.

Chapter 8 Conclusions and Suggestions for Future Work

The fundamental thesis of this work is that a SIP-enabled RTI, using an application conference for communication, is a valuable step towards solving not only the RTI interoperability issue but general interoperability problems as well. In researching this thesis a SIP-enabled, distributed communication system was developed based on a conferencing paradigm. This system creates a Conferencing Infrastructure upon which custom HLA RTI LRCs are used to implement the behaviour of a conference-based model of the HLA semantics resulting in the SIP-RTI. The CI effectively extracts the HLA communication requirements into a separate, general-purpose communication component that implements the requirements of the ISO/OSI session layer in this application. This component eventually might lead to the development of a general interoperability solution for distributed systems. In addition, the use of the Session Initiation Protocol in implementing this communication system provides it with the potential to solve other RTI and general distributed application issues. Examples include firewall traversal, provision of general security features, system discovery and the negotiation and provision of various network services, such as QoS, over the communication channels established between applications using this system.

This chapter presents the conclusion of this work, identifies the contributions it makes to existing research in the areas of HLA and distributed systems communication research and provides recommendations for further related work.

8.1 Conclusion

The immediate goal of this research is met. A conferencing-based RTI was developed using communication techniques that might, eventually, lead to the solution of the RTI interoperability issue. In addition, its ability to support general interoperability, the larger vision of this work, has been demonstrated.

8.2 Contributions

A multitude of contributions, in various areas of research, are made by this research. They are enumerated in point form below for easy reference.

8.2.1 The work takes a step towards the development of a general interoperability solution for distributed systems by providing a common conference-based communication model and implementation. The CI is a unique package in that it supports application conferencing versus human-human conferencing, as is the target of most conferencing applications today. It is further unique in that it is one of the few fully distributed conferencing systems available. Finally, it allows participants to seamlessly join or leave a session "on-the-fly" while maintaining their state and that of the whole system.

8.2.2 The reintroduction of the OSI session layer. The CI is implemented as a separate application at layer 5 in the ISO/OSI communication reference model. Use of the CI forces applications to follow a common communication paradigm and thus further supporting interoperability.

8.2.3 A first step in solving the HLA RTI interoperability issue is realized by applying the conferencing paradigm to HLA semantics. The resulting crtiModel demonstrates the ability of the approach to support HLA semantics by incorporating HLA ownership transfer capabilities. In addition, the crtiModel provides a new way to look at HLA semantics – one, that possibly, will aid in quicker understanding of the HLA and is completely unique.

8.2.4 The SIP-RTI will be made available as open source and, if this work is continued, may ultimately lead to the first, fully compliant open source Java implementation of the IEEE 1516 specification. Because the CI is SIP-enabled and implemented as a separate component it can be used to provide applications with a multitude of other capabilities including firewall traversal, security, system discovery and communication channel services. This approach has not been considered in earlier attempts to add these abilities to the HLA.

8.2.5 The demonstration program demonstrates not only the SIP-RTI (and, as a result, the various theories, models and implementations it embodies) but the CI's potential to provide a general interoperability solution.

8.3 Future Work

This research may be continued in a multitude of ways and in various directions. The following short paragraphs identify those considered the most fruitful.

8.3.1 Complete implementation of the sLCC. The development should proceed by first adding the ability to remove items (i.e. removeAttendee(), removeFloor() methods) followed by support to transfer floor responsibility and, finally, addition of a quit() function which would allow nodes to gracefully leave a conferencing session. The work described in this dissertation resulted in a complete design of the sLCC, which is described in [81]. This reference should be used in pursuing this suggestion starting from the current implementation as described herein and in [82].

8.3.2 Complete development of the crtiModel. In addition to the work presented on HLA ownership management presented here the DDM services have already been designed for this model [8]. Time management services and unrealized support services should be added. This will result in a complete description of the HLA based on a conferencing paradigm. This will not only be useful in completing the sLRC implementation (see next paragraph) but as a teaching tool as well, providing another, possibly more understandable, view of the HLA semantics.

8.3.3 Complete the sLRC implementation based on a completed crtiModel. This will result in a Java-based, open source implementation of the SIP-RTI (the current work is licensed under the GNU General Public License [85], which requires further code that incorporates it be open source as well). The work should proceed with completion of the Object and Ownership Management services followed by addition of the DDM and required Support Services. Time Management services and remaining Support Services should then be added. Interaction support should also be added; this should probably be

done in conjunction with the addition of the related object class services but could be done after all object class support has been completed.

8.3.4 The use of the SIP opens the way to provide support for other issues currently affecting RTIs. Investigation into the use of the SIP, as provided by the CI, in solving these problems would be very useful research. An initial approach would be to incrementally expand the use of the SIP in negotiating inter-sLCC communication. The SIP could first be used in the termination of inter-sLCC communication sessions and then to add features to the CI to allow the negotiation of session channel services such as QoS, alternative transport layer protocols, and security services. Firewall traversal techniques, perhaps using [56] and [58] as a starting point, should also be developed.

8.3.5 Continuing research into the use of the CI to support RTI and general interoperability should be very fruitful. In the area of RTI interoperability the translation of an existing open source RTI (such as the FDK or CERTI systems) to use the conferencing communication services provided by the CI would be a possible approach. In the area of general interoperability the porting of other distributed systems, perhaps a gaming system, to CI use would be interesting and provide insight into its potential in this area.

8.3.6 Development of a centralized CI would result in a session layer that could be used in situations benefiting from a centralized, rather than fully distributed, communication service. The existing implementation can already be used in such a scenario by implementing a single sLCC stack on a centralized node, however, a customized implementation eliminating messaging components and other distributed mechanisms will be more efficient in this scenario.

8.3.7 Exploration of alternate session control mechanisms; either a different use of the SIP or the use of an alternate communication control protocol may result in the discovery of further benefits and uses of the conferencing approach.

References

[1] F. Kuhl, R. Weatherly, and J. Dahmann, Creating Computer Simulations An Introduction to the High Level Architecture, Englewood Cliffs, N.J.: Prentice Hall, 1999.

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

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

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

[5] T. Pearce and N. Farid, "If RTI's Have a Standard API, Why Don't They Interoperate?," in Proceedings of the Simulation Interoperability Standards Organization Fall Simulation Interoperability Workshop, 2004.

[6] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "SIP: Session Initiation Protocol," Network Working Group, Request for Comments: 3261, The Internet Society, [Online document] June 2002, [2006 July 5], Available at HTTP: http://www.ietf.org/rfc/rfc3261.txt

[7] S. Holben, "Converging Distributed Modelling & Simulation With The Session Initiation Protocol," in Proceedings of the Summer Computer Simulation Conference, Montreal, 2003.

[8] C. Van Ham and T. Pearce, "Implementing RTI Data Distribution Management Using SIP Services," in Proceedings of the Simulation Interoperability Standards Organization European Simulation Interoperability Workshop, Toulouse, France, 2005.

[9] M. Myjak, S. Sharp, W. Wennie, J. Riehl, D. Berkley, P. Nguyen, S. Camplin, and M. Roche, "Implementing Object Transfer in the HLA," in Proceedings of the Simulation Interoperability Standards Organization Spring Simulation Interoperability Workshop, Orlando, 1999.

[10] T. Pearce, S. Holben, and C. Van Ham, "Implementing RTI Object Ownership Transfer Using SIP Services," in Proceedings of the Simulation Interoperability Standards Organization Spring Simulation Interoperability Workshop, San Diego, 2005.

[11] S. Donovan, "The SIP INFO Method," Network Working Group, Request for Comments: 2976, The Internet Society, [Online document] Oct. 2000, [2006 July 5], Available at HTTP: http://www.ietf.org/rfc/rfc2976.txt

[12] A. Niemi, "Session Initiation Protocol (SIP) Extension for Event State Publication," Network Working Group, Request for Comments: 3903, The Internet Society, [Online document] Oct. 2004, [2006 July 5], Available at HTTP: http://www.ietf.org/rfc/rfc3903.txt

[13] A. Roach, "Session Initiation Protocol (SIP) - Specific Event Notification," Network Working Group, Request for Comments: 3265, The Internet Society, [Online document] June 2002, [2006 July 5], Available at HTTP: http://www.ietf.org/rfc/rfc3265.txt

[14] M, Handley and V. Jacobson, "SDP: session description protocol," Network Working Group, Request for Comments: 2327, The Internet Society, [Online document] April 1998, [2006 July 5], Available at HTTP: http://www.ietf.org/rfc/rfc2327.txt

[15] United States Department of Defense, Defence Modeling and Simulation Office, N. Beauregard Street, Suite 500 Alexandria, Virginia, 22311-1705, U.S.A.

[16] Alion Science and Technology, AEgis Technologies Group, Inc., Carnegie Mellon University Software Engineering Institute, Johns Hopkins University Applied Physics Laboratory, Science Applications International Corporation, "Transition of the DoD High Level Architecture to IEEE Standard 1516 – Version 1.0," Applied Research Laboratories, The University of Texas at Austin, Signal and Information Sciences Laboratory, Ed., United States Department of Defense, Defence Modeling and Simulation Office, [Online document] Oct. 2005, [2006 July 5], Available at HTTPs: https://www.dmso.mil/public/transition/hla/IEEE_1516_transition

[17] Science Applications International Corporation, Virtual Technology Corporation, Object Sciences Corporation and Dynamic Animations Systems, "High Level Architecture Run-Time Infrastructure RTI 1.3-Next Generation Programmer's Guide, Version 5," Alexandria, Va.: United States Department of Defense, Defense Modelling and Simulation Office, 2002.

[18] P. Grammer, "HLA DMSO RTI Commercialization Announcement," [Online document] Aug. 2002, [2006 July 5], Available at HTTPs: https://lists.dmso.mil/pipermail/hla-online/2002-August/000006.html

[19] Pitch Technologies AB, Nygatan 35, S-582 19 Linköping, Sweden.

[20] MÄK Technologies, 68 Moulton Street, Cambridge, MA, 02138, U.S.A.

[21] Virtual Technology Corporation, 5510 Cherokee Avenue, Suite 350, Alexandria, VA, 22312 U.S.A.

[22] D. Stratton, S. Parr, and J. Miller, "Developing an Open-Source RTI Community," in Proceedings of the Simulation Interoperability Standards Organization Fall Simulation Interoperability Workshop, Arlington, Va., 2004.

[23] FDK - Federated Simulations Development Kit, [Online document] Nov. 2003, [2006 July 5], Available at HTTP: http://www.cc.gatech.edu/computing/pads/fdk/

[24] FDK - Federated Simulations Development Kit, [Online homepage] Jan. 2001, [2006 July 5], Available at HTTP: http://www-static.cc.gatech.edu/computing/pads/fdk.html

[25] R. Fujimoto, T. McLean, K. Perumalla, and I. Tacic, "Design of High Performance RTI Software," in Proceedings of the Fourth IEEE International Workshop on Distributed Simulations and Real-Time Applications, San Francisco, pp. 89-96, 2002.

[26] Java Run-Time Infrastructure, [Online homepage] Sept. 2004, [2006 July 5], Available at HTTP: http://xperts.sce.carleton.ca/JavaRTI/

[27] A. Kapolka, "The Extensible Run-Time Infrastructure (XRTI): An Experimental Implementation of Proposed Improvements to the High Level Architecture," Monterey: The MOVES Institute, Naval Postgraduate School, 2003.

[28] Extensible Run-Time Infrastructure, [Online document] n.d., [2006 July 5], (not) Available HTTP: http://xperts.sce.carleton.ca/JavaRTI/

[29] P. Siron, "Design and Implementation of a HLA RTI Prototype at ONERA," in Proceedings of the Simulation Interoperability Standards Organization Fall Simulation Interoperability Workshop, Orlando, 1999.

[30] B. Breholee and P. Siron, "CERTI: Evolutions of the ONERA RTI Prototype," in Proceedings of the Simulation Interoperability Standards Organization Fall Simulation Interoperability Workshop, Orlando, 2002.

[31] CERTI - Summary, [Online document] Nov. 2002, [2006 July 5], Available at HTTP: http://savannah.nongnu.org/projects/certi/

[32] CERTI, [Online homepage] n.d., [2006 July 5], Available at HTTP: http://www.cert.fr/CERTI/index.en.html

[33] M. Adelantado, J-L. Bussenot, J-Y. Rousselot, P. Siron, and M. Betoule, "HP-CERTI: Towards a high Performance, high Availability Open Source RTI for Composable Simulations," in Proceedings of the Simulation Interoperability Standards Organization Fall Simulation Interoperability Workshop, Orlando, 2004.

[34] SISO - Simulation Interoperability Standards Organization, [Online homepage] n.d., [2006 July 5], Available at HTTP: http://www.sisostds.org/index.php

[35] M. Myjak, D. Clark, and T. Lake, "RTI Interoperability Study Group Final Report," in Proceedings of the Simulation Interoperability Standards Organization Fall Simulation Interoperability Workshop, Orlando, 1999.

[36] A. Tolk, "Avoiding another Green Elephant - A Proposal for the Next Generation HLA based on the Model Driven Architecture," in Proceedings of the Simulation Interoperability Standards Organization Fall Simulation Interoperability Workshop, Orlando, 2002.

[37] K. Mullally, G. Hall, D. Gordon, B. Pemberton, and C. Peabody, "Open, Message-Based RTI Implementations – A better, Faster, Cheaper Alternative to Proprietary, API-Based RTIs?," in Proceedings of the Simulation Interoperability Standards Organization Spring Simulation Interoperability Workshop, Orlando, 2003.

[38] General Dynamics, "General Dynamics HLA Direct™ Run-Time Infrastructure Protocol, Version 1.0, Draft", Orlando, n.d.

[39] J. Woodyard, "Open Run-Time Infrastructure Protocol Study Group Final Report," in Proceedings of the Simulation Interoperability Standards Organization Fall Simulation Interoperability Workshop, Orlando, 2004.

[40] Base Object Models (BOMs), [Online homepage] n.d., [2006 July 5], Available at HTTP: http://www.boms.info/

[41] W. Braudaway and R. Little, "The High Level Architecture's Bridge Federate," in Proceedings of the Simulation Interoperability Standards Organization Fall Simulation Interoperability Workshop, Orlando, 1997.

[42] C. Bouwens, D. Hurrell, and D. Shen, "Implementing Ownership Management Services With a Bridge Federate," in Proceedings of the Simulation Interoperability Standards Organization Spring Simulation Interoperability Workshop, Orlando, 1998.

[43] K. Briggs, "A Required RTI Gateway Standard As a Solution To RTI Interoperability," in Proceedings of the Simulation Interoperability Standards Organization Spring Simulation Interoperability Workshop, Orlando, 1998.

[44] CORBA, [Online homepage] n.d., [2006 July 5], Available at HTTP: http://www.corba.org/

[45] J. Siegel, "A preview Of CORBA 3," IEEE Computer, May, pp. 114-116, 1999.

[46] A. D'Ambrogio and D. Gianni, "Using CORBA to Enhance HLA Interoperability in Distributed and Web-Based Simulation," in Proceedings of the Nineteenth International Symposium on Computer and Information Sciences, Antalya, Turkey, 2004.

[47] K. Schug, A. Jayasumana, S. Gupta, and P. Srimani, "CORBA Based HLA/RTI Design Approach," Colorado State University Computer Science Technical Report CS-97-109, Fort Collins, Co., 1997.

[48] M. Myjak, S. Sharp, and K. Briggs, "Javelin," in Proceedings of the Simulation Interoperability Standards Organization Spring Simulation Interoperability Workshop, Orlando, 1999.

[49] L. Granowetter, "RTI Interoperability Issues – API Standards, Wire Standards, and RTI Bridges," in Proceedings of the Simulation Interoperability Standards Organization Spring Simulation Interoperability Workshop, Orlando, 2003.

[50] K. Morse, D. Drake, and R. Brunton, "Web Enabling HLA Compliant Simulations to Support Network Centric Applications," in Proceedings of the Command and Control Research and Technology Symposium, San Diego, 2004.

[51] K. Morse, "XMSF Profile Study Group Final Report," in Proceedings of the Simulation Interoperability Standards Organization Fall Simulation Interoperability Workshop, Orlando, 2005.

[52] E. O'Tuathail and M. Rose, "Using the Simple Object Access Protocol (SOAP) in Blocks Extensible Exchange Protocol (BEEP)," Network Working Group, Request for Comments: 3288, The Internet Society, [Online document] June 2002, [2006 July 5], Available at FTP: ftp://ftp.rfc-editor.org/in-notes/rfc3288.txt

[53] World Wide Web Consortium SOAP Specification web page, [Online document] 2004, [2006 July 5], Available at HTTP: http://www.w3.org/TR/soap/

[54] M. Rose, "The Blocks Extensible Exchange Protocol Core," Network Working Group, Request for Comments: 3080, The Internet Society, [Online document] Mar. 2001, [2006 July 5], Available at FTP: ftp://ftp.rfc-editor.org/in-notes/rfc3080.txt

[55] A. Elkins, J. Wilson, and D. Gracanin, "Security Issues in High Level Architecture Based Distributed Simulation," in Proceedings of the 2001 Winter Simulation Conference, Arlington, Va., 2001.

[56] S. Holben, R. Johnson, and M. Herald, "Converging Software Architecture with Next Generation Distributed Technologies," in Proceedings of the Simulation Interoperability Standards Organization Fall Simulation Interoperability Workshop, Orlando, 2004.

[57] P. Bieber, J. Cazin, P. Siron, and G. Zanon, "Security Extensions to ONERA HLA RTI Prototype," in Proceedings of the Simulation Interoperability Standards Organization Fall Simulation Interoperability Workshop, Orlando,1998.

[58] S. Holben, "Enterprise UDP Firewall Traversal Using SIP," in Proceedings of the Simulation Interoperability Standards Organization Fall Simulation Interoperability Workshop, Orlando, 2005.

[59] W. Zong, Y. Wang, W. Cai, and S. Turner, "Grid Services and Service Discovery for HLA-Based Distributed Simulation and Real-Time Applications," in Proceedings of the 8th IEEE International Symposium on Distributed Systems and Real-Time Applications, Budapest, pp. 116-124, 2004.

[60] I. Foster, H. Kishimoto, A. Savya, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. Von Reich, "The Open Grid Services Architecture, Version 1.0," [Online document] Jan 2005, [2006 July 5], Available at HTTP: http://www.gridforum.org/documents/GWD-I-E/GFD-I.030.pdf

[61] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling, P. Vanderbilt, Eds., "Open Grid Services Infrastructure," [Online document] June 2003, [2006 July 5], Available at HTTP: http://www.ggf.org/documents/GFD.15.pdf

[62] The Globus Alliance, [Online homepage] n.d., [2006 July 5], Available at HTTP: http://www.globus.org/

[63] B. Givens, "Positions For and Against an Open-Source RTI," in Proceedings of the Simulation Interoperability Standards Organization Fall Simulation Interoperability Workshop, Orlando, 2000.

[64] J. Rosenberg and H. Schulzrinne "Models for Multi Party Conferencing in SIP," Sipping Working Group, Internet-Draft, The Internet Society, [Online document] July 2002, [2006 July 5], Available at HTTP: http://www3.ietf.org/proceedings/02jul/I-D/draft-ietf-sipping-conferencing-models-01.txt,

[65] M. Barnes, C. Boulton, and O. Levin, "A Framework and Data Model for Centralized Conferencing," XCON Working Group, Internet-Draft, The Internet Society, [Online document] June 2006, [2006 July 5], Available at HTTP: http://www.ietf.org/internet-drafts/draft-ietf-xcon-framework-04.txt,

[66] The Internet Engineering Task Force, [Online homepage] n.d., [2006 July 5], Available at HTTP: http://www.ietf.org/

[67] P. Koskelainen, H. Schulzrinne, and X. Wu, "A SIP-based Conference Control Framework," in Proceedings of the 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video, Miami Beach, 2002.

[68] Bormann, Kutscher, Ott, and Trossen, "Simple Conference Control Protocol – Service Specification," Network Working Group, Internet-Draft, The Internet Society, [Online document] Feb. 2001, [2006 July 5], Available at HTTP: http://ietfreport.isoc.org/all-ids/draft-ietf-mmusic-sccp-01.txt

[69] N. Kauser and J. Crowcroft, "An Architecture of Conference Control Functions," in Proceedings of Photonics East 1999, Boston, 1999.

[70] J. Lennox and H. Schulzrinne, "A Protocol for Reliable Decentralized Conferencing," in Proceedings of the 13th International Workshop on Network and Operating Systems Support for Digital Audio and Video, Monterey, 2003.

[71] J. Rosenberg, "A Framework for Conferencing with the Session Initiation Protocol (SIP)," Network Working Group, Request for Comments: 4353, The Internet Society, [Online document] Feb. 2006, [2006 July 5], Available at FTP: ftp://ftp.rfc-editor.org/in-notes/rfc4353.txt,

[72] L. Chen, C. Luo, J. Li, and S. Li, "Digiparty – A Decentralized Multi-Party Video Conferencing System," Microsoft Corporation White Paper, Microsoft Research Asia, 2004.

[73] R. Mahy, B. Campbell, R. Sparks, J. Rosenberg, D. Petrie, and A. Johnston, "A Call Control and Multi-party usage framework for Session Initiation Protocol (SIP)," Sipping Working Group, Internet-Draft, The Internet Society, [Online document] Mar. 2006, [2006 July 5], Available at HTTP: http://www.ietf.org/internet-drafts/draft-ietf-sipping-cc-framework-06.txt,

[74] A. Johnston and O. Levin, "Session Initiation Protocol Call Control - Conferencing for User Agents," Sipping Working Group, Internet-Draft, The Internet Society, [Online document] June 2005, [2006 July 5], Available at HTTP: http://www.softarmor.com/wgdb/docs/draft-ietf-sipping-cc-conferencing-07.txt

[75] C. Leopold, Parallel And Distributed Computing: A Survey of Models, Paradigms and Approaches, New York: John Wiley and Sons, Inc., 2000.

[76] J. Farley, Java Distributed Computing, O'Reilly's CD Bookshelf, 1998.

[77] A. Tanenbaum and M. van Steen, Distributed Systems Principles and Paradigms, Upper Saddle River, N.J.: Prentice Hall, 2002.

[78] J. Dingel, D. Garlan, and C. Damon, "Bridging the HLA: Problems and Solutions," in Proceedings of the 6th IEEE International Workshop on Distributed Simulation and Real Time Applications, Fort Worth, Tex., pp. 33-43, 2002.

[79] R. Wyllys and P. Doty, "Notes on the 5-Layer and 7-Layer Models of Interconnection," The University of Texas at Austin, Graduate School of Library and Information Sciences, [Online document] Feb 2001, [2006 July 5], Available at HTTP: http://www.ischool.utexas.edu/~l38613dw/readings/NotesOnInterconnection.html

[80] National Institute of Standards and Technology NIST SIP, [Online homepage] Feb 2001, [2006 July 5], Available at HTTP: http://snad.ncsl.nist.gov/proj/iptel/

[81] C. Van Ham, "SIP-RTI SIP Local Conferencing Component (sLCC) Data Distribution Design," Technical Report SCE–06–12, Ottawa-Carleton Institute for Electrical Engineering, Carleton University, Faculty of Engineering, Department of Systems and Computer Engineering, Ottawa, 2006.

[82] C. Van Ham, "SIP-RTI SIP Local Conferencing Component (sLCC) Software Implementation," Technical Report SCE–06–10, Ottawa-Carleton Institute for Electrical Engineering, Carleton University, Faculty of Engineering, Department of Systems and Computer Engineering, Ottawa, 2006.

[83] C. Van Ham, "SIP-RTI SIP Local RTI Component (sLRC) Software Implementation," Technical Report SCE–06–11, Ottawa-Carleton Institute for Electrical Engineering, Carleton University, Faculty of Engineering, Department of Systems and Computer Engineering, Ottawa, 2006.

[84] Eclipse, [Online homepage] 2006, [2006 July 5], Available at HTTP: http://www.eclipse.org/platform/

[85] GNU General Public License, [Online homepage] June 2006, [2006 July 5], Available at HTTP: http://www.gnu.org/copyleft/gpl.html#SEC4

[86] The Streaming API for XML, [Online homepage] 2006, [2006 July 5], Available at HTTP: http://stax.codehaus.org/

[87] Simple API for XML processing (SAX), [Online homepage] n.d., [2006 July 5], Available at HTTP: http://www.saxproject.org/

[88] World Wide Web consortium Document Object Model, [Online homepage] Jan. 2005, [2006 July 5], Available at HTTP: http://www.w3.org/DOM/

Appendix A – sLCC API

This is a listing of all public calls supported by the sLCC. It does not show call arguments. Reference [82] provides a listing with these details.

public void connect()
Once user has constructed conference architecture, it is ready to participate in conference and therefore can request connection to the network.

public synchronized static LCCfe getLCCInstance()
Creates (if necessary) and returns a reference to the singleton LCCfe instance. Only one sLCC is allowed per node.

public CallerID generateCallerID()
Creates and returns a new callerID. A valid callerID is required by all of the API calls to identify the user making the call.

public FloorID getFloorID()
Similar to the use of callerIDs, floorIDs identify floors placed in conferences. Before a user may call addFloor() it must have a floorID which it and the sLCC will use to track the new floor with.

public Integer createConference()
Used to create a new conference and add it to the conference map.

public void addAttendee()
Used to add a new attendee (speaker or listener) to a conference.

public void addFloor()
Used to add a new available floor to a conference.

public void changeOwner()
Used to change the owner of an available floor or to make it unowned.

public void aquisitionRequestReply()
Used to reply to a floor acquisition request.

public void makeAnAnnouncement()
Used to make an announcement to a conference.

public void stopLCC()
Used to stop all threads created by the sLCC and exit it.

The following calls support access to information maintained in the sLCC. Rather than redundantly maintain this in the user application (e.g. the sLRC) these calls may be used.

public ArrayList getSubConfList()
Returns a conference's sub-conference listing.

public Integer getConfIDfmFloorID()
Returns the confID of the conference containing the floor passed in.

public ArrayList getAttendeeListFmConfID()
Returns the attendee list for a given conference.

public Conference checkForConference()
Checks if a given conference exists in the conference map and returns a reference to it.

public boolean checkForAttendee()
Identifies if a given conference contains the attendee type for the identified caller.

public Floor findFloorFromFloorID()
Checks if the identified floor exists in the conference map by searching all conferences for it.

public boolean checkFloorOwnership()
Checks if a given floor is owned by caller or unowned.

public void printConfStructure()
A simple dump of conference map for use in testing to ensure CI is properly built.

Appendix B – The sLCC Message DTD

This appendix shows the XML Document Data Type used for all inter-sLCC messages. This is defined in the implementation as file "slcc.LCCMsg.dtd" which must be installed on every node supporting an sLCC. The implementation uses the Streaming API for XML [86] parser. After comparison to the two other popular XML parser types (Simple API for XML [87] and Document Object Model [88] based) this parser was found to be best suited to this application as it supports on-the-fly, iterative parsing of XML file streams.

Each Conference class object in the sLCC contains a ReplyParser class object which parses all XML streams passed too it by the MsgIOThread connected to the remote sLCC that sent the stream.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- LCCMsg.dtd defines possible LCC control messages and user announcements in
XML terms -->
<!ELEMENT message (
                        conference,
                        reply,
                        commands?,
                        announcements?)>

    <!ELEMENT conference EMPTY>
      <!ATTLIST conference confID NMTOKEN #REQUIRED>

    <!ELEMENT reply EMPTY>
              <!-- is this a reply, if yes attribute = "T", else its an initial command -->
      <!ATTLIST reply TF NMTOKEN #REQUIRED>

    <!ELEMENT commands (command?)>
      <!ELEMENT command (fids*, sif-list?)>
        <!ATTLIST command name NMTOKEN #REQUIRED> <!--PI, SI etc -->

      <!ELEMENT fids (fid+)>
        <!ATTLIST fids id_type_list NMTOKEN #REQUIRED> <!-- uA, oA, O or Os -->

      <!ELEMENT announcements (announcement?)>
        <!ELEMENT announcement EMPTY>
          <!ATTLIST announcement the_announcement CDATA #REQUIRED>
```

Appendix C – sLCC Use Rules

These are the rules users must follow in using the sLCC.

Responsibility Rules

1. – when first created a uA is the responsibility of the sLCC in which it is created
2. – if a floor is created as an oA (i.e. creator has an S – see rules 4 & 5), it is responsibility of sLCC on which it is created
3. – responsibility transfers with ownership

Ownership Rules

4. – only users with an S may own or acquire ownership of an A
5. – if the creator of an oA has an S, it automatically takes ownership of that floor when it is created
6. – the creator of a uA does not take ownership of that floor when it is created (even if the creator has an S in the affected conference when the floor is placed), the creator must acquire the thereafter, if desired
7. – in requesting an ownership transfer of a floor, the new owner must be local
8. – a floor may be *unowned* – only the current owner can do so

Miscellaneous Rules

9. – Ss without oAs automatically use the conference O to make announcements
10. – an S with an oA cannot use an O, it only uses its oA
11. – an sLCC must participate in the parent conference before it can place anything in a sub-conference on behalf of its users
12. – a given user may have no, or, at most, one S and one L in any given conference
13. – a given user may own multiple floors through its S

Acronyms used:

L	- Listener
O	- Open (floor)
oA	- owned Available floor
S	- Speaker
uA	- unowned Available floor

Appendix D - IEEE 1516 Services Implemented by the sLRC

Federation Management

4.2 - Create Federation Execution
4.4 - Join Federation Execution

Declaration Management

5.2 - Publish Object Class Attributes
5.6 - Subscribe Object Class Attributes

Object Management

6.4 - Register Object Instance
6.5 - Discover Object Instance†
6.6 - Update Attribute Values
6.7 - Reflect Attribute Values†

Ownership Management

7.2 - Unconditional Attribute Ownership Divestiture
7.7 - Attribute Ownership Acquisition Notification†
7.8 - Attribute Ownership Acquisition
7.11 - Request Attribute Ownership Release†

Support Services

10.2 - Get Object Class Handle
10.4 - Get Attribute Handle

Java Classes Required and Implemented (from [3], Annex B)

FederateHandle
AttributeHandle
ObjectClassHandle
ObjectInstanceHandle
AttributeHandleSet
AttributeHandleSetFactory
AttributeHandleValueMap
AttributeHandleValueMapFactory

Note: The dagger (†) symbol indicates a callback.

Appendix E – Air Traffic Controller Demonstration Federation Object Model (FOM) File

The FOM, in addition to defining other information needed by an RTI, describes the data that can be exchanged between federates using two data representations: objects and interactions. They are defined in the FOM using an inheritance scheme. In this scheme, all object classes are sub-classes of HLAObjectRoot. Each sub-class object inherits the attributes of its parent. The HLAObjectRoot contains one attribute: the HLAprivilegeToDeleteObject attribute (the owner of which may delete a given object class instance).

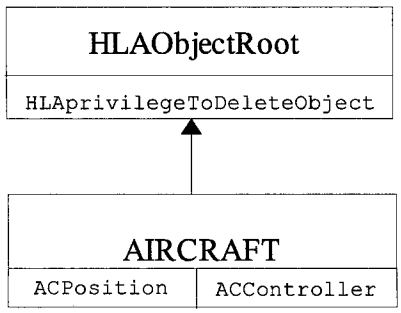

Figure E-1 Air Traffic Controller Federation Object Class Hierarchy Diagram

The object class hierarchy may be represented in an object class hierarchy diagram; the object class hierarchy diagram for the air traffic controller demonstration federation is shown in figure E-1. It is derived from the FOM listed below. A FOM is an XML document and must be based on the HLA document type definition found in [4]. The demonstration federation FOM defines one object class, "AIRCRAFT", which has two attributes: "ACPosition" and "ACController".

```
<?xml version="1.0"?>
<!DOCTYPE objectModel SYSTEM "HLA.dtd">
<objectModel
      DTDversion="1516.2"
      name="Example"
      type="FOM"
      version="1.0"
      date="2000-04-01"
      purpose="Demo FOM - Claude Van Ham"
      sponsor="Claude Van Ham">
      <objects>
            <objectClass name="HLAobjectRoot">
                  <attribute name="HLAprivilegeToDeleteObject"/>
                  <objectClass name="AIRCRAFT">
                        <attribute name="ACPosition"/>
                        <attribute name="ACController"/>
                  </objectClass>
            </objectClass>
      </objects>
</objectModel>
```

Appendix F - Air Traffic Controller Demonstration Federation Conference Map

The conference map shown in figure F-1 results when the FOM in Appendix E is parsed by an sLRC, the required conferences created in the CI and the attendees representative of the air traffic controller demonstration federation are placed by calls to the SIP-RTI's publish and subscribe services.

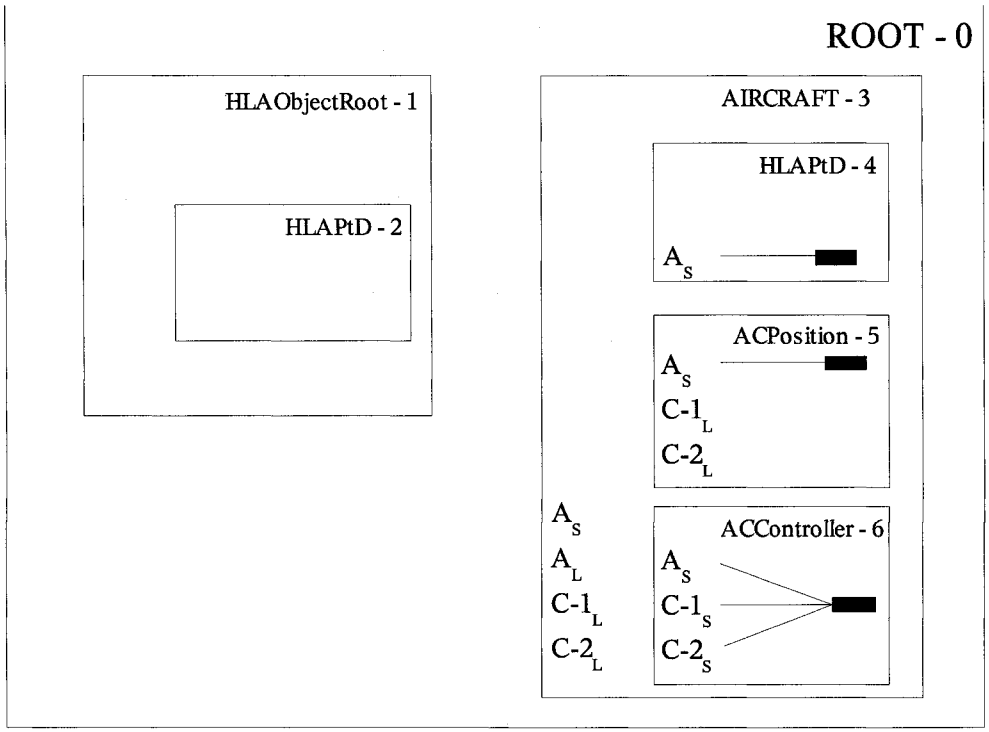

Figure F-1 Air Traffic Controller Federation Conference Map

The SIP-RTI creates all object class conferences as "first level" user conferences, that is they are the first level accessible to users, the ROOT being accessible only to the CI. Each object class conference is parent to its attribute conferences. In this diagram 'A', 'C-1' and 'C-2' represent the aircraft, controller-1 and controller-2 federates' attendees respectively. The numbers in the upper right corner of each conference box indicate its conference ID or "confID" which is used by the CI to identify each. They are preceded by the conference names which are taken from the FOM and used by the federates in identifying the related objects or attributes when requesting an RTI handle from the SIP-RTI.