Proceedings of the 1998 Winter Simulation Conference
D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, eds.

DISTRIBUTED SIMULATION MODELING: A COMPARISON OF HLA, CORBA, AND RMI

Arnold Buss

Operations Research Department
Naval Postgraduate School
Monterey, CA 93943 U.S.A.

Leroy Jackson

United States Army
TRADOC Analysis Center
Monterey, CA 93943 U.S.A.

ABSTRACT

The execution of distributed simulations has become increasingly important to the Department of Defense (DOD). This paper compares three architectures for supporting distributed computing, HLA, CORBA, and RMI. While the fundamental structure of each is similar, there are differences that can profoundly impact an application developer or the administrator of a distributed simulation exercise.

1 INTRODUCTION

The design and execution of distributed simulation models has become increasingly important to the Department of Defense (DOD). In recent years, the DOD has invested considerable resources in infrastructures for distributed simulation modeling. The current focus is the High Level Architecture (HLA) spearheaded by the Defense Modeling and Simulation Office (DMSO) (US Department of Defense, 1998). The HLA benefits greatly from two earlier DOD efforts: the Distributed Interactive Simulation (DIS) protocol standards, and the Aggregate Level Simulation Protocol (ALSP). There have been several efforts in the commercial sector to enable distributed computing. Two of the most viable recent efforts are the Common Object Request Broker (CORBA), by the Object Management Group (OMG), and Remote Method Invocation (RMI), from Sunsoft’s Java Development Kit (JDK).

There are many reasons why DOD has an interest in a common framework for performing distributed simulation. Declining defense budgets have increased the necessity for cost containment. Increased use of simulation for training, acquisition, and analysis promises to substantially reduce costs. A common architecture for distributed simulation enhances interoperability and reuse in various DOD simulation modeling efforts.

Each of these three architectures for distributed computing offers much to the problem of executing a distributed simulation model. Each has a fundamental world view that affects the structure of its architecture. This paper will compare the features of these three important technologies with particular focus on their impact on distributed simulation. Naturally, it is impossible to cover all aspects of the architectures, so this paper will only touch briefly on the salient features of each, emphasizing those aspects impacting the simulator.

In the following section we will discuss the core elements of distributed architectures. Sections 3 through 5 will touch on the important features of HLA, CORBA, and RMI, respectively, followed by a brief discussion and comparison in Section 6. Section 7 will present conclusions and recommendations.

2 BASIC ELEMENTS OF DISTRIBUTED ARCHITECTURES

We will focus our attention on three of the basic elements of distributed architectures: an object interface language, an object manager, and a naming service. In addition, we will consider issues such as the programming languages supported, the hardware and operating systems, and the network protocols used.

An object interface language is important for supporting distributed applications because they require a more abstract level of communications than ordinary applications. An object must make only minimal assumptions about the implementation of another object’s method since that method could involve objects on another machine. In contrast to a class’s definitions of methods, an interface is a contract for implementing objects and contains only a list of methods. Interfaces are therefore the ideal vehicle for providing interoperability between distributed objects. HLA and CORBA take a multilingual approach to distributed objects so they define their own separate interface specifications that are distinct from the implementing languages. RMI, on the other hand, is a language specific approach and thus uses the Java language interface for its interface specification. This considerably simplifies many design issues but limits the extent to which RMI can deal with distributed objects in other languages. Using interfaces is a critical factor for implementing exercises involving some combination of live, virtual, and constructive simulations, since the underlying mechanisms in these three types of simulation are fundamentally different.

The object manager is responsible for passing object references to requesting clients, instantiating objects as necessary and marshalling object requests between different machines. Objects can therefore be indifferent to whether invoking a given method actually executes in local code or remote code since the Object Manager hides the details. Conceptually, the object manager is a backbone through which objects on all machines communicate. The object manager may in fact be physically located on a server, located on both a client and a server or on a machine entirely separate from the client and server. In a well-designed architecture, however, the physical location of the object manager should be irrelevant to the application designer.

The naming service is the mechanism by which a server informs clients about objects available for access. The implementation of these services can range from a simple listing to a complex database of objects. Clients are able to discover the objects served, discover necessary signatures and arguments for various methods, obtain a reference to an object, and begin invoking methods on that object. This capability opens extremely flexible and dynamic possibilities for distributed computing, since the process of establishing communication between objects can be delayed until runtime and need not be hard-coded into applications.

A distributed architecture can be language-specific (RMI) or language-neutral (HLA, CORBA). A language-specific approach can assume more about the distributed objects--essentially all features of the language. The disadvantage is that legacy systems may be implemented in different, incompatible languages. The language-specific approach makes it difficult to incorporate these legacy programs. Language-neutral architectures can incorporate legacy applications written in any supported language, although the transition can involve difficult programming efforts. There are considerably more opportunities for interoperability between disparate programs that may not have been developed with distributed computing in mind. However, bindings must be provided for each language to be supported and, for interoperability to be truly achieved, each must be able to work with all others.

3 HLA

The highest priority effort in the Department of Defense (DOD) for modeling and simulation is the development of a common technical framework. The High Level Architecture (HLA) is the standard technical architecture for all DOD simulations. It consists of the major functional elements, the interface specifications and the design rules that together provide a common framework for specific system architecture designs. The HLA resulted from a process that included government, industry, and academia. The HLA has been accepted as a draft IEEE standard supported by the Simulation Interoperability Standards Organization (SISO).

HLA is applicable to a broad range of functional areas ranging from training to analysis to systems acquisition. HLA is applicable to constructive simulations with pure software representations, to man-in-the loop simulations, and to interfaces to live systems.

The HLA design principles envision federations of simulations composed from modular components with well-defined functionality and interfaces. A federation is the combination of a particular federation object model (FOM), a set of federates and the run-time infrastructure services (RTI). Federates include simulation utilities, simulations, and live player interfaces. The RTI is a distributed operating system for the federation. Specific simulation functionality is purposely separated from the general purpose, supporting, RTI.

There are three main components to the HLA: the HLA rules, the HLA interface specification, and HLA object model template (OMT).

3.1 HLA Rules

The first component of the HLA definition is the HLA Rules that describe the responsibilities of simulations with respect to the RTI in an HLA compliant federation. There are five federation rules and five federate rules (US Department of Defense, 1996a, 1998):

Federation Rules

(1) Federations shall have a FOM in OMT format.

(2) All representation of objects shall be in the federates and not the RTI.

(3) During federation execution, all exchange of FOM data shall be via the RTI.

(4) During federation execution, all federates shall interact with the RTI in accordance with the interface specification.

(5) During federation execution, an attribute of an instance of an object may be owned by only one federate at a given time.

Federate Rules

(6) Federates shall have a SOM in OMT format.

(7) Federates shall be able to update/reflect attributes and send/receive data in accordance with their SOM.

(8) Federates shall be able to transfer/accept attribute ownership in accordance with their SOM.

(9) Federates shall be able to vary the conditions under which they provide attribute updates in accordance with their SOM.

(10) Federates shall be able to manage local time in a way which will allow them to coordinate data exchange with other members of the federation.

3.2 HLA Interface Specification

The second component of the HLA definition is the interface specification, a standard for federates to interact with the RTI (Us Department of Defense, 1998). It defines how RTI services are accessed. The interface specification is provided as an application programmer interface (API) in several forms including CORBA IDL, C++, Ada95 and Java. The interface specification has six basic RTI service groups: federation management, declaration management, object management, ownership management, time management and data distribution management. Note that this "interface specification" is not related to the interface language discussed in Section 2.

Federates use Federation Management services for creation, dynamic control, modification and deletion of a federation execution. The HLA specification does not prevent a single software system from participating in a federation execution as multiple federates nor does it preclude a single software system from participating in multiple, independent federation executions. Current RTI implementations, however, may not necessarily support this feature. Federation Management services also include control checkpoint, pause, resume and restart features.

Federates use Declaration Management services to declare their intent to publish and subscribe to object attributes and interactions. Federates must invoke Declaration Management services prior to registering object instances, updating instance attribute values, and sending interactions. The effects of declaration management are independent of federation time.

Federates use Object Management services to deal with registration, modification and deletion of object instances and the sending and receipt of object interactions. Object Management services are complimented by Data Distribution Management services.

Federates use Ownership Management services to transfer ownership of instance attributes. This capability supports cooperative modeling in the federation.

Federates use Time Management services to coordinate the advance of logical time and maintain its relationship to real time. Time is represented as points along a federation time axis. Each federate may advance along the axis during federation execution, but that advance may be constrained by other federates. In general, time advances are coordinated with the Object Management services so that information is delivered in a causally correct and ordered fashion. Messages are either time stamp ordered or receive ordered.

Federates use Data Distribution Management (DDM) services to reduce the transmission and receipt of irrelevant data. DDM adds to the normal Object Management the ability to further refine the data requirements at the instance attribute level. These DDM services support the efficient routing of data.

3.3 HLA Object Model Template

The third component of the HLA definition is the Object Model Template (OMT), a common method for prescribing the information contained in the HLA object model for each federation and simulation (US Department of Defense, 1996a, 1996b). OMT is the interface language for HLA. Object models describe the set of shared objects in a simulation or federation, the attributes and interactions of these objects, and the level of detail at which the objects represent the real world including their spatial and temporal resolution. The HLA OMT provides a common representational framework for object model documentation. The OMT fosters simulation interoperability and the reuse of simulations.

There are two types of object models in HLA, Federation Object Models (FOMs) and Simulation Object Model (SOMs). Both types of models are documented using the OMT. The FOM contains all shared information (objects, attributes, interactions & parameters) essential for a particular federation. The SOM contains all federate information (objects, attributes, interactions & parameters) which is visible to other federates in a federation and all information from other federates that may be reflected in the federate.

An attribute is the named portion of an object's state. An interaction is a change in the sending object state which may cause a state change in another, receiving, object. A parameter is the information associated with an interaction provided by the sending object to the receiving object(s). Federates update attributes by providing the new instance attribute value for an attribute, and reflect attribute changes by receiving the new instance attribute value for an attribute.

HLA's approach to interoperability is through the ability to publish and subscribe to attributes and interactions. These are discovered through the federation's FOM. Local object interaction is substantially different from remote interaction, since the latter is possible only by the receipt of the change in a subscribed attribute.

In HLA, the object interface language is defined using the OMT, the object manager is the RTI and the naming service is the federation execution (A federation execution is an instance of the Create Federation Execution service invocation and entails executing the federation with a specific FOM and an RTI, and using various execution details.)

4 CORBA

CORBA is a non-commercial venture by the Object Management Group (OMG), a consortium of over 800 members that was founded in 1989 (Object Management Group, 1998; Orfali and Harkey, 1998). It is the oldest and perhaps the most mature of the three architectures we consider in this paper. CORBA is an extremely large and complex collection of specifications and protocols, and in a brief paper such as this, we can only touch on its most salient features.

4.1 CORBA Interface Language

The interface language for CORBA programs is the Interface Definition Language (IDL). The IDL syntax is essentially that of C++, except that IDL defines interfaces rather than implementations. In a CORBA application, the IDL is written first, then compiled into code in one of the supported languages. The elements defined in the IDL are then implemented in that language using the generated code as the basis. IDL has four primary elements: modules, interfaces, operations, and attributes.

A module is a namespace that bundles one or more interfaces. An interface is a collection of attributes and operations that correspond to an object. An interface may be viewed as a contract to implement the defined operations as corresponding methods with identical signatures and return types, and to provide the appropriate accessor methods corresponding to attributes. Interfaces may define an inheritance hierarchy. All objects implementing an interface must have methods corresponding to that interface's operations and attributes as well as those of all inherited interfaces. Multiple inheritance of interfaces is supported; however, an interface cannot inherit from two interfaces having the same name for an operation or for an attribute. CORBA 2.0 specifies that an object can have only one interface. However, for CORBA 3.0 there are proposals to support multiple interfaces.

Attributes correspond to instance variables and are used to represent data. Attributes are either basic types, constructed types, or object references. The possible basic types are the usual primitive data types (short, int, long, float, double, boolean, etc). The constructed types roughly correspond to those available in C++. A struct can be defined using typedef. A sequence corresponds to a variable-length array. Finally, the any type may be used to represent any kind of data. Any is a very powerful and flexible way of representing data, since it is self-describing. An object reference is used to invoke methods on an object. Although not specified in IDL, an attribute's implementation typically uses accessor methods rather than providing direct access to an instance variable. An attribute may be defined to be read-only or both read and write.

An Operation corresponds to a method and, like a method, is identified by its name, signature, and return type. The arguments of an operation may be defined to be in, out, or inout, depending on whether the argument is passed from the calling object to the invoked object, from the invoked object to the calling object, or both.

CORBA is language neutral in the sense that CORBA clients and servers may be implemented in any of the supported languages and be able to work together without even knowing each other's language. More importantly, under CORBA any participant need make no assumptions regarding the implementing language of other CORBA clients or servers. Currently the supported languages are C, C++, Smalltalk, Ada, Cobol, and Java. Defining interfaces rather than classes is a key element to this language neutrality.

4.2 CORBA Object Manager

The Object Manager for CORBA is the Object Request Broker (ORB). The ORB enables objects to send and receive messages from objects without regard to whether they are local or remote. All messages between client and server objects must go through the ORB. Typically, a client requests a reference to an object on a server with the intent of invoking methods on it.

Although the ORB is conceptually an entity between client and server objects, in fact it consists of software residing on both client and server machines. When the client requests an object, the message first goes to the ORB on its machine. The client ORB establishes communication with the ORB on the server. The server returns a reference to the requested object to its ORB, which passes it to the client ORB who returns it to the client. This is transparent to the programs, but does imply that every object participating in a CORBA application must reside on a machine with the ORB software installed.

CORBA has both static and dynamic means for a server to provide remote objects. The static method involves client and server "stubs," each of which is an interface to the actual object on the server. The server stub is often referred to as a "skeleton." A Stub links an object to the ORB on its machine and is typically generated from IDL, so that the modeler need not be concerned with writing calls to the ORB. To implement a distributed object in this manner, one starts with the IDL for the class and generates the stubs and skeletons. Only the skeleton code is used on the server, while only the stub code is used on the client. The stub and skeleton need not even be in the same language, since all communication is done via the ORB. For example, a C++ object on a server could have Java client stubs generated from its IDL. Importantly, neither the client nor the server has to make any assumptions about the other's language.

4.3 CORBA Naming Service

The static use of stubs does require each client to have specific code for the desired remote object, which must be created and compiled before the remote objects may be obtained. CORBA provides a Dynamic Invocation Interface (DII) to avoid the need for pre-compiled stubs. With DII the client first "discovers" the remote object by one of several mechanisms. Next, the interface to that object is obtained so the client can determine which method it wants to invoke. Information about the method (argument list, return type, and exceptions thrown) is then obtained. The invocation request is created and sent, invoking that method on the remote object.

A server can create an Interface Repository containing interfaces that may be dynamically accessed by clients. An interface repository may be browsed to locate classes that are of interest to a client. Interface repositories contain the IDL for the published objects.

For CORBA clients and servers operating on a single local area network (LAN) communication is more or less automatic. The ORB software on each machine is typically able to discover all other ORBs on the LAN, so interoperability is transparent to each client. CORBA provides the ability to communicate with ORBs on different LANs using the Internet Inter-Orb Protocol (IIOP), which extends TCP/IP.

5 RMI

Beginning with the Java Development Kit 1.1 (JDK1.1), the Java programming language has included Remote Method Invocation (RMI) as part of the standard Java libraries (Javasoft, 1997). RMI support is contained in four sparse packages in the JDK1.1 distribution devoted to identifying a remote object (via a "marker" interface) and throwing remote exceptions, registering remote objects, serving remote objects, and performing remote garbage collection. RMI is an object-oriented type of Remote Procedure Call (RPC). As its name implies, it involves invoking a method on a remote object. RMI is designed to minimize the differences between using ordinary (local) and remote objects.

RMI is Java-centric, and shares Java's platform-independence. The requirements for implementing an RMI client or server are simply having an implementation of the Java Virtual Machine (JVM), and RMI classes are immediately portable to all JVM platforms.

5.1 RMI Interface Language

Unlike HLA and CORBA, RMI does not define its object interface language separately from the implementing language. Rather, RMI uses Java's own interface syntax as its object interface language (Javasoft, 1997; Farley, 1997). This considerably simplifies application design and programming, since it is in one rather than two languages. A Java interface is denoted as being remote by inheriting the java.rmi.Remote interface, a "marker" interface that does not require any methods to be implemented, and serves only as an identifier to the compiler. All methods defined in a remote interface must throw a java.rmi.RemoteException or one of its subclasses.

5.2 Object Manager

RMI approaches the implementation of remote objects in essentially the same manner as CORBA. Each remote interface may be compiled separately into a client-side stub and a server-side skeleton. Users developing from scratch may subclass java.rmi.server.UnicastRemoteObject, which provides the functionality for serving remote objects on a point-to-point basis. Remote objects will typically subclass UnicastRemoteObject and implement the desired remote interface. The remote object is instantiated and bound to a name using the java.rmi.Naming class. The Naming class uses a system similar to that of Uniform Resource Locator (URL): "rmi://host:port/object name".

An important aspect of RMI is the Security Manager. Since running remote objects involves a potential security risk, RMI requires that an instance of java.rmi.RMISecurityManager be used to implement a security policy. The RMISecurityManager is responsible for determining whether methods are being invoked locally or remotely and protecting against potentially unsafe operations. If an instance of RMISecurityManager has not been set, then only classes from the local machine may be loaded into the application. The RMISecurityManager is extremely restrictive in that all non-remote operations are disabled. The programmer may, however, define his own security manager instead.

5.3 RMI Naming service

The java.rmi.registry.Registry class provides methods for binding remote objects to names, listing the available objects on a server, and looking up a desired remote object. The registry actually plays the role of a "server" in RMI, since the registry must be run as a separate process before objects can be registered and served. A client may use the registry to locate remote classes, download the appropriate client stubs, and begin invoking methods. However, the client does not need any special processes, since the mechanisms for client behavior are all contained in the JDK specification, of which RMI is a part. Remote connections in RMI are made using JDK's built-in networking capabilities, with packets being transmitted using TCP/IP.

Since RMI is Java-based, it may only interact with non-Java applications via the Java Native Interface (JNI). While technically feasible, JNI's complexity does not appear to offer substantial benefit for porting legacy applications at this time. As with most Java-centric approaches, interfacing with non-Java legacy applications remains problematic.

6 COMPARISONS

There are a number of disparities between the three approaches. Both CORBA and HLA are concerned with legacy applications, possibly in different languages. CORBA's basic approach is to provide support for CORBA-compliant middleware that communicates with legacy applications. Since CORBA supports most major language bindings, the middleware can be implemented in the most convenient language and be guaranteed to interoperate with any CORBA client. Although HLA has API's for C++, Ada, and Java, the responsibility for interoperability between federates in different languages is placed on the RTI implementers. This adds a burden to implementing an HLA federation in multiple languages, in contrast to CORBA imposing absolutely no overhead whatsoever for cross-language compatibility. RMI being a Java-based technology is essentially not cross-language at all. Interoperability to non-Java programs must be done via JNI and has no common interface, as with CORBA. However, Java's inherent cross-platform capabilities substantially increase the number of platforms on which distributed applications may be run.

CORBA and RMI are oriented towards general applications, whereas HLA is specifically targeted at distributed simulations. Consequently, HLA has considerably more infrastructure directly aimed at supporting simulation models through the Federation Rules and the simulation-specific services, such as Time Management. Since all simulations have the concept of a simulated clock, this service is essential to proper functioning of an HLA federation. However, a distributed CORBA or RMI application may not even have the notion of simulated time, so it would make no sense for either of those architectures to include such features. The HLA rules impose much stricter constraints on federates than either CORBA or RMI. For example, it is entirely feasible for a Java client to use both RMI and CORBA to communicate with remote objects. Indeed, limitations in the ability of CORBA to incorporate non-CORBA objects may necessitate such a possibility in some cases.

HLA provides publishing and subscription services but does not support direct communication between objects, as CORBA and RMI do. The communication between remote objects in either of the latter two architectures can be substantially more complex and expressive, essentially no less so than ordinary communication between objects.

HLA's notion of transfer of object ownership is a unique capability among the three architectures. This capability can be a powerful modeling tool in certain types of simulation. For example, an aircraft modeled in one application can carry missiles whose dynamics when launched are provided by another model. The aircraft's model can own the missile until it is launched, upon which time ownership is transferred to the missile's model. In CORBA and RMI, an object instantiated by a server is always owned by that server.

Both CORBA and RMI use specific communication protocols for network transmission. RMI uses TCP/IP, the most common internet protocol, whereas CORBA defines its own Internet Inter-Orb Protocol (IIOP) that builds on TCP/IP. HLA, on the other hand, does not specify a protocol, but leaves the choice up to the RTI implementers. This is unfortunate, since it may preclude interoperability between RTI implementations by different vendors. CORBA 1.0 likewise did not specify protocols, and early ORB implementations suffered exactly such a lack of interoperability.

A singular advantage of RMI over the other two architectures involves security. As described above, the RMISecurityManager ensures that no hostile code can have access to local resources. There are classes in JDK that implement encryption and digital signatures, which can be used to transmit sensitive information in the clear and verify the identity of the sender.

7 CONCLUSIONS

HLA, CORBA, and RMI take similar approaches to enabling distributed computing and have roughly analogous mechanisms for the three basic components, object interface language, object manager, and the naming service.

For situations involving legacy simulation models written in different languages, HLA provides more than the other two, in large part due to its orientation toward simulation and its simulation-related services. If legacy databases are involved, however, CORBA is probably the best choice for implementing middleware to serve the database's information to distributed clients. For situations in which much of the implementation is new, RMI is perhaps the superior choice due to its tighter relationship with the implementing language and the superiority of Java as an Object-Oriented language.

REFERENCES

Farley, Jim (1997) *Java Distributed Computing*, O'Reilly, Cambridge, MA.

Javasoft, Inc. (1997) *Remote Method Invocation Specification.*

Object Management Group (1998) *The Common Object Request Broker: Architecture and Specification.*

Orfali, Robert and Dan Harkey (1998) *Client/Server Programming with Java and CORBA*, John Wiley and Sons, Inc., New York, NY.

US Department of Defense (1996) *High Level Architecture Object Model Template,* IEEE P1516.2.

US Department of Defense (1998) *High Level Architecture Interface Specification,* Version 1.3 of IEEE P1516.1, M&S HLA – Federate I/F Spec, DRAFT 1 of 20 April 1998.

US Department of Defense (1996) *High Level Architecture Glossary*.

AUTHOR BIOGRAPHIES

ARNOLD H. BUSS is a Visiting Assistant Professor of Operations Research at the Naval Postgraduate School. He received a BA in Psychology from Rutgers University, and MS in Systems and Industrial Engineering from the University of Arizona, and a Ph.D. in Operations Research from Cornell University. His research interests include simulation modeling and object-oriented software design. He is a member of INFORMS, MORS, POMS, and IIE.

LEROY A. JACKSON, Major, US Army, is an artillery officer with over 20 years of enlisted and commissioned service. He earned a B.A. in Mathematics from Cameron University in 1990 and an M.S. in Operations Research from the Naval Postgraduate School in 1995. He is currently assigned as an operations research analyst at the U.S. Army Training and Doctrine Command (TRADOC) Analysis Center (TRAC) Research Activities in Monterey, California and he continues his graduate studies at the Naval Postgraduate School.