Title: Generic Distributed Simulation Architecture

Author(s): Clay P. Booker

Submitted to:
DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, make any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.
DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.
Generic Distributed Simulation Architecture

by

Clay P. Booker, TSA-3

ABSTRACT

A Generic Distributed Simulation Architecture is described that allows a simulation to be automatically distributed over a heterogeneous network of computers and executed with very little human direction. A prototype Framework is presented that implements the elements of the Architecture and demonstrates the feasibility of the concepts. It provides a basis for a future, improved Framework that will support legacy models. Because the Framework is implemented in Java, it may be installed on almost any modern computer system.

INTRODUCTION

There are a variety of circumstances where a simulation is naturally distributed. For example, in the case of a test and evaluation setting, items of physical hardware and even human operators may be interacting with each other and with software components in a simulation. Clearly, the distinct pieces of hardware and the human operators constitute disbursed elements of the simulation, and the additional distribution of the software elements over an assembly of computers is not remarkable. However, the ability to distribute almost any simulation over a collection of computers can offer many advantages depending on the circumstances; among these advantages are:

1. More than one CPU can be allocated to a single problem without special hardware requirements.
2. Existing hardware, such as computers and networks, can be utilized in off-hours. New and even esoteric, special-purpose computers need not be purchased for a particular set of large problems.
3. Using a heterogeneous collection of computers in the distributed framework, the user can match the requirements of an individual simulation component to a particular computer’s strengths. For example, a user might place a graphics intensive component of the simulation on a Silicon Graphics computer while a CPU intensive component is executed on a DEC Alpha.

Unfortunately, distributed simulations can be difficult to implement, control, and use. They are frequently required to run in a heterogeneous environment, and for most programming languages that necessitates extensive localization of the component code for each computer as well as localization and complex installation of the distributed simulation’s infrastructure. In addition, the components making up a simulation must somehow be disbursed among the computers and then synchronized each time the simulation is to be run; this necessitates either intensive operator involvement in manually moving, compiling, and linking code which is prone to error, or an automated system to disseminate the simulation elements among the computers.

In order to realize the benefits of a Distributed Simulation Architecture and to minimize the difficulties, a Generic Distributed Simulation Architecture (GDSA) is proposed here. It is implemented in Java to avoid most localization problems that previously had been inherent with a heterogeneous system, but it can easily accommodate native code thus preserving the ability to use legacy models.
ELEMENTS OF A DISTRIBUTED SIMULATION

The physical infrastructure of a distributed simulation is fundamentally a collection of computers connected by way of a network. On each computer a Player, a self-contained application that represents any entity in the simulation, executes and communicates via some mechanism with other Players in the simulation. A Simulation Engine may impose synchronization and manage the collection of Players that make up the distributed simulation. This concept is shown conceptually in Figure 1.

Details may differ from implementation to implementation; some may interpose an Object Request Broker (ORB) between the Players to effect communication and coordination, but the essential relationship is effectively that as represented in Figure 1.

The elements of a distributed simulation may be broken down further as required. In Figure 1, the time synchronization is broken out of the Simulation Engine; this is not necessary, but is probably expedient for a real time simulation. Of course the Simulation engine may control the Time Server or work cooperatively with it. The other two main elements are:

1. Communication. Typically, communication between all elements is implemented via sockets, but simulation designers may impose other layers in the communication element.

For example, in Java implementations, Remote Method Invocation (RMI) which allows the
transmission of objects over a network is available, but RMI is simply a protocol based on sockets.

2. Coordination. Besides time synchronization, Players require coordination during the initialization stage, while executing, and in the termination stage. For real-time simulations, a synchronized set of clocks may be sufficient for coordination during execution, but synchronization of the time-independent initialization phases is required. For example, suppose that in initialization phase N, Player A requires some information from Player B; if Player B has not reached the initialization phase where that information is determined, it will not be able to pass that information to Player A. Accordingly, it is generally desirable to conduct the initialization stages so that no Player is allowed to proceed to initialization phase N+1 until all Players have completed phase N. For faster-than-real-time simulations, more rigorous coordination during execution may be required than that provided by a computer clock.

The Generic Distributed Simulation Architecture consists of the elements and their linkage as depicted in Figure 1 along with the selection of the particular types of links between the elements; RMI is chosen for coordination links, and Player-to-Player links are left up to the simulation developer. The further implementation of particular enabling components, applications programmed to link via the specified protocol and perform management tasks, constitutes a GDSA Framework. The particular applications making up the GDSA Framework are discussed in the following section.
In the past, distributed simulations have been successfully built based on the generalized concepts shown in Figure 1. However, most suffer from the need to localize the software to each computer's hardware and Operating System (OS). SAMSON\textsuperscript{1}, which is a real-time, distributed simulation suitable for a test and evaluation environment, is built along these lines; for SAMSON's applications, speed of communication is paramount, and the Simulation Engine and communication links are optimized accordingly. Other distributed simulations where communication speed is less of a requirement, may use ORBs for coordination and communication, and as mentioned previously, the diagram in that case would differ somewhat from Figure 1 with the addition of more layers. With ORBs, localization is more abstracted, but the ORB still must be native to the hardware on which it executes, and the Player may require some localization. The GDSA Framework avoids most of the localization problem by implementing the elements in Java. As a result, each element may be installed on any computer in the distributed simulation network, by simply copying the Jar containing the elements to that computer and invoking the Java Virtual Machine (JVM).

A Player in the GDSA Framework may be either a software component of the simulation or a software intermediary to a hardware component in the simulation. For example it might represent a whole physical entity such as an aircraft, an aggregation of physical entities such as a battalion, or a subset of a physical entity such as a missile guidance system that resides in a particular aircraft. On the other hand, a Player may be the mediator between the rest of the simulation and a physical item of hardware, which in turn may be operated and viewed by a human. This mediator handles all the simulation specific tasks such as communication and coordination and translates appropriately for the attached hardware. The GDSA requires the software Player to be serialized and reside in a Jar with all its required classes. The Player may communicate with other Players over the network via RMI or Sockets or by any other means the developer chooses; the GDSA Framework imposes no Player-to-Player communication requirements or restrictions. For applications, such as real-time simulations where speed of communication is important, communication via Sockets may be the preferable means of Player-to-Player communication as explored in Appendix A.

The prototype GDSA Framework consists of three fundamental software modules that allow a user to execute a set of Players in the GDSA environment:

- **PlayerServer:** The PlayerServer accepts Jars transmitted to it over the network, attaches a SlaveRTA to the Player in the Jar, and spawns the SlaveRTA, and thus the Player to which the SlaveRTA is attached, as a separate process. It provides a unique port for each Player through the SlaveRTA and supplies the host name and port number of the MasterRTA to the SlaveRTA. It uses RMI to communicate with the MasterRTA.

- **MasterRTA:** The MasterRTA is the Master Run Time Adapter (RTA) that contacts each PlayerServer, transmits the Jar containing a Player and its classes to the PlayerServer, and coordinates the simulation. Once all SlaveRTAs have established a link back to the MasterRTA and indicated readiness to proceed, it starts the simulation. The MasterRTA then steps each Player through initialization phases 0 - 1, directs Players to connect to each other (The MasterRTA provides port and host of each other Player.), and then steps each Player through initialization phases 1-2. The MasterRTA will not allow any Player to continue to the next initialization phase or connection step until all the Players have completed the current initialization phase; in this way, the GDSA Framework guarantees that all Players are synchronized during initialization. After the initialization stage, the MasterRTA allows the collection of Players to run, and it stays out of the way so as not to place a burden on the simulation. Finally, the MasterRTA can cause each Player to terminate which in turn results in the termination of the simulation. As mentioned above, it uses RMI to communicate with the PlayerServers. It also uses RMI to communicate with SlaveRTAs.
**SlaveRTA:** The SlaveRTA is a Slave Run Time Adapter that attaches to a single Player; it attaches to that Player by extracting the serialized Player from the Player's Jar that has been written to the local file space by the PlayerServer and instantiating it. The SlaveRTA then passes essential distributed simulation information to its Player which, at this time, consists just of the unique port number assigned to the Player by the PlayerServer. Once the attached Player is successfully executing, the SlaveRTA contacts the MasterRTA and indicates that it is ready to proceed. Under the control of the MasterRTA, the SlaveRTA passes initialization, connect, and run directives to its Player. Once the simulation begins, the SlaveRTA just stands by for a terminate command from MasterRTA; at that point it handles the termination of its attached Player. As mentioned above, the SlaveRTA communicates with the MasterRTA via RMI.

The steps in using the GDSA Framework to distribute a Player and the ultimate execution are depicted in Figure 2. Note the interplay of the three modules described above.
Figure 3 exhibits an Event Trace that shows the distribution, initialization, execution, and termination steps in more detail.

With this Framework, a simulation’s Players can be distributed over a heterogeneous collection of computers with no human intervention except to initially specify the computer on which each Player is to execute. The Framework only requires a PlayerServer running as a daemon on each computer to be used in the simulation; it and the MasterRTA cooperate in the distribution and execution of Players. With some modification, the Framework could even, optionally, assign each Player to a computer and thus avoid even that human intervention. While the control over the distributed simulation Players by the MasterRTA is rudimentary in the prototype Framework, its control and coordination functions could easily be enhanced.
MANAGEMENT OF LOCAL RESOURCES

The PlayerServer is minimally localized for the computer on which it runs. That is, it is given access to the local resources on the computer on which it executes; it is not localized in the sense that it is compiled for the particular computer on which it runs since it is written in Java and is thus completely portable code. It accomplishes this simple localization with two files that provide information specific to its computer. The files are reread between each invocation of the PlayerServer so that localization parameters may be changed between simulation runs if desired or necessary.

Paths and Jars

The PlayerServer requires a path to the Java Virtual Machine and the appropriate Java classes Jar in order to spawn Player processes on a given computer. While some platforms allow these paths to be specified as universal variables, not all do. Further, a given computer may have more than one JVM installed and the PlayerServer should be directed to the desired one. These paths are specified in the file, psprefs.txt. Also specified here are the local Jars that the PlayerServer is to make available, via the classpath keyword, to any Player that it executes and the Jars necessary for the SlaveRTA.

Scripts

psprefs.txt also specifies the type of scripts permissible on the computer on which the particular PlayerServer is executing. In some cases, the PlayerServer cannot spawn the SlaveRTA directly as a process, but must use the local scripting mechanism to accomplish that task.

Player Windows

In many cases, Players may have graphical capabilities. They may display state or other data in one or more windows. For PCs and Macintoshes, this does not typically present a problem, but on Unix computers this can be problematic; Unix operating systems will typically not allow an application to display a window until a window manager is running which would necessitate a user logged-on at the console. psprefs.txt has an entry that indicates whether or not a Player executing on the particular computer may put up its own window; the PlayerServer ultimately provides this information to the Player as a System Property.

Ports

Any computer with an IP address has 65535 logical ports numbered from 1 to 65535. Ports from 1 to 1023 are commonly allocated for system services, especially under Unix where root privileges are required to use them. For example, HTTP Services such as a Web server usually listens at port 80 while email services typically use port 25. As a result, the set of ports available to a Player is heavily dependent on the particular computer on which it executes. Moreover, because a port may be used by only one process at a time, the allocation of ports among the Players running in the GDSA must be carefully allotted.
A file, `portblocks.txt`, is provided to the PlayerServer to inform it of a computer's available ports and the blocks in which they are to be allocated. For example, a `portblocks.txt` file might resemble:

```
PortBlock: 5000 3
PortBlock: 5003 2
PortBlock: 5008 8
PortBlock: 5014 5
```

**Listing 1.** An example `portblocks.txt` file.

This example `portblocks.txt` file is depicted in Figure 4. There, by way of example, ports 5012 and 5013, though normally available for use by Players are temporally unavailable perhaps because of transient software running outside the scope of the GDSA.

```
Figure 4. PortBlocks.
```

The PlayerServer keeps the allocated PortBlocks as reusable resources. As shown in the Event Trace of Figure 5, PortBlocks are checked out to a particular simulation run as they are needed. Before any PortBlock is checked out, the ports it contains are each checked for accessibility. If any port is found to be unavailable at that time, as for example ports 5012 and 5013 noted in Figure 4, they are not made available to the current run by the PortBlock and a Player is offered the next available port instead. As a result, it is highly unlikely that any Player would be given a *bad* port at which to listen.
Using the PortBlock allocation scheme, it is possible for a PlayerServer to manage Players from two or more concurrent but independent simulations running on the same computer. Figure 6 shows how two Master Run Time Adapters may access the same PlayerServer and set up their respective Players to run on the same computer completely independent of each other. Because PortBlocks are checked out as needed, each Player is guaranteed its own unique port at which to listen. If all of a particular PlayerServer's available PortBlocks are in use, a Simulation is forced to wait until resources become available before it may proceed. This port allocation shown in Figure 6 may be generalized for any number of concurrent, independent simulations.
FUTURE ENHANCEMENTS

The GDSA Framework described here is simply a prototype. In working with the prototype, it was found that the proposed Architecture is satisfactory for both real-time and faster-than-real-time simulations. While the prototype Framework is completely usable, it lacks many desirable features that should be incorporated in the reference Framework that anyone might use to execute a distributed simulation. These future improvements leading to the reference Framework include an enhanced distributed simulation engine, improved security, incorporation of legacy models in the form of native code, analysis and restart capabilities, and synchronization options.

Distributed Simulation Engine

The current use of Run Time Adapters that are either Master or Slave is too restrictive. An RTA should not impose a particular initialization scheme on any given Player. Some sets of Players may require fewer or more initialization phases than those anticipated in the prototype MasterRTA itinerary, and while connections between Players are made after initialization Phase 1 in the prototype, other simulations may require the Player-to-Player connections to be initiated at a different stage.

To remove this restriction, it is proposed that the MasterRTA and SlaveRTA set be replaced with a single, simple Run Time Adapter that is neither Master nor Slave and simply executes the serialized Player that is given to it by the PlayerServer; this simple RTA does not communicate with any other RTA. The functions of the MasterRTA and SlaveRTA combination are replaced by a distributed Simulation Engine as depicted in Figure 7.

![Figure 7. Elements of a Composed, Distributed Simulation.](image-url)
This distributed Simulation Engine is a reusable component that may be replaced in any given Player under a Composition Environment which is described elsewhere. As such, a set of Players may simply be composed with the proper distributed Simulation Engine component and then be ready to participate in a given simulation. This distributed Simulation Engine component has the proper number of initialization steps, time of Player-to-Player connection and other specific simulation dependent requirements embedded in it and customizable at composition time. Thus, if a particular set of Players require 10 initialization steps, the appropriate distributed Simulation Engine is made a part of each Player and configured in the Composition Environment. This is in contrast with providing a new, customized GDSA Framework with a specialized MasterRTA and SlaveRTAs set that can accommodate the initialization requirements of any particular set of Players. With a distributed Simulation Engine, the GDSA Framework becomes very simple and can accommodate any special requirements because the accommodation of the special requirements resides in the replaceable distributed Simulation Engine component and is no longer part of the GDSA Framework itself. This moves simulation dependent tasks from the Framework, which is intended to be generic, to the Simulation Engine which is where such tasks properly belong.

Figure 8 shows, in a time wise fashion, how Players are distributed and connected in this proposed reference Framework. Note the use of simply a RunTimeAdapter rather than the MasterRTA/SlaveRTA set. Further note that the RTAs do not communicate with each other; instead the distributed Simulation Engine components communicate with each other to provide coordination and control of the simulation.

Figure 8. Generic Distributed Simulation Framework.
This use of a distributed Simulation Engine also gives Players more reusability. Players may play in a monolithic, distributed, or even a mixed environment simply by editing them in the Composition Environment, replacing the Simulation Engine component and the Communication component with the appropriate set and configuring that set. Conceivably, Players may be verified and validated independent of the particular Simulation Engine and Communication components used and thus remain verified and validated in any simulation environment.

Security

In future implementations, a higher degree of security should be implemented in the GDSA Framework. Since the PlayerServer will execute the serialized, conforming Player passed to it in a Jar, it can render the computer on which it executes particularly vulnerable to malicious attack. Yet, because the PlayerServer uses RMI exclusively, it already has some degree of protection from attack since any attacker would be forced to mimic the objects that the PlayerServer expects. However, in the near term, a Password authentication protocol can and should be added to the PlayerServer. If necessary, the Jar containing the Player could even be encrypted and signed before it is transmitted to the PlayerServer to protect its contents from interception.

Beyond the PlayerServer, it is doubtful that security measures would be needful for the Framework. The Run Time Adapters are not particularly vulnerable since they only establish one-to-one communication with each other via specified host name and port number and even that link is to be abandoned in the reference Framework. In the prototype, the RTA communication messages primarily consist of coordination commands.

For Players which are provided by the user and not part of the Framework, the developer may wish to implement a variety of security measures depending on the data being exchanged by the Players. Certainly it is not difficult to encrypt the Player-to-Player messages.

Legacy Models

The use of Native code in the GDSA needs exploration. Java provides a well-defined and easily implemented interface to native code called Java Native Interface (JNI). Essentially native code is contained in a native library which is directly referenced by the Java code; the JVM handles the library loading and unloading tasks for the user’s code. Accordingly, to make use of native libraries, a Player’s native library(s) may be included in its Jar. The PlayerServer need only have information on the native library’s location in the Jar and the name it must have when it is unpacked from the Jar and placed in the local file system; this information could be included as an addition to the Jar’s META-INF directory. Incorporation of legacy models in the GDSA Framework is then a straightforward task once the Jar-PlayerServer infrastructure is defined. The designer need only write a Java wrapper that passes information to and from the legacy model and communicates with the rest of the GDSA Players in the simulation’s prescribed manner.

Analysis and Restart

Often the point of running a simulation is to examine the results after it has executed. Accordingly, any simulation environment should have some mean of collecting the Player output so that analysis tools may present the results to the user. In a distributed simulation, this is a particularly pressing requirement because the output may be scattered over a variety of computers, and the user must be assured that the distributed data collected for a particular simulation run actually belongs to that run. At the current time, the prototype PlayerServer does incorporate a SlaveProcessWatcher which unobtrusively tracks the execution of any Player spawned as a separate process by the PlayerServer. Currently, the SlaveProcessWatcher simply waits for the watched process, the Player, to complete and then performs some simple clean-up chores. In the future, the
SlaveProcessWatcher could be tasked to collect all the Player’s output and move it to a specified, central Repository at the end of the simulation. With this additional feature, the user could then access a given simulation’s area of the specified Repository and be assured that the output there is a complete set of data from the particular simulation run.

Similar to the requirements to collect simulation data for analysis, is the ability to restart a simulation at some point in time or event space. In the past this has been accomplished by a variety of means notably by writing a local file that contains a snapshot of the Player’s state at any given time. With the Java-based GDSA, it is possible to go beyond this simple restart capability. The local Run Time Adaptor or Simulation Engine could be instructed to serialize the Player at any given time and write that Player to a local file which in turn would be stored in the specified Repository at the end of the simulation; this strategy makes use of the Persistence feature of Java. Any such saved, serialized Player could be instantiated and made to continue to run from the point at which it was saved. If a Composition Environment (See Reference) is available, a serialized Player can also be edited to change any state parameters desired for the restart.

Synchronization

Synchronization for time dependent simulations is critical to the proper, reliable performance of a simulation. If a set of computers on which a distributed simulation is executed are not properly synchronized, then otherwise identical time dependent runs will differ because causality is typically determined by a chain of events which may occur in a different order if the distributed Players are not properly synchronized. In real time simulations with humans in the simulation through an intermediary Player, faulty time synchronization can be frustrating and confound the objectives of the simulation. On the other hand, a properly designed discrete events simulation engine may not be sensitive to the time synchronization of a set of distributed computers; here the sequence of events which is tracked and disbursed by a central simulation engine determines causality in a reliable manner.

Achieving sufficient time synchronization of a set of computers can be difficult. In the past, many schemes have been proposed with varying degrees of merit. At this time, it appears that a Global Positioning Satellite receiver and card at each computer may be the most reliable and satisfactory means to synchronize all the computers to one clock. More investigation into this problem is merited.
CONCLUSIONS

The prototype GDSA Framework presented here demonstrates that the GDSA is not only feasible, but also flexible enough to host virtually any distributed simulation. Notably, the GDSA has the following attributes:

- The GDSA Framework is unobtrusive both to the simulation and to the computers on which it is installed. Only the PlayerServer element runs as a daemon on the computer and requires full time use of a port. The rest of the Framework is invoked and executed only as needed.

- The Run Time Adapters provide a high level of control over the dispersed elements of the simulation. Therefore is it is possible to globally cause data writes or serialized Player writes to the local file system enabling future restart and analysis capabilities.

- The simulation is automatically distributed by the Framework without human intervention. The user merely specifies the computer on which each Player is to run; even this sort of direction could be left to the Framework if the user wishes. This automatic dispersal of the simulation avoids human error and provides a high degree of confidence that the distributed simulation that is executing is the desired one.

The timing studies included in Appendix A indicate that the GDSA and the Java Virtual Machine are suitable for a real-time simulation environment. While the Message passing in Java is a bit slower than that in highly optimized C code, it appears that a Java implementation could closely approach C Socket Message passing speed if a parsing scheme is substituted for the Message Object scheme.

Pursuing the enhancement suggestions recommended here can greatly enhance the GDSA prototype Framework giving it better security, legacy model support, integration into a Composition Environment for eased and enhanced editing of a simulation, and the automatic collection of Player output in a centralized Repository for both analysis and restart capability.
APPENDIX A

A Timing Study of RMI and Socket based message passing.

There are two fundamental network communication modes resident in the Java language; these are Remote Method Invocation (RMI) and Sockets. A Socket is an old and common scheme that was popularized with the Berkeley Systems Design Unix systems. RMI is similar to the older Remote Procedure Call (RPC) but is unique to Java; currently, there is some work to make RMI more compatible with the language independent CORBA. From the inception of this study, and confirmed here, it appeared that RMI has considerably more overhead than Socket communication which can significantly slow down a distributed application. Accordingly, the small study presented here attempts to quantify the difference between RMI and Socket communication so that the developer may choose between them based on the message passing speed requirements of a given application.

The test is a simple one that consists of a Client and Server pair, each of which may be executing on the same or different computers connected via a network. The Server is started first and seizes a port at which it listens in the case of Socket communication or publishes an communication object in the case of RMI. The Client is then started given the port number and host of the Server. The Client establishes a communication link with the Server and then proceeds to send 10,000 messages to the Server with no pause between each message. The Server receives each message and immediately responds with a corresponding message of its own to the Client. In short, the Client sends 10,000 messages, the Server receives 10,000 messages and sends 10,000 messages, and the Client receives 10,000 messages; the time required for each task is noted. The RMI test is coded to be non-blocking; that is, when a message is received, it is put on a queue and handled by a separate Thread so that the calling Thread may immediately return from the method call; this is similar in behavior as Socket communication. The transactions for both RMI and Socket message passing are illustrated in the Event Trace of Figure A1.1.

![Event Trace of the Client–Server transactions.](image-url)
Rather than parsing text messages at the receiving end, messages are objects which are executed at the receiving end by the MessageHandler in order to obtain the data contained in the message. Figure A1.2 illustrates the hierarchy of Messages used in the timing study.

![Message hierarchy diagram]

**Figure A1.2.** Message hierarchy.

For the timing study only two message types are needed. One is the IDMessage which simply contains primitive identification information; it is used in the 10,000 message transactions. The DisconnectMessage terminates the test.

The Client tracks the amount of time required to send the 10,000 messages to the Server and the time to receive the 10,000 response messages from the Server. The Server tracks the time required to receive the 10,000 messages from the Client. There is an additional timing concern with Socket communication because the Message object is transmitted to the receiver via an ObjectOutputStream-ObjectInputStream pair. In current Java Virtual Machine implementations, the ObjectOutputStream-ObjectInputStream pair maintains a Hashtable of objects passed through the Stream. The use of such a Hashtable can make the application of an ObjectStream more efficient in that if a particular object is passed through the stream multiple times, only a reference to it need be passed after the first time rather than the actual object itself every time. However, because objects that have passed through the stream continue to be referenced in the Hashtable indefinitely, the Garbage Collector is unable to free their memory space even when they are no longer in use at either end of the Stream. Accordingly, a growing number of message transactions consumes more and more memory at both ends of the communication channel; this resembles a memory leak which should not occur in Java. To clear the Hashtable and thus enable the Garbage Collector to free memory for out of date and no longer referenced messages, the ObjectOutputStream has a reset() method. In the course of this study, it was discovered that excessive use of the reset() method can significantly degrade the overall speed of Socket communication, and that the degree of this slow down depends strongly on the computer pairs that are communicating. Accordingly, as part of the timing study, the message timing is correlated with the rate at which reset() is called.

The computers used in the timing study are listed in Table A1.1. The Reference name is the one used to refer to the computers in the subsequent timing plots. The items on the reference name row identify the hardware, Operating System, and Java version for that computer. Four of the five computers are Unix platforms and one is a Windows NT Notebook with a 3Com 10Base-T PCMCIA card. No attempt has been made to optimize network performance on the test computers because this test is designed primarily to quantitatively distinguish between RMI and Socket communication and to determine the impact of the rate at which reset() is called on Socket communication. However, the Client-Server software presented here may be employed to investigate the message passing characteristics in considerable detail.
A Macintosh 8500/233 running Macintosh Runtime Java (MRJ) 2.1ea3, which corresponds to Java Version 1.1.6, was initially included in the test, but network performance becomes unstable as the rate at which `reset()` is called increased; a bug report has been filed with Apple, but at the time of this writing, the problem has not been corrected.

The timing study results show that RMI communication is generally slower than Socket communication. For some platforms such as the machine referenced as “HP”, RMI is considerably slower than the best Socket communication timing; for the worst case presented here, RMI can be 20 times slower than the best Socket communication. For better cases such as the IBM, RMI is about 10 times slower than the best Socket communication.

The timing study also demonstrates that excessive use of the `ObjectOutputStream.reset()` method can make Socket communication markedly slower than RMI. It appears that, in general, a call to `reset()` once per 100 messages sent should have no noticeable impact on the speed of Socket communication for the computers tested; the only exception is a Client-Server pair both executing on the referenced Sun where any call to `reset()` appears to adversely impact the message passing speed. For a specific application, the developer must consider both the required speed of message passing and the memory-use implications of accumulating old, used messages and pick an appropriate rate at which to call `ObjectOutputStream.reset()`.

Below, six plots are presented depicting the results of the timing study. In each plot, the average time to pass one of the 10,000 messages is shown as a function of the rate at which `reset()` is called. This average time to pass a message is the average of the time interval over which the Client receives the Server’s 10,000 messages and the time interval over which the Server receives the Client’s 10,000 messages; these two intervals are typically very close. The time required to initially send the first 10,000 messages is not considered here; it is almost always smaller than the receiving time intervals, but it is thought that the receiving time better represents the message passing time in a real application environment. The RMI message-passing time is plotted as a horizontal, dashed line on each plot. It does not depend on the `reset()` call rate since `ObjectOutputStream.reset()` is never overtly called for RMI. The computer to which the Server is connected is shown at the right side of the plot or, sometimes, next to the line within the plot.

Figure A1.3 shows the average message passing time when the Client and Server both execute on the same computer. It is clear from this plot, that RMI can be markedly slower than Socket message passing. Figure A1.4 shows the message passing time for the case where the machine referenced as “IBM” serves as the Server; note that this configuration gives the lowest message passing rate of all the configurations. The remaining plots, Figures A1.5 through A1.8 show the cases where the Sun, the Alpha, the SGI, and the HP host the Server, respectively.
Figure A1.3. Time per message for Server and Client both on the same computer.

Figure A1.4. Time per message with IBM acting as Server.
Figure A1.5. Time per message with Sun acting as Server.

Figure A1.6. Time per message with Alpha acting as Server.
Figure A1.7. Time per message with SGI acting as Server.

Figure A1.8. Time per message with HP acting as Server.
The preceding plots could be redrawn to better display some data. When the ordinate axis uses a log scale, the individual computer message passing times are more distinct, but that is not the purpose of this study which is primarily to distinguish between the message passing time of Socket and RMI communication. Further, it should be noted that only TCP/IP rather than Datagram Sockets were tested in this study; it is uncertain whether Datagram Sockets could properly handle object passing for objects beyond a particular size, but the use of Datagram Sockets would be an interesting follow on study.

The message passing design presented here uses a Message object as described above to reduce the differences between RMI and Socket communication and thus provide a fair comparison of the two. The use of a Message object is attractive as it requires no parsing at the receiving end, but instantiation of a Message object and the reinstantiation of that object at the ObjectOutputStream end could be time consuming and add to the overall message passing time. In applications, where the fastest transmission of a large number of messages is vital, the developer may wish to design a specialized message passing scheme that transmits a message as a series of bytes through the Socket which must be parsed at the receiving end. This writer has had previous experience with such a scheme implemented in highly optimized C code and run on most of the computers used in the test presented here. The message passing times for a very similar test are close to those presented here for passing Message objects; abandoning the Message object for a stream of bytes that must be parsed might improve the message passing time by as much as a factor of 8, but a factor of about 1.5 appears to be more likely overall from a comparison to the old data. Since RMI is fundamentally an object passing architecture, it is unlikely to show any speed increase if any other scheme is substituted for the Message object.

The Client-Server source is provided below. Using this code, a developer may characterize the message passing characteristics of a particular collection of computers to be used in a distributed application. In fact, such a candidate collection of computers should be characterized using the code below before that configuration is to be used for serious distributed computing. With the data from such tests, the user has a good idea of how far the distributed application may be stressed on a given computer configuration, and thus the user has an indication of when the data provided by that application may be suspect because of late messages which are, in essence, late events.

The Client-Server sources below must be compiled and linked against the Java Generic Library (jgl3.1.0.jar); no version earlier than 3.1 may be used because the package name was changed with 3.1. Further, for the RMI tests, rmic must be run on the BootstrapRMI and RemoteHandlerImpl classes to create the proper Stubs and Skels; they may be placed in a jar called RepositoryStubsAndSkels.jar. To execute the Server the user specifies that the class TestServer contains the main method and provides the arguments:

4. “Socket” or “RMI” to select either version of the Server.
5. The port number on which the Server is to listen (Socket version) or publish its communication object (RMI version).
6. For the Socket version, the number of messages sent that trigger an ObjectOutputStream.reset() call.

For example, the C-shell script on the Sun might be:

```
java -classpath .:/ServerClasses.jar:/RepositoryStubsAndSkels.jar:/jgl3.1.0.jar:
/usr/local/jdk1.1.7A/lib/classes.zip TestServer Socket 5055 1
```
Running the Client is similar, the user specifies that the class `TestClient` contains the main, and provides the arguments:

3. “Socket” or “RMI” to select either version of the Client.
4. The port number at which the Server is listening.
5. The name of the host on which the Server is executing.
6. For the Socket version, the number of messages sent that trigger an `ObjectOutputStream.reset()` call.

For example, the C-shell script on the Sun might be:

```
java -classpath ./ClientClasses.jar:/RepositoryStubsAndSkels.jar:/jgl3.1.0.jar
/usr/local/jdk1.7A/lib/classes.zip TestClient Socket 5055 waggle.lanl.gov 1
```
/* TestServer.java */
import standinprefix.booker.simulation.remoterepository.*;

public class TestServer
{
    public static void main( String args[] )
    {
        TestServer server = new TestServer();
        int port = Integer.parseInt( args[1] );

        if ( args[0].equals( "RMI" ) ) server.listenRMI( port );
        else if ( args[0].equals( "Socket" ) )
        {
            int countToReset = Integer.parseInt( args[2] );

            server.listenSocket( port, countToReset );
        }
    }

    public void listenRMI( int port )
    {
        RMIServer server = new RMIServer( port );
    }

    public void listenSocket( int port, int countToReset )
    {
        SocketServer server = new SocketServer( port, countToReset );
    }
}

/* TestClient.java */
import standinprefix.booker.simulation.remoterepository.*;
import standinprefix.booker.simulation.remoterepository.message.*;
import java.net.*;
import java.rmi.*;

/**
 * A simple class to start up the Client and connect either via
 * RMI or Socket. The Client sends the same message 10,000 times to
 * the Server and prints out the time required to send those messages.
 * Each message requests a reply from the Server.
 *
 * @version 1.0 1998 April
 * @author C. P. Booker
 */
public class TestClient
{
    /**
     * Make the connection via RMI or Socket based on the command line
     * arguments.
     */
    public static void main( String args[] )
    {
        TestClient client = new TestClient();
        int port = Integer.parseInt( args[1] );
        if ( args[0].equals( "RMI" ) )
            client.connectRMI( port, args[2] );
        else if ( args[0].equals( "Socket" ) )
            client.connectSocket( port, args[2] );
    }
}
else if ( args[0].equals( "Socket" ) )
{
    int countToReset = Integer.parseInt( args[3] );
    client.connectSocket( port, args[2], countToReset );
}

/**
 * Connect via RMI.
 * @param port The port number for the Server.
 * @param candidate the Host name for the Server.
 */
public void connectRMI( int port, String candidate )
{
    if ( candidate != null )
    {
        String serverName = candidate;

        RemoteHandlerRMI client = null;

        // Create a MessageHandler that will receive messages.
        MessageHandler messageReceiver = new MessageHandler();

        // Create a RemoteHandler to interact with the Server.
        try
        {
            client = new RemoteHandlerRMI( null, messageReceiver, "Client" );
        } catch( Exception e )
        {
            System.out.println( "Cannot create Client's RemoteHandler, e = " + e );
        }

        // Attempt to connect to the server:
        Bootstrap server = null;
        try
        {
            server = (Bootstrap) Naming.lookup( "rmi://" + serverName + ":" + port + "/RepositoryServer" );
        } catch( Exception e )
        {
            System.out.println( "Error in connecting! e = " + e );
        }

        System.out.println( "Bootstrap = " + server );

        RemoteHandler serverHandler = null;
        if ( server != null && client != null )
        {
            try
            {
                serverHandler = (RemoteHandler) server.handShake( client, "Client" );
            } catch( Exception e )
            {
System.out.println( "Error in acquiring Server Handler, e = "+ e);
}
System.out.println( "Server Handler = " + serverHandler );
client.setRemoteHandler( serverHandler );
}

if ( serverHandler != null )
{
 // Send some messages!
 IDMessage idMessage = new IDMessage( "The Client*", true );
 DisconnectMessage disconnectMessage = new DisconnectMessage( "The Client", true );
 try{
 long beginClock = System.currentTimeMillis();
 for ( int i = 0; i<10000; i++ ) serverHandler.deliverMessage( idMessage );
 long finalClock = System.currentTimeMillis();
 System.out.println( "Time required to send 10,000 messages = " + 
 (finalClock - beginClock) + " ms " );
 serverHandler.deliverMessage( disconnectMessage );
} catch( Exception e )
{
 System.out.println( "TestRMIclient: Unable to send Client's message, e = "+ e );
 }
}
else System.out.println( "Error no Server named." );

/**
 * Connect via Socket.
 * @param port The port number for the Server.
 * @param candidate the Host name for the Server.
 * @param countToReset
 */
public void connectSocket( int port, String candidate, int countToReset )
{
 String host;
 Socket clientSocket;

 if ( candidate != null )
{
 String serverName = candidate;
 RemoteHandlerImpl client = null;

 // Create a MessageHandler that will receive messages.
 MessageHandler messageReceiver = new MessageHandler();

 try
 {
host = InetAddress.getByName( serverName ).getHostAddress;
) catch( Exception e )
{
    System.out.println( "Unable to open host = " + serverName );
    return;
}

// Open a socket to the host.
try {
    clientSocket = new Socket( host, port );
    System.out.println( "Client: connected to " + serverName + " (" + host + ")" );
} catch( Exception e )
{
    System.out.println( "Unable to open socket to " + serverName + " (" + host + ")" );
    return;
}

// Create a RemoteHandler to interact with the Server.
try {
    client = new RemoteHandlerSocket( clientSocket, messageReceiver, false, countToReset );
} catch( Exception e )
{
    System.out.println( "Cannot create Client's RemoteHandler, e = " + e );
}

// Send some messages!
IDMessage idMessage = new IDMessage( "The Client*", true );
DisconnectMessage disconnectMessage = new DisconnectMessage( "The Client", true );
try {
    long beginClock = System.currentTimeMillis();
    for ( int i = 0; i<10000; i++ )
    {
        IDMessage newMessage = new IDMessage( "The Client*", true );
        client.sendToRemote( newMessage );
    }
    client.sendToRemote( idMessage );
    long finalClock = System.currentTimeMillis();
    System.out.println( "Time required to send 10,000 messages = " + (finalClock - beginClock) + " ms " );
    messageReceiver.setClocko;
    client.sendToRemote( disconnectMessage );
} catch( Exception e )
{
    System.out.println( "Unable to send Client's message, e = " + e );
}
else System.out.println("Error no Server named.");
}
}

/* BootStrap.java */
package standinprefix.booker.simulation.remoterepository;

/**
 * BootStrap is the primary Server Interface which is exported, via RMI, to the world.
 * @version 1.0 1998 March
 * @author C. P. Booker
 */
public interface BootStrap extends java.rmi.Remote
{

/**
 * Get a connection to the server. The interface RemoteHandler is returned to
 * the Client so that the Client may send Messages to the Server.
 * @param serverHandler The exported interface to the Client.
 * @param clientHost A String identifying the Client.
 * @return A handle to the client-handling Thread resident in the Server.
 * If null,
 * then a Thread was not available to service the Client. RemoteHandler is an Interface.
 */
public RemoteHandler handShake( RemoteHandler serverHandler, String clientHost )
 throws java.rmi.RemoteException;
}

/* BootStrapRMI.java */
package standinprefix.booker.simulation.remoterepository;

import java.rmi.*;
import java.rmi.server.*;

/**
 */
public class BootStrapRMI extends UnicastRemoteObject implements BootStrap
{

/**
 * Create an instance of the BootStrapImpl. The constructor calls super() to explicitly export this remote object through the UnicastRemoteObject constructor.
 */
public BootStrapRMI() throws RemoteException
{ super(); }

/**
 * Get a connection to the server. The interface ClientHandler is returned to

public RemoteHandler handShake( RemoteHandler serverHandler, String remoteHost )
     throws RemoteException
     {
         RemoteHandler remoteHandler = null;

         // Create a Thread to service the Client:
         remoteHandler = new RemoteHandlerRMI( serverHandler, new MessageHandler(),
                                      remoteHost );

         return remoteHandler;
     }

/* BootStrapSocket .java */
package standinprefix.booker.simulation.remoterepository;

import java.io.*;
import java.net.*;
import standinprefix.booker.simulation.remoterepository.message.*;

public class BootStrapSocket extends Thread
     {
     protected int            port    = 0;
     protected boolean        runServer = true;
     protected ServerSocket   server   = null;
     protected MessageHandler handler = null;
     protected int            countToReset = 1;

     public BootStrapSocket( int port, int countToReset )
     {
         this.port    = port;
         this.countToReset = countToReset;
         try
         {
             server = new ServerSocket( port );
         } catch( Exception e )
         {
             System.out.println( "Error: cannot open ServerSocket, e = "+ e );
         }
         this.handler = handler;

         start();
     }
public void run()
{
    Socket client = null;

    while( runServer )
    {
        System.out.println( "Server: waiting for next connection" );
        Thread tempThread = Thread.currentThread();
        System.out.println( " Server Thread = " + tempThread );
        try
        {
            client = server.accept();
        } catch( IOException e )
        {
            System.out.println( "Server error: e = " + e );
            continue;
        }
        System.out.println( "Accepting new connection to " + client );
        MessageHandler handler = new MessageHandler();
        try
        {
            RemoteHandlerSocket subServer = new RemoteHandlerSocket( client,
                    handler, true,
                            countToReset
                    );
        } catch( Exception e ) { System.out.println( "BootStrapSocket: e = " +
                    e ); } }
    }

    /* MessageHandler.java */
    package standinprefix.booker.simulation.remoterepository;

    import java.util.*;
    import standinprefix.booker.simulation.remoterepository.message.*;

    /**
     * MessageHandler receives messages, through the Message Receiver Interface,
     * and acts on them.
     * @version 1.0. 1998 March
     * @author C. P. Booker
     */
    public class MessageHandler implements MessageReceiver
    {
        protected RemoteHandlerSender localToRemote = null;
        protected String remoteName = null;
        protected long clockTime = 0; // For timing -- remove in real thing!
        protected boolean idOther = false;
        protected long beginTime = 0;
        protected int numberOfIDs = 0;
/**
 * Create a MessageHandler with no handle to a RemoteHandler.
 */
public MessageHandler() {}

/**
 * A Temporary method to allow timing studies.
 */
public void setClock()
{ clockTime = System.currentTimeMillis(); }

// MessageReceiver Methods:
/**
 * Set the object that will send messages to the remote counterpart.
 * @param localToRemote RemoteHandlerImpl that delivers messages to the remote.
 */
public void setRemoteSender( RemoteHandlerSender localToRemote )
{ this.localToRemote = localToRemote; }

/**
 * Receive ID message from the Remote.
 * @param sender The handle to the Remote's RemoteHandler.
 * @param senderName The String containing the name of the Remote.
 * @param replyRequested If true, send a like ID Message to the sender.
 */
public void idSender( RemoteHandler sender, String senderName, boolean replyRequested )
{ if ( !idOther )
    { beginTime = System.currentTimeMillis();
      idOther = true;
    }
    ++numberOfIDs = 10000
    { long finalClock = System.currentTimeMillis();
      System.out.println( "******* Time to receive 10000 messages = " +
                      (finalClock - beginTime ) + " ms ");
      remoteName = senderName;
      if ( replyRequested )
      { IDMessage idMessage = new IDMessage( "The Receiver", false );
        try
        { localToRemote.sendToRemote( idMessage );
        } catch( Exception e )
        { System.out.println( "MessageHandler: Unable to send Receiver's message, e = " + e );
        }
    }
public void disconnect(RemoteHandlerImpl receiver, boolean handshaking)
{
    long currentClock = System.currentTimeMillis();
    System.out.println("Received disconnect message from " + remoteName);
    if (clockTime > 0)
        System.out.println("******* Timing: disconnect round trip = " +
                        (currentClock - clockTime) + " ms");
    if (handshaking)
    {
        try
        {
            localToRemote.sendToRemote(new DisconnectMessage("Terminal", false));
        } catch(Exception e)
        {
            System.out.println("Problem: cannot send disconnect message back to sender.");
        }
    }
    receiver.disconnect();
}

/* MessageReceiver.java */
package standinprefix.booker.simulation.remoterepository;
import java.util.*;
import standinprefix.booker.simulation.remoterepository.message.*;

public interface MessageReceiver
{
    /**
    * MessageReceiver is an Interface that receives Messages.
    *
    * @version 1.0 1998 March
    * @author C. P. Booker.
    * @see standinprefix.booker.simulation.remoterepository.Message
    */

    /**
    * Set the interface for sending messages to the remote.
    * @param local The RemoteHandlerSender that will ship messages to the remote.
    */
public void setRemoteSender(RemoteHandlerSender local);

/**
 * Receive the IDMessage.
 * @param sender The handle to the RemoteHandler at the sender.
 * @param receiver The handle the RemoteHandler that received the message.
 * @param senderName The String containing the id of the sender.
 * @param replyRequested A boolean, if true, an ID message is to be returned to the sender
 * to complete the handshake.
 */
public void idSender(RemoteHandler receiver, String senderName, boolean replyRequested);

/**
 * Receive the DisconnectMessage.
 * @param receiver The handle the RemoteHandler that received the message.
 * @param senderName The String containing the id of the sender.
 * @param handshaking A boolean, if true, a Disconnect Message is to be returned to the sender
 * to complete the disconnect.
 */
public void disconnect(RemoteHandlerImpl receiver, boolean handshaking);

package standinprefix.booker.simulation.remoterepository;
import standinprefix.booker.simulation.remoterepository.message.*;

/**
 * RemoteHandler is the Interface of any implementation on the Client or Server side
 * that receives messages from a Remote counterpart.
 *
 * @version 1.0 1998 March.
 * @author C. P. Booker
 */
public interface RemoteHandler extends java.rmi.Remote {

/**
 * Deliver a message to the Receiver.
 * @param message A Message sent to the Receiver.
 * @return true if the Receiver is active and received the message.
 * @see standinprefix.booker.simulation.remoterepository.message.Message
 */
public boolean deliverMessage(Message message) throws java.rmi.RemoteException;

/**
 * Test connection.
 * @return true if the remote connection is still available and active.
 */
public boolean isAlive() throws java.rmi.RemoteException;
}

/** RemoteHandlerImpl.java */
package standinprefix.booker.simulation.remoterepository;

import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
import com.objectspace.jgl.*;
import standinprefix.booker.simulation.remoterepository.message.*;

/**
 * RemoteHandlerImpl is the Implementation of the RemoteHandler.
 * @version 1.0 1998 March.
 * @author C. P. Booker
 */
public class RemoteHandlerImpl extends UnicastRemoteObject implements RemoteHandler, RemoteHandlerSender, Runnable
{
    protected boolean active = true;
    protected MessageReceiver messageReceiver = null;
    protected String remoteHost = null;
    protected Thread thread = null;

    /**
     * Construct the ClientHandlerImpl
     * @param client The ServerHandler that is to receive a Messages.
     * @param messageHandler The MessageReceiver that is to receive a Message.
     */
    public RemoteHandlerImpl( MessageReceiver messageReceiver, String remoteHost )
        throws RemoteException
    {
        super();
        if ( messageReceiver != null ) messageReceiver.setRemoteSender( this );
        this.messageReceiver = messageReceiver;
    }

    /**
     * Determine if specified.
     * @return true if the specified host is the one to which this RemoteHandlerImpl
     * is connected.
     */
    public boolean isHost( String host )
    {
        if ( !active ) return false;
        return true;
    }

    // Other methods and implementations
}
/**
 * Get the host to which this RemoteHandlerImpl is connected.
 * @return the host on the other end of the connection. null is
 * returned if the connection is no longer active.
 */
public String getHost()
{
    if ( !active ) return null;
    return remoteHost;
}

/**
 * Send a message to the counterpart on the remote machine.
 * @param message The Message to send to the remote machine.
 */
public boolean sendToRemote( Message message ) throws java.rmi.RemoteException
{
    return false;
}

/**
 * Handle a check on whether or not the connection is alive.
 * @return true if the connection is alive.
 */
public boolean remoteAlive()
{
    if ( !active ) return false;
    return true;
}

/**
 * Test connection.
 * @return true if the remote connection is still available and active.
 */
public boolean isAlive() throws java.rmi.RemoteException
{
    if ( active ) return true;
    else return false;
}

/**
 * Deliver a message to the Client.
 * @param message A Message sent to the Client.
 * @return true if the Client is active and received the message.
 * @see standinprefix.booker.simulation.remoterepository.Message
 */
public boolean deliverMessage( Message message ) throws java.rmi.RemoteException
{
    return false;
}

/**
 * Disconnect from the remote host
 */
public void disconnect()  
{  active = false;  }

/**  
* Get the MessageReceiver associated with this RemoteHandler.  
* @return The MessageReceiver associated with this RemoteHandler.  
*/
public MessageReceiver getMessageReceiver() { return messageReceiver; }

public void run()  
{  thread = Thread.currentThread();  
   System.out.println(">>> " + this + ": current Thread = " + thread );  
   System.out.println("******* Terminating connection with " + remoteHost +  
            " *******");  
}

/* RemoteHandlerRMI.java */
package standinprefix.booker.simulation.remoterepository;

import java.rmi.*;
import java.rmi.server. *;
import java.util.*;
import com.objectspace.jgl.*;
import standinprefix.booker.simulation.remoterepository.message.*;

/**  
* RemoteHandlerRMI is the Implementation of the RemoteHandler.  
*  
* @version 1.0 1998 March.  
* @author C. P. Booker  
*/
public class RemoteHandlerRMI extends RemoteHandlerImpl  
{  protected Queue pendingMessages = new Queue();
   protected RemoteHandler remote = null;
   protected long sleepTime = 600000;

   /**  
   * Construct the ClientHandlerImpl  
   * @param client The ServerHandler that is to receive a Messages.  
   * @param messageHandler The MessageReceiver that is to receive a Message.  
   */
   public RemoteHandlerRMI( RemoteHandler remote, MessageReceiver messageReceiver,  
                     String remoteHost ) throws RemoteException  
   {  super( messageReceiver, remoteHost );  
      setRemoteHandler( remote );  
      Thread tempThread = new Thread( this );  
}
tempThread.start();

tempThread.setPriority( Thread.NORM_PRIORITY -1 );

/**
 * Determine if the host to which this handler is connected matches that
 * specified.
 * @return true if the specified host is the one to which this
 * RemoteHandlerImpl
 * is connected.
 */
public boolean isHost( String host )
{
    if ( !active || remoteHost == null ) return false;
    if ( remoteHost.equals( host ) ) return true;
    return false;
}

/**
 * Get the host to which this RemoteHandlerImpl is connected.
 * @return the host on the other end of the connection. null is
 * returned if the connection is no longer active.
 */
public String getHost()
{
    if ( !active ) return null;
    return remoteHost;
}

/**
 * Send a message to the counterpart on the remote machine.
 * @param message The Message to send to the remote machine.
 */
public boolean sendToRemote( Message message ) throws java.rmi.RemoteException
{
    if ( !active ) return false;
    try
    {
        remote.deliverMessage( message );
    } catch( java.rmi.RemoteException e )
    {
        // Test to see if connection is active:
        try
        {
            if ( !remote.isAlive() )
            {
                active = false;
                synchronized( thread ) { thread.notify(); } // thread.notify();
                throw e;
            }
        } catch( java.rmi.RemoteException ex )
        {
        }
    }
}
active = false;
synchronized( thread ) { thread.notify(); }
throw ex;
}
return true;
}

/**
 * Handle a check on whether or not the connection is alive.
 * @return true if the connection is alive.
 */
public boolean remoteAlive()
{
    if ( !active ) return false;
    try
    {
        if ( remote.isAlive() ) return true;
        else
        {
            active = false;
            synchronized( thread ) { thread.notify(); }
            return false;
        }
    }
    catch( java.rmi.RemoteException e )
    {
        active = false;
        synchronized( thread ) { thread.notify(); }
        return false;
    }
}

/**
 * Test connection.
 * @return true if the remote connection is still available and active.
 */
public boolean isAlive()
{
    if ( active ) return true;
    else return false;
}

/**
 * Set the RemoteHandler to which this RemoteHandlerImpl may speak.
 * @param remoteHandler The RemoteHandler to which messages may be
 * transmitted.
 */
public void setRemoteHandler( RemoteHandler remote )
{
    this.remote = remote;
}

/**
 * Deliver a message to the Client.
 * @param message A Message sent to the Client.
 * @return true if the Client is active and received the message.
 */
* @see standinprefix.booker.simulation.remoterepository.Message
*/
public boolean deliverMessage( Message message ) throws java.rmi.RemoteException
{
    if ( !active ) return false;
    pendingMessages.push( message );
    if ( thread != null )
        synchronized( thread )
        { thread.notify(); }
    return true;
}

/**
 * Disconnect from the remote host
 */
public void disconnect()
{
    super.disconnect();
    if ( thread != null ) synchronized( thread ) { thread.notify(); }
}

/**
 * Get the MessageReceiver associated with this RemoteHandle.
 * @return The MessageReceiver associated with this RemoteHandle.
 */
public MessageReceiver getMessageReceiver() { return messageReceiver; }

public void run()
{
    thread = Thread.currentThread();
    System.out.println( ">>> " + this + ": current Thread = " + thread );
    while( active )
    {
        if ( pendingMessages.isEmpty() )
        {
            try
            {
                synchronized (thread) { thread.wait( sleepTime ); }
            } catch( Exception e )
            {
                System.out.println( "wait interrupted e = " + e ); } // No messages to process.
        }
        if ( !active ) break;
        try
        {
            Message newMessage = (Message) pendingMessages.pop();
            newMessage.execute( messageReceiver, this );
        } catch( InvalidOperation e )
        {
            try
            {
            } catch( e )
            {
            }
        }
}
if (! (remote.isAlive()) )
{
    active = false;
    break;
}
} catch( java.rmi.RemoteException ex )
{ // Connection is dead.
    active = false;
    break;
}
}

thread.yield();
}
System.out.println( "******* Terminating connection with " + remoteHost + " *******");
}

/* RemoteHandlerSocket.java */
package standinprefix.booker.simulation.remoterepository;

import java.io.*;
import java.net.*;
import java.util.*;
import com.objectspace.jgl.*;
import standinprefix.booker.simulation.remoterepository.message.*;

/**
* RemoteHandlerSocket is the Implementation of the RemoteHandler.
* @version 1.0 1998 March.
* @author C. P. Booker
*/
public class RemoteHandlerSocket extends RemoteHandlerImpl
{
    protected Socket remote = null;
    protected ObjectInputStream in = null;
    protected ObjectOutputStream out = null;
    protected boolean serverMode = false;

    // Items to count messages sent:
    protected int countToReset = 100000; // Number of msgs sent to force reset of
    protected int sentCount = 0;
    protected int totalSent = 0;

    /**
    * Construct the ClientHandlerImpl
    * @param client The ServerHandler that is to receive a Messages.
    * @param messageHandler The MessageReceiver that is to receive a Message.
    */
public RemoteHandlerSocket( Socket remote, MessageReceiver messageReceiver, boolean serverMode, int countToReset ) throws Exception
{
    super( messageReceiver, null );
    this.countToReset = countToReset;
    setRemoteHandler( remote );
    System.out.println( "Connecting to new client, socket = " + remote );
    InetAddress iA = remote.getInetAddress();
    remoteHost = iA.getHostName();
    System.out.println( "connecting to name = " + remoteHost );
    System.out.println( " ObjectOutputStream.reset () rate = 1/" +
    countToReset );

    // Establish an input stream:
    try
    {
        if ( serverMode )
        {
            in = new ObjectInputStream( remote.getInputStream() );
            System.out.println( "Established input stream = " + in );
            out = new ObjectOutputStream( remote.getOutputStream() );
            System.out.println( "Established output stream = " + out );
        }
        else
        {
            out = new ObjectOutputStream( remote.getOutputStream() );
            System.out.println( ">>>>> Established output stream = " + out );
            in = new ObjectInputStream( remote.getInputStream() );
            System.out.println( ">>>>> Established input stream = " + in );
        }
    } catch ( Exception e )
    {
        System.out.println( "Error in establish ObjectInputStream, e = " + e );
    } 

    Thread tempThread = new Thread( this );
tempThread.start();

tempThread.setPriority( Thread.NORM_PRIORITY -1 );
}

/**
* Send a message to the counterpart on the remote machine.
* @param message The Message to send to the remote machine.
* /
public boolean sendToRemote( Message message )
{
    try
    {
        totalSent++;
        out.writeObject( message );
        out.flush();
    }
if ( ++sentCount >= countToReset )
{
    sentCount = 0;
    out.reset();
}
} catch( Exception e )
{
    System.out.println( "RemoteHandlerImpl error: unable to send Message,"
            + message + " to " + remote );
    System.out.println( " e = " + e );
    System.out.println( " total messages sent = " + totalSent );
    System.out.println( " remote = " + remote );
    //reestablishConnection();
}
return true;
}

protected void reestablishConnection() {
    try {
        Socket tempRemote = new Socket( remote.getInetAddress(),
        remote.getPort() );
        remote = tempRemote;
        System.out.println( "Reset Connection: remote = " + remote );
        if ( serverMode )
        {  
in = new ObjectInputStream( remote.getInputStream() );
        System.out.println( "Established input stream = " + in );
        out = new ObjectOutputStream( remote.getOutputStream() );
        System.out.println( "Established output stream = " + out );
        }
        else
        {  
            out = new ObjectOutputStream( remote.getOutputStream() );
            System.out.println( ">>>>> Established output stream = " + out );
            in = new ObjectInputStream( remote.getInputStream() );
            System.out.println( ">>>>> Established input stream = " + in );
        }
    } catch ( Exception e )
    {
        System.out.println( "Error in establish ObjectInputStream, e = " + e ");
    }
    /*
    try
    {
        out = new ObjectOutputStream( remote.getOutputStream() );
    } catch( Exception e )
    {
        System.out.println( "Error reestablishConnection() : e = " + e );
    }
    */
protected void resetOutputStream()
{
    try
    {
        out.reset();
    } catch( Exception e )
    {
        System.out.println( "RemoteHandlerSocket.resetOutputStream, e = " + e);
    }
}

/**
 * Set the RemoteHandler to which this RemoteHandlerImpl may speak.
 * @param remoteHandler The RemoteHandler to which messages may be transmitted.
 */
public void setRemoteHandler( Socket remote )
{ this.remote = remote; }

/**
 * Disconnect from the remote host
 */
public void disconnect()
{
    super.disconnect();
    if ( thread != null ) thread.resume();
}

public void run()
{
    Message currentTransaction = null;
    System.out.println( "RemoteHandlerImpl running, input stream = " + in );
    while( active )
    {
        try
        {
            Object newObject = in.readObject();
            currentTransaction = (Message) newObject;
        } catch( Exception e )
        {
            System.out.println( "Error receiving message, e = " + e );
            return;
        }
        if ( currentTransaction != null )
        {
            currentTransaction.execute( messageReceiver, this );
        }
        else

    }
try {
    System.out.println("Closing connection to "+ remote);
    in.close();
} catch (Exception e) {
    System.out.println("Unable to close connection e = "+ e);
}

} // End: while (active)

} // End: class RemoteHandlerSender

/* RemoteHandlerSender.java */
package standinprefix.booker.simulation.remoterepository;

import standinprefix.booker.simulation.remoterepository.message.*;

/**
 * RemoteHandlerSender provides an interface to the RemoteHandler to
 * take "send" tasks.
 *
 * @version 1.0 1998 March
 * @author C. P. Booker
 */
public interface RemoteHandlerSender {
    /**
     * Send a message to the Remote.
     * @param message The message to be sent.
     * @return true if successful.
     */
    public boolean sendToRemote(Message message) throws java.rmi.RemoteException;

    /**
     * Check on the status of the Remote.
     * @return true if Remote is still connected and active.
     */
    public boolean remoteAlive();
}

/* RMIServer.java */
package standinprefix.booker.simulation.remoterepository;

import java.rmi.*;
import java.rmi.registry.*;

public class RMIServer {
    public RMIServer(int port) {
    }
Registry registry = null;

try {
    System.setSecurityManager( new RMISecurityManager() );
    System.out.println( "Initializing Repository Server" );
    BootStrapRMI bootStrap = new BootStrapRMI();
    registry = LocateRegistry.createRegistry( port );
    registry.rebind( "RepositoryServer", bootStrap );
    System.out.println( "Repository Server is ready at port #" + port );
} catch( Exception e ) {
    System.out.println( "Exception occured: " + e );
    e.printStackTrace();
}

/* SocketServer.java */
package standinprefix.booker.simulation.remoterepository;

public class SocketServer {
    public SocketServer( int port, int countToReset ) {
        try {
            System.out.println( "Initializing Socket Server" );
            BootStrapSocket bootStrap = new BootStrapSocket( port, countToReset );
            System.out.println( "Repository Server is ready at port #" + port );
        } catch( Exception e ) {
            System.out.println( "Exception occured: " + e );
            e.printStackTrace();
        }
    }
}

/* Message.java */
package standinprefix.booker.simulation.remoterepository.message;

import standinprefix.booker.simulation.remoterepository.*;

/**
 * Message is the parent class of messages exchanged between the Server
 * and Client. Message is an abstract class that must be extended to use.
 */
public abstract class Message implements java.io.Serializable {
    /**
     * Cause the Message to 'deliver' the message by providing it with an interface
     * constructed to receive the message data.
     * @param receiver The MessageReceiver interface that will receive the message.
     * @see standinprefix.booker.simulation.remoterepository.MessageReceiver
     */
    public abstract void execute( MessageReceiver messageReceiver, RemoteHandlerImpl receiver );
}

package standinprefix.booker.simulation.remoterepository.message;

import standinprefix.booker.simulation.remoterepository.*;

public class IDMessage extends Message {
    protected String senderName;
    protected boolean reply = true;

    /**
     * Construct an IDMessage.
     * @param senderName The name of the sender.
     * @param reply true if an IDMessage is requested of the recipient.
     */
    public IDMessage( String senderName, boolean reply )
    {
        this.senderName = senderName;
        this.reply = reply;
    }

    /**
     * Deliver the Message.
     * @param messageReceiver The interface that will receive the Message's data.
     * @param sender The RemoteHandler of the sender.
     * @param receiver The RemoteHandler of the receiver.
     */
    public void execute( MessageReceiver messageReceiver, RemoteHandlerImpl receiver )
}
if ( messageReceiver == null )
{
    System.out.println( "IDMessage: Error no MessageReceiver." );
    return;
}

messageReceiver.idSender( receiver, senderName, reply );
}

DisconnectMessage.java
package standinprefix.booker.simulation.remoterepository.message;

import standinprefix.booker.simulation.remoterepository.*;

/**
 * DisconnectMessage allows one Remote to disconnect from another.
 * @version 1.0 1998 March
 * @author C. P. Booker
 */
public class DisconnectMessage extends Message
{
    protected String senderName;
    protected boolean handshaking;

    public DisconnectMessage( String senderName, boolean handshaking )
    {
        this.senderName = senderName;
        this.handshaking = handshaking;
    }

    public void execute( MessageReceiver messageReceiver, RemoteHandlerImpl receiver )
    {
        if ( messageReceiver == null ) return;

        messageReceiver.disconnect( receiver, handshaking );
    }
}
APPENDIX B

Message Passing without parsing.

The GDSA definitely does not mandate the protocol for message passing between Players. However, in the course of developing the prototype Framework and in work with other applications, a message passing scheme that does not require parsing has been developed; because it may be of interest to users of the GDSA Framework, the scheme is documented here.

Any Message is an Object and descends from the abstract class, Message, which has one abstract method, execute(...) as shown in Listing A2.1:

```java
package lanl.tsa3.simulation.messaging.message;

import lanl.tsa3.simulation.identification.*;
import lanl.tsa3.simulation.messaging.*;
/**
 * Message is the abstract, parent class of messages exchanged between
 * Players
 * Message is an abstract class that must be extended to use.
 * *
 * @version 1.0 1998 April
 * @author C. P. Booker
 */
public abstract class Message implements java.io.Serializable {
    /**
     * Cause the Message to 'deliver' the message by providing it with an
     * interface
     * constructed to receive the message data.
     * @param receiver The MessageInterpreter interface that will receive
     * the message.
     * @see lanl.tsa3.simulation.repositorioy.MessageReceiver
     */
    public abstract void execute( MessageInterpreter messageInterpreter,
                                   PlayerID sender,
                                   PlayerID receiver );
}
```

Listing A2.1. The Message class.

A specific message simply extends Message and implements the execute method according to its data requirements as depicted in Figure A2.1.

![Message Hierarchy Diagram](image)

Figure A2.1. Message Hierarchy.
The `execute` method accepts a reference to a `MessageInterpreter` which is an interface, an implementation of which will accept the Message's data. The rest of the arguments, sender and receiver, just serve to identify the two, if necessary, when the data is actually delivered to the `MessageInterpreter`. When the `execute` method is called for a non-abstract descendant of `Message`, the `Message` first checks the `MessageInterpreter` for the interface necessary to accept the data using the Java `instanceof` keyword. If the `MessageInterpreter` cannot accept the Message's data, the `execute` method simply returns. If, on the other hand, the `MessageInterpreter` can accept the data, the `Message` casts it to the requisite interface and delivers the data via the appropriate method. Figure A2.2 shows the simplified Event Trace for *direct* Message passing; that is, for Message passing where the Players have references to each other such as found in monolithic code running on a single computer or with RMI communication schemes.

![Figure A2.2. Event Trace of direct Message Passing.](image)

Notably, the `Message` is passed to the receiver and placed on a queue in the receiver under one Thread. Another Thread takes the message off the queue and invokes its `execute` method with the appropriate arguments. Two Threads are used to insure that a single `Message` does not monopolize the CPU; this goal could be furthered by the appropriate assignment of different Thread Priorities to the two Threads.

The Event Trace for Socket based communication is shown in Figure A2.3. It is essentially the same as that for direct Message passing, but there is more overhead to accommodate the Socket connection. The `Message` is passed through the Socket connection via an
ObjectInputStream/ObjectOutputStream pair, and the receiving Thread simply waits until a complete Message is passed through the Stream. Thereafter, the implementation is identical to that of the direct Message passing scheme.

An attractive feature of this Message passing scheme that does not require parsing is that the message handling code is written as a reusable component. As a result, the Player may be built with either the direct or Socket-based Message Handling component. Messages are thus passed between Players without the Players being aware of whether they are communicating directly or over a network; the selection of a Message Passing scheme is transparent to the Player. In the future, a Message Handling component is planned that can handle both Message passing schemes simultaneously so that mixed mode communication is possible between Players, and the particular mode of communication with a particular Player is of no interest to either Player.
REFERENCES


