Reflective Remote Method Invocation

G. K. Thiruvathukal[1] *Tools of Computing LLC and Argonne National Laboratory Chicago, Illinois, U.S.A.*

L. S. Thomas[2] *Illinois Institute of Technology, Chicago, Illinois, U.S.A.*

A. T. Korczynski[3] *Illinois Institute of Technology, Chicago, Illinois, U.S.A.*

Abstract

Remote Method Invocation (RMI) is available in the current Java language design and implementation, providing the much-needed capability of allowing objects running in different Java processes to collaborate using a variation on the popular Remote Procedure Call (RPC).

Although RMI provides features which are desirable for high-performance distributed computing, its design and implementation are deficient in key areas of importance to the high-performance computing community in general. This paper addresses the key deficiencies of RMI and how these deficiencies affect the design and implementation of distributed object applications.

Reflective RMI (RRMI) is an open RMI implementation which makes better use of the object-oriented features of Java. RRMI is so-called reflective because it directly employs the reflection capabilities of the current Java language to invoke methods remotely. RRMI makes use of the dynamic class loader (a class called NetClassLoader) to allow client/server applications to be built for high-performance computing systems without having all of the.class files present on all nodes in a parallel computation. Among other features discussed are support for asynchronous remote method invocations with deferred reply and exception semantics.

Keywords: Common Object Request Broker Architecture (CORBA), Object Request Broker (ORB), Remote Method Invocation (RMI), Remote Procedure Call (RPC), Message Passing Interface (MPI), Reflection, Introspection, Exception Handling, Asynchronous, Synchronous, Delegation, and Futures.

1 Introduction

Is distributed computing easy in Java? Can it be made easier? Current options in Java for distributed computing are Common Object Request Broker Architecture (CORBA) and Remote Method Invocation (RMI).

Consider CORBA [13]. The Java language has made it easier to use CORBA as many of its built-in frameworks support CORBA well. CORBA is both a language independent and a location transparent framework, which means objects are interchangeable as long as the interfaces to the objects remain the same. The communication between different distributed objects is handled by an object request broker (ORB). Distributed objects communicate using the Internet Inter-ORB Protocol (IIOP). The Object Management Group (OMG), a consortium responsible for the design of CORBA, has not defined actual programmatic interfaces, the so-called Application Programming Interfaces (API), but rather each ORB vendor is given the latitude to define its API. IIOP is what makes it possible for different ORB implementations to

---

[1] All authors are members of the Java and High-Performance Computing Research Group. For more information, please visit **http://www.jhpc.org**. George K. Thiruvathukal can be reached at **george.k.thiruvathukal@acm.org**.

[2] Lovely S. Thomas can be reached at lovelyst@aol.com or **thomlov@rice.iit.edu**.

[3] Andy T. Korczynski can be reached at **andyk@rice.iit.edu**.

make use of one another's objects. What makes this possible is Interface Definition Language (IDL). OMG created the IDL to make the separation between interface and implementation clear. A language mapping is needed to translate an IDL file into the programming language of choice. A Java language mapping has recently been completed and is in the process of getting OMG approval.

CORBA itself is an excellent technology for building distributed applications involving multiple languages and commercial vendors; however, for high-performance distributed computing applications, often involving a single language, it imposes a heavy burden on programmers to create the remote object and deploy a large-scale application. The coding process is rather complicated:

- Define application interfaces using IDL.
- Generate stubs and skeletons from IDL.
- Inherit implementation classes from the generated skeleton classes.
- Create a server container object.
- Register the object.

CORBA also does not appear suitable for use with Java Applets. The large number of classes (~400) to be loaded by an application affects the performance, program size, and deployment costs significantly. Netscape Communicator bundles the VisiBroker ORB from Visigenic to avoid this problem. Bundling also does not appear to be a good solution when considering all vendor ORBs are different and the applet may need to embed another vendor ORB. Such an embedding would serve to make the client even more bulky.

Although CORBA is useful, it is perhaps a severe case of over-engineering in what many people believe will be an object-centered, possibly Java-enabled, write-once and run everywhere world. This led the Java language designers to propose an alternative method of object-brokering called Remote Method Invocation (RMI) [14]. Although certain aspects of RMI are similar to CORBA, RMI itself is more similar to Remote Procedure Call (RPC).

Developing an RMI application is much simpler than developing one in CORBA, particularly because an RMI program is always written the same way: there is only one implementation. The steps involved are as follows:

- Any object to be served must be inherit from RemoteServer or UnicastRemoteObject classes.
- Stubs and skeletons are reverse engineered using the **rmic** tool provided with the Java Developer's Kit.
- Application must be translated and *linked* with the generated stubs.
- The server must be registered.

Thus there are many striking similarities between CORBA and Java. The use of CORBA front loads much of the effort on the application designer. IDL is used to define the interfaces from which stubs and skeletons are generated. Java RMI generates the stubs and skeletons but forces that implementation to use RemoteServer or UnicastRemoteObject as a base class, primarily so the stubs and skeletons can be generated. This causes many problems for building applications with a modular design and in particular forces a design decision to be made up front: which objects will be remote?

CORBA and Java RMI provide strong support for synchronous remote method invocation. High-performance network computing demands support for a transparent asynchronous remote method invocation capability. CORBA comes close to supporting this need partially with one way calls; however, a better mechanism is needed to address the complicated matter of deferring replies and exceptions which arise from a one way call.

Toward the goal of maintaining the simplicity of Java RMI and enabling distributed applications to be developed specifically in research areas requiring the best balance of performance and features, this paper presents Reflective Remote Method Invocation (RRMI). RRMI is a class library which supports the core features of RMI but is not tied to the Java Virtual Machine in any way. It uses a descriptor-based scheme for defining and invoking remote methods. It supports synchronous and asynchronous forms of remote method invocation, borrowing ideas from the implementation of Message Passing Interface (MPI) and a delegation based mechanism for the implementation of asynchronous forms. RRMI has been tested under Solaris and Windows NT and is known to work with the appletviewer. The implementation depends on features of the current Java Developer's Kit (version 1.1 or later) and makes use of the Socket, SocketServer, Serializable, and the Java reflection package (java.io, java.net, and java.lang.reflect).

Organization of the Rest of the Paper

The remainder of this paper is organized as follows. First, we discuss Remote Method Invocation and Remote Procedure Call (RPC) as provided by Sun Microsystems. The discussion centers around whether RMI is a good fit for objects and, in particular, distributed objects. We then discuss whether RMI can easily be used for high-performance distributed object computing. Following the discussion of RMI, we present Reflective Remote Method Invocation (RRMI). During this presentation, we discuss the qualitative strengths and weaknesses of our approach and compare it to RMI. That latter part of the paper presents a number of code examples, preliminary performance data, conclusions, and future directions of the work.

2 Critique of RMI

Remote Method Invocation (RMI) represents the latest generation of technology based on the familiar Remote Procedure Call (RPC). This section addresses the relevant question of "What is RMI?" and presents a critique of RMI from the vantage points of object-oriented and distributed systems.

2.1 Remote Procedure Calls and Objects: A good fit?

The following discussion is intended for overview purposes only and not to be an all-encompassing discussion of all available RPC implementations. This background is necessary to understand the ensuing discussion of RMI, RRMI, and their application to distributed object computing.

Remote procedure calls are widely familiar to researchers in parallel and distributed systems. The popular RPC package has been developed commercially by Sun Microsystems (Sun RPC) and the Open Software Foundation (DCE RPC).

The basic idea of a remote procedure call is best understood by first establishing a client/server relationship between two entities. One entity acts as a supplier of functionality (a set of procedures). The other entity acts as a consumer and wishes to use the functionality provided by the supplier. RPC was designed to make it easy for a consumer to access functionality provided by a supplier. A number of assumptions are made:

- the supplier and consumer are both running processes
- the supplier provides a set of global functions which may be called remotely
- a network or other shared communication medium exists

In an RPC system, the principal design goal is to provide a local view of a remote procedure to a consumer. More precisely, the consumer wants a procedure call, albeit remote, to work the same way as local procedure call.

Practical RPC tools make this possible by doing the following:

- using interfaces
- generating stub procedures which can be used by a consumer to wrap the actual code which calls the remote procedure
- providing a registry service for the supplier
- providing support for migrating data between different architectures (e.g., XDR encoding)

An important aspect of RPC systems is the presence of suppliers (servers) and consumers (clients). The server is used to provide or extend functionality. Thus the supplier has the active role in a distributed computation, while the consumer has a more passive role. This is often a severe restriction in many applications, especially parallel and distributed applications, and leads to the somewhat awkward and asymmetrical notion of a server callback which allows a consumer to act somewhat as a supplier but only to the server with which it registers a callback. As an interesting aside, the CORBA framework for distributed object computing employs callbacks in a similar fashion.

The notion of remote procedure calls is particularly useful as a starting point for building distributed systems; however, there are many limitations. Before exploring these limitations in depth, it pays to consider the object world for a moment. The object paradigm can be considered a variation on the client/server theme of RPC. An object itself is commonly defined in the literature as being “an instance of a class”. The class mechanism defines a set of attributes and a set of methods (variables and member functions are the terms now in most widespread use). An object-oriented system is built by constructing instances of a class, called objects, which employ the services (methods) of other objects to effect computation or system behavior. This suggests there is a natural client/server relationship between objects, because a given object can act as a client of another object simply by invoking one of its methods. It is precisely this client/server relationship that led to the use of remote procedure calls as a way of remotely invoking methods when two objects are not co-located in a given process.

The client/server relationship between objects, however, is not completely useful. Virtually all objects in an object-oriented system are suppliers of functionality. Objects that do not serve functionality are called data objects. Because objects tend to be suppliers as well as consumers, the overall architecture tends to shift from being client/server to server/server. While RPC itself is a language-independent notion, RPC implementations which make the distinction between client and server do not appear to be a good basis for object oriented systems.

Object oriented systems are characterized by a highly dynamic nature. Objects come and go, often with very high turnover rates. An RPC implementation in such a world must be smarter than in the non-object world. In the case of Java, a remote procedure call could be issued by an object which ceases to exist after making the call. As Java supports exception handling, facilities must be provided to address exceptions when the caller disappears.

Most object-oriented languages provide pointer and reference semantics which are not location transparent. Without engaging in a discussion of why languages have been developed this way (or discussing the von Neumann model), this presents a number of problems for RPC implementations. RPC programmers in C and C++ always had to provide serialization code to migrate data structures passed as parameters to or returned from remote procedure calls. Java programmers do not have to do this, but remote method calls cause the reference semantics of Java to be turned into value semantics. This specific problem led to the design and popularity of CORBA as a solution, but CORBA continues to exhibit many problems of RPC when it comes to presenting the view of a symmetrical server/server relationship.

Stubs and skeletons are a particularly good idea for non-object oriented systems; however, the whole purpose of generating them is to discover interfaces (in particular, public interfaces). Interface discovery is possible via reflection (or introspection), an often overlooked capability in the latest Java programming language. The ability to call methods using reflection eliminates the need for a RemoteObject and stub/skeleton generation and thus make it possible for any object to be served and used remotely. Reflective RMI makes use of this capability as a core part of the framework, and support is provided to make it (almost) as easy as RMI from the standpoint of the caller.

2.2 RMI and High-Performance Computing: A good fit?

In high-performance computing systems, TCP/IP is not the only protocol spoken between computers. Often, multiple protocols are used as in the Globus metacomputing toolkit. RMI in its current instantiation only supports TCP/IP. Because RMI does not have an open design and is tied to the virtual machine, it cannot be easily extended by any organization other than Sun to support other protocols. The MPICH [7] (Message Passing Interface on Channels) research demonstrated successfully that making the message passing interface open and allowing different transport devices to be provided by different implementors led to a widely usable message passing layer which worked in all high-performance computing environments.

The current RMI implementation is tied to the Java Virtual Machine. Often the case for doing this is either a performance or security argument. Neither appears to apply. Reflective RMI (the subject of this paper) is implemented as a class library which achieves nearly identical performance to RMI and provides security (inasmuch as Java is considered secure).

RemoteObject is a required base class for any object which is to be used remotely. This presents a serious problem for high-performance and object-oriented codes, where only single inheritance is provided. The research presented in this paper allows any object to be supplied at any time. The fact that RemoteObject is required breaks the natural client/server relationships between objects, because programmers are forced to create a new class before a given object can be used.

3 Reflective Remote Method Invocation

Reflective RMI is so-called reflective because it directly employs reflection to invoke remote methods. This stands in contrast with RMI, where stubs and skeletons are used as in the familiar RPC. We now present the elements of the design of RRMI. Following this design discussion, we will provide detailed examples of how the implementation of RRMI is used to construct distributed programs from different patterns supported by RRMI.

3.1 Extending the Class Loader to be Network Aware

The *network class loader* is a central aspect of the design. A similar approach is employed in ObjectSpace Voyager [12]; a rudimentary sketch of how to implement one is discussed in [2]. The basic idea of a network class loader is to provide an extension to the Java class loader, which has an abstract class defined in class java.lang.ClassLoader. We have extended this class to provide a number of different class loaders. In our current implementation, we use a *central server* approach wherein consumers may load classes from a given server. Before proceeding, this design decision does serialize all requests for a given class to be loaded; however, this decision does not affect performance of the underlying RRMI mechanism for invoking remote methods. We are experimenting with other designs which employ replication and push technologies to keep a set of consumers coherent, but it is beyond the scope of this paper to discuss these designs as it represents future work. The network class loader itself is supplied in the current implementation as two classes NetServer and NetClient to allow RRMI to be used in much the same manner as RMI for Java-to-Java client/server development.

3.2 Remote Method Descriptor Management

Descriptor-based method invocation is used as a preferred mechanism to calling a local stub procedure. The Icon programming language [6] successfully demonstrated the practical use of procedure descriptors as a user-level method of invoking procedures. Using descriptors complicates slightly the manner by which remote methods are invoked; however, the FunctionBuilder class (discussed shortly) provides an elegant solution to remedy this difficulty.

Two classes are provided to effect a remote method call: Descriptor and ActualParams. The Descriptor class is constructed straightforwardly with the name of the method being described and a method addParamDesc which allows each parameter to the method to be defined. For example, if the remote function is defined as below:

```
public void multiply(Matrix another);
```

A descriptor could be constructed as follows:

```
Descriptor multiplyDescriptor = new Descriptor("multiply");
multiplyDescriptor.addParamDesc("Matrix", "another");
```

You could then bind parameters and call the procedure (via a local broker) as follows:

```
ActualParams ap = new ActualParams(multiplyDescriptor);
ap.addParam("another, new IdentityMatrix(4,4));
Object reply = localBroker.invoke(ap);
```

Note that all remote procedure invocations result in an object being returned. C++ and Java both do not consider the return type in the method signature. We provide facilities in RRMI which will allow a type guard to be established to ensure the return type is valid and a meaningful cast can be performed.

Remote method descriptors are not the most user-friendly interfaces from a programming standpoint. RRMI provides the FunctionBuilder class to facilitate the management of remote method descriptors. One must still create an ActualParams object (which is reusable and mutable) to invoke the actual method; however, there is a great deal of comfort achieved knowing the remote function has been described appropriately.

In general the client wants to be able to do:

```
result = remoteObject.remoteMethod(a1, a2, ..., aN);
```

The FunctionBuilder class allows a descriptor to be generated from a method declaration. Any Java method declaration can be specified to the FunctionBuilder class constructor as follows:

```
FunctionBuilder fb = new FunctionBuilder("public void
multiply(Matrix rhs)");
```

The FunctionBuilder also provides a setFunction method to change the function descriptor to be built.

```
FunctionBuilder fb = new FunctionBuilder();
fb.setFunction("public void multiply(Matrix rhs)");
```

Much of what is shown here could be automated by using a simple preprocessor. It may not be apparent immediately, but the key advantage of using a descriptor-based scheme is the ability to migrate away from the current RMI scheme which relies on the RemoteObject base class being inherited by objects to be used remotely. Also, the descriptor-based scheme makes it easy to invoke methods over arbitrary transports, such as MPI and Nexus. We have a related ongoing effort precisely in this area.

3.3 Remote Method Invocation Mechanisms

Thus far, we have discussed the network class loader and descriptor-based procedure calling. These mechanisms represent the core of what is needed to build a flexible and open remote method invocation mechanism. This also represents a significant shift from the design of RMI and CORBA implementations. In RRMI, classes can be loaded into the application at any time, thus allowing a supplier of objects to be dynamically configurable. Pure Java language mechanisms are tapped to achieve this result. As well, the descriptor-based scheme allows any object to serve functionality to consumers. This is particularly important from an engineering perspective, where a programmer often times is forced to perform various twistings and contortions to coerce the design of a distributed application into the RMI-imposed world of remote objects. We wish to point out CORBA does not completely solve the problem either. Most implementations impose similar burdens on the programmer.

Essentially, RRMI presents a lightweight broker to the programmer with robust facilities for remote method invocation. RRMI supports both synchronous and asynchronous remote method invocations, borrowing some ideas from MPI in the latter case of asynchronous calls.

The synchronous case has already been unveiled in presenting how the network class loader used from a supplier and consumer perspective.

Asynchronous calls present some additional complications from an implementation perspective. First, multiple asynchronous calls can be outstanding, taken strictly from the consumer's perspective. MPI has introduced the notion of a completion handle for the purpose of tracking the progress of an outstanding, non-blocking send/receive call. Second, the method of an object where the call was made may terminate. Thus the calling context is lost. Third, the object itself may cease to exist. Fourth, there is the ugly matter of exception handling.

This leads to a number of possibilities:

- The caller is not interested in the reply nor the exception.
- The caller wants to do other processing and then poll for the reply and/or exception.

- The calling method terminates; perhaps a running thread will continue to monitor for completion and await the reply/exception.
- The object where the call was made ceases to exist; another object listens for the reply/exception.

One way remote method invocations are important for high-performance computing. If a call can be performed one-way, less state information is maintained for handling the call and fewer communication steps are involved. RRMI supports this with the invokeAsyncNoReply call via the local NetClient.

The second case is often used in message passing styles of programming. An asynchronous call is posted with the idea of it completing somewhat soon but allowing other processing to occur. One might argue: Why not simply use a thread to invoke a synchronous procedure? The RRMI perspective is that threads programming (which not only involves threads but the often disturbing condition variables) is not for everyone, and many programmers find it difficult to use. In RRMI, the asynchronous forms of invoke employ threads to allow programmers to use multithreading implicitly. RRMI supports the second case with the invokeAsync call, which returns a RemoteMethod object instance to the user. Note: this object is intended to be used as a completion handle only. It in no way defines the actual remote method. Descriptors are used for this purpose.

The third case is a bit more complicated, in particular, due to the possibility of a non-local exception being generated. RRMI addresses this concern by deferring the handling of an exception and coupling exception handling with the facilities for getting the reply. As described in the second case, a RemoteMethod object allows one to track the completion of an asynchronous remote method invocation. This class provides a method called getReply which allows the programmer to obtain the reply, if it has been received. Exceptions are delivered along with any reply received. This forces the programmer to handle the exception when obtaining the reply. This means the exception is re-thrown locally by the getReply method itself.

The final and fourth case is where the object from where the call was made ceases to exist and another object listens for the reply/exception. For this purpose, RRMI employs the delegation-based listener model of Java. Two interfaces are provided: RemoteReplyListener and RemoteExceptionListener. By separating replies and exceptions into two different interfaces, some optimizations can be done to minimize the amount of state information and communication cost. The listener model also allows a little more sophistication to be achieved in remote method calling. For example, the result of a call can be broadcast locally or remotely by having a number of local listeners await it and then (concurrently) perform another remote method call in the listener method itself. This capability very much looks like macro-dataflow.

4 Code Examples

Thus far, this paper has been concerned with concepts and principles. Section "Remote Method Invocation Mechanisms" presented a discussion of synchronous and asynchronous remote procedure calls and described these cases in some detail along with a sketch of how these cases are handled in RRMI. Each of these cases is presented in the following series of code examples.

RRMI allows the server and clients to create objects on the server. For the purpose of making all of our examples more self-contained, the clients use the createRemoteObject method of the NetClient class to dynamically load and create an instance of a class on the server.

Figure 1 demonstrates how to perform a synchronous remote procedure call. The client obtains a reference to a remote object of a hypothetical class called Hello (which has a method named "hello"). This is illustrated in the body of the main method of class Worker. First, the client creates an instance of NetClient and specifies the host and port where a NetServer can be found. If this is not successful, an exception will be thrown. The client then creates a remote object of class Hello. This results in the Hello class being loaded on the server, if it has not been loaded already. Then the instance is created and a remote reference is returned to the client. In RRMI, RemoteObject is used to refer to remote objects as opposed to being the base class for implementing remote objects, as in RMI.

Once the remote object reference has been obtained, any method which is publicly available in the class Hello can be executed remotely. It is possible the class Hello may depend on classes not available to the client. Using the NetClient, additional classes may be loaded on the client side as well. To invoke a remote method, the client must build a Descriptor instance which describes the method to be called. In this and all subsequent examples, the method to be called is public void hello(String name). The descriptor is constructed with the name of the remote method to be invoked. Then a series of addParamDesc calls is performed to add the parameters in order of appearance. The parameters must be specified as a <class name, parameter name> pair[4].

Finally, the remote method can be invoked against the remote object. The synchronous form of invocation has the same name as found in Java's reflection specification: *invoke*. Other forms of invoke are provided for asynchronous calls, which are presented in the remaining examples.

Exceptions which are raised remotely in the synchronous case must be handled locally. The try block will catch any exception which is raised remotely.

Figure 1: Synchronous RRMI with local reply/exception semantics.

```
import org.jhpc.rrmi.*;
import java.io.*;

public class Worker
{
   public static void main(String args[])
   {
      try {
         NetClient broker = new NetClient("tiamat.mcs.anl.gov:1000");
         RemoteObject myObject = broker.createRemoteObject("Hello");
         if (myObject != null) {
            Descriptor methodDesc = new Descriptor("hello");
            methodDesc.addParamDesc("java.lang.String", "name");
            ActualParams actuals = new ActualParams(methodDesc);
            actuals.addParam("name", "George");
            Object result = myObject.invoke(actuals);
         }
      }
      catch (Exception e) {
         System.err.println(e);
      }
   }
}
```

[4] Scalar data types are not supported in the present implementation. We have figured out an elegant way to address this issue. We believe most users will prefer to work with real objects (e.g., use **Integer** in lieu of **int**).

RMI programmers are accustomed to having a more local view of a remote procedure call. To that end, RRMI attempts to bring more user-friendliness to the manipulation of descriptors. Specifically, the FunctionBuilder class is provided to allow one or more actual Java method declarations (headers) to be parsed to generate a Descriptor instance. This is illustrated in Figure 2. A future release of RRMI will build descriptors for all publicly defined methods in a class, making it even easier.

Figure 2: Synchronous RRMI employing FunctionBuilder for descriptors.

```
import org.jhpc.rrmi.*;
import java.io.*;

class Worker
{
  public static void main(String args[])
  {
    try {
      NetClient broker = new NetClient("tiamat.mcs.anl.gov:8000");
      RemoteObject myObject = broker.createRemoteObject("Hello");
      if (myObject != null) {
        FunctionBuilder fb =
          new FunctionBuilder("public void hello(String name)");
        Descriptor methodDesc = fb.getDescriptor();
        ActualParams actuals = new ActualParams(methodDesc);
        actuals.addParam("name", "George");
        Object result = myObject.invoke(actuals);
      }
    } catch (Exception e) {
      System.err.println(e);
    }
  }
}
```

Figure 3 demonstrates how the asynchronous capabilities of RRMI can be exploited when the completion semantics are intended to be local. The setup for performing the asynchronous call is exactly the same as the synchronous call. Invocation is performed using the *invokeAsynchronous* method. This method returns a completion handle object which can be used to test for the completion of the call using polling. A similar method for non-blocking sends and receives is used by Message Passing Interface (MPI) research.

Figure 3: Asynchronous RRMI with local reply and exception management semantics.

```
import org.jhpc.rrmi.*;
import java.io.*;

class Worker
{
  public static void main(String args[]) {
    try {
      NetClient broker = new NetClient("tiamat.mcs.anl.gov:8000");
      RemoteObject myObject = broker.createRemoteObject("Hello");
      RemoteMethod methodHandle;
      if (myObject != null) {
        FunctionBuilder fb =
```

```
            new FunctionBuilder("public void hello(String name)");
          Descriptor methodDesc = fb.getDescriptor();
          ActualParams actuals = new ActualParams(methodDesc);
          actuals.addParam("name", "Lovely");
          methodHandle = myObject.invokeAsynchronous(actuals);

          // other computation could occur here

          Object result;
          while (!methodHandle.isComplete()) {
            try {
              result = methodHandle.getReply();
            } catch(Exception e) {
              // the remote exception, if it occurred, is re-thrown
              // and must be caught locally
            }
          }
        }
      }
      catch (Exception e) {
        System.err.println(e);
      }
    }
  }
```

Polling is not necessarily the most efficient and reliable way to work with asynchronous calls, especially in an object-oriented language like Java. The Java language provides extensive support for event handling using a listener model based on delegation. Delegation has long been used in the object-oriented community and will not be defined here[5].

Example 4 demonstrates how this idea has been adapted to help support asynchronous remote method invocation. Two interfaces are provided in Java: RemoteReplyListener and RemoteExceptionListener. This allows an object to be specified which can handle replies and/or exceptions which occur remotely without the need for polling as shown in Figure 3. In Figure 4, a worker thread is created by the main method which in turn fires off an asynchronous remote method call. The thread then loops forever. Eventually, a reply or exception will occur, in which case the Worker instance will be notified via one of the interface methods.

Figure 4: Asynchronous RRMI in a thread using RemoteReplyListener and RemoteExceptionListener.

```
import org.jhpc.rrmi.*;
import java.io.*;

class Worker extends Thread implements RemoteReplyListener,
RemoteExceptionListener
{
  RemoteMethod completionHandle;

  public static void main(String args[]) {
    Worker w = new Worker();
```

[5] The Self research presents an approach to object-orientation based entirely on delegation. Bertrand Meyer presents an excellent discussion on delegation versus inheritance in his classic text on Object Oriented Software Construction.

```
      w.start();
    }

    public void run() {
      try {
        NetClient broker = new NetClient(args[0]);
        RemoteObject myObject = broker.createRemoteObject(args[1]);
        if (myObject != null) {
          Descriptor methodDesc = new Descriptor("hello");
          methodDesc.addRemoteReplyListener(this);
          methodDesc.addRemoteExceptionListener(this);
          methodDesc.addParamDesc("java.lang.String", "name");
          ActualParams actuals = new ActualParams(methodDesc);
          actuals.addParam("name", "Andy");
          completionHandle = myObject.invokeAsynchronous(actuals);
        }
      }
      catch (Exception e) {
        System.err.println(e);
      }
      while(true);
    }

    public void remoteReply(RemoteMethod m, Object result) {
      if (completionHandle == m) {
        // the RRMI call has completed
      }
    }

    public void remoteException(RemoteMethod m, Exception e) {
      if (completionHandle == m) {
        // the RRMI call resulted in a remote exception
      }
    }
  }
```

5 Performance Data/Preliminary Results

We conducted an experiment between two SPARC machines running Solaris 2.51 on a 10 Mbps LAN to determine the cost of calling a parameterless procedure to obtain base performance data. A loop of one thousand synchronous RMI calls was done to determine the average cost of calling a procedure. Regular RMI performed at 3.7 milliseconds, while Reflective RMI performed at 3.8 milliseconds. There is no statistical difference between the two times. For this experiment we used the standard Java Developer's Kit, version 1.1.3, running on Solaris 2.5 without the runtime performance pack installed.

The important result here is that our current results suggest RRMI performs almost identically to RMI for procedure calling on TCP/IP. This is a significant result as we use a descriptor-based calling scheme, Java's introspection capabilities, and dynamically load classes into the running application.

6 Conclusions

This paper has presented RRMI. RRMI is very similar to RMI in principle. It is designed to support peer-to-peer remote procedure calls between consumers and suppliers. The work presented here is a first-generation product designed to mimic the behavior of the current RMI offering from Sun Microsystems.

RRMI itself is intended to be a basis for further investigation of a Java-to-Java technology which allows interaction of objects on multiple computers, particularly a large number of networked computers. Thus, an important design consideration was to provide facilities which are open and extensible. To that end, RRMI is implemented as a class library which runs with the current Java implementation and does not require any changes to the virtual machine. We did not believe changes to the language or virtual machine are appropriate as these technologies are presently proprietary to Sun. There is no reason to believe these technologies will become open any time soon.

Because we have been successful in creating an open design and implementation, we contend RRMI will allow a number of interesting investigations to be conducted. Of particular interest at the moment is the use of alternative transports for performing remote method invocations as well as the use of multiple transports. The RMI implementation from Sun will be difficult, if not impossible, to extend to support alternative transport layers. By the time this paper is published, we will be able to report on MPI as an alternative transport. We are working on an abstract transport interface to allow third-party transports to be incorporated in RRMI. Visit http://www.jhpc.org for more information.

Aside from an open design, RRMI offers many useful features to the distributed object programmer. Descriptor-based procedure calling enables any object to be served. This allows for much more effective use of inheritance, because the user application is no longer forced to inherit from RemoteObject. One may argue this is a matter of style; however, the current Java language does not support multiple inheritance (the authors hope it never will), and the primary role RemoteObject appears to have in RMI is much the same as the Serialization interface: it is a tagging mechanism.

The use of a network class loader also offers significant deployment advantages in a large-scale distributed application. Suppliers and consumers can basically start with a fixed and small set of classes. These classes, the network server and network client classes, can then dynamically load classes and create objects on all nodes of the computation. We are using these features of RRMI to build a ActiveJava (formerly JavaNow) which is a distributed computing environment based on actors, dataflow, data-parallel, task-parallel, and Linda.

RRMI presents a clean and clear design for the management of synchronous and asynchronous remote procedure calls. Using a technique called use-case analysis [8], we have been able to identify the usage patterns of these different calls. This has led to a very simple design which also accounts for performance and flexibility. An important aspect of the design is the streamlined and elegant handling of exceptions. Both local and non-local reply and exception semantics are addressed thoroughly.

7 Future Work/Status

RRMI is presently being used to implement ActiveJava [18], a coordination environment similar to JavaSpaces [17] and ObjectSpace Voyager [12], targeted to both high-performance scientific and commercial computing applications. More information about ActiveJava can be found at http://www.jhpc.org.

At the time of writing, it is not known whether RRMI works with Netscape Communicator or Internet Explorer. It is known to work with the appletviewer. Sun has previewed a technology called Java Activator which will allow users to download and run a JVM outside of the browser which renders applets inside the browser. Individuals interested in using RRMI will be able to do so in the Activator-enabled environment which is fully compatible with the appletviewer that ships with JDK.

We have some improvements planned to further simplify the method calling scheme to the level found in RMI. In particular, we are planning the development of a simple preprocessor to turn a local view of a remote method call into code similar to that shown in our examples. For those who are willing to live with the descriptors as is, we will provide code to generate remote method descriptors using the reflection interfaces to iterate over all methods of a given class.

We also have plans to incorporate NexusJava [4] and MPI [7] transports for RRMI. NexusJava was created primarily for interoperability with other Nexus applications and thus does not support all of the features of Nexus (e.g., multimethod communication, etc.). Nonetheless, should these communication protocols become available, RRMI would be immediately available to use them, if NexusJava were augmented to support other communication protocols.

Locating and naming objects is an another area we are currently researching. The use of **rmiregistry** is awkward and bulky for practical applications beyond simple client/server. The next generation of RRMI will provide a complete naming, registration, and location service for objects and will be based on open standards, such as Lightweight Directory Access Protocol (LDAP).

8 Acknowledgments

The authors wish to thank Nina Wilfred for proofreading and editing the final version of this paper and Gregor von Laszewski for making a number of useful suggestions for improving the presentation of the paper.

9 Bibliography

9.1 Papers and Books

1 G. Agha and C. Hewitt, *Concurrent Programming Using Actors*, In A. Yonezawa and M. Tokoro, Object-Oriented Concurrent Programming, MIT Press, 1987.

2 P. Chan and R. Lee, *The Java Class Libraries: An Annotated Reference*, Addison-Wesley, Reading, Massachusetts, 1997.

3 I. Foster, J. Geisler, C. Kesselman, and S. Tuecke, *Managing multiple communication methods in high-performance networked computing systems*, Journal of Parallel and Distributed Computing Systems, 40 (1997), pp. 35-48.

4 I. Foster, G. K. Thiruvathukal, and S. Tuecke, *Technologies for ubiquitous supercomputing: a Java interface to the Nexus communication system*, Concurrency: Practice and Experience, Vol. 9(6), 465-475.

5 I. Foster, C. Kesselman, and S. Tuecke, *The Nexus approach to integrating multithreading and communication*, Journal of Parallel and Distributed Computing, 37 (1996), pp. 70-82.

6 R. Griswold and M. Griswold, *The Icon Programming Language, 2nd Edition,* Addison-Wesley, 1992.

7 W. Gropp, E. Lusk, N. Doss, and A. Skjellum, *A high-performance, portable implementation of the MPI message passing interface standard*, Parallel Computing, 22 (1996), pp. 789-828.

8 I. Jacobsen, M. Christerson, P. Jonsson, and G. Overgaard, *Object-oriented Software Engineering--A Use Case Driven Approach*, Addison-Wesley, Wokingham, England, 1992.

9 B. Meyer, *Object Oriented Software Construction, 2nd Edition*. Prentice Hall. 1997.

10 R. R. Raje, J. I. Williams, and M. Boyles, *An Asynchronous Remote Method Invocation (ARMI) mechanism for Java*, Concurrency: Practice and Experience, Vol. 9(11), pp. 1207-1211.

11 W. Yu and A. Cox, *Java/DSM: A platform for heterogeneous computing*. Concurrency: Practice and Experience, Vol. 9(11), 1213-1224.

9.2 Web Sites

12 ObjectSpace Voyager, *http://www.objectspace.com/*

13 CORBA 2.0 Specification, *http://www.omg.org/*

14 RMI and Object Serialization Specification, *http://www.javasoft.com/*

15 JDK 1.1 Documentation, *http://www.javasoft.com/*

16 Visigenic Software, *http://www.visigenic.com/*

17 JavaSpaces, *http://www.javasoft.com/*

18 ActiveJava (and JavaNow), *http://www.jhpc.org/*