A Report on RMI and RPC
Submitted by
Sudharshan Reddy B

Abstract: This report mainly explains the RMI and RPC technologies. In the first part of the paper the RMI technology is briefly explained and in the second part of the paper the RPC technology is explained. The final part of the paper deals with the advantages and disadvantages of one technology over the other. In this paper the performance and implementation issues are generalized as these issues differ slightly from each application. Since these technologies extension of another the subtle differences with respect to performance changes from application to application.

What is RMI?
RMI is an acronym for remote method invocation method, is part of the core java API. The central idea behind this technology is the ability to call the methods of a remote object, shielding the programmer from mundane socket handling while promoting cleaner software architecture.

In java, you can invoke method calls on objects that reside on another computer without having to move those objects as to the machine making the method call. Such method calls are remote method invocations.

RMI applications are often comprises of two separate programs: a server and a client. A typical server application creates some remote objects, makes references to them accessible and waits for clients to invoke methods on these remote objects. A typical client application gets a remote reference to one or more remote objects in the server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred as to a distributed object application.

WHY?
RMI allows java programs to register their classes methods with a server that does the port arbitration in much the same way that RPC does. Once this has been set up, sending messages or invoking methods on the remote process is as simple as invoking method in a local object. This functionality facilitates rapid development of distributed applications, saving you the need to implement data conversion or transmission protocols.

RMI is dependent on the abilities to serialize object to turn an object into serial representation that is suitable for transmission over the network connection and then reconstruct it on the on receipt. This is necessary for remote methods that take objects as parameters as well as objects that have objects or return values.

Advantages of RMI:
The primary advantage is simplicity and clean implementation, leading to more maintainable robust and flexible applications. This isn’t to say a system can’t be written using sockets in place of RMI, Just that RMI removes a great deal of mundane tasks- such as parsing and switch logic. Since RMI has the potential to reduce great deal of code, more complex systems can be built with relative ease. The greatest benefit doesn’t don’t revolve around of use, however RMI allows us to create a distributed system while at the same time decoupling the client server objects. RMI is not the first an API to put these benefits on the table, but it’s a pure java solution for doing so. This means its possible to create
zero-install client for your users. A system can use RMI to its advantage in several ways:

- There's no client installation needed, only a Java 1.1-capable browser (or a JRE, for applications)
- If the DBMS is changed (I mean if we change from Access to ORACLE) then only the server objects need to be recompiled, while the server interface and the client remain the same.
- All the portions are easily distributed and the development teams can be given a section of the distributed architecture to work on. This simplifies coding and allows a group to leverage its talents better.
- It is safe and secure. RMI uses built-in Java security mechanisms that allow your system to be safe when users downloading implementations. RMI uses the security manager defined to protect systems from hostile applets to protect your systems and network from potentially hostile downloaded code. In severe cases, a server can refuse to download any implementations at all.
- Distributed Garbage Collection: RMI uses its distributed garbage collection feature to collect remote server objects that are no longer referenced by any clients in the network. Analogous to garbage collection inside a Java Virtual Machine, distributed garbage collection lets you define server objects as needed, knowing that they will be removed when they no longer need to be accessible by clients.

Disadvantages of RMI

RMI is slightly less efficient than the sockets because of the additional “layer” involved and because it must deal with the registry in order to communicate. Another concern is creating multithreaded servers safely; a common mistake is to assume the default threading will allow you to ignore code that ensures our server is thread-safe and robust. If you want to implement a concurrent user system you’ll need to provide the proper structure for doing so.

Architecture Overview:

The system mainly consists of 4 layers
1. Application layer
2. Proxy layer
3. Remote reference layer
4. Transport layer

The boundary of each layer is defined by a specific interface and protects each layer, therefore is independent of the next and can be replaced by an alternate implementation without affecting the other layers in the system.

Stub/Skeleton Layer: Stub/Skeleton Layer is the interface between application and rest of the RMI system. This layer does not deal with specifics of transport but transmits data to the Remote Reference Layer. A Stub for a remote-object is the client-side proxy for the remote object. A Skeleton for a remote object is a server-side entity that contains a method that dispatches calls to the actual remote object implementation.

Remote Reference Layer: Remote reference layer deals with the lower level transport interface. This layer is responsible for carrying out a specific remote reference protocol that is independent of client stubs and server skeletons. Remote Reference Layer has two components client-side components and server-side components. Client-side components contains information specific to the remote server. Server-side components
implements specific remote reference semantics prior to declaring a remote method invocation to the skeleton. Remote Reference Layer transmits data to the transport layer via the abstraction of a stream-oriented connection.

Transport Layer: Transport layer is responsible for connection setup, connection management and keeping track of dispatching to remote objects residing in the transport’s address space.

Garbage Collection of remote Objects: It is desirable to automatically delete those remote objects that are no longer referenced by any client. For this purpose RMI uses a reference counting garbage collection algorithm. RMI runtime keeps track of all live references within each JVM. When any client does not reference a remote object, the RMI refers to it using a weak reference. The week reference allows the JVM’s garbage collector to discard the object if no other local references to the object exist. The distributed garbage collected algorithm interacts with the local JVM’s garbage collector and deletes those objects.

What is RPC?

An RPC (Remote procedure call) technology is a standarized way of exchanging data using a single protocol or a series of protocols, depending on the RPC implementation. One or more clients (in terms of different products or projects, not of multiple instances) can connect to a server and exchange messages. The important thing is that if you use RPC technologies, every client on any operating system and platform can exchange messages with a server also running on any operating system and platform you like, as long as the RPC technology you want to use is supported (either as library or implemented in the application itself). Basically spoken, the HTTP protocol is a very specialized RPC technology.

Architecture Overview:

The request/reply communication paradigm is at the heart of a Remote Procedure Call (RPC) mechanism. RPC is a popular mechanism for structuring distributed systems because it is based on the semantics of a local procedure call—the application program makes a call into a procedure without regard for whether it is local or remote, and blocks until the call returns. A complete RPC mechanism actually involves two major components:

- A protocol that manages the messages sent between the client and the server processes and deals with the potentially undesirable properties of the underlying network;
- Programming language and compiler support to package the arguments into a request message on the client machine and then translate this message back into the arguments on the server machine (and likewise with the return value). This piece of the RPC mechanism is usually called a stub compiler.

When the calling process calls a procedure, the action performed by that procedure will not be the actual code as written, but code that begins network communication. It has to connect to the remote machine, send all the parameters down to it, wait for replies, do the right thing to the stack and return. This is the client side stub.

The server side stub has to wait for messages asking for a procedure to run. It has to read the parameters, and present them in a suitable form to execute the procedure locally. After execution, it has to send the results back to the calling process.
1. The client calls the local stub procedure. The stub packages up the parameters into a network message. This is called marshalling.
2. Networking functions in the O/S kernel are called by the stub to send the message.
3. The kernel sends the message(s) to the remote system. This may be connection-oriented or connectionless.
4. A server stub unmarshals the arguments from the network message.
5. The server stub executes a local procedure call.
6. The procedure completes, returning execution to the server stub.
7. The server stub marshals the return values into a network message.
8. The return messages are sent back.
9. The client stub reads the messages using the network functions.
10. The message is unmarshalled and the return values are set on the stack for the local process.

Generating stubs

Common RPC methods use implicit typing. This means that both the server stub and the client stub must agree exactly on what the parameter types are for any remote call. If this were done by hand, then obscure errors would result. So it must be done automatically.

For a normal procedure call, the compiler is able to look at the specification of the procedure and do two things: generate the correct code for placing arguments on the stack when a procedure is called, and generate correct code for using these parameters within the procedure.

In RPC, this is more complex. The compiler must generate separate stubs, one for the client stub embedded in the application, and one for the server stub for the remote machine. The compiler must know which parameters are in parameters and which are out. In parameters are sent from the client to server, out parameters are sent back.

RPC Port Mapper Program

Client programs must find the port numbers of the server programs that they intend to use. Network transports do not provide such a service; they merely provide process-to-process message transfer across a network. A message typically contains a transport address consisting of a network number, a host number, and a port number.

RPC Authentication:

The caller may not want to identify itself to the server, and the server may not require an ID from the caller. However, some network services, such as the Network File System (NFS), require stronger security. Remote Procedure Call (RPC) authentication provides a certain degree of security. RPC Authentication Protocol, NULL Authentication, UNIX Authentication, Data Encryption Standard (DES) Authentication, DES Authentication Protocol, Diffie-Hellman Encryption are the following parts of RPC authentication. RPC deals only with authentication and not with access control of individual services. Each service must implement its own access control policy and reflect this policy as return statuses in its protocol.

RPC Features:

The features of Remote Procedure Call (RPC) include batching calls, broadcasting calls, callback procedures, and using the select subroutine. Batching allows a client to send an arbitrarily large sequence of call messages to a server. Broadcasting allows a client to send a data packet to the network and wait for numerous replies. Callback procedures permit a server to become a client and make an RPC callback to the client's process. The select subroutine examines the I/O descriptor sets whose addresses are passed in the readfields, writefields, and exceptfields parameters to see if some of their descriptors are ready for reading or writing, or have an exceptional condition pending. It then returns the total number of ready descriptors in all the sets.
When is Distinct RPC for Java a better choice than RMI and Why?

Distinct RPC for Java is the clear winner when any of the following is important:

1. Whenever you need to interoperate with C or C++
2. When compatibility with legacy systems is required
3. When ease of programming is an issue. RPC is smaller and much easier to program with compared with CORBA based programs.
4. When your distributed application is requesting the execution of functions on a remote system and speed is an issue. A typical procedure call in a distributed application consists of a function call issued by the client to a server. The server executes the function and returns the result to the client. In most cases the call itself and the returned results require the transmission of just a few packages, with the workload being the processing done on the server side.

We have taken some time to write test applications in both Distinct's Java RPC and RMI to illustrate the speed issue. In all our tests, Distinct ONC RPC/XDR for Java resulted 40% to 50% faster than RMI. We are making available two of the test programs used in this analysis.

Perhaps the most fundamental difference between most existing RPC systems and java RMI can be explained as follows. In most existing systems the writing an IDL interface is a static wire protocol, which defines the way the stub of one member of the distributed computation will interact with the skeleton that belongs to another part of the distributed computation. In the RMI system, the interaction point has moved into the address space of the client, which is a remote object and is defined in terms of java interface. This interface implementation comes from a remote object itself and is dynamically loaded when needed. This can vary in remote objects that appear from the client's point of view to be of same type because the client only knows that the remote objects are of atleast some type.

References:

3. “Remote procedure calls and java remote method invocation” Jim waldo, Sun Microsystems