www.ijecs.in
International Journal Of Engineering And Computer Science ISSN:2319-7242
Volume 4 Issue 6 June 2015, Page No. 12599-12603

Implementing Java Distributed Objects

Meenakshi[#1], *Sanjiv Kumar Singh*[#2]
M.Tech, Assistant Professor,
Department of CSE,
Shri Balwant College of Engineering &Technology,
DCRUST University

Abstract— Organizations rely on information to make effective business decisions and corporate intranets are changing the way organizations conduct business. As networking technologies continue to improve, with increasing bandwidth and reliability, effective distributed computing is becoming a reality. Organizations are relying on internet technologies to be the conduit for employees to access and manipulate corporate information. Having timely and accurate information is essential for effective management practices and optimization of limited resources. Information can be stored effectively and efficiently in
Server object & Client object can access it through distributed computing. Two popular methods for distributed computing are RMI and CORBA. RMI uses rmi protocol and CORBA uses IIOP protocol.

Keywords— RMI, CORBA, SOAP, IIOP

I. Introduction

Java, as a relatively simple, object-oriented, secure and portable language, is also a flexible and powerful programming system for distributed computing. Program development with Java results in software that is portable across multiple machine architectures and operating systems. Distributed programming in Java is supported by remote method invocation (RMI), object serialization, reflection, a Java security manager and distributed garbage collection. Java RMI is designed to simplify the communication between objects in different virtual machines allowing transparent calls to methods in remote virtual machines.

Organizations rely on information to make effective business decisions and corporate intranets are changing the way organizations conduct business. Information can be stored effectively and efficiently in
Server object & Client object can access it through distributed computing. Two popular methods for distributed computing are Remote Method Invocation (RMI) and Common Object Request Broker Architecture (CORBA). RMI uses rmi protocol and CORBA uses Internet Inter-ORB Protocol (IIOP).

Java is destined to become a language for distributed computing. Java Development Kit (JDK) comes with a broad range of classes for network and distributed programming. In this paper we focus on two different approaches: Remote Method Invocation (RMI) and Common Object Request Broker Architecture (CORBA) with their implementation.

II. Literature Review

The first version of JVMs had poor support for monitoring Java programs. Initially there was a simple debugger, jde, attached to the Java Development Kit (JDK). Then, there was an instrumented Java virtual machine build for JDK version 1.16 to support the collection of profiling data generated when executing a Java program. This approach was developed until version 2 of the Java platform. All JVMs for the new Java platform were equipped with interfaces for debugging (JVMDI) [6] and profiling (JVMPI) [7]. A new release of Java 2 Platform version 1.5, called Tiger, contains a new native profiling interface called JVMTI which is intended to replace JVMPI and JVMDI. JVMTI aims to cover the full range of native in-process tools access, which in addition to profiling, includes monitoring, debugging and, potentially, a wide variety of other code analysis tools.

Most of the tools for JVM versions from 1.2 to 1.4 are based on the Java Virtual Machine Profiling Interface (JVMPI) [7]. Starting with JDK 1.2 SDK it also includes an example profiler agent for efficiency examination called hprof [8], which can be used to build professional profilers. A Heap Analysis Tool (Hat) [9] enables one to read and analyze profile reports of the heap generated by the hprof tool and may be used e.g. for debugging "memory leaks". Tracer [10] is a debugger which provides traditional features, e.g. a variable watcher, breakpoints and line-by-line execution. J-Sprint [11] provides information about what parts of a program consume the most of execution time and memory. JProfiler [12], targeted at JEE and JSE applications, provides information on CPU and memory usage, thread profiling and VM. Its visualization tool shows the object references chain, execution control flow, thread hierarchy and general information about JVM using special displays.

All these tools have similar features: memory,
performance, code coverage analysis, program debugging, thread deadlock detection and class instrumentation, but many of them are designed to observe a single-process Java application and do not support directly monitoring a distributed environment based on RMI middleware, except for JaViz [17], which is intended to supplement the existing performance

analysis tools with tracing client/server activities to extend Java's profiling support for distributed environments.

III. Distributed Applications

The term *distributed applications*, is used for applications that require two or more autonomous computers or processes to cooperate in order to run them. Thus, the distributed system considered in this thesis, involves three resources, processing, data and user interface. Both processing and data can be distributed over many computers. The user interface is usually local to the user so that the graphical interface, which consumes high bandwidth, does not have to be transmitted from one location to another (figure 1).

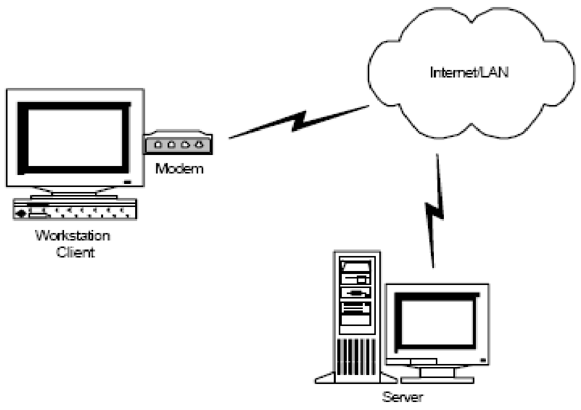


Figure 1: A Typical Distributed Application Scenario

In distributed computing, the computer network is used to support the execution of program units, called processes that cooperate with one another to work towards a common goal. This approach has become popular due to a number of developments like:

▪ Increase in the number of personal computers

▪ Low cost of establishing computer networks with the advancement of technology

▪ Computer manufacturers now offer networking software as a part of the basic operating system

▪ Computer networks are now an established way of disseminating information

The modern client/server model uses **proxy objects** for server and client respectively. The client calls the proxy, making a regular method call. The client proxy contacts the server. Similarly, a second proxy object on the server communicates with the client proxy, and it makes regular calls to the server object.

Methods of proxies' communications: There are three different methods with which proxies communicate with each other.

1. **RMI**, the Java Remote Method Invocation technology, supports method calls between distributed Java objects.

2. **CORBA**, the Common Object Request Broker Architecture, supports method calls between objects of any programming language. CORBA uses the Internet Inter-ORB Protocol or IIOP to communicate between objects.

3. **SOAP**, the Simple Object Access Protocol, is also programming – language neutral. However, SOAP uses an XML-based transmission format

IV. Remote Method Invocation with JDBC

RMI allows a Java object that executes on one machine to invoke a method of the Java object that executes on another machine. This is an important feature, because it allows building distributed application. All the RMI classes are available in jave.rmi package. To use different classes of this package we must import the java.rmi package in the beginning of the Java program.

One main application where RMI is used client/server. The server receives requests from a client, processes it & returns the result. For example, the client seeking product information can query a Ware House object on the server. It calls a remote method, *find*, which has one parameter: a Customer object. The *find* method returns an object to the client: the Product object (Figure 2).

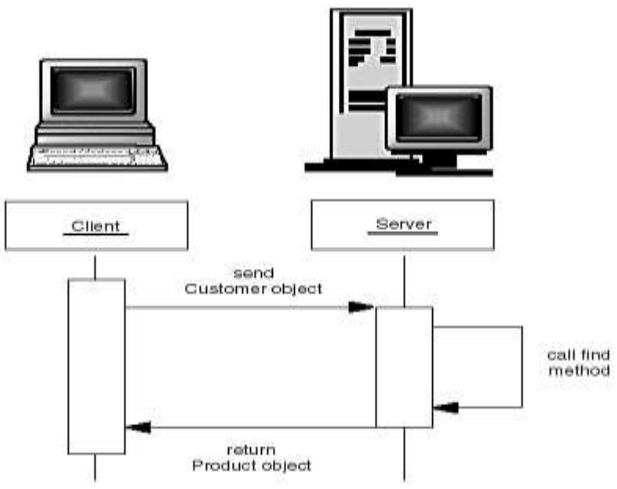


Figure 2: RMI using Client & Server Object

In RMI terminology, the object whose method makes the remote call is called the *client object*. The remote object is called the *server object*.

Following steps are required for creating RMI object:

i. Creating the Remote Interface

The RMI process begins with an interface that defines the methods accessible to the RMI client. As with all other abstract classes, the interface serves only to define the remote object's methods and parameters; they do not contain any actual programming logic.

ii. Creating the Remote Object

After the creation of the remote interface, we continue by building the actual remote object. The remote object implements the interface and incorporates the program code that will be run when its methods are called. This object performs the actual execution process, or invocation, of the RMI.

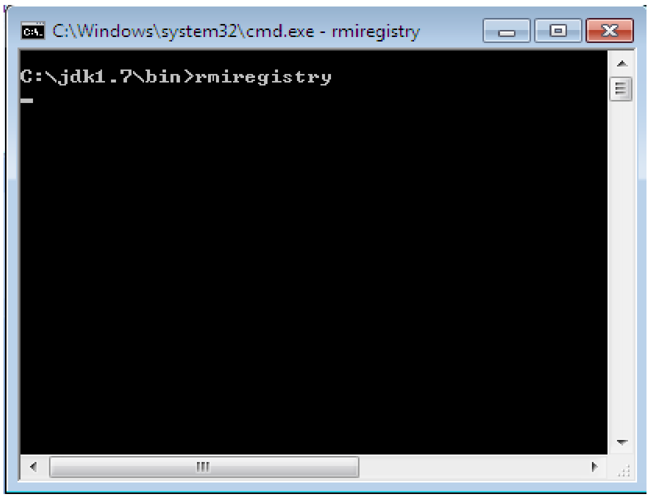
iii. Creating a RMI Client

The remote object that we just created will be used within a background process running on the Web server. It represents the "server" side of the "distributed" computing model. The "client" side can be constructed either as a Java application or as an applet.

iv. Create a Registration Program (server)

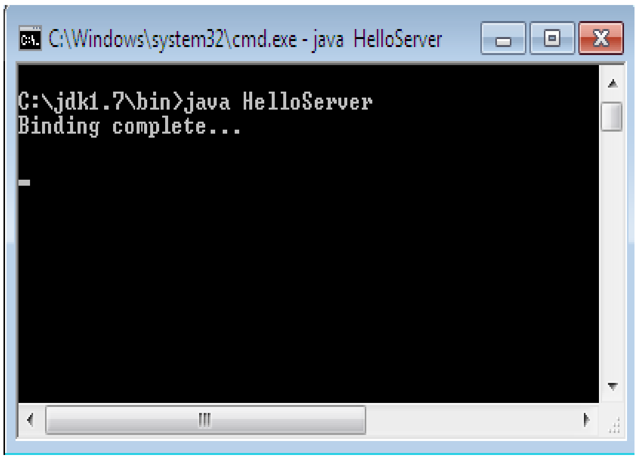
We need to bind the remote object to the RMI registry.

Result of above implementation is shown in figure 3, 4 & 5 below:



```
C:\jdk1.7\bin>rmiregistry
_
```

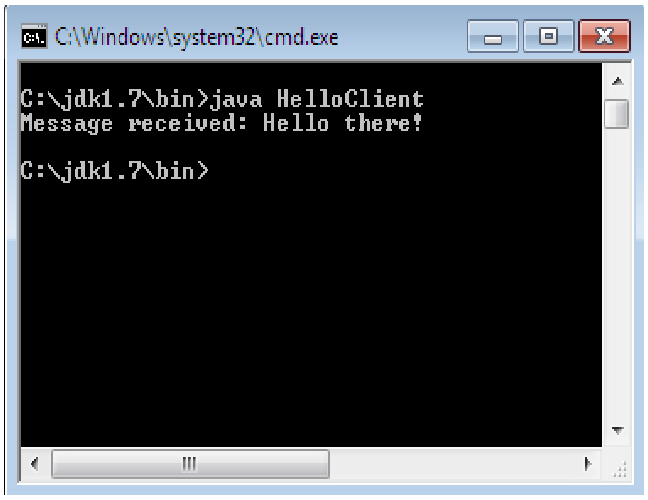
Figure 3: Starting the RMI registry.



```
C:\jdk1.7\bin>java HelloServer
Binding complete...

_
```

Figure 4: Output of Hello Server



```
C:\jdk1.7\bin>java HelloClient
Message received: Hello there!

C:\jdk1.7\bin>
```

Figure 5: Output from the *Hello Client* program

V. Common Object Request Broker Architecture with JDBC

Though RMI is a powerful mechanism for distributing and processing objects in a platform-independent manner, it has one significant drawback. it only works with objects that have been created using Java. Convenient though it might be if Java were the only language used for creating software objects, this simply is not the case in the real world.

A more generic approach to the development of distributed systems is offered by CORBA (Common Object Request Broker Architecture), which allows objects written in a variety of programming languages to be accessed by client programs which themselves may be written in a variety of programming languages. Another fundamental difference between RMI and CORBA is that, whereas RMI uses Java to define the interfaces for its objects, CORBA uses a special language called **Interface Definition Language (IDL)** to define those interfaces. In order for any ORB to provide access to software objects in a particular programming language, the ORB has to provide a *mapping* from the IDL to the target language. Mappings currently specified include ones for Java, C++, C, Smalltalk, COBOL and Ada.

At the client end of a CORBA interaction, there is a code **stub** for each method that is to be called remotely. This stub acts as a proxy (a 'stand-in') for the remote method. At the server end, there is **skeleton** code that also acts as a proxy for the required method and is used to translate the incoming method call and any parameters into their implementation-specific format, which is then used to invoke the method implementation on the associated object. Method invocation passes through the stub on the client side, then through the ORB and finally through the skeleton on the server side, where it is executed on the object. For a client and server using the same ORB, Figure 6 shows the process.

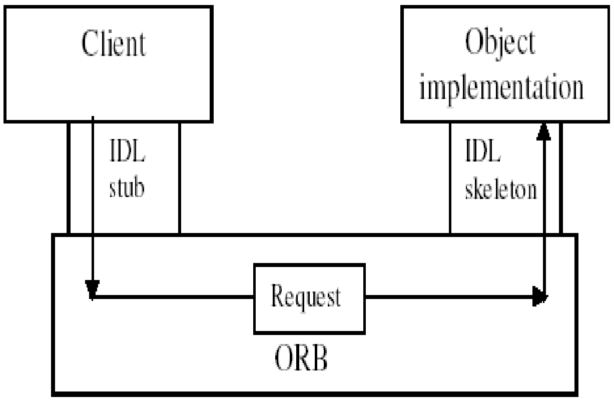


Figure 6: Remote method invocation when client and server are using the same ORB.

In order to illustrate a distributed computing model using CORBA, as well as its ability to provide persistency in Internet computing, we will build a simple application using the JDBC. CORBA allows an applet to communicate with one instance of a particular remote object that was previously registered with the CORBA Naming Service.

i. Creating the Remote Interface

All CORBA applications begin with an interface that defines the methods accessible to the CORBA client. This interface is written in the Interface Definition Language and converted to a native language. In our case, the language is Java.

ii. Creating a CORBA Server Object

After the creation of the IDL interface and the associated Java interface classes, we continue by building the remote object's implementation. The remote object extends the base skeleton object, _JIImplBase that was created by the **idltojava** tool. It also incorporates the program logic that will be run when its methods are called by a CORBA client. The server object is also referred to as a ***servant***.

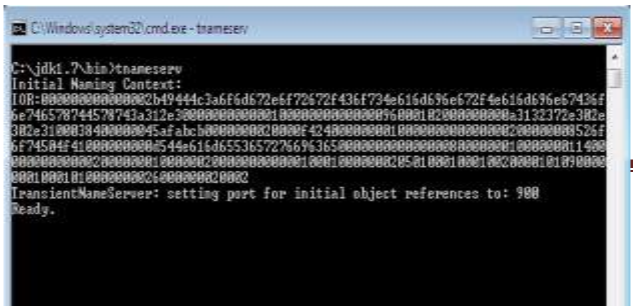
iii. Creating a CORBA Client

The servant object that we just created will be executed as a background process running on the Web server. It represents the "server" side of the "distributed" computing model. The "client" side can be constructed either as a Java application or as an applet.

iv. Create a Servant Bootstrap Program

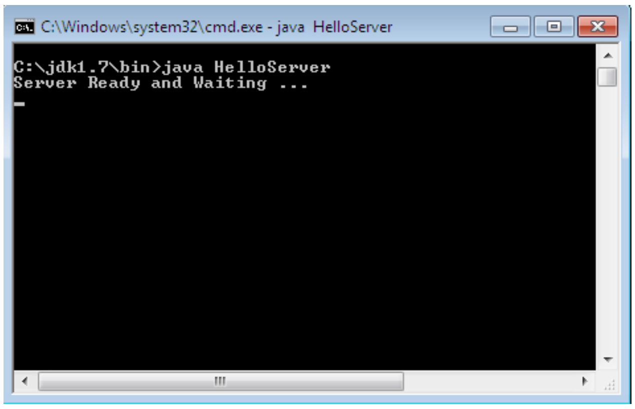
The final step in setting up our CORBA server object is to create a bootstrap program for the servant. This is done through a small program that initializes the object to the CORBA server and binds it with the Naming Service.

Result of above implementation is shown in figure 3, 7, 8 & 9 below:



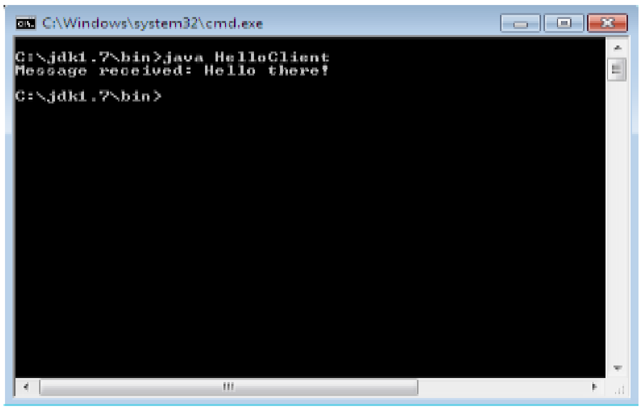
```
C:\jdk1.7\bin>tnameserv
Initial Naming Context:
IOR:000000000000002b49444c3a6f6d672e6f72672f436f734e616d696e672f4e616d696e67436f
6e746578744578743a312e30000000000000010000000000000096000102000000000a3132372e302e
302e310003840000000045afabcb000000000020000f4240000000001000000000000000200000000526f
6f74504f41000000000d544e616d6553657276696365000000000000000800000001000000011400
000000000002000000010000000200000000000010001000000002050100010001000200000101090000
0001000101000000002600000000020002
TransientNameServer: setting port for initial object references to: 900
Ready.
```

Figure 7: Starting the CORBA naming service



```
C:\jdk1.7\bin>java HelloServer
Server Ready and Waiting ...
_
```

Figure 8: The result of Server program JIStart



```
C:\jdk1.7\bin>java HelloClient
Message received: Hello there!

C:\jdk1.7\bin>
```

Figure 9: The result of Client program

VI. Simple Object Access Protocol

IBM, Lotus Development Corporation, Microsoft, Develop Mentor and User land Software developed and drafted SOAP, which is an HTTP-XML-based protocol that enables applications to communicate over the Internet, by using XML documents called *SOAP messages*.

SOAP is compatible with any object model, because it includes only functions and capabilities that are absolutely necessary for defining a communication framework. Thus, SOAP is both platform and software independent, and any programming language can implement it. SOAP supports transport using almost any conceivable protocol. SOAP binds to HTTP and follows the HTTP request–response model.

SOAP also supports any method of encoding data, which enables SOAP-based applications to send virtually any type information (e.g., images, objects, documents, etc.) in SOAP messages. A SOAP message contains an *envelope*, which describes the content, intended recipient and processing requirements of a message. The optional ***header*** *element* of a SOAP message provides processing instructions for applications that receive the message.

VII. Conclusion

During the first two decades of their existence, computer systems were highly centralized. A computer was usually placed within a large room and the information to be processed had to be taken to it. This had two major flaws, a) the concept of a single large computer doing all the work and b) the idea of users bringing work to the computer instead of bringing the computer to the user. This was followed by 'stand alone PCs' where the complete application had to be loaded on to a single machine. Each user has his/her own copy of the software. The major problems were a) sharing information and b) redundancy.

These two concepts are now being balanced by a new concept called computer networks. In computer networking a large number of separate but interconnected computers work together. An application that requires two or more computers on the network is called a network application. The client–server model is a standard model for network applications. A server is a process that is waiting to be contacted by a client process so that the server can do something for it. A client is a process that sends a request to the server.

REFERENCES

[1] Bubak M, Funika W, Metel P, Orłowski R and Wism¨uller R 2002 Proc. 4 th Int. Conf. PPAM
2001, Naleczow, Poland, LNCS 2328 315

[2] Bubak M, Funika W, Smetek M, Kilianski Z and Wism¨uller R 2003 Proc. 10 th European
PVM/MPI Users' Group Meeting, Venice, Italy, LNCS 2840 447

[3] Bubak M, Funika W, Wism¨uller R, Metel P and Orłowski R 2003 Future Generation
Computer Systems 19 651

[4] Bubak M, Funika W, Smetek M, Kilianski Z and Wism¨uller R 2004 Proc. 5 th Int. Conf.
PPAM, Czestochowa, Poland, LNCS 3019 352

[5] Funika W, Bubak M, Smetek M and Wism¨uller R 2004 Proc. Int. Conf. on Computational
Science, Cracow, Poland, LNCS 3038 472

[6] Sun Microsystems: Java Virtual Machine Profiler Interface (JVMDI),
http://java.sun.com/j2se/1.4.2/docs/guide/jpda/jvmdi-spec.html

[7] Sun Microsystems: Java Virtual Machine Profiler Interface (JVMPI), http://java.sun.com/j2se
/1.4.2/docs/guide/jvmpi/jvmpi.html

[8] The SDK Profiler, http://www.javaworld.com
/javaworld/jw-12-2001/jw-1207-hprof.html

[9] Sun's Heap Analysis Tool (HAT) for Analysing Output from hprof,
http://java.sun.com/developer/onlineTraining/Programming/JDCBook/hat bin.zip

[10] JTracer Tool, http://amslib.free.fr/

[11] Java Profiler J-Sprint, http://www.j-sprint.com/

[12] JProbe, http://java.quest.com/jprobe/jprobe.shtml

[13] JView, http://www.devstream.com/