Implementation of Socket Programming and RMI Using Simulating Environment

Hemant Kumar Srivastava, Rounak Sinha, Sumita Gupta

Abstract-- A network comprises of collection of nodes and the medium which connects them. These nodes are connected to each other for the reason of data transmission, which can be in any form i.e. messages, media, function invocation etc. The data transmission can be carried out in two modes: Synchronous transfer and Asynchronous transfer. In this expanded world where data transmission plays a vital role for communication in any firm, it becomes very necessary to use the network intelligently for effective transmission with less traffic overhead. This paper deals with two such techniques which can be selected according to the need of any organization. The two techniques are Socket Programming and Remote Method Invocation. Socket Interface is one of the fundamental technologies underlying the internet. If socket programming is implemented at hardware level, the communication between hosts will occur much faster than it is today. Socket programming can be implemented at both hardware and software. In this paper both Socket programming and RMI have been discussed in detail and comparative study is done between them on different parameters.

Many papers and reviews have been published that deals with Socket programming and Remote Method Invocation (RMI). As to start off with the survey, this paper contains the history, background and detailed description of Socket Programming and Remote method Invocation (RMI), clearly mentioning what they are. This paper will also focuses on the comparative study of both Socket Programming and RMI , their advantages and disadvantages, their function, how they work and different scenarios where they can be used.

Index-- Network programming, Socket programming. Remote Method Invocation, Distributed Object Technology, TCP/IP, UDP, Socket, Class, Methods, Client, Server, Protocols, Ports, Streams, Remote, Registry, Skeleton, Stub

1 INTRODUCTION

The term network programming refers to writing programs that execute across multiple devices, in which the devices are all connected to each other using a network. A network is all about connecting different machines together for purpose of sharing devices, jobs or communication. The most exciting aspect of Java is that it incorporates an easy to use, cross platform model for network communication. The *java.net* package of J2SE APIs contains a collection of classes and interfaces that provide low level as well as high level communication details, allowing you to write programs that focus on solving the problem at hand. The *java.net* package supports two common network protocols:

- *TCP*: TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically over the Inernet Protocol (IP), and is referred as TCP/IP.

- *UDP*: UDP stands for User Datagram Protocol, a connectionless protocol, that allows for packets of data to be transmitted between applications.[1]

Distributed object system means to combine networking and object oriented programming. A distributed system includes nodes that perform computation. A node may be a personal computer, a mainframe or any other device.[2] The main focuses of distributed object technology are:

- It performs location transparency i.e. making easy access and using remote object.
- It locates the remote class and provides a reference to them.
- Enabling remote method call includes passing parameter and returning values.
- Notify program for network failure or any other problems.

RMI is simple method for developing and deploying distributed object applications in Java environment. The java.rmi and java.rmi.server packages contain the interfaces and classes that define the functionality of JAVA.RMI systems.

2 HISTORY, OVERVIEW AND DETAILED DISCUSSION ON SOCKET PROGRAMING:

Sockets were developed in 1981 at the University of California, Berkeley. The project was sponsored by ARPA (Advanced Research Pojects Agency) in 1980. Initially, in 1983, the sockets were referred as Berkeley Sockets and were implemented on 4.2 BSD UNIX operating system. It was also known as Berkeley Software Distribution socket API. The main objective

- *Hemant Kumar Srivastava is currently pursuing B.Tech in Computer Engineering from AmityUniversity, Noida, India, E-mail: hemant19914@gmail.com*
- *Rounak Sinha is currently pursuing B.Tech in Computer Engineering from AmityUniversity, Noida, India, E-mail:rounaks@sify.com*
- *Sumita Gupta, Assistant Professor, Amity University, Noida, India, E-mail: sumitagoyal@gmail.com*

was the transport of TCP/IP software to UNIX. In 1986, AT&T introduced the Transport Layer Interface (TLI) with socket like functionality, which was more network independent. UNIX includes both TLI and Sockets after SVR4.[3]

2.1 What Exactly Is a Socket?

The notion of a socket only allows a single computer to serve many different clients at once, as well as serving many different types of information. This is managed by the introduction of a port which is a numbered socket on a particular machine. A server process is said to "listen" to a port until a client connects to it. A server is allowed to accept multiple clients connected to the same port number, although each session is unique.[4] To manage multiple client connections, a server process must be multithreaded or must have some other means of multiplexing simultaneous I/O.

Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to a server. When the connection is made, the server creates a socket object on its end of the communication. The client and server can now communicate by writing to and reading from the socket. The java.net.Socket class represents a socket, and the java.net.ServerSocket class provides a mechanism for the server program to listen for clients and establish connections with them.

2.2 Steps for Establishing a TCP Connection Between Two Computers Using Sockets:

Step 1: The server instantiates a ServerSocket object, denoting which port number communication is to occur on.

Step 2: The server invokes the accept() method of the ServerSocket class. This method waits until a client connects to the server on the given port.

Step 3: After the server is waiting, a client instantiates a Socket object, specifying the server name and port number to connect to.

Step 4: The constructor of the Socket class attempts to connect the client to the specified server and port number. If communication is established, the client now has a Socket object capable of communicating with the server.

Step 5: On the server side, the accept() method returns a reference to a new socket on the server that is connected to the client's socket.

After the connections are established, communication can occur using I/O streams.[5] Each socket has both an OutputStream and an InputStream. The client's OutputStream is connected to the server's InputStream, and the client's InputStream is connected to the server's OutputStream.TCP is a two way communication protocol, so data can be sent across both streams at the same time.[6] There are following useful classes providing complete set of methods to implement sockets.

2.3 ServerSocket Class Methods:

The *java.net.ServerSocket* class is used by server applications to obtain a port and listen for client requests.[7] The ServerSocket class has four constructors:

1.public ServerSocket(int port) throws IOException

Attempts to create a server socket bound to the specified port. An exception occurs if the port is already bound by another application.

2.public ServerSocket(int port, int backlog) throws IOException

Similar to the previous constructor, the backlog parameter specifies how many incoming clients to store in a wait queue.[8]

3.public ServerSocket(int port, int backlog, InetAddress address) throws IOException

Similar to the previous constructor, the InetAddress parameter specifies the local IP address to bind to. The InetAddress is used for servers that may have multiple IP addresses, allowing the server to specify which of its IP addresses to accept client requests on.[7]

4.public ServerSocket() throws IOException

Creates an unbound server socket. When using this constructor, use the bind() method when you are ready to bind the server socket. If the ServerSocket constructor does not throw an exception, it means that your application has successfully bound to the specified port and is ready for client requests.[9]

Common methods of the ServerSocket class:

1.public int getLocalPort()

Returns the port that the server socket is listening on. This method is useful if 0 is passed as the port number in a contructor.

2.public Socket accept() throws IOException

Waits for an incoming client. This method blocks until either a cleint connects to server on specified port or the socket times out assuming that the time-out value has been set by setSoTimeout() method. Otherwise this method blocks indefinitely.

3.public void setSocketTimeot(int timeout)

Sets the time-out value for how long the server socket waits for a client during the accept().

4.public void bind(SocketAddress host, int backlog)

Binds the socket to specified serverand port in the SocketAddress object. It is used in case of no object contructor.

When the ServerSocket invokes accept(), the method does not return until a client connects. After a client does connect, the ServerSocket creates a new Socket on an unspecified port and returns a reference to this new Socket. A TCP connection now exists between the client and server, and communication can begin.

2.4 Socket Class Methods:

The *java.net.Socket* class represents the socket that both the client and server use to communicate with each other. The client obtains a Socket object by instantiating one, whereas the server obtains a Socket object from the return value of the accept() method.

The Socket class has five constructors that a client uses to connect to a server:

1.public Socket(String host, int port) throws UnknownHostException, IOException.

This method attempts to connect to the specified server at the specified port. If this constructor does not throw an exception, the connection is successful and the client is connected to the server.

2.public Socket(InetAddress host, int port) throws IOException

This method is identical to the previous constructor, except that the host is denoted by an InetAddress object.

3.public Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException.

Connects to the specified host and port, creating a socket on the local host at the specified address and port.

4.public Socket(InetAddress host, int port, InetAddress localAddress, int localPort) throws IOException.

This method is identical to the previous constructor, except that the host is denoted by an InetAddress object instead of a String

5.public Socket()

Creates an unconnected socket. Use the connect() method to connect this socket to a server.

When the Socket constructor returns, it does not simply instantiate a Socket object but it actually attempts to connect to the specified server and port.

Some methods of interest in the Socket class are listed here which can be invoked by both the client and server:

1.public void connect(SocketAddress host, int timeout) throws IOException

This method connects the socket to the specified host. This method is needed only when you instantiated the Socket using the no-argument constructor.

2.public InetAddress getInetAddress()

This method returns the address of the other computer that this socket is connected to.

3.public int getPort()

Returns the port the socket is bound to on the remote machine.

4.public int getLocalPort()

Returns the port the socket is bound to on the local machine.

5.public SocketAddress getRemoteSocketAddress()

Returns the address of the remote socket.

6.public InputStream getInputStream() throws IOException

Returns the input stream of the socket. The input stream is connected to the output stream of the remote socket.

7.public OutputStream getOutputStream() throws IOException

Returns the output stream of the socket. The output stream is connected to the input stream of the remote socket.

8.public void close() throws IOException

Closes the socket, which makes this Socket object no longer capable of connecting again to any server.

2.5 InetAddress Class Methods

This class represents an Internet Protocol (IP) address.Some methods which are required while doing socket programming:

1.static InetAddress getByAddress(byte[] addr)

Returns an InetAddress object given the raw IP address .

2.static InetAddress getByAddress(String host, byte[] addr)

Create an InetAddress based on the provided host name and IP address.

3.static InetAddress getByName(String host)

Determines the IP address of a host, given the host's name.

4.String getHostAddress()

Returns the IP address string in textual presentation.

5.String getHostName()

Gets the host name for this IP address.

6.static InetAddress InetAddress getLocalHost()

Returns the local host.

7.String toString()

Converts this IP address to a String.

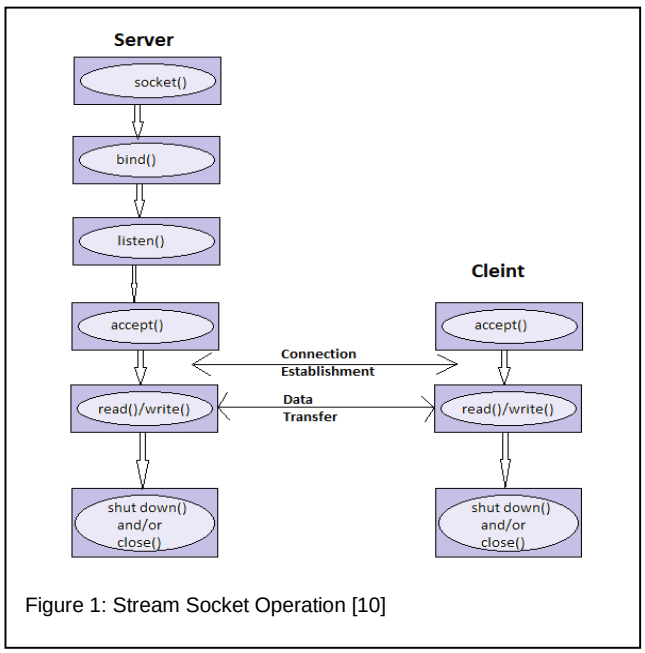

Figure 1: Stream Socket Operation [10]

Fig. 1 depicts the stream socket operation for any program.

2.6 Comparison Between TCP/IP And UDP

Table 1 shows the difference between TCP/IP & UDP on different parameters

Table 1. TCP/IP Vs UDP

S.No.	Parameter	TCP/IP	UDP
1.	Acronym	Transmission Control Protocol/ Internet Protocol	User Datagram Protocol
2.	Function	Connection bases transfer of messages	Connectionless transfer of messages
3.	Ordering of data packets	Rearranges in the order specified	No inherent order as packets are independent of each other
4.	Speed of transfer	Slower than UDP	Faster, as no error checking for packets
5.	Reliability	Guarantees that data transferred remains intact and received in same order as sent.	There is no guarantee that messages or packets sent would reach at all
6.	Header Size	20 bytes	80 bytes
7.	Error Checking	TCP does error checking	Does error checking, but no recovery options
8.	Acknowledgement	Acknowledgement Segment	No Acknowledgement
9.	Examples	HTTP, HTTPs, FTP, SMTP, Telnet, etc.	DNS, DHCP, TFTP, SNMP, RIP, VOIP, etc.

2.7 Sample Code Illustrating the Socket Programming for Chat-Server in Java

2.7.1Chat Client

```
public class ChatClient
{
        // Declarations
 ChatClient()
 {
        //codes
 };
 private void ConnectToServer()
 {
        // code
 }
 private void SendMessageToServer(String Message)
{
 }
 private void InitializeAppletComponents()
 {       // Applet Initailization
 }
 private void LoginToChat()
 {
    ConnectToServer();
 }
 public static void main(String[] args) {
   ChatClient mainFrame = new ChatClient();
 }
}
```

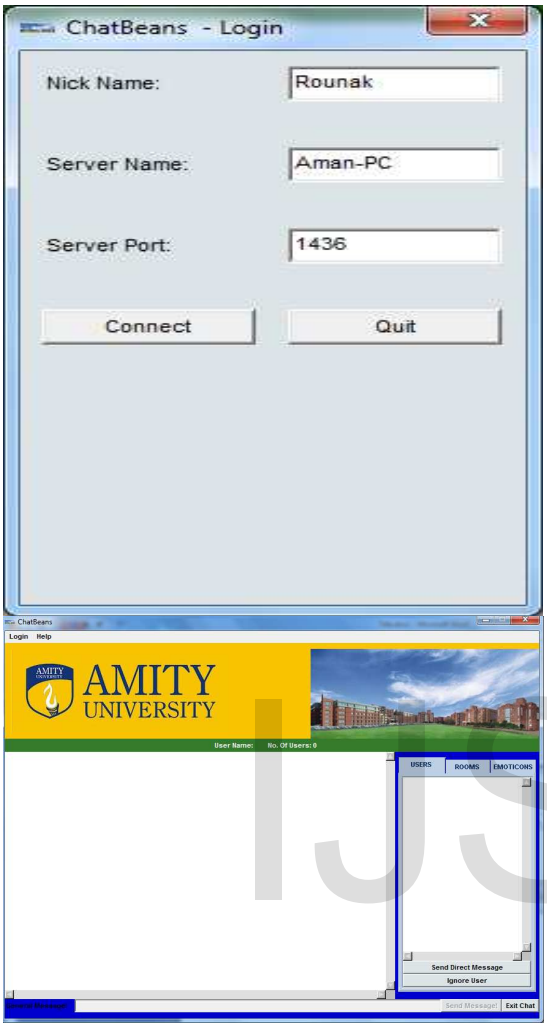

Figure 2: Login Window & Chat Window

When the code is executed, the generated output is shown in Fig. 2

2.7.2 Chat Server

Figure 3: Server interface

```
public class ChatServer
{
        //Declarations
 public ChatServer()
{
        //Codes
 };
  }
 private void SendMessageToClient
 (Socket clientsocket, String message)
 {
        //Codes
 }
 public static void main(String[] args) {
    ChatServer mainFrame = new ChatServer();
     mainFrame.setVisible(true);
 }
 }
```

When the code on server end is executed, the generated output is shown in Fig. 3

3 HISTORY, OVERVIEW AND DETAILED DISCUSSION ON REMOTE METHOD INVOCATION (RMI)

The Java RMI-IIOP specification was created to simplify the development of CORBA applications, while preserving all major benefits. It was developed by Sun Microsystems and IBM, combining features of Java RMI technology with features of CORBA technology.[11] RMI-IIOP denotes the Java Remote Method Invocation (RMI) interface over the Internet Inter-Orb Protocol (IIOP), which delivers Common Object Request Broker Architecture (CORBA) distributed computing capabilities to the Java 2 platform. It is based on two specifications: the Java Language Mapping to OMG IDL, and CORBA/IIOP 2.3.1. [12]

3.1 What is RMI?

It is a technique to call a method or object of class from a remote location. It serves as a client server relationship.
Client Object: The object whose method makes a remote call is called client object.
Server Object: The remote object is called server object.
When a client port wants to invoke a remote method on a remote object, it actually calls an ordinary method of java program language i.e. encapsulated in a packaged object called stub. It resides on client machine, not on a

server. This packaging uses a device independent encoding the parameter is called parameter marshalling. A number of methods exist for developing applications from low level socket programming to COM, DICOM, CORBA, RMI[13]

The Java Remote Method Invocation (RMI) system allows an object running in one Java virtual machine to invoke methods on an object running in another Java virtual machine. RMI provides for remote communication between programs written in the Java programming language.

RMI applications often comprise two separate programs, a server and a client. A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects. A typical client program obtains a remote reference to one or more remote objects on a 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 to as a distributed object application.

Distributed object applications need to do the following:

- Locate remote objects. Applications can use various mechanisms to obtain references to remote objects. For example, an application can register its remote objects with RMI's simple naming facility, the RMI registry. Alternatively, an application can pass and return remote object references as part of other remote invocations.
- Communicate with remote objects. Details of communication between remote objects are handled by RMI. To the programmer, remote communication looks similar to regular Java method invocations.
- Load class definitions for objects that are passed around. Because RMI enables objects to be passed back and forth, it provides mechanisms for loading an object's class definitions as well as for transmitting an object's data.

3.2 RMI Architecture

As shown in Fig 5, the server calls the registry to associate (or bind) a name with a remote object. The client looks up the remote object by its name in the server's registry and then invokes a method on it. The illustration also shows that the RMI system uses an existing web server to load class definitions, from server to client and from client to server, for objects when needed. The RMI architecture can be well understood by Fig. 4

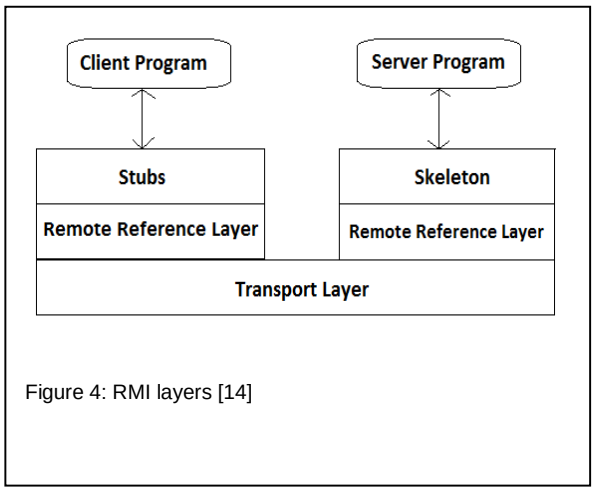

Figure 4: RMI layers [14]

3.3 RMI Registry

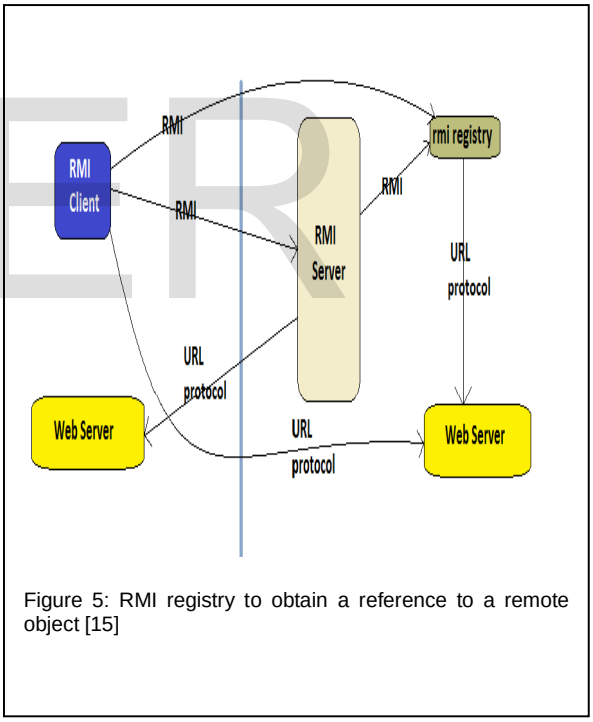

Figure 5: RMI registry to obtain a reference to a remote object [15]

3.4 Advantages of RMI

3.4.1 Object Oriented: RMI can pass full objects as arguments and return values, not just predefined data types. This means that you can pass complex types, such as a standard Java hash table object, as a single argument. In existing RPC systems you would have to have the client decompose such an object into primitive data types, ship those data

types, and the recreate a hash table on the server. RMI lets you ship objects directly across the wire with no extra client code. [16]

3.4.2 Mobile Behavior: RMI can move behavior (class implementations) from client to server and server to client. For example, you can define an interface for examining employee expense reports to see whether they conform to current company policy. When an expense report is created, an object that implements that interface can be fetched by the client from the server. When the policies change, the server will start returning a different implementation of that interface that uses the new policies. The constraints will therefore be checked on the client side-providing faster feedback to the user and less load on the server-without installing any new software on user's system. This gives you maximal flexibility, since changing policies requires you to write only one new Java class and install it once on the server host.

3.4.3 Design Patterns: Passing objects lets you use the full power of object oriented technology in distributed computing, such as two- and three-tier systems. When you can pass behavior, you can use object oriented design patterns in your solutions. All object oriented design patterns rely upon different behaviors for their power; without passing complete objects-both implementations and type-the benefits provided by the design patterns movement are lost.

3.4.4 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.

3.4.5 Easy to Write/Easy to Use: RMI makes it simple to write remote Java servers and Java clients that access those servers. A remote interface is an actual Java interface. A server has roughly three lines of code to declare itself a server, and otherwise is like any other Java object. This simplicity makes it easy to write servers for full-scale distributed object systems quickly, and to rapidly bring up prototypes and early versions of software for testing and evaluation. And because RMI programs are easy to write they are also easy to maintain.

3.4.6 Connects to Existing/Legacy Systems: RMI interacts with existing systems through Java's native method interface JNI. Using RMI and JNI you can write your client in Java and use your existing server implementation. When you use RMI/JNI to connect to existing servers you can rewrite any parts of you server in Java when you choose to, and get the full benefits of Java in the new code. Similarly, RMI interacts with existing relational databases using JDBC without modifying existing non-Java source that uses the databases.

3.4.7 Write Once, Run Anywhere: RMI is part of Java's "Write Once, Run Anywhere" approach. Any RMI based system is 100% portable to any Java Virtual Machine*, as is an RMI/JDBC system. If you use RMI/JNI to interact with an existing system, the code written using JNI will compile and run with any Java virtual machine.

3.4.8 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.

3.4.9 Parallel Computing: RMI is multi-threaded, allowing your servers to exploit Java threads for better concurrent processing of client requests.

3.5 Sample code illustrating the RMI

3.5.1 Interface

```
import java.rmi.*;
import java.rmi.server.*;
public interface remoInter extends Remote
{
    public String message() throws
    RemoteException;
}
```

3.5.2 Server

The sample code for server end is given and the simulation of code is shown in Fig. 6

```
import java.rmi.*;
import java.rmi.server.*;
public class remoServer extends UnicastRemoteObject
implements remoInter
{
    public remoServer() throws RemoteException
    {
            super();
     }
    public String message() throws RemoteException
    {
            return"Hello World";
    }
    public static void main(String args[])
   {
     try
   {
   remoServer r=new remoServer();
   Naming.rebind("TestServer",r);
   System.out.println("The server is ready");
   }
   catch(Exception e)
  {
  }
  }
  }
```

3.5.3 Client

The sample code for server end is given and the simulation of code is shown in Fig. 7

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

public class remoClint
{
public static void main(String args[])
{
try{
remoInter s=(remoInter)
Naming.lookup("TestServer");
System.out.println(s.message());
}
catch(Exception e)
{
}
}
}
```

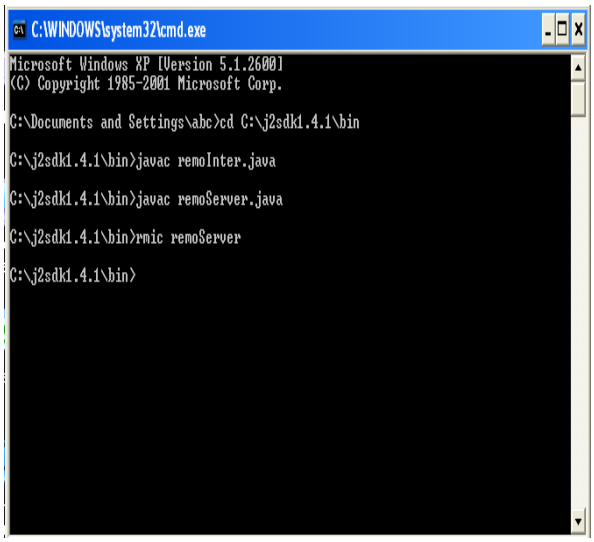

Figure 6: RMI Sever

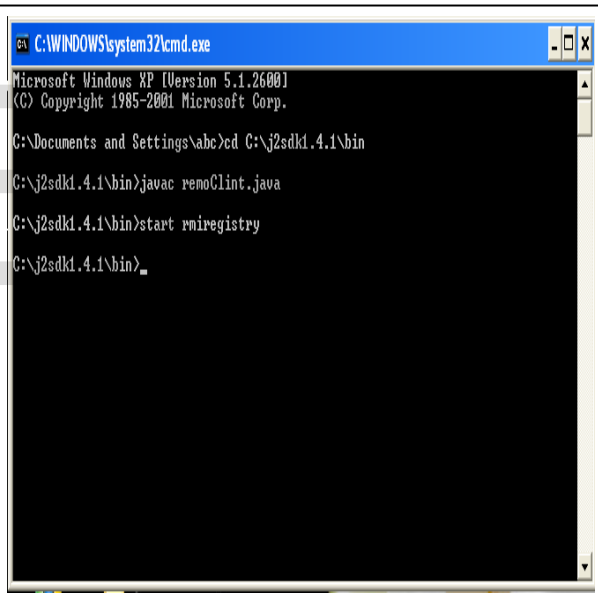

Figure 7: RMI Client

4 MY OBSERVATIONS ON SOCKET PROGRAMMING AND RMI

RMI enables a higher level of abstraction in programming. It hides the details of socket server, socket, connection, and sending or receiving data. It even implements a multithreading server under the hood, whereas with socket-level programming you have to explicitly implement threads for handling multiple

clients. The socket API is closely related to the operating system, and hence has less execution overhead. For applications which require high performance, this may be a consideration.

RMI clients can directly invoke the server method, whereas socket-level programming is limited to passing values. Socket-level programming is very primitive.

As an analogy, socket-level programming is like programming in assembly language, while RMI programming is like programming in a high-level language.

Sockets seem easier to control, where something like CORBA could take a lot of time to learn. RMI would limit you to client to needing to be another JAVA application. Programming games, chat programs, streaming media, file transfer use sockets and read/write bytes. SOCKETS have tighter control over what is sent, you can optimize the streams, by buffering, or compressing date. Reasons for using Socket programming over RMI:

1. RMI is Strictly Java. It can't be used with other code outside Java. This can be fixed by using Socket Programming.
2. Cannot guarantee that a client will always use the same thread in consecutive calls.
3. I think it is more hackable, security needs to be monitored more closely.

In socket programming we can handle exactly which socket is being used, specifying TCP or UDP, handling all formatting of messages travelling between client and server. The best part is that it is language/platform independent. On the other hand RMI hides much of the network specific code and ports. RMI is mainly used for communication between JAVA programs. A comparative study of socket programming and RMI is shown in the Table 2.

Table 2
Comparison between RMI & Socket programming

S.No.	Parameters	Socket Programming	RMI
1.	*Abstraction*	Low level	High level
2.	*Multi-threading*	Explicit	Implicit
3.	*Platform*	Platform independent	Platform dependent, Strictly uses JAVA
4.	*Execution Overhead*	Less	More
5.	*Usage*	For passing values, messages between Client and Server	For invocation for methods, used for communication between JAVA programs
6.	*Speed*	Faster	Comparatively Slower
7.	*Best Suited*	For communication between participants, suitable for message transfer, gaming, streaming media, file transfer	For communication between applications
8.	*Handling ports*	Easy	Very Difficult
9.	*Control*	Tighter control over sent data	Flexible, can't guarantee for the usage of same thread
10.	*Requirement*	Not necessarily JAVA required on both client and server end	JAVA required at both the ends
11.	*Optimization of streams*	Yes	No
12.	*Security*	High	Moderate
13.	*Language*	Low level	High level

5 CONCLUSION AND FUTURE WORK

In this paper an attempt was made to investigate in to socket programming and remote method invocation. The advantage of Socket Programming over RMI is discussed. Though Socket programming is best suitable for communication between clients and server but advent of RMI can't be ignored. It is also equally important but for very specific tasks. Due to the constraint of length and scope of the paper, Socket programming and RMI can't be easily summarized but an effort had been made to show the comparison between them. There are works done on socket programming ever since its advent. This is proved as Windows, Macintosh and UNIX provide interoperability

with socket interface. Java is taking over socket programming however the portability can't be matched. Keeping all aspects on the table I believe socket programming has yet to evolve in its types and ways it will perform in applications. All these will require the thought to have faster and reliable transactions between the server and clients.

6 ACKNOWLEDGEMENTS

The Success of this research work would have been uncertain without the help and guidance of a dedicated group of people in our institute AMITY UNIVERSITY, Noida. We would like to express our true and sincere acknowledgements as the appreciation for their contributions, encouragement and support. The researchers also wish to express gratitude and warmest appreciation to people, who, in any way have contributed and inspired the researchers.

REFERENCES

[1] Tanenbaum, "Computer Networks", Second edition
[2] A. Raju," Advanced Java"
[3] Introduction to Sockets, web.njit.edu/~gblank/cis604/Lectures/604Sockets.ppt
[4] Q. Charatan and A. Kans, "Java in two semesters" , 2nd edition McGraw Hills publication,2006
[5] Introduction to Socket Programming, http://www.cs.rutgers.edu/~pxk/rutgers/notes/sockets/index.html
[6] J. F. Kurose and K. W Ross, "Computer Networking- A Top-Down Approach featuring the Internet" , 2nd Edition (Addison Wesley World Student Edition)
[7] Author P. Burden," Socket Programming"
[8] Author G. McMillan," Socket Programming HOWTO"
[9] Socket Concepts: http://publib.boulder.ibm.com/infocenter/iseries
[10] Oracle.com- http://docs.oracle.com/cd/E19683-01/816-5042/6mb7bck68/index.html
[11] Wikipedia.org- http://en.wikipedia.org/wiki/RMI-IIOP
[12] Java SE Core Technologies – CORBA/RMI-IIOP http://www.oracle.com/:ORACLE
[13] H. Schildt, The Complete Reference, Seventh Edition, Chap. 27
[14] dsnet.tu-plovdiv.bg
[15] Oralce.com- docs.oracle.com
[16]http://findaccountingsoftware.com/directory/rmi-corporation/rmi-advantage