Networking Basics

At the core of Java's networking support is the concept of a *socket*. At the core of Java's networking support is the concept of a *socket*. A socket identifies an endpoint in a network. The socket paradigm was part of the 4.2BSD Berkeley UNIX release in the early 1980s. Because of this, the term *Berkeley socket* is also used. Sockets are at the foundation of modern networking because a socket allows a single computer to serve many different clients at once, as well as to serve many different types of information. This is accomplished through the use 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. To manage multiple client connections, a server process must be multithreaded or have some other means of multiplexing the simultaneous I/O.

Socket communication takes place via a protocol. *Internet Protocol (IP)* is a low-level routing protocol that breaks data into small packets and sends them to an address across a network, which does not guarantee to deliver said packets to the destination. *Transmission Control Protocol* (TCP) is a higher-level protocol that manages to robustly string together these packets, sorting and retransmitting them as necessary to reliably transmit data. A third protocol, *User Datagram Protocol (UDP),* sits next to TCP and can be used directly to support fast, connectionless, unreliable transport of packets.

Once a connection has been established, a higher-level protocol ensues, which is dependent on which port you are using. TCP/IP reserves the lower 1,024 ports for specific protocols. Many of these will seem familiar to you if you have spent any time surfing the Internet. Port number 21 is for FTP; 23 is for Telnet; 25 is for e-mail; 43 is for who is; 79 is for finger; 80 is for HTTP; 119 is for netnews—and the list goes on. It is up to each protocol to determine how a client should interact with the port.

For example, HTTP is the protocol that web browsers and servers use to transfer hypertext pages and images. It is a quite simple protocol for a basic page-browsing web server. Here's how it works. When a client requests a file from an HTTP server, an action known as a *hit,* it simply sends the name of the file in a special format to a predefined port and reads back the contents of the file. The server also responds with a status code to tell the client whether or not the request can be fulfilled and why.

A key component of the Internet is the *address*. Every computer on the Internet has one. An Internet address is a number that uniquely identifies each computer on the Net. Originally, all Internet addresses consisted of 32-bit values, organized as four 8-bit values. This address type was specified by IPv4 (Internet Protocol, version 4). However, a new addressing scheme, called IPv6 (Internet Protocol, version 6) has come into play. IPv6 uses a 128-bit value to represent an address, organized into eight 16-bit chunks. Although there are several reasons for and advantages to IPv6, the main one is that it supports a much larger address space than does IPv4.

To provide backward compatibility with IPv4, the low-order 32 bits of an IPv6 address can contain a valid IPv4 address. Thus, IPv4 is upwardly compatible with IPv6. Fortunately, when

using Java, you won't normally need to worry about whether IPv4 or IPv6 addresses are used because Java handles the details for you. Just as the numbers of an IP address describe a network hierarchy, the name of an Internet address, called its *domain name,* describes a machine's location in a name space. For example, **www.osborne.com** is in the *COM* domain (reserved for U.S. commercial sites); it is called *osborne* (after the company name), and *www* identifies the server for web requests. An Internet domain name is mapped to an IP address by the *Domain Naming Service (DNS).* This enables users to work with domain names, but the Internet operates on IP addresses.

Key classes, interfaces, and exceptions in java.net package simplifying the complexity involved in creating client and server programs are:

The Classes

- ContentHandler
- DatagramPacket
- DatagramSocket
- DatagramSocketImpl
- HttpURLConnection
- InetAddress
- MulticastSocket
- ServerSocket
- Socket
- SocketImpl
- URL
- URLConnection
- URLEncoder
- URLStreamHandler

The Interfaces

- ContentHandlerFactory
- FileNameMap
- SocketImplFactory
- URLStreamHandlerFactory

Exceptions

- BindException
- ConnectException
- MalformedURLException
- NoRouteToHostException
- ProtocolException
- SocketException
- UnknownHostException
- UnknownServiceException

Topic: Socket Programming

TCP/IP Socket Programming RGPV[Dec 2014(3)]

The two key classes from the java.net package used in creation of server and client programs are:

ServerSocket RGPV[Dec 2014(7)]

Socket

A server program creates a specifi c type of socket that is used to listen for client requests (server socket),In the case of a connection request, the program creates a new socket through which it will exchange data with the client using input and output streams. The socket abstraction is very similar to the fi le concept: developers have to open a socket, perform I/O, and close it. Figure given below illustrates key steps involved in creating socket-based server and client programs.

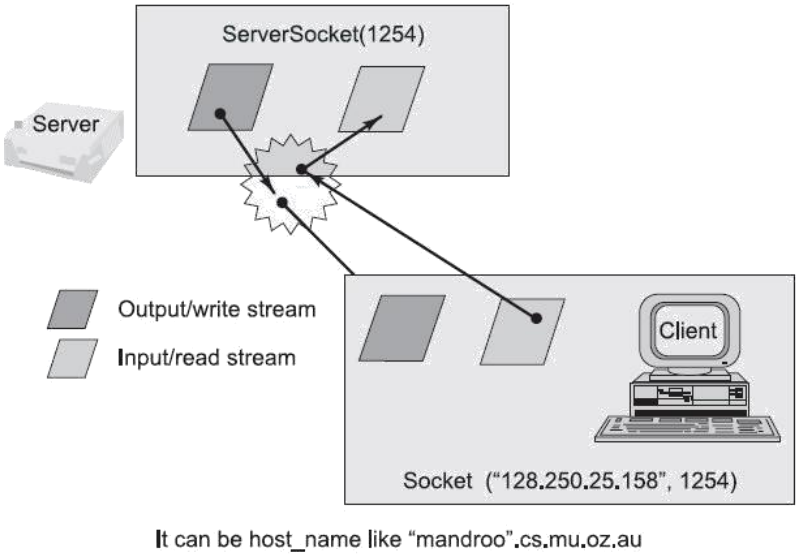


Fig. 5.1: Socket Based Server and Clients

A simple Server Program in Java The steps for creating a simple server program are:

1. Open the Server Socket: **[RGPV/Dec 2010(10)]**
ServerSocket server = new ServerSocket(PORT);
2. Wait for the Client Request:
Socket client = server.accept();
3. Create I/O streams for communicating to the client DataInputStream is =
new DataInputStream(client.getInputStream()); DataOutputStream os =
new DataOutputStream(client.getOutputStream());
4. Perform communication with client
Receive from client: String line = is.readLine();
Send to client: os.writeBytes(Hello\n);
5. Close socket:
client.close();

An example program illustrating creation of a server socket, waiting for client request, and then responding to a client that requested for connection by greeting it is given below:

Program

```
// SimpleServer.java: A simple server
program. import java.net.*;
import java.io.*;
public class SimpleServer {
public static void main(String args[]) throws IOException {
// Register service on port 1254
// Get a communication stream associated with the
socket OutputStream s1out = s1.getOutputStream();
DataOutputStream dos = new DataOutputStream (s1out);
// Send a string!
dos.writeUTF( Hi there );
// Close the connection, but not the server
socket dos.close();
s1out.close();
s1.close();
}
}
```

A simple Client Program in Java The steps for creating a simple client program are:

1. Create a Socket Object: **[RGPV/Dec 2010(10)]**

```
Socket client = new Socket(server, port_id);
```

2. Create I/O streams for communicating with the server.

```
is = new DataInputStream(client.getInputStream());
os = new DataOutputStream(client.getOutputStream());
```

3. Perform I/O or communication with the server:

Receive data from the server: String line = is.readLine();
Send data to the server: os.writeBytes(Hello\n);

4. Close the socket when done:

```
client.close();
```

An example program illustrating establishment of connection to a server and then reading a message sent by the server and displaying it on the console is given below:

```
// SimpleClient.java: A simple client program.
import java.net.*;
import java.io.*;
public class SimpleClient {
public static void main(String args[]) throws IOException {
// Open your connection to a server, at port 1254

// Get an input file handle from the socket and read the input
InputStream s1In = s1.getInputStream(); DataInputStream dis
= new DataInputStream(s1In);
```

```
String st = new String (dis.readUTF());
System.out.println(st);
// When done, just close the connection and exit
dis.close();
s1In.close();
s1.close();
}
}
```

Running Socket Programs

Compile both server and client programs and then deploy server program code on a machine which is going to act as a server and client program, which is going to act as a client. If required, both client and server programs can run on the same machine. To illustrate execution of server and client programs, let us assume that a machine called mundroo.csse.unimelb.edu.au on which we want to run a server program as indicated below:

[raj@mundroo] java SimpleServer

The client program can run on any computer in the network (LAN, WAN, or Internet) as long as there is no fi rewall between them that blocks communication. Let us say we want to run our

client program on a machine called gridbus.csse.unimelb.edu.au as follows:

[raj@gridbus] java SimpleClient The client program is just establishing a connection with the server and then waits for a message. On receiving a response message, it prints the same to the console.

The output in this case is:
Hi there

which is sent by the server program in response to a client connection request. It should be noted that once the server program execution is started, it is not possible for any other server program to run on the same port until the fi rst program which is successful using it is terminated. Port numbers are a mutually exclusive resource. They cannot be shared among different processes at the same time.

Topic: Proxy Server and TCP/IP Client Socket

Proxy Server

A server that sits between a client application, such as a Web browser, and a real server. It intercepts all requests to the real server to see if it can fulfill the requests itself. If not, it forwards the request to the real server.

Proxy servers have two main purposes:

- **Improve Performance:** Proxy servers can dramatically improve performance for groups of users. This is because it saves the results of all requests for a certain amount of time. Consider the case where both user X and user Y access the World Wide Web through a proxy server. First user X requests a certain Web page, which we'll call Page 1. Sometime later, user Y requests the same page. Instead of forwarding the request to the Web server where Page 1 resides, which can be a time-consuming operation, the proxy server simply returns the Page 1 that it already fetched for user X. Since the proxy server is often on the same network as the user, this is a much faster operation. Real proxy servers support hundreds or thousands of users. The major online services such as America Online, MSN and Yahoo, for example, employ an array of proxy servers.
- **Filter Requests:** Proxy servers can also be used to filter requests. For example, a company might use a proxy server to prevent its employees from accessing a specific set of Web sites.

TCP/IP Client Socket

There are two kinds of TCP sockets in Java. One is for servers, and the other is for clients. The **ServerSocket** class is designed to be a listener, which waits for clients to connect before doing anything. Thus, **ServerSocket** is for servers. The **Socket** class is for clients. It is designed to connect to server sockets and initiate protocol exchanges. Because client sockets are the most commonly used by Java applications, they are examined here.

The creation of a **Socket** object implicitly establishes a connection between the client and server. There are no methods or constructors that explicitly expose the details of establishing that connection. Here are two constructors used to create client sockets:

Socket(String *hostName*, int *port*) throws UnknownHostException, IOException	Creates a socket connected to the named host and port.
Socket(InetAddress *ipAddress*, int *port*) throws IOException	Creates a socket using a preexisting InetAddress object and a port.

Socket defines several instance methods. For example, a **Socket** can be examined at any time for the address and port information associated with it, by use of the following methods:

InetAddress getInetAddress( )	Returns the **InetAddress** associated with the **Socket** object. It returns null if the socket is not connected.
int getPort( )	Returns the remote port to which the invoking **Socket** object is connected. It returns 0 if the socket is not connected.
int getLocalPort( )	Returns the local port to which the invoking **Socket** object is bound. It returns –1 if the socket is not bound.

You can gain access to the input and output streams associated with a **Socket** by use of the **getInputStream( )** and **getOuptutStream( )** methods, as shown here. Each can throw an **IOException** if the socket has been invalidated by a loss of connection. These streams are used exactly like the I/O streams described in Chapter 19 to send and receive data.

InputStream getInputStream( ) throws IOException	Returns the **InputStream** associated with the invoking socket.
OutputStream getOutputStream( ) throws IOException	Returns the **OutputStream** associated with the invoking socket.

Several other methods are available, including **connect( )**, which allows you to specify a new connection; **isConnected( )**, which returns true if the socket is connected to a server; **isBound( )**,
which returns true if the socket is bound to an address; and **isClosed( )**, which returns true if the socket is closed.

The following program provides a simple **Socket** example. It opens a connection to a whois port (port 43) on the InterNIC server, sends the command-line argument down the socket, and then prints the data that is returned. InterNIC will try to look up the argument as a registered Internet domain name, and then send back the IP address and contact information for that site.

```
/ Demonstrate
Sockets. import
java.net.*; import
java.io.*; class Whois {
public static void main(String args[]) throws Exception
{ int c;
// Create a socket connected to internic.net, port 43.
Socket s = new Socket("internic.net", 43);
// Obtain input and output streams.
InputStream in = s.getInputStream();
OutputStream out = s.getOutputStream();
// Construct a request string.
String str = (args.length == 0 ? "osborne.com" : args[0]) + "\n";
// Convert to bytes.
byte buf[] =
str.getBytes(); // Send
request. out.write(buf);
// Read and display response.
while ((c = in.read()) != -1) {
System.out.print((char) c);
}
s.close();
```

}
}
If, for example, you obtained information about **osborne.com**, you'd get something similar to the following:

Whois Server Version 1.3
Domain names in the .com, .net, and .org domains can now be registered
with many different competing registrars. Go to http://www.internic.net
for detailed information.
Domain Name: OSBORNE.COM
Registrar: NETWORK SOLUTIONS, INC.
Whois Server: whois.networksolutions.com
Referral URL: http://www.networksolutions.com
Name Server: NS1.EPPG.COM
Name Server: NS2.EPPG.COM

Socket Primitives

Socket Primitives in TCP

Primitive	Meaning
SOCKET	Create a new communication end point
BIND	Attach a local address to a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Block the caller until a connection attempt arrives
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

Topic: UDP Socket Programming

UDP Socket Programming

As already said, TCP guarantees the delivery of packets and preserves their order on destination. Sometimes these features are not required and since they do not come without performance costs, it would be better to use a lighter transport protocol. This kind of service is accomplished by the UDP protocol which conveys *datagram packet s*.

Datagram packets are used to implement a connectionless packet delivery service supported by the UDP protocol. Each message is transferred from source machine to destination based on information contained within that packet. That means, each packet needs to have destination address and each packet might be routed differently, and might arrive in any order. Packet delivery is not guaranteed.
The format of datagram packet is:

Java supports datagram communication through the following classes:
DatagramPacket
DatagramSocket

The class DatagramPacket contains several constructors that can be used for creating packet object.

One of them is:
DatagramPacket(byte[] buf, int length, InetAddress address, int port);

This constructor is used for creating a datagram packet for sending packets of length length to the specifi ed port number on the specifi ed host. The message to be transmitted is indicated in the first argument.

The key methods of DatagramPacket class are:
byte[] getData()
Returns the data buffer.
int getLength()

Returns the length of the data to be sent or the length of the data received.
void setData(byte[] buf)

Sets the data buffer for this packet.
void setLength(int length) Sets the length for this packet.

The class DatagramSocket supports various methods that can be used for transmitting or receiving data a datagram over the network. The two key methods are: void send(DatagramPacket p)

Sends a datagram packet from this socket.
void receive(DatagramPacket p)

Receives a datagram packet from this socket. A simple UDP server program that waits for client's requests and then accepts the message (datagram) and sends back the same message is given below. Of course, an extended server program can manipulate client's messages/request and send a new message as a response.

Topic: URLs and Connectivity in Java

URL

The URL provides a reasonably intelligible form to uniquely identify or address information on the Internet. URLs are ubiquitous; every browser uses them to identify information on the Web. Within Java's network class library, the **URL** class provides a simple, concise API to access information across the Internet using URLs.

All URLs share the same basic format, although some variation is allowed. Here are two examples: **http://www.osborne.com/** and **http://www.osborne.com:80/index.htm**. A URL specification is based on four components. The first is the protocol to use, separated from the rest of the locator by a colon (:). Common protocols are HTTP, FTP, gopher, and file, although these days almost everything is being done via HTTP (in fact, most browsers will proceed correctly if you leave off the http:// from your URL specification).

The second component is the host name or IP address of the host to use; this is delimited on the left by double slashes (//) and on the right by a slash (/) or optionally a colon (:). The third component, the port number, is an optional parameter, delimited on the left from the host name by a colon (:) and on the right by a slash (/). (It defaults to port 80, the predefined HTTP port; thus, :80 is redundant.) The fourth part is the actual file path. Most HTTP servers will append a file named **index.html** or **index.htm** to URLs that refer directly to a directory resource. Thus, **http://www.osborne.com/** is the same as **http://www.osborne.com/ index.htm**.

Java's **URL** class has several constructors; each can throw a **MalformedURLException**. One commonly used form specifies the URL with a string that is identical to what you see displayed in a browser:

URL(String *urlSpecifier*) throws MalformedURLException

The next two forms of the constructor allow you to break up the URL into its component parts:

URL(String *protocolName*, String *hostName*, int *port*, String *path*) throws MalformedURLException
URL(String *protocolName*, String *hostName*, String *path*) throws MalformedURLException

Another frequently used constructor allows you to use an existing URL as a reference context and then create a new URL from that context. Although this sounds a little contorted, it's really quite easy and useful.

URL(URL *urlObj*, String *urlSpecifier*) throws MalformedURLException The following example creates a URL to Osborne's download page and then examines its properties:

```
// Demonstrate URL.
import java.net.*;
class URLDemo {
public static void main(String args[]) throws MalformedURLException {
```

```
URL hp = new URL("http://www.osborne.com/downloads");
System.out.println("Protocol: " + hp.getProtocol());
System.out.println("Port: " + hp.getPort());
System.out.println("Host: " + hp.getHost());
System.out.println("File: " + hp.getFile());
System.out.println("Ext:" + hp.toExternalForm()); }

}
```

When you run this, you will get the following output:

```
Protocol: http
Port: -1
Host: www.osborne
File: /downloads
Ext:http://www.osborne/downloads
```

Notice that the port is **–**1; this means that a port was not explicitly set. Given a **URL** object, you can retrieve the data associated with it. To access the actual bits or content information of a **URL**, create a **URLConnection** object from it, using its **openConnection( )** method, like this:

```
urlc = url.openConnection()
```

openConnection( ) has the following general form:

```
URLConnection openConnection( ) throws IOException
```

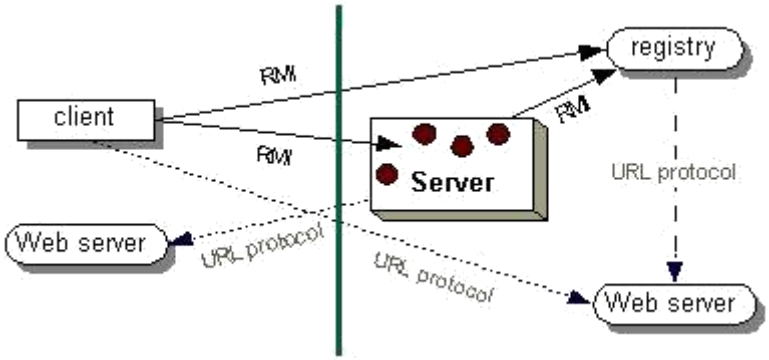
It returns a **URLConnection** object associated with the invoking **URL** object. Notice that it may throw an **IOException**.

Topic: Remote Method Invocation

Remote Method Invocation(RMI)

RMI Provides a distributed object capability for Java applications l Allows a Java method to obtain a reference to a remote object and invoke methods of the remote object nearly as easily as if the remote object existed locally. The remote object can be in another JVM on the same host or on different hosts across the network. It uses object serialization to marshal and unmarshal method arguments. It supports the dynamic downloading of required class files across the network.

RMI Application



RMI Stubs And Skeletons

RMI uses stub and skeleton objects to provide the connection between the client and the remote object. A *stub* is a *proxy* for a remote object which is responsible for forwarding method invocations from the client to the server where the actual remote object implementation resides. A client's reference to a remote object, therefore, is actually a reference to a local stub. The client has a local copy of the stub object. A *skeleton* is a server-side object which contains a method that dispatches calls to the actual remote object implementation. A remote object has an associated local skeleton object to dispatch remote calls to it.

Note: Java 2 (JDK1.2) does not require an explicit skeleton class. The skeleton object is automatically provided on the server side.

A method can get a reference to a remote object by

- looking up the remote object in some directory service. RMI provides a simple directory service called the RMI registry for this purpose.
- by receiving the remote object reference as a method argument or return value

Developing RMI Application

An object becomes remote-enabled by implementing a remote interface, which has these characteristics:
A remote interface extends the interface java.rmi.Remote.

Each method of the interface declares java.rmi.RemoteException in its throws clause, in addition to any application-specific exceptions

Steps To Develop An RMI Application

1. Design and implement the components of your distributed application
2. Define the remote interface(s)
3. Implement the remote object(s)
4. Implement the client(s)
5. Compile sources and generate stubs (and skeletons)
6. Make required classes network accessible
7. Run the application

Example-One

The classic Hello, World Example using RMI!

l First, define the desired remote interface:

```
import java.rmi.*;
/**
* Hello
Interface. */
public interface IHello extends Remote {
public String sayHello() throws RemoteException;
}
```

A class that implements this remote interface can be used as a remote object. Clients can remotely invoke the sayHello() method which will return the string Hello, World to the client.

Next, provide an implementation of the remote object. We'll implement the remote object as a server.

- The remote object server implementation should:
- Declare the remote interfaces being implemented
- Define the constructor for the remote object
- Provide an implementation for each remote method in the remote interfaces
- Create and install a security manager
- Create one or more instances of a remote object
- Register at least one of the remote objects with the RMI remote object registry (or some other naming service), for bootstrapping purposes.

To make things simple, our remote object implementation will extend java.rmi.server.UnicastRemoteObject. This class provides for the exporting of a remote object by listening for incoming calls to the remote object on an anonymous port.

Here's the server for our remote object:

```
import java.rmi.*;
import java.rmi.server.*;
// Hello Server.
public class HelloServer extends UnicastRemoteObject
```

```
implements IHello {
private String name;
```

```
public HelloServer(String name) throws RemoteException {
super();
this.name = name;
}
public String sayHello() {return "Hello, World!";}
public static void main(String[] args) {
// Install a security manager!
System.setSecurityManager(new
RMISecurityManager()); try {
// Create the remote object.

// Register the remote object as "HelloServer".
Naming.rebind("rmi://serverhost/HelloServer", obj);
System.out.println("HelloServer bound in registry!");
}
catch(Exception e) {
System.out.println("HelloServer error: " + e.getMessage());
e.printStackTrace();
}
}
}
```

Next, we need to write our client application:

```
import java.rmi.*;
// Hello Client.
public class HelloClient {
public static void main(String[] args) {
// Install a security manager!
System.setSecurityManager(new
RMISecurityManager()); try {
// Get a reference to the remote object.
IHello server =
(IHello)Naming.lookup("rmi://serverhost/HelloServer");
System.out.println("Bound to: " + server);
//Invoke the remote method.
System.out.println(server.sayHello());
}
catch(Exception e) {
e.printStackTrace();
}
}
}
```

Now we can compile the client and server code:

javac IHello.java
javac HelloServer.java
javac HelloClient.java

l We next use the rmic utility to generate the required stub and skeleton classes:

rmic HelloServer

l This generates the stub and skeleton classes:

HelloServer_Stub.class

HelloServer_Skel.class (Not needed in Java 2)

Our next step would be to make the class files network accessible. For the moment, let's assume that all these class files are available locally to both the client and the server via their CLASSPATH. That way we do not have to worry about dynamic class downloading over the network. We'll see in the next example how to properly handle that situation.

l The files that the client must have in its CLASSPATH are:

IHello.class

HelloClient.class

HelloServer_Stub.class

l The files that the server must have in its CLASSPATH are:

IHello.class

HelloServer.class

HelloServer_Stub.class

HelloServer_Skel.class (Not needed in Java 2)

Now, we are ready to run the application:

On the server:

Start the rmiregistry:

rmiregistry &

Start the server:

java -Djava.security.policy=policy HelloServer

On the client:

Start the client:

java -Djava.security.policy=policy HelloClient

Get this wonderful output on the client:

Hello, World!

Topic: Client Server Communication

Client-Server Model

Client - entity that makes a request for a service

Server - entity that responds to a request and provides a service.

The predominant networking protocol in use today is the Internet Protocol (IP). The main API for writing client-server programs using IP is the Berkeley socket API. The java.net package provides classes to abstract away many of the details of socket-level programming, making it simple to write client-server applications.

Client Example

```
import java.net.*;
import java.io.*;
/**
* Client Program.
* Connects to a server which converts text to uppercase.
* Server responds on port 2345.
* Server host specified on command line: java Client server_host
*/
public class Client {
public static void main(String args[]) {
Socket s;
String host;
int port = 2345;
DataInputStream is;
DataInputStream ui;
PrintStream os;
String theLine; host
= args[0];
try {
s = new Socket(host, port);
is = new
DataInputStream(s.getInputStream()); os =
new PrintStream(s.getOutputStream()); ui =
new DataInputStream(System.in);
System.out.println("Enter Data"); while(true) {
theLine = ui.readLine(); if
(theLine.equals("end")) break;
os.println(theLine);
System.out.println(is.readLine());
}
os.close();
is.close();
ui.close();
```

```
s.close();
}



catch(UnknownHostException e) {
System.out.println( Can't find " + host);
}
catch(SocketException e) {

System.out.println("Could not connect to    + host);
}
catch(IOException e) {
System.out.println(e);
}
}}
```

Server Example

```
import java.net.*;
import java.io.*;
/**
* Server Program.
* Converts incoming text to uppercase and sends converted
* text back to client.
* Accepts connection requests on port 2345.
*/
public class Server {
public static void main(String args[]) {
ServerSocket theServer;
Socket con;
PrintStream ps;
DataInputStream dis;
String input;
int port = 2345;
boolean flag = true;
try {
theServer = new ServerSocket(port);
con = theServer.accept();
dis = new DataInputStream(con.getInputStream());
ps = new PrintStream (con.getOutputStream());
while(flag == true) {
input = dis.readLine();
if ( input == null ) break;
ps.println(uppers(input));
}
con.close();
dis.close();
ps.close();
theServer.close();
}
catch(NullPointerException e){
System.out.println("NPE" + e.getMessage());
}
```

```
catch(IOException e) {
System.out.println(e);
}



}
public static String uppers(String input) {
char let;
StringBuffer sb = new StringBuffer(input);
for (int i = 0; i < sb.length(); i++) {
let = sb.charAt(i);
let = Character.toUpperCase(let);
sb.setCharAt(i,let);
}
return sb.toString();
}
}
```

Topic- RMI Services

Registry Service for RMI

The registration of the remote object must be done by the server in order for the client to look it up, is called the RMI Registry. In RMI, the client must contact an RMI registry, so that the server side application will be able to contact the client's registry which points the client in the direction of the service. The client registers the service with the registry so that it is transparent to even for the server.
The class rebind () method of java.rmi.Naming class is used to specify the port number. For example if the registry is running on a port number 3271 of an application named HelloRMIRegistry the following is the usage of the URL to reference the remote object:
Naming.rebind ("//myhost:3271/ HelloRMIRegistry ", obj);

The URL stored on the web page needs to specify the non-default port number. When the server's remote objects created by the server can include the URL from which the stub class can dynamically be downloaded to the client. The following example depicts this:

```
java -Djava.rmi.server.codebase=http://myhost/~username/codebase/
examples.ExampleRMIURL
```

where ExampleRMIURL is the name of the application.

bind an object to the registry

If an object implements the java.rmi.Remote interface, an object is to be bound to registry context. Each registry context implements the Referenceable interface.
The object factory is implemented by the RegistryContextFactory which converts the registry references into the corresponding registry contexts or remote objects. To construct the registry constructs, the URL of the registry must be determined. In this way the remote objects will be bounded with registry contexts.

The methods of registering and gaining access to the Remote Object

The methods of remote objects are to be invoked by implementing the java.rmi.Remote interface.

Methods:

bind(): binds the specified name to the remote object. The name parameter of this method should be in an URL format.

unbind(): Destroys the binding for a specific name of a remote method in the registry

rebind(): Binds again the specified name to the remote object. The current binding will be replaced by rebinding.

list(): Returns the names of the names that were bound to the registry in an array form. These names are in the form of URL-formatted string.
lookup(): A stub, a reference will be returned for the remote object which is related with a specified name.