© 2014 IJIRT | Volume 1 Issue 6 | ISSN : 2349-6002

SERVER/CLIENT NETWORKING AT JAVA PLATFORM

Vibhu Chinmay , Shubham Sachdeva

Student (B.tech5th sem) Department of Electronics and Computers Engineering

Dronacharya College of Engineering, Gurgaon-123506, India

Abstract: **Socket programming furnish the communication mechanism between the two systems using TCP. A client program itself makes a socket on its end of the communication and tries to connect that socket to a server. When the connection is made, the server creates a object of socket on its end of the communication. The client server can now easily 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 can make communication easily with them.**

I. INTRODUCTION

1. NETWORKING

1.1. Socket Overview

A socket is one end-point of a two-way communication link between two programs running on the network.

A server application normally listens to a specific port waiting for connection requests from a client. When a connection request arrives, the client and the server establish a dedicated connection over which they can communicate. During the connection process, the client is assigned a local port number, and binds a socket to it. The client talks to the server by writing to the socket and gets information from the server by reading from it. Similarly, the server gets a new local port number (it needs a new port number so that it can continue to listen for connection requests on the original port). The server also binds a socket to its local port and communicates with the client by reading from and writing to it.

The client and the server must agree on a protocol--that is, they must agree on the language of the information transferred back and forth through the socket.

1.2. Client or Server

You often hear the term client/server mentioned in the context of networking. It seems complicated when you read about it in corporate marketing statements, but it is actually quite simple. The **client–server model** of computing is a distributed application that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients. Often clients and servers communicate over a computer network on separate hardware, but both client and server may reside in the same system. A server machine is a host that is running one or more server programs which share their resources with clients. A client does not share any of its resources, but requests a server's content or service function. Clients therefore initiate communication sessions with servers which await incoming requests. The power grid of the house is the server, and the lamp is a power client. The server is a permanently available resource, while the client is free to "unplug" after it is has been served. In Berkeley sockets, the notion of a socket allows a single computer to serve many different clients at once, as well as serving many different types of information. This feat 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. To manage multiple client connections, a server process must be multithreaded or have some other means of multiplexing the simultaneous I/O. Here's an example of a client requesting a single file, /index.html, and the server replying that it has successfully found the file and is sending it to the client:

Server	**Client**
Listens to port 80.	Connects to port 80.
Accepts the connection.	Writes “GET /index.html HTTP/1.0\n\n”.
Reads up until the second end-of-line (\n).	
Sees that GET is a known command and that HTTP/1.0 is a valid protocol version.	
Reads a local file called /index.html.	
Writes “HTTP/1.0 200 OK\n\n”.	“200” means “here comes the file.”
Copies the contents of the file into the socket.	Reads the contents of the file and Displays it.
Hangs up.	Hangs up.

Obviously, the HTTP protocol is much more complicated than this example shows, but this is an actual transaction that you could have with any web server near you.

1.3. Proxy Servers

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 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

2.1. The Networking Classes & Interfaces

The classes contained in the java.net package are listed here:

Authenticator (Java 2)	InetSocketAddress (Java 2, v1.4)	SocketImpl
ContentHandler	JarURLConnection (Java 2)	SocketPermission
DatagramPacket	MulticastSocket	URI (Java 2, v1.4)
DatagramSocket	NetPermission	URLClassLoader (Java 2)
DatagramSocketImpl	NetworkInterface (Java 2, v1.4)	URL
HttpURLConnection	PasswordAuthentication (Java 2)	URLConnection
InetAddress	ServerSocket	URLDecoder (Java 2)
Inet4Address (Java 2, v1.4)	Socket	URLEncoder
Inet6Address (Java 2, v1.4)	SocketAddress (Java 2, v1.4)	URLStreamHandler

Some of these are to support the new IPv6 addressing scheme. Others provide some added flexibility to the original **java.net** package. Java 2, version 1.4 also added functionality, such as support for the new I/O classes, to several of the preexisting networking classes. Most of the additions made by Java 2, version 1.4 are beyond the scope of this chapter, but three new classes, **Inet4Address**, **Inet6Address**, and **URI**, are briefly discussed at the end. The **java.net** package’s interfaces are listed here:

ContentHandlerFactory	SocketImplFactory	URLStreamHandlerFactory
FileNameMap	SocketOptions	DatagramSocketImplFactory (added by Java 2, v1.3)

2.2. ServerSocket Class Method

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

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.

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.

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

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.

Here are some of the common methods of the **ServerSocket** class:

public int getLocalPort()

Returns the port that the server socket is listening on. This method is useful if you passed in 0 as the port number in a constructor and let the server find a port for you.

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

public void setSoTimeout(int timeout)
Sets the time-out value for how long the server socket waits for a client during the accept().

public void bind(SocketAddress host, int backlog)
Binds the socket to the specified server and port in the SocketAddress object. Use this method if you instantiated the ServerSocket using the no-argument constructor.

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.

2.3. Socket Class Method

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:

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.

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.

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.

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

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. Notice that both the client and server have a Socket object, so these methods can be invoked by both the client and server.

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.

public InetAddress getInetAddress()
This method returns the address of the other computer that this socket is connected to.
public int getPort()
Returns the port the socket is bound to on the remote machine.
public int getLocalPort()
Returns the port the socket is bound to on the local machine.
public SocketAddress getRemoteSocketAddress()
Returns the address of the remote socket.
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.
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
public void close() throws IOException
Closes the socket, which makes this Socket object no longer capable of connecting again to any server

2.4. InetAddress Class Method

This class represents an Internet Protocol (IP) address. Here are following useful methods which you would need while doing socket programming:
static InetAddress getByAddress(byte[] addr)
Returns an InetAddress object given the raw IP address.
static InetAddress getByAddress(String host, byte[] addr)
Create an InetAddress based on the provided host name and IP address.
static InetAddress getByName(String host)
Determines the IP address of a host, given the host's name.
String getHostAddress()
Returns the IP address string in textual presentation.
String getHostName()
Gets the host name for this IP address.
static InetAddress InetAddress getLocalHost()
Returns the local host.
String toString()
Converts this IP address to a String.
byte[] getAddress( )
Internet addresses are looked up in a series of hierarchically cached servers. That means that your local computer might know a particular name-to-IP-address mapping automatically, such as for itself and nearby servers. For other names, it may ask a local DNS server for IP address information. If that server

doesn't have a particular address, it can go to a remote site and ask for it. This can continue all the way up to the root server, called InterNIC (internic.net). This process might take a long time, so it is wise to structure your code so that you cache IP address information locally rather than look it up repeatedly.

3. TCP/IP SOCKETS

3.1. TCP/IP Client Sockets

TCP/IP sockets are used to implement reliable, bidirectional, persistent, point-to- point, stream-based connections between hosts on the Internet. A socket can be used to connect Java's I/O system to other programs that may reside either on the local machine or on any other machine on the Internet. 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. The **Socket** class is designed to connect to server sockets and initiate protocol exchanges. 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 ofestablishing that connection. Here are two constructors used to create client sockets:

Socket(String *hostName*, int *port*) Creates a socket connecting the local host to the Named host and port; can throw an **UnknownHostException** or an **IOException**. **Socket(InetAddress ipAddress, int port)** Creates a socket using a preexisting **netAddress** object and a port; can throw an **IOException**. 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. **nt getPort( )** Returns the remote port to which this **Socket** object is connected. **int getLocalPort( )** Returns the local port to which this **Socket** object is connected.

Once the **Socket** object has been created, it can also be examined to gain access to the input and output streams associated with it. Each of these methods can throw an **IOException** if the sockets have been invalidated by a loss of connection on the Net. These streams are used exactly like the I/O streams to send and receive data.

InputStream getInputStream( ) Returns the **InputStream** associated with the invoking socket.
OutputStream getOutputStream( ) Returns the **OutputStream** associated with the invoking socket.
THE JAVA LIBRARY Java 2, version 1.4 added the **getChannel( )** method to **Socket**. This method returns a channel connected to the **Socket** object. Channels are used by the new I/O classes contained in **java.nio.**

3.2. TCP/IP Server Sockets

The **ServerSocket** class is used to create servers that listen for either local or remote client programs to connect to them on published ports. Since the Web is driving most of the activity on the Internet, this section develops an operational web (http) server. **ServerSocket**s are quite different from normal **Socket**s. When you create a **ServerSocket**, it will register itself with the system as having an interest in client connections. The constructors for **ServerSocket** reflect the port number that you wish to accept connections on and, optionally, how long you want the queue for said port to be. The queuelength tells the system how many client connections it can leave pending before it should simply refuse connections. The default is 50. The constructors might throw an **IOException** under adverse conditions. Here are the constructors:

ServerSocket(int *port*)	Creates server socket on the specified port with a queue length of 50. Creates a server socket on the specified port with

a maximum queue length of *maxQueue.*
ServerSocket(int *port*, int *maxQueue*, InetAddress *localAddress*)
Creates a server socket on the specified port with a maximum queue length of maxqueue. On a multihomed host,localaddress specifies the IP address to which this socket binds.

ServerSocket has a method called**accept( )**, which is a blocking call that will wait for a client to initiate communications, and then return with a normal

Socket that is then used for communication with the client. Java 2, version 1.4 added the**getChannel( )** method to **ServerSocket**. This method returns achannel connected to the **ServerSocket** object. Channels are used by the new I/O classes contained in **java.nio.**

4. EXAMPLES

4.1. Socket Client Example

The following Greeting Client is a client program that connects to a server by using a socket and sends a greeting, and then waits for a response.

```
// File Name GreetingClient.java

import java.net.*;
import java.io.*;

publicclassGreetingClient
{
publicstaticvoid main(String[] args)
{
String serverName =args[0];
int port =Integer.parseInt(args[1]);
try
{
System.out.println("Connecting to "+ serverName
+" on port "+ port);
Socket client =newSocket(serverName, port);
System.out.println("Just connected to "
+client.getRemoteSocketAddress());
OutputStream outToServer
=client.getOutputStream();
DataOutputStreamout=
newDataOutputStream(outToServer);

out.writeUTF("Hello from "
+client.getLocalSocketAddress());
InputStream inFromServer =client.getInputStream();
DataInputStreamin=
newDataInputStream(inFromServer);
System.out.println("Server says "+in.readUTF());
client.close();
}catch(IOException e)
{
e.printStackTrace();
}
}
}
```

4.2. Socket Server Example

The following Greeting Server program is an example of a server application that uses the Socket class to listen for clients on a port number specified by a command-line argument:

```
// File Name GreetingServer.java
import java.net.*;
import java.io.*;

publicclassGreetingServerextendsThread
{
privateServerSocket serverSocket;

publicGreetingServer(int port)throwsIOException
{
serverSocket=newServerSocket(port);
serverSocket.setSoTimeout(10000);
}

publicvoid run()
{
while(true)
{
try
{
System.out.println("Waiting for client on port "+
serverSocket.getLocalPort()+"...");
Socket server =serverSocket.accept();
System.out.println("Just connected to "
+server.getRemoteSocketAddress());
DataInputStreamin=
newDataInputStream(server.getInputStream());
System.out.println(in.readUTF());
DataOutputStreamout=
newDataOutputStream(server.getOutputStream());
out.writeUTF("Thank you for connecting to "
+server.getLocalSocketAddress()+"\nGoodbye!");
server.close();
}catch(SocketTimeoutException s)
{
System.out.println("Socket timed out!");
break;
}catch(IOException e)
{
e.printStackTrace();
break;
}
}
}
publicstaticvoid main(String[] args)
{
int port =Integer.parseInt(args[0]);
try
{
Thread t =newGreetingServer(port);
t.start();
}catch(IOException e)
{
e.printStackTrace();
}
}
}
```

Compile client and server and then start server as follows:

```
$ javaGreetingServer6066
Waitingfor client on port 6066...
```

Check client program as follows:

```
$ javaGreetingClient localhost 6066
Connecting to localhost on port 6066
Just connected to localhost/127.0.0.1:6066
Server says Thank you for connecting to
/127.0.0.1:6066
```

REFERENCES

1.http://ijera.com/papers/Vol3_issue1/GU311299130 5.pdf
2.Mc Graw Hill "The Complete Reference: Java 2" Fifth Edition
3.http://www.tutorialspoint.com/java/java_networkin g.htm
4.http://docs.oracle.com/javase/tutorial/networking/so ckets/clientServer.html