1. Networking Basics

1.1 Sockets, Port

Understanding Socket

Socket is an object which establishes a communication link between two ports. A socket is an endpoint of a two-way communication link between two programs running on the network. Socket is bound to a port number so that the TCP layer can identify the application that Data is destined to be sent. The working mechanism of the socket is as shown in the figure below;

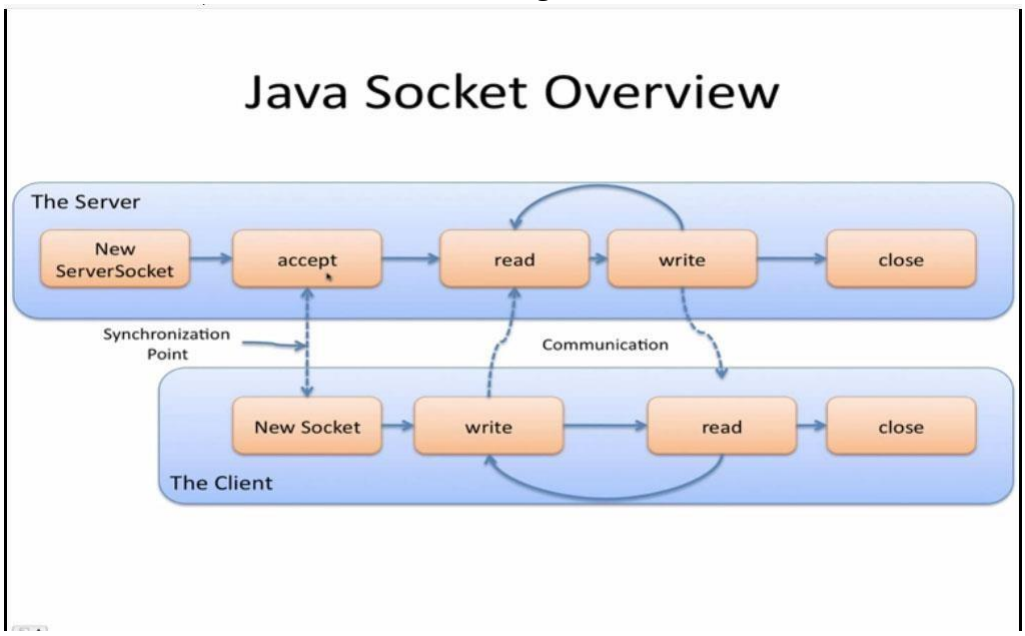

Fig: overview of java socket

Working mechanism of the socket is as shown in the table below by the example of the day/time server:

SN	Server Program	Client Program
1	Informs the user that it has started	
2	Sets up a server to listen for client on port; certain port say 1234	
3	Waits for a client to make a connection	
4		Asks the user for the IP address of the available server for example day/time server
5		After the user inputs the day/time server, connections on port 1234
6		Establishes the input stream on the connection
7	Establishes the output stream on the connection.	
8	Writes the day/time to the stream.	
9		Reads the day/time from the stream
10	Closes the connection	Closes the connection

Understanding Port:

Clients connect to servers via objects known as ports. A port serves as a channel through which several clients can exchange data with the same server or with different servers. Ports are usually specified by

numbers. Some ports are dedicated to special servers or tasks. For example, many computers reserve port number 13 for the day/time server, which allows clients to obtain the date and time. Port number 80 is reserved for a Web server, and so forth. Most computers also have hundreds or even thousands of free ports available for use by network applications.

1.2 Proxy Servers

It is the intermediary server computer that offers a computer network service to allow clients to connect with different servers. A client connects to the proxy server, then request various services such as a file, connection, web page or other resources available on different server. The Proxy provides the resource either by connecting to specified server or by serving it from a cache. If the resources have been cached before, the proxy server will return them to the client computers. If not cached, it will connect to the relevant servers and request the resources on behalf of the client computers. Then it caches resources from the remote servers which in turn responses to the clients.

The advantages of the proxy servers are as given below;

- **Improve performance:** Proxy server can improve performance for groups of users because it saves the results of all requests for certain amount of time.
- **Filter request:** It can filter the content also. So, it can be used as a filter or firewall sometimes.
- **Increased Browsing Speed:** Proxy servers can increase browsing speed due to the concept of caching.
- **Security:** It can act as an intermediary between user's computer and the internet to prevent from attack and unexpected access. It can also hide ip address of the client computer.

1.3 Internet Addressing URL

The URL allows uniquely identifying or addressing information on the Internet. 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.

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, etc. 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. The fourth part is the actual file path.

Example showing the use of URL:

```
import java.net.*;

class URLDemo {
      public static void main(String args[]) throws MalformedURLException {
            URL hp = new URL("http://localhost/project");
            System.out.println("Protocol: " + hp.getProtocol());
            System.out.println("Port: " + hp.getPort());
            System.out.println("Host: " + hp.getHost());
            System.out.println("File: " + hp.getFile());
      }
}
```

When we run this, you will get the following output:
Protocol: http
Port: -1
Host: localhost
File: /project
Notice that the port is –1; this means that a port was not explicitly set. Given a URL object, we can retrieve the data associated with it.

2. URL Connections

URLConnection is a general-purpose class for accessing the attributes of a remote resource. Once we make a connection to a remote server, we can use URLConnection to inspect the properties of the remote object before actually transporting it locally. These attributes are exposed by the HTTP protocol specification and only make sense for URL objects that are using the HTTP protocol.
URLConnection defines several methods. Some are as follows:

int getContentLength( )	Returns the size in bytes of the content associated with the resource. If the length is unavailable, –1 is returned.
String getContentType( )	Returns the type of content found in the resource. This is the value of the content-type header field. Returns null if the content type is not available.
long getDate( )	Returns the time and date of the response represented in terms of milliseconds since January 1, 1970 GMT.
long getExpiration( )	Returns the expiration time and date of the resource represented in terms of milliseconds since January 1, 1970 GMT. Zero is returned if the expiration date is unavailable.
String getHeaderField(int idx)	Returns the value of the header field at index idx (Header field indexes begin at 0). Returns null if the value of idx exceeds the number of fields.
String getHeaderField(String fieldName)	Returns the value of header field whose name is specified by fieldName. Returns null if the specified name is not found.
String getHeaderFieldKey(int idx)	Returns the header field key at index idx. (Header field indexes begin at 0.) Returns null if the value of idx exceeds the number of fields.
Map<String, List<String>> getHeaderFields( )	Returns a map that contains all of the header fields and values.
long getLastModified( )	Returns the time and date, represented in terms of milliseconds since January 1, 1970 GMT, of the last modification of the resource. Zero is returned if the last-modified date is unavailable.
InputStream getInputStream( ) throws IOException	Returns an InputStream that is linked to the resource. This stream can be used to obtain the content of the resource.

To access the actual bits or content information of a URL, create a URLConnection object from it, using its openConnection( ) method. The following example creates a URLConnection using the openConnection( ) method of a URL object .

```
import java.net.*;
class URLDemo {
      public static void main(String args[]) throws MalformedURLException {
            URL hp = new URL("http://localhost/project");
```

```
            URLConnection hpcon = hp.openConnection();
            int len = hpCon.getContentLength();
            System.out.println("Content-Length: " + len);
      }
}
```

3. java.net- Networking Classes and Interfaces

Java supports both the TCP and UDP protocol families. TCP is used for reliable stream-based I/O across the network. UDP supports a simpler, hence faster, point-to-point datagram-oriented model. The classes contained in the java.net package are as follows:

Authenticator	Inet6Address	ServerSocket
CacheRequest	InetAddress	Socket
CacheResponse	InetSocketAddress	SocketAddress
ContentHandler	InterfaceAddress (Added by Java SE 6.)	SocketImpl
CookieHandler	JarURLConnection	SocketPermission
CookieManager (Added by Java SE 6.)	MulticastSocket	URI
DatagramPacket	NetPermission	URL
DatagramSocket	NetworkInterface	URLClassLoader
DatagramSocketImpl	PasswordAuthentication	URLConnection
HttpCookie (Added by Java SE 6.)	Proxy	URLDecoder
HttpURLConnection	ProxySelector	URLEncoder
IDN (Added by Java SE 6.)	ResponseCache	URLStreamHandler
Inet4Address	SecureCacheResponse	

The **java.net** package's interfaces are as follows:

ContentHandlerFactory	DatagramSocketImplFactory	SocketOptions
CookiePolicy (Added by Java SE 6.)	FileNameMap	URLStreamHandlerFactory
CookieStore (Added by Java SE 6)	SocketImplFactory	

Example to display the IP address of current host:

```
import java.net.*
public class HostInfo{
      public static void main(String args[]){
            try{
                  InetAddress ipaddress=InetAddress.getLocalHost();
                  System.out.println("IP address:\n"+ipaddress);
```

```
            }catch(Exception e){
                  System.out.println("Unknown Host");
            }
      }
}
```

Note: We can also get the IP address of any machine by name as follows:

```
import java.net.s
import java.util.*;
public class HostInfo{
      public static void main(String args[]){
            try{
                  Scanner sc=new Scanner(System.in);
                  String name=sc.next();
                  InetAddress ipaddress=InetAddress.getByName(name);
                  System.out.println("IP address:\n"+ipaddress);
            }catch(Exception e){
                  System.out.println("Unknown Host");
            }
      }
}
```

Implementing TCP/IP based Server and Client

TCP/IP sockets are used to implement reliable, bidirectional, persistent, point-to-point, stream-based connections between hosts on the Internet. There are two kinds of TCP sockets in Java. One is for servers, and the other is for clients.

The following steps occur when establishing a TCP connection between two computers using sockets:

1. The server initializes a ServerSocket object, denoting in which port number communication is to start.
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.
3. After the server is waiting, a client instantiates a Socket object, specifying the server name and port number to connect to.
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.
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. 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. There are following useful classes providing complete set of methods to implement sockets.

TCP/IP Server Socket

The TCP/IP server socket is used to communicate with the client machine. The steps to get a server up and running are shown below:

1. Create a server socket and name the socket
2. Wait for a request to connect, a new client socket is created here
3. Read data sent from client and send data back to client
4. Close client socket
5. Loop back if not told to exit
6. Close server socket

```
import java.io.*;
import java.net.*;
public class echoServer {
	public static void main(String[] args) throws IOException {
		int portNumber = 5432;
		String serverIP = "localhost";
		try{
			ServerSocket serverSocket=new ServerSocket(portNumber) ;
			System.out.println("server started at:"+portNumber);
			Socket clientSocket=serverSocket.accept();
			System.out.println("got connection
from"+clientSocket.getInetAddress()+"port number:"+clientSocket.getPort());

			PrintWriter out=new PrintWriter(clientSocket.getOutputStream(),true);
			BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
			String inputLine;
			while((inputLine=in.readLine())!=null){
				System.out.println("Got message:"+inputLine+"from the client");
				out.println(inputLine);
			}
		}catch(IOException e){ System.out.println("exception caught by server");
		}
	}
}
```

TCP/IP Client socket

The TCP/IP client socket is used to communicate with the server. Following are the steps client needs to take in order to communicate with the server.

1. Create a socket with the server IP address
2. Connect to the server, this step also names the socket
3. Send data to the server
4. Read data returned (echoed) back from the server
5. Close the socket

```
import java.io.*;
import java.net.*;
public class echoClient {
	public static void main(String[] args){
		int portNumber = 5432;
		String hostname = "localhost";
		try{
			Socket echoSocket=new Socket(hostname,portNumber);
			System.out.println("client 1: connected to: " +
echoSocket.toString());
			PrintWriter out=new PrintWriter(echoSocket.getOutputStream(),true);
			System.out.println("client 1: Do you want to send to the echo
server");
			BufferedReader in = new BufferedReader(new
InputStreamReader(echoSocket.getInputStream()));
```

```
                    BufferedReader stdIn = new BufferedReader(new
InputStreamReader(System.in));
                    String userInput;
                    while((userInput=stdIn.readLine())!=null){
                          out.println(userInput);
                          System.out.println("do you want to send anything else:");
                          System.out.println("echo:"+in.readLine());
                    }
              }catch(IOException e){
                    System.out.println("exception caught by client");
              }
        }
}
```

5. Datagrams- Datagram Packet, Datagram Server and Client

DatagramPacket class represents a datagram packet. Datagram packets are used to implement a connectionless packet delivery service. Each message is routed from one machine to another, based solely on information contained within that packet. Multiple packets sent from one machine to another might be routed differently and might arrive in any order. Packet delivery is not guaranteed.

Datagrams are bundles of information passed between machines. Once the datagram has been released to its intended target, there is no assurance that it will arrive or even that someone will be there to catch it. Likewise, when the datagram is received, there is no assurance that it hasn't been damaged in transit or that whoever sent it is still there to receive a response.

Java implements datagrams on top of the UDP protocol by using two classes: the **DatagramPacket** object is the data container, while the **DatagramSocket** is the mechanism used to send or receive the DatagramPackets.

DatagramSocket :

It defines four public constructors. They are as shown below:

Constructor	**Meaning**
DatagramSocket( ) throws SocketException	It creates a DatagramSocket bound to any unused port on the local computer.
DatagramSocket(int port) throws SocketException	It creates a DatagramSocket bound to the port specified by port.
DatagramSocket(int port, InetAddress ipAddress) throws SocketException	It constructs a DatagramSocket bound to the specified port and InetAddress.
DatagramSocket(SocketAddress address) throws SocketException	It constructs a DatagramSocket bound to the specified SocketAddress.

SocketAddress is an abstract class that is implemented by the concrete class InetSocketAddress. InetSocketAddress encapsulates an IP address with a port number. All can throw a SocketException if an error occurs while creating the socket.

DatagramSocket defines many methods. Two of the most important are send( ) and receive( ), which are shown here:

- void send(DatagramPacket packet) throws IOException
- void receive(DatagramPacket packet) throws IOException

The send( ) method sends packet to the port specified by packet. The receive method waits for a packet to be received from the port specified by packet and returns the result. Other methods give us access to

various attributes associated with a DatagramSocket. Some of the methods are:

Methods	**Meaning**
InetAddress getInetAddress( )	If the socket is connected, then the address is returned. Otherwise, null is returned.
int getLocalPort( )	Returns the number of the local port.
int getPort( )	Returns the number of the port to which the socket is connected. It returns –1 if the socket is not connected to a port.
boolean isBound( )	Returns true if the socket is bound to an address. Returns false otherwise.
boolean isConnected( )	Returns true if the socket is connected to a server. Returns false otherwise.
void setSoTimeout(int millis) throws SocketException	Sets the time-out period to the number of milliseconds passed in millis.

DatagramPacket

DatagramPacket defines several constructors. Four are shown here:

Constructor	**Meaning**
DatagramPacket(byte data[ ], int size)	It specifies a buffer that will receive data and the size of a packet. It is used for receiving data over a DatagramSocket.
DatagramPacket(byte data[ ], int offset, int size)	It allows us to specify an offset into the buffer at which data will be stored.
DatagramPacket(byte data[ ], int size, InetAddress ipAddress, int port)	It specifies a target address and port, which are used by a DatagramSocket to determine where the data in the packet will be sent.
DatagramPacket(byte data[ ], int offset, int size, InetAddress ipAddress, int port)	It transmits packets beginning at the specified offset into the data.

DatagramPacket defines several methods. In general, the get methods are used on packets that are received and the set methods are used on packets that will be sent.

Methods	**Meaning**
InetAddress getAddress( )	Returns the address of the source (for datagrams being received) or destination (for datagrams being sent).
byte[ ] getData( )	Returns the byte array of data contained in the datagram. Mostly used to retrieve data from the datagram after it has been received.
int getLength( )	Returns the length of the valid data contained in the byte array that would be returned from the getData( ) method. This may not equal the length of the whole byte array.
int getOffset( )	Returns the starting index of the data.
int getPort( )	Returns the port number.
void setAddress(InetAddress ipAddress)	Sets the address to which a packet will be sent. The address is specified by ipAddress.

void setData(byte[ ] data)	Sets the data to data, the offset to zero, and the length to number of bytes in data.
void setData(byte[ ] data, int idx, int size)	Sets the data to data, the offset to idx, and the length to size.
void setLength(int size)	Sets the length of the packet to size.
void setPort(int port)	Sets the port to port.

A Datagram Example

The following example implements a very simple networked communications client and server. Messages are typed into the window at the server and written across the network to the client side, where they are displayed.

// demonstrate datagrams.

```
import java.net.*;
class WriteServer {
	public static int serverPort = 998;
	public static int clientPort = 999;
	public static int buffer_size = 1024;
	public static DatagramSocket ds;
	public static byte buffer[] = new byte[buffer_size];

	public static void theServer() throws Exception {
		int pos=0;
		while (true) {
			int c = System.in.read();
			switch (c) {
			case -1:
				System.out.println("Server Quits.");
				break;
			case '\r':
				break;
			case '\n':
				ds.send(new DatagramPacket(buffer,pos,
InetAddress.getLocalHost(),clientPort));
				pos=0;
				break;
			default:
				buffer[pos++] = (byte) c;
			}
		}
	}
	public static void theClient() throws Exception {
		while(true) {
			DatagramPacket p = new DatagramPacket(buffer, buffer.length);
			ds.receive(p);

			System.out.println(new String(p.getData(), 0, p.getLength()));
		}
	}
	public static void main(String args[]) throws Exception {
		if(args.length == 1) {
			ds = new DatagramSocket(serverPort);
			theServer();
```

```
            } else {
                  ds = new DatagramSocket(clientPort);
                  theClient();
            }
      }
}
```

Note: In above example the client program is given but not server program.