Network Programming

Java provides sockets API to support communication over the Internet. The classes in `java.net` package allow network applications to be built. These classes implement a platform independent interface to access the communication services provided by the TCP/IP protocol (the standard protocol used in the Internet for communication between hosts). Each host connected to the Internet implements the TCP/IP protocol. Sockets API define sockets to send/receive data to/from other hosts connected to the Internet. A socket can be seen as a door between the network applications and the TCP/IP protocol suite. Therefore, by using sockets, an application sends/receives messages to/from other applications running in any host connected to the Internet.

Each host connected to the Internet has a unique IP address and optionally a name. In order to send/receive messages to/from other applications running in the other hosts, an application should first create a socket in the local system. Since each host may be running more than one network application, each application should create at least one new socket to communicate with other applications. Each socket has an associated unique port number (an integer between `0-65535`). The host name or IP address is used to uniquely address a host in the Internet, and port number is used to address a specific application on that host. Therefore, in order to send a message to a network application, the name or address of the host running the application and the port number of the socket used by the application should be known.

There are two kinds of services provided by the TCP/IP protocol suite:

> **Connection oriented - reliable**: TCP protocol provides this service. The protocol guarantees error-free, ordered delivery of bytes.
> **Connectionless - unreliable**: UDP protocol provides this service. The protocol does not guarantee error-free, ordered delivery of messages. Some messages may be lost during its journey from sending application to receiving application and order of messages may not be preserved.

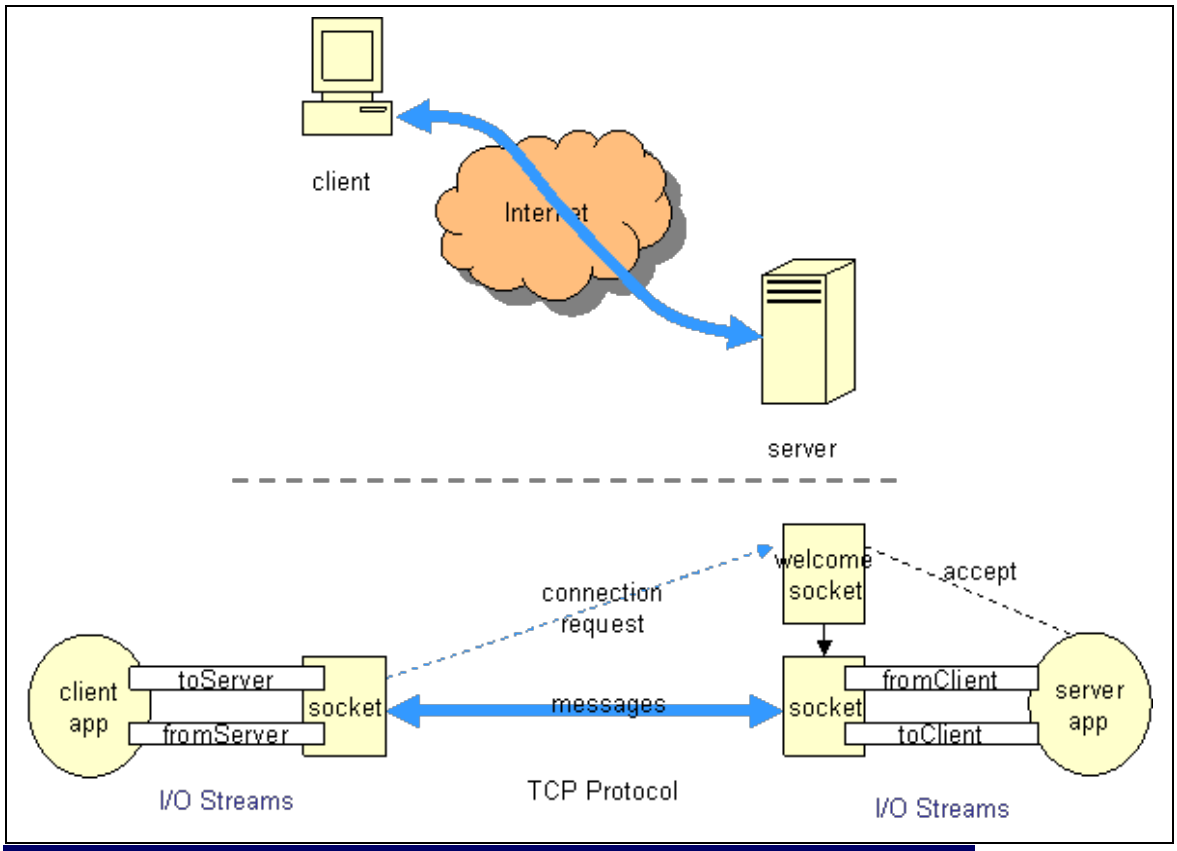
Most of the network applications use a client/server mode of interaction, where a client application contacts to the server application and requests some service from the server. Then, server responds to the client's request. Server applications usually run on a machine with fixed/known address, and accept connection requests with a socket with well-known port number. The server extracts the client's address and port number from the request and sends its reply to that address.

Reliable Communication Using TCP

TCP provides a connection oriented-reliable, full duplex (two way) byte stream delivery service to network applications. That is, it transfers bytes from one application to another reliably (without loss and corruption, while preserving byte order).

In order to communicate, a client application must first contact to the server application. Hence, the server application must first be running and must have created a `ServerSocket` that welcomes client's contact. Client contacts to the server by creating a `Socket` instance and specifying the host address and port number of the server application. Then, client establishes a connection to the server. When contacted by a client, a new socket is created by the server application and communication takes place through these sockets. This allows server application to serve multiple clients at the same time.

A network application can send/receive messages to/from a socket (and the application at the other side of the connection) using streams. After the communication completed, the sockets should be closed to free system resources.



TCP Client

Suppose that the server application runs on a machine with the address `serverAddress` (if both applications will run on the same host, `"localhost"` string can be used as the address) and waits at the port `serverPort`. In order to contact to the server, the client should first create a socket by specifying `serverAddress` and

`serverPort` as:

```
Socket clientSocket = new Socket(serverAddress, serverPort);
```

If the server accepts the connection request, above statement will return a `Socket` instance through which messages can be sent/received. In order to send and receive messages, the `Stream` interfaces provided by the `Socket` class could be used. The streams attached to the socket could be obtained as:

```
InputStream fromServer = clientSocket.getInputStream();
OutputStream toServer = clientSocket.getOutputStream();
```

If required, these streams can be wrapped within some filter streams. For example, if only single-line text messages will be sent/received, these streams could be wrapped in `BufferedReader`/`DataOutputStream` as:

```
BufferedReader fromServer = new BufferedReader(new
   InputStreamReader(clientSocket.getInputStream()));
DataOutputStream toServer = new DataOutputStream(
   clientSocket.getOutputStream);
```

The output stream's `writeBytes()` method could be used to send a request string to the server as:

```
String request = "hello world!";
toServer.writeBytes(request + "\n");
```

Note: The newline character is used in above statement to mark the end of the request message. Since TCP provides a byte-stream delivery service, message boundaries are not preserved. Therefore, special markers may be needed to delineate successive messages.

The server's response could be received by using `readLine()` method of the `fromServer` stream as:

```
String response = fromServer.readLine();
```

Note: This method call will block until a newline character is received from the server.

When finished, the socket could be closed as:

```
clientSocket.close();
```

TCP Server

Server applications usually serve more than one client at a time. Therefore, the server application should be capable of accepting connection requests from several clients and serving these clients simultaneously. For this purpose, the server first creates a welcoming socket bound to a well known port number, `serverPort`, and waits for connection requests from clients as:

```
ServerSocket welcomingSocket = new ServerSocket(serverPort);
Socket connectionSocket = welcomingSocket.accept();
```

Each call to `accept()` method blocks until a client connects to the server, and returns a `Socket` instance which will be used to receive/send messages from/to the corresponding client. In order to support multiple clients at the same time, `accept()` method could be called in a separate thread and each `connectionSocket` returned could be controlled in a new thread. This corresponds to a multi-threaded server application.

After having `connectionSocket`, two streams `toClient` and

`fromClient` could be obtained, and they can be used to send a message or receive a message from client as explained in the previous section.

Unreliable Communication Using UDP

Reliable communication is achieved with some overheads such as connection establishment delay and large average delay due to retransmissions. TCP also adjusts the speed of transmission by monitoring the congestion level in the network. However, some network applications like multimedia applications, are sensitive to delay and transmission rate. However, they can tolerate a reasonable amount of loss. For such applications, UDP is the most appropriate choice.

UDP is a no-frills, bare-bone protocol providing best-effort connectionless service. That is, some of the messages may be lost or messages arrive to destination out of order. UDP applications exchange packets of data, called datagrams. A datagram is a self contained message sent over the network. The UDP service is accessed by `DatagramSocket`s and network applications send/receive `DatagramPacket`s through the `DataGramSocket`s.

In the following sections, a simple client/server application using UDP will be explained. As in the reliable communication case, the server application runs on a host with a well known address and waits requests from a socket bound to a well known port number. However, in UDP case, there will not be a connection establishment phase, and through a single socket, it is possible to send/receive datagrams to/from multiple hosts.

UDP Client

Like TCP, UDP service is access through sockets. In order to access UDP service, a DatagramSocket instance has to be created. The `DatagramSocket` constructor optionally accepts a port number to bind the socket to. If Port number is not specified, the operating system will assign an unused port number to the new socket. Usually client applications do not specify port numbers while creating sockets. A datagram socket could be created as:

```
DatagramSocket clientSocket = new DatagramSocket();
```

It is possible to send/receive packets to/from different network applications through a single socket. Therefore, while sending a message, the destination of the message should also be specified. The messages should be encapsulated in `DatagramPacket`s to be sent through a socket. In addition, a packet should contain address and port number to deliver the message. Therefore, a simple message could be sent to an application bound to port number `port`, running on a host with name hostName as:

```
String request = "hello world!";
byte msg[] = request.getBytes();
```

```
InetAddress address = InetAddress.getByName(hostName);
DatagramPacket packet = new DatagramPacket(msg,
                            msg.length, adress, port);
clientSocket.send(packet);
```

In order to receive messages from other applications, `DatagramSocket`'s `receive()` method could be used. The `receive()` method also requires a placeholder for the arriving packet. Therefore, a new `DatagramPacket` instance should be created and given to `receive()` method as:

```
byte msg[] = new char[1024];
DatagramPacket packet = new DatagramPacket(msg, msg.length);
clientSocket.receive(packet);
```

Note: The `receive()` method blocks until a datagram arrives to the application. To avoid blocking, `DatagramSocket`'s `setSoTimeout(timeout)` method could be used. If no datagram appears in timeout duration, the method returns and `SocketTimeoutException` is raised.

UDP Server

The implementation of Server is very similar to the client's implementation. However there are two important points to ponder while implementing servers.

The first difference from the client is that the server's port number should be known by the clients. Therefore, the server port number should be specified while creating the socket as:

```
DatagramSocket serverSocket = new ServerSocket(port);
```

The second difference is that upon receiving a request from the client, the client's address and port number should be extracted from the arriving packet as:

```
InetAddress IPAddress = receivePacket.getAddress();
int port = receivePacket.getPort();
```

These values can be used to prepare and send reply to the client.