SOCKET PROGRAMMING

Tanjina Helaly

Network Programming & Protocol

- The term *network programming* refers to writing programs that execute across multiple devices (computers), in which the devices are all connected to each other using a network.
- 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.

Java .net package

- The java.net package provides support for the two common network protocols:
- **TCP**
  - TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.
- **UDP**

TCP/IP Sockets

- **Definition1:** A socket is one end-point of a two-way communication link between two programs running on the network.

- **Definition2**: Socket is the Java programming model to establish the communication link between two processes.

- **Definition3**: 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. they are examined here.

TCP/IP Sockets

- 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.
    - 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.
  - The **Socket** class is for clients. It is designed to connect to server sockets and **initiate** protocol exchanges.

How does it work

- The three things we have to learn to get the socket working are:
  - Establish the initial connection between the client and server
  - Send messages to the server/client
  - Receive messages from the server/client

Steps to establish connection between the client and server

- Step#1:
  - A server application normally listens to a specific port waiting for connection requests from a client.
    - TCP/IP reserves the lower 1,024 ports for specific well known protocols.
      - Port number 21 is for FTP;
      - 23 is for Telnet;
      - 25 is for e-mail;
      - 43 is for whois;
      - 80 is for HTTP
    - For user app we have to use port >1024
  - From Java side:
    - Server application creates a ServerSocket object, on a specific port
    - Code

```
ServerSocket serverSock = new ServerSocket(4242);
```

Steps to establish connection between the client and server

- Step#2:
  - Client knows the IP address and port number and send request.
  - 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.
  - Once the connection is established, the client communicates with the server by reading from and writing to the socket.
  - From Java side:
    - Client makes a Socket connection request to the server application using the IP address and port
    - Code:

      Socket sock = new Socket ("190.165.1.103" , 4242);
      - This is Client side socket

Steps to establish connection between the client and server

- Step#3:
  - The server accept the request and 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.
  - From Java side:
    - Server accept the Client's request and return a socket object(on a different port) on server side that knows how to communicate with the client.
    - Code

      Socket sock = serverSock.accept();
      - This is server side socket

Send messages to the server/client

- Create a OutputStream object from the Socket object. This could be done from any side (either client or server) whoever wants to send a message.
- Code: OutputStream s1Out = *sock*.getOutputStream();

  - Once you get the OutputStream object, you can use any of the class you used for writing e.g. OutputStreamWriter, BufferedWriter, DataOutputStream, Printwriter
  - Example1:

```
BufferedWriter bf = new BufferedWriter(new OutputStreamWriter(s1Out));
bf.write("Hello World0!");
```

  - Example2:

```
DataOutputStream dos = new DataOutputStream(s1Out);
dos.writeUTF("hello");
```

  - Example3:

```
PrintWriter pw = new PrintWriter(s1Out);
pw.print("hello");
pw.println("hello");
```

Receive messages from server/client

- Create a InputStream object from the Socket object. This could be done from any side (either client or server) whoever wants to send a message.

- Code:
  - InputStream s1In = *sock*.getInputStream();

  - Once you get the InputStream object, you can use any of the class you used for writing e.g. InputStreamWriter, BufferedReader, DataInputStream, Printwriter

  - Example1:

```
BufferedReader bf = new BufferedReader(new InputStreamReader(s1In));
String data = bf.readLine();
```

  - Example2:

```
DataInputStream dis = new DataInputStream(s1.getInputStream());
String data = dis.readUTF();
```

Details of some classes

Client Socket/Socket

- The creation of a **Socket** object implicitly establishes a connection between the clientand 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*) 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.

Client Socket/Socket

- **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.

Client Socket/Socket

- 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 20 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.

Server Socket

Constructors	
Constructor	**Description**
`ServerSocket()`	Creates an unbound server socket.
`ServerSocket(int port)`	Creates a server socket, bound to the specified port.
`ServerSocket(int port, int backlog)`	Creates a server socket and binds it to the specified local port number, with the specified backlog.
`ServerSocket(int port, int backlog, InetAddress bindAddr)`	Create a server with the specified port, listen backlog, and local IP address to bind to.

Server Socket

Methods	
Modifier and Type	**Method and Description**
Socket **accept**()	Listens for a connection to be made to this socket and accepts it.
void **bind**(**SocketAddress** endpoint)	Binds the ServerSocket to a specific address (IP address and port number).
InetAddress **getInetAddress**()	Returns the local address of this server socket.
int **getLocalPort**()	Returns the port number on which this socket is listening.
SocketAddress **getLocalSocketAddress**()	Returns the address of the endpoint this socket is bound to, or null if it is not bound yet.

InetAddress

- The **InetAddress** class is used to encapsulate both the **numerical IP** address and the **domain name** for that address.
- You interact with this class by using the name of an IP host, which is more convenient and understandable than its IP address.
- The **InetAddress** class hides the number inside. **InetAddress** can handle both IPv4 and IPv6 addresses.
  - 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.

InetAddress - Example

- The following example prints the addresses and names of the local machine and two Internet web sites:

```
// Demonstrate InetAddress.
import java.net.*;
class InetAddressTest
{
    public static void main(String args[]) throws UnknownHostException {
        InetAddress Address = InetAddress.getLocalHost();
        System.out.println(Address);
        Address = InetAddress.getByName("www.HerbSchildt.com");
        System.out.println(Address);
        InetAddress SW[] = InetAddress.getAllByName("www.nba.com");
        for (int i=0; i<SW.length; i++)
        System.out.println(SW[i]);
    }
}
```

InetAddress - Output

- Here is the output produced by this program. (Of course, the output you see may be slightly different.)
  - default/166.203.115.212
  - www.HerbSchildt.com/216.92.65.4
  - www.nba.com/216.66.31.161
  - www.nba.com/216.66.31.179

Example code – Client side

```
import java.io.*;
import java.net.Socket;
public class ClientExample {
    public static void main(String[] args) {
        Socket s1 = null;
        try{
            s1 = new Socket("localhost",5401);
            System.out.println("Client: Connection Extablished");
            BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(s1.getOutputStream()));
            for(int i = 0; i<5; i++){
                bw.write("Hello:"+i);
                Thread.sleep(200);
                bw.newLine();
                bw.flush();
            }
            bw.close();
            if (!s1.isClosed())
                        s1.close();
            }
        catch(IOException | InterruptedException e) {
        }
    }
}
```

Example code – Server side

```
import java.io.*;
import java.net.*;
public class ServerExample {
    public static void main(String[] args) {
        ServerSocket s = null;
        Socket s1 = null;
        try{
            s = new ServerSocket(5401);
            s1 = s.accept();
            System.out.println("Connection Extablished");
            BufferedReader br = new BufferedReader(new InputStreamReader(s1.getInputStream()));

            String inp = br.readLine();
            while (inp != null && !inp.equals("exit")) {
                System.out.println("Read:" + inp );
                inp = br.readLine();
            }
            System.out.println("Read:" + inp );
            br.close();
            if (!s1.isClosed())
                        s1.close();
        } catch(IOException e) {
        }
    }
}
```

References

- Java: Complete Reference - Chapter 22