Advanced Programming-- Java

Socket Programming

Computer Science

The Client-Server Paradigm

- Each of these applications use the *client-server* paradigm, which is roughly

1. One program, called the *server* blocks waiting for a client to connect to it
2. A client connects
3. The server and the client exchange information until they're done
4. The client and the server both close their connection

Basic Terminology 1

- Hosts:
  - Devices connected to the internet.
  - In addition to computers, can be routers printers storage devices etc.
- Internet Addresses
  - Every host on the Internet is identified by a unique, four-byte Internet Protocol (IP) address.
  - Two versions exists: IPv4 and IPv6.
  - The IPv4 version is written in *dotted quad* format like 199.1.32.90 where each byte is an unsigned integer between 0 and 255.
  - The IPv6 expands the address space to 2^{128} and is supported by Java.

Basic Terminology 2

- Ports:
  - In general a host has only one Internet address
  - This address is subdivided into 65,536 *ports* , numbered from 0-65535.
  - Ports are logical abstractions that allow one host to communicate simultaneously with many other hosts
  - Servers listen on a port.
  - Many services run on well-known ports. For example, http tends to run on port 80
  - These port numbers are reserved so you can't use them when you write your own server.
  - User-level process/services generally use port number value >= 1024.

Basic Terminology 3

- Multiple clients can be communicating with a server on a given port. Each client connection is assigned a separate *socket* on that port.
- Client applications get a port and a socket on the client machine when they connect successfully with a server.
- A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.

InetAddress

- InetAddress is Java's representation of an IP address.

- Creating an InetAddress Instance
  - InetAddress has no public contructor, so you must obtain instances via a set of static methods. To get the InetAddress instance for a domain name:

    InetAddress address = InetAddress.getByName(“yahoo.com");
  - To get the InetAddress matching a String representation of an IP address:

    InetAddress address = InetAddress.getByName("78.46.84.171");
  - Localhost (the computer the program is running on):

    InetAddress address = InetAddress.getLocalHost();
- Instances of this class are used together with UDP DatagramSockets and normal Socket's and ServerSocket's.

InetAddress Example

```
Public class InetExample{
        public static void main (String [] args){
                try{
                  System.out.println(InetAddress.getByName(“yahoo.com”));
                  System.out.println(InetAddress.getByName(“google.com”));
                  System.out.println(InetAddress.getByName(“121.52.147.9”));
                  System.out.println(InetAddress.getLocalHost());
                  System.out.println(InetAddress.getLoopBackAddress());
                }
                catch(UnknownHostException e){
                e.printStackTrace();
                }
        }
}
```

TCP

- The transport layer comprises two types of protocols:
  - TCP is a connection-oriented protocol that provides a reliable flow of data between two computers. Example applications that use such services are HTTP, FTP, and Telnet.
  - UDP is a protocol that sends independent packets of data, called *datagrams*, from one computer to another with no guarantees about arrival and sequencing. Example applications that use such services include Clock server and Ping.

TCP/IP Socket Programming

- Socket:
  - This class implements one side of a two-way connection between your Java program and another program on the network.
  - The Socket class sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program.
  - By using the Socket class instead of relying on native code, your Java programs can communicate over the network in a platform-independent fashion.

- ServerSocket
  - This class implements server sockets.
  - A server socket waits for requests to come in over the network.
  - It performs some operation based on that request, and then possibly returns a result to the requester.

Example TCP Server

1. Open the Server Socket:

   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

5. Receive from client

   String line = is.readLine();

6. Send to client:

   os.writeBytes("Hello\n");

5. Close socket:

   client.close();

TCP Server Code

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

TCP Client Example

1. Create a Socket Object:

   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();

TCP Client Example

```
import java.net.*;
import java.io.*;
public class TCPClient {
public static void main(String args[]) throws IOException {
// Open your connection to a server, at port 1254
Socket s1 = new Socket("localhost",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();}
}
```

UDP

- UDP protocol uses *packets.*
- *Connectionless*
- *Packets can arrive out of order*
- *Packet delivery is not guaranteed.*
- The format of datagram packet is:

  | Msg | length | Host | serverPort |
- Java supports datagram communication through the following classes:
- DatagramPacket
- DatagramSocket

The DatagramPacket Class

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

DatagramPacket(byte[] buf, int length, InetAddress address, int port);

- 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 methods that can be used for transmitting or receiving data a datagram over the network.
  - void send(DatagramPacket p) // Sends a datagram packet from this socket.
  - void receive(DatagramPacket p) //Receives a datagram packet from this socket.

Simple UDP Server

```
public class UDPServer{
public static void main(String args[]){
    DatagramSocket aSocket = null;
    try { int socket_no = 5556;
    aSocket = new DatagramSocket(socket_no);
    byte[] buffer = new byte[1000];
    while(true) {
    DatagramPacket request = new DatagramPacket(buffer, buffer.length);
    aSocket.receive(request);
    DatagramPacket reply = new DatagramPacket(request.getData(),
    request.getLength(),request.getAddress(),
    request.getPort());
    aSocket.send(reply); }
    }
    catch (SocketException e) { System.out.println("Socket: " +
    e.getMessage()); }
    catch (IOException e) { System.out.println("IO: " + e.getMessage()); }
    finally { if (aSocket != null) aSocket.close(); }
}}
```

UDPClient

```
public class UDPClient {
public static void main(String args[]){
DatagramSocket aSocket = null;
try {
aSocket = new DatagramSocket();
byte [] m = " This message will be echoed".getBytes();
InetAddress aHost = InetAddress.getLoopbackAddress();
int serverPort = 5556;
DatagramPacket request = new DatagramPacket(m, m.length, aHost, serverPort);
aSocket.send(request);
byte[] buffer = new byte[1000];
DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
aSocket.receive(reply);
System.out.println("Reply: " + new String(reply.getData()));
}
catch (SocketException e) { System.out.println("Socket: " + e.getMessage()); }
catch (IOException e) { System.out.println("IO: " + e.getMessage()); }
finally { if (aSocket != null) aSocket.close();}
}}
```