March 24 Lecture Transcript – (Socket Programming – part 2)

Slide 1:

The goal of today's lecture is to look into the design (i.e., coding in Java) of a simple client-server application, as outlined in the provided figure.

In particular, we will …

- Design a **client program** which is expected to:
  a) Open and connect a client-side socket with a particular (already open and waiting) server-side socket. Note here, the server (i.e., server-side socket) is expected to be running on a different/remote machine.
  b) Receive a line of text that the (client) user generates/types through the keyboard, and send this line of text to the server through the (client) socket created in a).
  c) Listen for and receive a response from the server through the (client) socket created in a).
  d) Print the server's response on the monitor of the client's machine.

- Design a **server program** which is expected to:
  a) Open a server-side socket and put this socket in the 'listening mode'.
  b) Listen for and receive a line of text from a client through the socket created in a)
  c) Convert the line of text received in b) to upper-case and send it back to the client from b).
  d) Print the server's response on the monitor of the client's machine.

In the first half of the lecture, we will design/code the above described client and server programs using UDP sockets (i.e., using UDP as the transport-layer protocol that supports the client-server communication).
In the second half of the lecture, we will design/code the above described client and server programs using TCP sockets …

Slide 2:

The figure (flow chart) shown in this slide outlines the operation and interactions between the previously described **client and server programs, assuming these programs deploy UDP (i.e., UDP sockets in Java)**. We can see from the shown flow chart, that

- The first step in the operation of the client program is to instantiate a connectionless (UDP) DatagramSocket, and then use this socket to send and receive a (UDP) packet containing (simple) text.
- The first step in the operation of the server program is to instantiate a connectionless (UDP) socket as well – but, unlike the client's UDP socket, the server's UDP socket will initially (just) 'listen' for incoming traffic until it receives a packet from the client. Once the client packet is received, the server will extract the text from the received packet, then convert the text to uppercase, and finally send the uppercase text back to the client through the same socket.
  NOTE: It should be very much obvious that (chronologically) the server program should be initiated/running FIRST (i.e., BEFORE the client program)! Keep this in mind when creating and testing your own client / server programs.

Also, note on this slide (and in the remainder of this lecture) that we specifically assume that the client program is hypothetically running on a computer in the PRISM lab – specifically, on **jun07.cse.yorku.ca**. The server program is hypothetically hosted on one of the departmental's servers – **blue.cse.yorku.ca**. In your own experimentation with client / server programs you may be running both programs/codes on the same machine.

Furthermore, in this slide (and lecture) we use the full symbolic DNS names of the two machines running our client and server programs, as Java (and other programming languages) have a way of converting the symbolic DNS names into respective IP addresses. And, clearly, we need to use the client's and server's IP addresses in order to be able to instantiate their sockets, and have these sockets exchange (send and receive) data with each other. (Recall, a socket is uniquely defined as a combination of an IP address and a port number),

In the next 2 slides (Slide 3 and 4), we will specifically study the Java code for our **UDP client**.
In Slide 5 and 6, we will study the Java code for our **UDP server**.

Slides 3 and 4:

To better understand the design of our **UDP client**, first take a look at the illustration provided in the upper-right corner and observe that – our program actually sits between the User and the Transport Layer, hence the program needs to have 2 different buffers:

- One buffer that will receive the text from the User (i.e., text is typed in by the user), before this text is placed inside a UDP packet and sent to the server. We will name this buffer ***sendData*** **-see Line 3** of the provided UDPClient code!
- Another buffer that will be used to store the text extracted from the server's UDP packet reply before the text is shown to the User. We will name this buffer ***receiveData*** -**see Line 4** in the provided code!

Lines 1 and 2, a BufferReader object required to capture the User's input from the keyboard is instantiated.

In **Line 5**, a UDP socket (in Java, this socket is implemented by means of DatagramSocket class) is also instantiated – the name of this (UDP socket) object/instance is ***clientSocket***.
Note: if DatagramSocket() constructor is used without any arguments, then the instantiated object will be set/bound to a port number arbitrarily chosen by the OS. However, if we wanted to bind this socket to a specific port number (e.g., port number 9999), when we could use/call a different version of the constructor – DatagramSocket(9999).

In **Line 6**, the IP address of the server machine (i.e., the machine running/executing the server program – in our case it happens to be blue.cse.yorku.ca,) is 'captured' using Java's InetAddress address class and its static method getByName() (both of which were discussed in the last lecture). Put another way, this line of code uses the server's symbolic name as a String and converts it to an IP address of type InetAddress. The given InetAddress object/variable is named ***IPAddress***.

In **Line 7**, a stream/line of (input) characters is captured from the user and stored in a String-type variable ***sentence***.

In **Line 8**, the captured String ***sentence*** is converted into bytes and stored in (the previously instantiated) ***sendData*** buffer.

In **Lines 9 and 10**, a UDP packet (***sendPacket***) is created – the bytes from ***sendData*** buffer become the payload of this packet, and the previously set ***IPAddress*** and port number 7777 are placed in the headers of this packet as its 'destination IP address' (IP header) and 'destination port number' (UDP header).
Note: 1) Here we assume that the server is listening on port 7777 of the server's host machine. 2) Even though we do not explicitly provide the IP address and the port number of the client machine, those values will be supplied by the OS, and appropriately placed in the IP and UDP headers of the outgoing packet.

In **Line 11**, the previously created UDP packet is sent out through the previously instantiated ***client Socket***.

In **Lines 12 and 13**, a 'packet holder' ***receivePacket*** is created. This 'packet holder' is created utilizing the previously instantiated ***receiveData*** buffer, and will be used to capture (i.e., temporarily store the content of) the UDP response packet sent back by the server.

In **Line 14**, ***client Socket*** proactively listens for and 'captures' an incoming UDP packet.
Note: in the case of our client program, the only other application that is expected to send any packet (back) is our own server – hence, we do not have to put extra effort into checking/validating where the captured incoming packet has come from. However, in applications/programs where multiple other programs/machines could be potentially sending responses/packets to the same client socket, we would have to add additional lines of code (after Line 14) which would 'unwrap' the received packet, and determine/validate the actual sender …

In **Line 15**, the payload of the received packet (***receivePacket***) is extracted and stored in a String-type variable – ***modifiedSentence***.

In **Line 16**, ***modifiedSentence*** is printed/displayed to the user.

In **Line 17**, ***client Socket*** is closed.

Slides 5 and 6:

This and subsequent slide provide the code for **UDP server**.

In **Line 1**, the server's UDP socket – ***serverSocket*** – is instantiated and bound to port 7777.

In **Lines 2 and 3**, ***receiveData*** buffer (which in this case will be used to store the content/payload of the incoming client's packet) and ***sendData*** buffer (which in this case will be used to store the content/payload of the reply-packet sent back to the client) are instantiated.

Line 4 used to indirectly put the ***serverSocket*** into 'listening' mode – i.e., repetitive mode of 'capturing' incoming data.

In **Lines 5 and 6**, a 'UDP packet holder' ***receivePacket*** for captured incoming packets is created.

In **Line 7**, every time ***serverSocket*** senses/observes incoming data, that data is captured/stored in ***receivePacket***.

In **Line 8**, the content/payload of ***receivePacket*** is extracted into a String ***sentence***.

In **Line 9**, the IP address of the client machine (i.e., the machine that has created and sent ***receivePacket***) is extracted and stored in ***IPAddress***.

In **Line 10**, the port number of the client machine (i.e., the machine that has created and sent ***receivePacket***) is extracted and stored in ***port***.

In **Line 11**, ***sentence*** from Line 8 is converted to upper-case.

In **Line 12**, the capitalized ***sentence*** is placed inside the previously instantiated buffer ***sendData***.

In **Lines 13 and 14**, a UDP packet is created - ***sendPacket***. The content of ***sendData*** buffer from Line 12 becomes the payload of this packet, and ***IPAddress*** and ***port*** are placed in the IP header and UDP header of the given packet.

In **Line 15**, ***sendPacket*** is sent back to the client.

Slide 7:

The figure (flow chart) shown in this slide outlines the operation and interactions between the previously described **client and server programs, but this time we assume that the two programs deploy TCP (i.e., TCP sockets in Java)**. We can see from the shown flow chart, that:

- Given the connection-oriented nature of TCP protocol, the client and server sockets first have to establish a connection before being able to exchange any data.
- However, once the connection is established between the two socket, the order of actions is very similar to what we have seen in the case of UDP …

Slides 8 and 9:

The code for **TCP client** is outlined in these slides …

In **Lines 1 and 2**, similar to what we have seen in the case of UDP client, two instance variables of type String are declared – ***sentence*** which will store the input provided by the User, and ***modifiedSentence*** which will store the uppercase version of the sentence returned by the server.

In **Lines 3 and 4**, again similar to what we have seen in the case of UDP client, a BufferReader (which will allow the user-typed input to be captured) is instantiated

In **Line 5**, an ACTIVE connection-oriented client socket (***clientSocket***) is instantiated. This specific socket actively initiates the three-way handshake procedure and ultimately creates a (permanent) connection with a server running on the machine blue.cse.yorku.ca and the port number 5555.
Note: Any data sent through such a (connected) socket will now, by default, go to the specified server and port number – without the need for these parameters to be explicitly supplied. This is very different from how a UDP client/socket works, where every time we need to send data through an established UDP socket, an explicit UDP packet needs to be created, and the identity of the receiving machine and port number has to be explicitly specified.

In **Lines 6 and 7**, a handle over ***clientSocket***'s output buffer (***outToServer***) is created – i.e., a buffer where the data that is to be sent out of ***clientSocket*** (i.e., sent to the server) is temporarily stored.

In **Lines 8 and 9**, a handle over ***clientSocket***'s input buffer (***inFromServer***) is created – i.e., a buffer where the data received through ***clientSocket*** (i.e., received from the server) is temporarily stored.

With lines 1 to 9, the connection with the server has been established, and all the necessary variables/holders have been set up. In the remaining lines of the program/code, the data is actually captured from the user, and then sent to and received from the server …

In **Line 10**, the user's input/sentence is captured in ***sentence***.

In **Line 11**, the bytes of ***sentence*** are placed into ***outToServer*** buffer of ***clientSocket*** and then automatically sent/passed to the server.

In **Line 12**, the bytes of the server's response are received in ***inFromServer*** buffer of ***clientSocket*** and then stored in ***modifiedSentence***.

In **Line 13**, ***modifiedSentence*** is displayed to the user.

In **Line 14**, ***clientSocket*** is closed.

Slides 10 and 11:

The code for **TCP server** is outlined in these slides …

In **Lines 1 and 2**, similar to what we have seen in the case of TCP client, two instance variables of type String are declared – ***clientSentence*** which will store the text received from the client, and ***capitalizedSentence*** which will store the uppercase version of the sentence that is to be sent back to the client.

In **Line 3**, a connection-oriented PASSIVE server socket (***welcomeSocket***) is instantiated. In contrast to the active socket that we've used (i.e., have seen) in the TCP client program, the role of this socket is NOT to initiate the three-way handshake procedure, put instead just to passively listen for and respond to the requests for connection that arrive from other (client) programs/sockets.

Line 4 is used to indirectly put the ***welcomeSocket*** into 'listening' mode – i.e., repetitive mode of 'capturing' connection requests from different clients.

In **Line 5**, every time the ***welcomeSocket*** observes/accepts a new request for connection from a client (say client A), a new dedicated client-type of socket (***dedicatedSocket***) is created for this connection with client A. All subsequent data exchanges with client A will now go through ***dedicatedSocket***.

In **Lines 6 and 7**, a handle over ***dedicatedSocket***'s input buffer that serves to receive data from Client A is created - ***inFromClient*** buffer.

In **Lines 8 and 9**, a handle over ***dedicatedSocket***'s output buffer that serves to send data to Client A is created - ***outToClient*** buffer.

In **Line 10**, the data actually sent by Client A and captured in ***inFromClient*** is stored in ***clientSentence***.

In **Line 11**, ***clientSentence*** from Line 10 is converted to upper case and stored in ***capitalizedSentence***.

In **Line 11**, ***capitalizedSentence*** from Line 11 is placed in ***outToClient*** buffer and automatically passed back to Client A.