Lecture 12
Network Programming

IAT 267 Intro to Technological Systems

Organizational Items

- Assignment 3 posted
- Marks on webct
  - Please check your marks and let us know by next week (December 1st) if there are any errors, inaccuracies
- Project presentations next week
  - During workshop time

Topics for today

- TCP and UDP
- UDP programming
- TCP programming

TCP and UDP

- UDP = User Datagram Protocol
- TCP = Transmission Control Protocol
- When a developer creates a new application for the Internet, one of the first decisions that the developer must make is whether to use UDP or TCP
- Each of these protocols offers a different service model to the applications.

UDP Services

- **Lightweight** transport protocol with a minimalist service model
- **Connectionless**, no handshaking before the two processes start to communicate
- UDP provides an **unreliable** data transfer service. When a process sends a message into a UDP socket, UDP provides no guarantee that the message will ever reach the receiving socket
- Messages that do arrive to the receiving socket may arrive out of order.

UDP Services

- UDP does not include a flow control or congestion control mechanism, so a sending process can pump data into a UDP socket at any rate it pleases. Although all the data may not make it to the receiving socket, a large fraction of data may arrive.

- Because UDP does not use acknowledgments or retransmissions that can slow down the delivery of useful real-time data, developers of real-time applications often choose to run their applications over UDP.

- UDP provides no guarantee on delay.

TCP Services

- The TCP service model includes a **connection-oriented service** and a **reliable** data transfer service. When an application invokes TCP for its transport protocol, the application receives both of these services from TCP.

- Connection-oriented service: TCP has the client and server exchange control information with each other before the application-level messages begin to flow. This so-called handshaking procedure (part of the TCP protocol) alerts the client and server, allowing them to prepare for a transfer of packets.

TCP Services

- After the handshaking phase, a TCP connection is said to exist between the sockets of the two processes. The connection is a full-duplex connection, in that the two processes can send messages to each other over the connection at the same time.

- When the application is finished sending messages, it must tear down the connection. The service is referred to as "connection-oriented" (or a "virtual circuit" service), because the two processes are connected end-to-end in a very loose manner without any support from the intermediate nodes.

TCP services

- Reliable transport service: The communicating processes can rely on TCP to deliver all the messages sent without error and in the proper order. When one side of the application passes a stream of bytes into a socket, it can count on TCP to deliver the same stream of data to the receiving socket, with no missing or duplicate bytes.

- TCP includes an end-to-end flow control mechanism, which regulates sender transmission based on the availability of receiving buffer.

- TCP also includes a congestion control mechanism.

Network programming

- An application can open a UDP socket or a TCP socket.
- The TCP socket gives transport-level connection-oriented reliable byte-stream service to the application.
- On the other hand, the UDP socket provides transport-level connectionless unreliable datagram service to the application.

Addressing

- Whenever we need to deliver something to one specific destination among many, we need an address.
- We need a **port number**, to choose among multiple addresses running on the destination host.
- The destination port number is needed for delivery; the source port number is needed for the reply.

Port Numbers

- In the Internet model, the port numbers are 16-bit integers between 0 and 65,535.
- The client program defines itself with a port number chosen randomly by the transport layer software running on the client host (ephemeral port number).
- The server must also define itself with a port number. The port number on the server cannot be chosen randomly.
- The Internet uses universal port numbers for servers: these are called **well-known port numbers**.

Socket Address

- Process-to-process delivery needs two identifiers:
  - IP address
  - Port number
    - These are needed at each end to make a connection
- The combination of an IP address and a port number is called a socket address.
- The client socket address defines the client process uniquely, just as the server socket address defines the server process uniquely.
- **IP address + port number = socket address**

Socket API

- The Socket API (network application programming interface) is the interface that the OS provides for its networking subsystem
- The Socket API is the place to start when implementing a computer network application
- The API defines operations for:
  - Creating a socket
  - Attaching the socket to the network
  - Sending and receiving messages through the socket
  - Closing the socket

UDP

- Connectionless, unreliable
- Will not guarantee the successful or orderly delivery of messages from one end to the other
- To avoid overrunning the receive buffer, the sending application might need to pace and regulate its transmission

- UDP packets (datagrams) have a fixed-sized header of 8 bytes.

UDP Datagram

- The fields of the header are as follows:
  - **Source port number**: the port number used by the process running on the source host (16 bits)
  - **Destination port number**: port number used by the process running on the destination host (16 bits)
  - **Length**: a 16-bit field that defines the total length of the datagram (header + data)
  - **Checksum**: field used to detect errors over the entire datagram. The calculation of the checksum and its inclusion in the datagram are optional.

Port Numbers - UDP

- Some well-known port numbers used by UDP:

Port	Protocol	Description
7	Echo	Echoes a received datagram back to the sender
9	Discard	Discards any datagram that is received
11	Users	Active users
13	Daytime	Returns the date and time
17	Quote	Returns a quote of the day
19	Chargen	Returns a string of characters
53	Nameserver	Domain Name Service

UDP – example of uses

- UDP is suitable for a process that requires simple request – response communication with little concern for flow and error control

- UDP is suitable control for multicasting. Multicasting capabilities are embedded in the UDP software, but not in the TCP software.

- Also used by multimedia applications.

UDP Programming (1)

- Java's implementation of UDP is split into two classes:
  - *DatagramPacket*
  - *DatagramSocket*

- The *DatagramPacket* class: puts bytes of data into UDP packets (datagrams)

- *DatagramSocket*: sends and receives UDP datagrams

UDP Programming (2)

- To send data, the developer puts the data in a *DatagramPacket* and send the packet using a *DatagramSocket*.
- To receive data, you receive a *DatagramPacket* object and the read the contents of the packet.
- In UDP, everything about a datagram, including the address to which it is directed, is included in the packet itself; the socket needs to know only the local port on which to listen or send.

Client-Side UDP Programming (1)

- 1. Create a UDP Socket
  - Create an object of the *DatagramSocket* class
    - *DatagramSocket clientSocket = new DatagramSocket();*
  - Find the IP address of the server by creating objet of the *InetAddress* class:
    - *InetAddress IPAddress = InetAddress.getByName ("localhost");*

Client-Side UDP Programming (2)

- 2. Format your message
  - Create an array of byte objects large enough to hold your message
    - *byte[] sendData = new byte [1024];*
  - Use the methods from the *String* class to correctly format your message
  - Use the *getBytes()* method from the *String* class to put your message into a *byte[]*
    - *sendData = someString.getBytes();*

Client-Side UDP Programming (3)

- 3. Create a datagram packet to send
  - Create a new *DatagramPacket* object that holds the message and IP address and port of the server
    - *DatagramPacket sendPacket = new DatagramPacket (sendData, sendData.length, IPAddress, port);*

- 4. Send the packet to the server
  - Use the send method from your *DatagramSocket* object to send a message to the server
    - *clientSocket.send(sendPacket);*

Client-Side UDP Programming (4)

- 5. Read the server's response
  - Create an array of byte objects large enough to hold your message
    - *byte[] receivedData = new byte [1024];*
  - Use the receive method from your *DatagramSocket* object to read from the server
    - *clientSocket.receive(receivePacket);*
  - Note: you will not be able to receive more bytes than allocated in your byte array

Client-Side UDP Programming (5)

- 6. Close the socket
  - Use the *close()* method from your *Socket* object to end the UDP connection.
    - *clientSocket.close();*

Server-Side UDP Programming (1)

- 1. Create a UDP Socket bound to a listening port
  - Create an object of the *DatagramSocket* class
    - *DatagramSocket serverSocket = new DatagramSocket (port);*
- 2. Read the client's request
  - Create an array of byte objects large enough to hold your message
    - *byte [] receiveData = new byte [1024];*
  - Use the receive method from your *DatagramSocket* object to read from the server
    - *serverSocket.receive (receivePacket);*

Sever-Side UDP Programming (2)

- 3. Perform some processing
  - This will depend on the function of the server
- 4. Format your message
  - Create an array of byte objects large enough to hold your message
    - *byte [] sendData = new byte [1024];*
  - Use the methods from the *String* class to correctly format your message
  - Use the *getBytes()* method from the *String* class to put your message into a *byte[]*
    - *sendData = someString.getBytes();*

Server-Side UDP Programming (3)

- 5. Create a datagram packet to send
  - Find the IP address and port of the client
    - *InetAddresss IPAddress = receivePacket.getAddress();*
    - *Int clientPort = receivePacket.getPort();*
  - Create a new *DatagramPacket* object that holds the message and IP address and port of the server
    - *DatagramPacket sendPacket = new DatagramPacket (sendData, sendData.length, IPAddress, clientPort);*

Server-Side UDP Programming (4)

- 6. Send the packet to the client
  - Use the send method from your *DatagramSocket* object to send a message to the server
    - *serverSocket.send(sendPacket);*
- 7. Wait for another client
  - Your code should be in a loop that allows multiple clients to contact your server

TCP (1)

- Reliable and connection-oriented
- Stream oriented protocol: allows the sending process to deliver data as a stream of bytes and the receiving process to obtain data as a stream of bytes
- TCP creates an environment in which the two processes seem to be connected by an imaginary tube that carries their data across the internet

TCP (2)

- Because the sending and the receiving processes may not produce and consume data at the same speed, TCP needs buffers for storage. There are two buffers, the sending buffer and the receiving buffer.

- TCP is a connection-oriented protocol. It establishes a virtual path between the source and destination.

- The transmission requires two procedures:
  - Connection establishment
  - Connection termination

TCP (3)

- To keep track of all the different events happening during connection establishment, connection termination, and data transfer, the TCP software is implemented as a finite state machine.

- Possible states of TCP:
  - CLOSED: there is no connection
  - LISTEN: the server is waiting for calls from the client
  - SYN-SENT: a connection request is sent; waiting for acknowledgement
  - SYN-RCVD: a connection request is received

TCP (4)

- ESTABLISHED: connection is established
- FIN-WAIT-1: the application has requested the closing of the connection
- FIN-WAIT-2: the other side has accepted the closing of connection
- TIME-WAIT: waiting for retransmitted segments to die
- CLOSE-WAIT: the server is waiting for the application to close
- LAST-ACK: the server is waiting for the last acknowledgement

Client-Side TCP Programming (1)

- 1. Open a TCP socket to the server
  - Create an object of the *Socket* class
    - *Socket clientSocket = new Socket ("localhost",6789);*
    - You now have a way to communicate with the server. In this example, the server is 'localhost' listening on port 6789.
  - Open a communication stream for writing to the socket by creating an object of the *DataOutputStream* class.
    - *DataOutputStream outToServer = new DataOutputStream (clientSocket.getOutputStream());*
    - You can now use the *writeBytes()* method to send data to the other side of the socket.

Client- Side TCP Programming (2)

- - Open a communication stream for reading from the socket by creating an object of the *BufferedReader* class

  – *BufferedReader inFromServer = new BufferedReader(new InputStreamReader (clientSocket.getInputStream()));*

  – You can now use the *readLine()* method to read data one line at a time from the server.

Client-Side TCP Programming (3)

- 2. Format your message
  - Use the methods from the *String* class to correctly format your message

- 3. Send a message to the server
  - Use the *writeBytes()* method from your *DataOutputStream* object to send a message to the server
    - *outToServer.writeBytes(sentence);*

Client-side TCP Programming (4)

- Read the server's response
  - Use the *readLine()* method from your *BufferedReader* object to read one line from the server
    - *String modifiedSentence = inFromServer.readLine();*

- Close the socket
  - Use the *close()* method from your *Socket* object to end the TCP connection
    - *clientSocket.close();*

Server-Side TCP Programming (1)

- 1. Open a TCP listening socket
  - Create an object of the *ServerSocket* class.
    - *ServerSocket welcomeSocket = new ServerSocket (6789);*
    - You now have a way to communicate for a client to contact your server. In this example, the server is listening on port 6789.
- 2. Wait for a client to connect and create a new socket for communication
  - Use the *accept()* method from the *ServerSocket* class.
    - *Socket connectionSocket = welcomeSocket.accept ();*
  - The *accept()* method will wait (or, block) until the server is contacted by a client on the appropriate port. Once the connection (TCP handshake) has been made, a new socket will be created for communication with the client.

Server-side TCP Programming (2)

- Open a communication stream for writing to the socket by creating an object of the *DataOutputStream* class.
  - *DataOutputStream outToClient = new DataOutputStream (connectionSocket.getOutputStream();*
  - You can now use the *writeBytes()* method to send data to the other side of the socket
- Open a communication stream for reading from the socket by creating an object of the *BufferedReader* class.
  - *BufferedReader inFromClient = new BufferedReader (new InputStreamReader (connectionSocket. getInputStream())) ;*
  - You can now use the *readLine()* method to read data one line at a time from the client.

Server-side TCP Programming (3)

- 3.Read the client's request
  - Use the *readLine()* method from your *BufferedReader* object to read one line from the client
    - *String clientSentence = inFromClient. readLine();*
- 4. Perform some processing
  - This will depend on the function of the server
- 5. Send a message to the client
  - Use the *writeBytes()* method from the *DataOutputStream* object to send a message to the client
    - *outToClient.writeBytes (capitalizedSentence);*

Server-Side TCP Programming (4)

- 6. Close the socket
  - Use the *close()* method from your *Socket* object to end the TCP connection
    - *connectionSocket.close();*

- 7. Wait for another client
  - Your code should be in a loop that allows multiple clients to contact your server.

Thank you

Questions?