Data Communication & Networks
G22.2262-001

Session 10 - Main Theme
Java Sockets

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences
Agenda

- Internet Transport-Layer Protocols
- Multiplexing / Demultiplexing
- Socket Programming
Part I

*Internet Transport-Layer Protocols*
Internet Transport-Layer Protocols

- Reliable, in-order delivery: **TCP**
  - congestion control
  - flow control
  - connection setup
- Unreliable, unordered delivery: **UDP**
  - no-frills extension of “best-effort” IP
- Services not available:
  - delay guarantees
  - bandwidth guarantees
Part II

Multiplexing / Demultiplexing
Multiplexing/Demultiplexing

Demultiplexing at rcv host:
delivering received segments to correct socket

Multiplexing at send host:
gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)

= socket  = process

<table>
<thead>
<tr>
<th>application</th>
<th>P3</th>
<th>transport</th>
<th>network</th>
<th>link</th>
<th>physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>host 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P1</th>
<th>application</th>
<th>P2</th>
<th>transport</th>
<th>network</th>
<th>link</th>
<th>physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>host 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P4</th>
<th>application</th>
<th>transport</th>
<th>network</th>
<th>link</th>
<th>physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>host 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

= process
How Demultiplexing Works

• Host receives IP datagrams
  – each datagram has source IP address, destination IP address
  – each datagram carries 1 transport-layer segment
  – each segment has source, destination port number (recall: well-known port numbers for specific applications)

• Host uses IP addresses & port numbers to direct segment to appropriate socket

TCP/UDP segment format

- source port #
- dest port #
- other header fields
- application data (message)
- 32 bits
Connectionless Demultiplexing

• Create sockets with port numbers:
  DatagramSocket
    mySocket1 = new DatagramSocket(99111);
  DatagramSocket
    mySocket2 = new DatagramSocket(99222);

• UDP socket identified by two-tuple:
  (dest IP address, dest port number)

• When host receives UDP segment:
  – checks destination port number in segment
  – directs UDP segment to socket with that port number

• IP datagrams with different source IP addresses and/or source port numbers directed to same socket
DatagramSocket serverSocket = new DatagramSocket(6428);

SP provides “return address”
Connection-Oriented Demux

• TCP socket identified by 4-tuple:
  – source IP address
  – source port number
  – dest IP address
  – dest port number

• recv host uses all four values to direct segment to appropriate socket

• Server host may support many simultaneous TCP sockets:
  – each socket identified by its own 4-tuple

• Web servers have different sockets for each connecting client
  – non-persistent HTTP will have different socket for each request
Connection-Oriented Demux (cont.)

Client IP: A

SP: 9157
DP: 80
S-IP: A
D-IP: C

Server IP: C

SP: 5775
DP: 80
S-IP: B
D-IP: C

Client IP: B
Part III

Socket Programming
Socket Programming

Goal: learn how to build client/server application that communicate using sockets

Socket API
- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by apps
- client/server paradigm
- two types of transport service via socket API:
  - unreliable datagram
  - reliable, byte stream-oriented
Socket Programming Using TCP

**Socket:** a door between application process and end-end-transport protocol (UCP or TCP)

**TCP service:** reliable transfer of bytes from one process to another
Socket Programming With TCP

Client must contact server
- server process must first be running
- server must have created socket (door) that welcomes client’s contact

Client contacts server by:
- creating client-local TCP socket
- specifying IP address, port number of server process
- When client creates socket: client TCP establishes connection to server TCP

When contacted by client, server TCP creates new socket for server process to communicate with client
- allows server to talk with multiple clients
- source port numbers used to distinguish clients (more in Chap 3)

TCP provides reliable, in-order transfer of bytes (“pipe”) between client and server
Stream Jargon

• A **stream** is a sequence of characters that flow into or out of a process

• An **input stream** is attached to some input source for the process (e.g., keyboard or socket)

• An **output stream** is attached to an output source (e.g., monitor or socket)
Example client-server app:

1) client reads line from standard input (\texttt{inFromUser} stream), sends to server via socket (\texttt{outToServer} stream)

2) server reads line from socket

3) server converts line to uppercase, sends back to client

4) client reads, prints modified line from socket (\texttt{inFromServer} stream)
Client/Server Socket Interaction: TCP

**Server (running on hostid)**

- create socket, port=x, for incoming request:
  - `welcomeSocket = ServerSocket()`
- wait for incoming connection request:
  - `connectionSocket = welcomeSocket.accept()`
- read request from `connectionSocket`
- write reply to `connectionSocket`
- close `connectionSocket`

**Client**

- create socket, connect to hostid, port=x:
  - `clientSocket = Socket()`
- send request using `clientSocket`
- read reply from `clientSocket`
- close `clientSocket`
Example: Java Client (TCP)

```java
import java.io.*;
import java.net.*;
class TCPClient {

    public static void main(String argv[]) throws Exception {
        String sentence;
        String modifiedSentence;

        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));

        Socket clientSocket = new Socket("hostname", 6789);

        DataOutputStream outToServer =
            new DataOutputStream(clientSocket.getOutputStream());

        // Create input stream
        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));

        // Create client socket, connect to server
        Socket clientSocket =
            new Socket("hostname", 6789);

        // Create output stream attached to socket
        DataOutputStream outToServer =
            new DataOutputStream(clientSocket.getOutputStream());
    }
}
```
Example: Java Client (TCP), cont.

```java
BufferedReader inFromServer =
    new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));

sentence = inFromUser.readLine();
outToServer.writeBytes(sentence + '\n');

modifiedSentence = inFromServer.readLine();
System.out.println("FROM SERVER: " + modifiedSentence);

clientSocket.close();
```
Example: Java Server (TCP)

```java
import java.io.*;
import java.net.*;

class TCPServer {

    public static void main(String argv[]) throws Exception {
        String clientSentence;
        String capitalizedSentence;
        ServerSocket welcomeSocket = new ServerSocket(6789);

        while(true) {
            Socket connectionSocket = welcomeSocket.accept();
            BufferedReader inFromClient =
                new BufferedReader(new InputStreamReader(connectionSocket.getInputStream()));
            String clientSentence;
            String capitalizedSentence;
        }
    }
}
```
Example: Java Server (TCP), cont.

```java
DataOutputStream outToClient = new DataOutputStream(connectionSocket.getOutputStream());
clientSentence = inFromClient.readLine();
capitalizedSentence = clientSentence.toUpperCase() + '\n';
outToClient.writeBytes(capitalizedSentence);
}
%
End of while loop, loop back and wait for another client connection
```
Socket Programming With UDP

UDP: no “connection” between client and server
- no handshaking
- sender explicitly attaches IP address and port of destination to each packet
- server must extract IP address, port of sender from received packet

UDP: transmitted data may be received out of order, or lost

application viewpoint

UDP provides unreliable transfer of groups of bytes (“datagrams”) between client and server
Client/Server Socket Interaction: UDP

Server (running on hostid)

- create socket, port=x, for incoming request:
  - serverSocket = DatagramSocket()
- read request from serverSocket
- write reply to serverSocket specifying client host address, port number

Client

- create socket, clientSocket = DatagramSocket()
- Create, address (hostid, port=x, send datagram request using clientSocket
- read reply from clientSocket
- close clientSocket
Example: Java Client (UDP)

Client process

Output: sends packet (TCP sent “byte stream”)

Input: receives packet (TCP received “byte stream”)

sendPacket
inFromUser
keyboard
monitor

receivePacket

Client socket

UDP packet

UDP socket

to network from network
Example: Java Client (UDP)

```java
import java.io.*;
import java.net.*;

class UDPCClient {
    public static void main(String args[]) throws Exception {
        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));
        DatagramSocket clientSocket = new DatagramSocket();
        InetAddress IPAddress = InetAddress.getByName("hostname");
        byte[] sendData = new byte[1024];
        byte[] receiveData = new byte[1024];
        String sentence = inFromUser.readLine();
        sendData = sentence.getBytes();
```
Example: Java Client (UDP), cont.

Create datagram with data-to-send, length, IP addr, port

```java
datagramPacket sendPacket =
    new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
```

Send datagram to server

```java
clientSocket.send(sendPacket);
```

Read datagram from server

```java
datagramPacket receivePacket =
    new DatagramPacket(receiveData, receiveData.length);

clientSocket.receive(receivePacket);

String modifiedSentence =
    new String(receivePacket.getData());

System.out.println("FROM SERVER:" + modifiedSentence);
clientSocket.close();
}
```
import java.io.*;
import java.net.*;

class UDPServer {
    public static void main(String args[]) throws Exception {
        DatagramSocket serverSocket = new DatagramSocket(9876);
        byte[] receiveData = new byte[1024];
        byte[] sendData = new byte[1024];

        while (true) {
            DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
            serverSocket.receive(receivePacket);
            System.out.println("Message received:");
            System.out.println("\tFrom: ", receivePacket.getAddress(), "; ", receivePacket.getPort());
            System.out.println("\tData: ");
            System.out.println("\t\t\t", new String(receiveData));
            sendData = receiveData;
            DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length,
                    receivePacket.getAddress(), receivePacket.getPort());
            serverSocket.send(sendPacket);
        }
    }
}
Example: Java Server (UDP), cont.

```java
String sentence = new String(receivePacket.getData());
InetAddress IPAddress = receivePacket.getAddress();
int port = receivePacket.getPort();

String capitalizedSentence = sentence.toUpperCase();
sendData = capitalizedSentence.getBytes();

DatagramPacket sendPacket =
    new DatagramPacket(sendData, sendData.length, IPAddress, port);
serverSocket.send(sendPacket);
```

Get IP addr

port #, of
sender

Create datagram
to send to client

Write out
datagram
to socket

End of while loop,
loop back and wait for
another datagram
Part IV

Conclusion
Assignment & Readings

- **Final Project (due 05/19/15)**
  - Assigned after the last class

- **Readings**
  - Java.Net Package Documentation on Sun’s Java Web site
  - [http://java.sun.com/docs/books/tutorial/networking/sockets/](http://java.sun.com/docs/books/tutorial/networking/sockets/)
Next Session:
IP Multicast