Topic #03
Today’s Objectives

- Sockets
  - What they are (including types)
  - What they do
  - How applications use them

- A bit about HW#1
Types of Sockets

There are two types of sockets and each do different things

- Sockets are the bridge between the Application Layer and the Transport Layer
- Remember that the Transport Layer is “value-added services” on top of IP (end-to-end delivery)

Types

- Stream (aka TCP) [not to be confused with streaming]
- Datagram (aka UDP)
Services Provided by Each Type

Stream (aka TCP)
- Connections
- In-order delivery
- Reliability
- Congestion Control
- Multiplexing/Demultiplexing (sockets)

Datagram (aka UDP)
- Multiplexing/Demultiplexing (sockets)
Using Sockets

- Use of a socket, from the perspective of sending and receiving data, is the same regardless of whether it is TCP or UDP.

- In other words, the extra functionality of TCP is hidden from the user.
  - With TCP it just seems like everything that is sent is always received in order at the other end.
  - The magic happens inside of the kernel.
Inside a Host

![Diagram of communication layers within a host](image)
The Four Pieces of Information

- Four pieces of information serve to differentiate communication from one application to any other application
  - Source IP address
  - Source Port
  - Destination IP address
  - Destination Port

- Key rule: no two applications can use the same port (but one application can use two ports)
Four Pieces of Information
Client and Server Behavior

- Clients are easier

- Open a socket by specifying: the type, the destination IP address, and the destination port
  - The source port is chosen by the OS as the next available (i.e., unused) local port
  - Some ports are reserved (0-1024)
    - Many in this range are well known (e.g., Port 80 = web)
  - Ports span the range from 0 to 65,535
Client and Server Behavior

- Servers are slightly harder
  - Have to deal with the possibility of multiple clients simultaneously

- Generally work by opening a socket and specifying the type and the local port on which to listen
  - Then sit and listen until a packet is received
  - When a packet is received, do stuff
Designing Application Layer Protocols

- A fairly straightforward process
  - Since you control the design of the client and the server, whatever rules you make, you build into the client and server

- The hard part is dealing with the corner cases and “what-ifs”—part of the objective of HW#1

- The harder part is trying to write out a specification for a protocol
  - Will your HW#1 clients work with others’ servers?
Adding Application Layer Headers

- Also a fairly straightforward process
  - Again, since you are in control of the client and server, you can program them to do whatever you want

- Ex: put the length at the beginning of every packet
  - Have to be somewhat specific: a 4-byte integer
  - Harder to parse variable length packet headers
    - But not impossible

- Just need to make sure, the header is processed correctly at the other end
Types of Sockets

- It isn’t quite true that there are only two types of sockets

- There are other types, the most interesting of which is a “raw socket”, a socket directly to the IP layer with no transport layer protocol.
  - Used for some protocols (like ICMP... e.g., for ping)

- Interesting things can be done with raw sockets
Chapter 2: Application layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 P2P file sharing
- 2.7 Socket programming with TCP
- 2.8 Socket programming with UDP
- 2.9 Building a Web server
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

* a *host-local*, application-created, *OS-controlled* interface (a “door”) into which application process can both send and receive messages to/from another application process
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)

application viewpoint
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, eg, keyboard or socket.
- An output stream is attached to an output source, eg, monitor or socket.
Socket programming with TCP

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) and Client

- Server:
  - 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

TCP connection setup
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());
        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));
        Socket clientSocket = new Socket("hostname", 6789);
        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;

            ServerSocket welcomeSocket = new ServerSocket(6789);

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

Create output stream, attached to socket

DataOutputStream outToClient =
   new DataOutputStream(connectionSocket.getOutputStream());

Read in line from socket

clientSentence = inFromClient.readLine();

capitalizedSentence = clientSentence.toUpperCase() + '\n';

Write out line to socket

outToClient.writeBytes(capitalizedSentence);

End of while loop, loop back and wait for another client connection
Chapter 2: Application layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 P2P file sharing
- 2.7 Socket programming with TCP
- 2.8 Socket programming with UDP
- 2.9 Building a Web server
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
- 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 (UDP sent “byte stream”)

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

Client UDP socket
Example: Java client (UDP)

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

class UDPClient {
    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

Send datagram
to server

Read datagram
from server

DatagramPacket sendPacket =
new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
clientSocket.send(sendPacket);
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);
            // Create datagram socket at port 9876
            // Create space for received datagram
            // Receive datagram
        }
    }
}
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);
```
Chapter 2: Application layer

- 2.1 Principles of network applications
  - app architectures
  - app requirements
- 2.2 Web and HTTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 P2P file sharing
- 2.7 Socket programming with TCP
- 2.8 Socket programming with UDP
- 2.9 Building a Web server
Building a simple Web server

- handles one HTTP request
- accepts the request
- parses header
- obtains requested file from server’s file system
- creates HTTP response message:
  - header lines + file
- sends response to client

- after creating server, you can request file using a browser (e.g., IE explorer)
- see text for details
A Few Notes About HW#1

- It isn’t designed to be hard, rather it is designed so that you have the experience of doing some socket programming.

- As the assignment says, you can “borrow” code:
  - No sense re-writing socket code when it exists everywhere—including in examples in the textbook.
  - But “borrowed code” is like copying text from a book… it must be identified and cited—use comments.

- Because it is hard to put every single requirement into the assignment, the process will to be to send Daniel and I questions.