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 with 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, 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 (inFromUser stream), sends to server via socket (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 (inFromServer stream)
Client/server socket interaction: TCP

**Server (running on hostid)**

- create socket, port=x, for incoming request:
  
  \[
  \text{welcomeSocket = ServerSocket();}
  \]

- wait for incoming connection request:
  
  \[
  \text{connectionSocket = welcomeSocket.accept();}
  \]

- read request from connectionSocket

- write reply to connectionSocket

- close connectionSocket

**Client**

- create socket, connect to hostid, port=x:
  
  \[
  \text{clientSocket = Socket();}
  \]

- 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());
        BufferedReader inFromServer = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));

        String modifiedSentence = transform(sentence);
        outToServer.writeUTF(modifiedSentence);
    }

    public static String transform(String sentence) {
        // Transformation logic
    }
}
```

---
Example: Java client (TCP), cont.

Create input stream attached to socket
BufferedReader inFromServer =
    new BufferedReader(new
    InputStreamReader(clientSocket.getInputStream()));

Send line to server
sentence = inFromUser.readLine();
outToServer.writeBytes(sentence + "\n");

Read line from server
modifiedSentence = inFromServer.readLine();
System.out.println("FROM SERVER: " + modifiedSentence);
clientSocket.close();

Example: Java server (TCP)

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()));
Example: Java server (TCP), cont

```java
DataOutputStream outToClient = new DataOutputStream(connectionSocket.getOutputStream());
clientSentence = inFromClient.readLine();
capitalizedSentence = clientSentence.toUpperCase() + '
';
outToClient.writeBytes(capitalizedSentence);
}
}
```
Client/server socket interaction: UDP

**Server (running on hostid)**

- Create socket, `port=x`, for incoming request:
  ```java
  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**

- **Input:** receives packet (TCP received "byte stream")
- **Output:** sends packet (TCP sent "byte stream")
Example: Java client (UDP)

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();

        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();
    }
}
Example: Java server (UDP)

```java
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);
            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);
        }
    }
}
```

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);
```
Socket programming: references

C-language tutorial (audio/slides):
- "Unix Network Programming" (J. Kurose),
  http://manic.cs.umass.edu/courses2.html

Java-tutorials:
- "All About Sockets" (Sun tutorial),
  http://java.sun.com/docs/books/tutorial/networking/sockets
- "Socket Programming in Java: a tutorial" (Java World),
  http://www.javaworld.com/javaworld/jw-12-1996/jw-12-sockets.html

Chapter 2: Summary

Our study of network apps now complete!

- application service requirements:
  - reliability, bandwidth, delay
- client-server paradigm
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP

- specific protocols:
  - HTTP
  - FTP
  - SMTP, POP, IMAP
  - DNS
- socket programming
- content distribution
  - caches, CDNs
  - P2P
Chapter 2: Summary

Most importantly: learned about protocols

- typical request/reply message exchange:
  - client requests info or service
  - server responds with data, status code
- message formats:
  - headers: fields giving info about data
  - data: info being communicated
- control vs. data msgs
- in-band, out-of-band
- centralized vs. decentralized
- stateless vs. stateful
- reliable vs. unreliable msg transfer
- “complexity at network edge”
- security: authentication