Chapter 2: Application Layer

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

---

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Principles of network applications</td>
</tr>
<tr>
<td></td>
<td>2.2 Web and HTTP</td>
</tr>
<tr>
<td></td>
<td>2.3 FTP</td>
</tr>
<tr>
<td></td>
<td>2.4 Electronic Mail</td>
</tr>
<tr>
<td></td>
<td>2.5 DNS</td>
</tr>
<tr>
<td></td>
<td>2.6 P2P file sharing</td>
</tr>
<tr>
<td></td>
<td>2.7 Socket programming with TCP</td>
</tr>
<tr>
<td></td>
<td>2.8 Socket programming with UDP</td>
</tr>
<tr>
<td></td>
<td>2.9 Building a Web server</td>
</tr>
</tbody>
</table>

---

A note on the use of these ppt slides:

- We're making these slides freely available to all (faculty, students, readers).
- They're in PowerPoint form so you can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a lot of work on our part. In return for use, we only ask the following:
  - If you use these slides (e.g., in a class) in substantially unaltered form, that you mention their source (after all, we'd like people to use our book!)
  - If you post any slides in substantially unaltered form on a web site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

All material copyright 1996-2004 J.F. Kurose and K.W. Ross, All Rights Reserved
**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

- Controlled by application developer
- Controlled by operating system

---

**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.

---

**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's socket
  - 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's socket
  - 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**

---

**Java TCP Socket Programming**

**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:
  - 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());

        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()));
            System.out.println("FROM CLIENT:" + inFromClient.readLine());
            modifiedSentence = inFromClient.readLine();
            System.out.println("TO SERVER:" + modifiedSentence + "n");
            dataOutputStream.writeBytes(modifiedSentence + ":n");
        }
    }
}
```
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

2. Application Layer

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

Client/server socket interaction: UDP

Server (running on hostid)

- create socket, port=x, for incoming request: serverSocket = DatagramSocket(

Client

- create socket, clientSocket = DatagramSocket()

- Create address (hostid, port=x), send datagram request using clientSocket

- read request from serverSocket
- write reply to serverSocket specifying client host address, port number
- read reply from clientSocket
- close clientSocket
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();

        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);
            byte[] receiveData = new byte[1024];
            byte[] sendData = new byte[1024];
            String sentence = inFromUser.readLine();
            sendData = sentence.getBytes();

            DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
            serverSocket.send(sendPacket);

            System.out.println("FROM SERVER:" + modifiedSentence);
            clientSocket.close();
        }
    }
}
```
Example: Java server (UDP), cont

Example:

```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);
```

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

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

Chapter 2: Summary

- Application architectures
  - client-server
  - P2P
  - hybrid
- application service requirements:
  - reliability, bandwidth, delay
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP
- specific protocols:
  - HTTP
  - FTP
  - SMTP, POP3, IMAP
  - DNS
  - socket programming
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"