

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

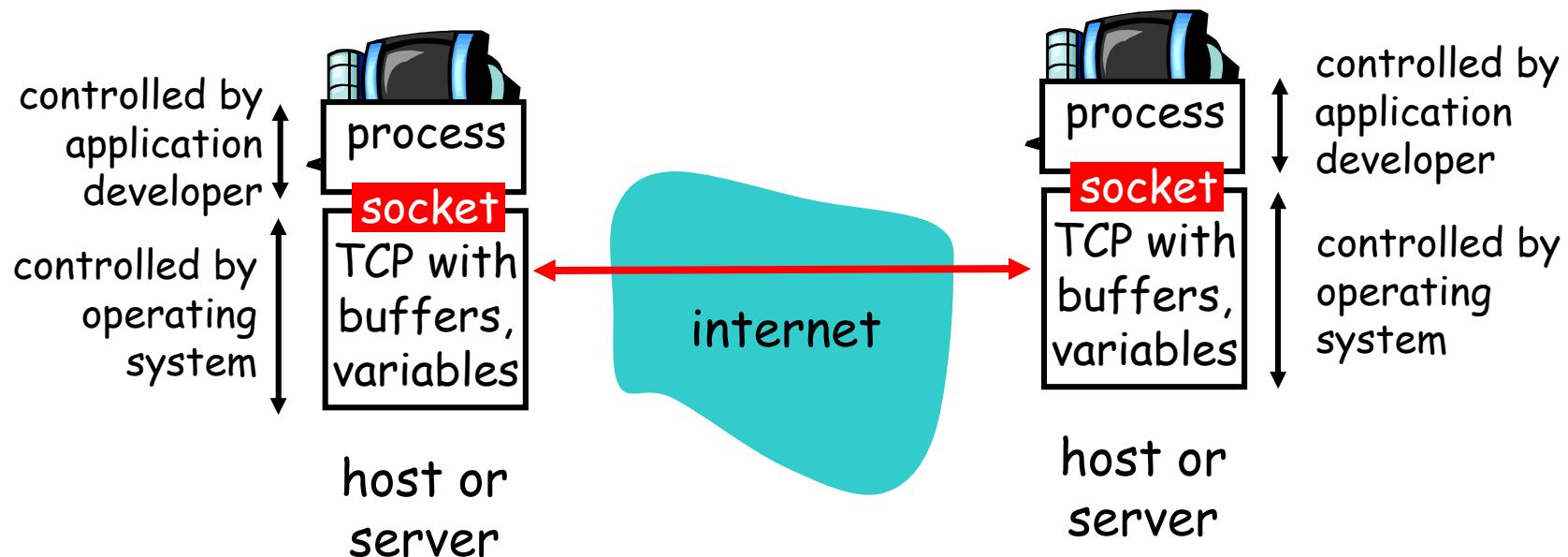
socket

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

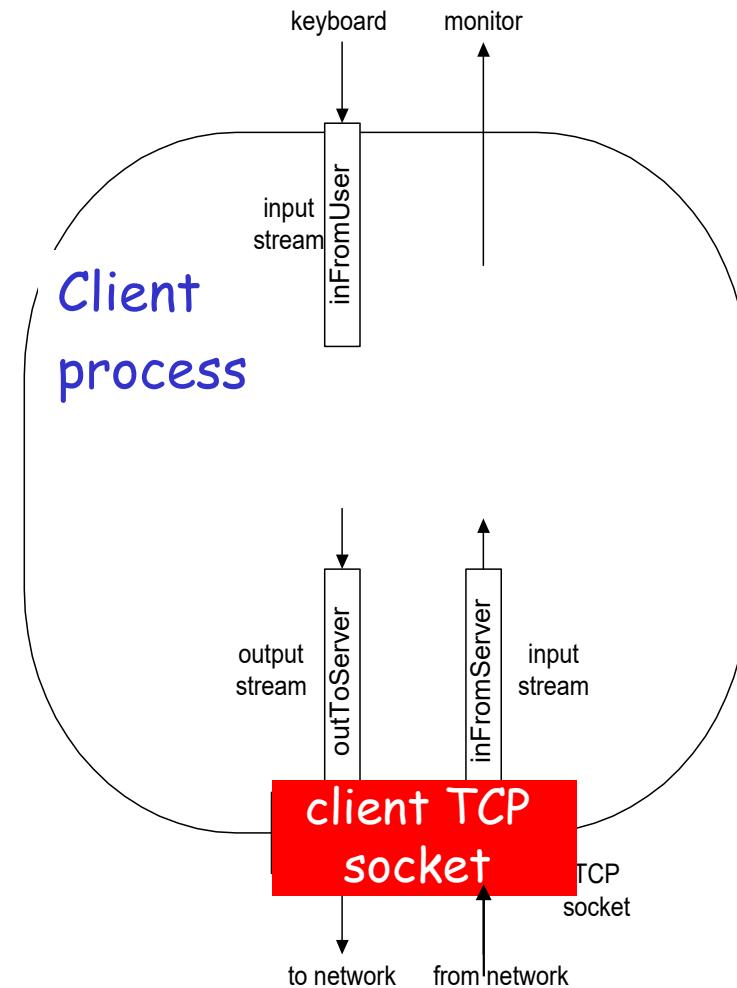
Stream jargon

- 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, e.g., keyboard or socket.
- An **output stream** is attached to an output source, e.g., monitor or socket.

Socket programming with TCP

Example client-server app:

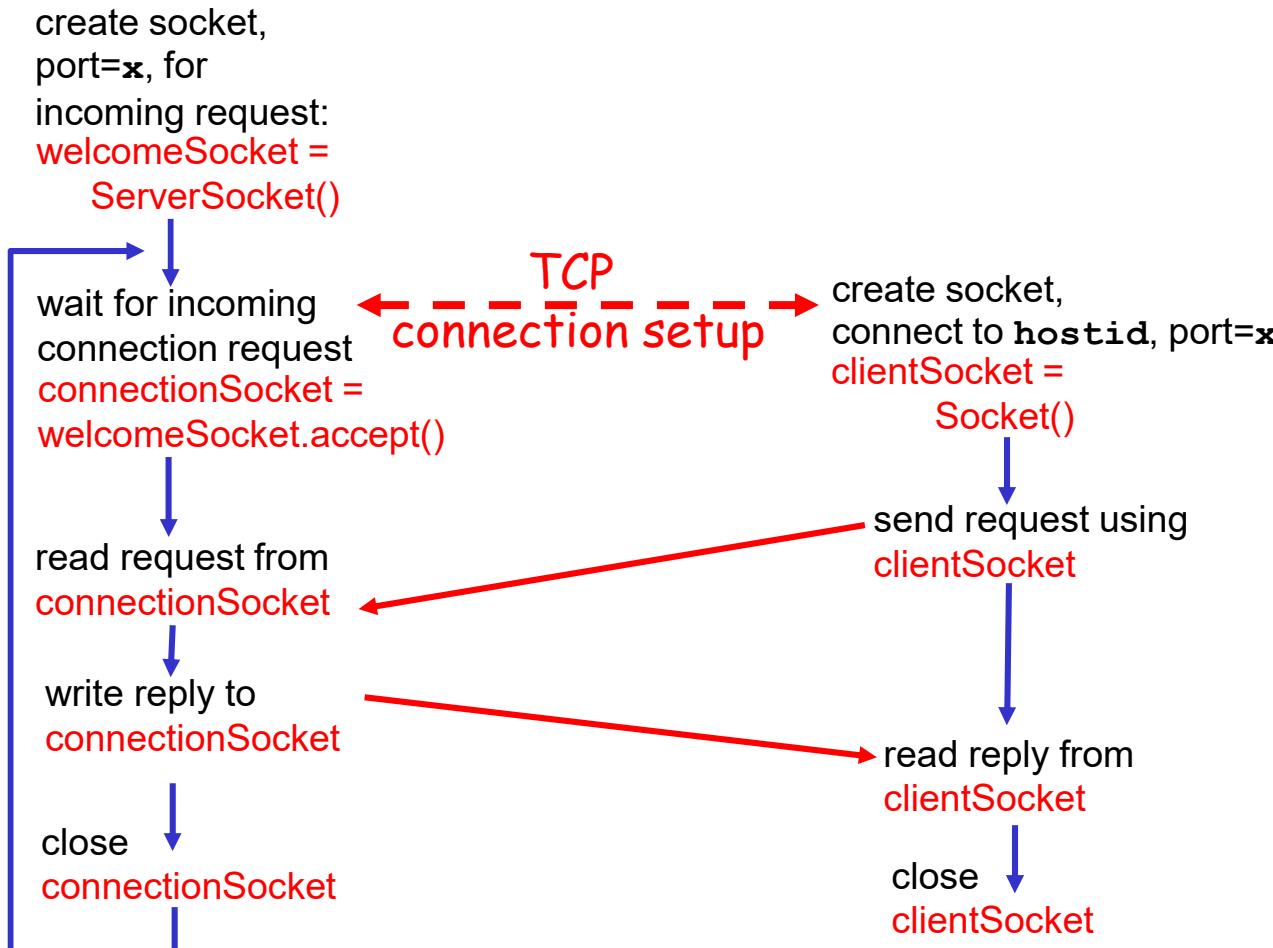
- 1) client reads line from standard input (`inFromUser` stream), sends to server via socket (`outToServer` stream)
- 2) server reads line from socket
- 3) server converts line to uppercase, sends back to client
- 4) client reads, prints modified line from socket (`inFromServer` stream)



Client/server socket interaction: TCP

Server (running on `hostid`)

Client



Example: Java client (TCP)

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

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

        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.

```
        Create  
        input stream  
attached to socket ]→ BufferedReader inFromServer =  
new BufferedReader(new  
InputStreamReader(clientSocket.getInputStream()));  
  
        Send line  
to server ]→ sentence = inFromUser.readLine();  
  
        Read line  
from server ]→ outToServer.writeBytes(sentence + '\n');  
  
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()));


```

Create welcoming socket at port 6789 → ServerSocket welcomeSocket = new ServerSocket(6789);

Wait, on welcoming socket for contact by client → while(true) {
 Socket connectionSocket = welcomeSocket.accept();

Create input stream, attached to socket → BufferedReader inFromClient =
 new BufferedReader(new
 InputStreamReader(connectionSocket.getInputStream()));

Example: Java server (TCP), cont

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

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()
```



Client

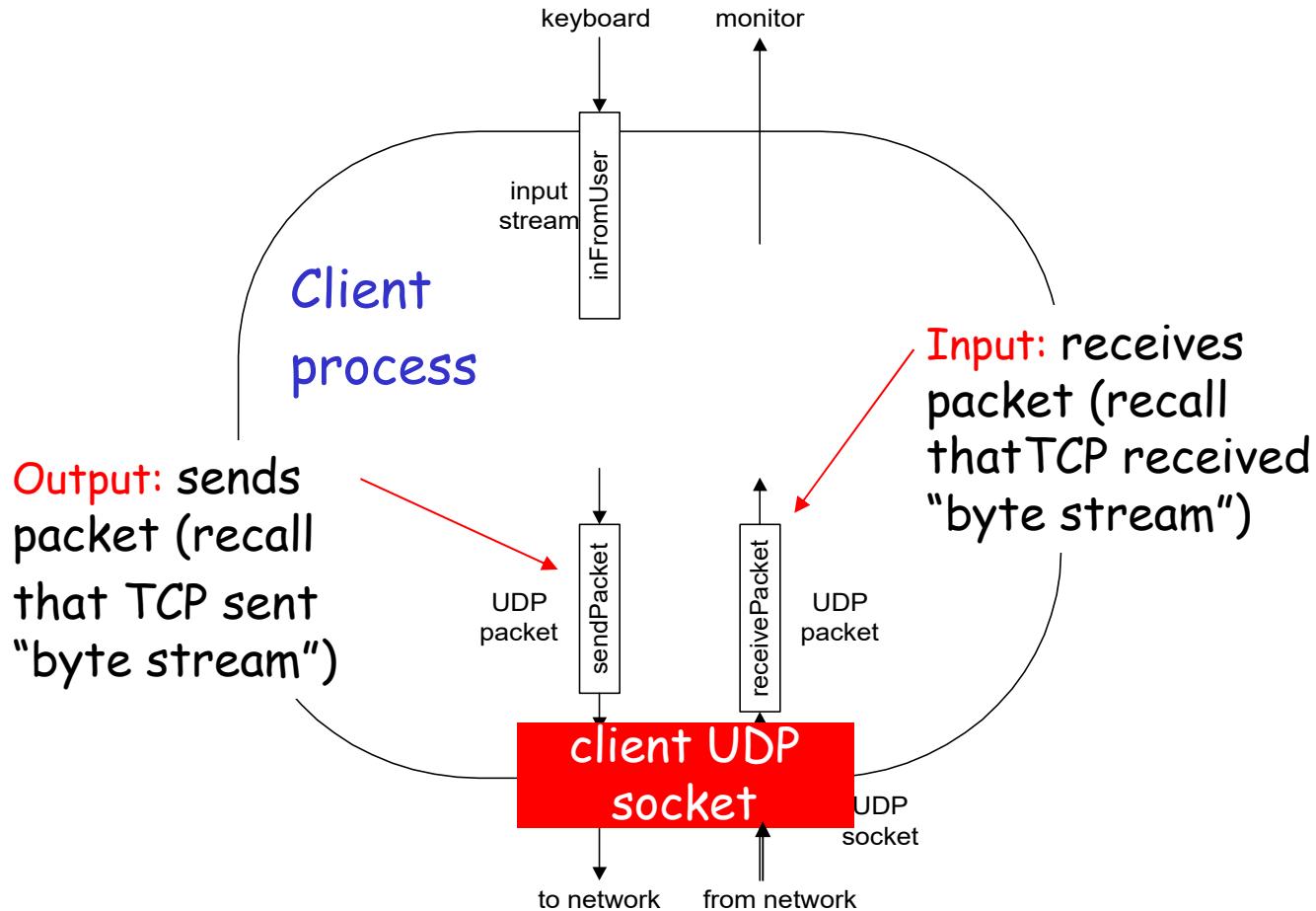
```
create socket,  
clientSocket =  
DatagramSocket()
```

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

```
read reply from  
clientSocket
```

```
close  
clientSocket
```

Example: Java client (UDP)



Example: Java client (UDP)

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

class UDPClient {
    public static void main(String args[]) throws Exception
    {
        Create
 stream] → BufferedReader inFromUser =
        Create
 client socket] → new BufferedReader(new InputStreamReader(System.in));

        clientSocket = new DatagramSocket();
        Translate
 hostname to IP
address using DNS] → 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] DatagramPacket sendPacket =  
new DatagramPacket(sendData, sendData.length, IPAddress, 9876);  
  
Send datagram  
to server] clientSocket.send(sendPacket);  
  
Read datagram  
from server] 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)

```
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

```
String sentence = new String(receivePacket.getData());  
Get IP addr  
port #, of  
sender } → InetAddress IPAddress = receivePacket.getAddress();  
int port = receivePacket.getPort();  
  
String capitalizedSentence = sentence.toUpperCase();  
  
sendData = capitalizedSentence.getBytes();  
Create datagram  
to send to client } → DatagramPacket sendPacket =  
new DatagramPacket(sendData, sendData.length, IPAddress,  
port);  
  
Write out  
datagram  
to socket } → serverSocket.send(sendPacket);  
}  
} } → End of while loop,  
loop back and wait for  
another datagram
```

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