Network Applications: TCP Socket Programming; File Transfer Protocol; HTTP/1.0

Qiao Xiang, Congming Gao

https://sngroup.org.cn/courses/cnns-xmuf23/index.shtml

10/10/2023

This deck of slides are heavily based on CPSC 433/533 at Yale University, by courtesy of Dr. Y. Richard Yang.

Outline

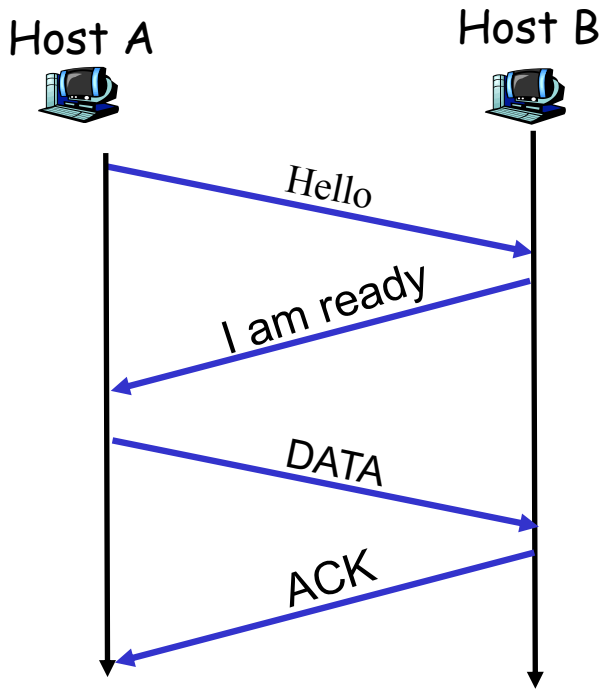
- Admin. and recap
- Network application programming
  - UDP sockets
  - TCP sockets
- Network applications (continue)
  - File transfer (FTP) and extension
  - HTTP
    - HTTP/1.0

Admin.

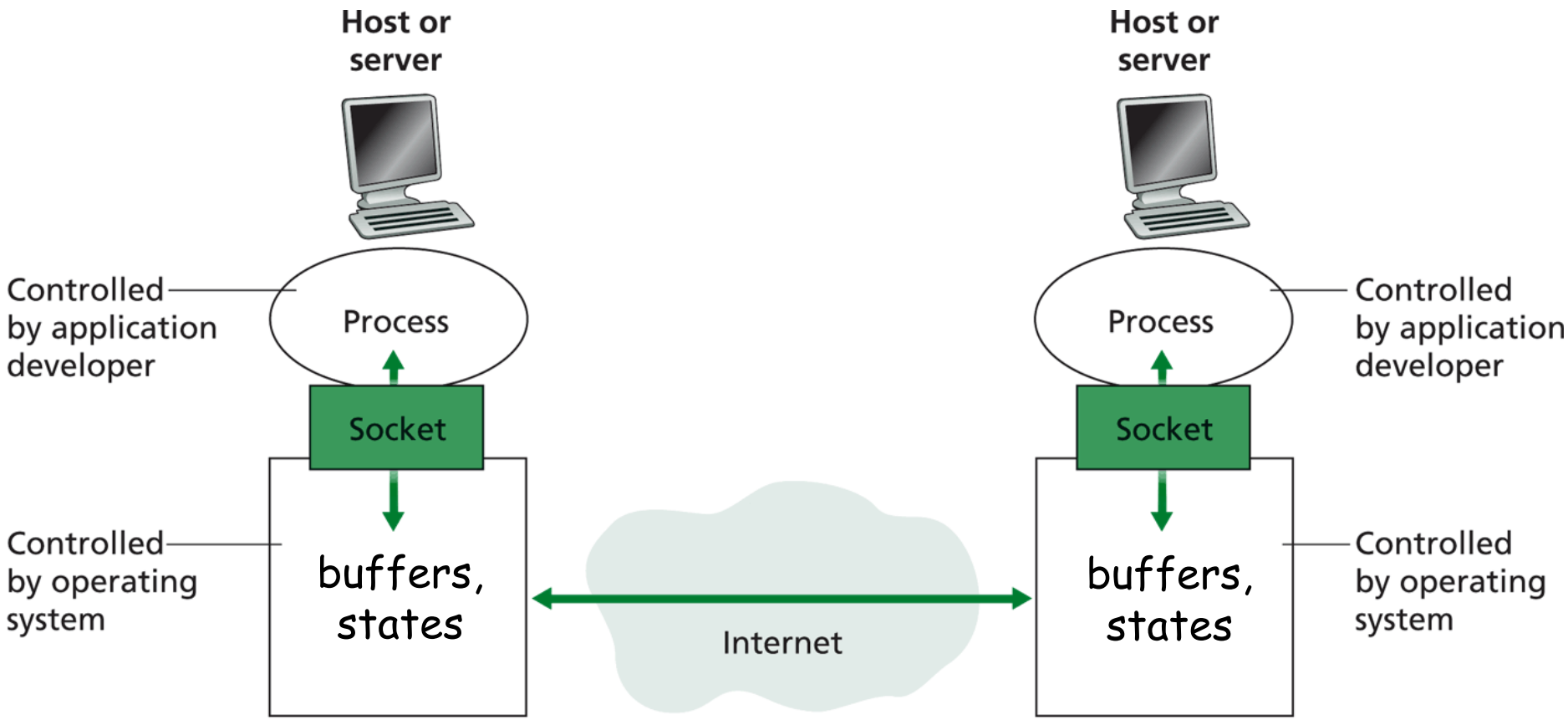
- Lab assignment 2 due Oct. 19
  - LLM-related tools are strictly prohibited for this assignment

Recap: Services Provided by Transport

- User data protocol (UDP)
  - multiplexing/demultiplexing
- Transmission control protocol (TCP)
  - multiplexing/demultiplexing
  - reliable data transfer
  - rate control: flow control and congestion control



Big Picture: Socket

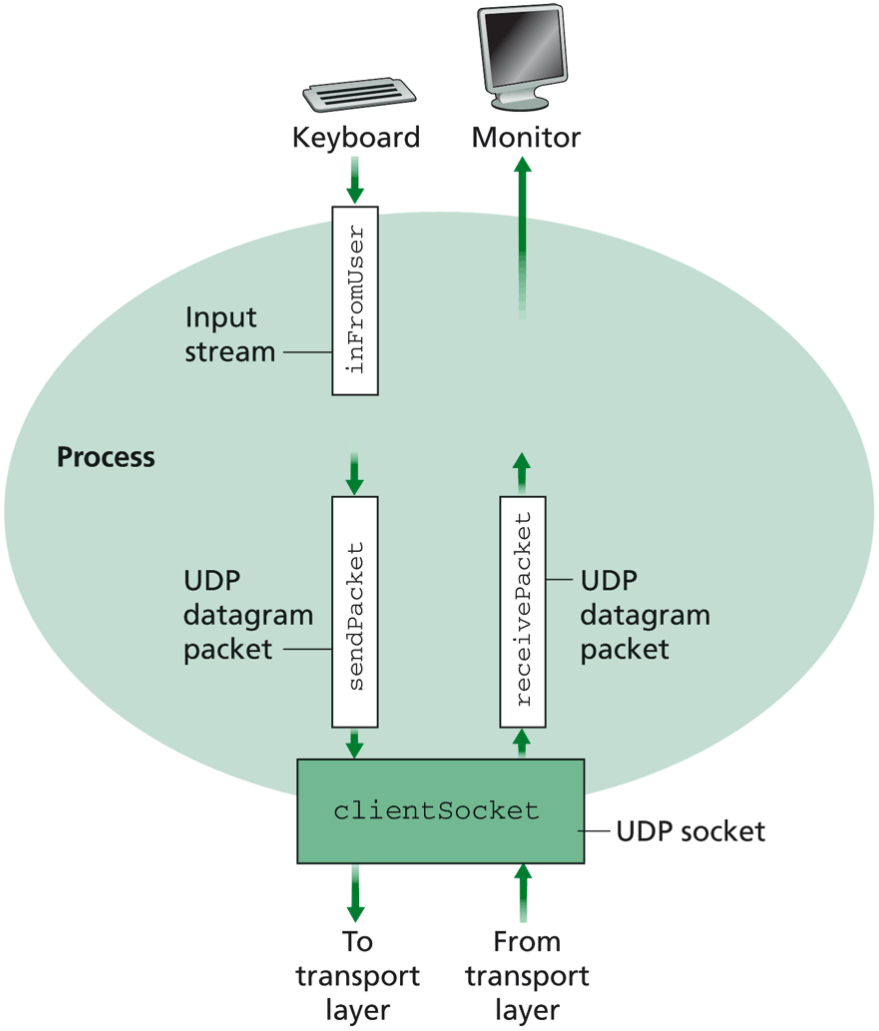


Discussion on Example Code

- A simple upper-case UDP echo service is among the simplest network service.
- Are there any problems with the program?

Example: UDPClient.java

- A simple UDP client which reads input from keyboard, sends the input to server, and reads the reply back from the server.



Demo

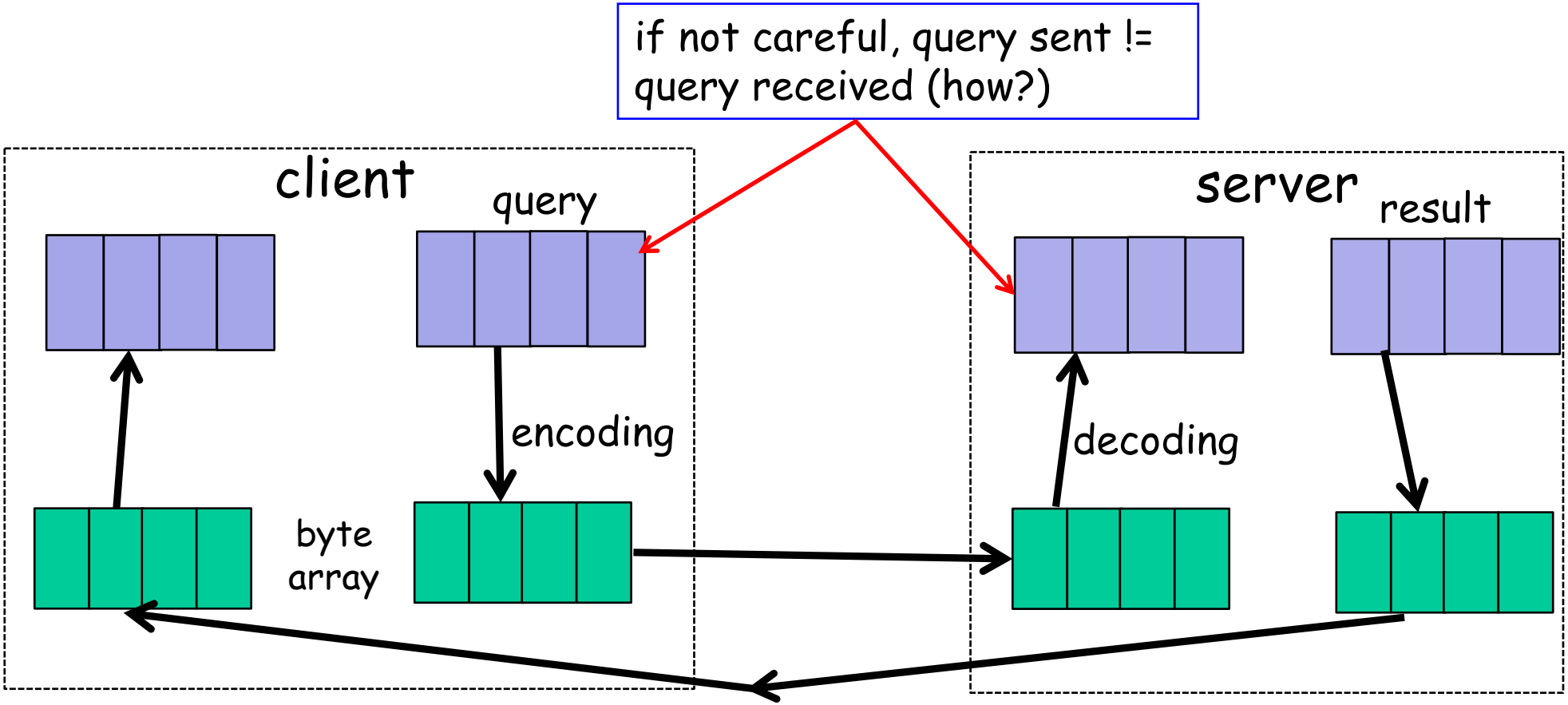
%ubuntu: java UDPServer
%netstat to see buffer

%ubuntu: java UDPClient <server>

%wireshark to capture traffic

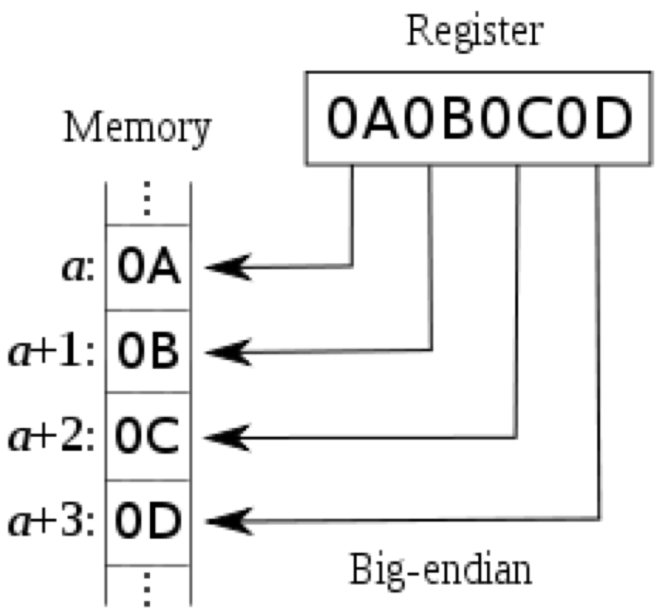
Data Encoding/Decoding

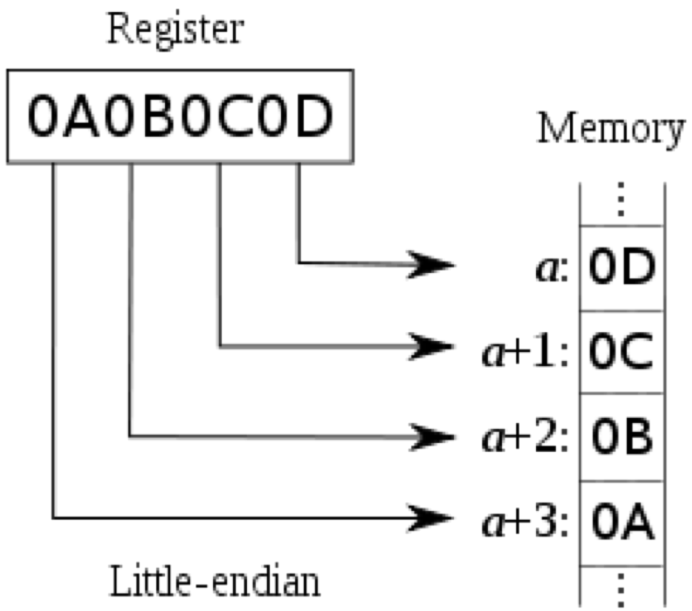
- Rule: ALWAYS pay attention to encoding/decoding of data



Example: Endianness of Numbers

- int var = 0x0A0B0C0D

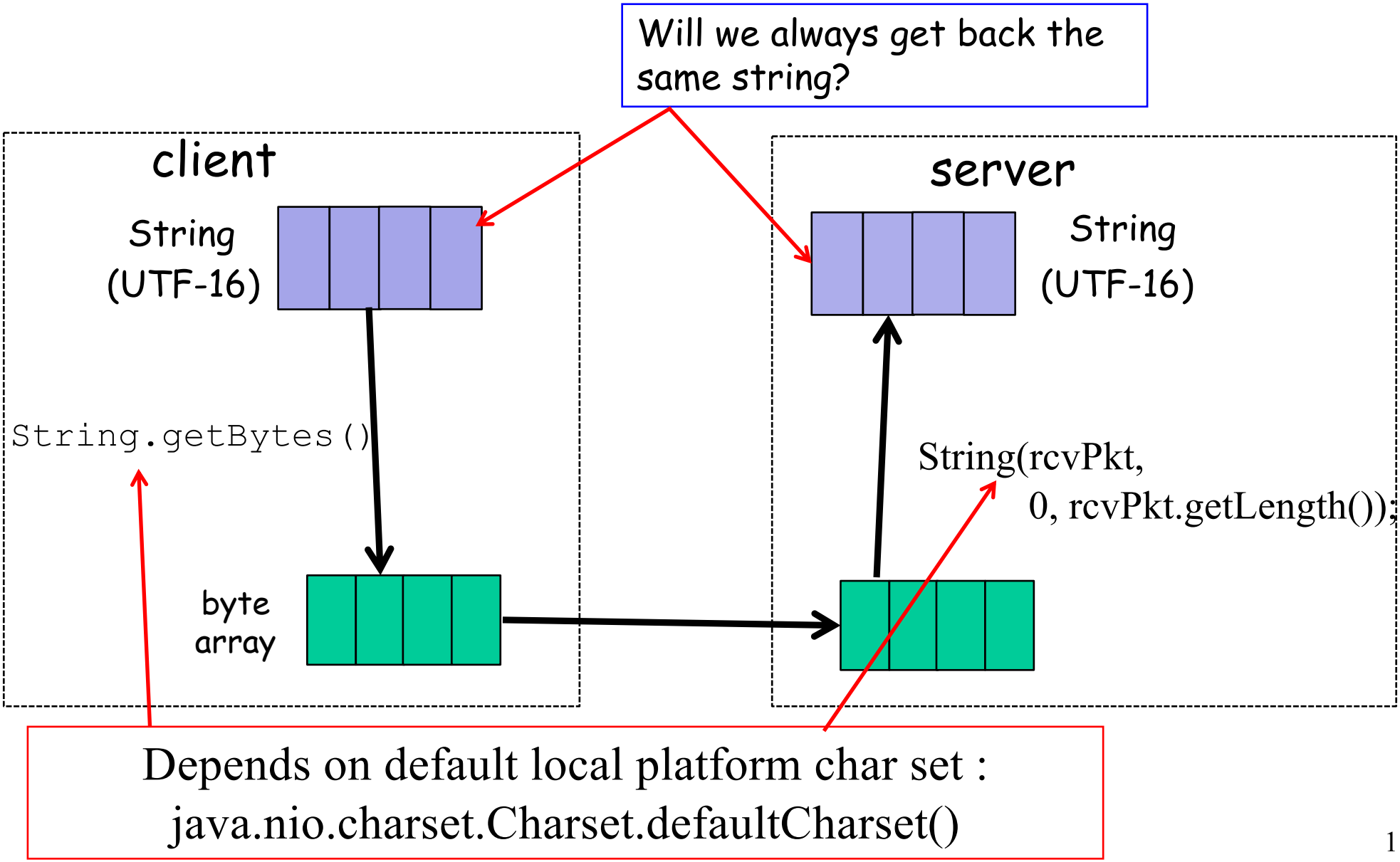




ARM, Power PC, Motorola 68k, IA-64 Intel x86

- sent != received: take an int on a big-endian machine and send a little-endian machine

Example: String and Chars



Example: Charset Troubles

- Try
  - java EncodingDecoding UTF-8 UTF-16

Encoding/Decoding as a Common Source of Errors

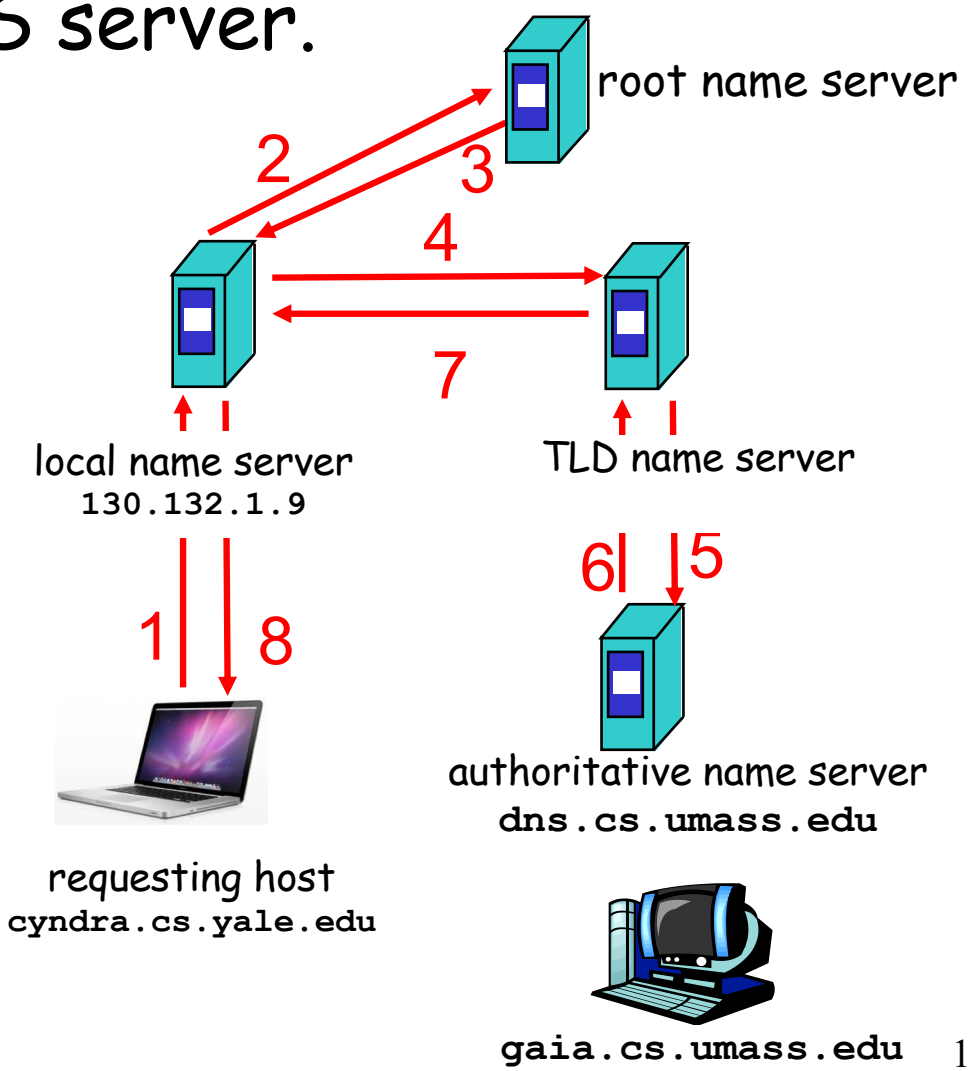
- Please read chapter 2 (Streams) of Java Network Programming for more details
  - Java stream, reader/writer can always be confusing, but it is good to finally understand

- Common mistake even in many (textbook) examples:
  - http://www.java2s.com/Code/Java/Network-Protocol/UseDatagramSockettosendoutandreceiveDatagramPacket.htm

Exercise: UDP/DNS Server Pseudocode

- Modify the example UDP server code to implement a local DNS server.

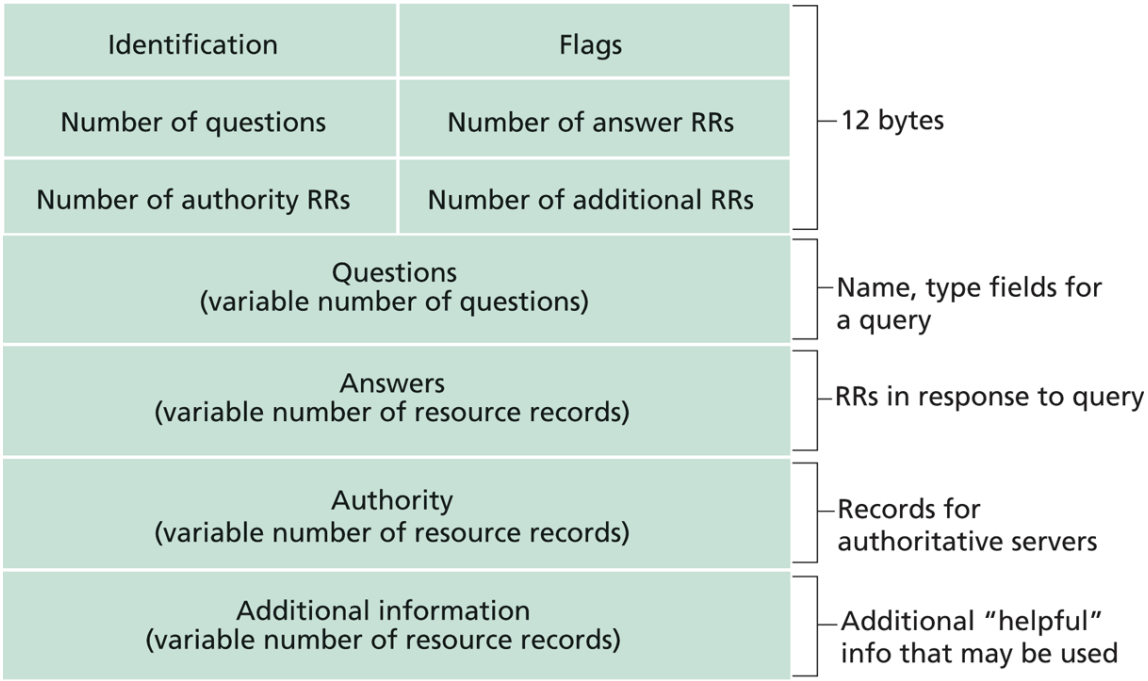
Identification	Flags	12 bytes
Number of questions	Number of answer RRs	
Number of authority RRs	Number of additional RRs	
Questions (variable number of questions)		Name, type fields for a query
Answers (variable number of resource records)		RRs in response to query
Authority (variable number of resource records)		Records for authoritative servers
Additional information (variable number of resource records)		Additional "helpful" info that may be used



UDP/DNS Implementation

- Standard UDP demultiplexing (find out return address by src.addr/src.port of UDP packet) does not always work

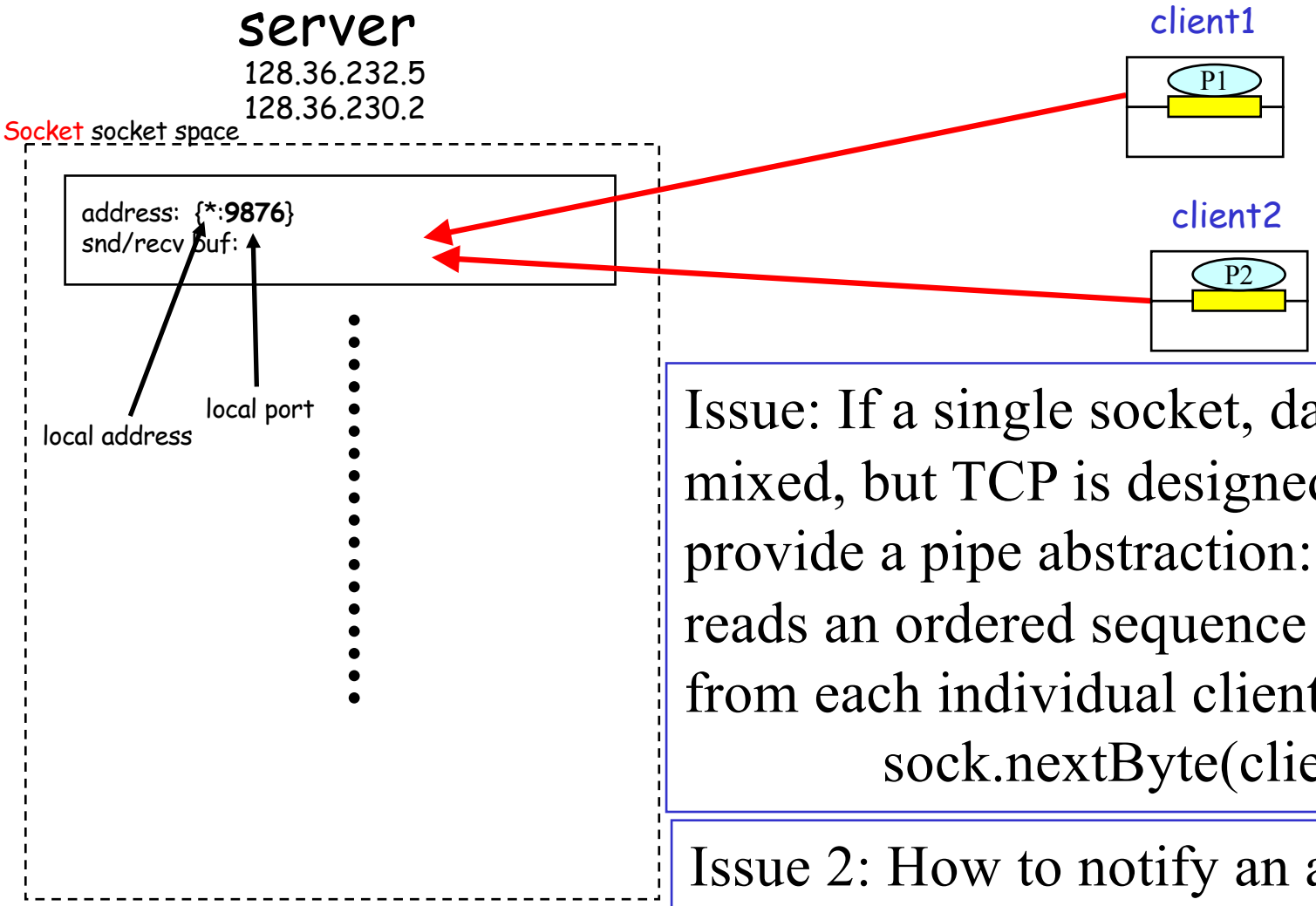
- DNS solution: identification: remember the mapping



Outline

- Admin. and recap
- Network application programming
  - Overview
  - UDP
  - *Basic TCP*

TCP Socket Design: Starting w/ UDP



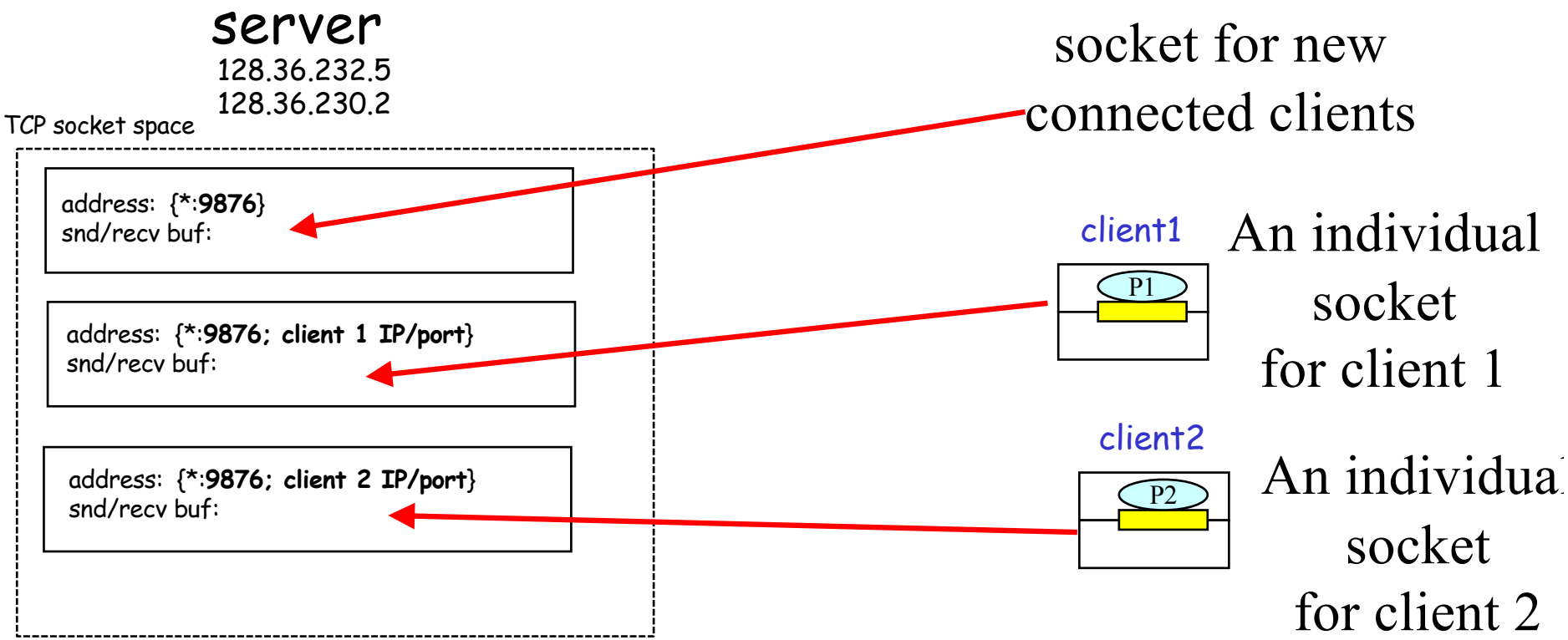
Issue: If a single socket, data can be mixed, but TCP is designed to provide a pipe abstraction: server reads an ordered sequence of bytes from each individual client.

sock.nextByte(client1)?

Issue 2: How to notify an app that a new client is connected?

newClient = sock.getNewClient()?

BSD TCP Socket API Design



Q: How to decide where to put a new TCP packet?

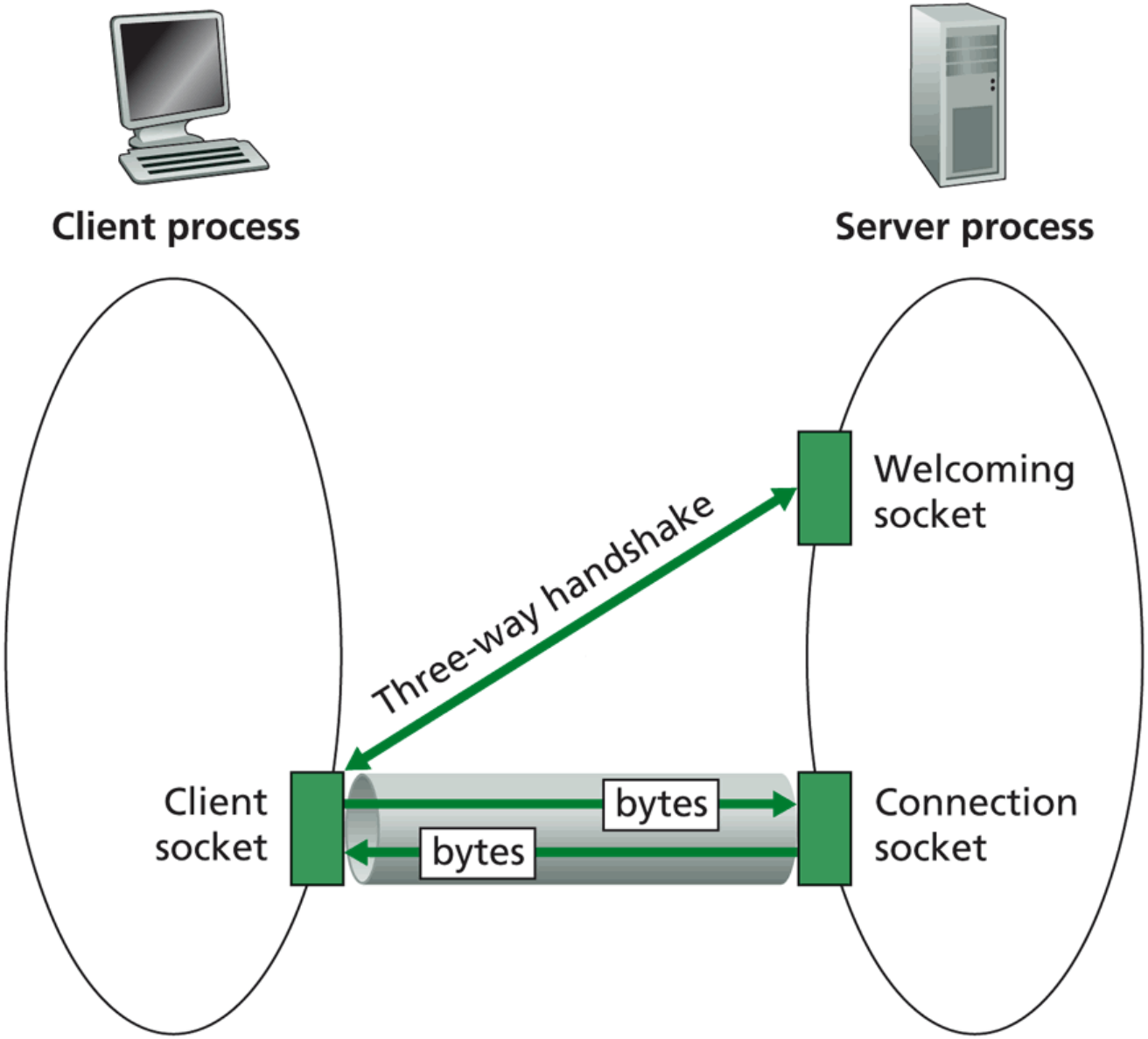
A: Packet demutiplexing is based on four tuples:
(dst addr, dst port, src addr, src port)

TCP 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
  - different connections/sessions are automatically separated into different sockets

-Welcome socket: the waiting room
-connSocket: the operation room

TCP Socket Big Picture



Client/server Socket Workflow: TCP

Server (running on `hostid`)

create socket,
port=`x`, for
incoming request:
welcomeSocket =
ServerSocket(x)

wait for incoming
connection request
connectionSocket =
welcomeSocket.accept()

read request from
connectionSocket

write reply to
connectionSocket

close
connectionSocket

TCP connection setup

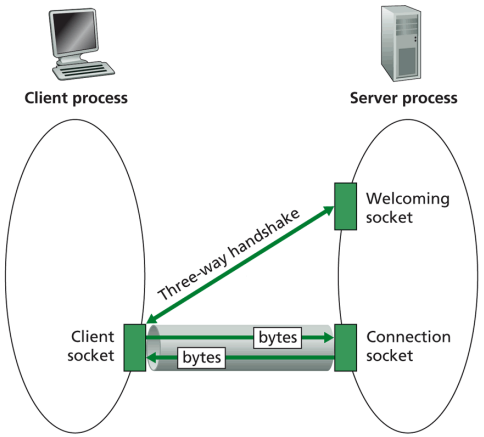
Client

create socket,
connect to hostid, port=`x`
clientSocket =
Socket()

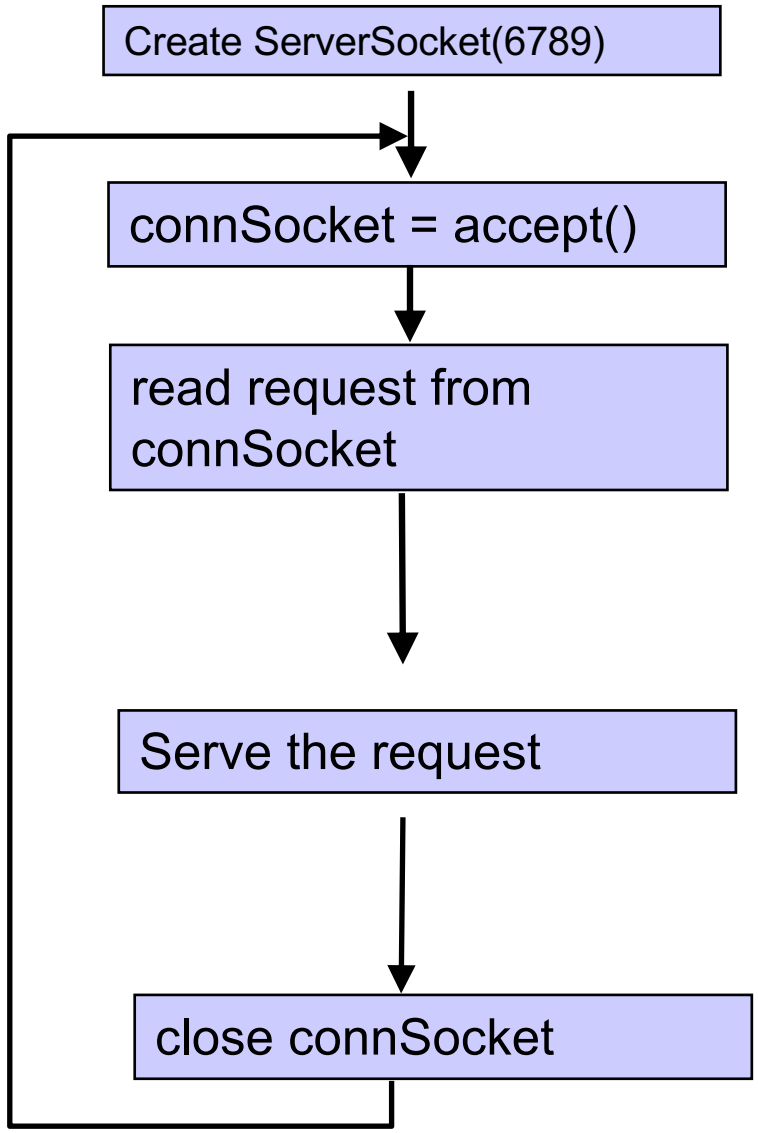
send request using
clientSocket

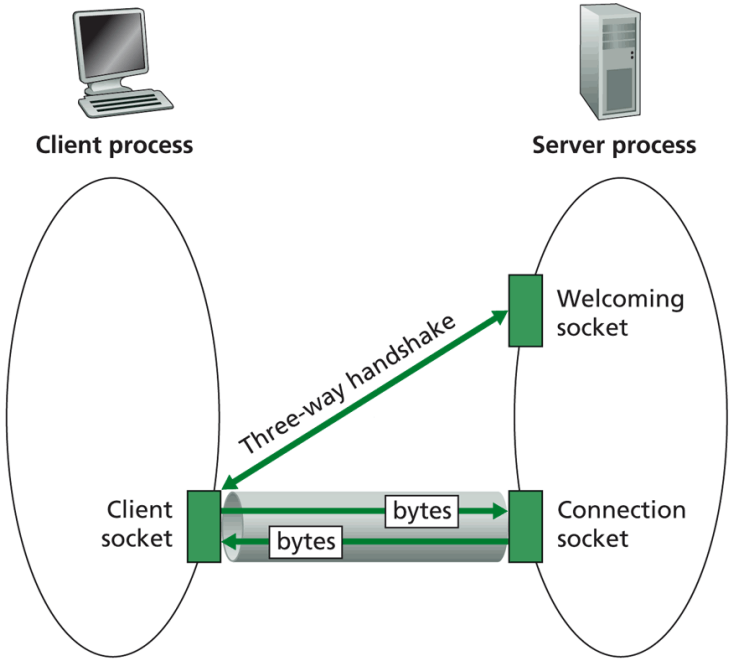
read reply from
clientSocket

close
clientSocket



Server Flow





-Welcome socket: the waiting room
-connSocket: the operation room

ServerSocket

- **ServerSocket**()
  - creates an unbound server socket.
- **ServerSocket**(int port)
  - creates a server socket, bound to the specified port.
- **ServerSocket**(int port, int backlog)
  - creates a server socket and binds it to the specified local port number, with the specified backlog.
- **ServerSocket**(int port, int backlog, InetAddress bindAddr)
  - creates a server with the specified port, listen backlog, and local IP address to bind to.

- **bind**(SocketAddress endpoint)
  - binds the ServerSocket to a specific address (IP address and port number).
- **bind**(SocketAddress endpoint, int backlog)
  - binds the ServerSocket to a specific address (IP address and port number).

- Socket **accept**()
  - listens for a connection to be made to this socket and accepts it.

- **close**()
  closes this socket.

(Client)Socket

- **Socket**(InetAddress address, int port)
  - creates a stream socket and connects it to the specified port number at the specified IP address.
- **Socket**(InetAddress address, int port, InetAddress localAddr, int localPort)
  - creates a socket and connects it to the specified remote address on the specified remote port.
- **Socket**(String host, int port)
  - creates a stream socket and connects it to the specified port number on the named host.

- **bind**(SocketAddress bindpoint)
  - binds the socket to a local address.

- **connect**(SocketAddress endpoint)
  - connects this socket to the server.
- **connect**(SocketAddress endpoint, int timeout)
  - connects this socket to the server with a specified timeout value.

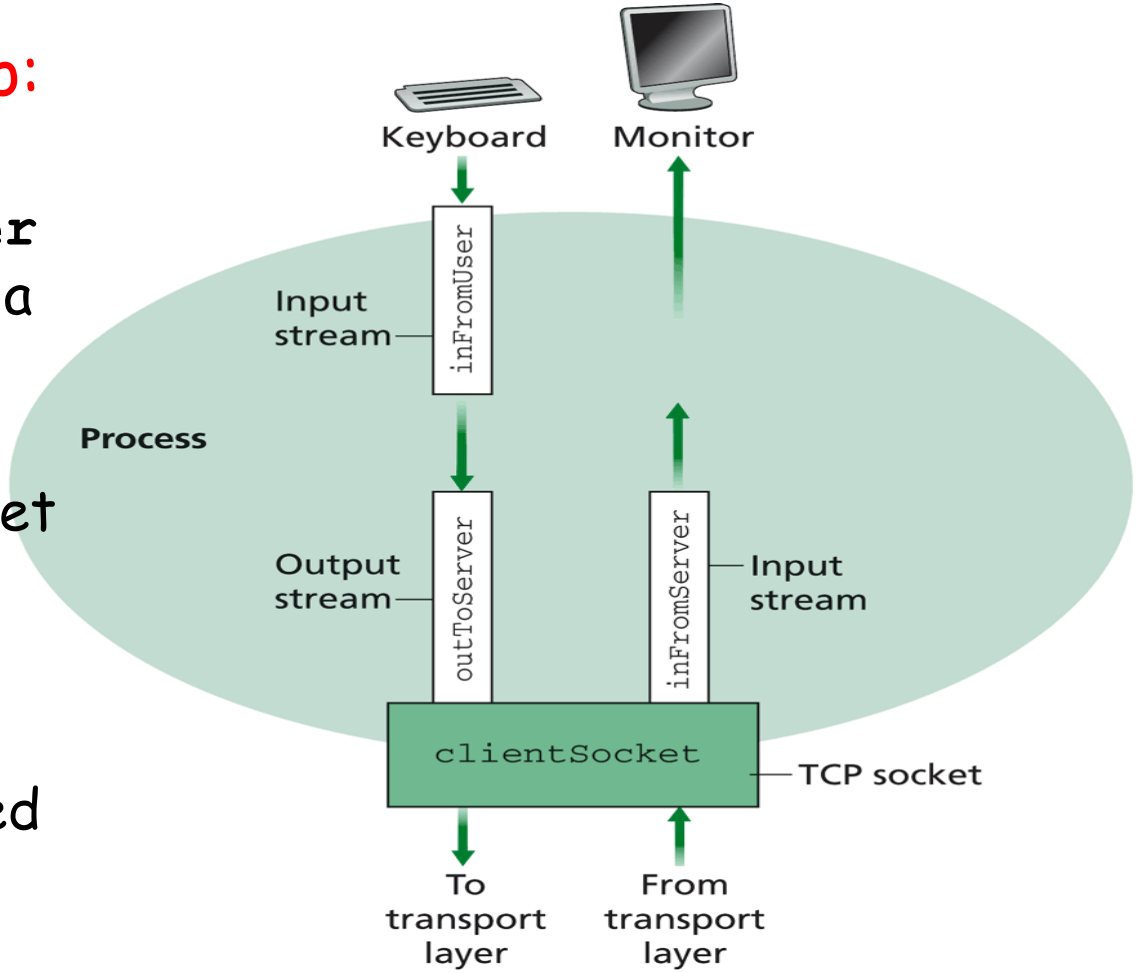
- InputStream **getInputStream**()
  - returns an input stream for this socket.
- OutputStream **getOutputStream**()
  - returns an output stream for this socket.

- **close**()
  closes this socket.

Simple TCP Example

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)



Example: Java client (TCP)

```
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));
        sentence = inFromUser.readLine();

        Socket clientSocket = new Socket("server.name", 6789);

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

Create input stream

Create client socket, connect to server

Create output stream attached to socket

OutputStream

- public abstract class OutputStream
  - public abstract void write(int b) throws IOException
  - public void write(byte[] data) throws IOException
  - public void write(byte[] data, int offset, int length) throws IOException
  - public void flush( ) throws IOException
  - public void close( ) throws IOException

InputStream

- public abstract class InputStream
  - public abstract int read( ) throws IOException
  - public int read(byte[] input) throws IOException
  - public int read(byte[] input, int offset, int length) throws IOException
  - public long skip(long n) throws IOException
  - public int available( ) throws IOException
  - public void close( ) throws IOException

Example: Java client (TCP), cont.

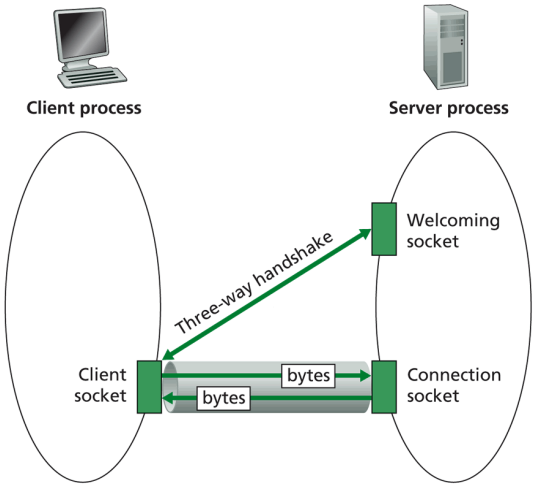
Send line to server

Create input stream attached to socket

Read line from server

```
outToServer.writeBytes(sentence + '\n');

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

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);
```

Create welcoming socket at port 6789

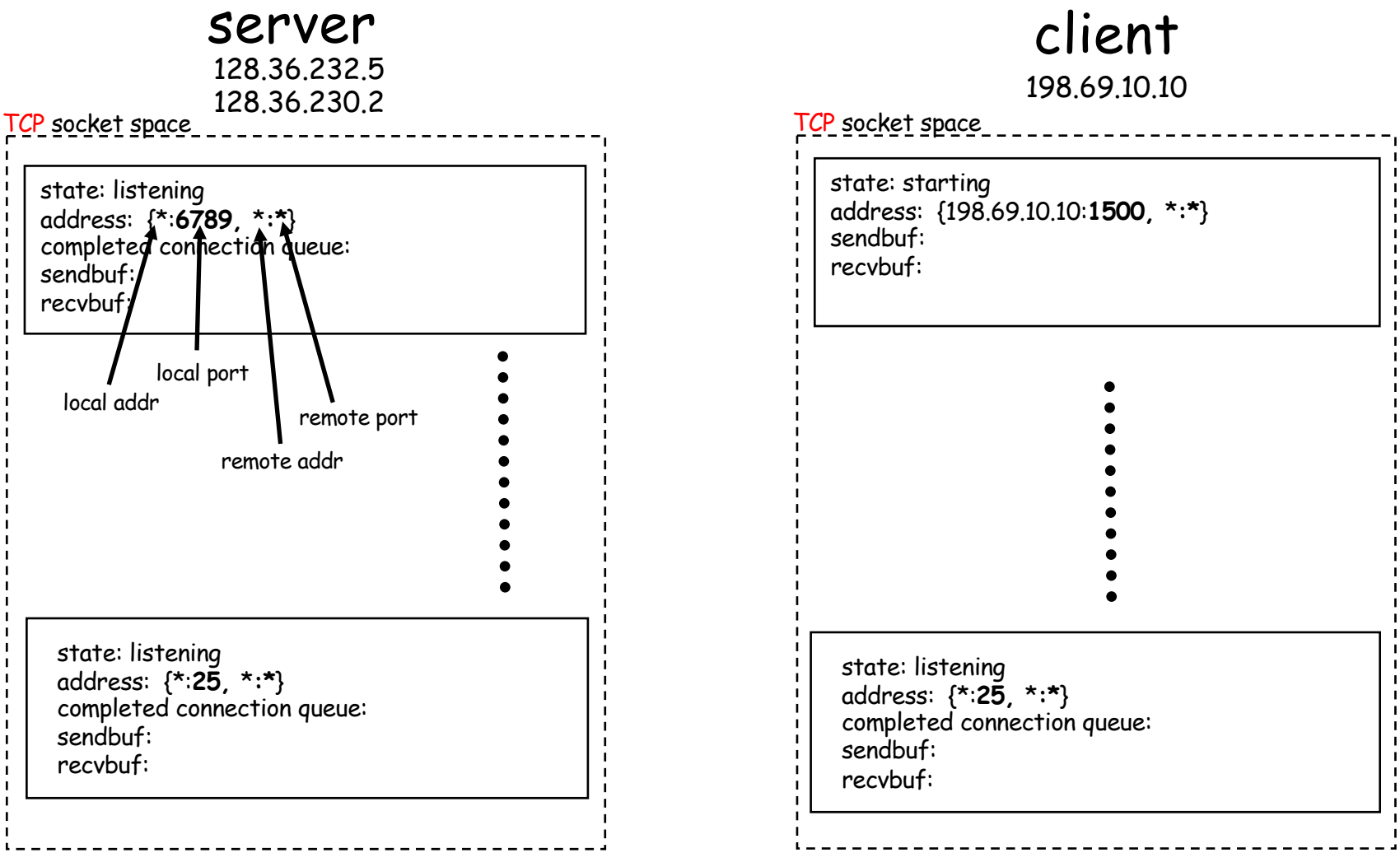
Demo

% on MAC

start TCPServer

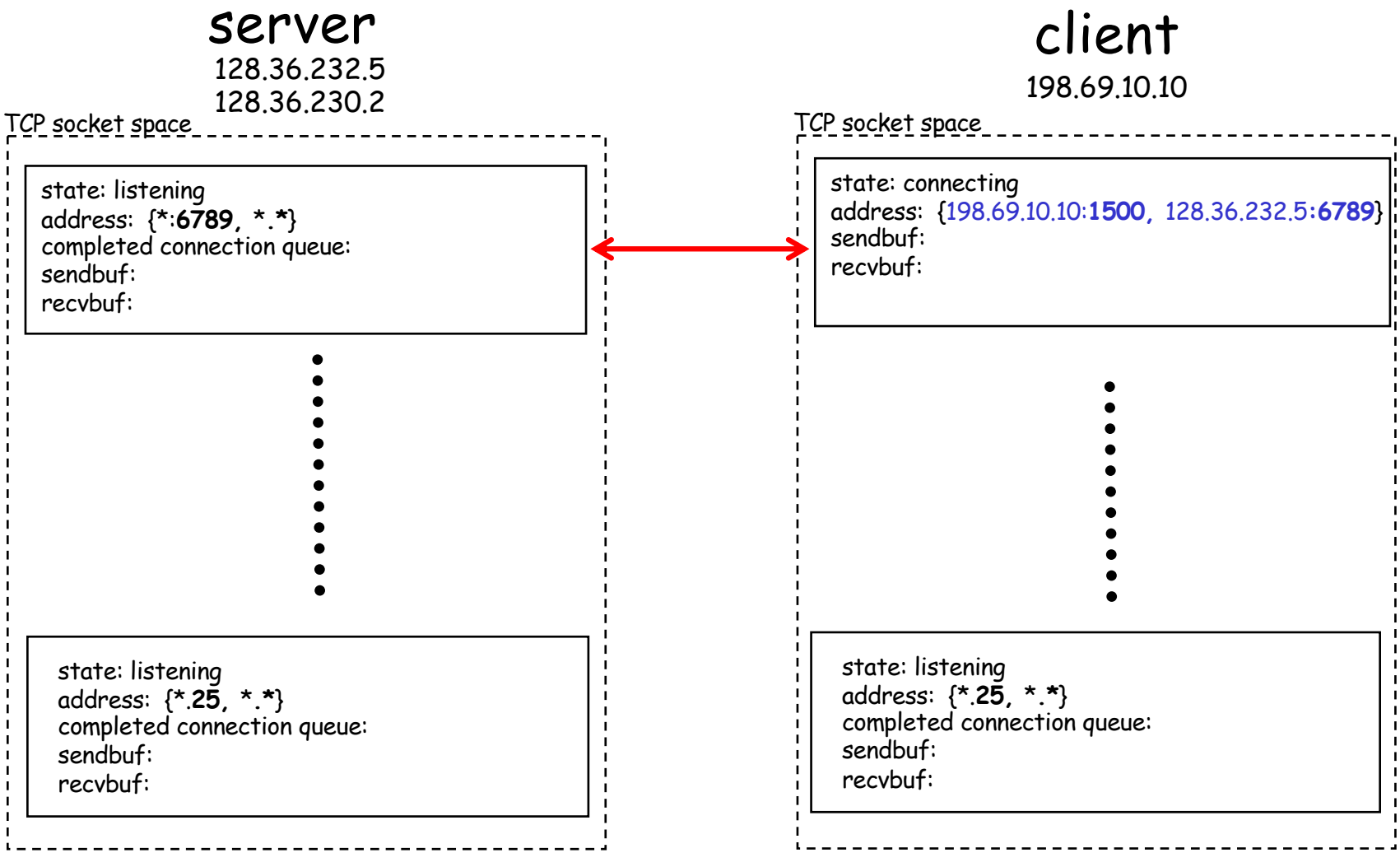
wireshark to capture our TCP traffic
tcp.srcport==6789 or tcp.dstport==6789

Under the Hood: After Welcome (Server) Socket



%netstat -p tcp -n -a

After Client Initiates Connection



%ubuntu java TCPClient <server> 6789

Example: Client Connection Handshake Done

server
128.36.232.5
128.36.230.2

TCP socket space

state: listening
address: {*:**6789**, *:*}
completed connection queue:
{128.36.232.5.**6789**, 198.69.10.10.**1500**}
sendbuf:
recvbuf:

state: listening
address: {*:**25**, *:*}
completed connection queue:
sendbuf:
recvbuf:

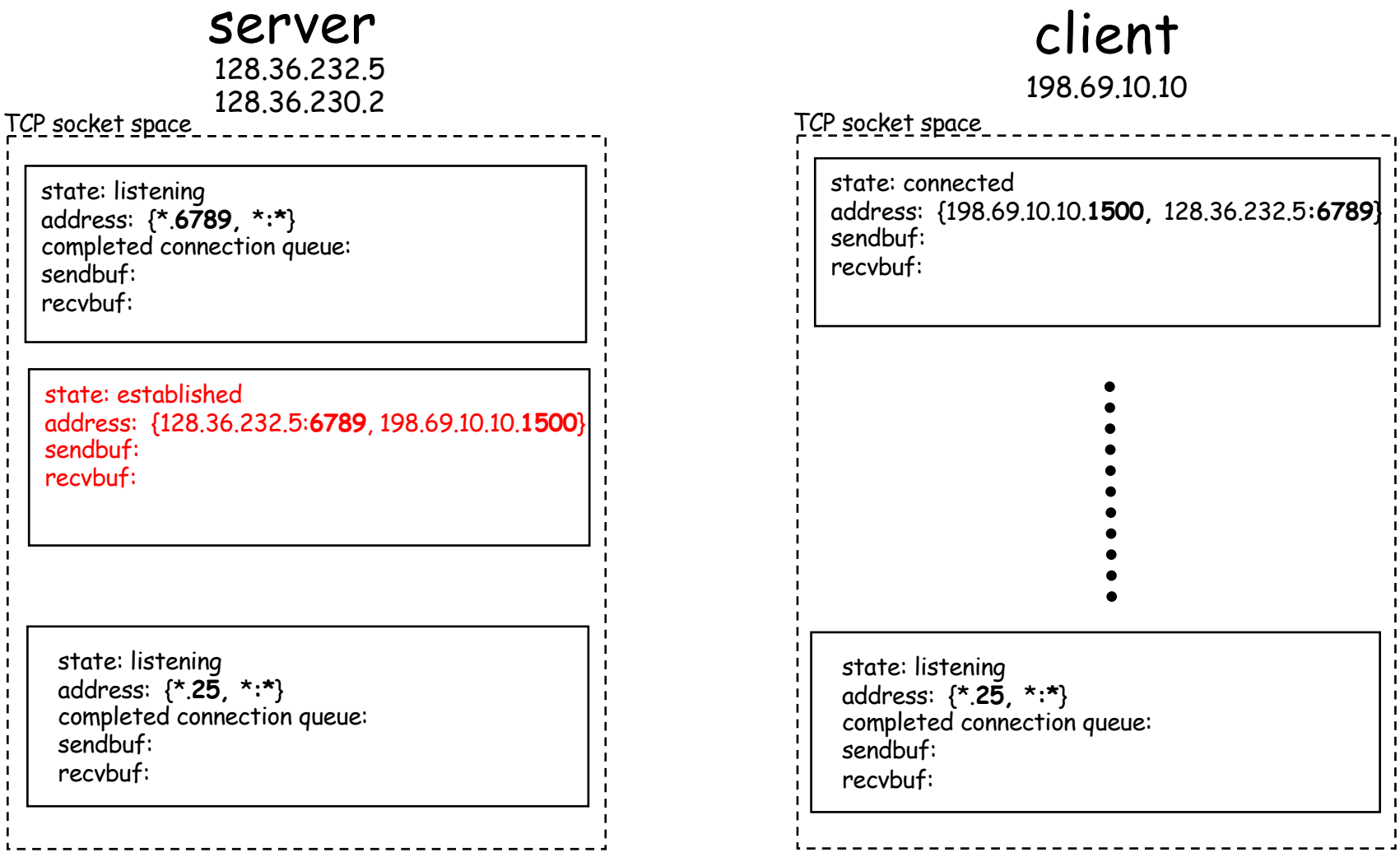
client
198.69.10.10

TCP socket space

state: connected
address: {198.69.10.10:**1500**, 128.36.232.5:**6789**}
sendbuf:
recvbuf:

state: listening
address: {*:**25**, *:*}
completed connection queue:
sendbuf:
recvbuf:

Example: Client Connection Handshake Done



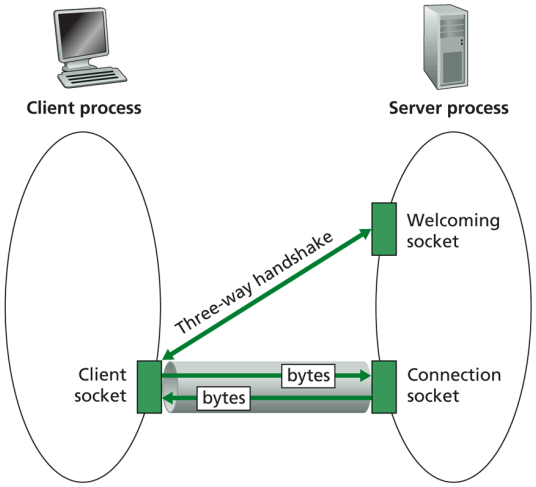
Packet demutiplexing is based on (dst addr, dst port, src addr, src port)

Packet sent to the socket with the best match!

Demo

- What if more client connections than backlog allowed?
  - We continue to start java TCPClient

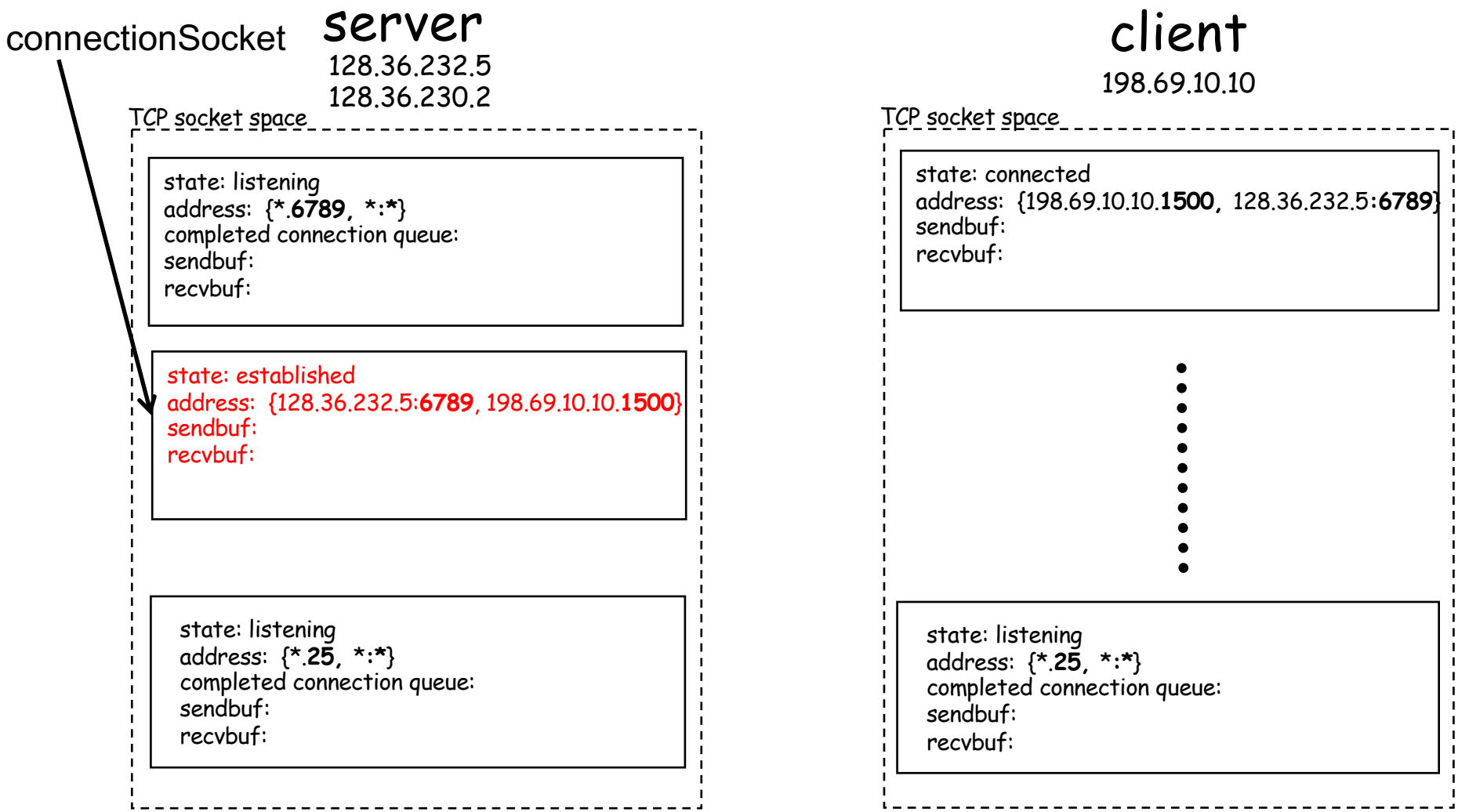
Example: Java server (TCP)




```
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();
```

Wait, on welcoming socket for contact by client

Example: Server accept()



Example: Java server (TCP): Processing

Create input stream, attached to socket

```
BufferedReader inFromClient =
    new BufferedReader(new
        InputStreamReader(connectionSocket.getInputStream()));
```

Read in line from socket

```
clientSentence = inFromClient.readLine();

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

```
        }
    }
}
```

Example: Java server (TCP): Output

Create output stream, attached to socket

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

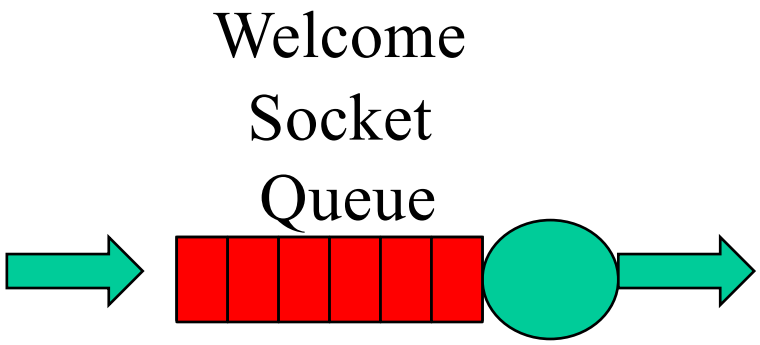
Write out line to socket

```
outToClient.writeBytes(capitalizedSentence);
    }
  }
}
```

End of while loop, loop back and wait for another client connection

Analysis

- Assume that client requests arrive at a rate of lambda/second
- Assume that each request takes $1/\mu$ seconds
- A basic question
  - How big is the backlog (welcome queue)



Analysis

- Is there any interop issue in the sample program?

Analysis

- Is there any interop issue in the sample program?
  - DataOutputStream writeBytes(String) truncates
    - http://docs.oracle.com/javase/1.4.2/docs/api/java/io/DataOutputStream.html#writeBytes(java.lang.String)

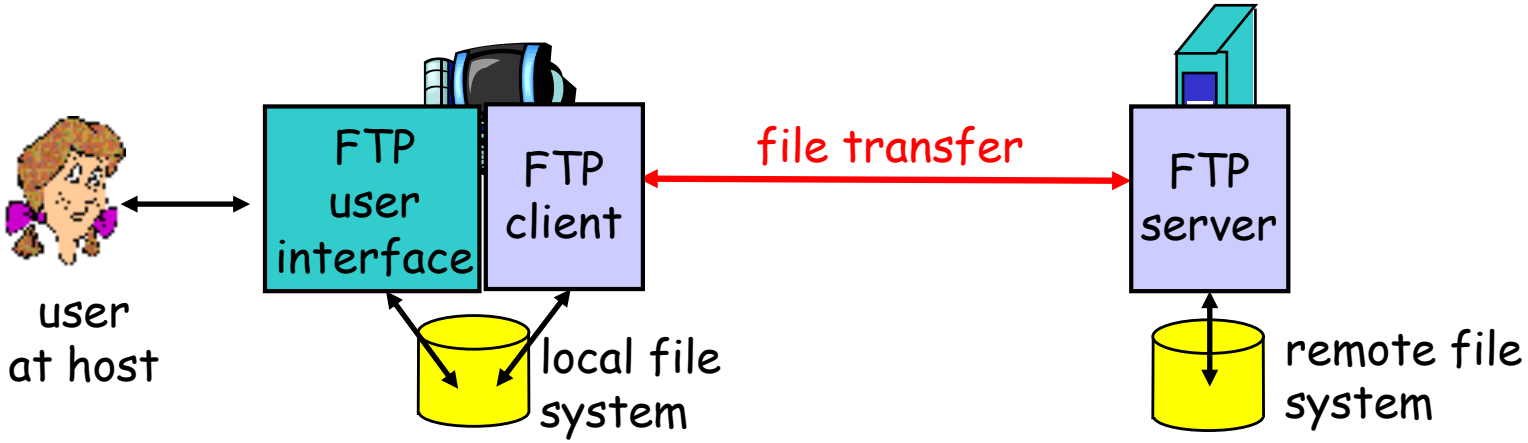
Summary: Basic Socket Programming

- They are relatively straightforward
  - UDP: DatagramSocket
  - TCP: ServerSocket, Socket
- The main function of socket is multiplexing/demultiplexing to application processes
  - UDP uses (dst IP, port)
  - TCP uses (src IP, src port, dst IP, dst port)
- Always pay attention to encoding/decoding

Outline

- Admin. and recap
- Network application programming
  - UDP sockets
  - TCP sockets
- Network applications (continue)
  - *File transfer (FTP) and extension*

FTP: the File Transfer Protocol



- Transfer files to/from remote host
- Client/server model
  - *client:* side that initiates transfer (either to/from remote)
  - *server:* remote host
- ftp: RFC 959
- ftp server: port 21/20 (smtp 25, http 80)

FTP Commands, Responses

Sample commands:

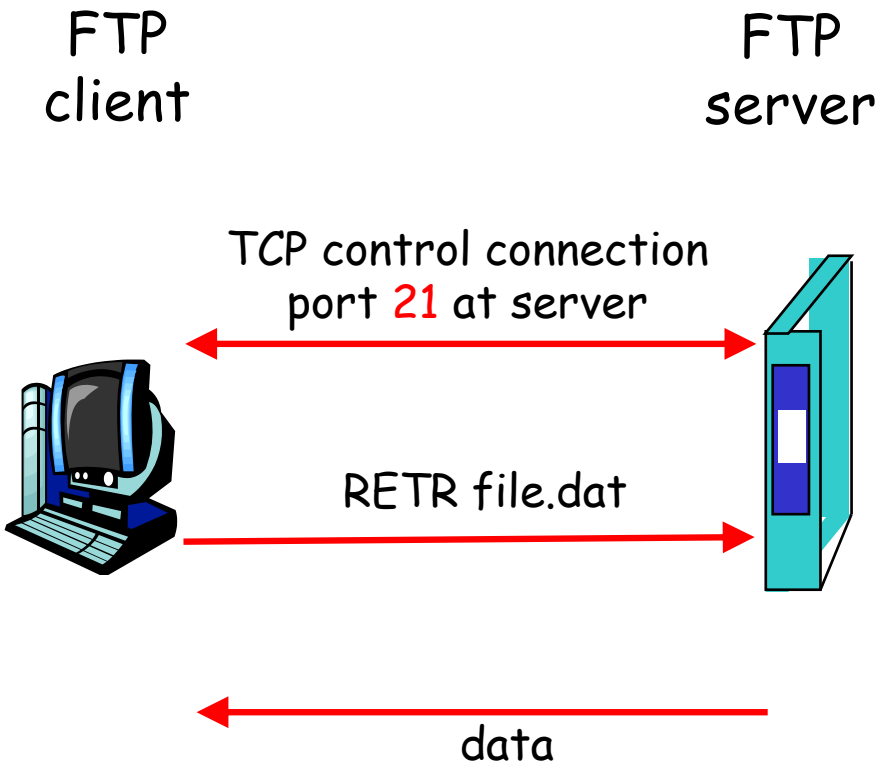
- sent as ASCII text over control channel
- **USER *username***
- **PASS *password***
- **PWD** returns current dir
- **STAT** shows server status
- **LIST** returns list of file in current directory
- **RETR filename** retrieves (gets) file
- **STOR filename** stores file

Sample return codes

- status code and phrase
- **331 Username OK, password required**
- **125 data connection already open; transfer starting**
- **425 Can't open data connection**
- **452 Error writing file**

FTP Protocol Design

- What is the simplest design of data transfer?



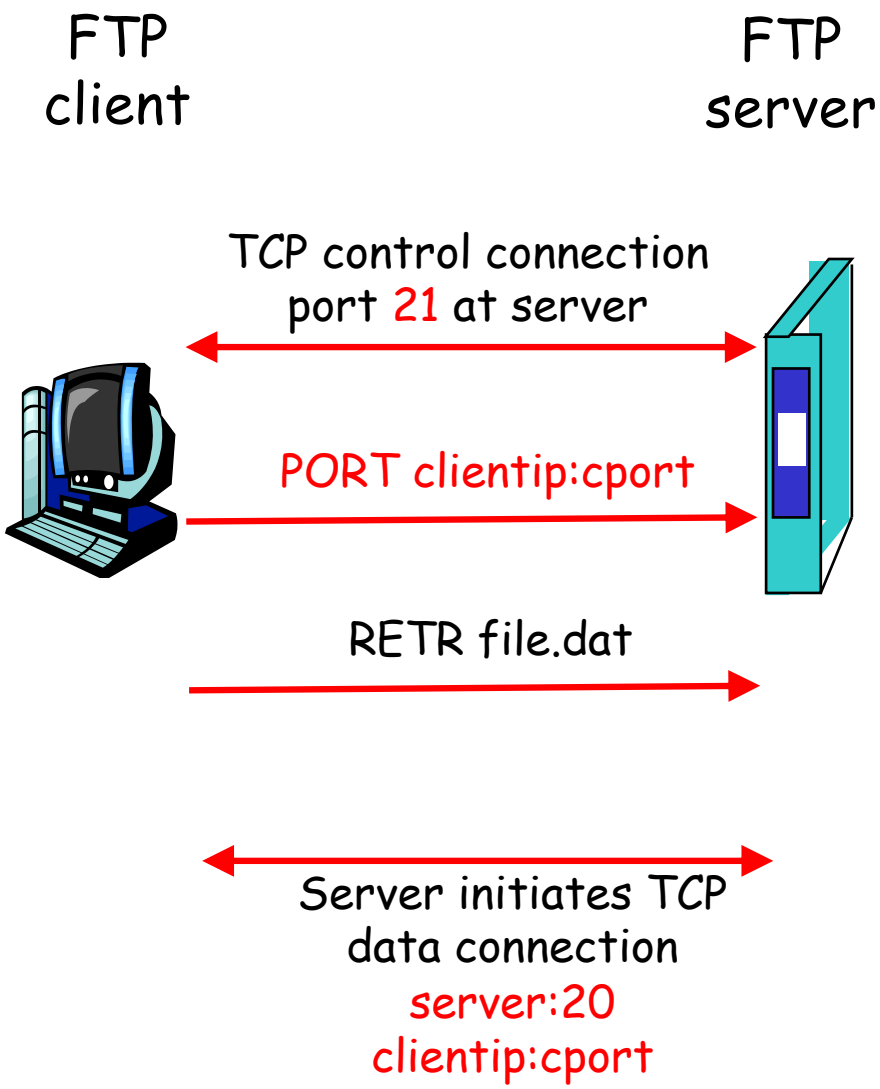
FTP: A Client-Server Application with Separate Control, Data Connections

- Two types of TCP connections opened:
  - A control connection: exchange commands, responses between client, server.
    "out of band control"
  - Data connections: each for file data to/from server

Discussion: why does FTP separate control/data connections?

Q: How to create a new data connection?

Traditional FTP: Client Specifies Port for Data Connection



Example using telnet/nc

- Use telnet for the control channel
  - telnet ftp.ietf.org 21
  - user anonymous
  - pass your_email
  - port 10,90,61,172,4,1
  - list

client IP address

port number

- use nc (NetCat) to receive/send data with server
  - nc –v –l 1025

Problem of the Client PORT Approach

- Many Internet hosts are behind NAT/firewalls that block connections initiated from outside