Network Applications: TCP Network Programming; File Transfer Protocol

Y. Richard Yang

http://zoo.cs.yale.edu/classes/cs433/

9/25/2018

Outline

- Admin and recap
- Basic network applications
  - Email
  - DNS
- Network application programming
  - UDP sockets
  - TCP sockets
- Network applications (continue)
  - File transfer (FTP) and extension

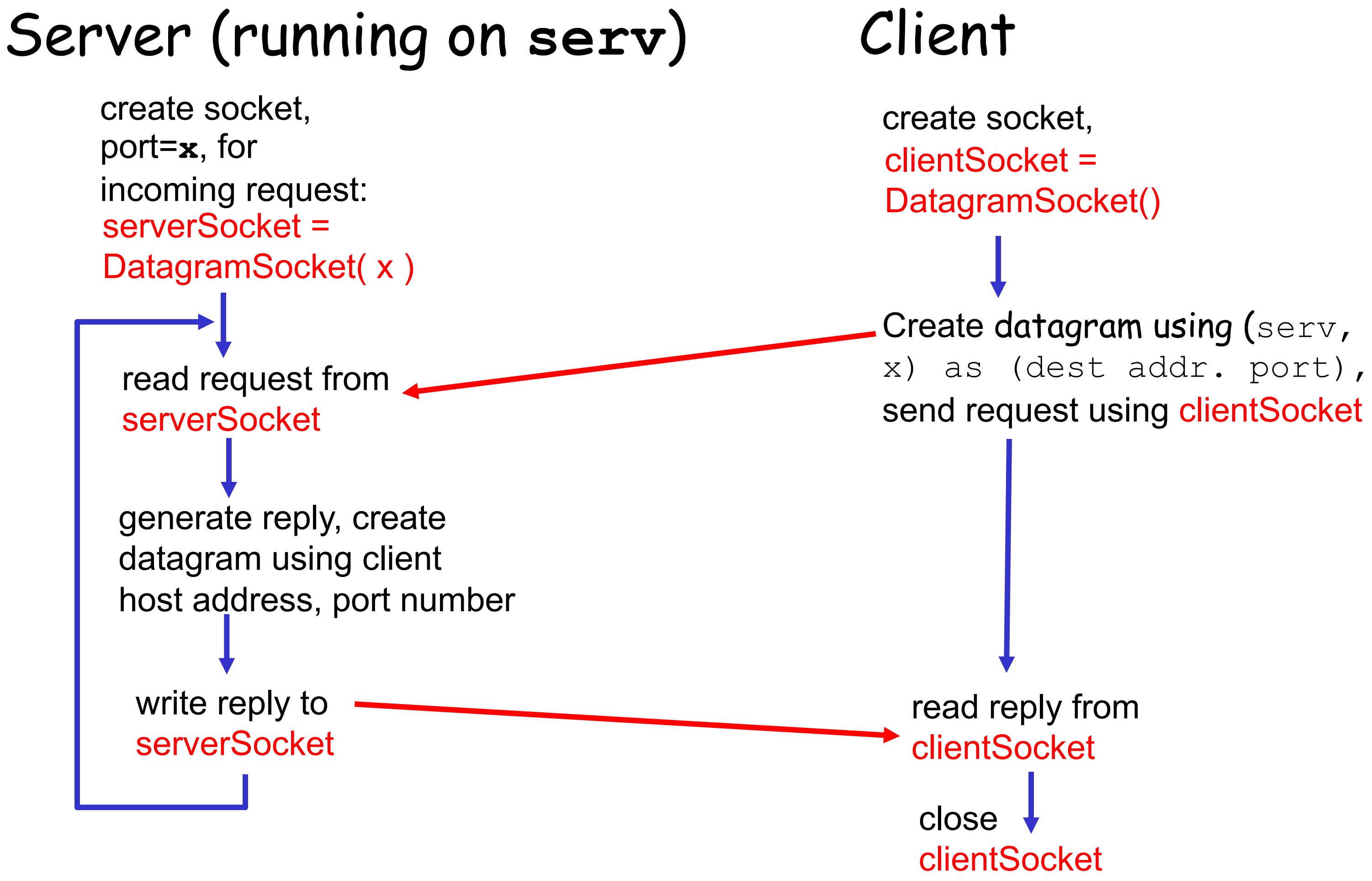
Admin

- Assignment one returned
  - Check w/ Geng if you have any questions
- Assignment two
  - Due Wednesday next week

Recap: DNS Extension/Alternative

- Many interesting design features, from architecture to message format
- remaining issues
  - Security, e.g., DNSSEC
  - Limited service type, requirement of a server
- Extension
  - mDNS, DNS-SD
- Alternative design
  - Linda

Recap: Connectionless UDP: Big Picture (Java version)



Server (running on **serv**)

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

read request from serverSocket

generate reply, create datagram using client host address, port number

write reply to serverSocket

Client

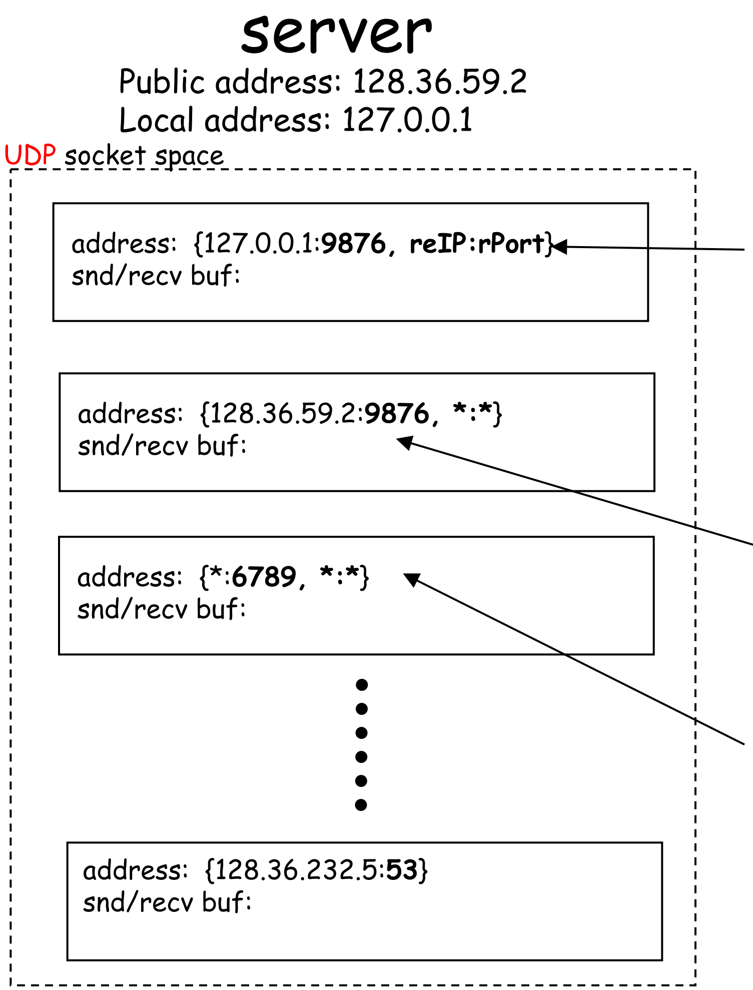
create socket, clientSocket = DatagramSocket()

Create datagram using (`serv, x) as (dest addr. port)`, send request using clientSocket

read reply from clientSocket

close clientSocket

Recap: UDP Sockets and Multiplexing



server

Public address: 128.36.59.2

Local address: 127.0.0.1

UDP socket space

address: {127.0.0.1:**9876**, **reIP**:**rPort**}
snd/recv buf:

address: {128.36.59.2:**9876**, ***:***}
snd/recv buf:

address: {*:**6789**, ***:***}
snd/recv buf:

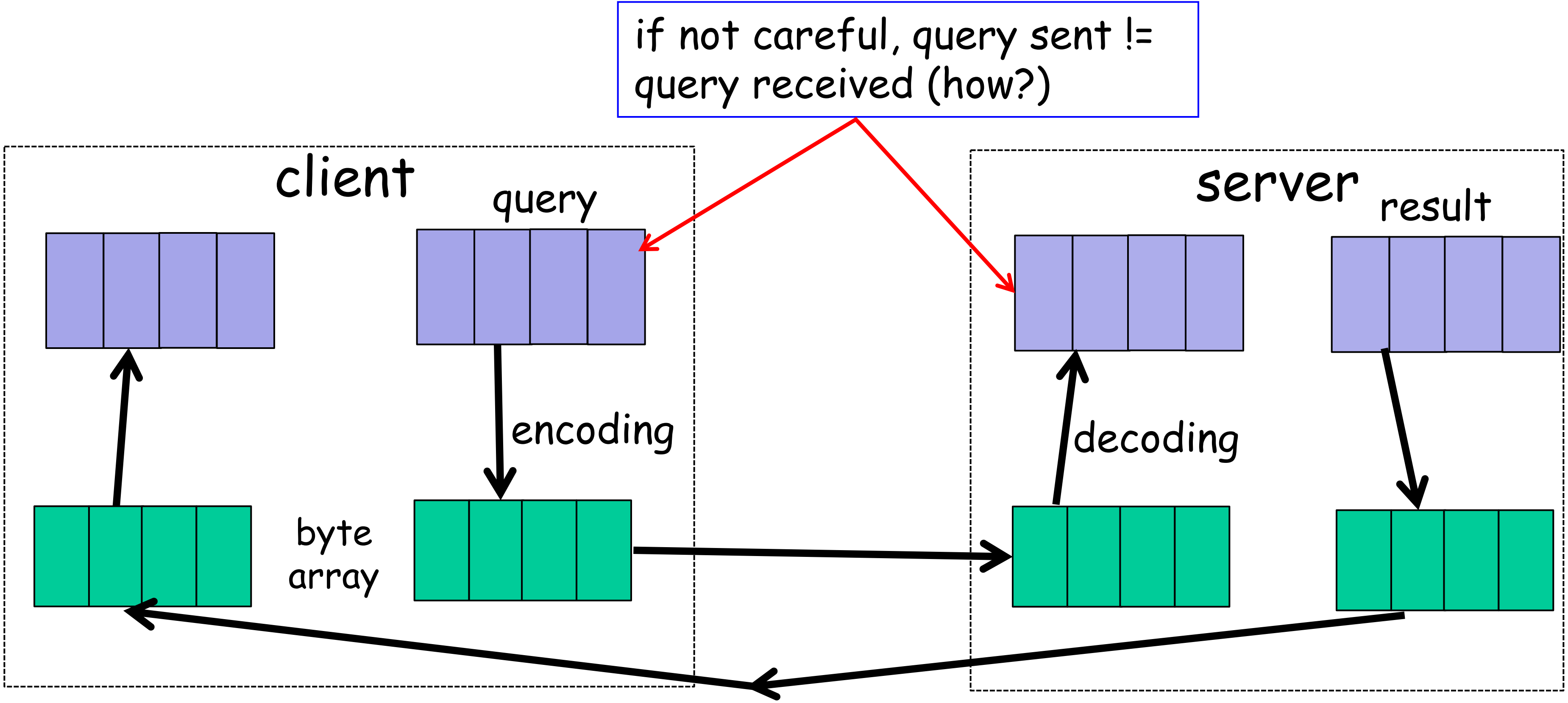
address: {128.36.232.5:**53**}
snd/recv buf:

```
InetAddress sIP1 =
  InetAddress.getByName("localhost");
DatagramSocket ssock1 =
  new DatagramSocket(9876, sIP1);
ssock1.connect( rAddr, rPort );

InetAddress sIP2 =
 InetAddress.getByName("128.36.59.2");
DatagramSocket ssock2 =
  new DatagramSocket(9876,sIP2);

DatagramSocket serverSocket =
  new DatagramSocket(6789);
```

UDP demutiplexing is based on matching filter. But typically we say UDP multiplexes on dest port (in most cases).

Recap: Msg Parsing (Decoding)/Encoding

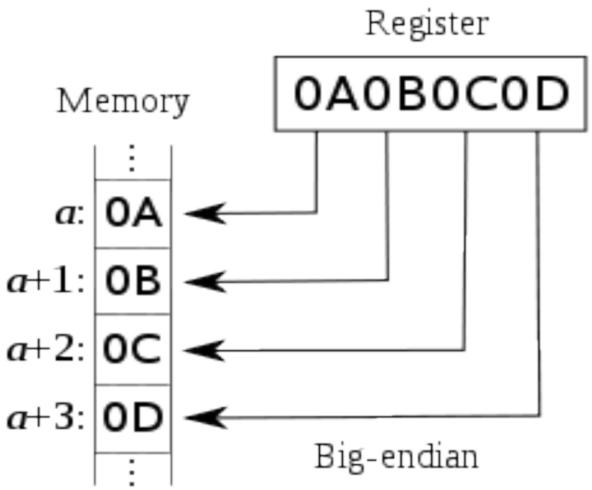
- Typically message parsing and processing is straightforward, with one rule: ALWAYS pay attention to encoding/decoding of data

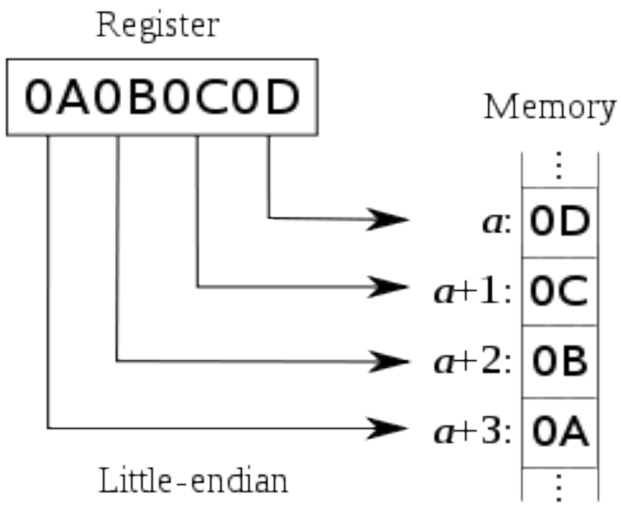


if not careful, query sent != query received (how?)

Example: Endianness of Numbers

- int var = 0x0A0B0C0D



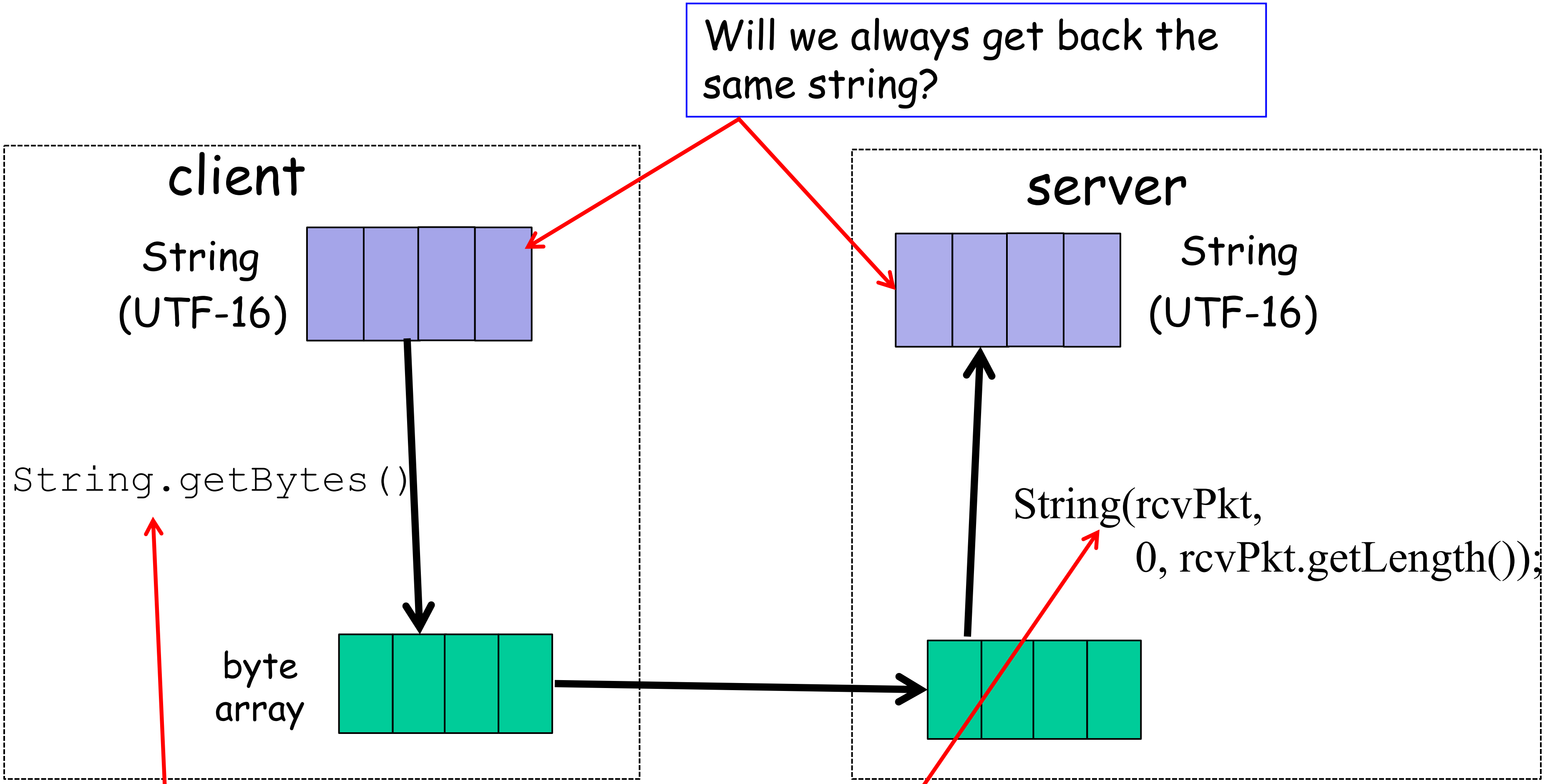


ARM, Power PC, Motorola 68k, IA-64

Intel x86

- sent != received: take an int on a big-endian machine and send a little-endian machine
- Java virtual machine uses big-endian and most networking protocols are big-endian.

Example: String and Chars



Will we always get back the same string?

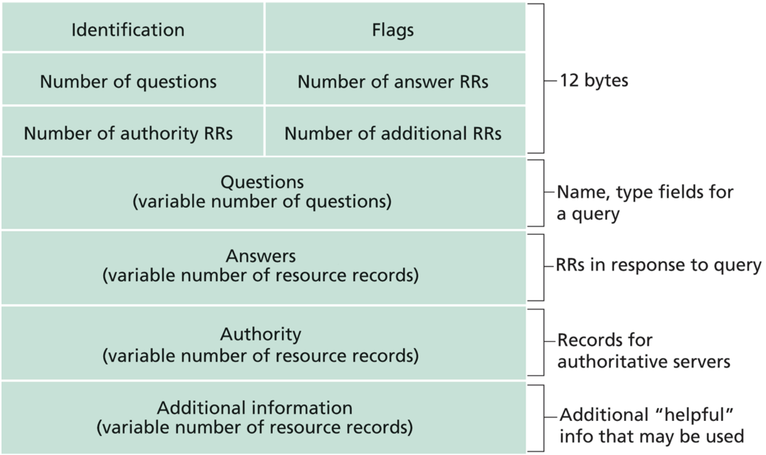
`String.getBytes()`

String(rcvPkt,
0, rcvPkt.getLength());

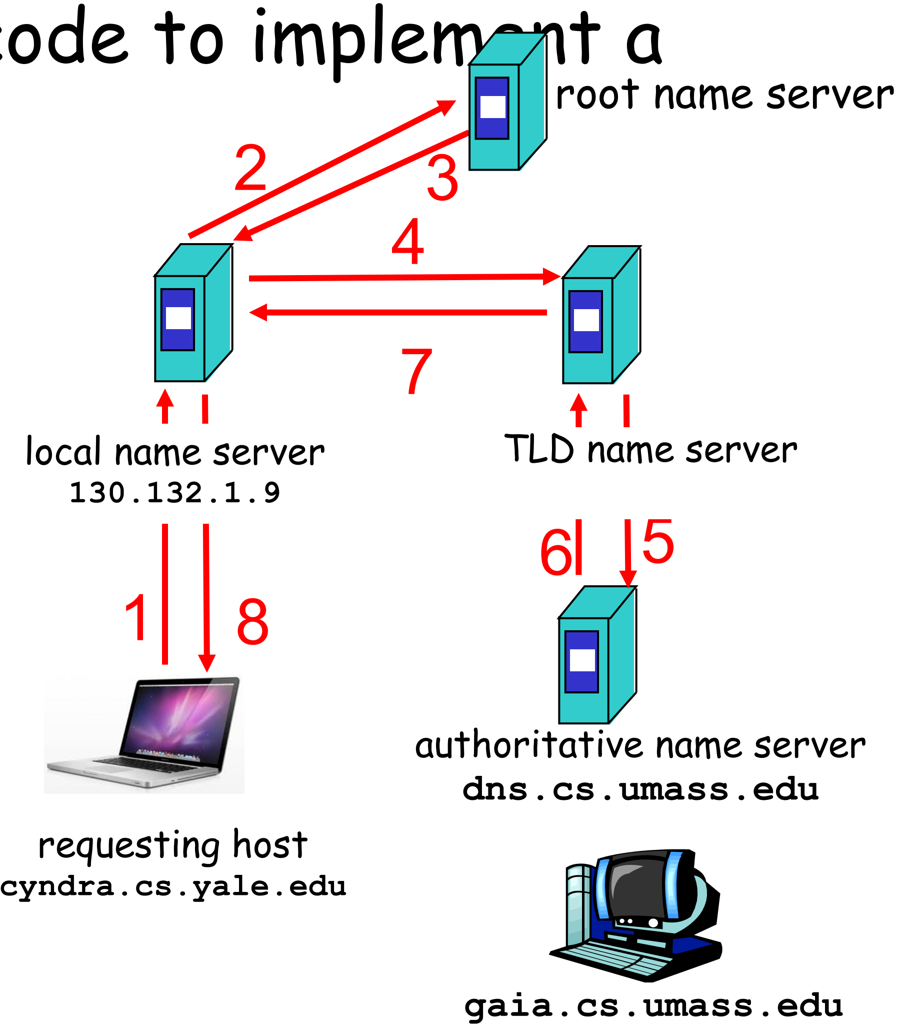
Depends on default local platform char set :
java.nio.charset.Charset.defaultCharset()

Offline Exercise: UDP/DNS Server Pseudocode

- Think about how you may 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



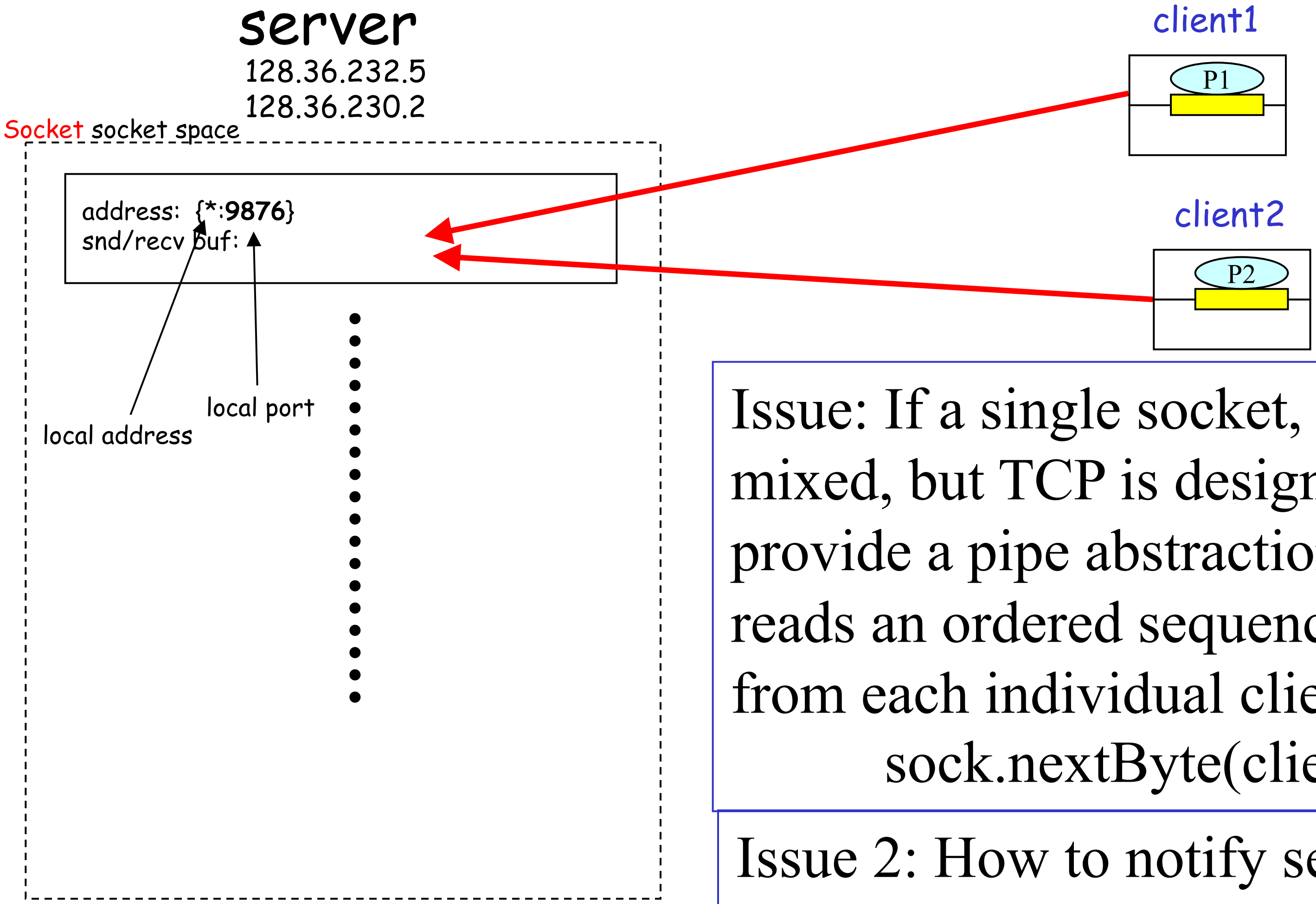
Multicast on Top of UDP

- MulticastSocket is derived from DatagramSocket:
  - https://docs.oracle.com/javase/7/docs/api/java/net/MulticastSocket.html
  - joinGroup specifies the port (UDP filter) to receive packets
- Two simple examples
  - MulticastSniffer.java
  - MulticastSender.java

Outline

- Admin and recap
- Network application programming
  - Overview
  - UDP socket programming
  - Basic TCP socket programming

TCP Socket Design: Starting w/ UDP



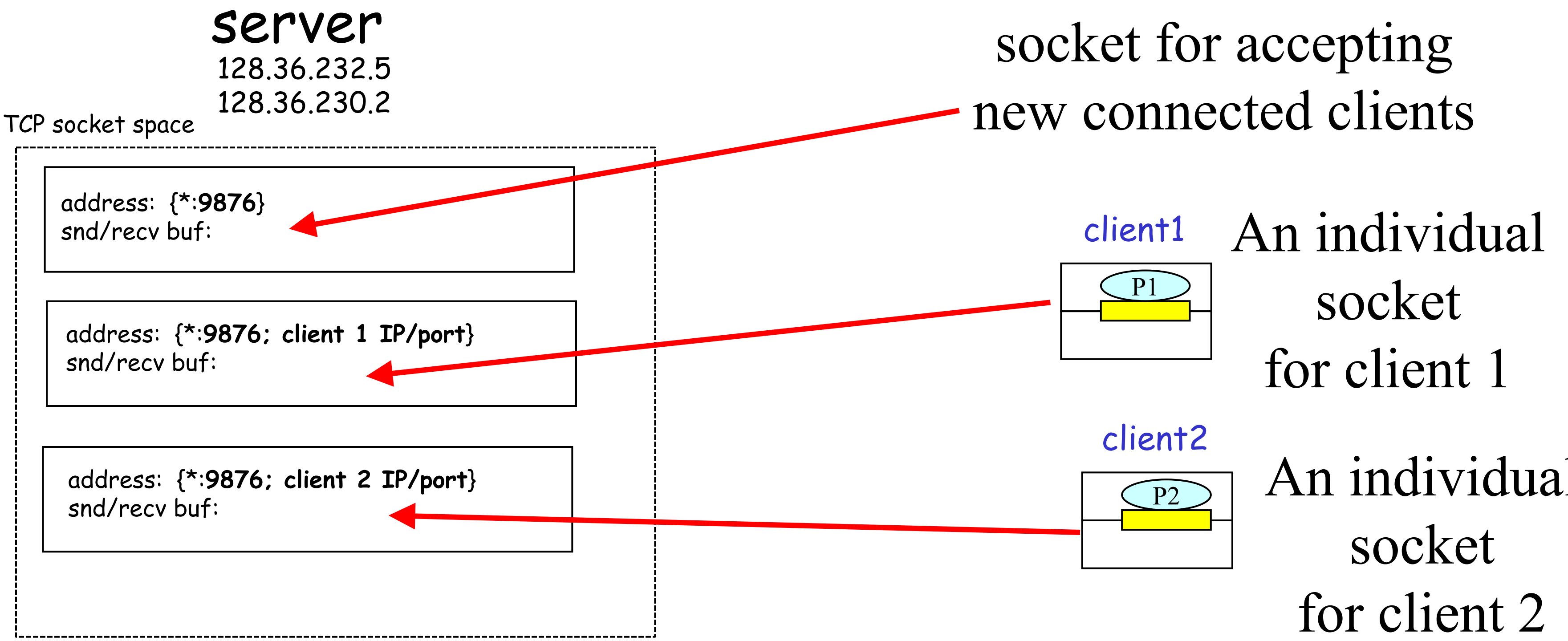
Issue: If a single socket, data can be mixed, but TCP is designed to provide a pipe abstraction: server reads an ordered sequence of bytes from each individual client.

sock.nextByte(client1)?

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

newClient = sock.getNewClient()?

BSD TCP Socket API Design



server
128.36.232.5
128.36.230.2

TCP socket space

address: {*:**9876**}
snd/recv buf:

address: {*:**9876; client 1 IP/port**}
snd/recv buf:

address: {*:**9876; client 2 IP/port**}
snd/recv buf:

socket for accepting new connected clients

client1

An individual socket for client 1

client2

An individual socket for client 2

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

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

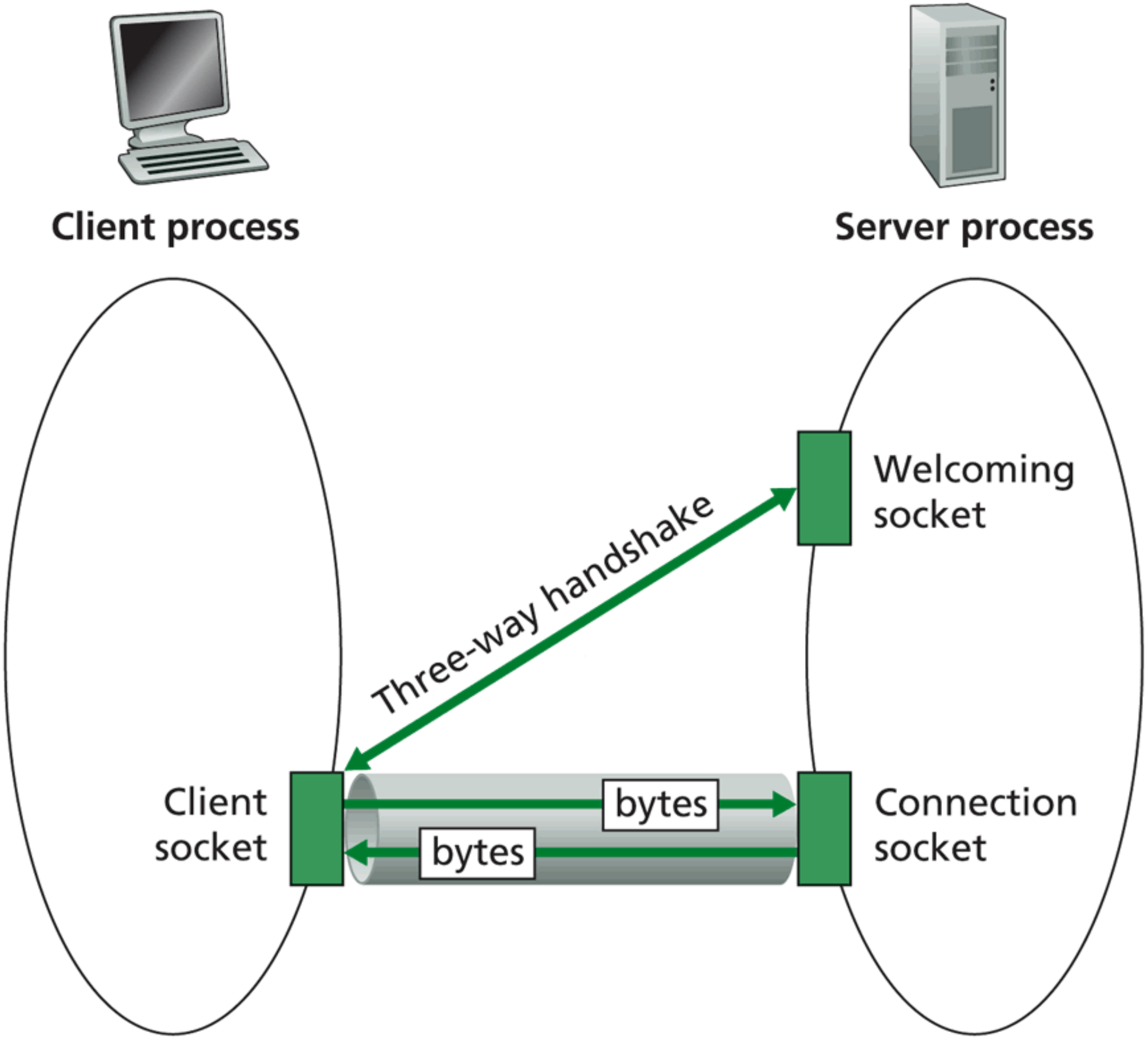
TCP Connection-Oriented Demux

- 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

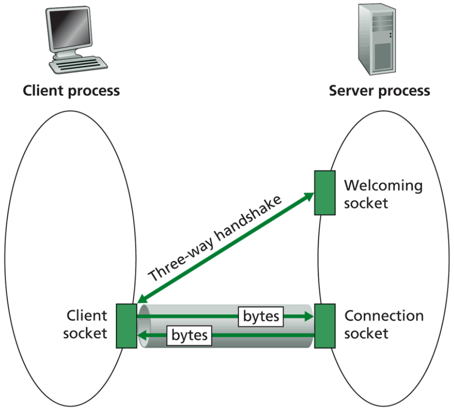
TCP Socket Big Picture

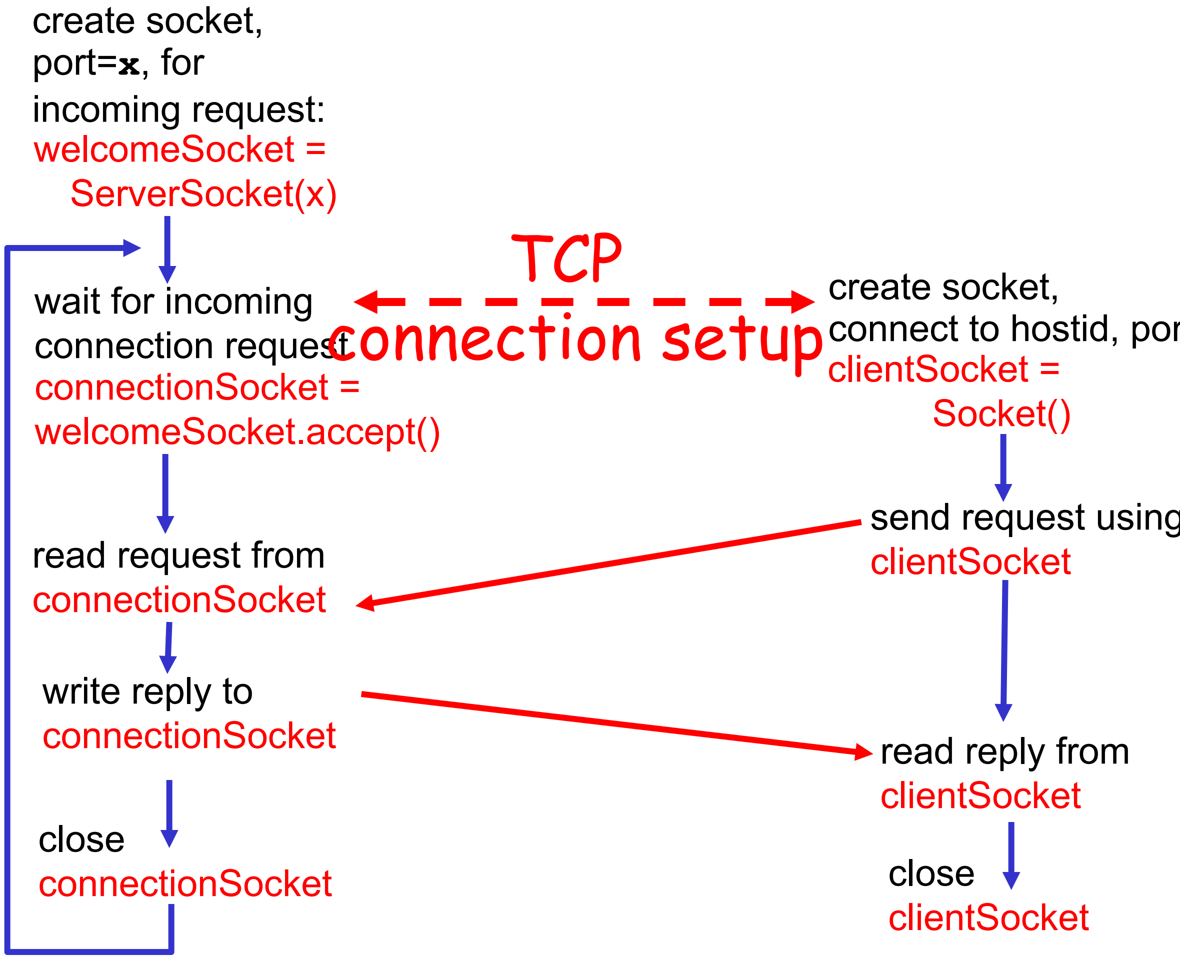


Client/server Socket Workflow: TCP

Server (running on **hostid**)

Client





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

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

TCP
connection setup

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

send request using
clientSocket

read request from
connectionSocket

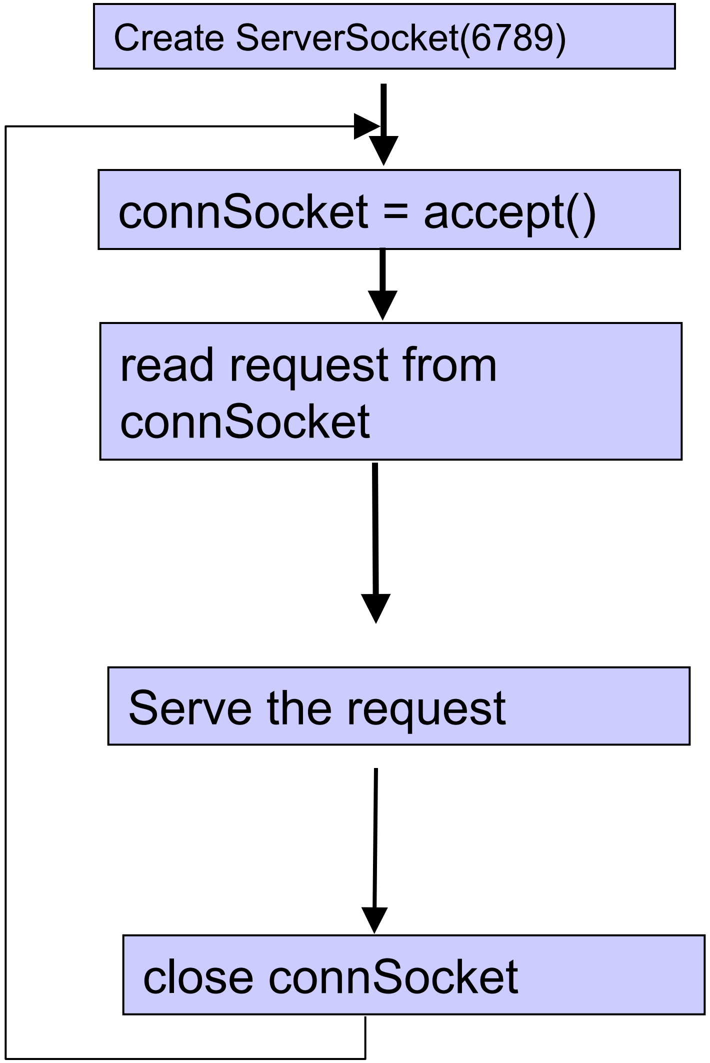
write reply to
connectionSocket

read reply from
clientSocket

close
connectionSocket

close
clientSocket

Server Flow



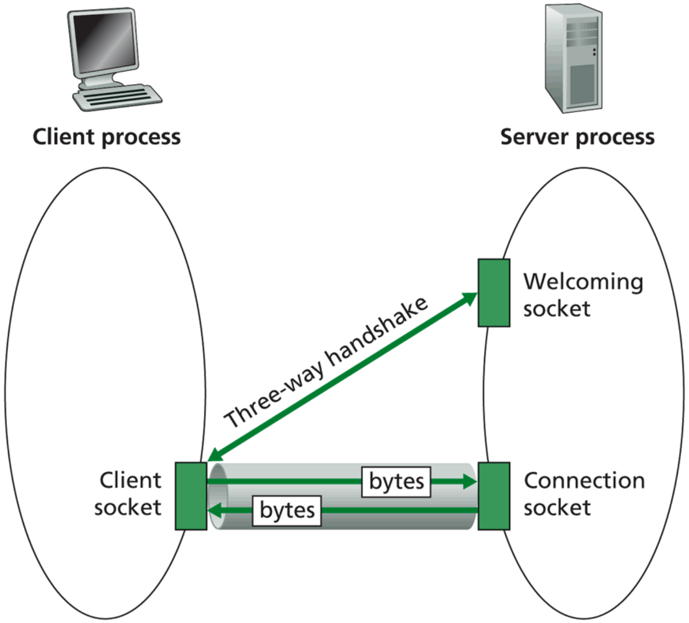
Create ServerSocket(6789)

connSocket = accept()

read request from connSocket

Serve the request

close connSocket



-**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

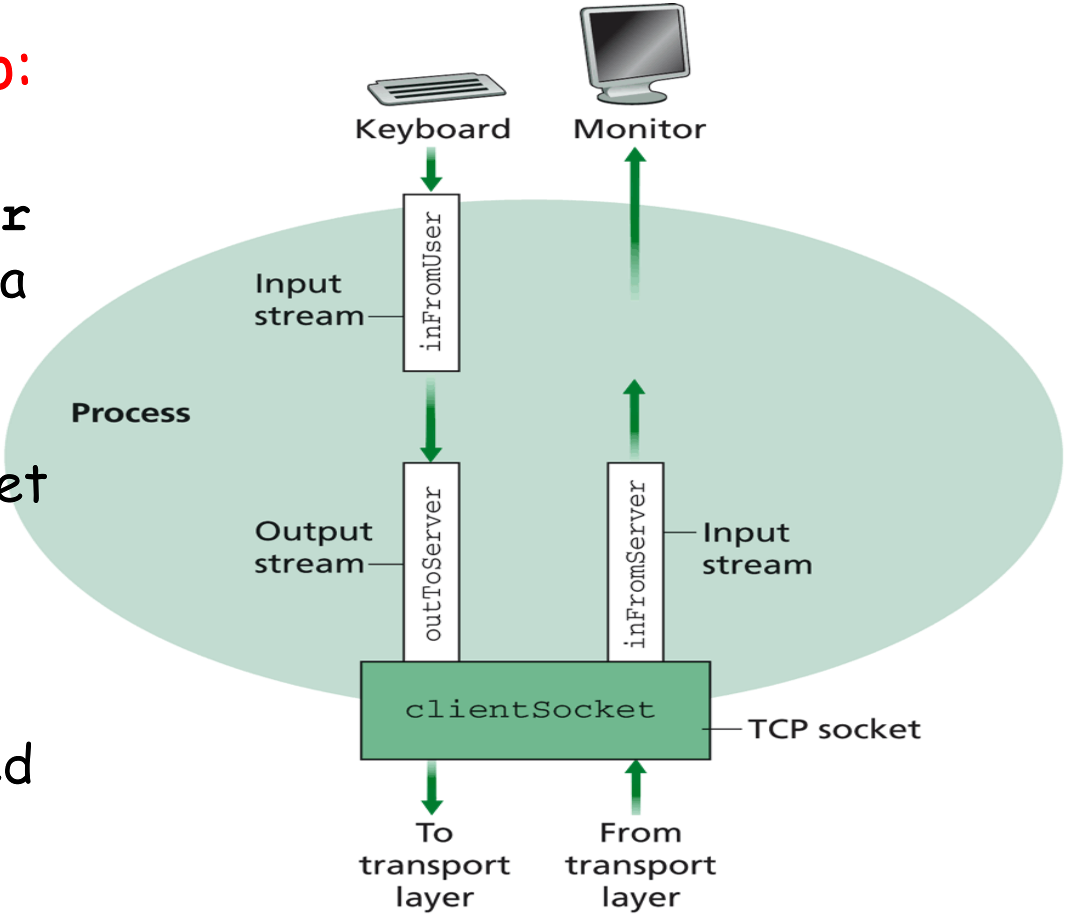
Example client-server app:

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;
```

Create input stream

```
    BufferedReader inFromUser =
      new BufferedReader(new InputStreamReader(System.in));
    sentence = inFromUser.readLine();
```

Create client socket, connect to server

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

Create output stream attached to socket

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

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

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

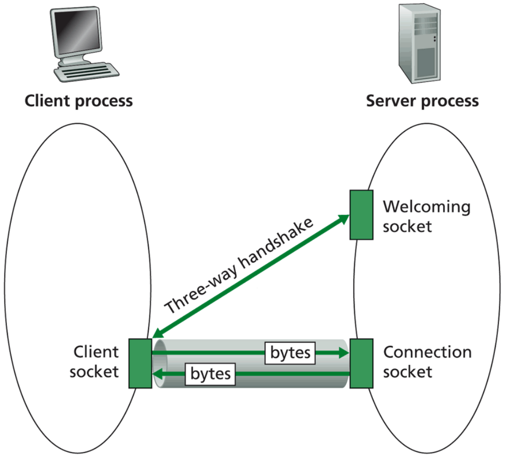
Create input stream attached to socket

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

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);
```

Create welcoming socket at port 6789

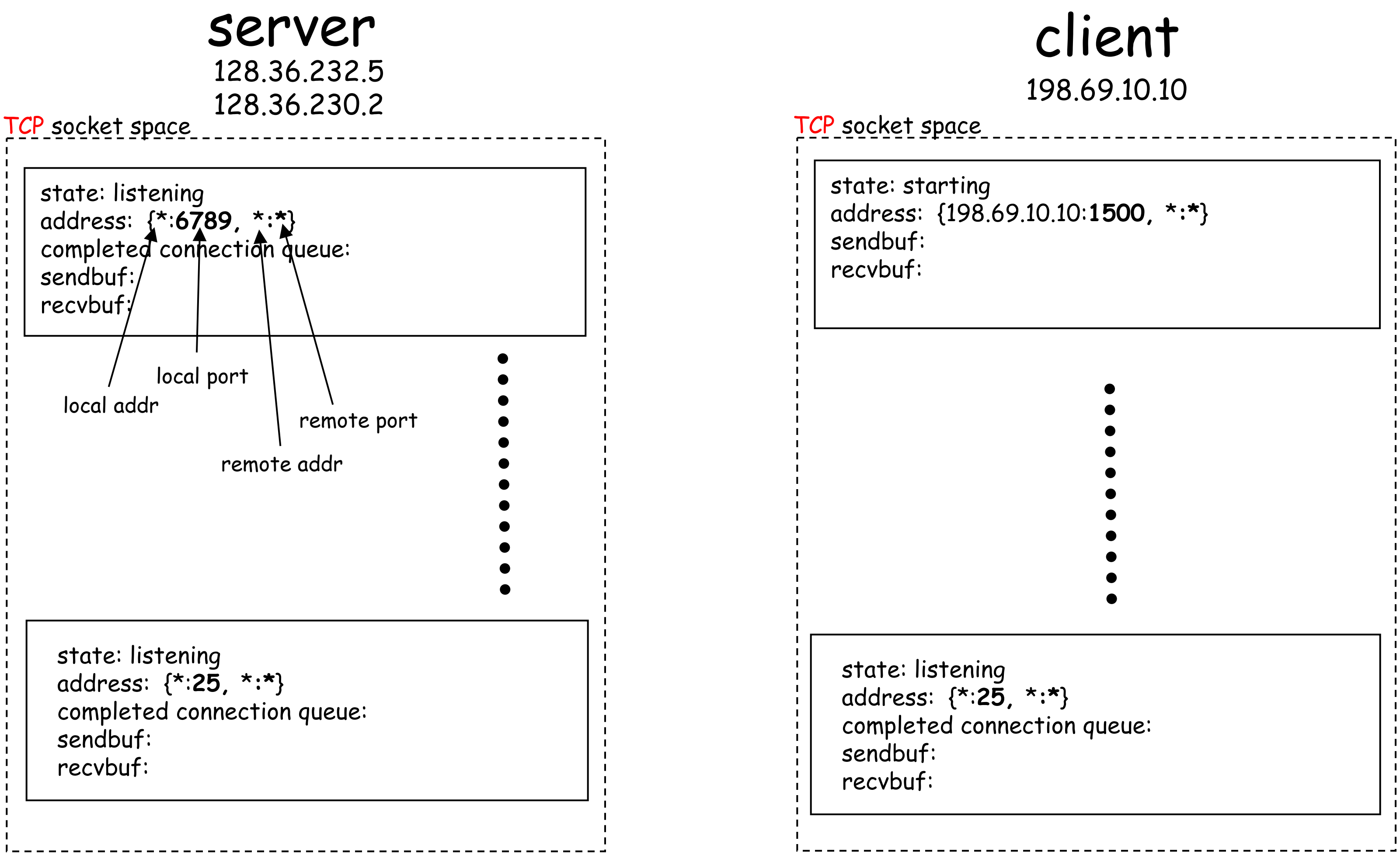
Demo

% on MAC

start TCPServer

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

Under the Hood: After Welcome (Server) Socket



server
128.36.232.5
128.36.230.2

TCP socket space

state: listening
address: {*:**6789**, *:*}
completed connection queue:
sendbuf:
recvbuf:

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

client
198.69.10.10

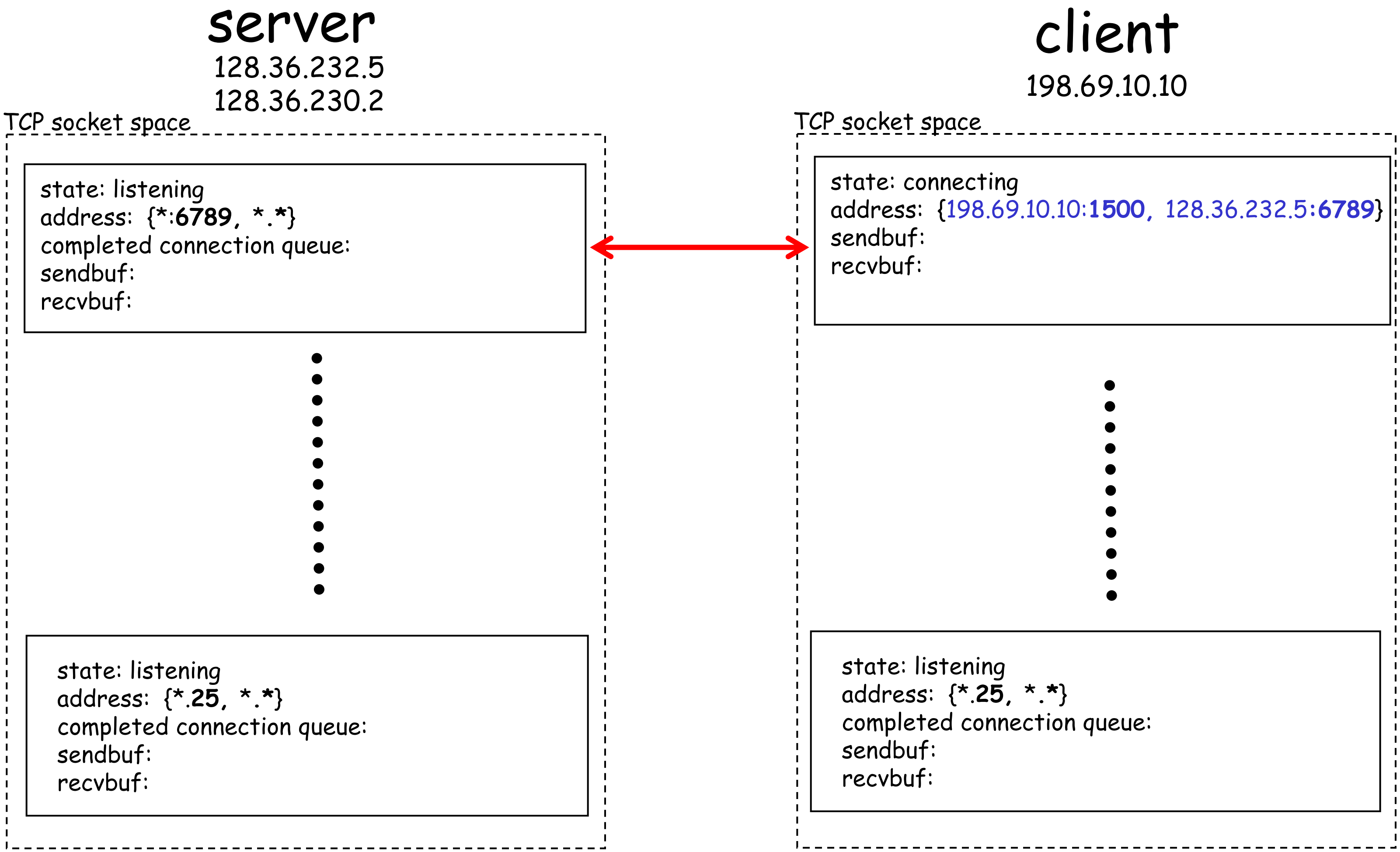
TCP socket space

state: starting
address: {198.69.10.10:**1500**, *:*}
sendbuf:
recvbuf:

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

```
%netstat -p tcp -n -a
```

After Client Initiates Connection



server
128.36.232.5
128.36.230.2

TCP socket space

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

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

client
198.69.10.10

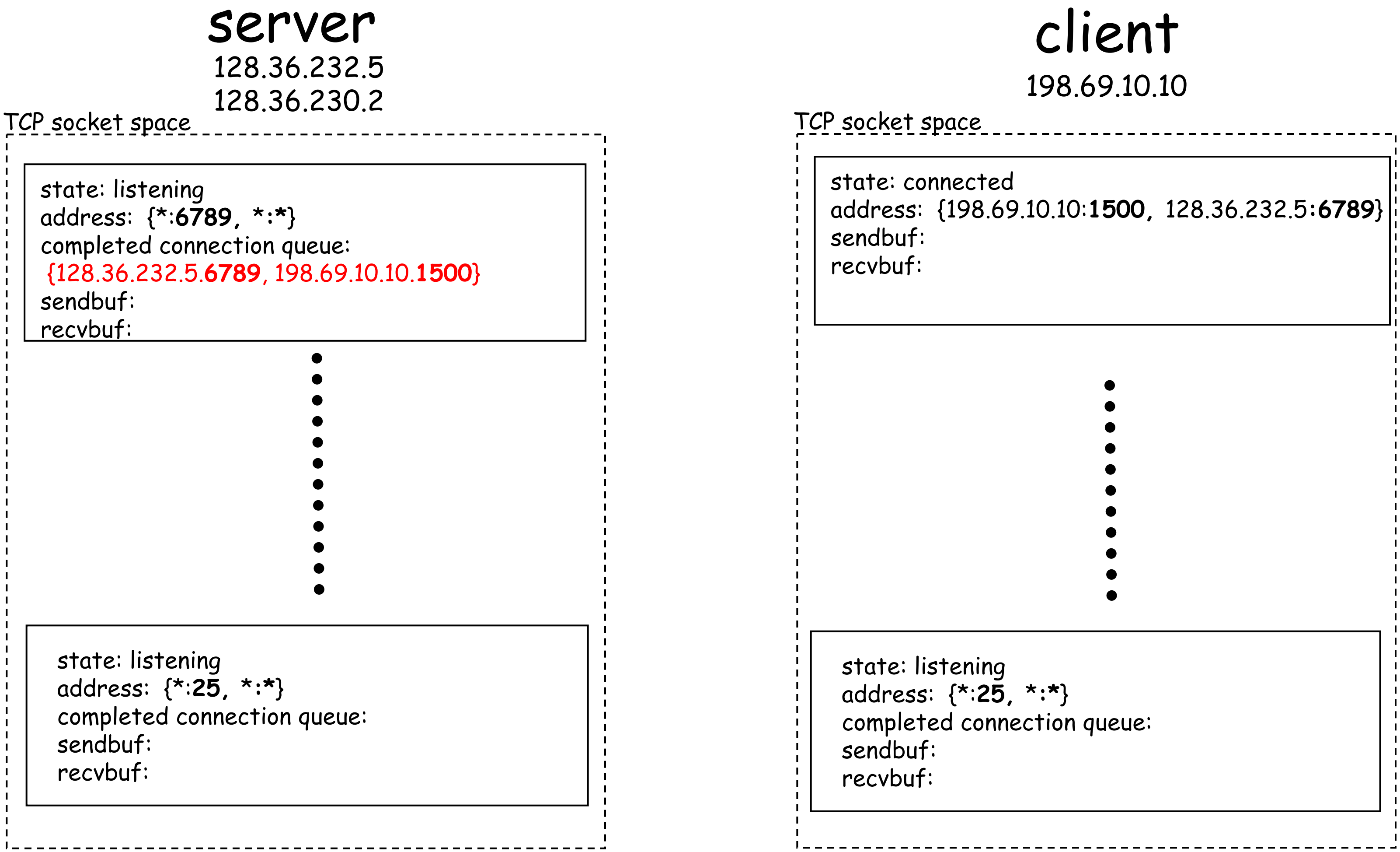
TCP socket space

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

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

```
%cicada 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:

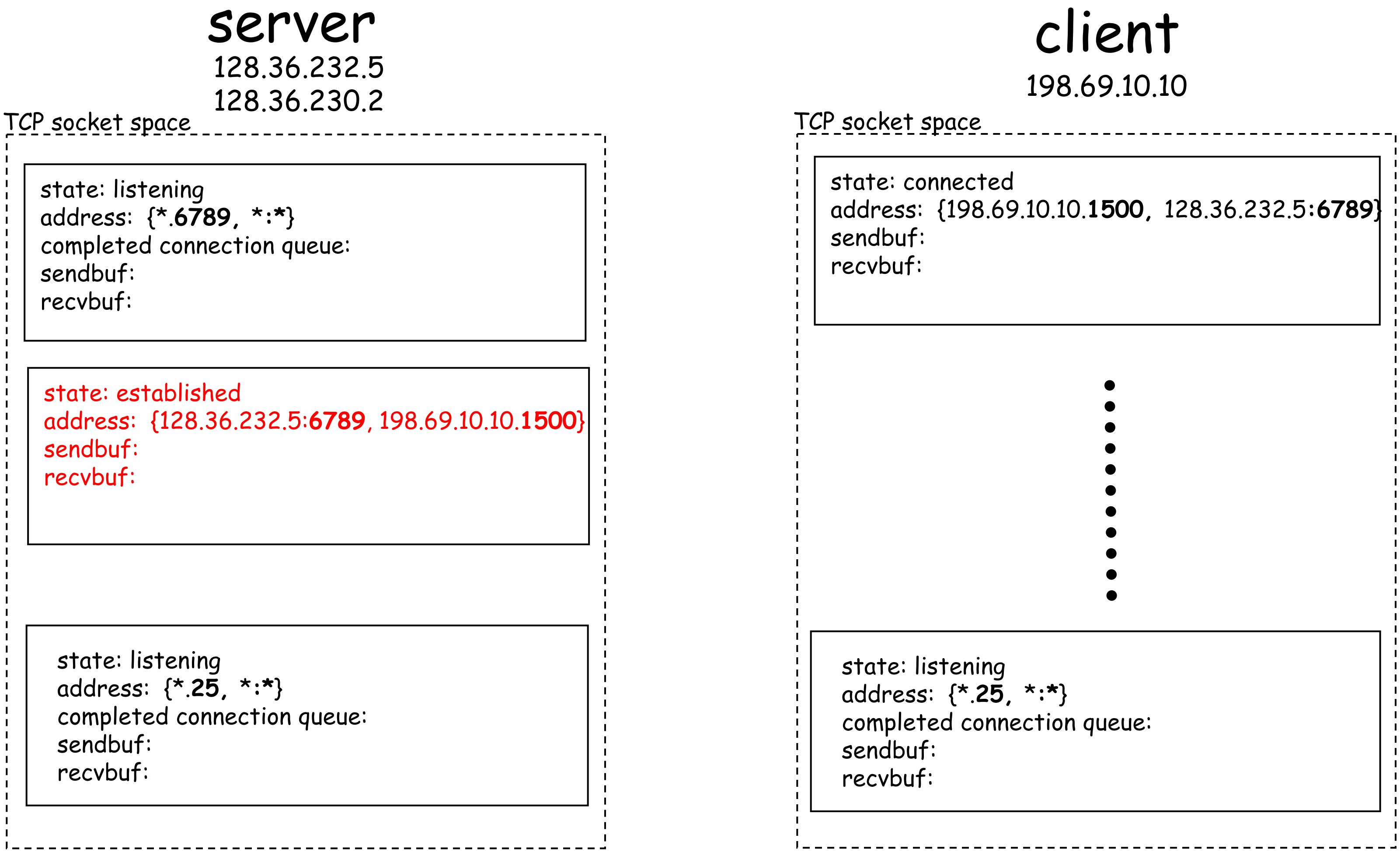
client
198.69.10.10

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



server
128.36.232.5
128.36.230.2

TCP socket space

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

state: established
address: {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:

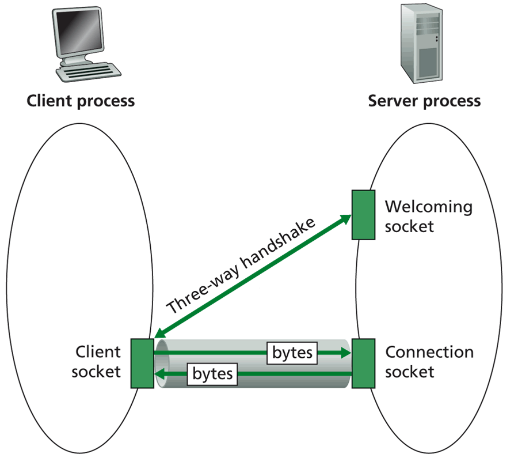
Packet demutiplexing is based on (dst addr, dst port, src addr, src port)

Packet sent to the socket with the best match!

Demo

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

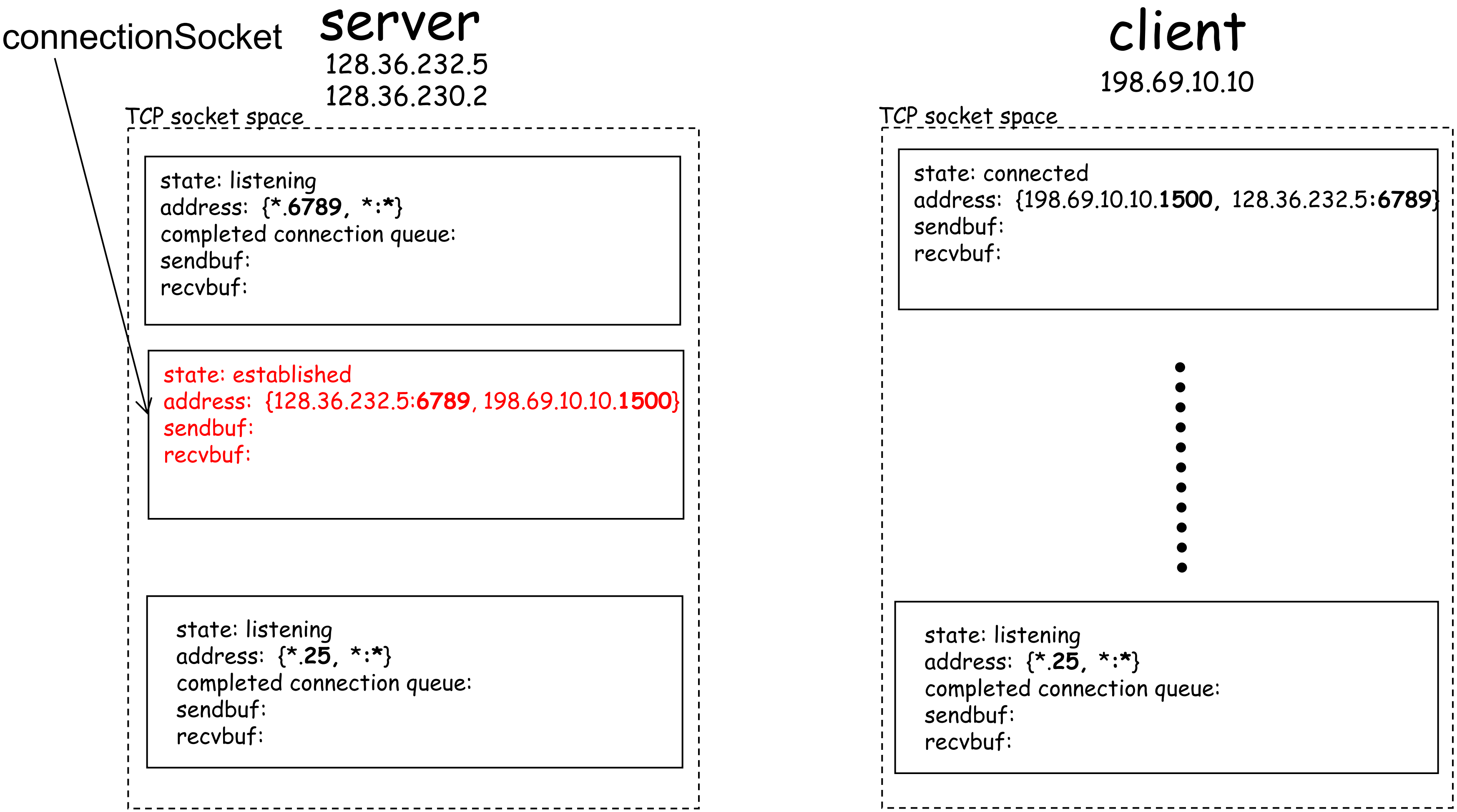
Example: Java server (TCP)



```
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()



connectionSocket

server

128.36.232.5
128.36.230.2

TCP socket space

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

state: established
address: {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: 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

Write out line to socket

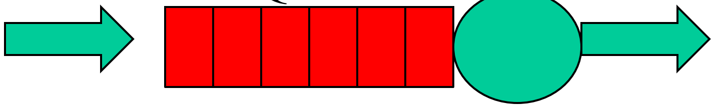
```
          DataOutputStream  outToClient =
           new DataOutputStream(connectionSocket.getOutputStream());

          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)

Welcome
Socket
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)

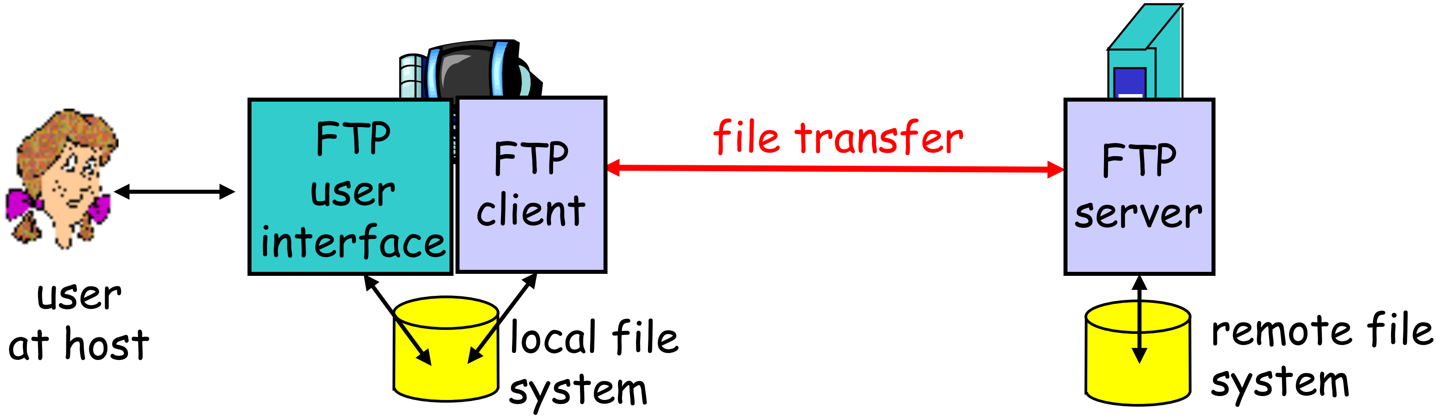
Summary: Basic Socket Programming

- They are relatively straightforward
  - UDP: DatagramSocket, MulticastSocket
  - 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
- Basic network applications
  - Email
  - DNS
- 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:

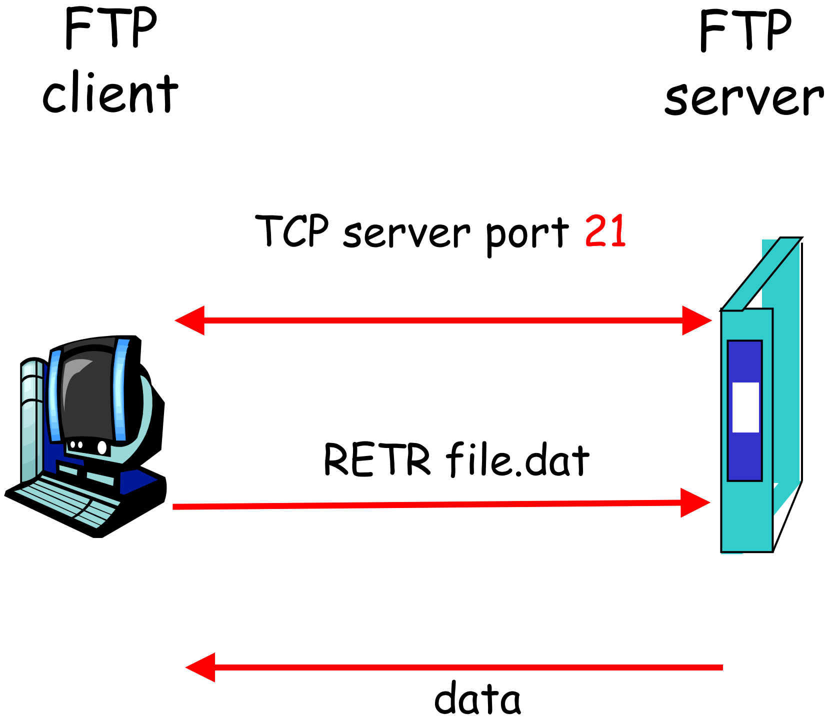
- sent as ASCII text over control channel
- **USER** ***username***
- **PASS** ***password***
- **PWD** returns current dir
- **STAT** shows server status
- **LIST** returns list of file in current directory
- **RETR filename** retrieves (gets) file
- **STOR filename** stores file

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?



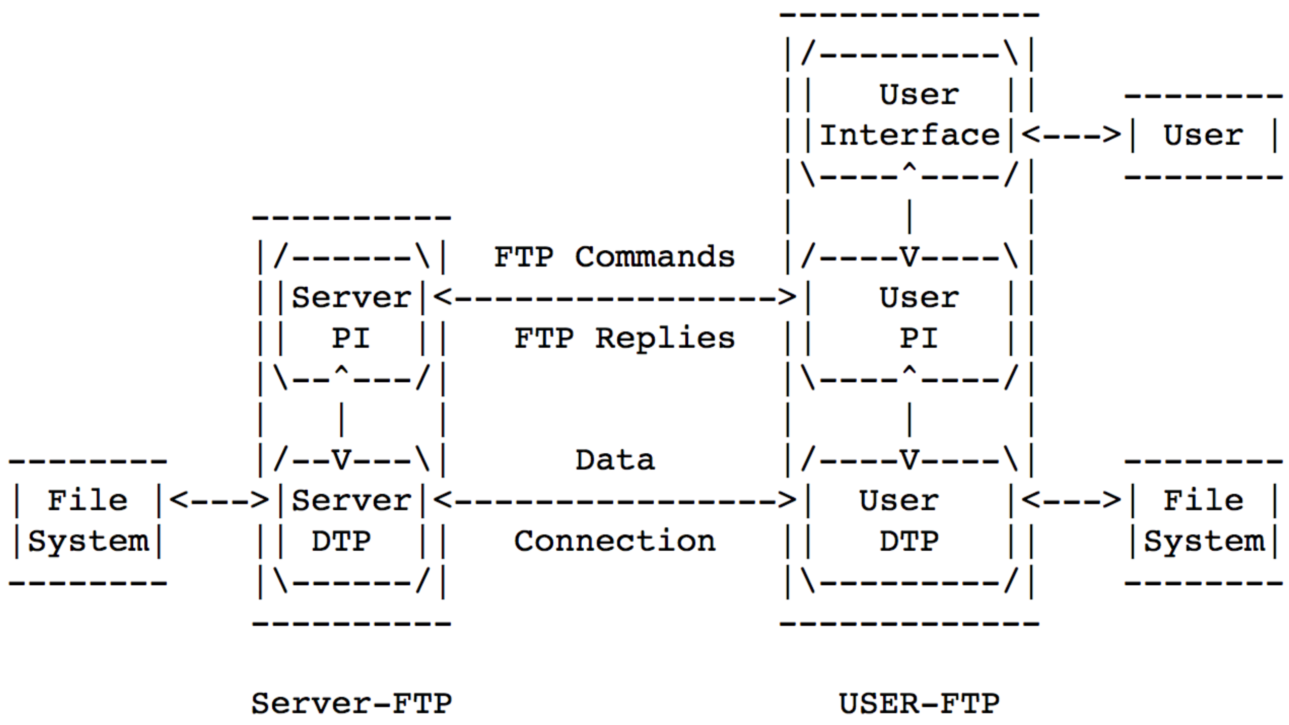
FTP: A Client-Server Application with Separate Control, Data Connections

- 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?

FTP Control/Data Connection Structure



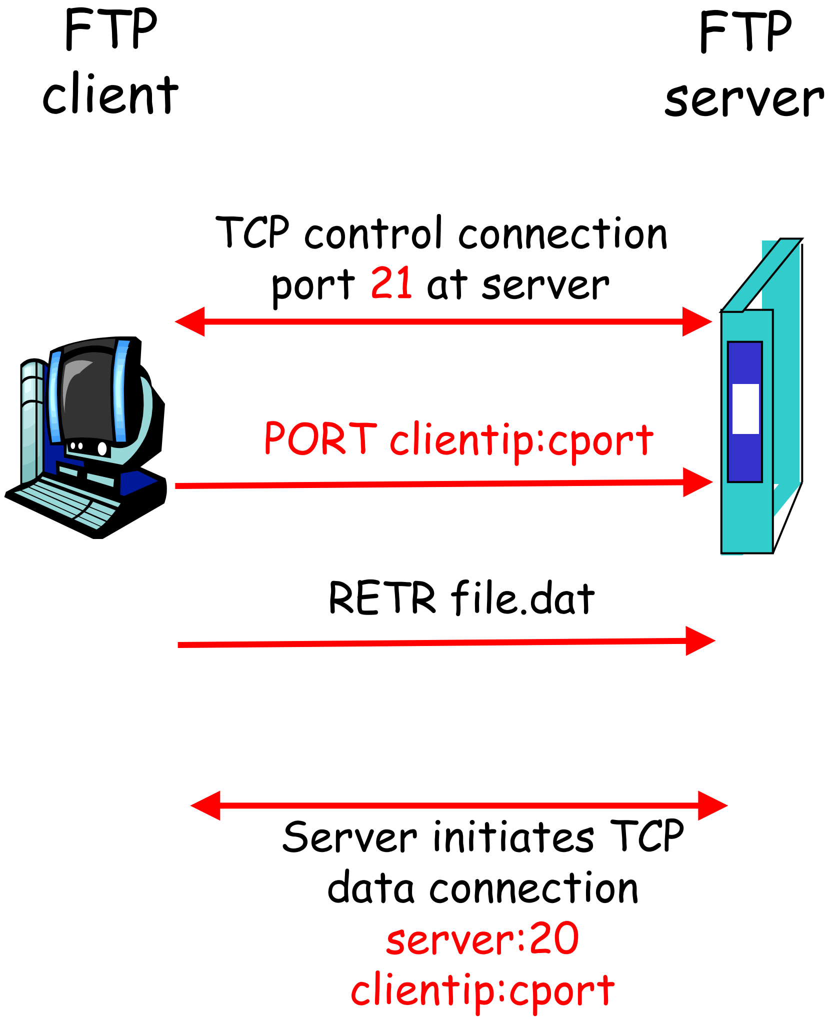
```
                                            -------------
                                            |/---------\|
                                            ||   User  ||    --------
                                            ||Interface|<--->| User |
                                            |\----^----/|    --------
                  ----------                |     |     |
                  |/------\|  FTP Commands  |/----V----\|
                  ||Server|<---------------->|   User  ||
                  ||  PI  ||   FTP Replies  ||    PI   ||
                  |\--^---/|                |\----^----/|
                  |   |    |                |     |     |
      --------    |/--V---\|      Data      |/----V----\|    --------
      | File |<--->|Server|<---------------->|  User   |<--->| File |
      |System|    || DTP  ||   Connection   ||   DTP   ||    |System|
      --------    |\------/|                |\---------/|    --------
                  ----------                -------------

                  Server-FTP                   USER-FTP
```

```
      NOTES: 1. The data connection may be used in either direction.
             2. The data connection need not exist all of the time.
```

Figure 1 Model for FTP Use

Q: How to create a new data connection?

Traditional FTP: Client Specifies Port for Data Connection



FTP client

FTP server

TCP control connection port 21 at server

PORT clientip:cport

RETR file.dat

Server initiates TCP data connection
server:20
clientip:cport

FTP Control/Data Connection Flexibility

```
        Control     ------------   Control
        ---------->| User-FTP  |<-----------
        |          | User-PI   |           |
        |          |   "C"     |           |
        V          ------------            V
--------------                        --------------
| Server-FTP |   Data Connection      | Server-FTP |
|    "A"     |<---------------------->|    "B"     |
-------------- Port (A)      Port (B) --------------
```

Figure 2

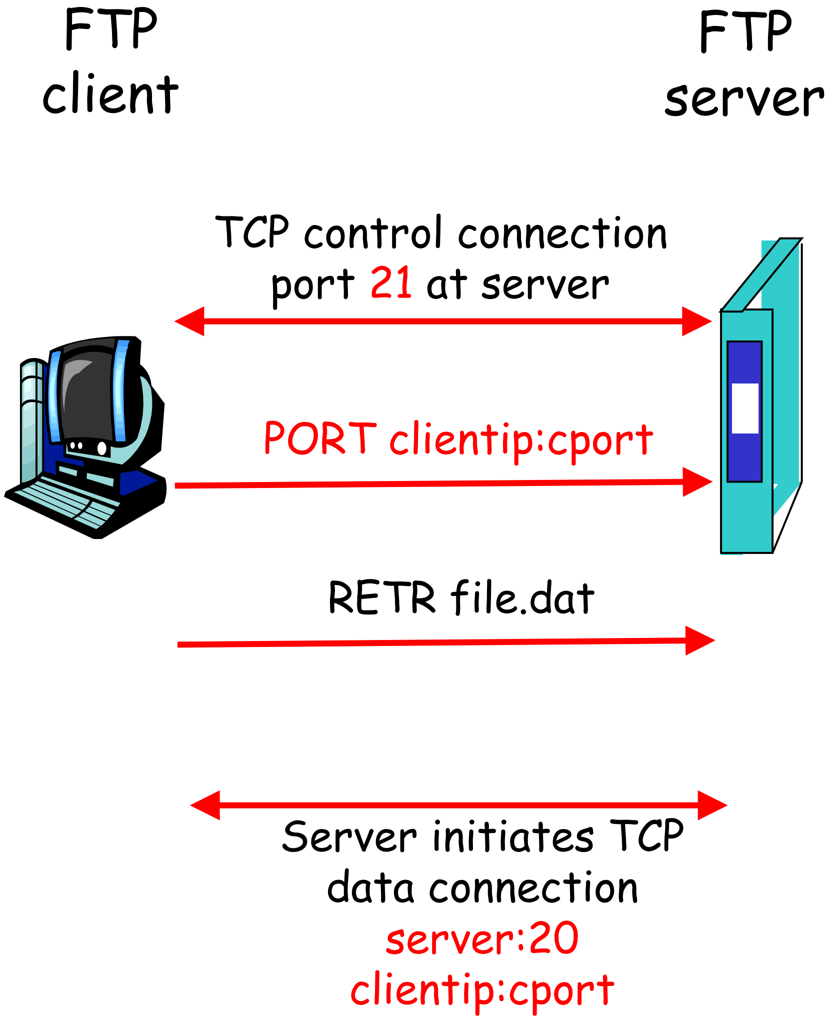
Example using telnet/nc

- Use telnet for the control channel
  - telnet ftp.gnu.org 21
  - user, pass
  - port 172,27,10,223,4,1
  - list

- use nc (NetCat) to receive/send data with server
  - nc -v -l 1025
  - Use our own TCPServer

Problem of the Client PORT Approach

- Many Internet hosts are behind NAT/firewalls that block connections initiated from outside – requirement: client initiate data connection



FTP client

FTP server

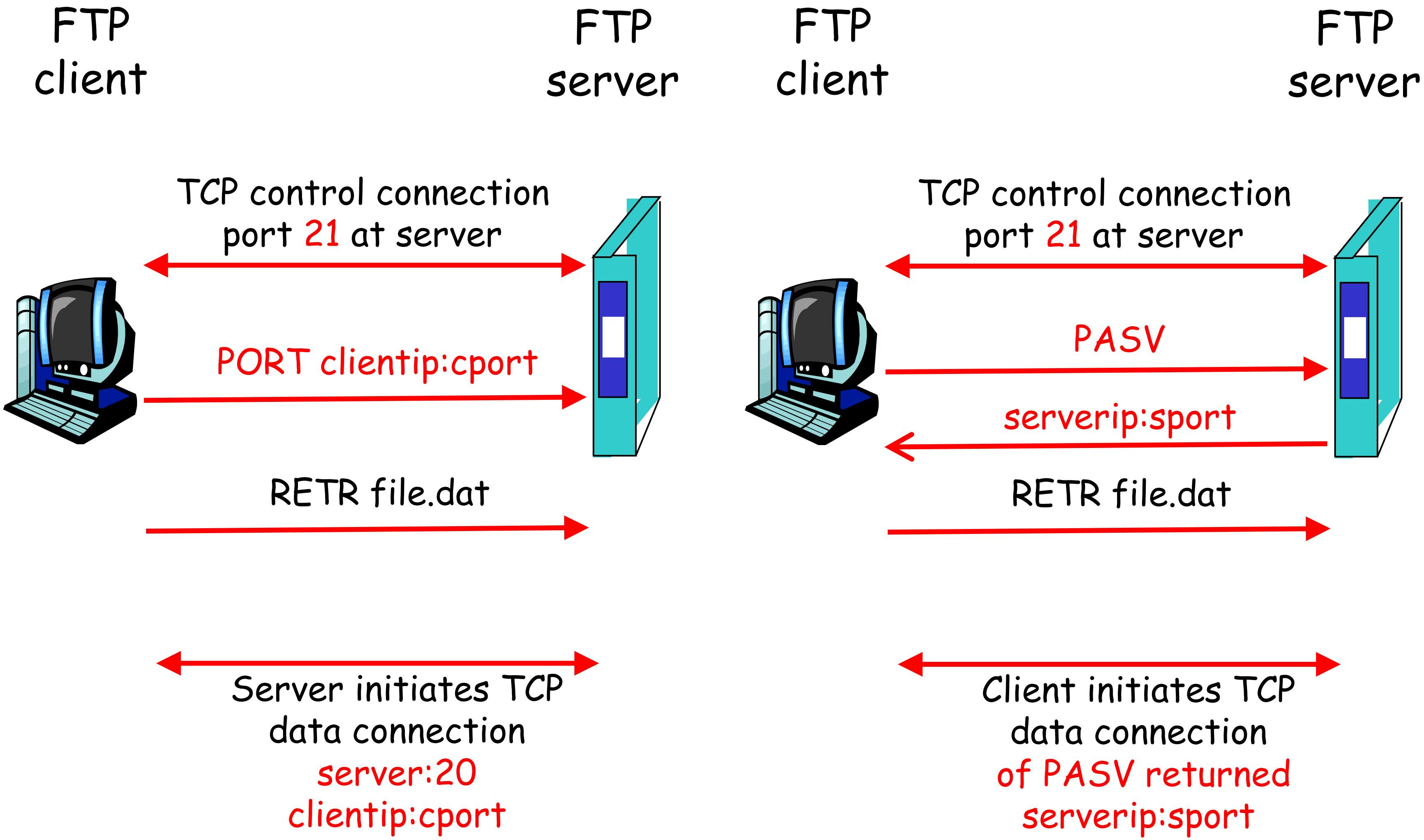
TCP control connection port 21 at server

PORT clientip:cport

RETR file.dat

Server initiates TCP data connection server:20 clientip:cport

FTP PASV: Server Specifies Data Port, Client Initiates Connection



FTP client

FTP server

TCP control connection
port 21 at server

PORT clientip:cport

RETR file.dat

Server initiates TCP
data connection
server:20
clientip:cport

FTP client

FTP server

TCP control connection
port 21 at server

PASV

serverip:sport

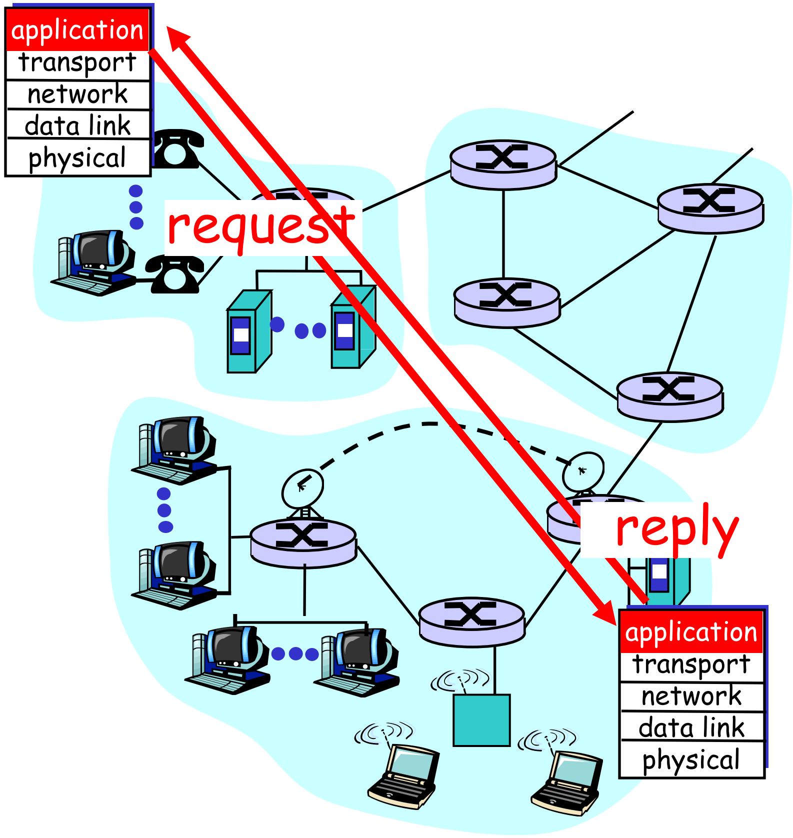
RETR file.dat

Client initiates TCP
data connection
of PASV returned
serverip:sport

Demo

- Use Wireshark to capture FTP traffic
  - wireshark: host ftp.freebsd.org
  - Using chrome/commandline to visit ftp://ftp.freebsd.org
    - First standard, then passive

FTP Evaluation



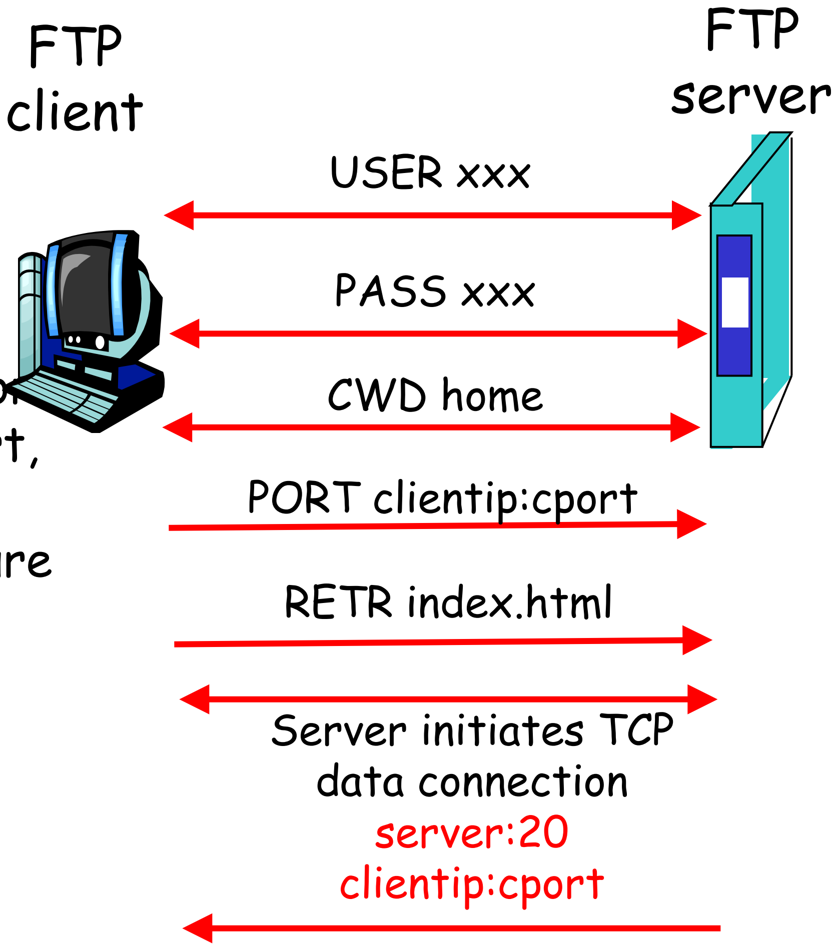
Key questions to ask about a C-S application

- Is the application extensible?
- Is the application scalable?
- How does the application handle server failures (being robust)?
- How does the application provide security?

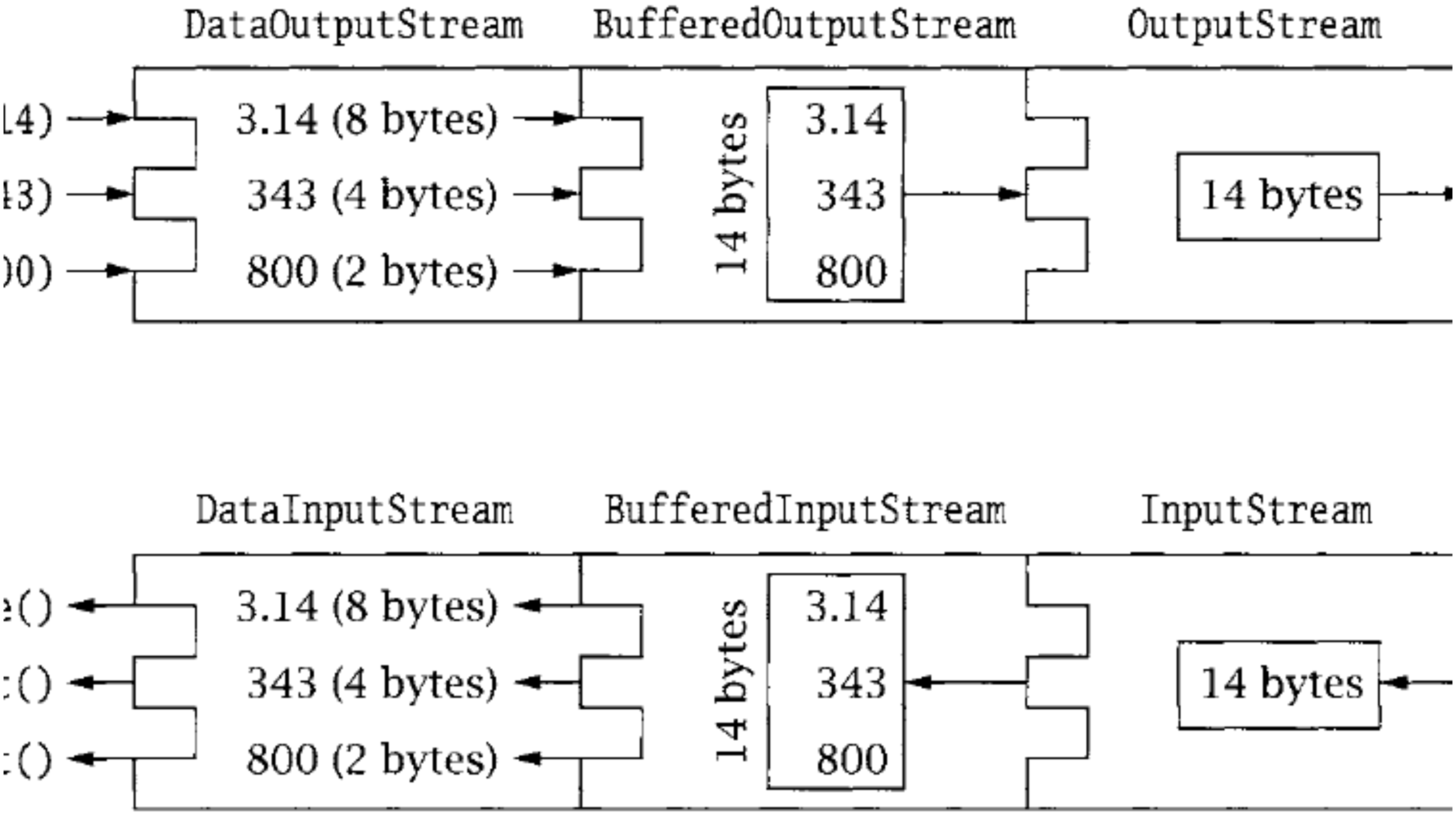
What are some design features of the FTP protocol you consider interesting/can take away?

Summary: FTP Features

- A stateful protocol
  - state established by commands such as
    - USER/PASS, CWD, TYPE
- Multiple TCP connections
  - A control connection
    - commands specify parameters for the data connection: (1) data port, transfer mode, representation type, and file structure; (2) nature of file system operation e.g., store, retrieve, append, delete, etc.
  - Data connections
    - Two approaches: PORT vs PASV



DataStream


DataOutputStream
BufferedOutputStream
OutputStream
l4)
l3)
)0)
3.14 (8 bytes)
343 (4 bytes)
800 (2 bytes)
14 bytes
3.14
343
800
14 bytes
DataInputStream
BufferedInputStream
InputStream
e()
:()
:()
3.14 (8 bytes)
343 (4 bytes)
800 (2 bytes)
14 bytes
3.14
343
800
14 bytes