CMSC417

Computer Networks

Prof. Ashok K. Agrawala

September 6, 2018

Overview

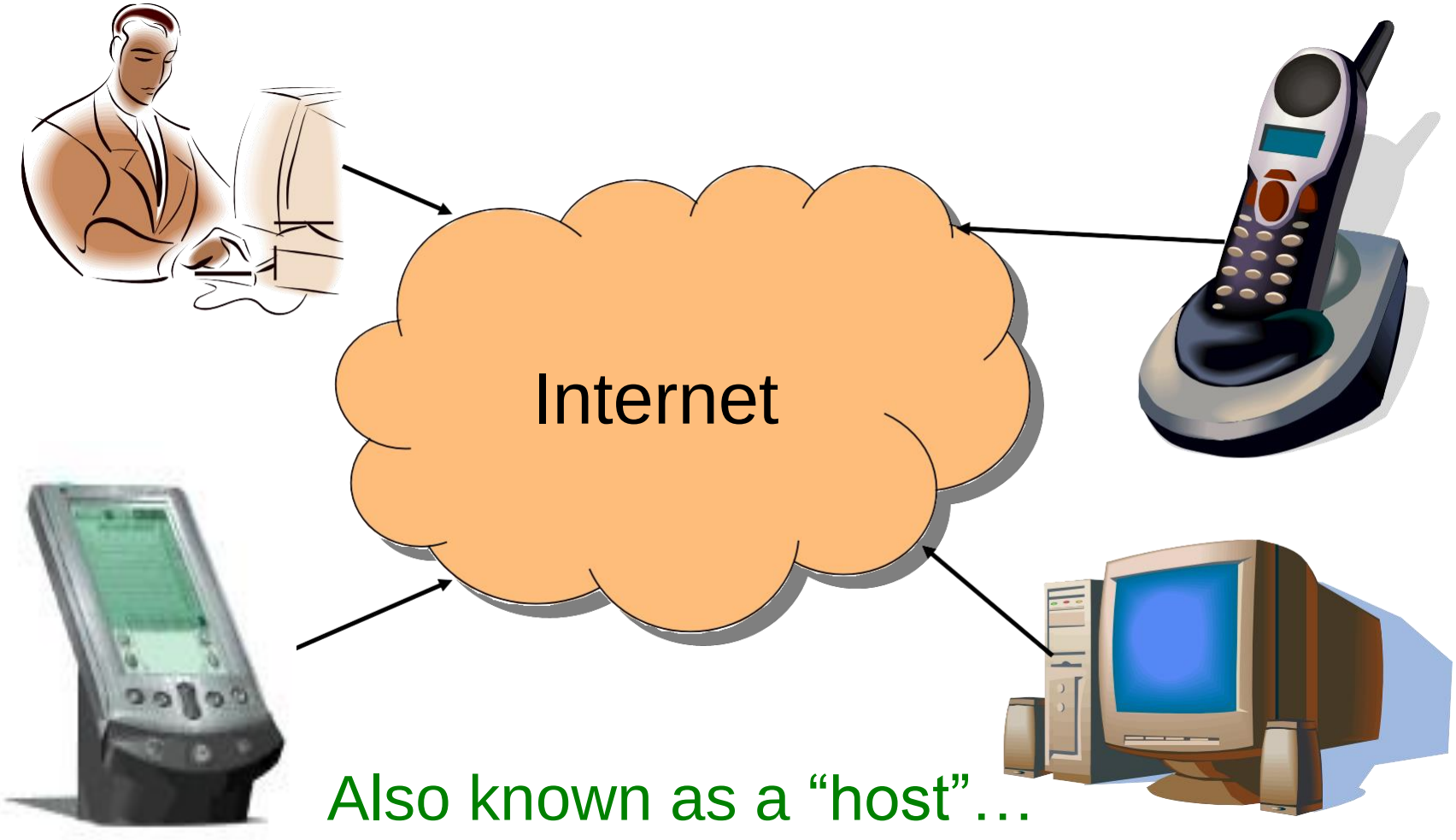
- Client-server paradigm
  - End systems
  - Clients and servers
- Sockets
  - Socket abstraction
  - Socket programming in UNIX
- **File-Transfer Protocol (FTP)**
  - Uploading and downloading files
  - Separate control and data connections

What is a host?

(from whatis.com)

- a computer or other device that communicates with other hosts on a network
- does not include intermediary devices such as switches or routers (these are often categorized as nodes)

End system: computer on the net



Also known as a “host”…

Clients and servers

Client program

- Running on end host
- Requests service
- e. g. Web browser

Server program

- Running on end host
- Provides service
- e. g. Web server

Clients are not necessarily human

Example: Web crawler (or spider)

- Automated client program
- Tries to discover and download many Web pages for indexing
- Forms the basis of search engines like Google
- Indexing allows search engine functionality to be performed more efficiently

Spider client: how it works

- Start with a base list of popular Web sites
- Download the Web pages
- Parse the HTML files to extract hypertext links
- Download these Web pages, too
- And repeat, and repeat, and repeat…

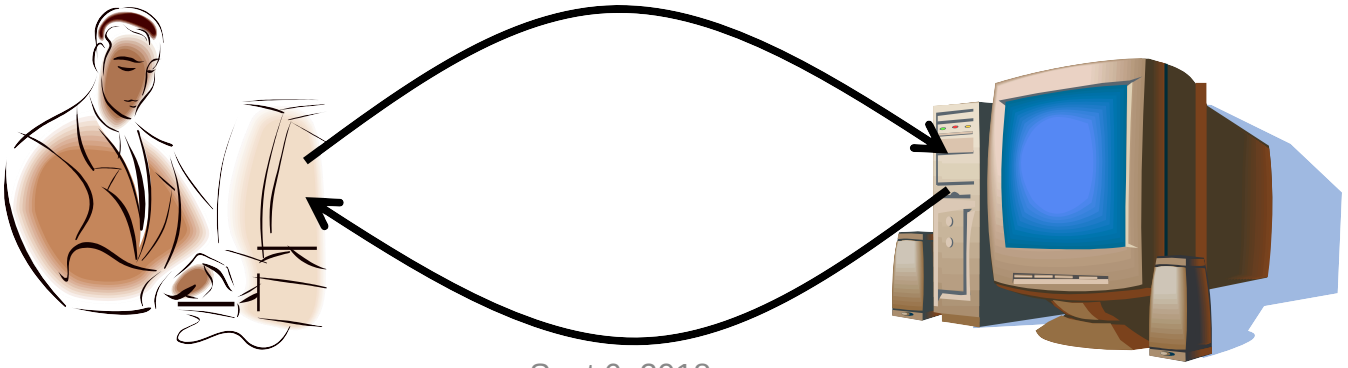
Client-server communication

Client “sometimes on”

- Initiates a request to the server when interested
- e. g. Web browser on your laptop or cell phone
- Doesn’t communicate directly with other clients
- Needs to know the server’s address

Server is “always on”

- Fulfills requests from many client hosts
- e. g. Web server for the www.cnn.com Web site
- Doesn’t initiate contact with the clients
- Needs a fixed, well-known address so that clients can access it



Peer-to-peer communication

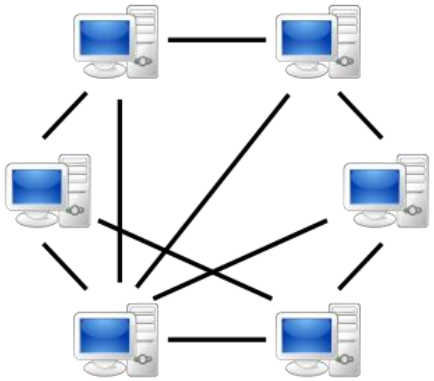
(from Wikipedia)

Instead of one server that is always on:

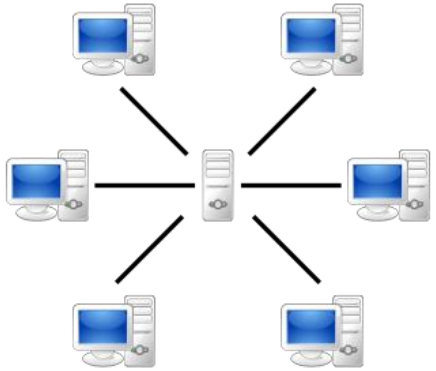
- Hosts can come and go, and change addresses
- Hosts may have a different address each time
- Hosts are known as “peers” and are equally privileged
- Peers make portions of their processing power available to each other

Peer-to-peer communication

(from Wikipedia)



Peer-to-Peer Network



Client-Server Model

Example: peer-to-peer file sharin

- Any host can request files, send files, query to find where a file is located, respond to queries, and forward queries
- Scalability by harnessing millions of peers
- Each peer acting as both a client and server

Client and server processes

What is the difference between a program and a process?

Program: a collection of code

```
s = “Hello world”
print(s)
```

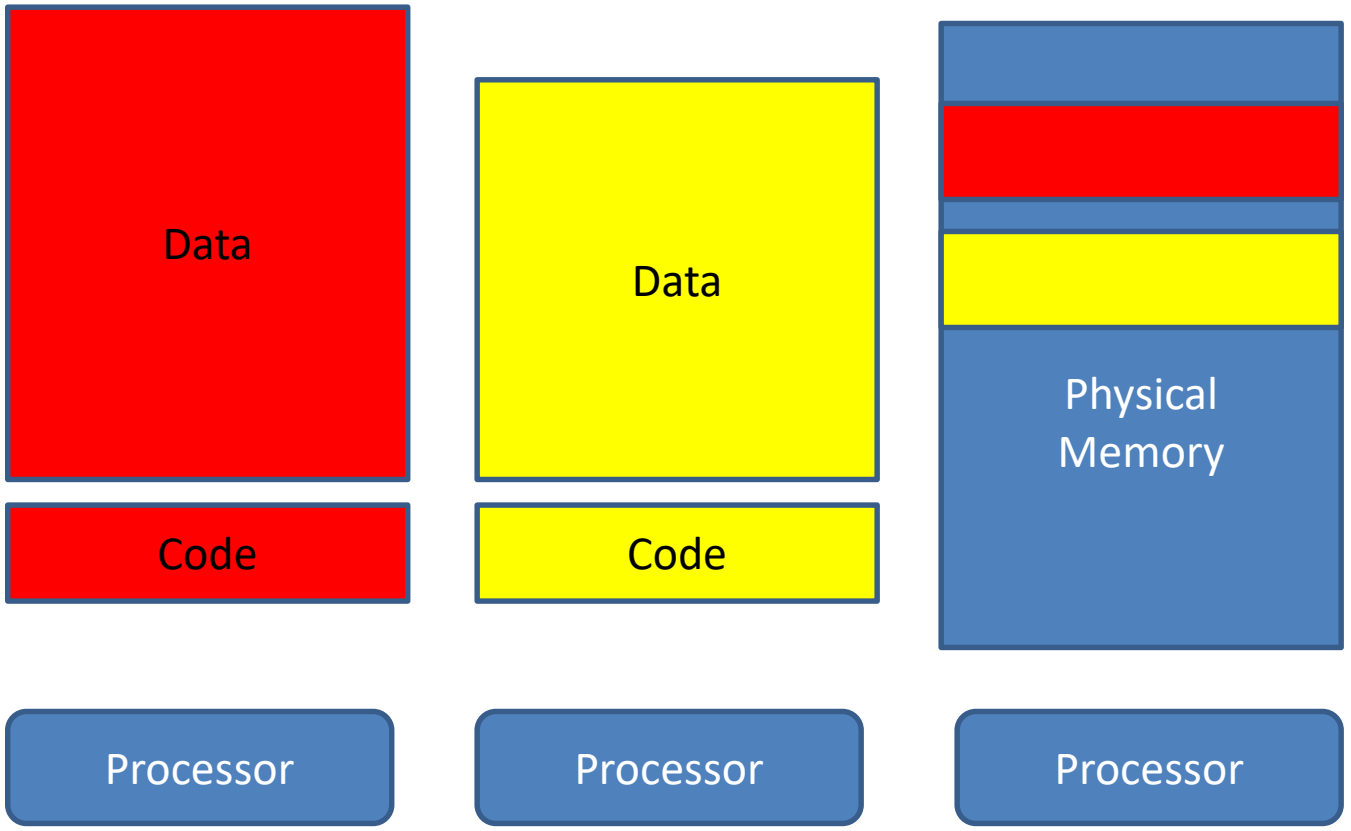
Process: a running program on a host

```
  PID TTY          TIME CMD
13634 ttys000    0:00.02 -bash
13642 ttys000    0:00.06 vi
```

Processes

Each Process has its own

- Code
- Address Space



Communication between processes

- **Same end host:**
  - inter-process communication;
  - governed by the operating system on the end host

- **Different end hosts:**
  - exchanging messages;
  - governed by network protocols

Client and server processes

Client process:

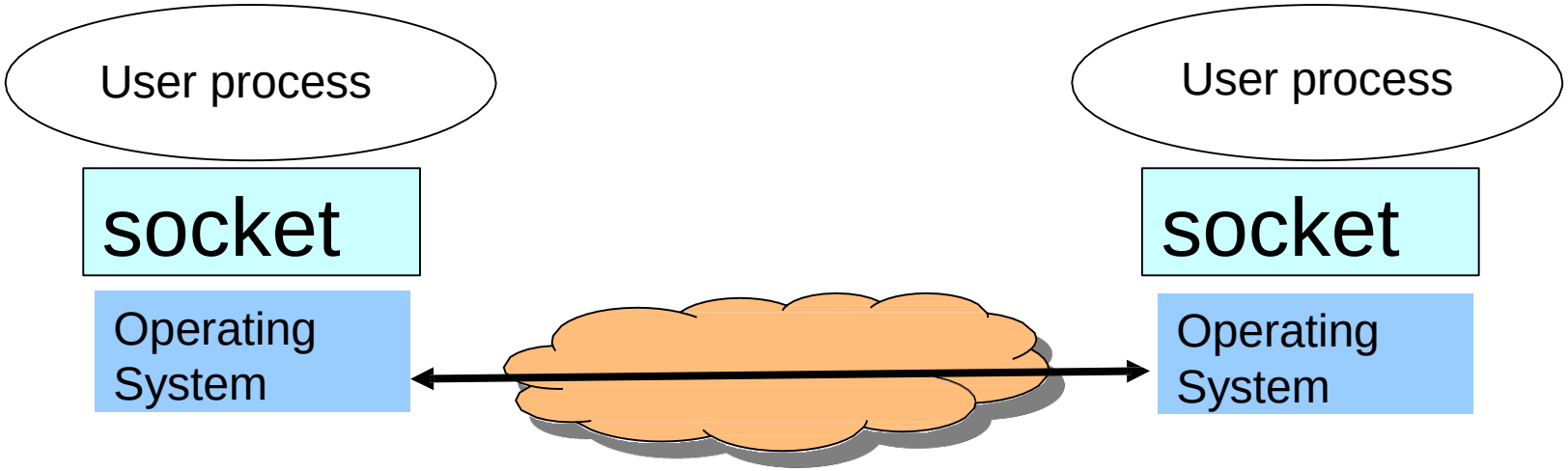
process that initiates communication

Server process:

process that waits to be contacted

Socket: end point of communication

When sending a message from one process to another (e. g. from client to server), the message must traverse the underlying network.

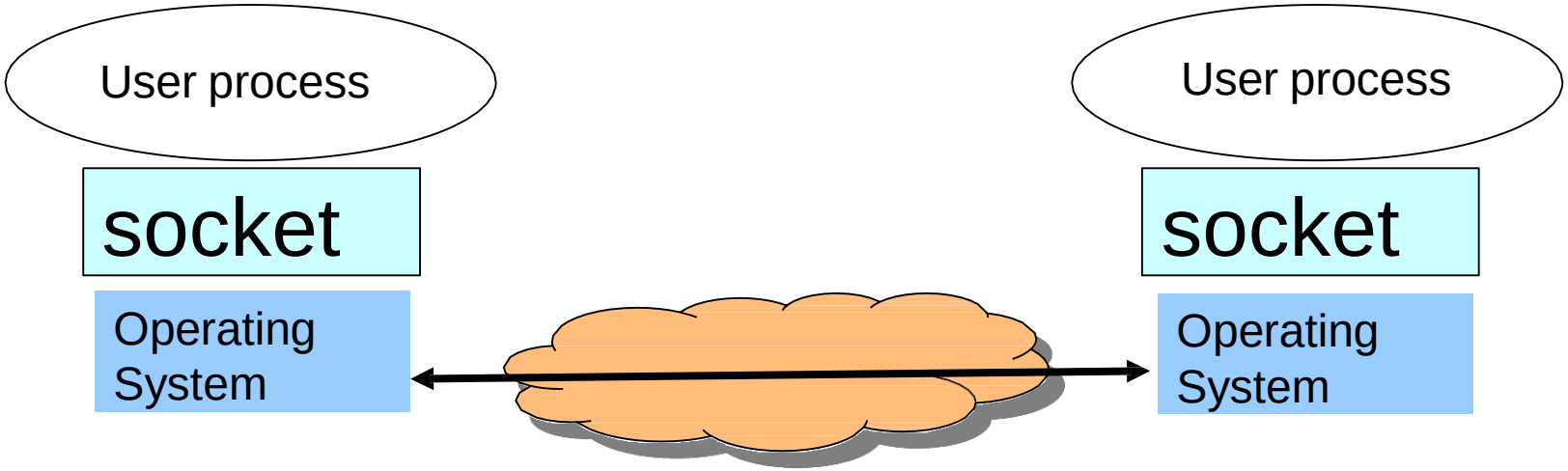


Socket: end point of communication

Processes send and receive through a “socket”. Sockets are like mail slots in a house where messages can go in and out.

Socket: end point of communication

A socket as an Application Programming Interface (API) supports the creation of network applications.



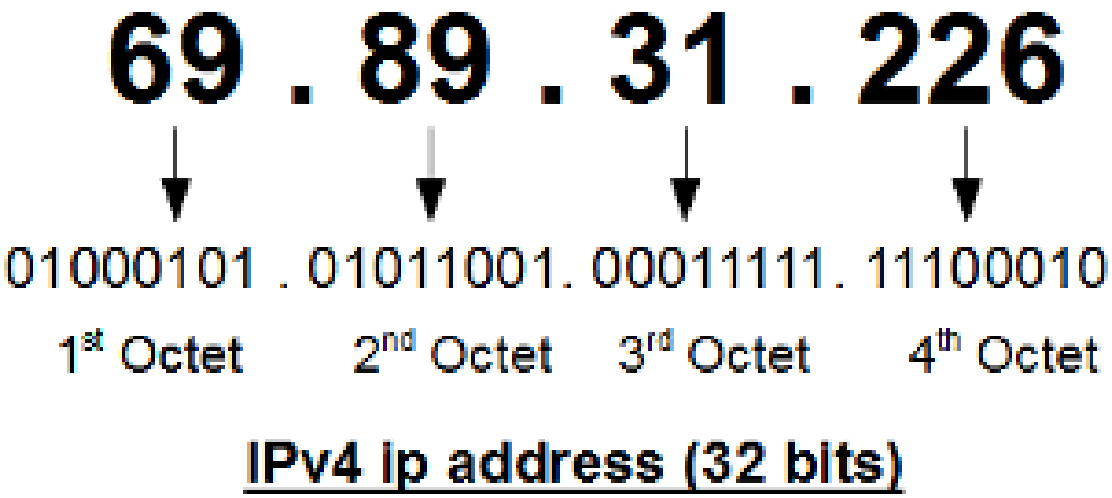
Identifying the receiving process

The sending process must identify the receiver.

- Name or address of the receiving end host
- Identifier that specifies the receiving process
- *Example:* to send mail to someone in an apartment, you need the address of the building and the apartment number

Identifying the receiving host

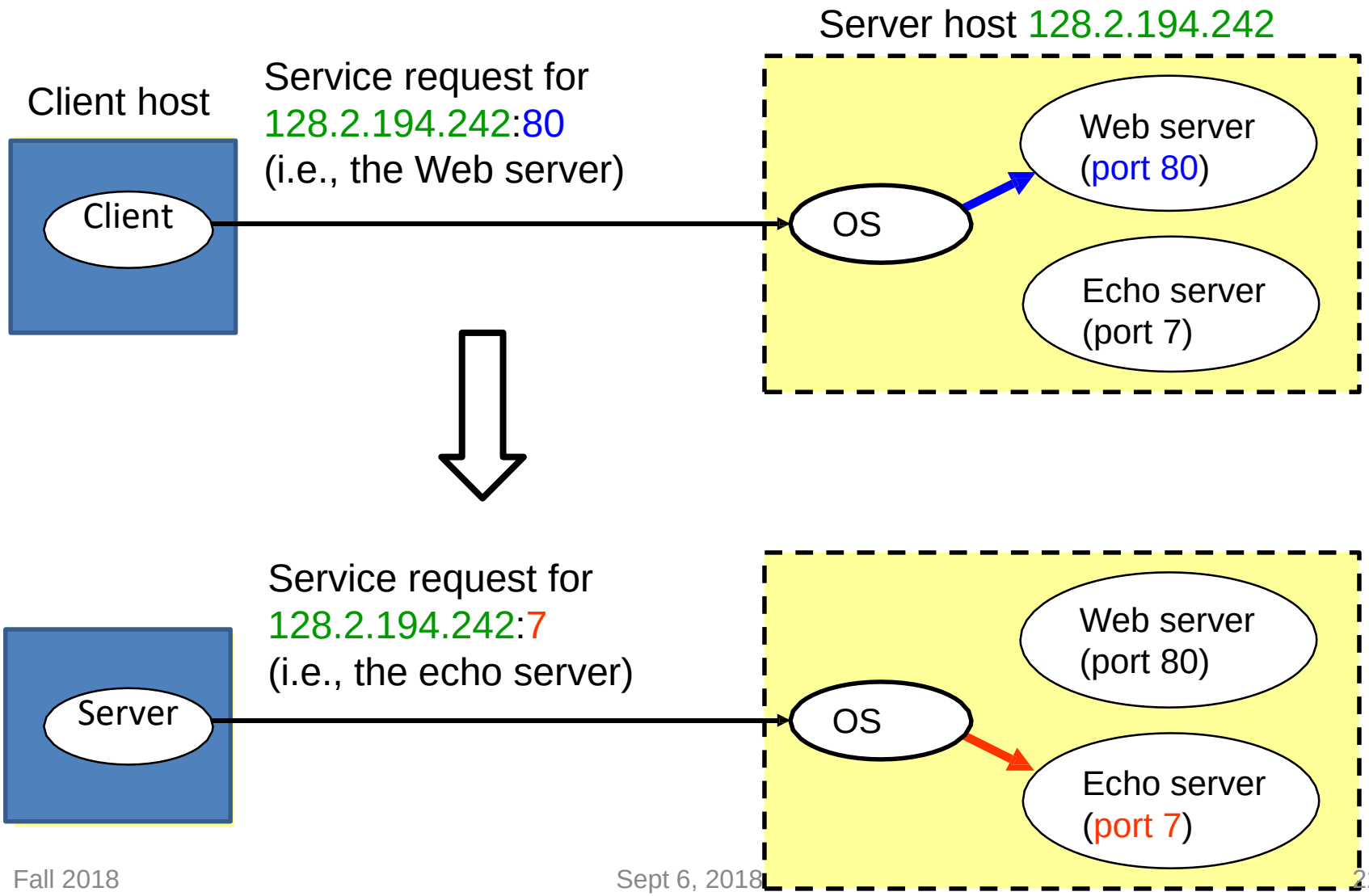
- Destination address that uniquely identifies the host
- **Use an IP address:** a 32-bit quantity



Identifying the receiving process

- Host may be running many different processes (just as an apartment complex has many apartments)
- A destination port uniquely identifies the socket
- A **port number** is a 16-bit quantity that specifies the socket (and therefore the process) that you want to access

Using ports to identify services



Which port number should I use, and when?

Popular applications have well-known ports.

- Port 80 for Web
- Port 25 for e-mail
- Port 22 for ssh
- To see more well-known ports, visit http://www.iana.org

Well-known vs. ephemeral ports

A server has **a well-known port** (e. g. port 80); any port between 0 and 1023.

A client will choose an unused **ephemeral** (i. e. temporary) **port** between 1024 and 65535 that will last as long as it is interacting with the server.

Uniquely identified traffic

- both the server and the client *each* have an IP address and a port number
- their traffic can now be uniquely identified
- an underlying transport protocol used for their communication (e. g. TCP or UDP)

Delivering the data: division of labor

- Network
  - Deliver data packet to the destination host
  - Based on the destination IP address
- Operating system
  - Deliver data to the destination socket
  - Based on the protocol and destination port number
- Application
  - Read data from the socket
  - Interpret the data (e. g. render a Web page)

UNIX socket API

- Socket interface
  - Originally provided in Berkeley UNIX
  - Later adopted by all popular operating systems
  - Simplifies porting applications to different OSes
- In UNIX, everything is like a file
  - All input is like reading a file
  - All output is like writing a file
  - File is represented by an integer file descriptor
- System calls for sockets
  - Client: create, connect, write, read, close
  - Server: create, bind, listen, accept, read, write, close

Typical client program

1. Prepare to communicate
   a. Create a socket
   b. Determine server address and port number
   c. Initiate the connection to the server
2. Exchange data with the server
   – Write data to the socket
   – Read data from the socket
   – Do stuff with the data (e. g. render a Web page)
3. Close the socket

Creating a socket with socket()

```
int socket(int domain, int type, int protocol)
```

- Returns a descriptor (or handle) for the socket
- Originally designed to support any protocol suite
- *Domain:* protocol family (`PF_INET` for the Internet)
- *Type:* semantics of the communication
  - `SOCK_STREAM:` reliable byte stream
  - `SOCK_DGRAM:` message-oriented service
- *Protocol:* specific protocol
  - `UNSPEC:` unspecified
  - (`PF_INET` and `SOCK_STREAM` already implies TCP)

Connecting the socket to the server

- Translating the server's name to an address
  - `struct hostent *gethostbyname(char *name)`
  - Argument: the name of the host (e.g. "www.cnn.com")
  - Returns a structure that includes the host address
- Identifying the service's port number
  - `struct servent *getservbyname(char *name, char *proto)`
  - Arguments: service (e. g. `ftp`) and protocol (e. g. `tcp`)
- Establishing the connection
  - `int connect(int sockfd, struct sockaddr *server_address, socketlen_t addrlen)`
  - Arguments: socket descriptor, server address, and address size
  - Returns 0 on success, and -1 if an error occurs

Sending data

```
ssize_t write(int sockfd, void *buf, size_t len)
```

- Arguments: socket descriptor, pointer to buffer of data to send, and length of the buffer
- Returns the number of characters written, and -1 on error

Receiving data

```
ssize_t read(int sockfd, void *buf, size_t len)
```

- Arguments: socket descriptor, pointer to buffer to place the data, size of the buffer
- Returns the number of characters read (where 0 implies "end of file"), and -1 on error

Closing the socket

```
int close(int sockfd)
```

Byte ordering: little and big endian

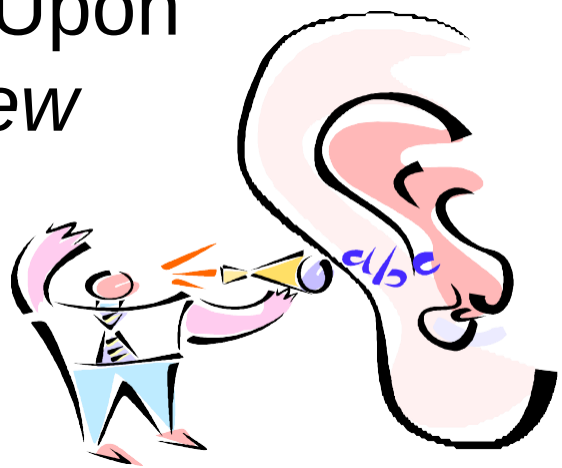
- Hosts differ in how they store data (e. g. four-byte number (byte3, byte2, byte1, byte0))
- **Little endian** (“little end comes first”) Intel PCs!!!
  - Low-order byte stored at the lowest memory location
  - Byte0, byte1, byte2, byte3
- **Big endian** (“big end comes first”)
  - High-order byte stored at lowest memory location
  - Byte3, byte2, byte1, byte 019IP is big endian (aka “network byte order”)
  - Use `htons()` and `htonl()` to convert to network byte order
  - Use `ntohs()` and `ntohl()` to convert to host order

Why can’t sockets hide these details?

- Dealing with endian differences is tedious.
- Couldn’t the socket implementation deal with this by swapping the bytes as needed? No, **swapping depends on the data type**.
  - Two-byte short int: (byte 1, byte 0) vs. (byte 0, byte 1)
  - Four-byte long int: (byte 3, byte 2, byte 1, byte 0) vs. (byte 0, byte 1, byte 2, byte 3)
  - String of one-byte characters: (char 0, char 1, char 2, …) in both cases
- Socket layer doesn’t know the data types
  - Sees the data as simply a buffer pointer and a length
  - Doesn’t have enough information to do the swapping

Servers differ from clients

- **Passive open.** Prepare to accept connection, but don't actually establish one until hearing from a client.
- **Hearing from multiple clients.** Allow a backlog of waiting clients in case several try to start a connection at once
- **Create a socket for each client.** Upon accepting a new client, create a *new* socket for the communication



Typical server program

- Prepare to communicate
  - Create a socket
  - Associate local address and port with the socket
- Wait to hear from a client (passive open)
  - Indicate how many clients-in-waiting to permit
  - Accept an incoming connection from a client
- Exchange data with the client over new socket
  - Receive data from the socket
  - Do stuff to handle the request (e. g. get a file)
  - Send data to the socket
  - Close the socket
- Repeat with the next connection request

Server preparing its socket

- Bind socket to the local address and port number
  - `int bind (int sockfd, struct sockaddr *my_addr, socklen_t addrlen)`
  - Arguments: socket descriptor, server address, address length
  - Returns 0 on success, and -1 if an error occurs
- Define how many connections can be pending
  - `int listen(int sockfd, int backlog)`
  - Arguments: socket descriptor and acceptable backlog
  - Returns 0 on success, and -1 on error

Accepting a new client connection

```
int accept(int sockfd, struct sockaddr *addr, socketlen_t *addrlen)
```

- Arguments: socket descriptor, structure that will provide client address and port, and length of the structure
- Returns descriptor for a new socket for this connection

Questions

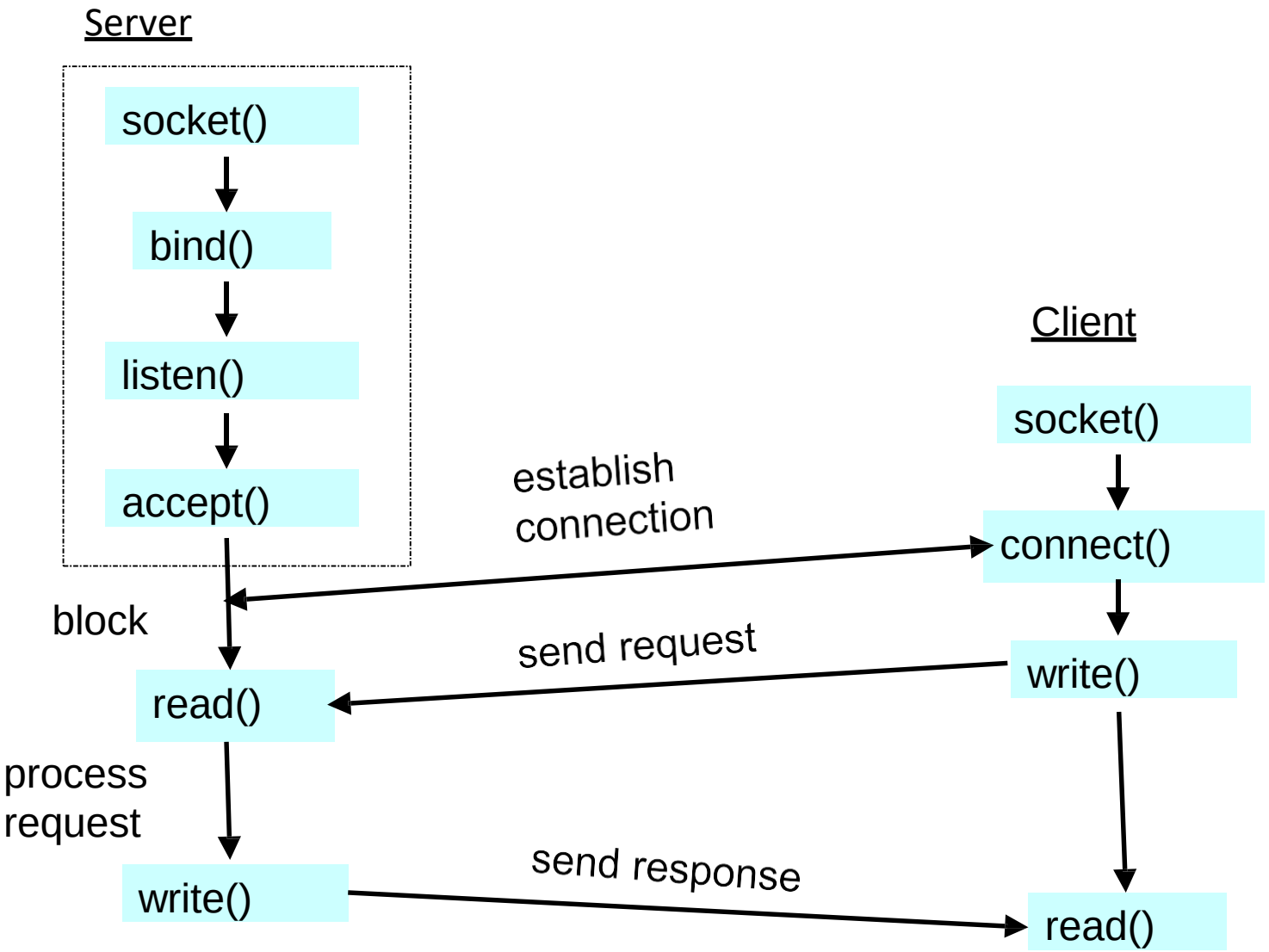
Q. What happens if no clients are around?

A.The `accept()` call blocks waiting for a client

Q. What happens if too many clients are around?

A. Some connection requests don't get through, but that's okay, because the Internet makes no promises

Putting it all together



Serving one request at a time?

Inefficient!

- The server can only process one request at a time
- All other clients must wait until previous one is done

Solution: Time share the machine

Alternate between fulfilling different requests.

1. **Do a little work on one request, then switch to another.** Examples:
   - read HTTP request
   - locate the associated file
   - read the disk
   - transmit
2. **Start a new process to handle each request.** (OS shares CPU across processes)

OR try some hybrid of these two approaches.

Wanna see real clients and servers?

- Apache Web server
  - Open source server first released in 1995
  - Name derives from “a patchy server” ;-)
  - Software available online at http://www.apache.org
- Mozilla Web browser
  - http://www.mozilla.org/developer/
- Sendmail
  - http://www.sendmail.org/
- BIND Domain Name System
  - Client resolver and DNS server
  - http://www.isc.org/index.pl?/sw/bind/

Advice for assignments

- Familiarize yourself with the socket API
  - Read the online references
  - Read the manual pages (e. g. `man socket`)
  - Feeling self-referential? Do `man man`!
- Write a simple socket program first
  - e. g. simple echo program
  - e. g. simple FTP client that connects to server

File Transfer Protocol (FTP)

- Allows a user to copy files to/from remote hosts. The client program…
  - connects to FTP server
  - provides a login id and password
  - allows the user to explore the directories
  - and download and upload files with the server
- A predecessor of the Web (RFC 959 in 1985). It requires users to…
  - know the name of the server machine
  - have an account on the machine
  - find the directory where the files are stored
  - know whether the file is text or binary
  - know what tool to run to render and edit the file
- No URL, hypertext, or helper applications

FTP protocol

- Control connection (on server port 21)
  - Client sends commands and receives responses
  - Connection persists across multiple commands
- FTP commands
  - Specification includes more than 30 commands
  - Each command ends with a carriage return and a line feed (“`\r\n`” in C)
  - Server responds with a three-digit code and optional human-readable text (e. g. “226 transfer completed”)
- Try it at the UNIX prompt
  - ftp ftp.cs.umd.edu
  - ID “anonymous” and password as your e-mail address

Example commands: authentication

- *USER:* specify the user name to log in as
- *PASS:* specify the user's password

Example commands: exploring the files

- *`LIST:`* list the files for the given file specification
- *`CWD:`* change to the given directory

Example commands: downloading and uploading files

- *TYPE:* set type to ASCII (A) or binary image (I)
- *RETR:* retrieve the given file
- *STOR:* upload the given file

Example commands: closing the connection

QUIT: close the FTP connection

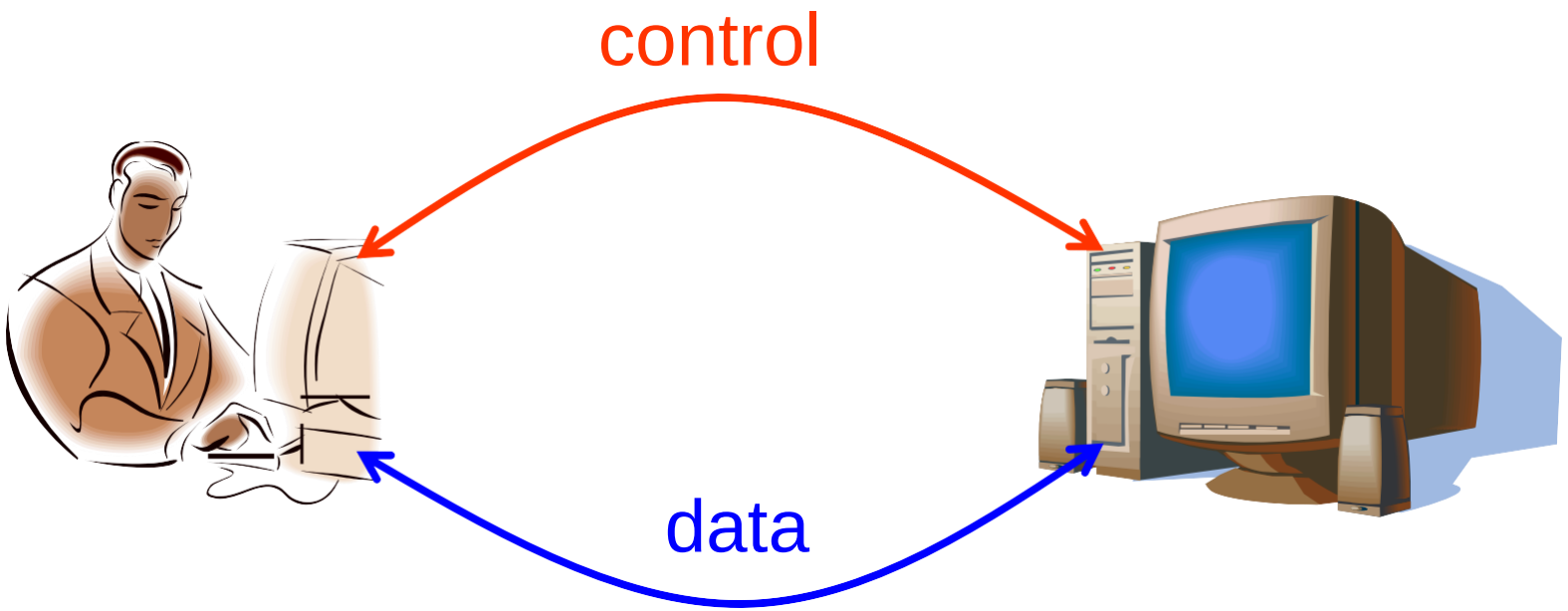
Server response codes

Code	Name	Meaning
1XX	Positive Preliminary Reply	The action is being started but expect another reply before sending the next command.
2XX	Positive Completion Reply	The action succeeded and a new command can be sent.
3XX	Positive Intermediary Reply	The command was accepted but another command is now required.
4XX	Transient Negative Completion Reply	The command failed and should be retried later.
5XX	Permanent Negative Completion Reply	The command failed and should not be retried.

FTP data transfer

Separate data connections:

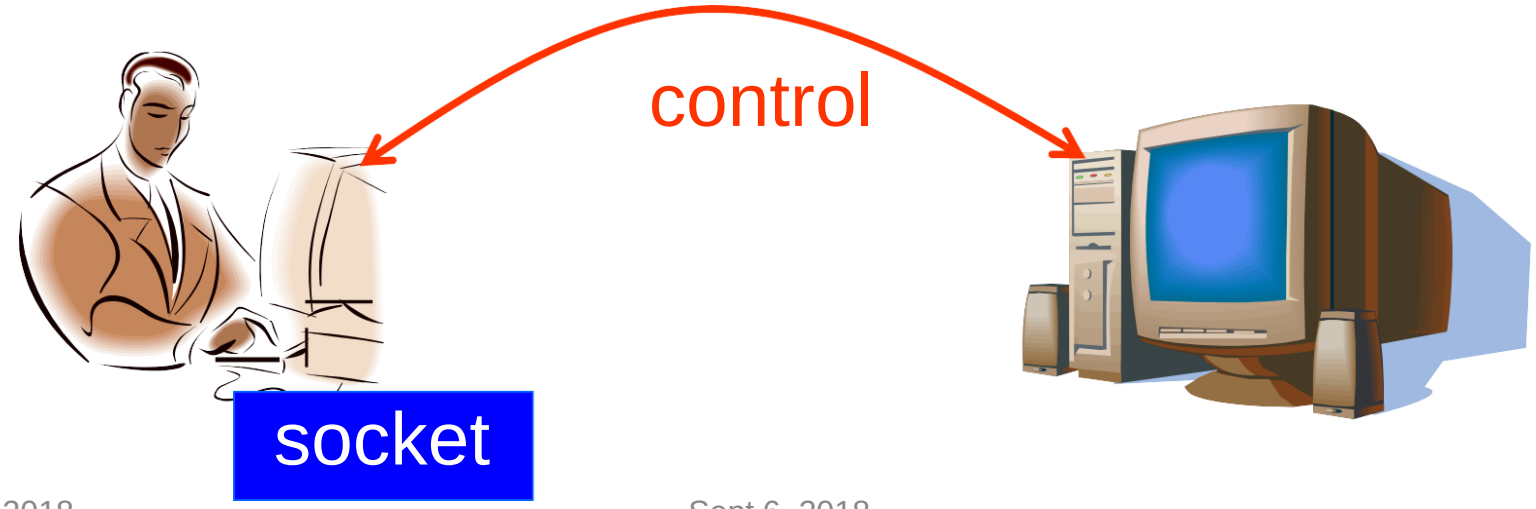
- To send lists of files (`LIST`)
- To retrieve a file (`RETR`)
- To upload a file (`STOR`)



Creating the data connection

The client acts like a server.

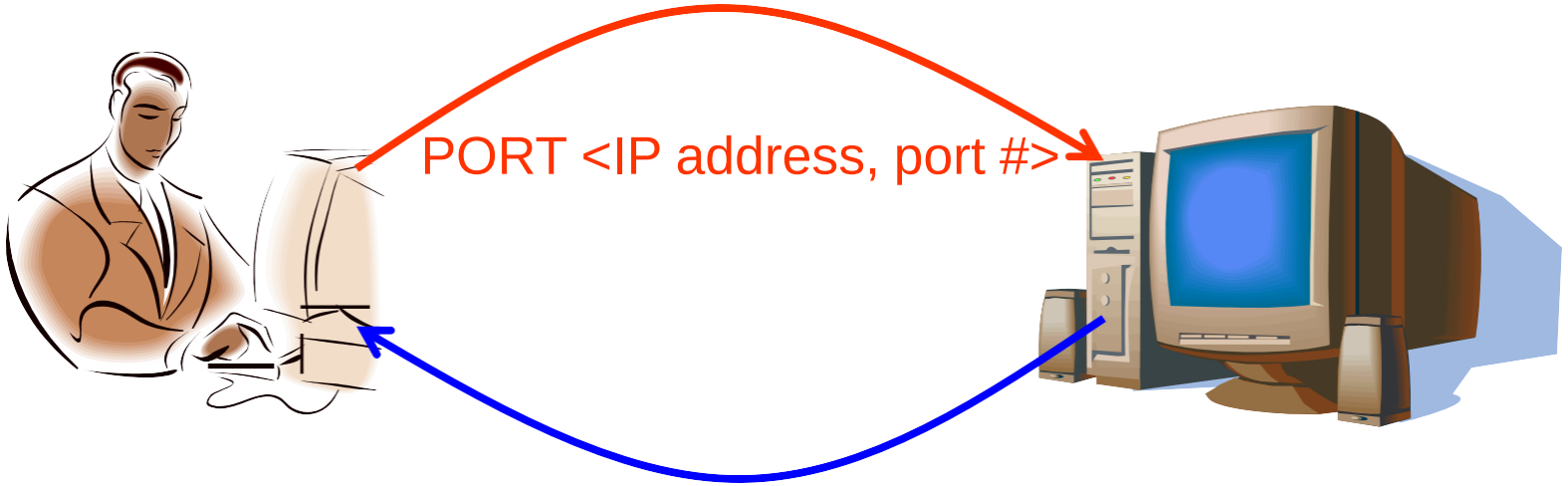
- Creates a socket
- Acquires an ephemeral port number
- Binds an address and port number
- Waits to hear from the FTP server



Creating the data connection

The server doesn't know the port number!

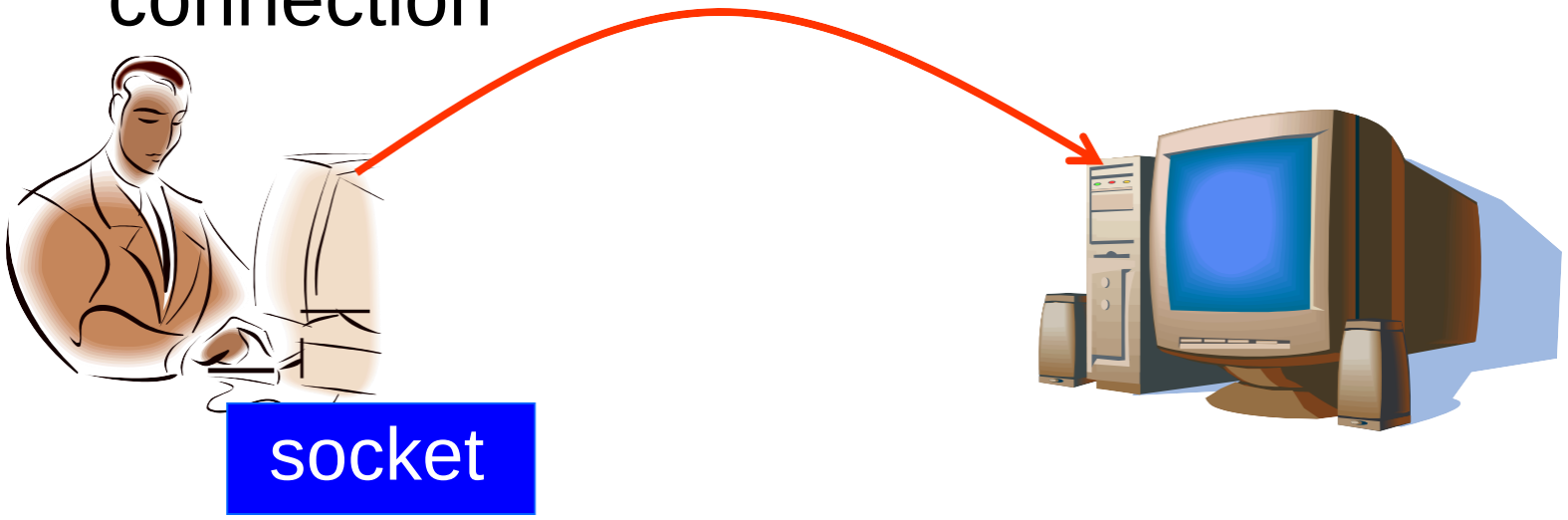
- So, the client tells the server the port number
- Using the PORT command on the control connection



Creating the data connection

The server initiates the data connection.

- The server connects to the socket on the client machine
- The client accepts to complete the connection



Out-of-band control

(from Wikipedia)

- a characteristic of network protocols
- passes control data on a separate connection from main data
- **Control data**: e. g. user ID, password, put/get commands

Why out-of-band control?

- Avoids need to mark the end of the data transfer
  - Data transfer ends by closing of data connection
  - Yet, the control connection stays up
- Aborting a data transfer
  - Can abort a transfer without killing the control connection
  - This avoids requiring the user to log in again
  - Done with an `ABOR` on the control connection
- Third-party file transfer between two hosts
  - Data connection could go to a different hosts
  - Sends a different client IP address to the server
  - e. g. user coordinates transfer between two servers

Summary

- Client-server paradigm
  - Model of communication between end hosts
  - Client asks, and server answers
- Sockets
  - Simple byte-stream and messages abstractions
  - Common application programmable interface
- File-Transfer Protocol (FTP)
  - Protocol for downloading and uploading files
  - Separate control and data connections (out-of-band control)