

CSMC 412

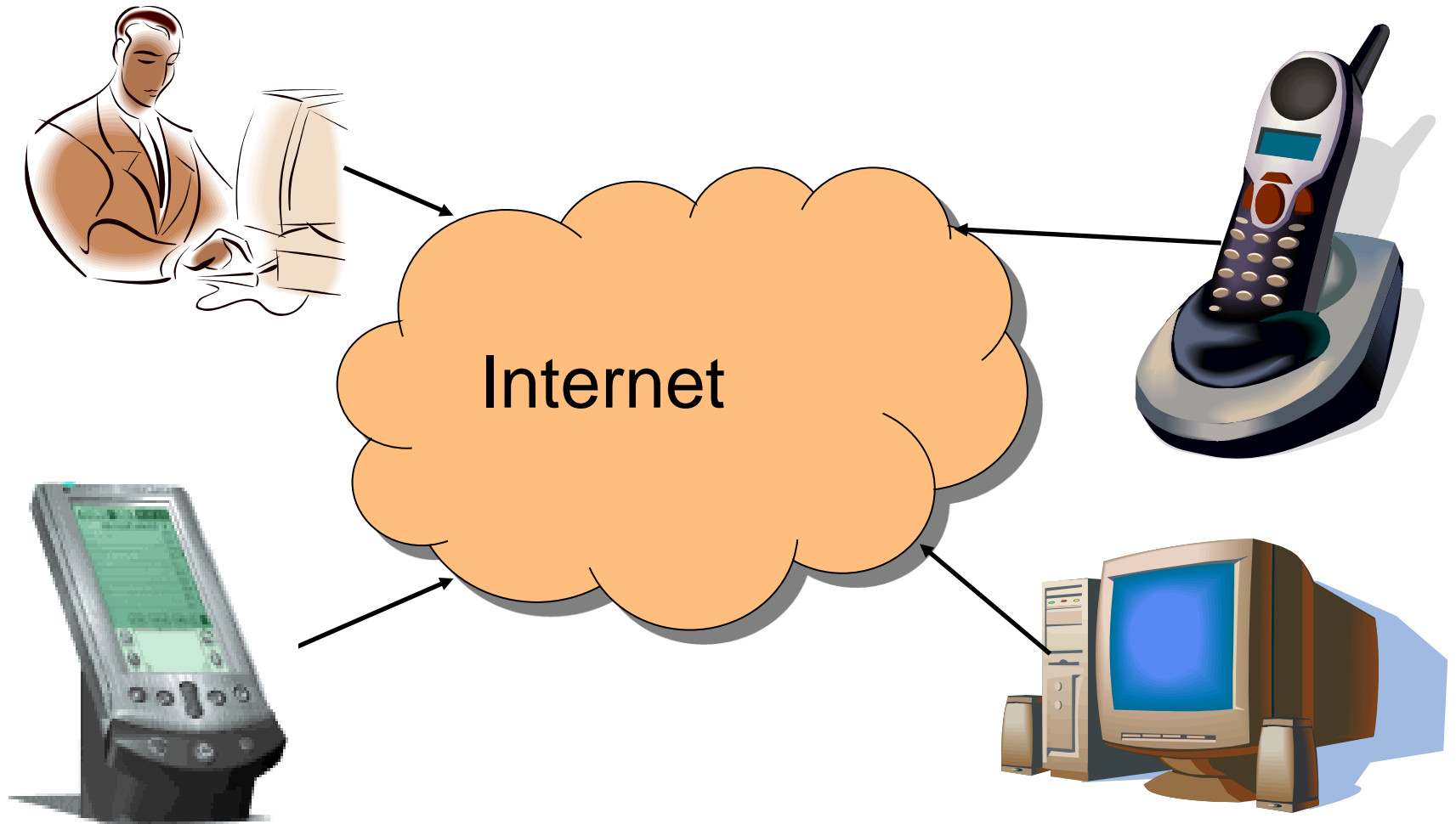
Computer Networks Prof. Ashok K Agrawala

© 2011 Ashok Agrawala
Set 2

Contents

- 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

End System: Computer on the 'Net

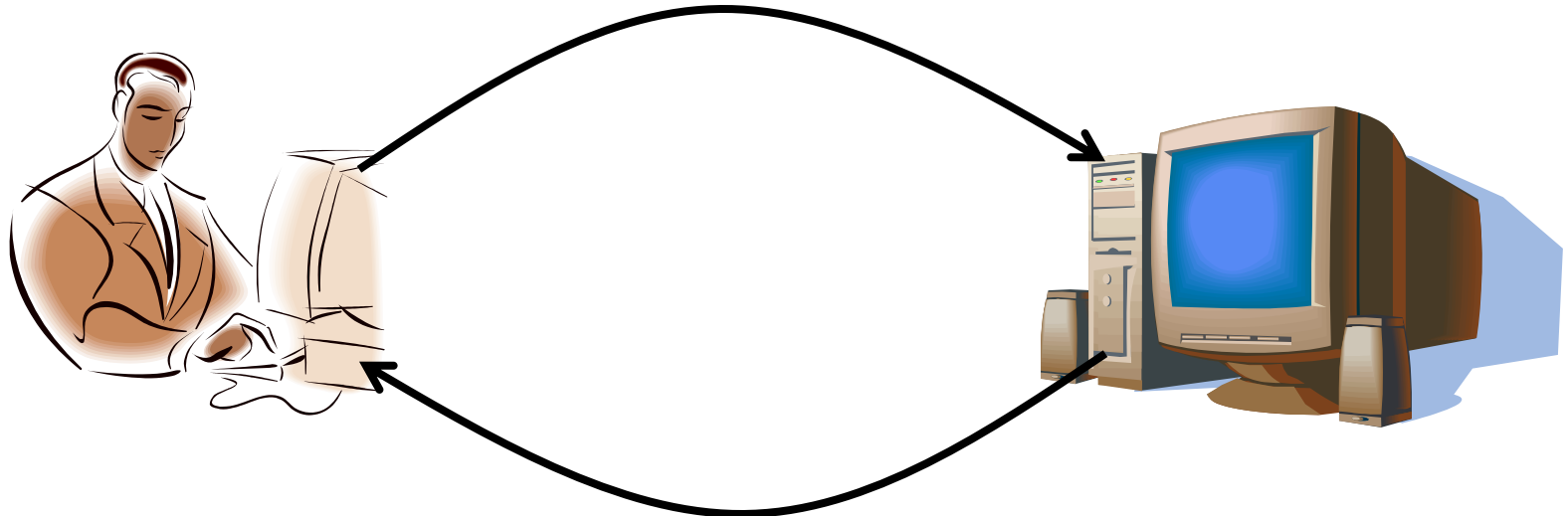


Also known as a "host"...

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

GET /index.html

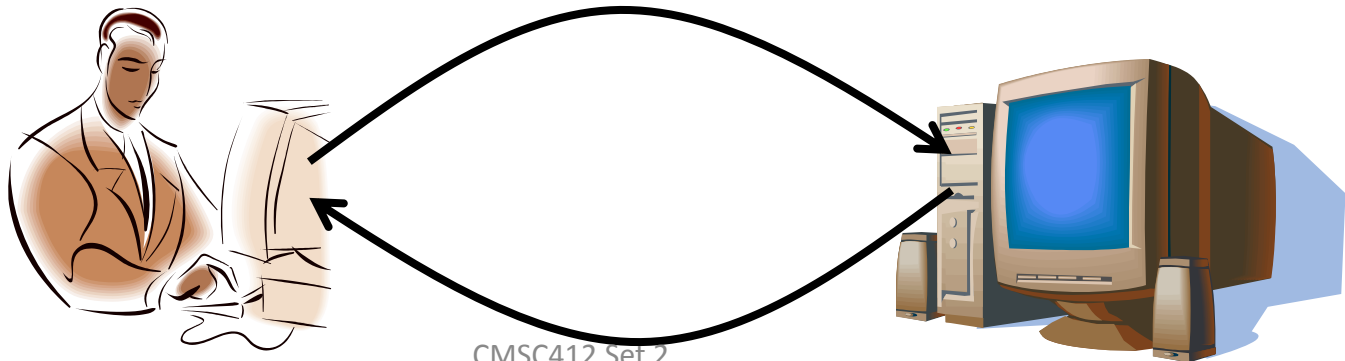


Clients Are Not Necessarily Human

- Example: Web crawler (or spider)
 - Automated client program
 - Tries to discover & download many Web pages
 - Forms the basis of search engines like Google
- Spider client
 - 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”
 - Services 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



Peer-to-Peer Communication

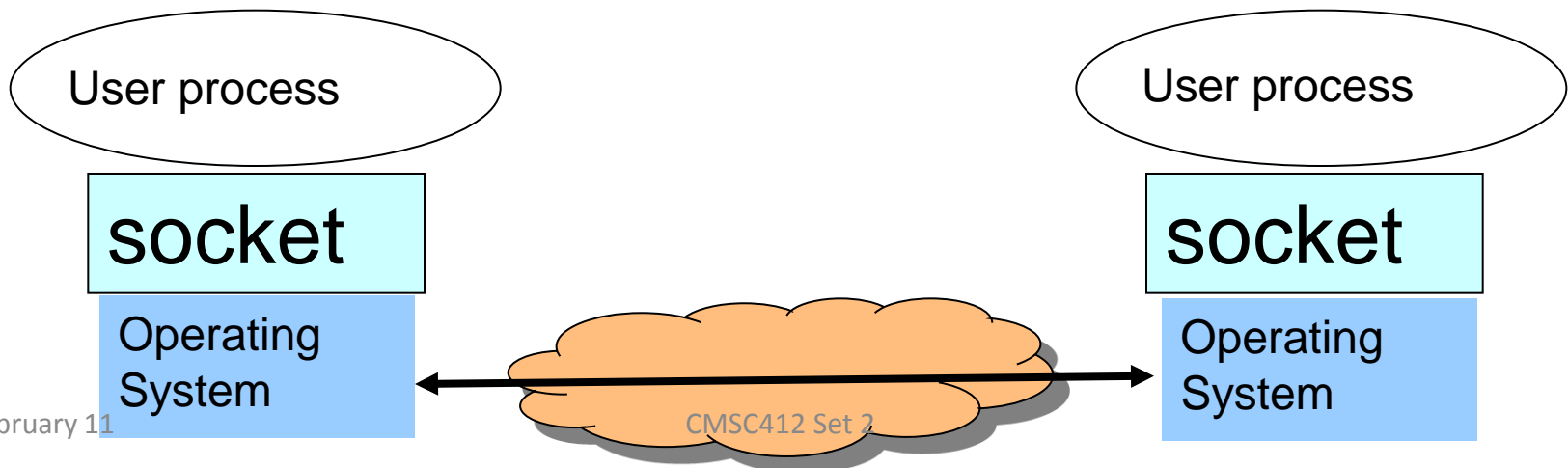
- No always-on server at the center of it all
 - Hosts can come and go, and change addresses
 - Hosts may have a different address each time
- Example: peer-to-peer file sharing
 - Any host can request files, send files, query to find where a file is located, respond to queries, and forward queries
 - Scalability by harnessing millions of peers
 - Each peer acting as both a client and server

Client and Server Processes

- Program vs. process
 - Program: collection of code
 - Process: a running program on a host
- 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 the 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

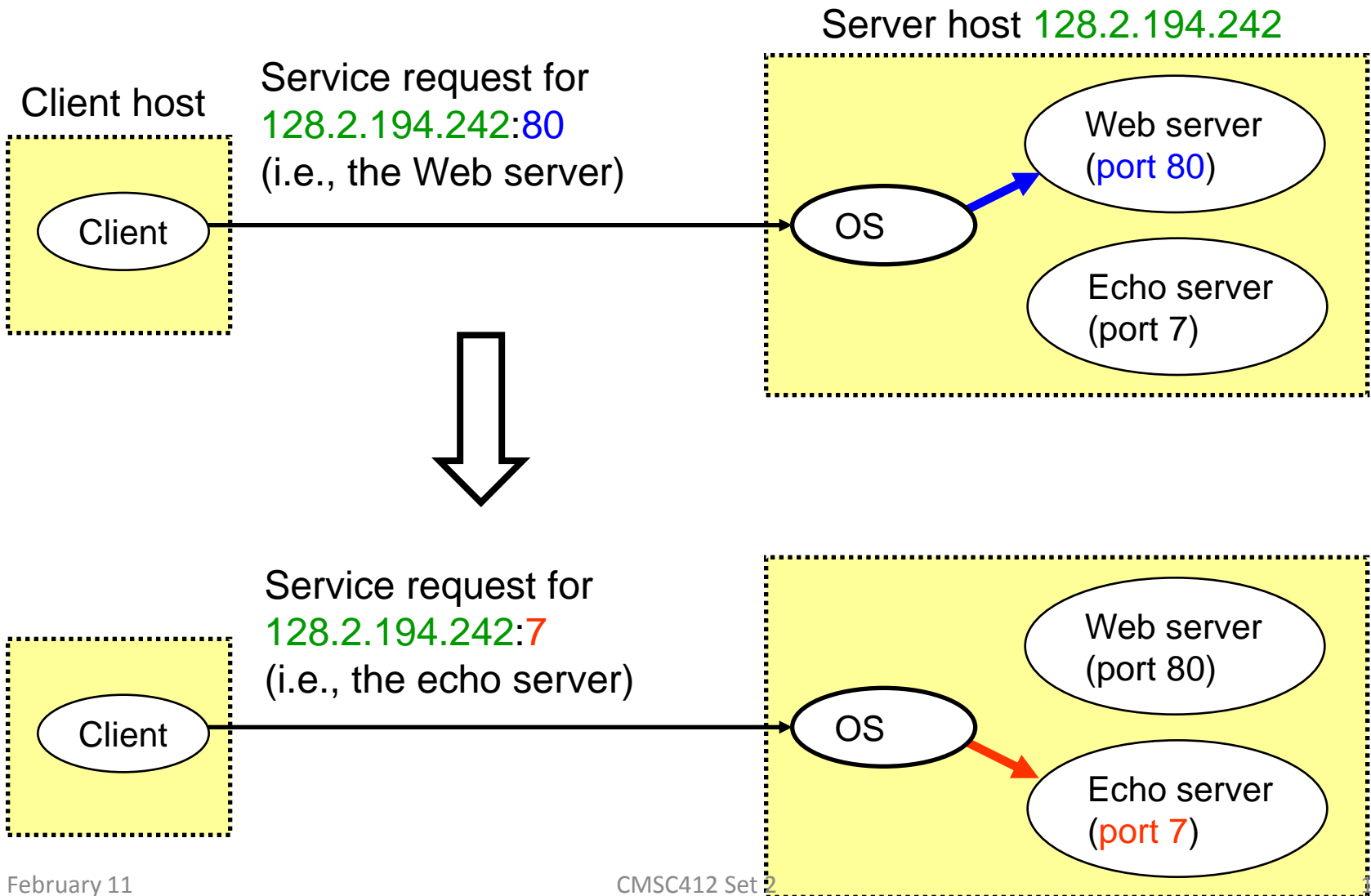
- Sending message from one process to another
 - Message must traverse the underlying network
- Process sends and receives through a “socket”
 - In essence, the doorway leading in/out of the house
- Socket as an Application Programming Interface
 - Supports the creation of network applications



Identifying the Receiving Process

- Sending process must identify the receiver
 - Name or address of the receiving end host
 - Identifier that specifies the receiving process
- Receiving host
 - Destination address that uniquely identifies the host
 - An IP address is a 32-bit quantity
- Receiving process
 - Host may be running many different processes
 - Destination port that uniquely identifies the socket
 - A port number is a 16-bit quantity

Using Ports to Identify Services



Knowing What Port Number To Use

- Popular applications have well-known ports
 - E.g., port 80 for Web and port 25 for e-mail
 - Well-known ports listed at <http://www.iana.org>
- Well-known vs. ephemeral ports
 - Server has a well-known port (e.g., port 80)
 - Between 0 and 1023
 - Client picks an unused ephemeral (i.e., temporary) port
 - Between 1024 and 65535
- Uniquely identifying the traffic between the hosts
 - Two IP addresses and two port numbers
 - Underlying transport protocol (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 #
- 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

- Prepare to communicate
 - Create a socket
 - Determine server address and port number
 - Initiate the connection to the server
- 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)
- Close the socket

Creating a Socket: `socket()`

- Operation to create a 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, socklen_t addrlen)*
 - Arguments: socket descriptor, server address, and address size
 - Returns 0 on success, and -1 if an error occurs

Sending and Receiving Data

- 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

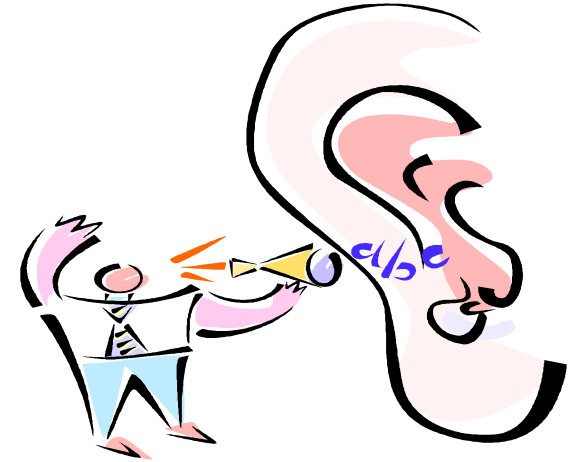
- Hosts differ in how they store data
 - E.g., four-byte number (byte3, byte2, byte1, byte0)
- Little endian (“little end comes first”) ← Intel PCs!!!
 - Low-order byte stored at the lowest memory location
 - Byte0, byte1, byte2, byte3
- Big endian (“big end comes first”)
 - High-order byte stored at lowest memory location
 - Byte3, byte2, byte1, byte 0
19IP 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 connections
 - ... 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

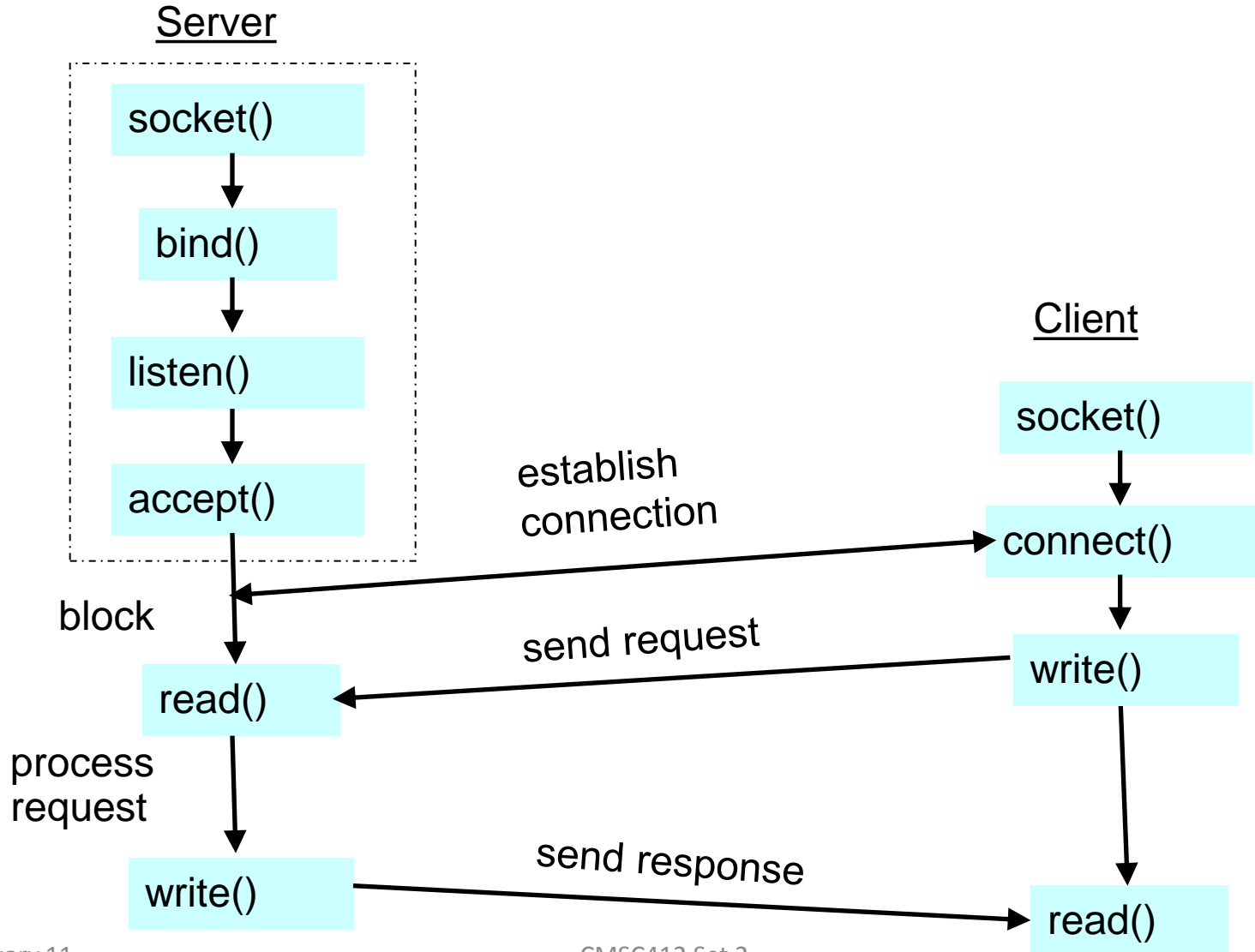
Server Preparing its Socket

- 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

- Accept a new connection from a client
 - *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
 - What happens if no clients are around?
 - The *accept()* call blocks waiting for a client
 - What happens if too many clients are around?
 - 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?

- Serializing requests is inefficient
 - Server can process just one request at a time
 - All other clients must wait until previous one is done
- Need to time share the server machine
 - Alternate between servicing different requests
 - Do a little work on one request, then switch to another
 - Small tasks, like reading HTTP request, locating the associated file, reading the disk, transmitting parts of the response, etc.
 - Or, start a new process to handle each request
 - Allow the operating system to share the CPU across processes
 - Or, 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 Assignment #1

- 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
 - Client program connects to FTP server
 - ... and provides a login id and password
 - ... and allows the user to explore the directories
 - ... and download and upload files with the server
- A predecessor of the Web (RFC 959 in 1985)
 - Requires user to know the name of the server machine
 - ... and have an account on the machine
 - ... and find the directory where the files are stored
 - ... and know whether the file is text or binary
 - ... and know what tool to run to render and edit the file
- That is, no URL, hypertext, and 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

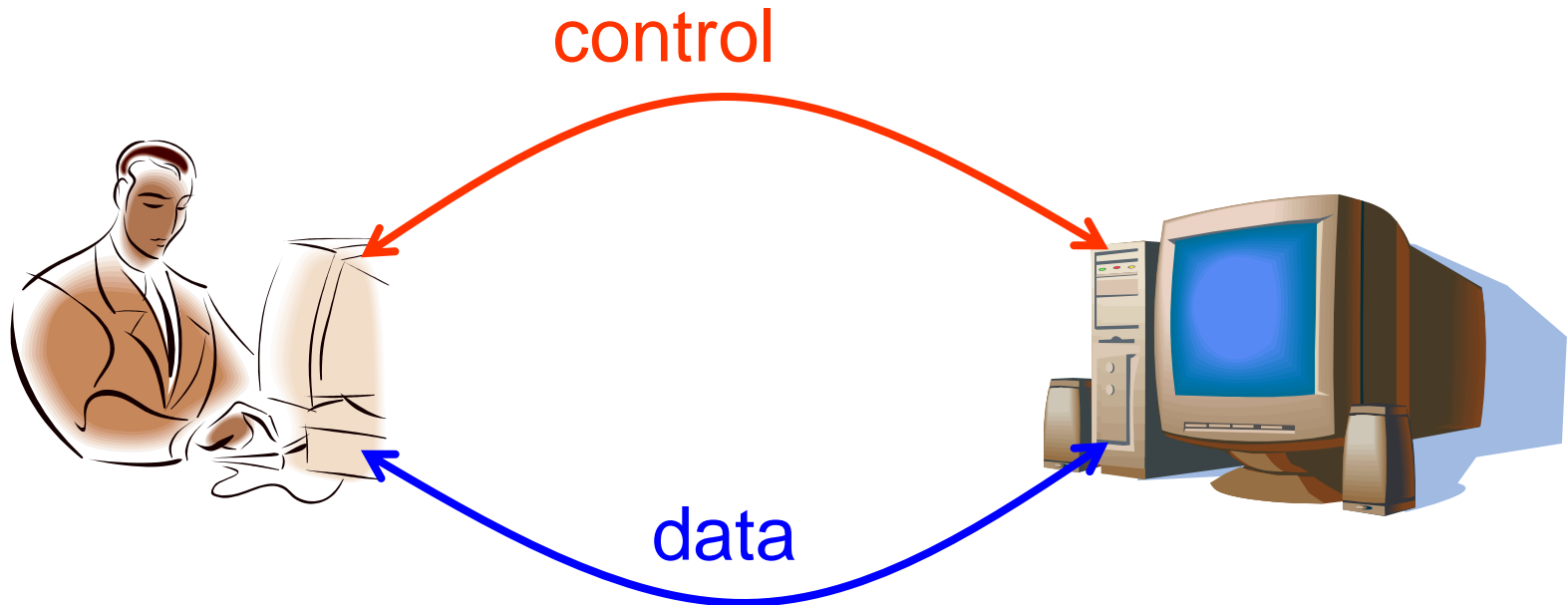
- Authentication
 - USER: specify the user name to log in as
 - PASS: specify the user's password
- Exploring the files
 - LIST: list the files for the given file specification
 - CWD: change to the given directory
- Downloading and uploading files
 - TYPE: set type to ASCII (A) or binary image (I)
 - RETR: retrieve the given file
 - STOR: upload the given file
- Closing the connection
 - QUIT: close the FTP connection

Server Response Codes

- 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 intermediate 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.

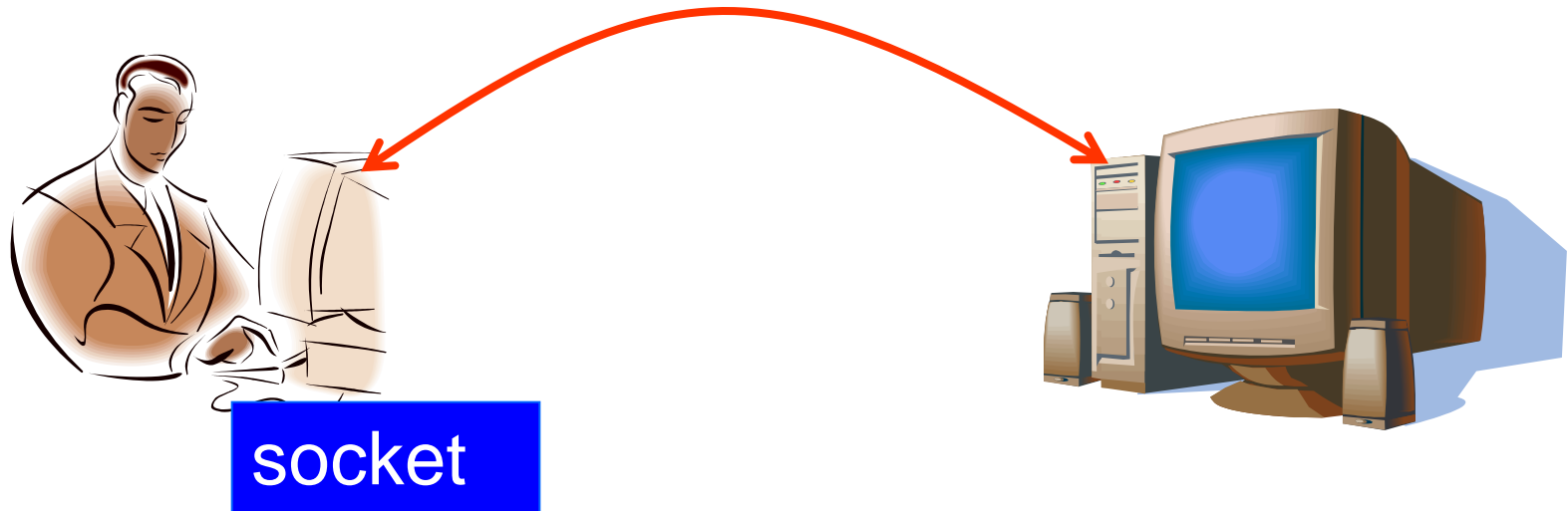
FTP Data Transfer

- Separate data connection
 - To send lists of files (LIST)
 - To retrieve a file (RETR)
 - To upload a file (STOR)



Creating the Data Connection

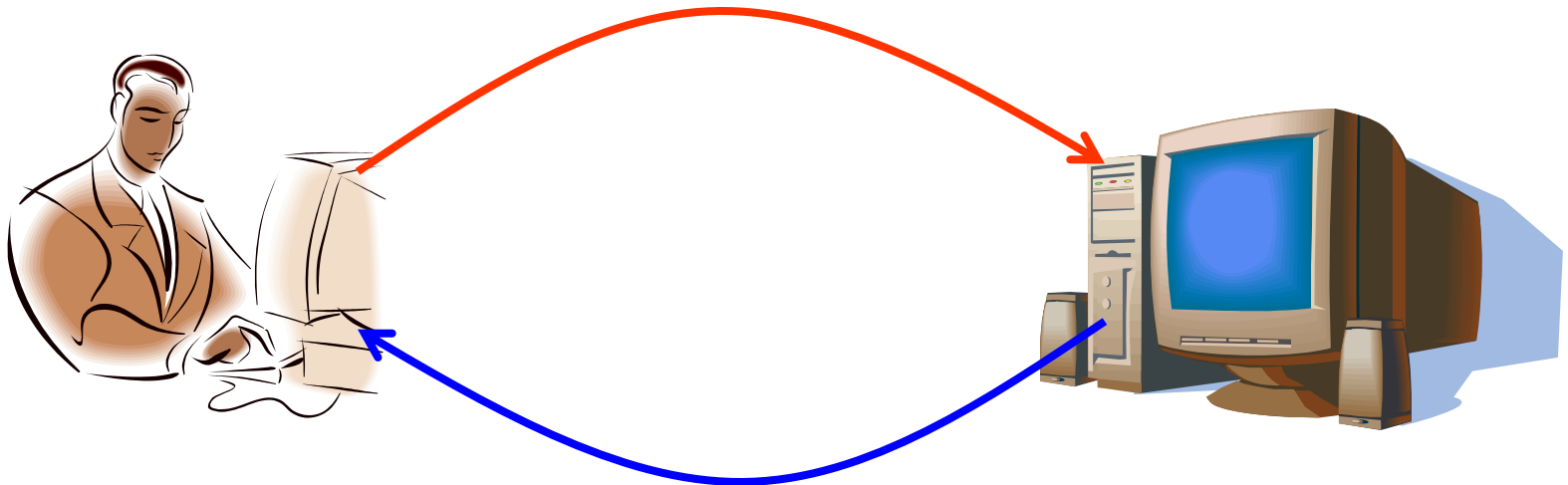
- Client acts like a server
 - Creates a socket
 - Client acquires an ephemeral port number
 - Binds an address and port number
 - Waits to hear from the FTP server
- control**



Creating Data Connection (cont.)

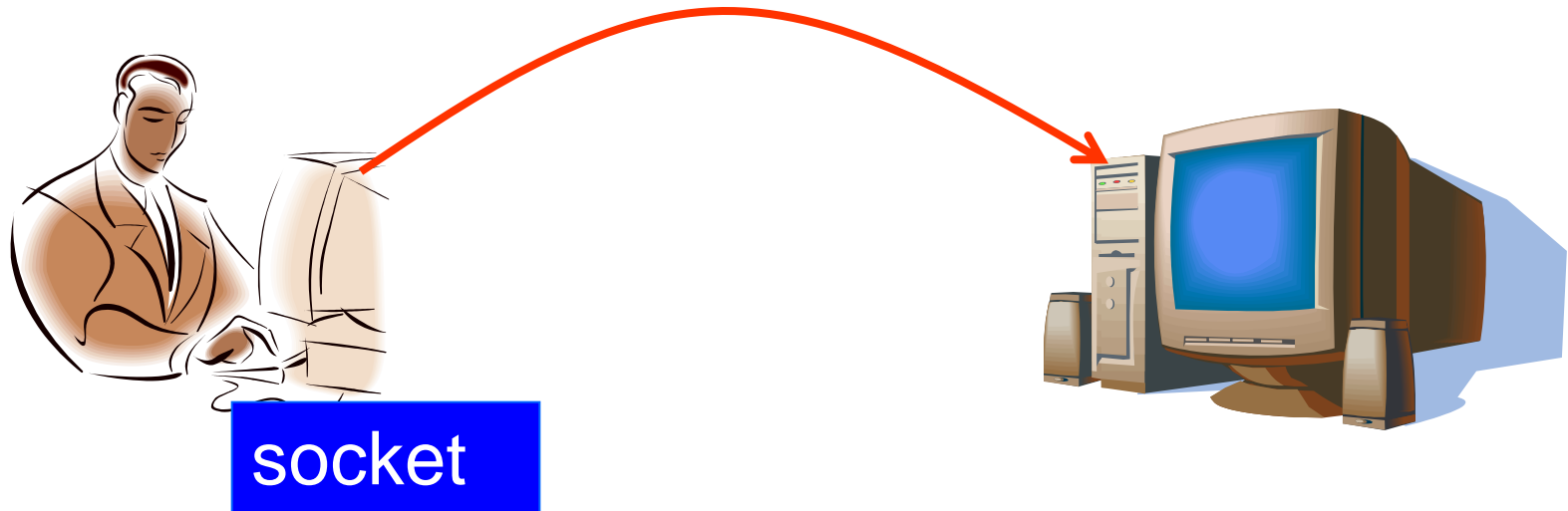
- But, the server doesn't know the port number
 - So, the client tells the server the port number
 - Using the PORT command on the control connection

PORT <IP address, port #>



Creating Data Connection (cont)

- Then, the server initiates the data connection
 - Connects to the socket on the client machine
 - ... and the client accepts to complete the connection



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
 - ... which 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
 - ... by sending a different client IP address to the server
 - E.g., user coordinates transfer between two servers

Closing

- 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