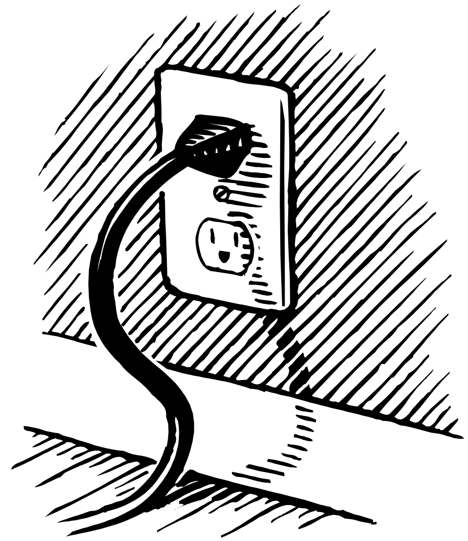


Networked Applications: Sockets

COS 461: Computer Networks
Spring 2010 (MW 3:00-4:20 in CS 105)

Michael Freedman
Teaching Assistants: Muneeb Ali and David Shue
http://www.cs.princeton.edu/courses/archive/spr10/cos461/

Class Logistics

- Slides and reading assignments online at
  - http://www.cs.princeton.edu/courses/archive/spr10/cos461/
  - Reading: chapter 1 and socket programming guides
- Course e-mail list
  - https://lists.cs.princeton.edu/mailman/listinfo/cos461
- Office hours
  - Muneeb: Thu 4:20-5:20pm
  - David : Fri 2:00-3:00pm

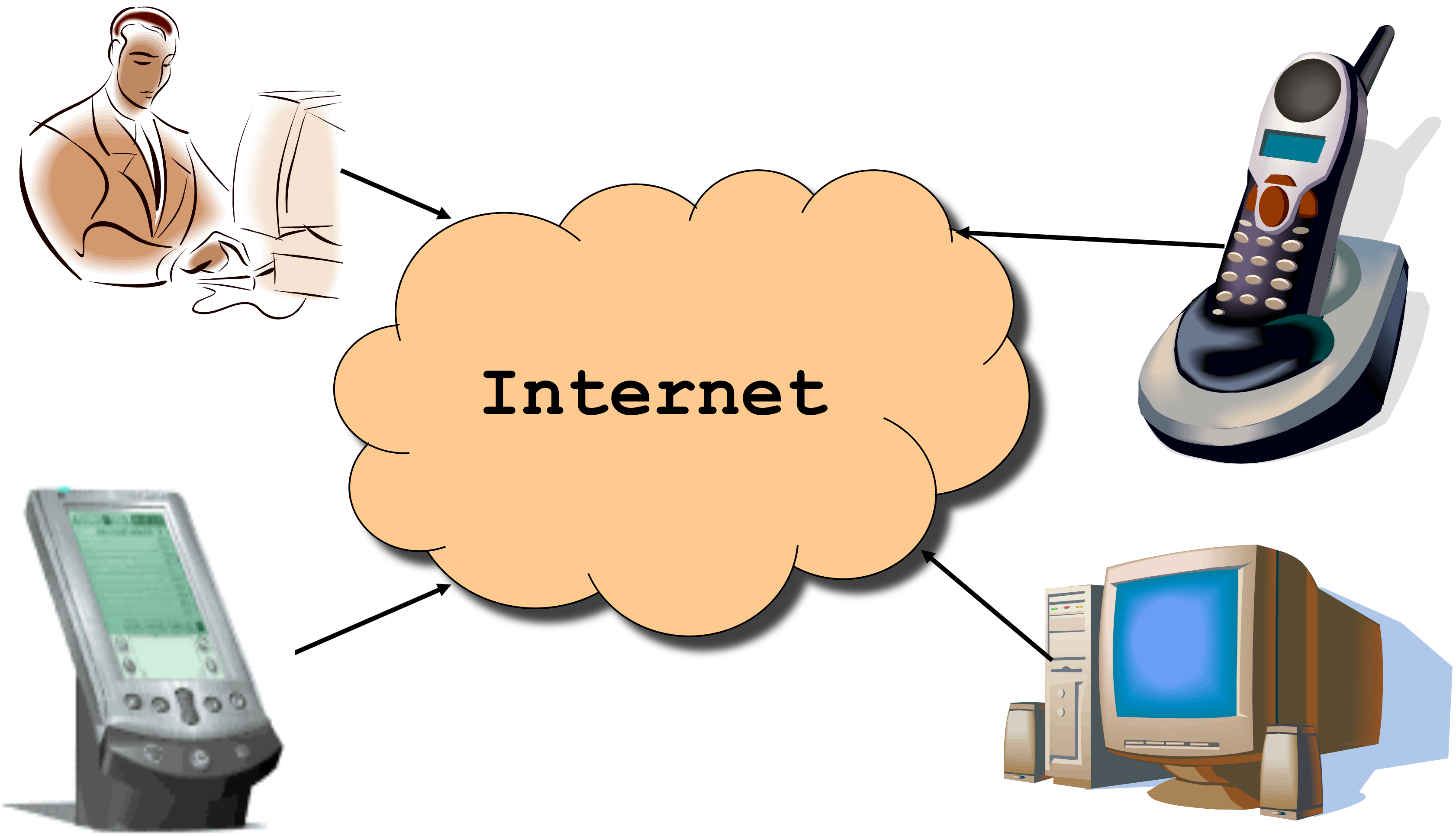
Class Logistics

- Computer accounts in FC 010
  - CS account (can request a CS "class account")
    - https://csguide.cs.princeton.edu/requests/account
    - SSH to portal.cs.princeton.edu with your CS account
  - Account on FC 010
    - For students who are enrolled in the class
    - https://csguide.cs.princeton.edu/resources/friend
    - SSH to labpc-XX.cs.princeton.edu with OIT password
- Programming assignment 1
  - Client and server programs to copy and print data
  - Assignment is posted on the course Web site
  - Due 11:59pm on Sunday February 14

Goals of Today's Lecture

- Client-server paradigm
  - End systems
  - Clients and servers
- Sockets
  - Socket abstraction
  - Socket programming in UNIX
- HyperText Transfer Protocol (HTTP)
  - URL, HTML, and HTTP
  - Clients, proxies, and servers
  - Example transactions using sockets

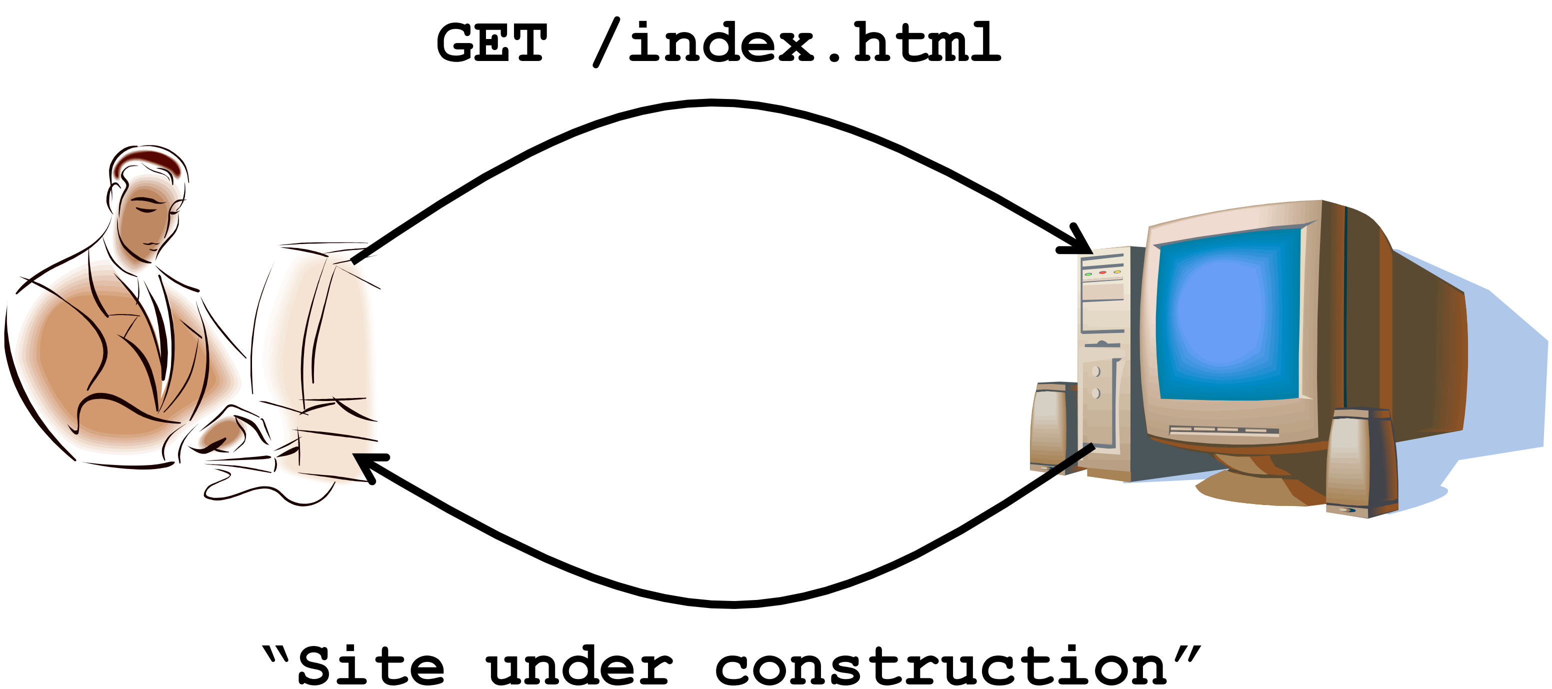
End System: Computer on the 'Net



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

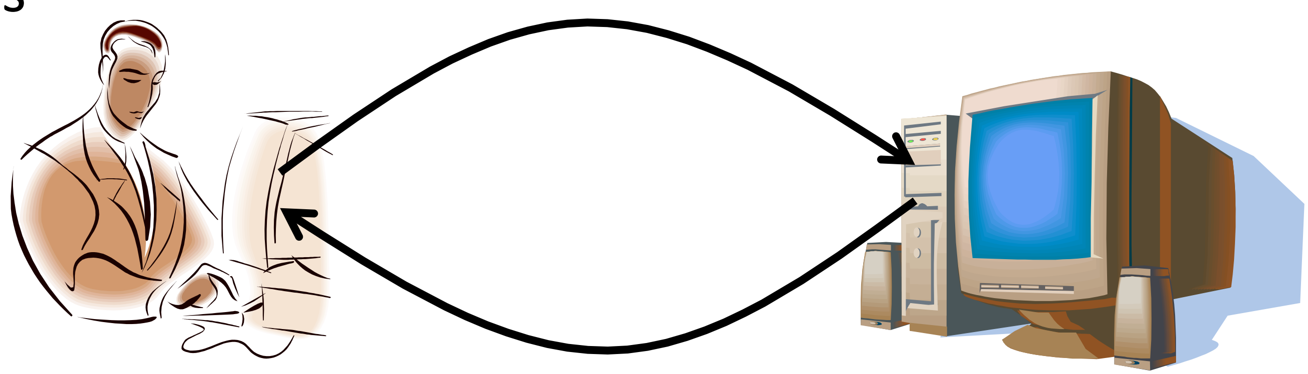


Clients Are Not Necessarily Human

- Example: Web crawler (or spider)
  - Automated client program
  - Tries to discover & 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 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 fixed, 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 a file's location, respond to queries, …
  - Scalability by harnessing millions of peers
  - Each peer acting as both a client and server
- Well, mostly no central server, but how to initially discover peers? ("bootstrapping")

Client and Server Processes

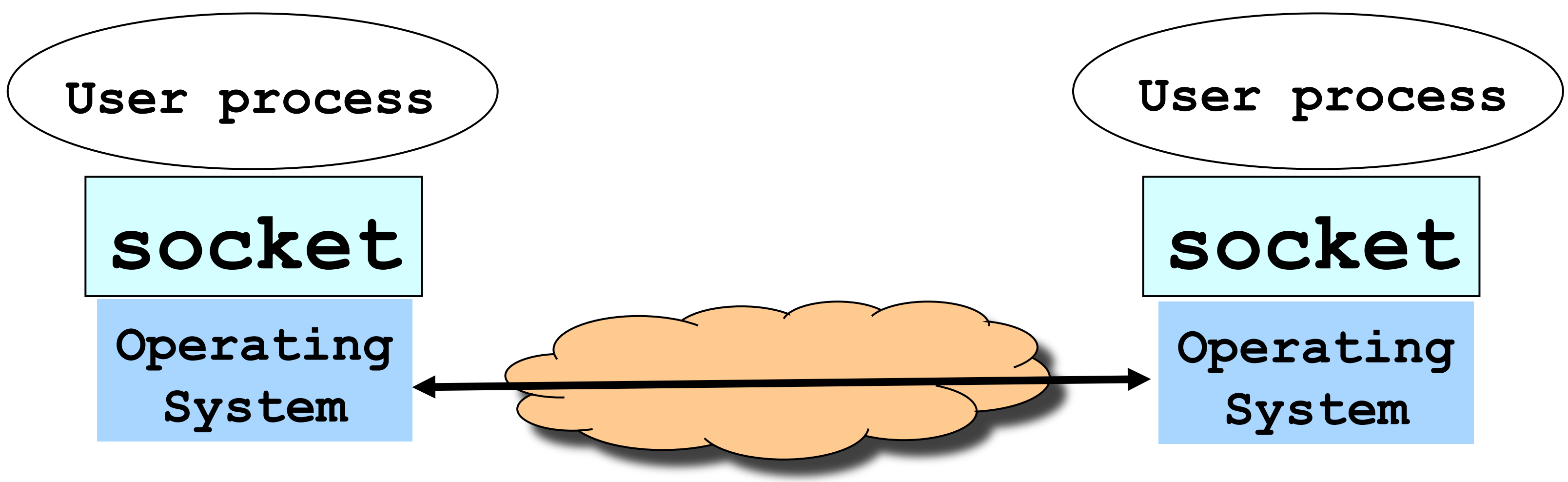
- Program vs. process
  - Program: collection of code
  - Process: a running program on a host
- Communication between processes
  - Same end host: inter-process communication
    - Governed by the operating system on the end host
  - Different end hosts: exchanging messages
    - Governed by the network protocols
- Client and server processes
  - Client process: process that initiates communication
  - Server process: process that waits to be contacted

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 destination port number (e.g., 80)
- Application
  - Read data from and write data to the socket
  - Interpret the data (e.g., render a Web page)

Socket: End Point of Communication

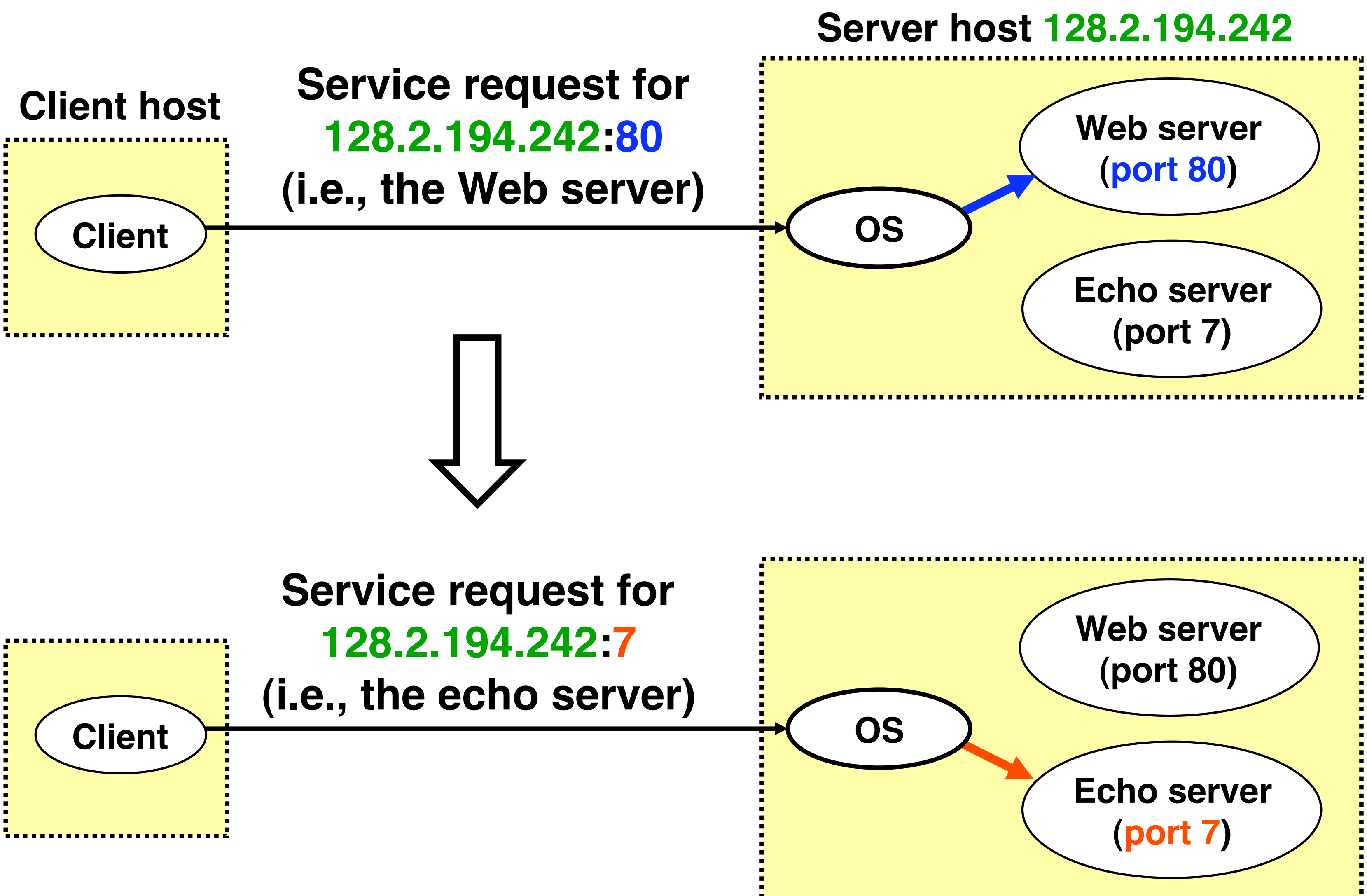
- Sending message from one process to another
  - Message must traverse the underlying network
- Process sends and receives through a “socket”
  - In essence, the doorway leading in/out of the house
- Socket as an Application Programming Interface
  - Supports the creation of network applications



Identifying the Receiving Process

- Sending process must identify the receiver
  - The receiving end host machine
  - The specific socket in a process on that machine
- Receiving host
  - Destination address that uniquely identifies the host
  - An IP address is a 32-bit quantity
- Receiving socket
  - 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
  - See http://www.iana.org/assignments/port-numbers
- Well-known vs. ephemeral ports
  - Server has a well-known port (e.g., port 80)
    - Between 0 and 1023 (requires root to use)
  - Client picks an unused ephemeral (i.e., temporary) port
    - Between 1024 and 65535
- Uniquely identifying traffic between the hosts
  - Two IP addresses and two port numbers
  - Underlying transport protocol (e.g., TCP or UDP)
  - This is the "5-tuple" I discussed last lecture

Port Numbers are Unique per Host

- Port number uniquely identifies the socket
  - Cannot use same port number twice with same address
  - Otherwise, the OS can't demultiplex packets correctly
- Operating system enforces uniqueness
  - OS keeps track of which port numbers are in use
  - Doesn't let the second program use the port number
- Example: two Web servers running on a machine
  - They cannot both use port "80", the standard port #
  - So, the second one might use a non-standard port #
  - E.g., http://www.cnn.com:8080

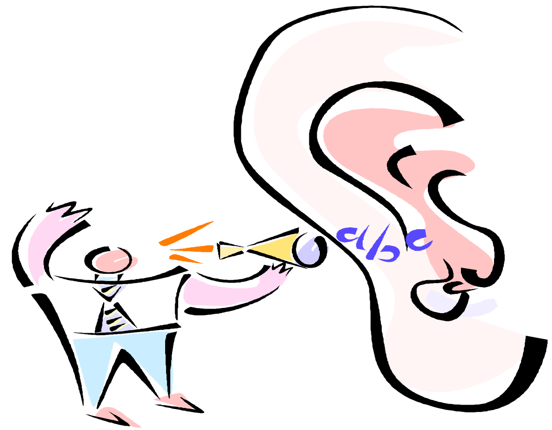
UNIX Socket API

- 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
- API implemented as system calls
  - E.g., connect, 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

Servers Differ From Clients

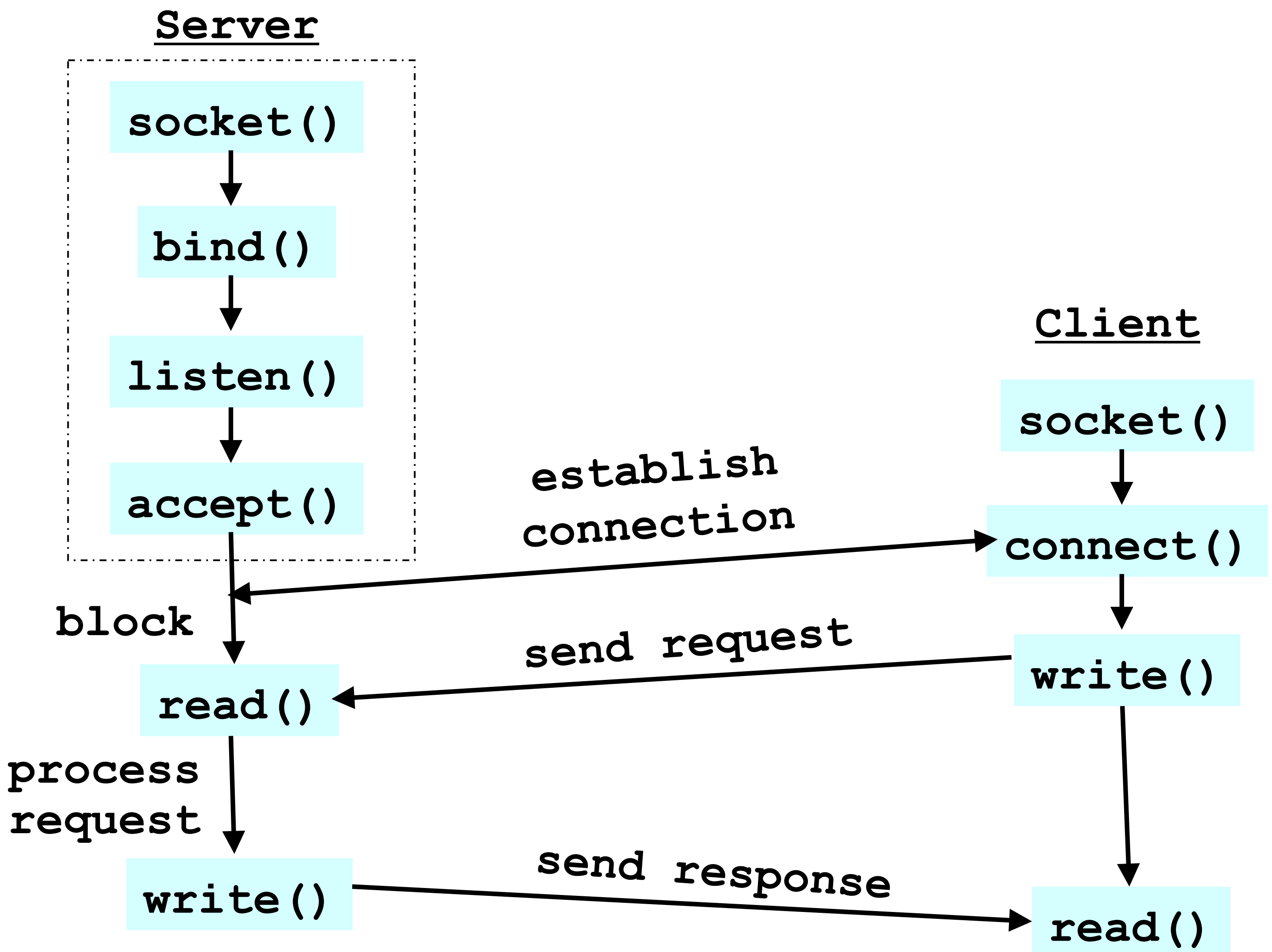


- Passive open
  - Prepare to accept connections
  - … but don't actually establish
  - … until hearing from a client
- Hearing from multiple clients
  - Allowing a backlog of waiting clients
  - ... in case several try to communicate at once
- Create a socket for each client
  - Upon accepting a new 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

Putting it All Together



Client Creating a Socket: socket()

- Creating a socket
  - `int socket(int domain, int type, int protocol)`
  - Returns a file descriptor (or handle) for the socket
  - Originally designed to support any protocol suite
- Domain: protocol family
  - PF_INET for the Internet (IPv4)
- Type: semantics of the communication
  - SOCK_STREAM: reliable byte stream (TCP)
  - SOCK_DGRAM: message-oriented service (UDP)
- Protocol: specific protocol
  - UNSPEC: unspecified
  - (PF_INET and SOCK_STREAM already implies TCP)

Client: Learning Server Address/Port

- Server typically known by name and service
  - E.g., "www.cnn.com" and "http"
- Need to translate into IP address and port #
  - E.g., "64.236.16.20" and "80"

- Translating the server's name to an address
  - **`struct hostent *gethostbyname(char *name)`**
  - Argument: host name (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")
  - Static config in `/etc/services`

Client: Connecting Socket to the Server

- Client contacts the server to establish connection
  - Associate the socket with the server address/port
  - Acquire a local port number (assigned by the OS)
  - Request connection to server, who hopefully accepts

- 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
    ```

Client: Sending 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 bytes written, and -1 on error

Client: Receiving Data

- 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
  - Why do you need len?
  - What happens if buf's size < len?
- Closing the socket
  - **`int close(int sockfd)`**

Server: Server Preparing its Socket

- Server creates a socket and binds address/port
  - Server creates a socket, just like the client does
  - Server associates the socket with the port number (and hopefully no other process is already using it!)
  - Choose port "0" and let kernel assign ephemeral port

- Create a socket
  - ```
    int socket (int domain,
                int type, int protocol)
    ```
- Bind socket to the local address and port number
  - ```
    int bind (int sockfd,
        struct sockaddr *my_addr,
        socklen_t addrlen)
    ```
  - Arguments: sockfd, server address, address length
  - Returns 0 on success, and -1 if an error occurs

Server: Allowing Clients to Wait

- Many client requests may arrive
  - Server cannot handle them all at the same time
  - Server could reject the requests, or let them wait
- Define how many connections can be pending
  - **`int listen(int sockfd, int backlog)`**
  - Arguments: socket descriptor and acceptable backlog
  - Returns a 0 on success, and -1 on error
- What if too many clients arrive?
  - Some requests don't get through
  - The Internet makes no promises…
  - And the client can always try again

Server: Accepting Client Connection

- Now all the server can do is wait…
  - Waits for connection request to arrive
  - Blocking until the request arrives
  - And then accepting the new request

- Accept a new connection from a client
  - **int accept(int sockfd,**

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

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

Server: 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
  – What makes this inefficient?
- May need to time share the server machine
  – Alternate between servicing different requests
    - Do a little work on one request, then switch when you are waiting for some other resource (e.g., reading file from disk)
    - "Nonblocking I/O"
  – Or, use a different process/thread for each request
    - Allow OS to share the CPU(s) across processes
  – Or, some hybrid of these two approaches

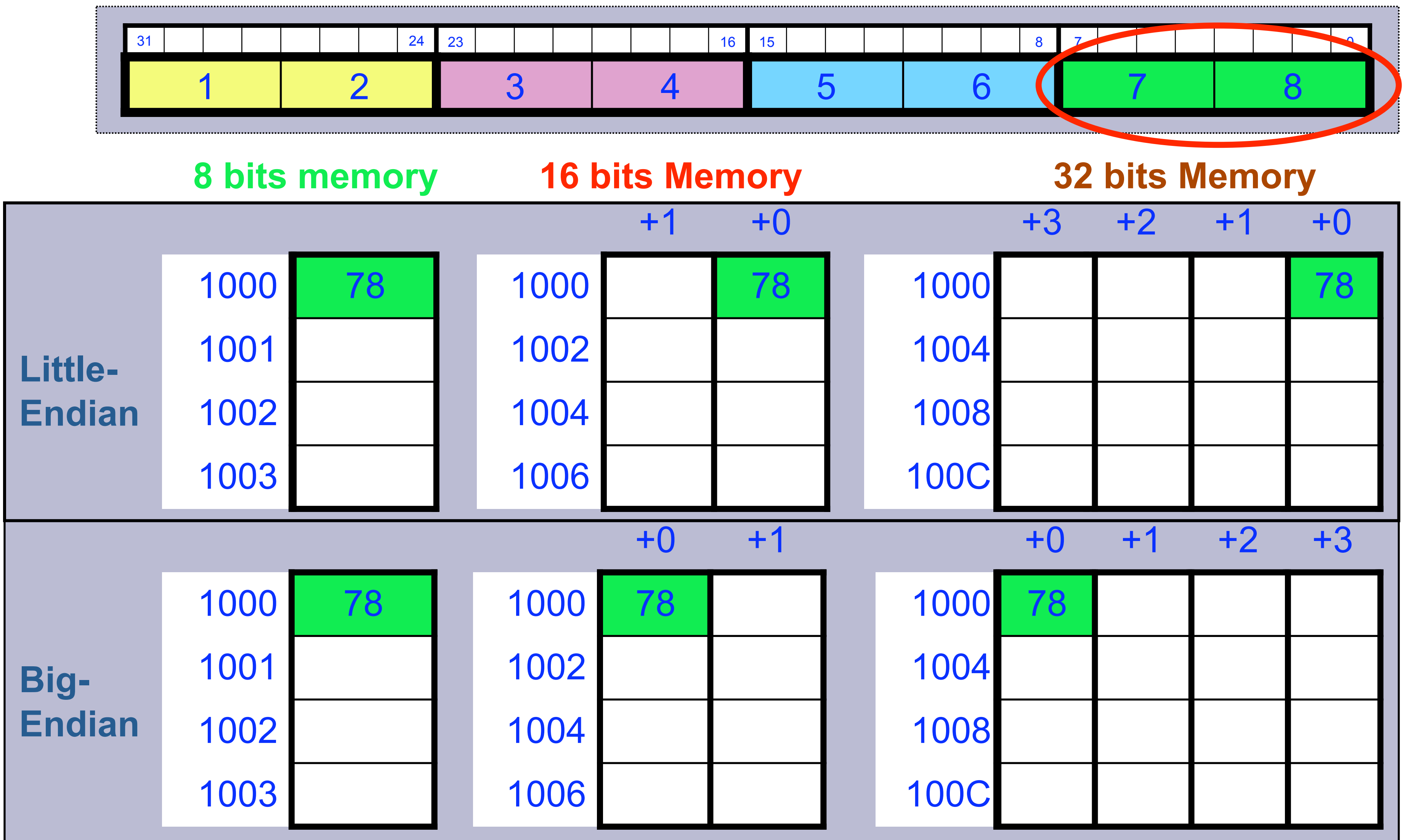
Client and Server: Cleaning House

- Once the connection is open
  - Both sides and read and write
  - Two unidirectional streams of data
  - In practice, client writes first, and server reads
  - … then server writes, and client reads, and so on
- Closing down the connection
  - Either side can close the connection
  - … using the **`close()`** system call
- What about the data still “in flight”
  - Data in flight still reaches the other end
  - So, server can **`close()`** before client finishes reading

One Annoying Thing: Byte Order

- Hosts differ in how they store data
  – E.g., four-byte number (byte3, byte2, byte1, byte0)
- Little endian ("little end comes first"): Intel x86's
  – 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
- Makes it more difficult to write portable code
  – Client may be big or little endian machine
  – Server may be big or little endian machine

Endian Example: Where is the Byte?



IP is Big Endian

- But, what byte order is used "on the wire"
  - That is, what do the network protocol use?
- The Internet Protocols picked one convention
  - IP is big endian (aka "network byte order")
- Writing portable code require conversion
  - Use htons() and htonl() to convert to network byte order
  - Use ntohs() and ntohl() to convert to host order
- Hides details of what kind of machine you're on
  - Use the system calls when sending/receiving data structures longer than one byte

Using htonl and htons

```
int sockfd = // connected SOCK_STREAM
u_int32_t my_val  = 1234;
u_int16_t my_xtra = 16;

u_short bufsize = sizeof (struct data_t);
char *buf = New char[bufsize];
bzero (buf,  bufsize);

struct data_t *dat = (struct data_t *) buf;
dat->value = htonl (my_val);
dat->xtra  = htons (my_xtra);

int rc = write (sockfd, buf, bufsize);
```

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
  - 2-byte short int: (byte 1, byte 0) vs. (byte 0, byte 1)
  - 4-byte long int: (byte 3, … byte 0) vs. (byte 0, … byte 3)
  - String of one-byte chars (char 0, char 1, char 2, …) in both
- 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
- Higher-layer with defined types can do this for you
  - Java object serialization, RPC "marshalling"

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/
- …

The Web as an Example Application

The Web: URL, HTML, and HTTP

- Uniform Resource Locator (URL)
  - A pointer to a "black box" that accepts request methods
  - Formatted string with protocol (e.g., http), server name (e.g., www.cnn.com), and resource name (coolpic.jpg)
- HyperText Markup Language (HTML)
  - Representation of hyptertext documents in ASCII format
  - Format text, reference images, embed hyperlinks
  - Interpreted by Web browsers when rendering a page
- HyperText Transfer Protocol (HTTP)
  - Client-server protocol for transferring resources
  - Client sends request and server sends response

Example: HyperText Transfer Protocol

Request

```
GET /courses/archive/spr09/cos461/ HTTP/1.1
Host: www.cs.princeton.edu
User-Agent: Mozilla/4.03
<CRLF>
```

Response

```
HTTP/1.1 200 OK
Date: Mon, 4 Feb 2009 13:09:03 GMT
Server: Netscape-Enterprise/3.5.1
Content-Type: text/plain
Last-Modified: Mon, 4 Feb  2008 11:12:23 GMT
Content-Length: 21
<CRLF>
Site under construction
```

Components: Clients, Proxies, Servers

- Clients
  - Send requests and receive responses
  - Browsers, spiders, and agents
- Servers
  - Receive requests and send responses
  - Store or generate the responses
- Proxies (see "HTTP Proxy" assignment!)
  - Act as a server for the client, and a client to the server
  - Perform extra functions such as anonymization, logging, transcoding, blocking of access, caching, etc.

Example Client: Web Browser

- Generating HTTP requests
  - User types URL, clicks a hyperlink, or selects bookmark
  - User clicks "reload", or "submit" on a Web page
  - Automatic downloading of embedded images
- Layout of response
  - Parsing HTML and rendering the Web page
  - Invoking helper applications (e.g., Flash, Flash)
- Maintaining a cache
  - Storing recently-viewed objects
  - Checking that cached objects are fresh

Client: Typical Web Transaction

- User clicks on a hyperlink: http://www.cnn.com/index.html
- Browser learns the IP address
  - Invokes `gethostbyname`(www.cnn.com)
  - And gets a return value of 64.236.16.20
- Browser creates socket and connects to server
  - OS selects an ephemeral port for client side
  - Contacts 64.236.16.20 on port 80
- Browser writes the HTTP request into the socket

  GET /index.html HTTP/1.1<CRLF>

  Host: www.cnn.com<CRLF>

In Fact, Try This at a UNIX Prompt…

```
labpc$ telnet www.cnn.com 80
GET /index.html HTTP/1.1
Host: www.cnn.com
<CRLF>
```

And you'll see the response…

Client: Typical Web Transaction (Cont)

- Browser parses the HTTP response message
  - Extract the URL for each embedded image
  - Create new sockets and send new requests
  - Render the Web page, including the images

- Opportunities for caching in the browser
  - HTML file
  - Each embedded image
  - IP address of the Web site

Web Server

- Website vs. Webserver
  - Website: collections of Web pages associated with a particular host name
  - Webserver: program that satisfies client requests for Web resources
- Handling a client request
  - Accept the socket
  - Read and parse the HTTP request message
  - Translate the URL to a filename (object)
  - Determine whether the request is authorized
  - Generate and transmit the response

Conclusions

- 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
- HyperText Transfer Protocol (HTTP)
  - Client-server protocol
  - URL, HTML, and HTTP
- Next Monday: IP packet switching!