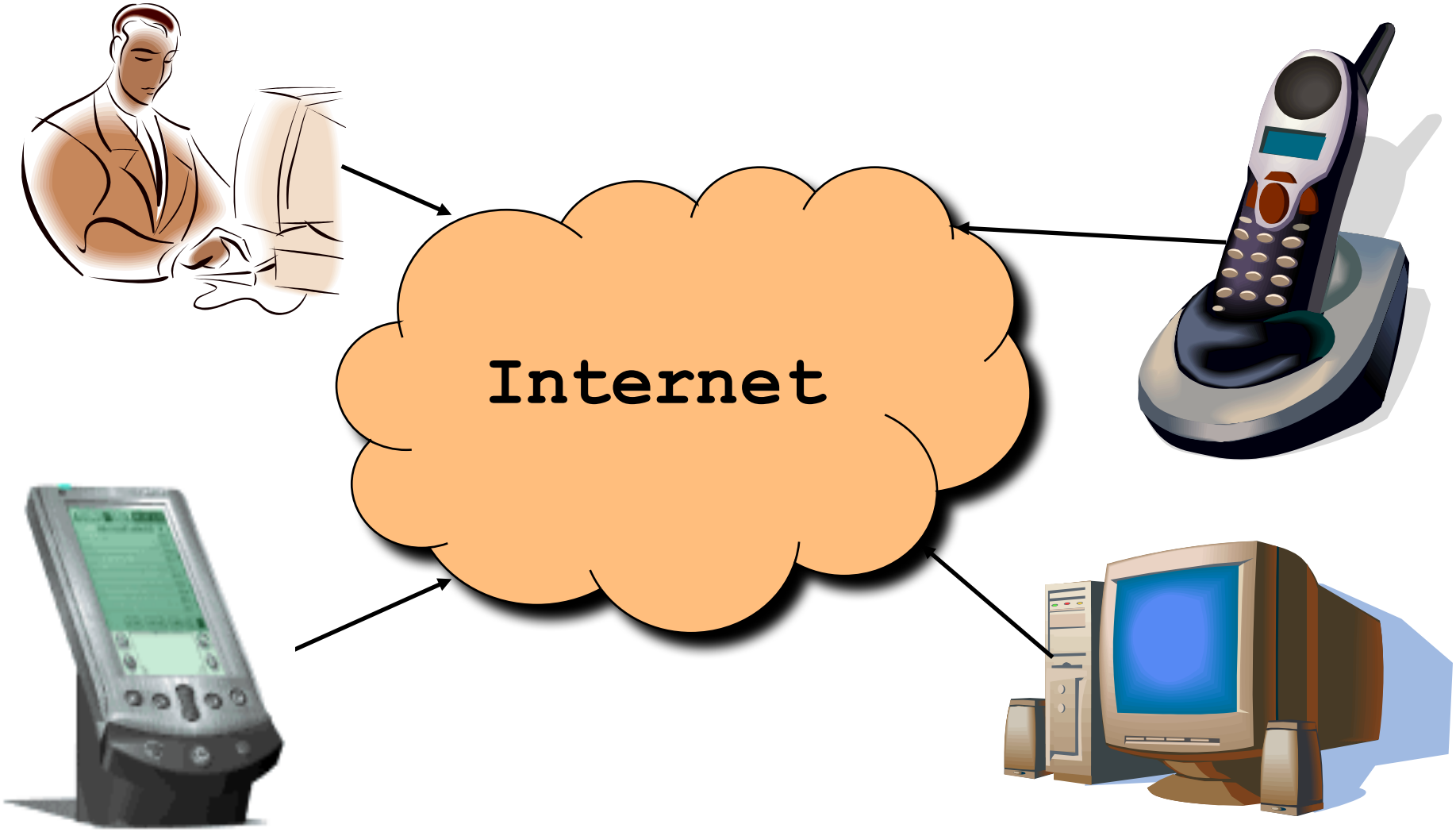
UNIVERSITY OF TORONTO

Socket Programming

Outline

- Client-server paradigm
- Sockets
  - Socket programming in UNIX

End System: Computer on the Net



Also known as a "host"…

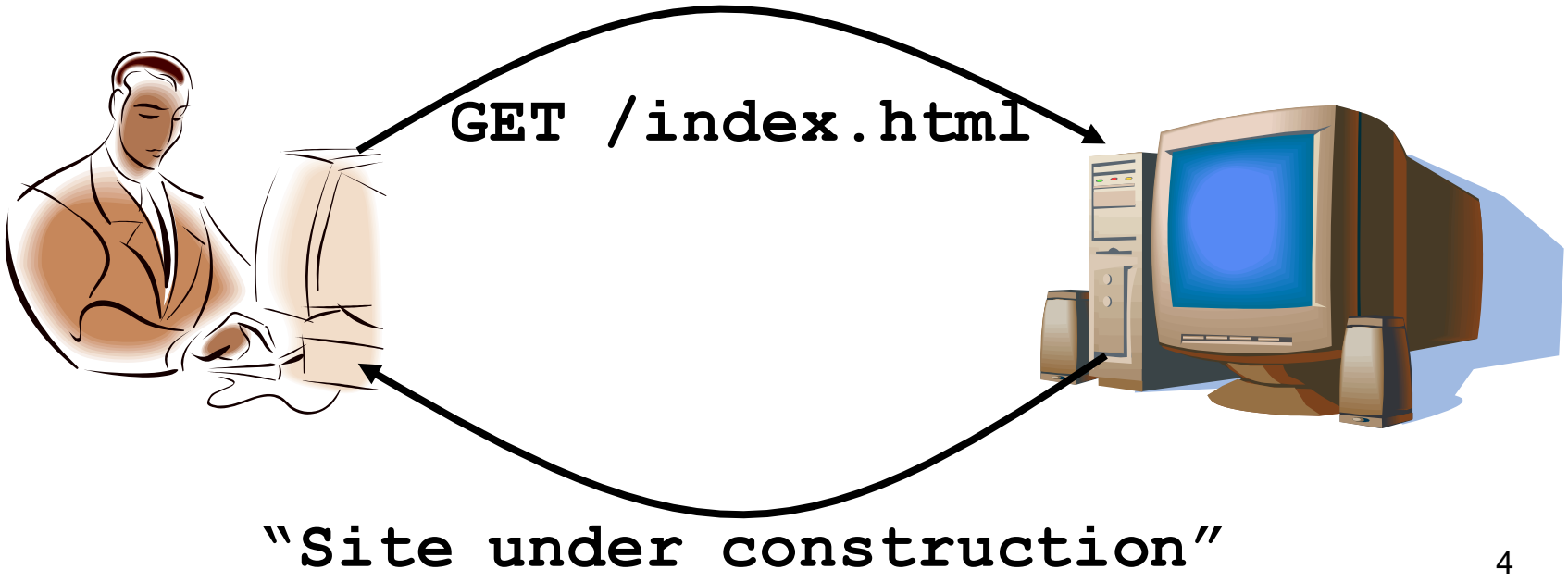
Clients and Servers

Client program

- Running on end host
- Requests service
- E.g., Web browser

Server program

- Running on end host
- Provides service
- E.g., Web server



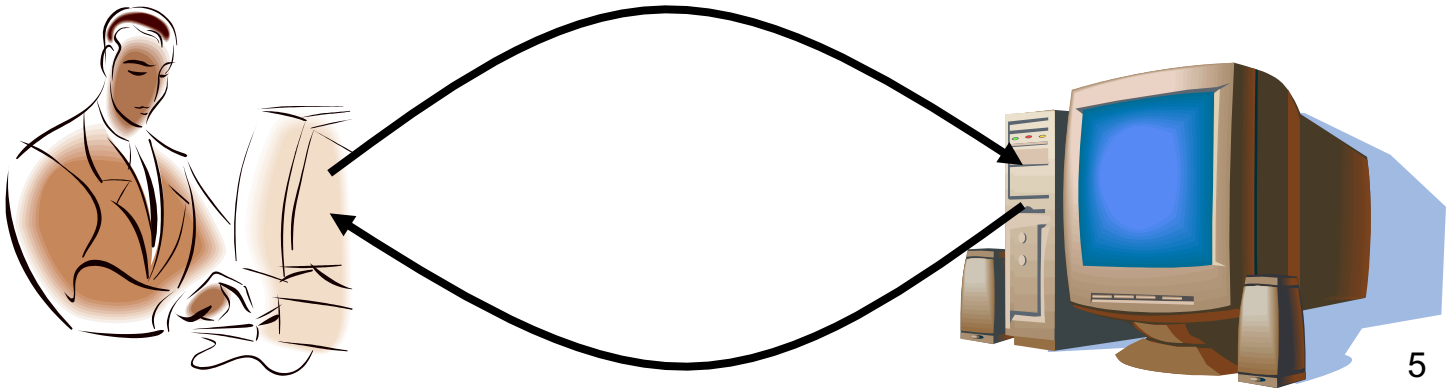
Client-Server Communication

Client

- Sometimes on
- Initiates a request to the server when interested
- E.g., web browser
- Needs to know the server's address

Server

- Always on
- Serve services to many clients
- E.g.,www.cnn.com
- Not initiate contact with the clients
- Needs a fixed address



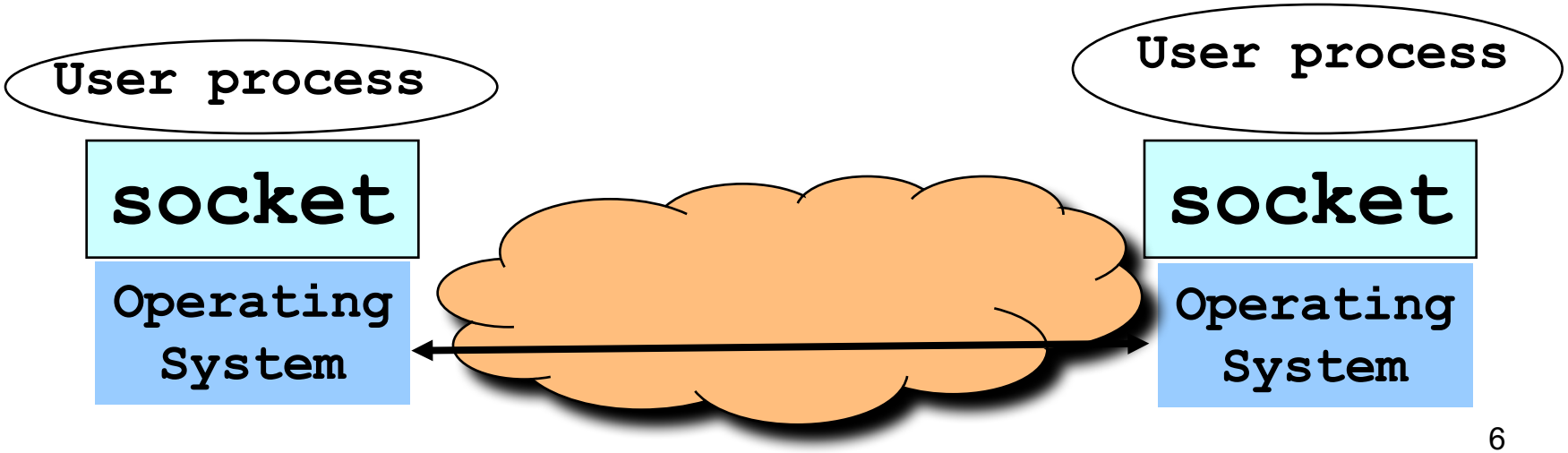
Socket: End Point of Communication

Processes send messages to one another

- Message traverse the underlying network

A Process sends and receives through a “socket”

- Analogy: the doorway of the house.
- Socket, as an API, supports the creation of network applications



UNIX Socket API

Socket interface

- A collection of system calls to write a networking program at user-level.
- Originally provided in Berkeley UNIX
- Later adopted by all popular operating systems

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
- Data written into socket on one host can be read out of socket on other host

System calls for sockets

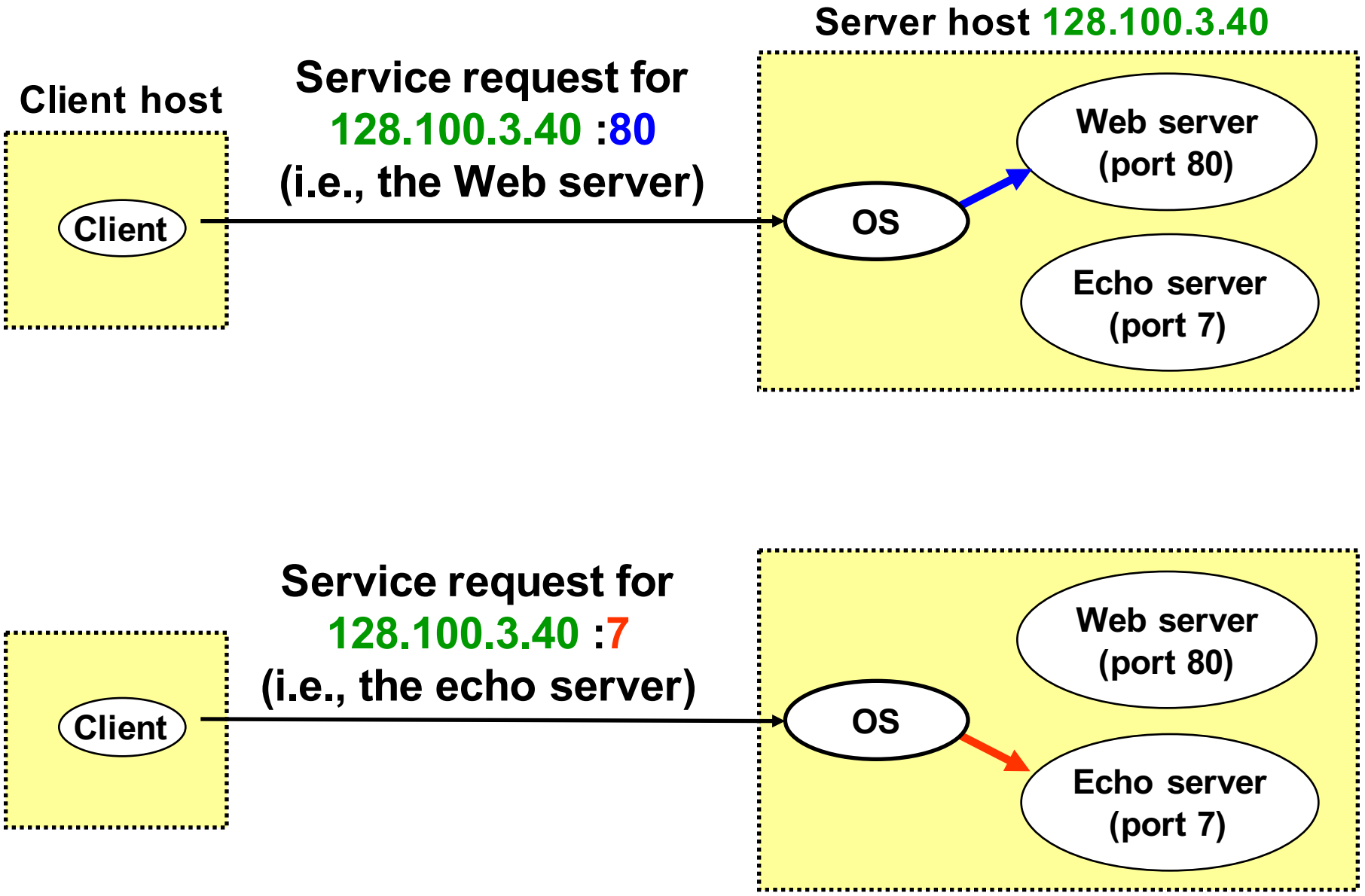
- Client: create, connect, write, read, close
- Server: create, bind, listen, accept, read, write, close

Typical Client Program

Prepare to communicate

- Create a socket
- Determine server address and port number
- Why do we need to have port number?

Using Ports to Identify Services



Socket Parameters

A socket connection has 5 general parameters:

- The protocol
  - Example: TCP and UDP.
- The local and remote address
  - Example: 128.100.3.40
- The local and remote port number
  - Some ports are reserved (e.g., 80 for HTTP)
  - Root access require to listen on port numbers below 1024

More on Prots

- Well-known ports: 0 – 1023
  - 80 => http 21 => ftp
  - 22 => ssh 25 => smtp (mail)
  - 23 => telnet 194 => irc
- Registered ports: 1024 – 49151
  - 2709 = supermon
  - 26000 = quake
  - 3724 = world of warcraft
- Dynamic (private) ports: 49152 – 65535
  - You should pick ports in this reange to avoid overlap

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

Important Functions for Client Program

- socket()

  create the socket descriptor
- connect()

  connect to the remote server
- read(),write()

  communicate with the server
- close()

  end communication by closing socket descriptor

Creating a Socket

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

- Returns a descriptor (or handle) for the socket
- Domain: protocol family
  - AF_INET for the Internet
- Type: semantics of the communication
  - SOCK_STREAM: Connection oriented
  - SOCK_DGRAM: Connectionless
- Protocol: specific protocol
  - UNSPEC: unspecified
  - (AF_INET and SOCK_STREAM already implies TCP)
- **E.g., TCP: sd = socket(AF_INET, SOCK_STREAM, 0);**
- **E.g., UDP: sd = socket(AF_INET, SOCK_DGRAM, 0);**

Connecting to the Server

- *int connect(int sockfd, struct sockaddr *server_address, socketlen_t addrlen)*
  - Arguments: socket descriptor, server address, and address size
  - Remote address and port are in struct sockaddr
  - Returns 0 on success, and -1 if an error occurs

Sending and Receiving Data

Sending data

- *write(int sockfd, void *buf, size_t len)*
  - Arguments: socket descriptor, pointer to buffer of data, and length of the buffer
  - Returns the number of characters written, and -1 on error

Receiving data

- *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

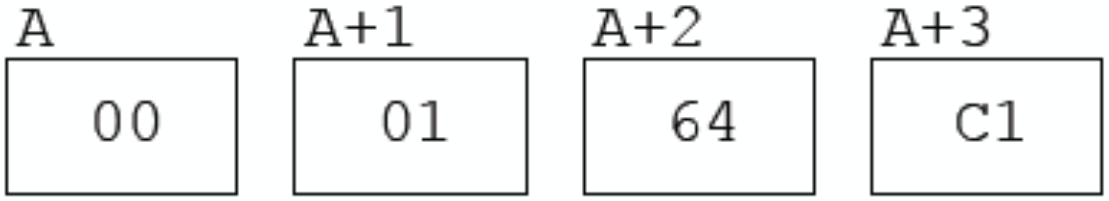
Closing the socket

- *int close(int sockfd)*

Byte Order

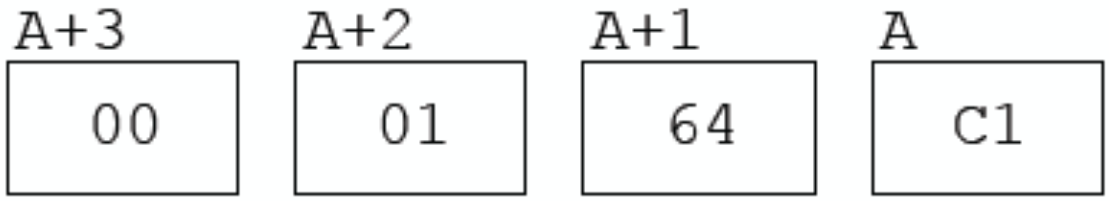
- Big-endian

91,329 =



- Little-endian

91,329 =



- Intel is little-endian, and Sparc is big-endian

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

IP 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

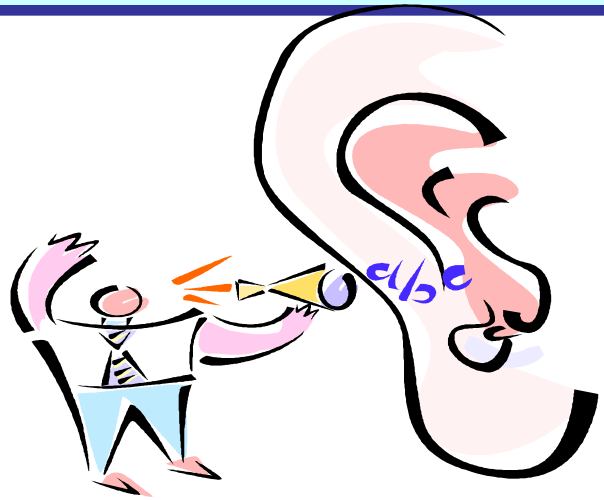
Network Newline

`\r\n` rather than just `\n`

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
- Send data to the socket
- Close the socket

Repeat with the next connection request

Important Functions for Server Program

- socket()
  create the socket descriptor
- bind()
  associate the local address
- listen()
  wait for incoming connections from clients
- accept()
  accept incoming connection
- read(),write()
  communicate with client
- close()
  close the socket descriptor

Socket Preparation for Server Program

Bind socket to the local address and port

- *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 the number of pending connections

- *int listen(int sockfd, int backlog)*
- Arguments: socket descriptor and acceptable backlog
- Returns 0 on success, and -1 on error

Accepting a New Connection

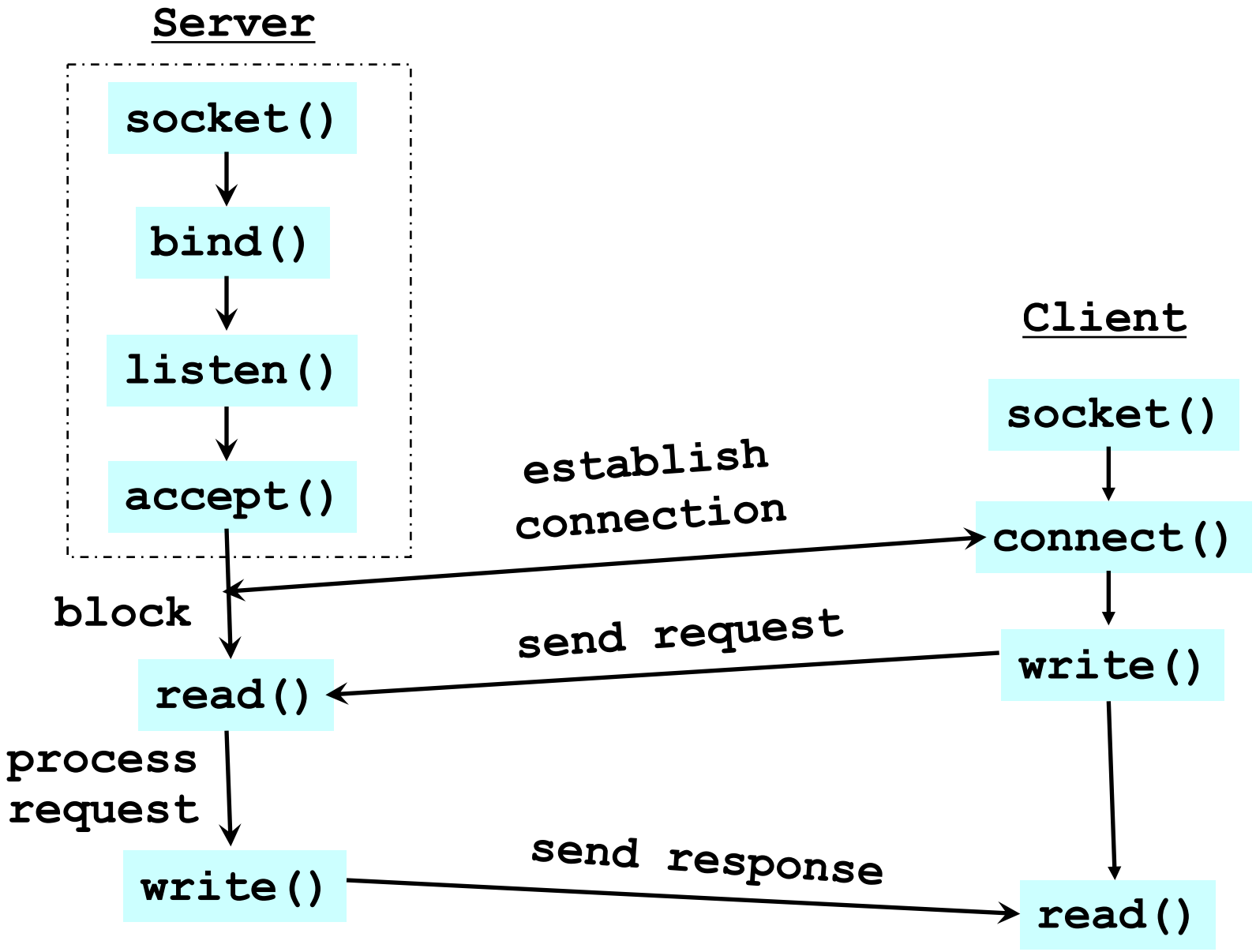
*int accept(int sockfd, struct sockaddr *addr, socketlen_t *addrlen)*

- 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
- 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

Server Operation

- **accept()** returns a new socket descriptor as output
- New socket should be closed when done with communication
- Initial socket remains open, can still accept more connections

Putting it All Together



Supporting Function Calls

gethostbyname() get address for given host name (e.g. 128.100.3.40 for name “cs.toronto.edu”);

getservbyname() get port and protocol for a given service e.g. ftp, http (e.g. “http” is port 80, TCP)

getsockname() get local address and local port of a socket

getpeername() get remote address and remote port of a socket

Useful Structures

```
struct sockaddr {
    u_short sa_family;
    char sa_data[14];
};
struct sockaddr_in {
    u_short sa_family;
    u_short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
};
struct in_addr {
    u_long s_addr;
};
```

Generic address,
"connect(), bind(), accept()"
<sys/socket.h>

Client and server addresses
TCP/UDP address
(includes port #)
<netinet/in.h>

IP address
<netinet/in.h>

Other useful stuff…

- Address conversion routines

– Convert between system's representation of IP addresses and readable strings (e.g. "128.100.3.40 ")

unsigned long inet_addr(char* str);

char * inet_ntoa(struct in_addr inaddr);

- Important header files:

<sys/types.h>, <sys/socket.h>, <netinet/in.h>, <arpa/inet.h>

- man pages
  - **socket, accept, bind, listen**

Helpful Tips

- Think carefully about the exact bytes you are sending and how the receiver will interpret them
- Pay attention to ends of strings and extra characters in char arays
- Byte order & network newlines
- Don’t assume full lines will arrive in a single read

Socket types

Stream Sockets: Delivery in a networked environment is guaranteed. If you send through the stream socket three items "A, B, C", they will arrive in the same order - "A, B, C". These sockets use TCP (Transmission Control Protocol) for data transmission. If delivery is impossible, the sender receives an error indicator. Data records do not have any boundaries.

Datagram Sockets: Delivery in a networked environment is not guaranteed. They're connectionless because you don't need to have an open connection as in Stream Sockets - you build a packet with the destination information and send it out. They use UDP (User Datagram Protocol).

Raw Sockets: These provide users access to the underlying communication protocols, which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more cryptic facilities of an existing protocol.

Sequenced Packet Sockets: They are similar to a stream socket, with the exception that record boundaries are preserved. This interface is provided only as a part of the Network Systems (NS) socket abstraction, and is very important in most serious NS applications. Sequenced-packet sockets allow the user to manipulate the Sequence Packet Protocol (SPP) or Internet Datagram Protocol (IDP) headers on a packet or a group of packets, either by writing a prototype header along with whatever data is to be sent, or by specifying a default header to be used with all outgoing data, and allows the user to receive the headers on incoming packets.

Learning More …

We are only talking about connection-oriented sockets in CSC209

If you want to know about other sockets, read chapters 56-61 in Kerrisk