Lecture 2: Introduction to Unix Network Programming
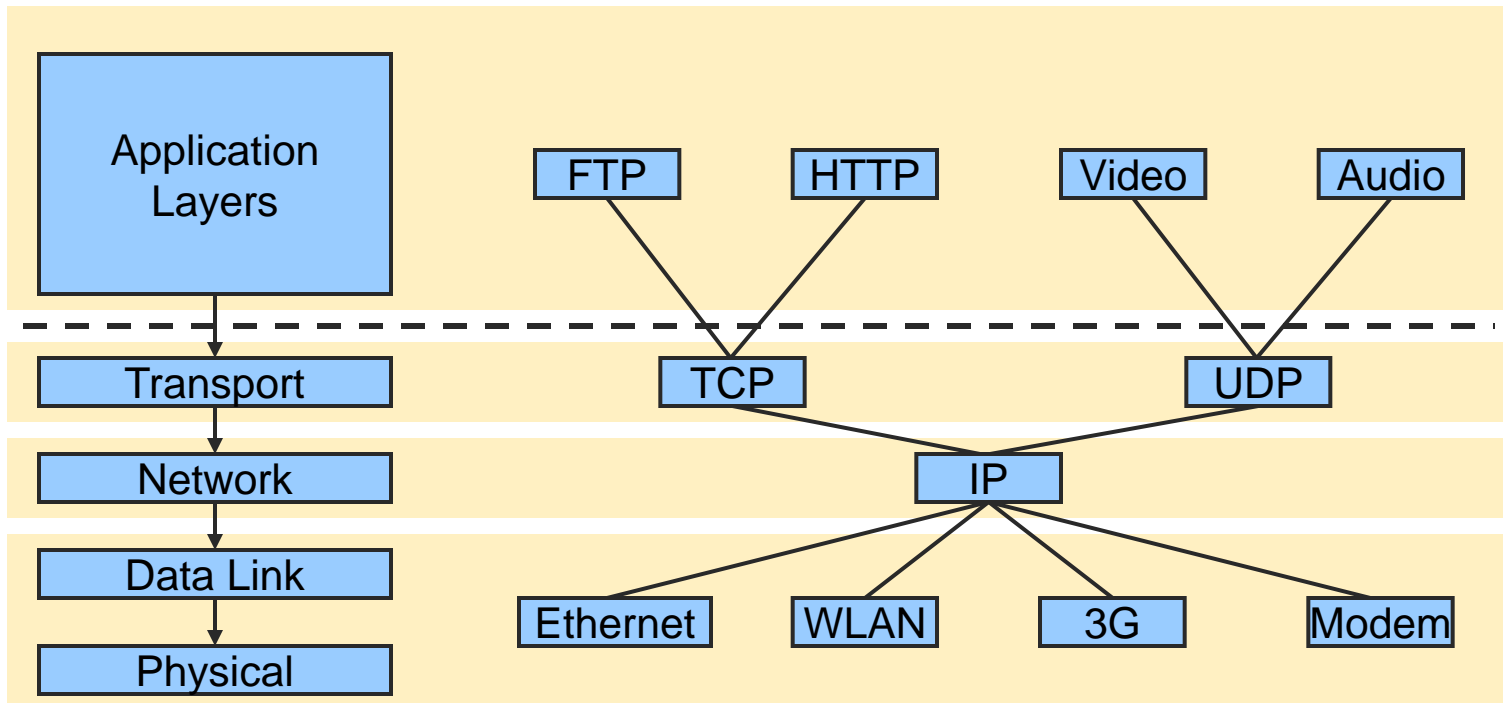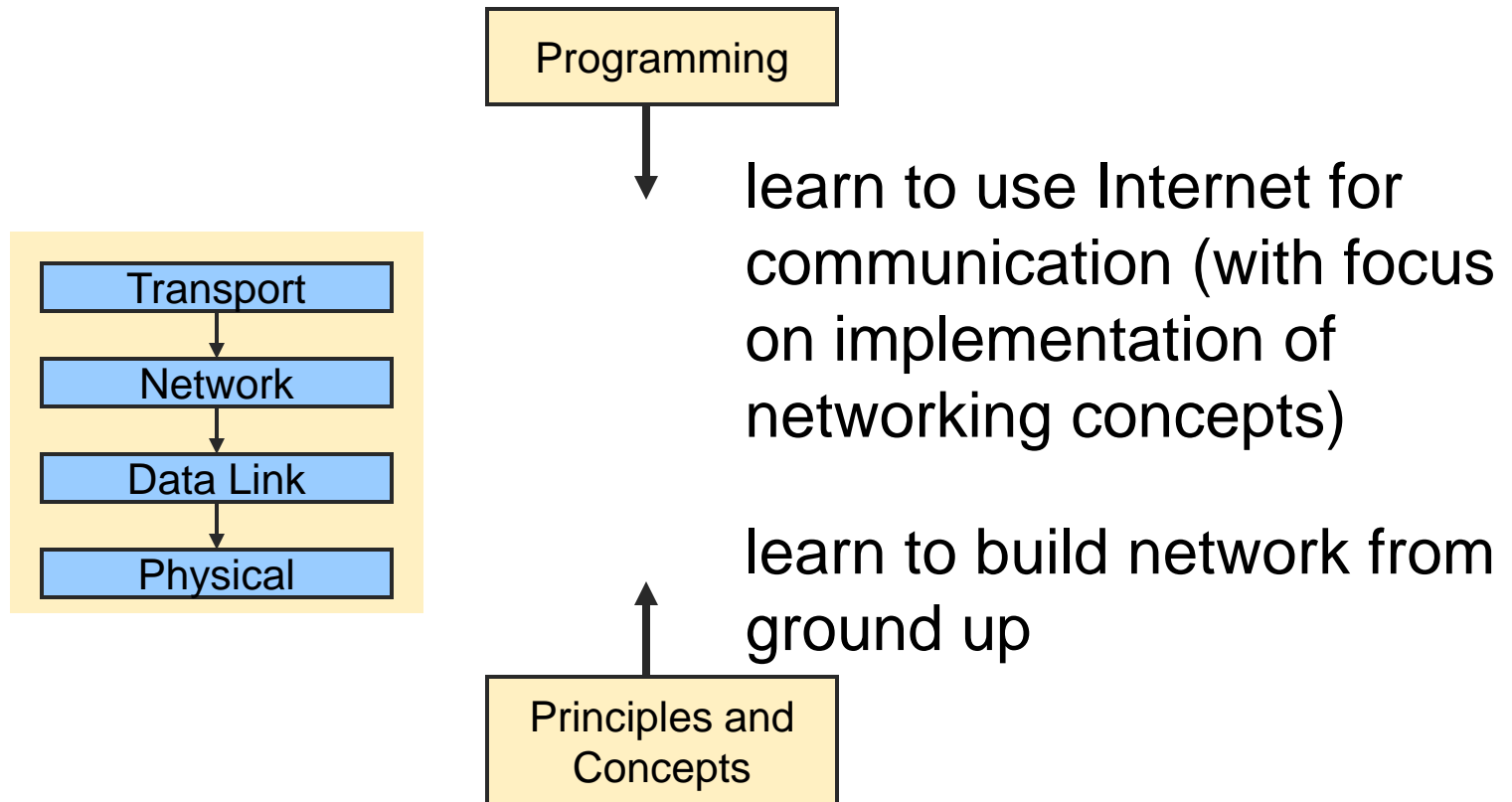
Reference: Stevens Unix Network Programming

Internet Protocols

Direction and Principles

Programming

⬇

learn to use Internet for communication (with focus on implementation of networking concepts)

Transport

⬇

Network

⬇

Data Link

⬇

Physical

learn to build network from ground up

Principles and Concepts

Network Programming

- How should two hosts communicate with each other over the Internet?
    - The "Internet Protocol" (IP)
    - Transport protocols: TCP, UDP

- How should programmers interact with the protocols?
    - Sockets API – application programming interface
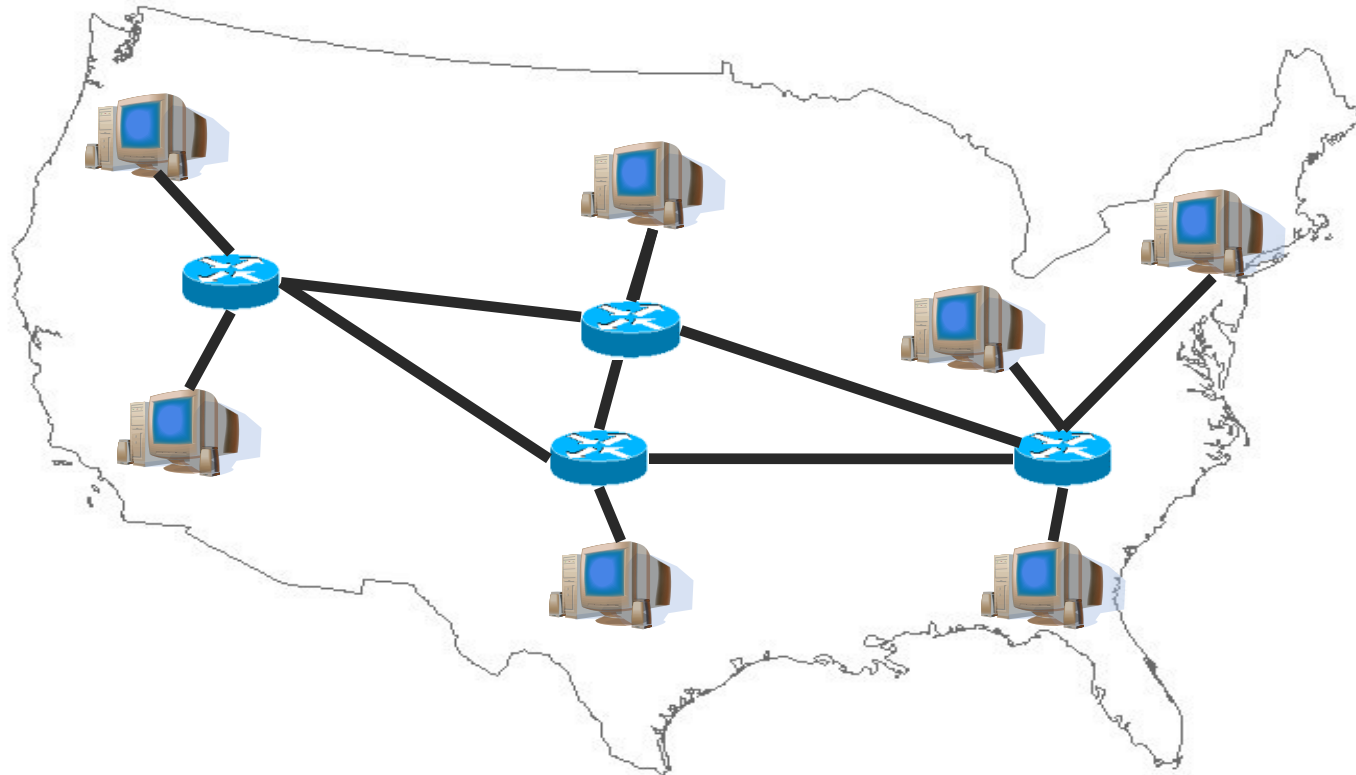   - De facto standard for network programming

Network Programming with Sockets

- ■ Sockets API
  - ○ An interface to the transport layer
    - ■ Introduced in 1981 by BSD 4.1
    - ■ Implemented as library and/or system calls
    - ■ Similar interfaces to TCP and UDP
    - ■ Can also serve as interface to IP (for super-user); known as "raw sockets"

How can many hosts communicate?

- Multiplex traffic with routers
- Question: How to identify the destination?
- Question: How to share bandwidth across different flows?

Identifying hosts with Addresses and Names

- **IP addresses**
  - Easily handled by routers/computers
  - Fixed length
  - E.g.: `128.121.146.100`

- **But how do you know the IP address?**
  - Internet domain names
  - Human readable, variable length
  - E.g.: `twitter.com`

- **But how do you get the IP address from the domain name?**
  - Domain Name System (DNS) maps between them

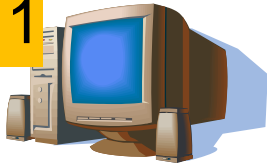# How can many hosts share network resources?
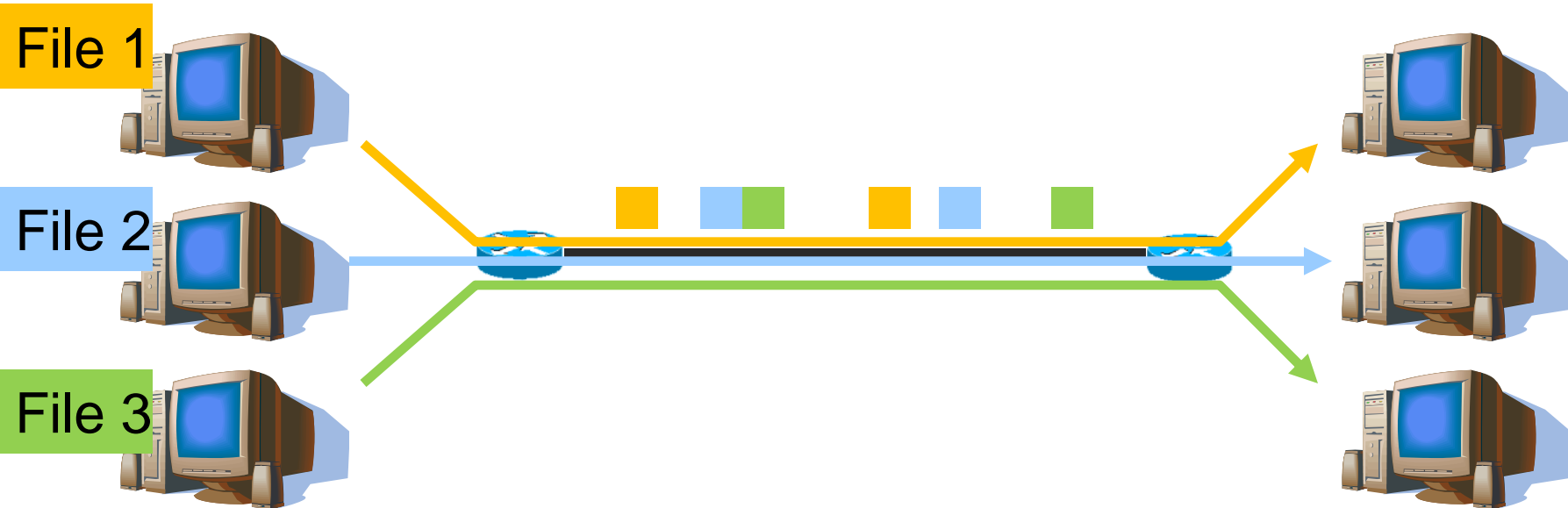
- Solution: divide traffic into "IP packets"
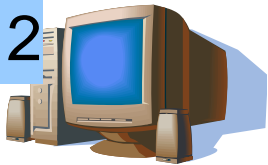 - At each router, the entire packet is received, stored, and then forwarded to the next router

How can many hosts share network resources?

File 1

File 2

File 3

header	data
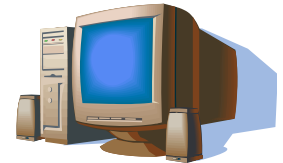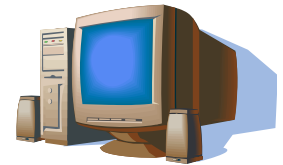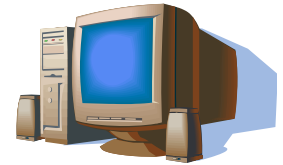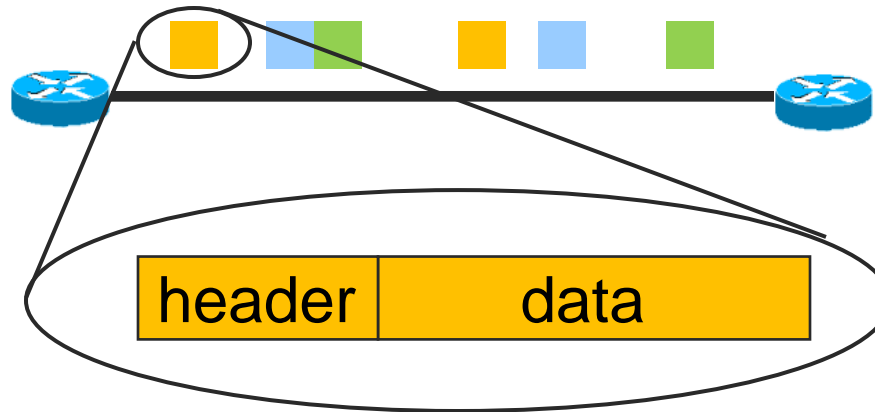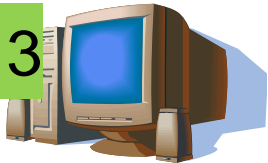
- Solution: divide traffic into "IP packets"
  - Use packet "headers" to denote which connection the packet belongs to
    - Contains src/dst address/port, length, checksum, time-to-live, protocol, flags, type-of-service, etc

Is IP enough?

- What if host runs multiple applications?
  - Use UDP: 16-bit "Port numbers" in header distinguishes traffic from different applications
- Or if content gets corrupted?
  - Use UDP: "Checksum" covering data, UDP header, and IP header detects flipped bits
- User Datagram Protocol (UDP)
  - Properties
    - Unreliable - no guaranteed delivery
    - Unordered - no guarantee of maintained order of delivery
    - Unlimited Transmission - no flow control
  - Unit of Transfer is "datagram" (a variable length packet)

Is UDP enough?

- What if network gets congested? Or packets get lost/reordered/duplicated?

- UseTransport Control Protocol (TCP)
  - Guarantees reliability, ordering, and integrity
  - Backs off when there is congestion
  - Connection-oriented (Set up connection before communicating, Tear down connection when done)
  - Gives 'byte-stream" abstraction to application
  - Also has ports, but different namespace from UDP

- Which one is better, TCP or UDP?

- Why not other hybrid design points?

How should we program networked apps?

- **How can we compose together programs running on different machines?**
  - Client-server model

- **What sort of interfaces should we reveal to the programmer?**
  - Sockets API

Client-Server Model

- A client initiates a request to a well-known server
- Example: the web

"GET index.html"
(request for web page)

"HTTP/1.0 200 OK…"
(response, including web page)

Client

Web server
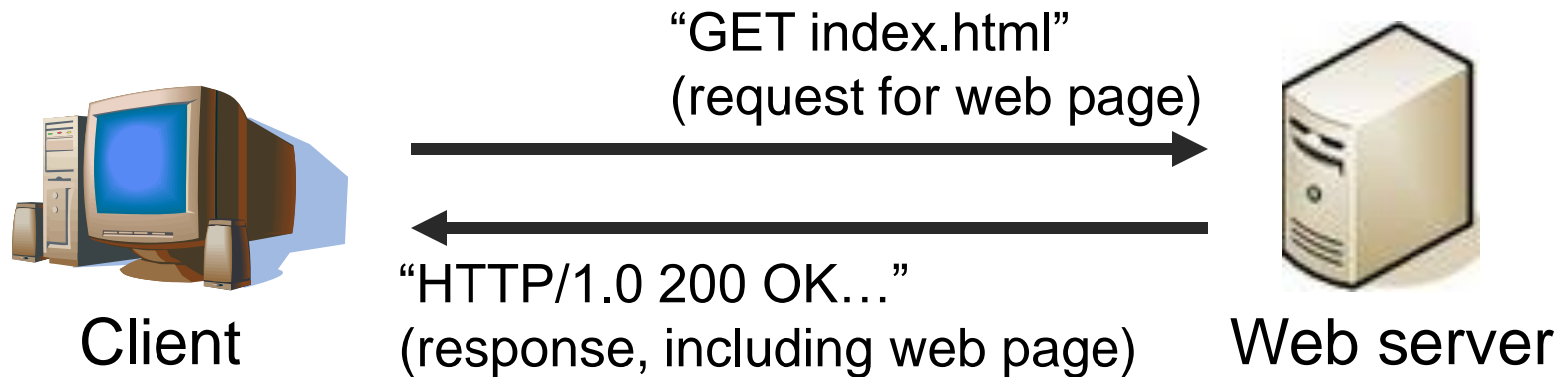
- Other examples: FTP, SSH/Telnet, SMTP (email), Print servers, File servers

Client-Server Model

- Server
  - Client
  - Client
  - Client
  - Client

Can you think of any network apps that are not client/server?
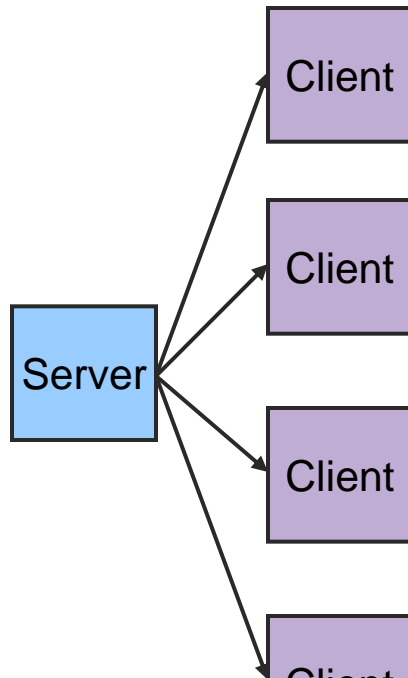
- **Asymmetric Communication**
  - Client sends requests
  - Server sends replies
- **Server/Daemon**
  - Well-known name and port
  - Waits for contact
  - Processes requests, sends replies
- **Client**
  - Initiates contact
  - Waits for response

Server-side service models

- **Concurrent**
  - Server processes multiple clients' requests simultaneously
- **Sequential**
  - Server processes only one client's requests at a time
- **Hybrid**
  - Server maintains multiple connections, but processes responses sequentially
- **Which one is best?**

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/**

What interfaces to expose to programmer?

- Stream vs. Datagram sockets
- Stream sockets
  - Abstraction: send a long stream of characters
  - Typically implemented on top of TCP
- Datagram sockets
  - Abstraction: send a single packet
  - Typically implemented on top of UDP

Stream sockets

send("This is a long sequence of text I would like to send to the other host")

"This is a long sequence of text I would like to send to the other host"=**recv**(socket)

Sockets API

Sockets API

"This is a long"
"sequence of text"
"I would like to send"
"to the other host"
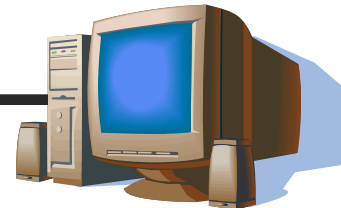
Datagram sockets

sendto("This is a long")
sendto("sequence of text")
sendto("I would like to send")
sendto("to the other host")
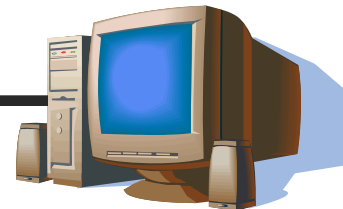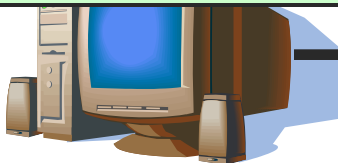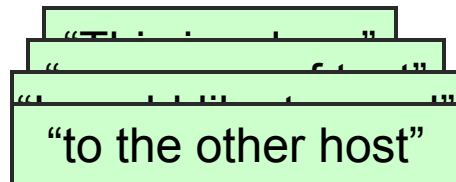
"This is a long"=**recvfrom**(socket)
"sequence of text"=**recvfrom**(socket)
"I would like to send"=**recvfrom**(socket)
"to the other host"=**recvfrom**(socket)

Sockets API	Sockets API

"This is a long"
"sequence of text"
"I would like to send"
"to the other host"

What specific functions to expose?

- Data structures to store information about connections and hosts

Socket Address Structure

- IP address:
```
struct in_addr {
    in_addr_t s_addr;        /* 32-bit IP address */
};
```

- TCP or UDP address:
```
struct sockaddr_in {
    short sin_family;        /* e.g., AF_INET */
    ushort sin_port;         /* TCP/UDP port */
    struct in_addr;          /* IP address */
};
```

Structure: **addrinfo**

- The **addrinfo** data structure (from **/usr/include/netdb.h**)
  - ○ Canonical domain name and aliases
  - ○ List of addresses associated with machine
  - ○ Also address type and length information

```
int ai_flags              Input flags
int ai_family             Address family of socket
int ai_socktype           Socket type
int ai_protocol           Protocol of socket
socklen_t ai_addrlen      Length of socket address
struct sockaddr *ai_addr  Socket address of socket
char *ai_canonname        Canonical name of service location
struct addrinfo *ai_next  Pointer to next in list
```

Address Access/Conversion Functions

```
#include <sys/types.h>

#include <sys/socket.h>

#include <netdb.h>

int getaddrinfo(const char *restrict node,
          const char *restrict service,
          const struct addrinfo *restrict hints,
          struct addrinfo **restrict res);
```

- Parameters
  - **node**: host name or IP address to connect to
  - **service**: a port number ("80") or the name of a service (found /etc/services: "http")
  - **hints**: a filled out struct addrinfo

Example: `getaddrinfo`

```
int status;
struct addrinfo hints;
struct addrinfo *servinfo;              // pointer to results

memset(&hints, 0, sizeof hints);        // empty struct
hints.ai_family = AF_UNSPEC;            // don't care IPv4/IPv6
hints.ai_socktype = SOCK_STREAM;        // TCP stream sockets

// get ready to connect
status = getaddrinfo("www.example.net", "3490", &hints,
    &servinfo);

// servinfo now points to a linked list of 1 or more struct
    addrinfos
```

What specific functions to expose?

- Data structures to store information about connections and hosts
- Functions to create a socket

Function: `socket`

`int socket (int family, int type, int protocol);`

- Create a socket.
  - Returns file descriptor or -1. Also sets **`errno`** on failure.
  - **`family`**: address family (namespace)
    - **`AF_INET`** for IPv4
    - other possibilities: **`AF_INET6`** (IPv6), **`AF_UNIX`** or **`AF_LOCAL`** (Unix socket), **`AF_ROUTE`** (routing)
  - **`type`**: style of communication
    - **`SOCK_STREAM`** for TCP (with **`AF_INET`**)
    - **`SOCK_DGRAM`** for UDP (with **`AF_INET`**)
  - **`protocol`**: protocol within family
    - typically 0

Example: `socket`

```
int sockfd, new_fd; /* listen on sock_fd, new
                             connection on new_fd */
struct sockaddr_in my_addr;        /* my address      */
struct sockaddr_in their_addr; /* connector addr */
int sin_size;

if ((sockfd = socket(AF_INET, SOCK_STREAM, 0))==-1){
   perror("socket");
   exit(1);
}
```

Example: **socket**

```
int sockfd, new_fd; /* listen on sock_fd, new
                        connection on new_fd */
struct sockaddr_in my_addr;      /* my address     */
struct sockaddr_in their_addr; /* connector addr */
int sin_size;

if ((sockfd = socket(AF_INET, SOCK_STREAM, 0))==-1){
   perror("socket");
   exit(1);
}
```

What specific functions to expose?

- Data structures to store information about connections and hosts

- Functions to create a socket

- Functions to establish connections

TCP Connection Setup
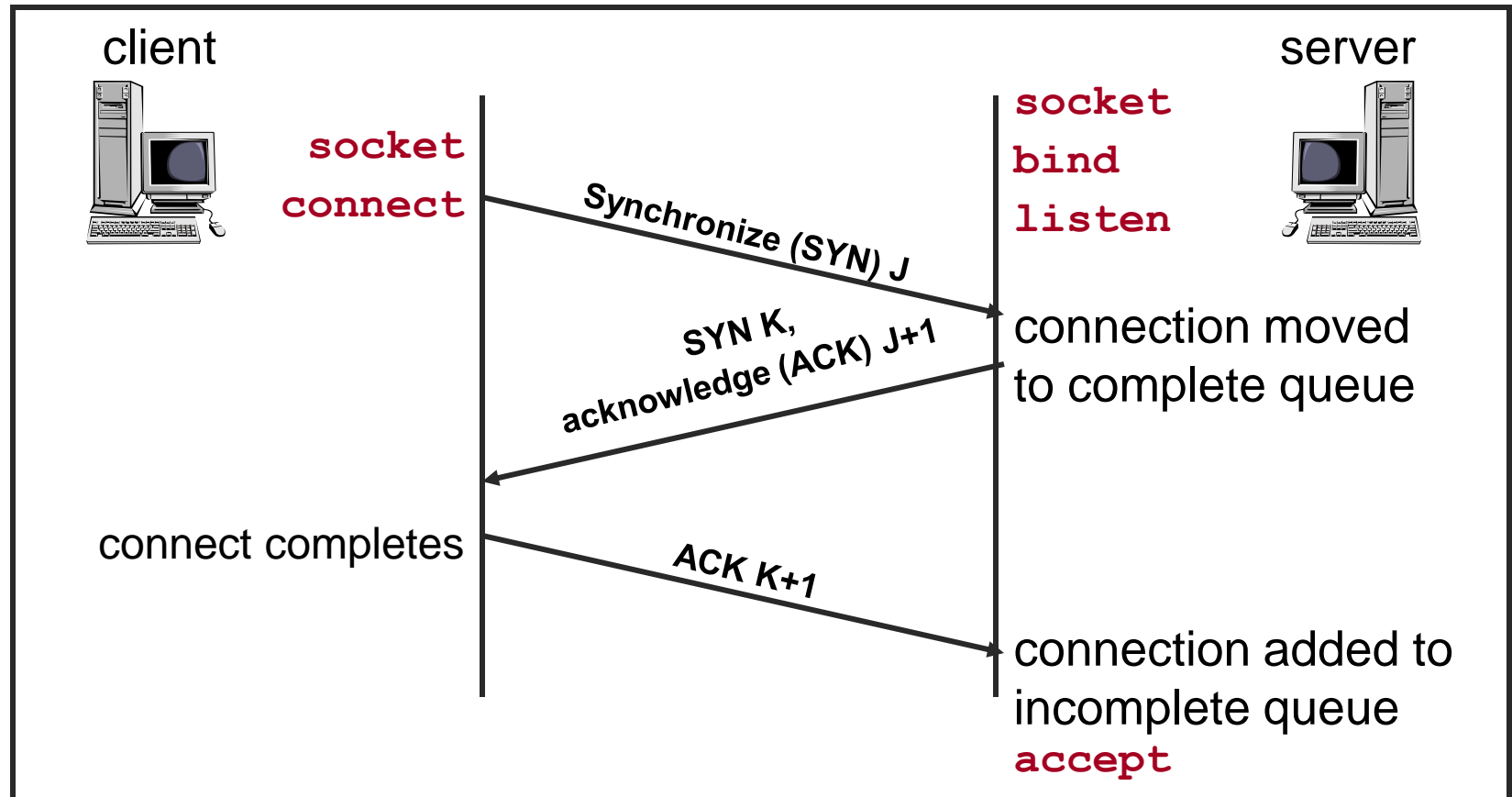
Function: **bind**

int bind (int sockfd, struct sockaddr* myaddr, int addrlen);

- Bind a socket to a local IP address and port number
  - Returns 0 on success, -1 and sets **errno** on failure
  - **sockfd**: socket file descriptor (returned from **socket**)
  - **myaddr**: includes IP address and port number
    - IP address: set by kernel if value passed is **INADDR_ANY**, else set by caller
    - port number: set by kernel if value passed is 0, else set by caller
  - **addrlen**: length of address structure
    - **= sizeof (struct sockaddr_in)**

TCP and UDP Ports

- Allocated and assigned by the Internet Assigned Numbers Authority
  - see **RFC 1700** (for historical purposes only)

1-512	■ standard services (see **/etc/services**) ■ super-user only
513-1023	■ registered and controlled, also used for identity verification ■ super-user only
1024-49151	■ registered services/ephemeral ports
49152-65535	■ private/ephemeral ports

Reserved Ports

Keyword	Decimal	Description	Keyword	Decimal	Description
	0/tcp	Reserved	time	37/tcp	Time
	0/udp	Reserved	time	37/udp	Time
tcpmux	1/tcp	TCP Port Service	name	42/tcp	Host Name Server
tcpmux	1/udp	TCP Port Service	name	42/udp	Host Name Server
echo	7/tcp	Echo	nameserver	42/tcp	Host Name Server
echo	7/udp	Echo	nameserver	42/udp	Host Name Server
systat	11/tcp	Active Users	nicname	43/tcp	Who Is
systat	11/udp	Active Users	nicname	43/udp	Who Is
daytime	13/tcp	Daytime (RFC 867)	domain	53/tcp	Domain Name Server
daytime	13/udp	Daytime (RFC 867)	domain	53/udp	Domain Name Server
qotd	17/tcp	Quote of the Day	whois++	63/tcp	whois++
qotd	17/udp	Quote of the Day	whois++	63/udp	whois++
chargen	19/tcp	Character Generator	gopher	70/tcp	Gopher
chargen	19/udp	Character Generator	gopher	70/udp	Gopher
ftp-data	20/tcp	File Transfer Data	finger	79/tcp	Finger
ftp-data	20/udp	File Transfer Data	finger	79/udp	Finger
ftp	21/tcp	File Transfer Ctl	http	80/tcp	World Wide Web HTTP
ftp	21/udp	File Transfer Ctl	http	80/udp	World Wide Web HTTP
ssh	22/tcp	SSH Remote Login	www	80/tcp	World Wide Web HTTP
ssh	22/udp	SSH Remote Login	www	80/udp	World Wide Web HTTP
telnet	23/tcp	Telnet	www-http	80/tcp	World Wide Web HTTP
telnet	23/udp	Telnet	www-http	80/udp	World Wide Web HTTP
smtp	25/tcp	Simple Mail Transfer	kerberos	88/tcp	Kerberos
smtp	25/udp	Simple Mail Transfer	kerberos	88/udp	Kerberos

Function: **listen**

int listen (int sockfd, int backlog);

- Put socket into passive state (wait for connections rather than initiate a connection)
  - Returns 0 on success, -1 and sets **errno** on failure
  - **sockfd**: socket file descriptor (returned from **socket**)
  - **backlog**: bound on length of unaccepted connection queue (connection backlog); kernel will cap, thus better to set high
  - Example:
    ```
    if (listen(sockfd, BACKLOG) == -1) {
            perror("listen");
            exit(1);
    }
    ```

Functions: **accept**

int accept (int sockfd, struct sockaddr* cliaddr, int* addrlen);

- Block waiting for a new connection
  - Returns file descriptor or -1 and sets **errno** on failure
  - **sockfd**: socket file descriptor (returned from **socket**)
  - **cliaddr**: IP address and port number of client (returned from call)
  - **addrlen**: length of address structure = pointer to **int** set to **sizeof (struct sockaddr_in)**

- **addrlen** is a **value-result** argument
  - the caller passes the size of the address structure, the kernel returns the size of the client's address (the number of bytes written)

Functions: **accept**

```
sin_size = sizeof(struct sockaddr_in);
if ((new_fd = accept(sockfd, (struct sockaddr*)
                   &their_addr, &sin_size)) == -1) {
    perror("accept");
    continue;
}
```

- How does the server know which client it is?
  - **their_addr.sin_addr** contains the client's IP address
  - **their_addr.port** contains the client's port number

  ```
  printf("server: got connection from %s\n",
         inet_ntoa(their_addr.sin_addr));
  ```

Functions: **accept**

- **Notes**
  - After **accept()** returns a new socket descriptor, I/O can be done using **read()** and **write()**
  - Why does **accept()** need to return a new descriptor?

Example: Server
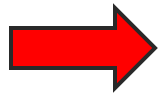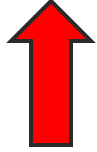
```
my_addr.sin_family = AF_INET; /* host byte order */
my_addr.sin_port = htons(MYPORT); /* short, network
                                     byte order    */
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
/* automatically fill with my IP                   */
bzero(&(my_addr.sin_zero), 8);    /* zero struct */

if (bind(sockfd, (struct sockaddr *)&my_addr,
        sizeof(struct sockaddr)) == -1) {
    perror("bind");
    exit(1);
}
```

Example: Server

```
if (listen(sockfd, BACKLOG) == -1) {
    perror("listen");
    exit(1);
}

while(1) { /* main accept() loop */
    sin_size = sizeof(struct sockaddr_in);
    if ((new_fd = accept(sockfd, (struct sockaddr*)
                    &their_addr,&sin_size)) == -1) {
            perror("accept");
            continue;
    }
    printf("server: got connection from %s\n",
    inet_ntoa(their_addr.sin_addr));
```

Function: **connect**

int connect (int sockfd, struct sockaddr* servaddr, int addrlen);

- Connect to another socket.
  - Returns 0 on success, -1 and sets **errno** on failure
  - **sockfd**: socket file descriptor (returned from **socket**)
  - **servaddr**: IP address and port number of server
  - **addrlen**: length of address structure
    - **= sizeof (struct sockaddr_in)**
- Can use with UDP to restrict incoming datagrams and to obtain asynchronous errors

Example: Client

```
their_addr.sin_family = AF_INET; /* interp'd by host */
their_addr.sin_port = htons (PORT);
their_addr.sin_addr = *((struct in_addr*)he->h_addr);
bzero (&(their_addr.sin_zero), 8);
/* zero rest of struct */
if (connect (sockfd, (struct sockaddr*)&their_addr,
               sizeof (struct sockaddr)) == -1) {
    perror ("connect");
    exit (1);
}
```

What specific functions to expose?

- Data structures to store information about connections and hosts

- Functions to create a socket

- Functions to establish connections

- Functions to send and receive data

TCP Connection Example
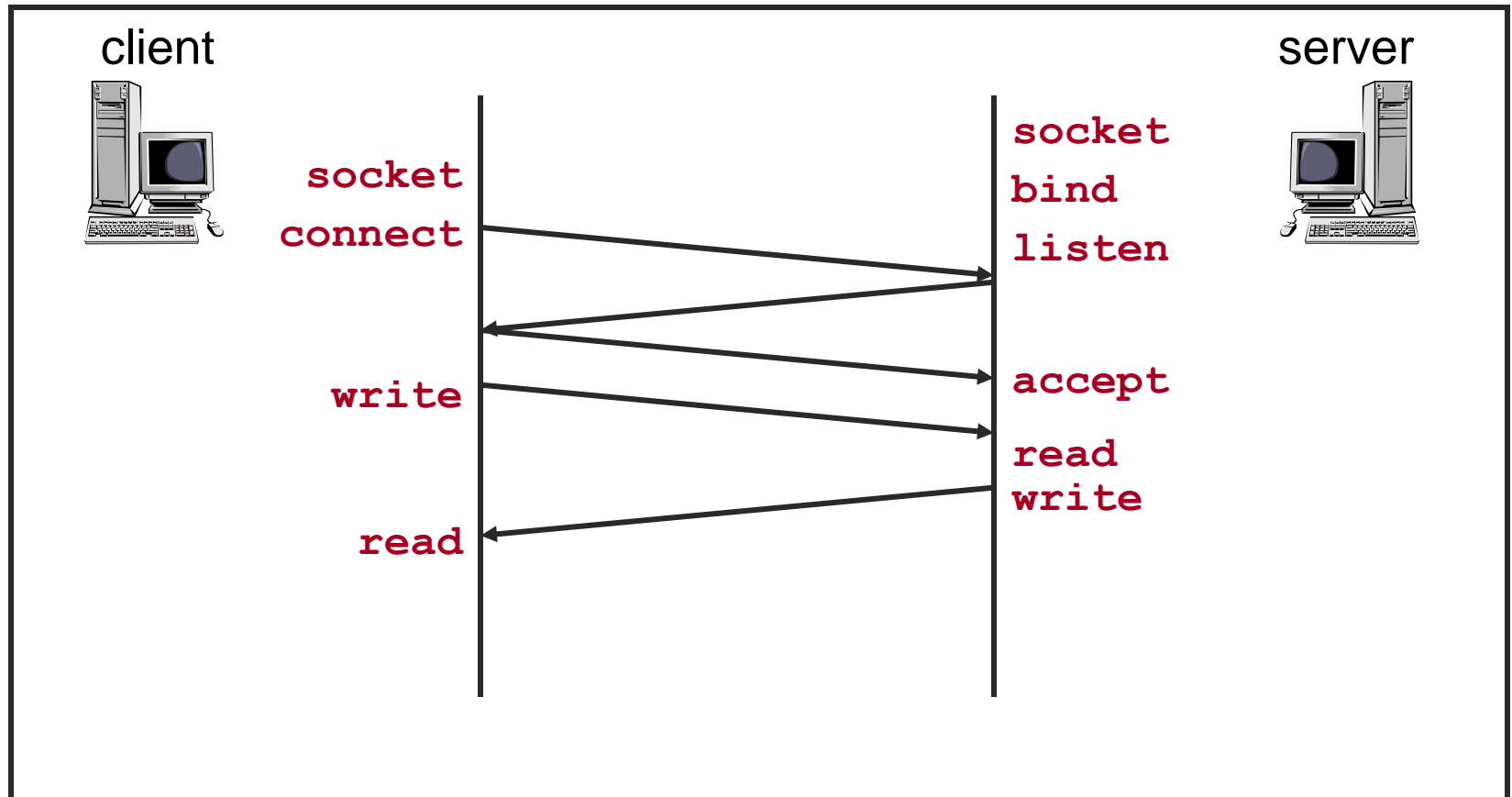
Functions: **write**

int write (int sockfd, char* buf, size_t nbytes);

- Write data to a stream (TCP) or "connected" datagram (UDP) socket
  - Returns number of bytes written or -1 and sets **errno** on failure
  - **sockfd**: socket file descriptor (returned from **socket**)
  - **buf**: data buffer
  - **nbytes**: number of bytes to try to write
  - Example:

    ```
    if((w = write(fd, buf, sizeof(buf))) < 0) {
        perror("write");
        exit(1);
    }
    ```

Functions: `write`

```
int write (int sockfd, char* buf, size_t nbytes);
```

- Notes
  - `write` blocks waiting for data from the client
  - `write` may not write all bytes asked for
    - Does not guarantee that `sizeof(buf)` is written
    - This is not an error
    - Simply continue writing to the device
  - Some reasons for failure or partial writes
    - Process received interrupt or signal
    - Kernel resources unavailable (*e.g.*, buffers)

Example: `writen`

```c
/* Write "n" bytes to a descriptor */
ssize_t writen(int fd, const void *ptr, size_t n) {
    size_t nleft;
    ssize_t nwritten;
    nleft = n;
    while (nleft > 0) {
        if ((nwritten = write(fd, ptr, nleft)) < 0) {
            if (nleft == n)
                return(-1); /* error, return -1 */
            else
                break; /* error, return amount written so far */
        }
        else
            if (nwritten == 0)
                break;
        nleft -= nwritten;
        ptr += nwritten;
    }
    return(n - nleft); /* return >= 0 */
}
```

`write` returned a potential error

0 bytes were written

Update number of bytes left to write and pointer into buffer

Functions: **send**

int send(int sockfd, const void * buf, size_t nbytes, int flags);

- Send data un a stream (TCP) or "connected" datagram (UDP) socket
  - Returns number of bytes written or -1 and sets **errno** on failure
  - **sockfd**: socket file descriptor (returned from **socket**)
  - **buf**: data buffer
  - **nbytes**: number of bytes to try to write
  - **flags**: control flags
    - MSG_PEEK: get data from the beginning of the receive queue without removing that data from the queue
- Example

  len = strlen(msg);

  bytes_sent = send(sockfd, msg, len, 0);

Functions: **read**

int read (int sockfd, char* buf, size_t nbytes);

- Read data from a stream (TCP) or "connected" datagram (UDP) socket
  - Returns number of bytes read or -1, sets **errno** on failure
  - Returns 0 if socket closed
  - **sockfd**: socket file descriptor (returned from **socket**)
  - **buf**: data buffer
  - **nbytes**: number of bytes to try to read
  - Example
    ```
    if((r = read(newfd, buf, sizeof(buf))) < 0) {
        perror("read"); exit(1);
    }
    ```

Functions: **read**

int read (int sockfd, char* buf, size_t nbytes);

- Notes
  - **read** blocks waiting for data from the client
  - **read** may return less than asked for
    - Does not guarantee that **sizeof(buf)** is read
    - This is not an error
    - Simply continue reading from the device

Example: `readn`

```c
/* Read "n" bytes from a descriptor */
ssize_t readn(int fd, void *ptr, size_t n) {
    size_t nleft;
    ssize_t nread;
    nleft = n;
    while (nleft > 0) {
        if ((nread = read(fd, ptr, nleft)) < 0) {
            if (nleft == n)
                return(-1); /* error, return -1 */
            else
                break; /* error, return amt read */
        }
        else
            if (nread == 0)
                break; /* EOF */
        nleft -= nread;
        ptr += nread;
    }
    return(n - nleft); /* return >= 0 */
}
```

read returned a potential error

0 bytes were read

Update number of bytes left to read and pointer into buffer

Functions: **recv**

```
int recv(int sockfd, void *buf, size_t nbytes,
    int flags);
```

- Read data from a stream (TCP) or "connected" datagram (UDP) socket
  - Returns number of bytes read or -1, sets **errno** on failure
  - Returns 0 if socket closed
  - **sockfd**: socket file descriptor (returned from **socket**)
  - **buf**: data buffer
  - **nbytes**: number of bytes to try to read
  - **flags**: see man page for details; typically use 0

Functions: **recv**

int read (int sockfd, char* buf, size_t nbytes);

- Notes
  - **read** blocks waiting for data from the client but does not guarantee that **sizeof(buf)** is read
  - Example
    ```
    if((r = read(newfd, buf, sizeof(buf))) < 0 {
        perror("read"); exit(1);
    }
    ```
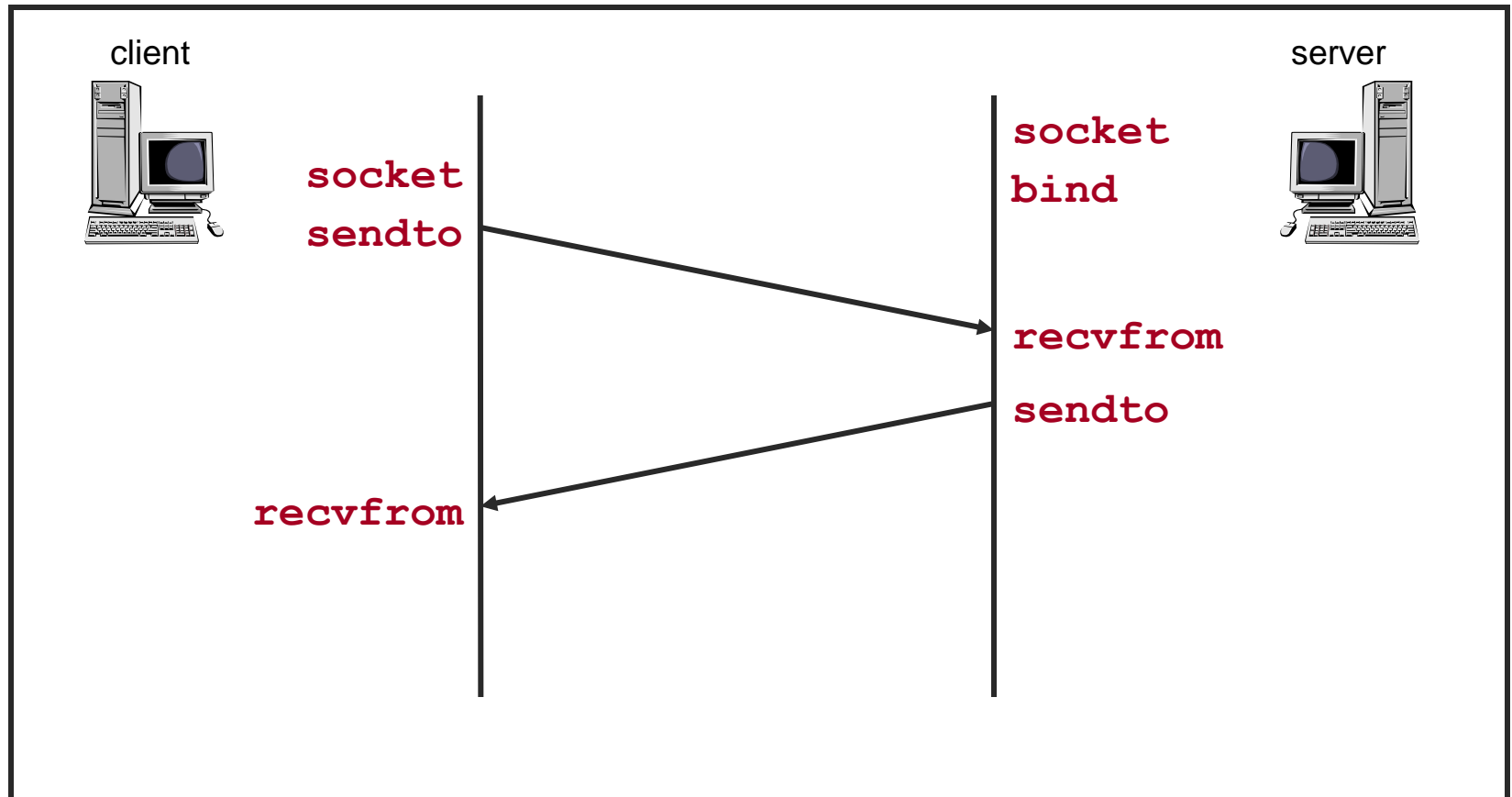
Sending and Receiving Data

- Datagram sockets aren't connected to a remote host
  - What piece of information do we need to give before we send a packet?
  - The destination/source address!

UDP Connection Example

Copyright ©: CS 438 Staff, University of Illinois

Functions: **sendto**

```
int sendto (int sockfd, char* buf, size_t nbytes,
    int flags, struct sockaddr* destaddr, int
    addrlen);
```

- Send a datagram to another UDP socket
  - Returns number of bytes written or -1 and sets **errno** on failure
  - **sockfd**: socket file descriptor (returned from **socket**)
  - **buf**: data buffer
  - **nbytes**: number of bytes to try to read
  - **flags**: see man page for details; typically use 0
  - **destaddr**: IP address and port number of destination socket
  - **addrlen**: length of address structure
    - **= sizeof (struct sockaddr_in)**

Functions: **sendto**

```
int sendto (int sockfd, char* buf, size_t nbytes,
    int flags, struct sockaddr* destaddr, int
    addrlen);
```

■ Example

```
n = sendto(sock, buf, sizeof(buf), 0,(struct
    sockaddr *) &from,fromlen);
if (n < 0)
    perror("sendto");
    exit(1);
}
```

Functions: **recvfrom**

int recvfrom (int sockfd, char* buf, size_t nbytes, int flags, struct sockaddr* srcaddr, int* addrlen);

- Read a datagram from a UDP socket.
  - Returns number of bytes read (0 is valid) or -1 and sets **errno** on failure
  - **sockfd**: socket file descriptor (returned from **socket**)
  - **buf**: data buffer
  - **nbytes**: number of bytes to try to read
  - **flags**: see man page for details; typically use 0
  - **srcaddr**: IP address and port number of sending socket (returned from call)
  - **addrlen**: length of address structure = pointer to **int** set to **sizeof (struct sockaddr_in)**

Functions: **recvfrom**

```
int recvfrom (int sockfd, char* buf, size_t
    nbytes, int flags, struct sockaddr* srcaddr,
    int* addrlen);
```

- Example

```
n = recvfrom(sock, buf, 1024, 0, (struct sockaddr
    *)&from,&fromlen);
if (n < 0) {
    perror("recvfrom");
    exit(1);
}
```

What specific functions to expose?

- Data structures to store information about connections and hosts

- Functions to create a socket

- Functions to establish connections

- Functions to send and receive data

- Functions to teardown connections

Functions: **close**

int close (int sockfd);

- ## Close a socket
  - ○ Returns 0 on success, -1 and sets **errno** on failure
  - ○ **sockfd**: socket file descriptor (returned from **socket**)

- ## Closes communication on socket in both directions
  - ○ All data sent before **close** are delivered to other side (although this aspect can be overridden)
- ## After **close**, **sockfd** is not valid for reading or writing

Functions: **shutdown**

int shutdown (int sockfd, int howto);

- Force termination of communication across a socket in one or both directions
  - Returns 0 on success, -1 and sets **errno** on failure
  - **sockfd**: socket file descriptor (returned from **socket**)
  - **howto**:
    - **SHUT_RD** to stop reading
    - **SHUT_WR** to stop writing
    - **SHUT_RDWR** to stop both

- **shutdown** overrides the usual rules regarding duplicated sockets, in which TCP teardown does not occur until all copies have closed the socket

Note on `close` vs. `shutdown`

- **`close()`**: closes the socket but the connection is still open for processes that shares this socket
  - The connection stays opened both for read and write
- **`shutdown()`**: breaks the connection for all processes sharing the socket
  - A read will detect **EOF**, and a write will receive **SIGPIPE**
  - **`shutdown()`** has a second argument how to close the connection:
    - 0 means to disable further reading
    - 1 to disable writing
    - 2 disables both

One tricky issue…

- Different processor architectures store data in different "byte orderings"
  - What is 200 in binary?
  - **1100 1001**?

    or
  - **1001 1100**?

One tricky issue…

- ## Big Endian vs. Little Endian
  - ○ Little Endian (Intel, DEC):
    - ■ Least significant byte of word is stored in the lowest memory address
  - ○ Big Endian (Sun, SGI, HP, PowerPC):
    - ■ Most significant byte of word is stored in the lowest memory address
  - ○ Example: `128.2.194.95`

Big Endian	128	2	194	95

Little Endian	95	194	2	128

One tricky issue…

- **Big Endian vs. Little Endian**
  - ○ Network Byte Order = Big Endian
    - Allows both sides to communicate
    - Must be used for some data (i.e. IP Addresses)
  - ○ What about ordering within bytes?
    - Most modern processors agree on ordering within bytes

Converting byte orderings

Solution: use byte ordering functions to convert.

```
int m, n;
short int s,t;

m = ntohl (n)    net-to-host long (32-bit) translation
s = ntohs (t)    net-to-host short (16-bit) translation
n = htonl (m)    host-to-net long (32-bit) translation
t = htons (s)    host-to-net short (16-bit) translation
```

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 charters: (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

Advanced Sockets: `signal`

- Problem: Socket at other end is closed
  - Write to your end generates **SIGPIPE**
  - This signal kills the program by default!

signal (SIGPIPE, SIG_IGN);

  - Call at start of main in server
  - Allows you to ignore broken pipe signals
  - Can ignore or install a proper signal handler
  - Default handler exits (terminates process)

Advanced Sockets

- Problem: How come I get "address already in use" from **bind()** ?
  - You have stopped your server, and then re-started it right away
  - The sockets that were used by the first incarnation of the server are still active

Advanced Sockets: **setsockopt**

```
int yes = 1;
setsockopt (fd, SOL_SOCKET,
  SO_REUSEADDR, (char *) &yes, sizeof
  (yes));
```

- Call just before **bind()**
- Allows bind to succeed despite the existence of existing connections in the requested TCP port
- Connections in limbo (e.g. lost final ACK) will cause bind to fail
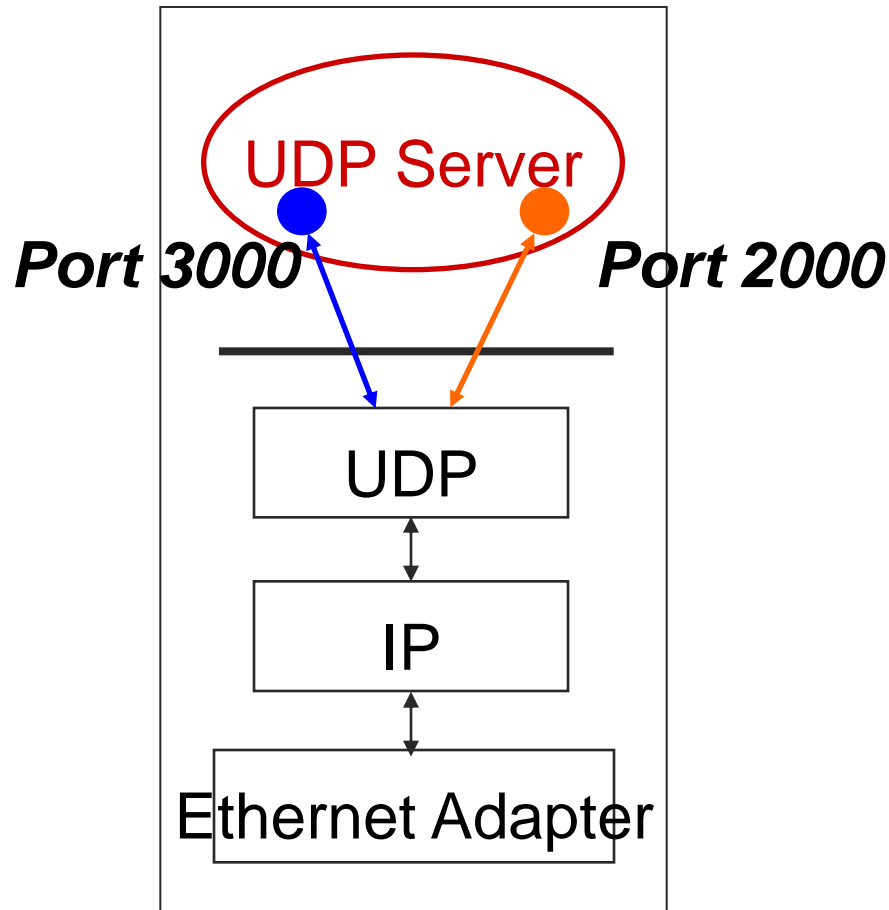
How to handle concurrency?

- Process requests serially
  - Slow – what if you're processing another request? What if you're blocked on `read()`?

A UDP Server

- How can a UDP server service multiple ports simultaneously?

UDP Server: Servicing Two Ports

```
int s1;                          /* socket descriptor 1 */
int s2;                          /* socket descriptor 2 */


/* 1) create socket s1 */
/* 2) create socket s2 */
/* 3) bind s1 to port 2000 */
/* 4) bind s2 to port 3000 */


while(1) {
    recvfrom(s1, buf, sizeof(buf), ...);
    /* process buf */
    recvfrom(s2, buf, sizeof(buf), ...);
    /* process buf */
}
```

What problems does this code have?

How to handle concurrency?

- Process requests serially
  - Slow – what if you're processing another request? What if you're blocked on **accept()**?

- Multiple threads/processes (e.g. Apache, Chrome)
  - Each thread/process handles one request
  - **fork(), pthreads**

- Synchronous I/O (e.g. Squid web proxy cache)
  - Maintain a "set" of file descriptors, whenever one has an "event", process it and put it back onto the set
  - **select(), poll()**

Select

int select (int num_fds, fd_set* read_set, fd_set* write_set, fd_set* except_set, struct timeval* timeout);

- Wait for readable/writable file descriptors.
- Return:
  - ○ Number of descriptors ready
  - ○ -1 on error, sets **errno**
- Parameters:
  - ○ **num_fds**:
    - number of file descriptors to check, numbered from 0
  - ○ **read_set, write_set, except_set**:
    - Sets (bit vectors) of file descriptors to check for the specific condition
  - ○ **timeout**:
    - Time to wait for a descriptor to become ready

File Descriptor Sets

```
int select (int num_fds, fd_set* read_set,
    fd_set* write_set, fd_set* except_set, struct
    timeval* timeout);
```

- **Bit vectors**
  - Only first **num_fds** checked
  - Macros to create and check sets

```
fds_set myset;
void FD_ZERO (&myset);     /* clear all bits */
void FD_SET (n, &myset);   /* set bits n to 1 */
void FD_CLEAR (n, &myset); /* clear bit n */
int FD_ISSET (n, &myset);  /* is bit n set? */
```

File Descriptor Sets

- **Three conditions to check for**
  - Readable:
    - Data available for reading
  - Writable:
    - Buffer space available for writing
  - Exception:
    - Out-of-band data available (TCP)

Building Timeouts with Select and Poll

- ## Time structure

Number of seconds since midnight, January 1, 1970 GMT

```
struct timeval {
    long tv_sec;          /* seconds */
    long tv_usec;   /* microseconds */
};
```

unix will have its own "Y2K" problem one second after 10:14:07pm, Monday January 18, 2038 (will appear to be 3:45:52pm, Friday December 13, 1901)

Select

- **High-resolution sleep function**
  - All descriptor sets `NULL`
  - Positive `timeout`
- **Wait until descriptor(s) become ready**
  - At least one descriptor in set
  - `timeout NULL`
- **Wait until descriptor(s) become ready or timeout occurs**
  - At least one descriptor in set
  - Positive `timeout`
- **Check descriptors immediately (poll)**
  - At least one descriptor in set
  - 0 `timeout`

Which file descriptors are set and what should the timeout value be?

Select: Example

```
fd_set my_read;
FD_ZERO(&my_read);
FD_SET(0, &my_read);

if (select(1, &my_read, NULL, NULL) == 1) {
    assert(FD_ISSET(0, &my_read);
    /* data ready on stdin */
}
```

What went wrong: after select indicates data available on a connection, read returns no data?

Select: Timeout Example

```
I nt main(void) {
    struct timeval tv;
    fd_set readfds;
    tv.tv_sec = 2;
    tv.tv_usec = 500000;
    FD_ZERO(&readfds);
    FD_SET(STDIN, &readfds);
    // don't care about writefds and exceptfds:
    select(1, &readfds, NULL, NULL, &tv);
    if (FD_ISSET(STDIN, &readfds))
        printf("A key was pressed!\n");
    else
        printf("Timed out.\n");
    return 0;
}
```

Wait 2.5 seconds for something to appear on standard input

`select()` vs. `poll()`

Which to use?

- **BSD-family** (e.g., FreeBSD, MacOS)
  - `poll()` just calls `select()` internally
- **System V family** (e.g., AT&T Unix)
  - `select()` just calls `poll()` internally

Concurrent programming with Posix Threads (pthreads)

- **Thread management**
  - Creating, detaching, joining, etc. Set/query thread attributes
- **Mutexes**
  - Synchronization
- **Condition variables**
  - Communications between threads that share a mutex

Creating a Thread

```
int pthread_create (pthread_t* tid,
    pthread_attr_t* attr, void*(child_main), void*
    arg);
```

- **pthread_create()** takes a pointer to a function as one of its arguments
  - **child_main** is called with the argument specified by **arg**
  - **child_main** can only have one parameter of type **void ***
  - Complex parameters can be passed by creating a structure and passing the address of the structure
  - The structure can't be a local variable

Example: pthreads

```c
#include <pthread.h>              void *PrintHello(void *threadid) {
#define NUM_THREADS 5                     printf("\n%d: Hello World!\n", threadid);
                                          pthread_exit(NULL);
                                  }


int main (int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc, t;

    for(t=0;t < NUM_THREADS;t++) {
        printf("Creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc) {
            printf("ERROR; pthread_create() return code is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

Example: `pthread_join()`

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 4

int main (int argc, char *argv[]) {
    pthread_t thread[NUM_THREADS];
    pthread_attr_t attr;
    int rc;
    long t;
    void *status;
```

```c
    /* Initialize and set thread detached
       attribute */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr,
        PTHREAD_CREATE_JOINABLE);

    for(t=0; t<NUM_THREADS; t++) {
        printf("Main: creating thread %ld\n", t);
        rc = pthread_create(&thread[t], &attr,
            BusyWork, (void *)t);
        if (rc) {
            printf("ERROR; return code is %d\n",
                    rc);
            exit(-1);
        }
    }
    /* Free attributes */
    pthread_attr_destroy(&attr);
```

Example: `pthread_join()`

```c
void *BusyWork(void *t) {
    int i;
    long tid;
    double result = 0.0;
    tid = (long)t;
    printf("Thread %ld starting...\n",
        tid);
    for (i=0; i<1000000; i++) {
        result = result + sin(i) * tan(i);
    }
    printf("Thread %ld result = %e\n",
        tid, result);
    pthread_exit((void*) t);
}
```

```c
int main (int argc, char *argv[]) {
...

    /* Wait for the other threads */
    for(t=0; t<NUM_THREADS; t++) {
        rc = pthread_join(thread[t], &status);
        if (rc) {
            printf("ERROR; return code is %d\n", rc);
            exit(-1);
        }
        printf("Main: status for thread %ld: %ld\n",
                t, (long)status);
    }

    printf("Main: program completed. Exiting.\n");
    pthread_exit(NULL);
}
```

Using pthreads

- ## When coding
  - Include **`<pthread.h>`** first in all source files

- ## When compiling
  - Use compiler flag **`–D_REENTRANT`**

- ## When linking
  - Link library **`-lpthread`**

pthread Error Handling

- pthreads functions do not follow the usual Unix conventions
  - Similarity
    - Returns 0 on success
  - Differences
    - Returns error code on failure
    - Does not set **errno**
  - What about **errno**?
    - Each thread has its own
    - Define **_REENTRANT** (**-D_REENTRANT** switch to compiler) when using pthreads

Summary

- ## Unix Network Programming
  - Transport protocols
    - TCP, UDP
  - Network programming
    - Sockets API, pthreads
- ## Next
  - Probability refresher
  - Direct link networks