

Lecture 2: Sockets Programming

CS/ECE 438: Communication Networks

Prof. Matthew Caesar

January 22, 2010

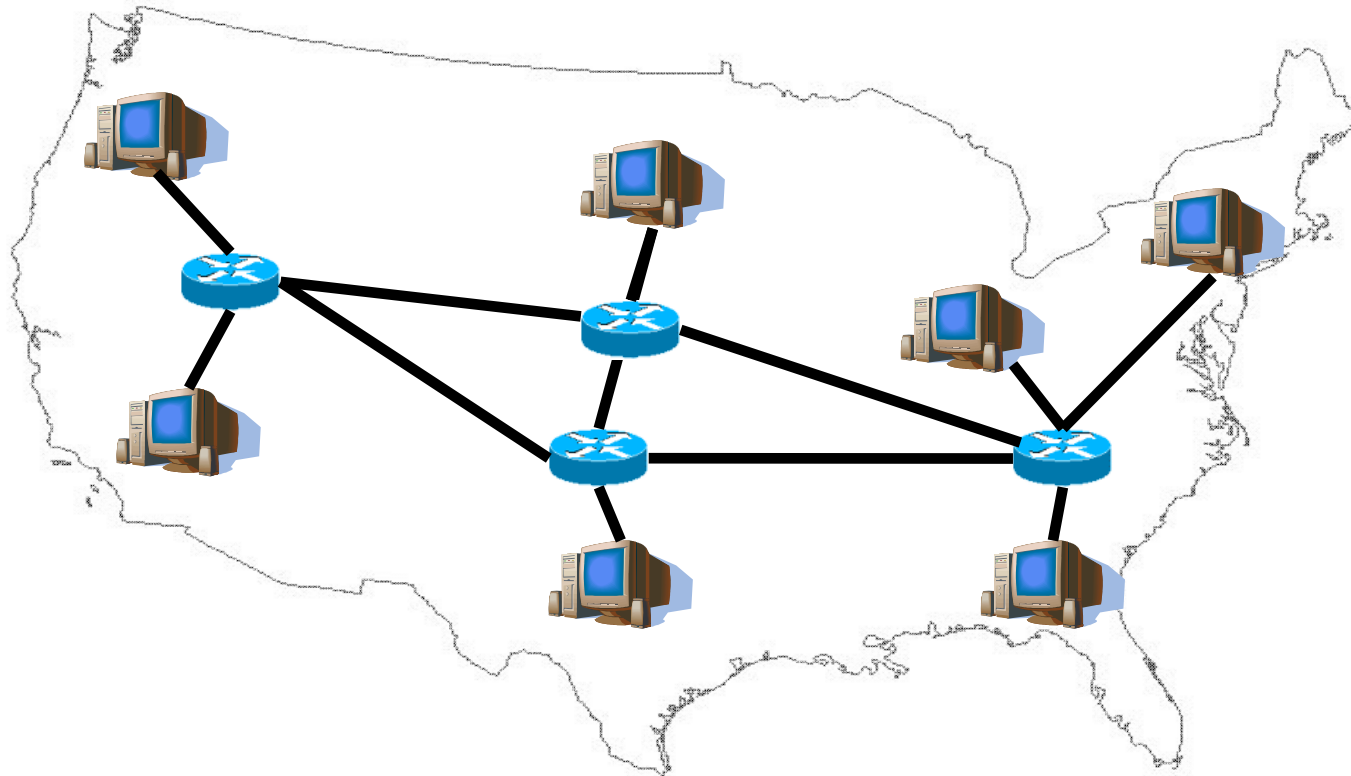
Network Programming with Sockets

- 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”

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

How can many hosts communicate?

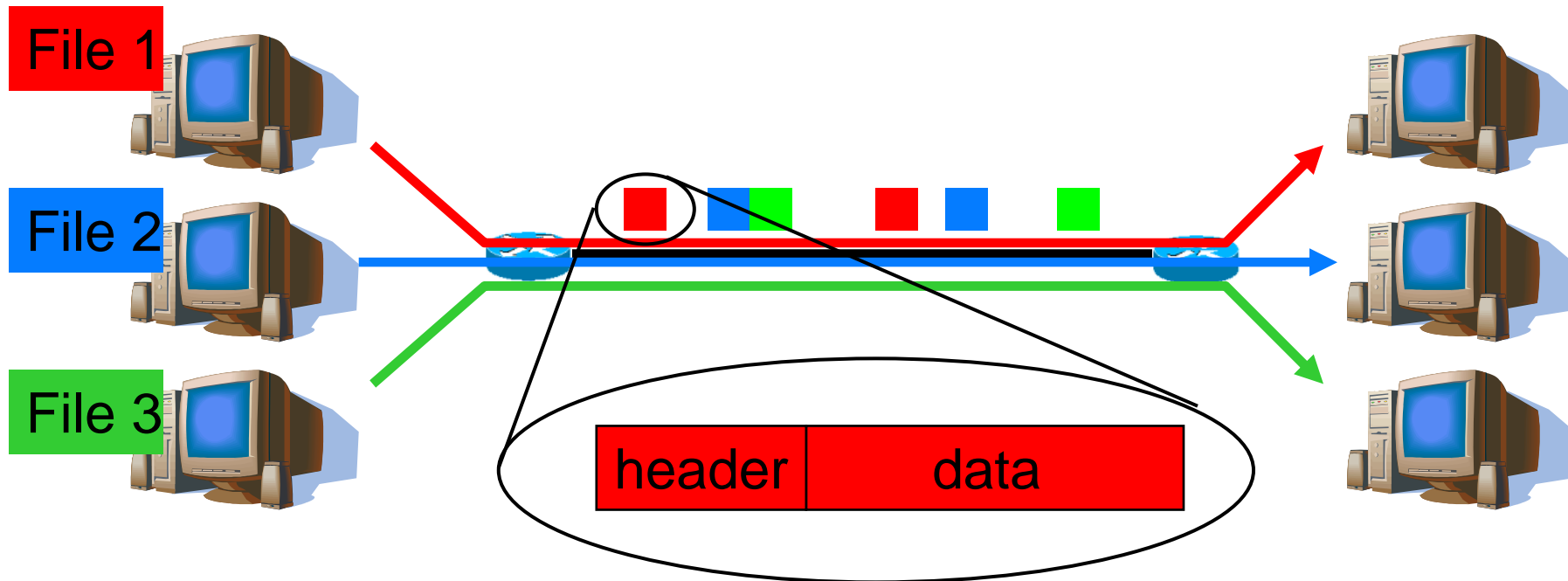


- Multiplex traffic with routers
- Question: How to identify the destination?
- Question: How to share bandwidth across different flows?

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
 - 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? Or if contents get corrupted?
- Solution: User Datagram Protocol (UDP)
 - 16-bit “Port numbers” in header distinguishes traffic from different applications
 - “Checksum” covering data, UDP header, and IP header detects flipped bits
 - Unit of Transfer is “datagram” (a variable length packet)
 - Properties:
 - *Unreliable* (no guaranteed delivery)
 - *Unordered* (no guarantee of maintained order of delivery)
 - *Unlimited Transmission* (no flow control)

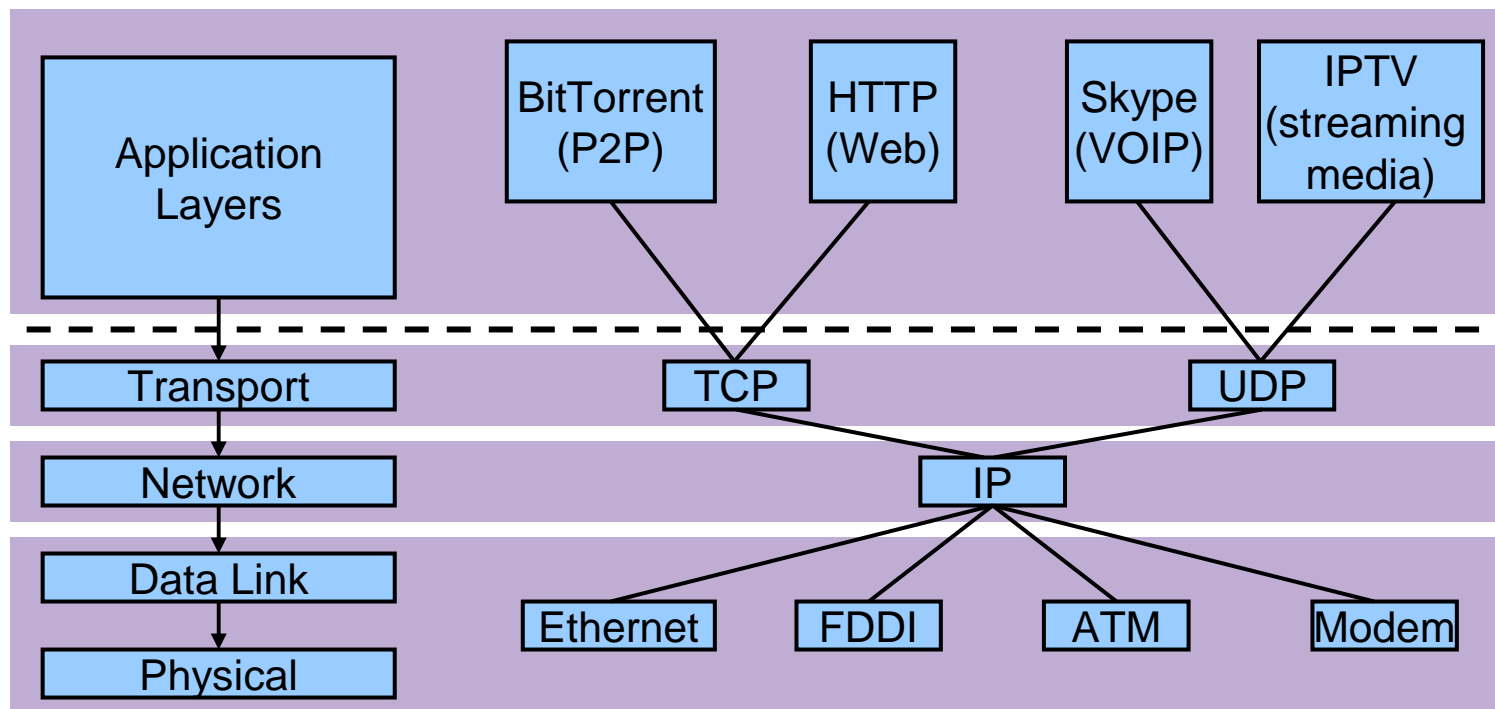
Is UDP enough?

- What if network gets congested? Or packets get lost/reordered/duplicated?
- Solution: Transport Control Protocol (TCP)
 - Uses “sequence numbers” and 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?

TCP Service

- **Reliable Data Transfer**
 - Guarantees delivery of all data
 - Exactly once if no catastrophic failures
- **Sequenced Data Transfer**
 - Guarantees in-order delivery of data
 - If A sends M1 followed by M2 to B, B never receives M2 before M1
- **Regulated Data Flow**
 - Monitors network and adjusts transmission appropriately
 - Prevents senders from wasting bandwidth
 - Reduces global congestion problems
- **Data Transmission**
 - Full-Duplex byte stream

Internet Protocols

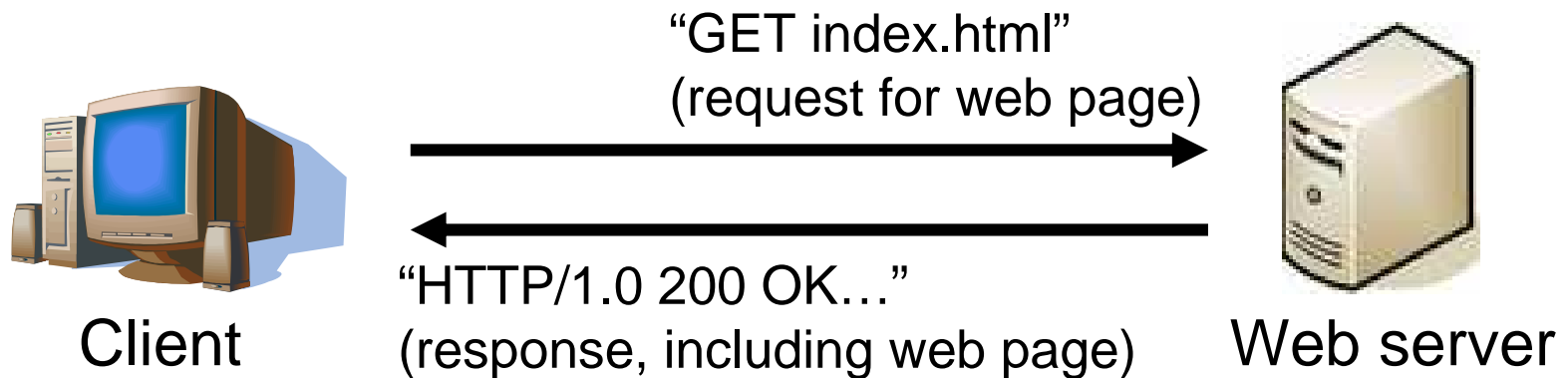


Next question: How should people program networked apps?

- 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

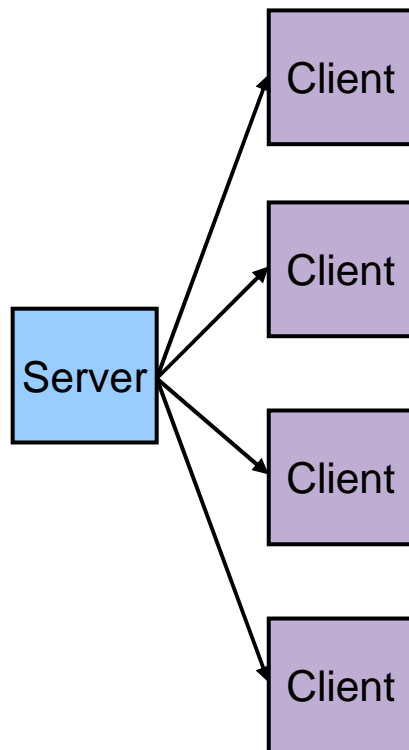
Client-Server Model

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



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

Client-Server Model



- 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
- Can you think of any network apps that are not client/server?

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?

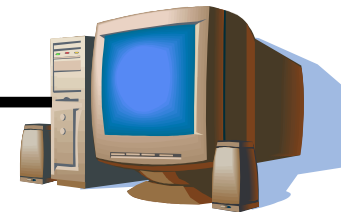
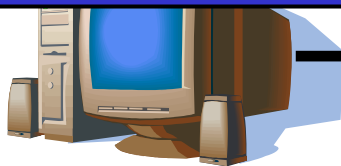
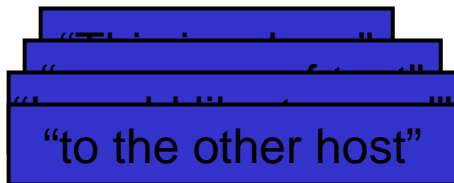
- Stream vs. Datagram sockets
- Stream sockets
 - Abstraction: send a long stream of characters
 - Typically implemented on top of TCP
- Datagram sockets
 - Abstraction: send a single packet
 - Typically implemented on top of UDP

Stream sockets

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



Sockets API



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



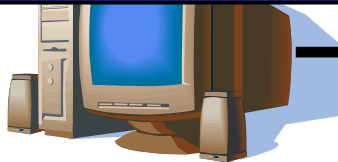
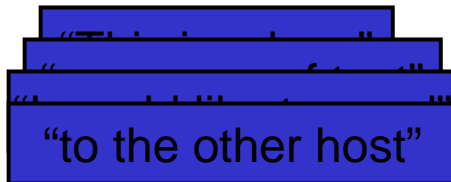
Sockets API

Datagram sockets

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



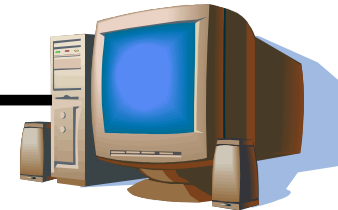
Sockets API



“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



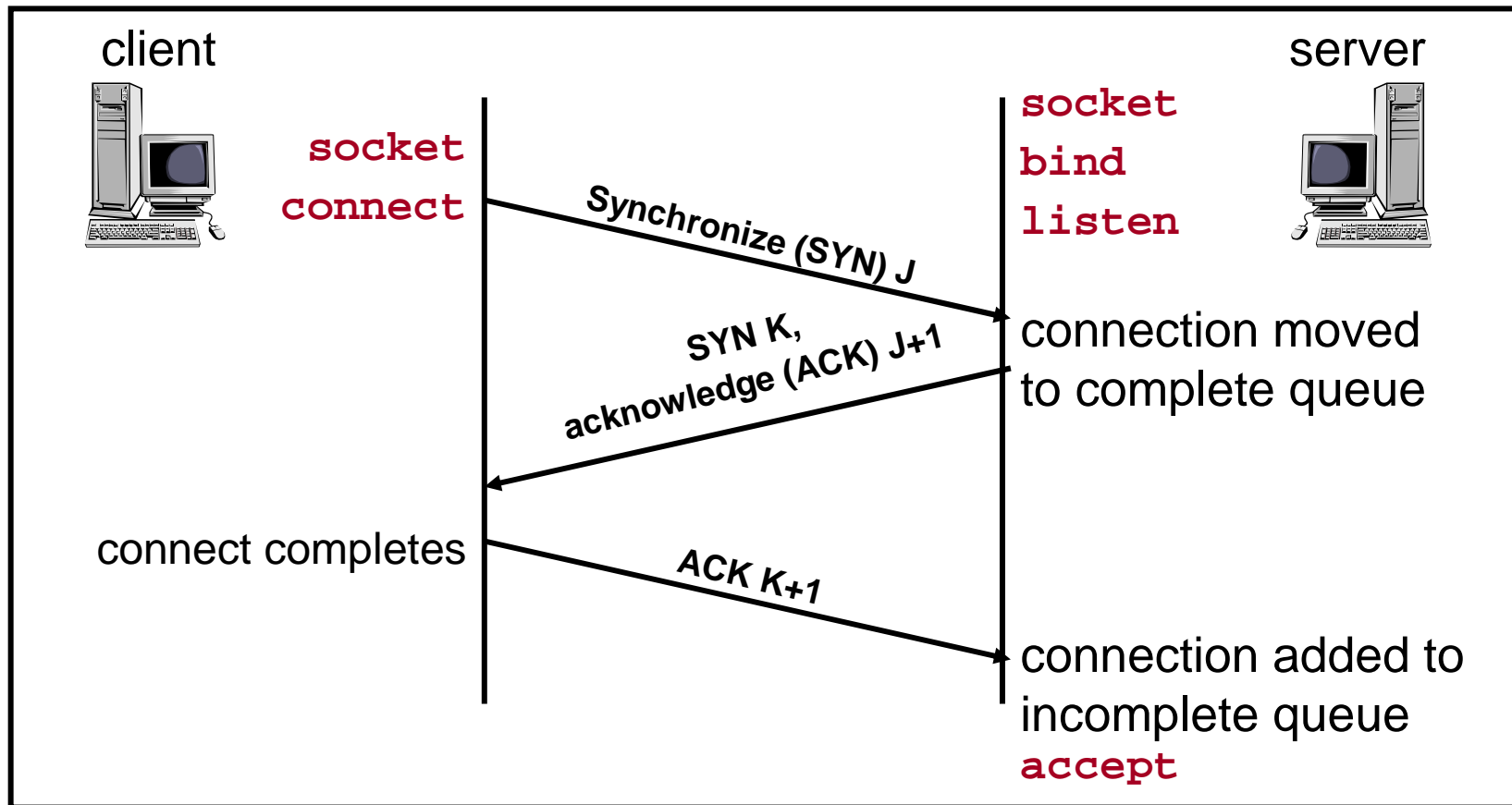
What specific functions to expose?

- Data structures to *store information about connections and hosts*
- Functions to *create and bind "socket descriptors"*
- Functions to *establish and teardown connections*
- Functions to *send and receive data over connections*

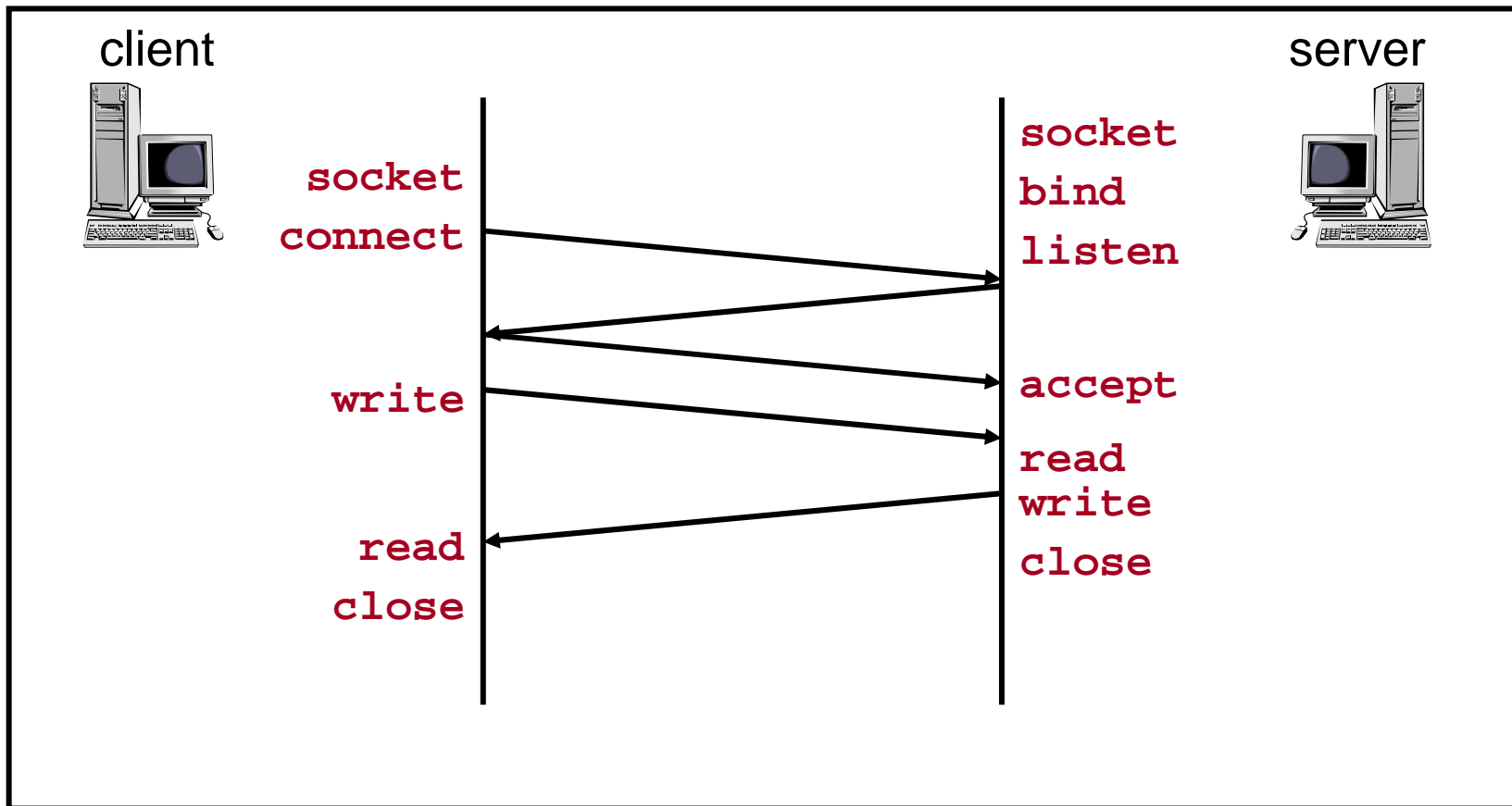
Example: TCP streaming client

1. Client specifies an IP address and port it wants to connect to.
2. The sockets library takes care of the connection setup details, and returns back a unique integer (a "socket").
3. When the application wants to send data, it specifies the socket number, and a pointer to the data it wants to send.
4. The library looks up in a table the IP/port information corresponding to that socket number, constructs a packet, puts that IP/port in the header, and sends the packet.

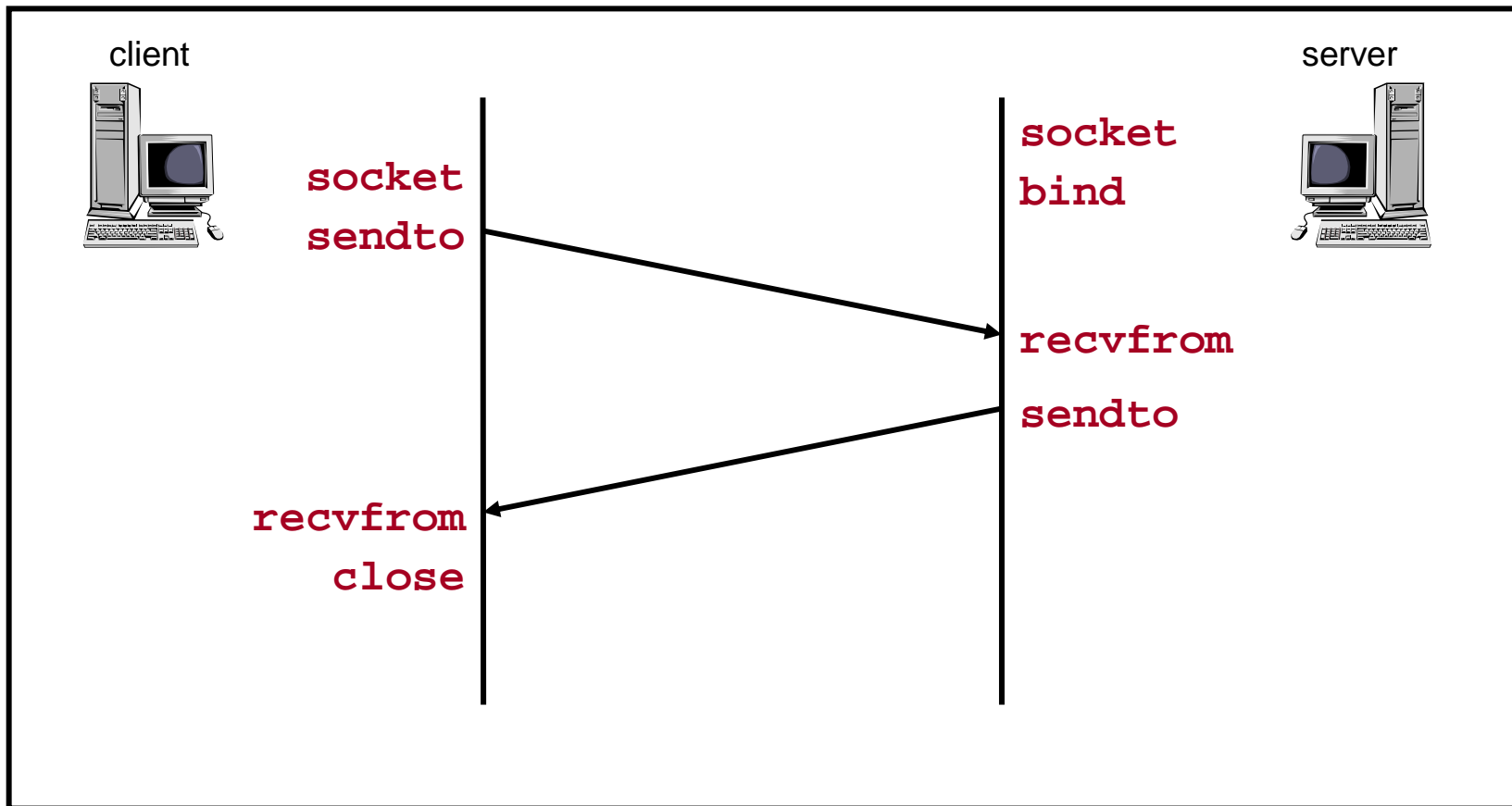
TCP Connection Setup



TCP Connection Example



UDP Connection Example



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

```
int sockfd, numbytes;
```

Socket descriptor (used to identify connections)

```
char buf[MAXDATASIZE + 1];
```

```
struct hostent* he;
```

Information about a host (domain name, list of addresses associated with machine, type of address, etc)

```
struct sockaddr_in their_addr;
```

```
/* connector's address information */
```

```
if (argc != 2) {
```

```
    fprintf (stderr, "usage: client hostname\n");
```

```
    exit (1);
```

```
}
```

Gets IP address and other info for specified hostname via DNS lookup

```
if ((he = gethostbyname (argv[1])) == NULL) {
```

```
    /* get the host info */
```

```
    perror ("gethostbyname");
```

```
    exit (1);
```

```
}
```

Returns an available socket descriptor

```
if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) == -1) {
```

```
    perror ("socket");
```

```
    exit (1);
```

```
}
```

Specifies identity of server we will connect to

```
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);
```

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

Returns an available socket descriptor

```
if ((numbytes = recv (sockfd, buf, MAXDATASIZE, 0))  
    == -1) {  
    perror ("recv");  
    exit (1);  
}
```

Receive data from server, put it into buf

```
buf[numbytes] = '\0';  
printf ("Received: %s", buf);  
close (sockfd);  
return 0;
```

Tell OS we are done with this socket, Which will clean up state and tear down the connection

```
}
```

```
// SERVER CODE
```

```
main()
```

```
{  
  int sockfd, new_fd;  
  struct sockaddr_in my_addr; /* my address */  
  struct sockaddr_in their_addr; /* connector address */  
  int sin_size;  
  
  if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {  
    perror("socket");  
    exit(1);  
  }  
  
  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);  
  }  
}
```

Socket descriptors (used to identify connections)

Returns an available socket descriptor

Specifies identity of server's end of the connection

Associates that identity to the socket

```
// SERVER CODE (continued)
```

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

Tell OS that we are willing
To accept connections on this socket

```
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;  
    }
```

Associate "new_fd" with the next client
that connects (block until one does)

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

```
    if (!fork()) { /* this is the child process */
```

```
        if (send(new_fd, "Hello, world!\n", 14, 0)  
            == -1)  
            perror("send");
```

```
        close(new_fd);  
        exit(0);
```

Send "hello world" to the
client connected to new_fd

```
    }  
    close(new_fd); /* parent doesn't need this */  
    /* clean up all child processes */  
    while(waitpid(-1, NULL, WNOHANG) > 0);
```

Tell OS we are done with this socket, which
will clean up state and tear down the connection

```
    }  
}
```

Sockets API details

- Data structures to store/convert information about hosts/connections
 - `inet_ntoa`, `inet_aton`, `gethostbyname`,
- Functions to create and bind socket descriptors
 - `socket`, `bind`, `listen`
- Functions to establish and teardown connections
 - `connect`, `accept`, `close`, `shutdown`
- Functions to send and receive data
 - `send`, `sendto`, `write`, `recv`, `recvfrom`, `read`

One tricky issue...

- Different processor architectures store data in different “byte orderings”
 - What is 200 in binary? 1100 1001? Or 1001 1100?
- **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
 - Host Byte Order can be Big or Little Endian
 - Network Byte Order = Big Endian
 - Allows both sides to communicate
 - Must be used for some data (i.e. IP Addresses)

Converting byte orderings

Solution: use byte ordering functions to convert. E.g.:

```
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 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

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 web server)
 - Each thread handles one request
 - `fork()`, `pthread`s
- 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

```
fd_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)

Timeout

- Structure

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

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`

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 */
}
```

- Question: which is better, pthreads or select?

Advanced Sockets

```
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

Concurrent programming with Posix Threads (pthreads)

- When coding
 - Include `<pthread.h>` first in all source files
- When compiling
 - Use compiler flag `-D_REENTRANT`
- When linking
 - Link library `-lpthread`

```
// PTHREADS EXAMPLE
```

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

```
    int n,i;
```

```
    pthread_t *threads;
```

```
    pthread_attr_t pthread_custom_attr;
```

```
    parm *p;
```

```
    if (argc != 2)
```

```
    {
```

```
        printf ("Usage: %s n\n where n is no. of threads\n",argv[0]);
```

```
        exit(1);
```

```
    }
```

```
    n=atoi(argv[1]);
```

```
    if ((n < 1) || (n > MAX_THREAD))
```

```
    {
```

```
        printf ("The no of thread should between 1 and  
                %d.\n",MAX_THREAD);
```

```
        exit(1);
```

```
    }
```

```
    threads=(pthread_t *)malloc(n*sizeof(*threads));
```

```
    pthread_attr_init(&pthread_custom_attr);
```

```
    p=(parm *)malloc(sizeof(parm)*n);
```

Contains thread information, acts as handle for thread

Specifies "attributes" for thread, like Scheduling policy/priority and stack size

Initializes attributes to default values (NULL)

```
/* Start up threads */
```

```
for (i=0; i<n; i++)
```

```
{
```

```
    p[i].id=i;
```

```
    pthread_create(&threads[i], &pthread_custom_attr, hello,  
                  (void *) (p+i));
```

```
}
```

Creates a pthread, assigns it attributes, triggers it to run function pointer

```
/* Synchronize the completion of each thread. */
```

```
for (i=0; i<n; i++)
```

```
{
```

```
    pthread_join(threads[i], NULL);
```

```
}
```

```
free(p);
```

```
}
```

Wait on termination of thread threads[i]

Thread function (passed in during pthread_create)

```
void *hello(void *arg)
```

```
{
```

```
    parm *p=(parm *)arg;
```

```
    printf("Hello from node %d\n", p->id);
```

```
    return (NULL);
```

```
}
```

pthread Creation

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

- Spawn a new posix thread
- Parameters:
 - `tid`:
 - Unique thread identifier returned from call
 - `attr`:
 - Attributes structure used to define new thread
 - Use `NULL` for default values
 - `child_main`:
 - Main routine for child thread
 - Takes a pointer (`void*`), returns a pointer (`void*`)
 - `arg`:
 - Argument passed to child thread

Sockets API details

- Data structures to store/convert information about hosts/connections
 - `inet_ntoa`, `inet_aton`, `gethostbyname`,
- Functions to create and bind socket descriptors
 - `socket`, `bind`, `listen`
- Functions to establish and teardown connections
 - `connect`, `accept`, `close`, `shutdown`
- Functions to send and receive data
 - `send`, `sendto`, `write`, `recv`, `recvfrom`, `read`

Socket Address Structure

- IP address:

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

- TCP or UDP address+port:

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

- all but sin_family in network byte order

Address Access/Conversion Functions

- All binary values are network byte ordered

```
struct hostent* gethostbyname (const char*  
    hostname);
```

- Translate DNS host name to IP address (uses DNS)

```
struct hostent* gethostbyaddr (const char*  
    addr, size_t len, int family);
```

- Translate IP address to DNS host name (not secure)

```
char* inet_ntoa (struct in_addr inaddr);
```

- Translate IP address to ASCII dotted-decimal notation (e.g., "128.32.36.37"); not thread-safe

Address Access/Conversion Functions

```
in_addr_t inet_addr (const char* strptr);
```

- Translate dotted-decimal notation to IP address; returns -1 on failure, thus cannot handle broadcast value "255.255.255.255"

```
int inet_aton (const char* strptr, struct in_addr inaddr);
```

- Translate dotted-decimal notation to IP address; returns 1 on success, 0 on failure

```
int gethostname (char* name, size_t namelen);
```

- Read host's name (use with `gethostbyname` to find local IP)

Socket Creation and Setup

- Include file `<sys/socket.h>`
- Create a socket
 - `int socket (int family, int type, int protocol);`
 - Returns file descriptor or -1.
- Bind a socket to a local IP address and port number
 - `int bind (int sockfd, struct sockaddr* myaddr, int addrlen);`
- Put socket into passive state (wait for connections rather than initiate a connection).
 - `int listen (int sockfd, int backlog);`

Functions: 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

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 or
`ftp://ftp.isi.edu/in-notes/iana/assignments/port-numbers`

1-512	<ul style="list-style-type: none">• standard services (see <code>/etc/services</code>)• super-user only
513-1023	<ul style="list-style-type: none">• registered and controlled, also used for identity verification• super-user only
1024-49151	<ul style="list-style-type: none">• registered services/ephemeral ports
49152-65535	<ul style="list-style-type: none">• private/ephemeral ports

Reserved Ports

Keyword	Decimal	Description
-----	-----	-----
	0/tcp	Reserved
	0/udp	Reserved
tcpmux	1/tcp	TCP Port Service
tcpmux	1/udp	TCP Port Service
echo	7/tcp	Echo
echo	7/udp	Echo
sysstat	11/tcp	Active Users
sysstat	11/udp	Active Users
daytime	13/tcp	Daytime (RFC 867)
daytime	13/udp	Daytime (RFC 867)
gotd	17/tcp	Quote of the Day
gotd	17/udp	Quote of the Day
chargen	19/tcp	Character Generator
chargen	19/udp	Character Generator
ftp-data	20/tcp	File Transfer Data
ftp-data	20/udp	File Transfer Data
ftp	21/tcp	File Transfer Ctl
ftp	21/udp	File Transfer Ctl
ssh	22/tcp	SSH Remote Login
ssh	22/udp	SSH Remote Login
telnet	23/tcp	Telnet
telnet	23/udp	Telnet
smtp	25/tcp	Simple Mail Transfer
smtp	25/udp	Simple Mail Transfer

Keyword	Decimal	Description
-----	-----	-----
time	37/tcp	Time
time	37/udp	Time
name	42/tcp	Host Name Server
name	42/udp	Host Name Server
nameserver	42/tcp	Host Name Server
nameserver	42/udp	Host Name Server
nicname	43/tcp	Who Is
nicname	43/udp	Who Is
domain	53/tcp	Domain Name Server
domain	53/udp	Domain Name Server
whois++	63/tcp	whois++
whois++	63/udp	whois++
gopher	70/tcp	Gopher
gopher	70/udp	Gopher
finger	79/tcp	Finger
finger	79/udp	Finger
http	80/tcp	World Wide Web HTTP
http	80/udp	World Wide Web HTTP
www	80/tcp	World Wide Web HTTP
www	80/udp	World Wide Web HTTP
www-http	80/tcp	World Wide Web HTTP
www-http	80/udp	World Wide Web HTTP
kerberos	88/tcp	Kerberos
kerberos	88/udp	Kerberos

Functions: 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

Establishing a Connection

- Include file `<sys/socket.h>`

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

- Connect to another socket.

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

- Accept a new connection. Returns file descriptor or -1.

Functions: 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

Functions: accept

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

- Accept 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)

Sending and Receiving Data

```
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.

```
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.

Sending and Receiving Data

```
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.

```
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 or -1.

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
- Some reasons for failure or partial writes
 - process received interrupt or signal
 - kernel resources unavailable (*e.g.*, buffers)

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 and 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

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: 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)`

Tearing Down a Connection

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

- Close a socket.
 - Returns 0 on success, -1 and sets `errno` on failure.

```
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.

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

Summary

- Transport protocols
 - TCP, UDP
- Network programming
 - Sockets API, pthreads