EE4210: Computer Communication Networks II

Socket Programming

NUS
National University of Singapore

Outline

- Principles of socket programming
- Example: TCP based client-server

Basics

- Transport layer: TCP/UDP
- C programming basics
- Data structures basics
- Windows: winsock library

Telephone Analogy

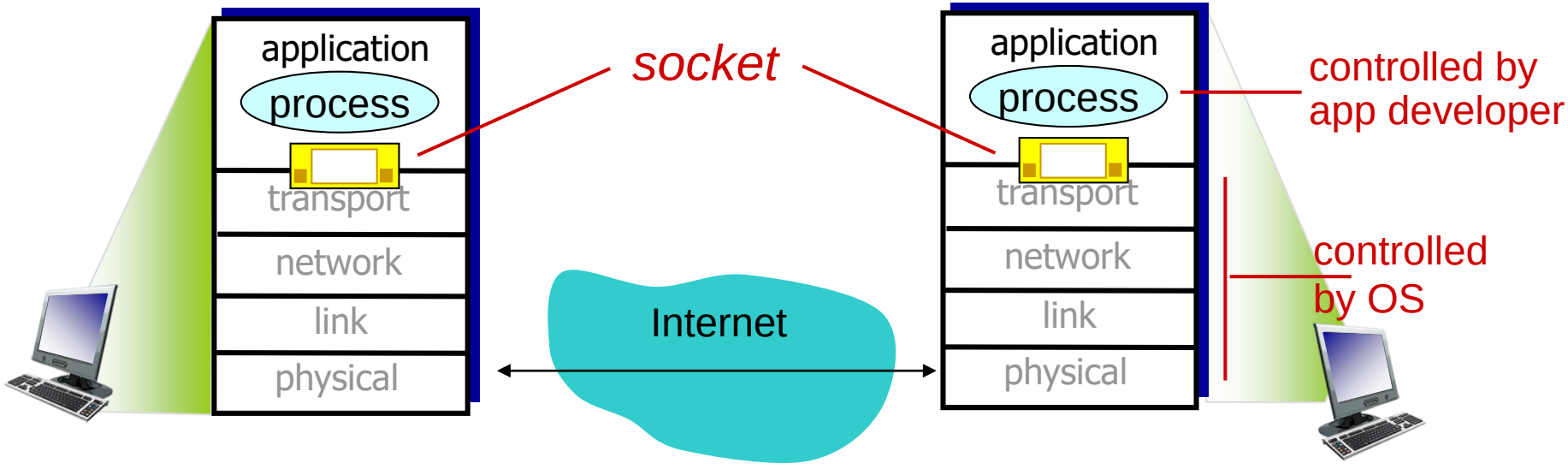
- Both sides need to have telephones installed
- Phone number is assigned to both sides
- Ringer is turned on to listen for calls
- Caller lifts phone and dials a number
- Phone rings and the receiver picks up the phone
- Conversation
- Both sides hang up once the call is over

In the Computer Universe

- socket(): endpoint for communication
- bind(): assign a unique (telephone) number
- listen(): wait for a call
- connect(): dial a number
- accept(): receive a call
- send(), recv(): conversation
- close(): hang up

Sockets

- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process



Sockets

- A socket is a file descriptor that lets an application read/write data from/to the network

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

```
int fd;            /* socket descriptor */
if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
      perror("socket");
      exit(1);
}
```

  - socket returns an integer (socket descriptor)
    - fd < 0 indicates that an error occurred
    - socket descriptors are similar to file descriptors
  - AF_INET: associates a socket with the Internet protocol family
  - SOCK_STREAM: selects the TCP protocol
  - SOCK_DGRAM: selects the UDP protocol

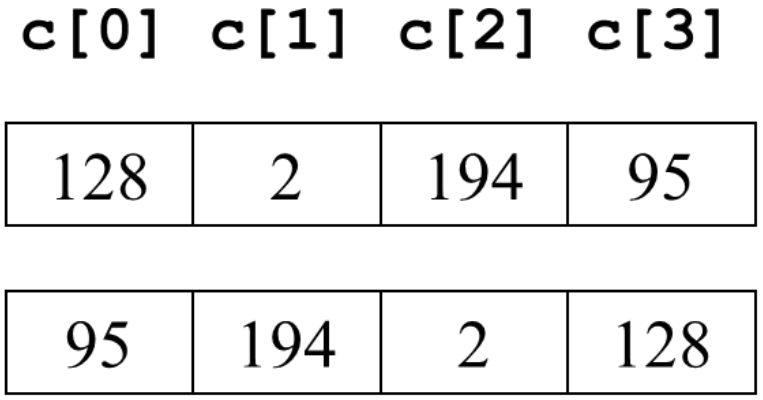
Internet Addressing Data Structure


```
#include <netinet/in.h>

/* Internet address structure */
struct in_addr {
        u_long s_addr;          /* 32-bit IPv4 address */
};                              /* network byte ordered */

/* Socket address, Internet style. */
struct sockaddr_in {
      u_char  sin_family;       /* Address Family */
      u_short sin_port;         /* UDP or TCP Port# */
                                /* network byte ordered */
      struct in_addr sin_addr;  /* Internet Address */
      char    sin_zero[8];      /* unused */
};
```

sin_family = AF_INET selects Internet address family

Byte Ordering

- Big Endian
  - Sun Solaris, PowerPC, ...
- Little Endian
  - i386, alpha, ...
- Network byte order = Big Endian



Internet Addressing Data Structure

- Converts between host byte order and network byte order
  - ‘h’ = host byte order
  - ‘n’ = network byte order
  - ‘l’ = long (4 bytes), converts IP addresses
  - ‘s’ = short (2 bytes), converts port numbers

```
#include <netinet/in.h>

unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

Create Socket

- A socket is a file descriptor that lets an application read/write data from/to the network

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

```
int fd;            /* socket descriptor */
if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
      perror("socket");
      exit(1);
}
```

  - socket returns an integer (socket descriptor)
    - fd < 0 indicates that an error occurred
    - socket descriptors are similar to file descriptors
  - AF_INET: associates a socket with the Internet protocol family
  - SOCK_STREAM: selects the TCP protocol
  - SOCK_DGRAM: selects the UDP protocol

Bind Socket to Address

- A socket can be bound to a port

```
int bind(int sockfd, const struct sockaddr
*addr, socklen_t addrlen);
```

```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;          /* used by bind() */

/* create the socket */

srv.sin_family = AF_INET; /* use the Internet addr family */
srv.sin_port = htons(80); /* bind socket 'fd' to port 80*/

/* bind: a client may connect to any of my addresses */
srv.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
      perror("bind"); exit(1);
}
```

Listen to the port

- listen() indicates that the server will accept a connection

```
int listen(int sockfd, int backlog);
```

```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;          /* used by bind() */

/* 1) create the socket */
/* 2) bind the socket to a port */

if(listen(fd, 5) < 0) {
      perror("listen");
      exit(1);
}
```

Accept Connections

- accept returns a new socket (newfd) with the same properties as the original socket (fd)

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

```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;          /* used by bind() */
struct sockaddr_in cli;          /* used by accept() */
int newfd;                       /* returned by accept() */
int cli_len = sizeof(cli);       /* used by accept() */

/* 1) create the socket */
/* 2) bind the socket to a port */
/* 3) listen on the socket */

newfd = accept(fd, (struct sockaddr*) &cli, &cli_len);
if(newfd < 0) {
      perror("accept"); exit(1);
}
```

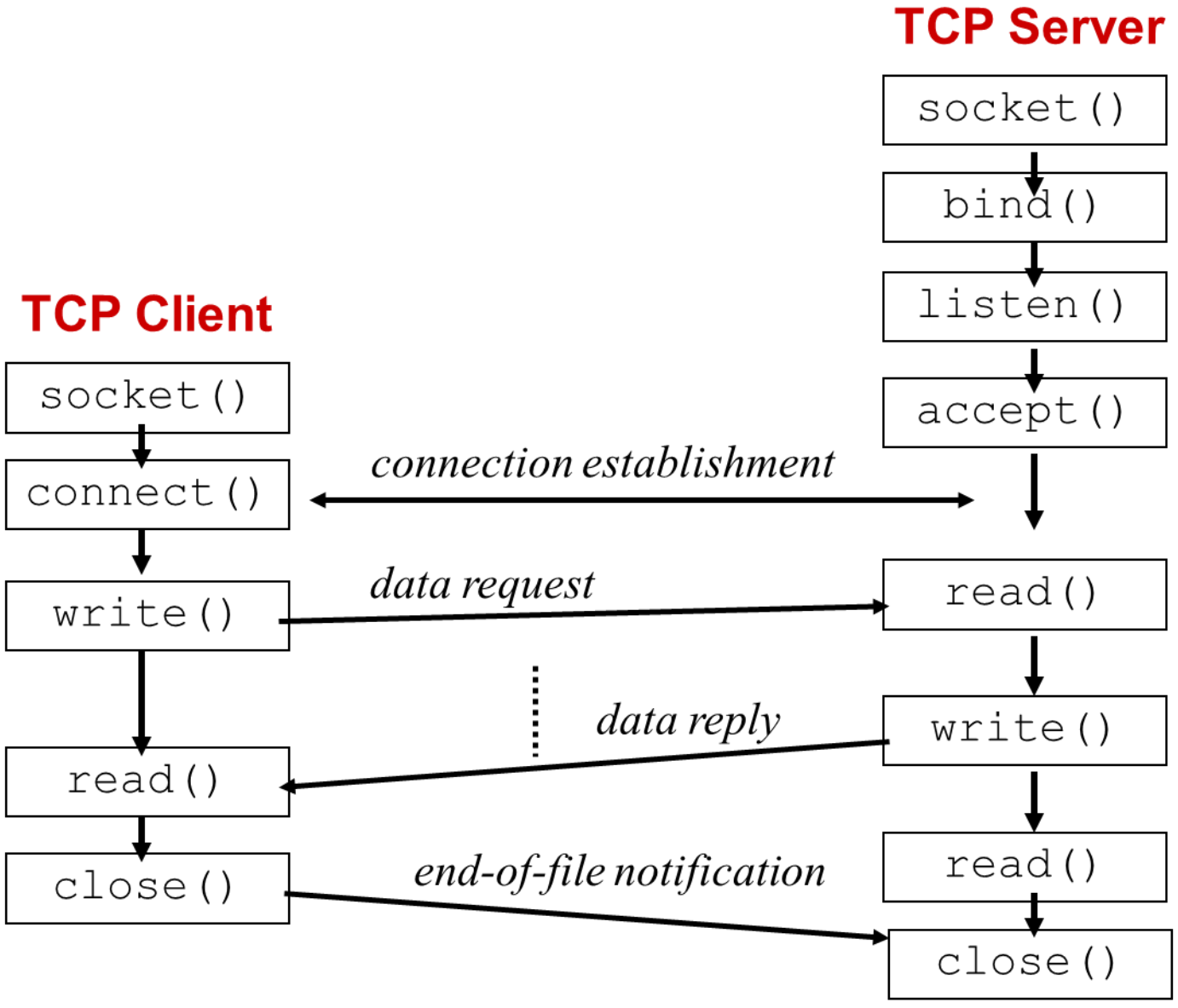
Read Data

- read blocks waiting for data from the client but does not guarantee that sizeof(buf) is read

```
int read(int fd, void *buf, size_t count);
```

```
int fd;                          /* socket descriptor */
char buf[512];                   /* used by read() */
int nbytes;                      /* used by read() */

/* 1) create the socket */
/* 2) bind the socket to a port */
/* 3) listen on the socket */
/* 4) accept the incoming connection */

if((nbytes = read(newfd, buf, sizeof(buf))) < 0) {
      perror("read"); exit(1);
}
```

Client


TCP Server
socket()
bind()
listen()
accept()
TCP Client
socket()
connect()
connection establishment
write()
data request
read()
write()
data reply
read()
read()
close()
end-of-file notification
close()

Dealing with IP Addresses

- IP Addresses are commonly written as strings (128.113.26.47), but programs deal with IP addresses as integers.

Converting strings to numerical address:

```
struct sockaddr_in srv;

srv.sin_addr.s_addr = inet_addr("128.2.35.50");
if(srv.sin_addr.s_addr == (in_addr_t) -1) {
      fprintf(stderr, "inet_addr failed!\n");
      exit(1);
}
```

Converting a numerical address to a string:

```
struct sockaddr_in srv;
char *t = inet_ntoa(srv.sin_addr);
if(t == 0) {
      fprintf(stderr, "inet_ntoa failed!\n");
      exit(1);
}
```

Dealing with IP Addresses

- gethostbyname provides interface to DNS
- Additional useful calls
  - gethostbyaddr : returns hostent given sockaddr_in
  - getservbyname: used to get service description (typically port number)

```
#include <netdb.h>

struct hostent *hp; /*ptr to host info for remote*/
struct sockaddr_in peeraddr;
char *name = "www.nus.edu.sg";

peeraddr.sin_family = AF_INET;
hp = gethostbyname(name)
peeraddr.sin_addr.s_addr = ((struct in_addr*)(hp->h_addr))->s_addr;
```

Connect to the Server

- connect() allows a client to connect to a server

```
int connect(int sockfd, const struct sockaddr
*addr, socklen_t addrlen);
```

```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;          /* used by connect() */

/* create the socket */

/* connect: use the Internet address family, and connect to
   port 80 on IP address "128.2.35.50" */
srv.sin_family = AF_INET;
srv.sin_port = htons(80);
srv.sin_addr.s_addr = inet_addr("128.2.35.50");

if(connect(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
      perror("connect"); exit(1);
}
```

Write the Data

- write() can be used with a socket

```
ssize_t write(int fd, const void *buf, size_t
count);
```

```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;          /* used by connect() */
char buf[512];                   /* used by write() */
int nbytes;                      /* used by write() */

/* 1) create the socket */
/* 2) connect() to the server */

/* Example: A client could "write" a request to a server */
if((nbytes = write(fd, buf, sizeof(buf))) < 0) {
      perror("write");
      exit(1);
}
```

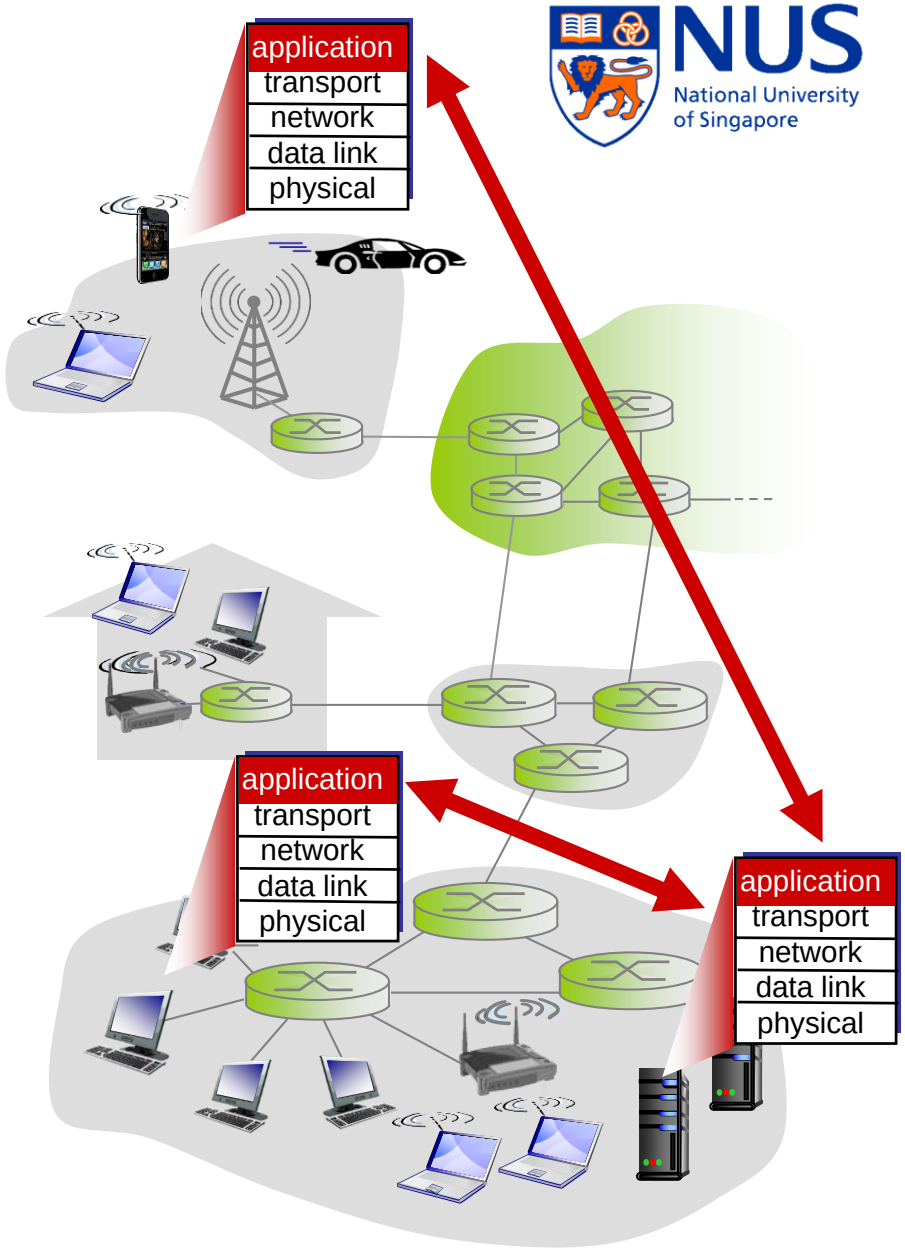
Creating a network app

write programs that:

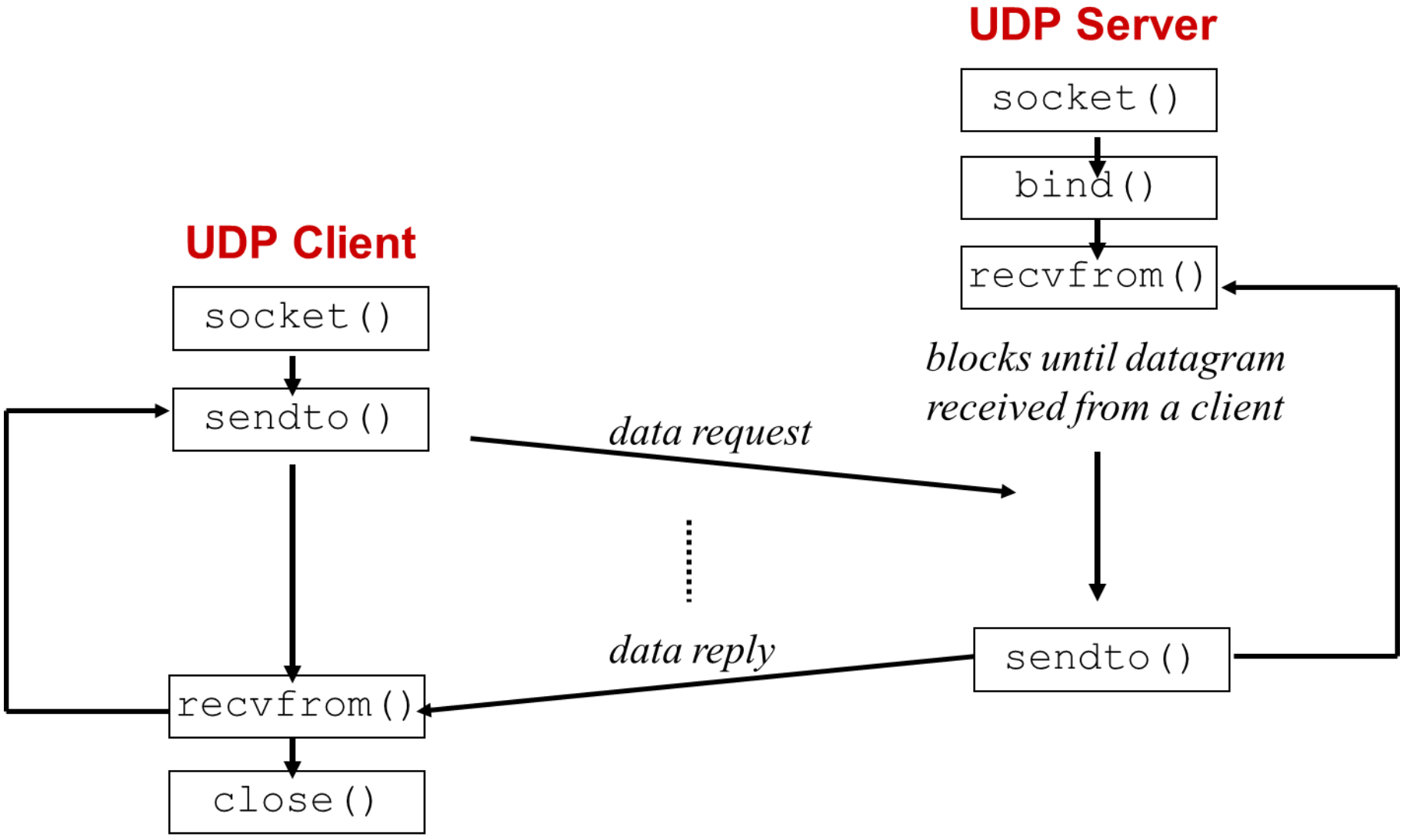
- run on (different) *end systems*
- communicate over network
- e.g., web server software communicates with browser software

no need to write software for network-core devices

- network-core devices do not run user applications
- applications on end systems allows for rapid app development, propagation

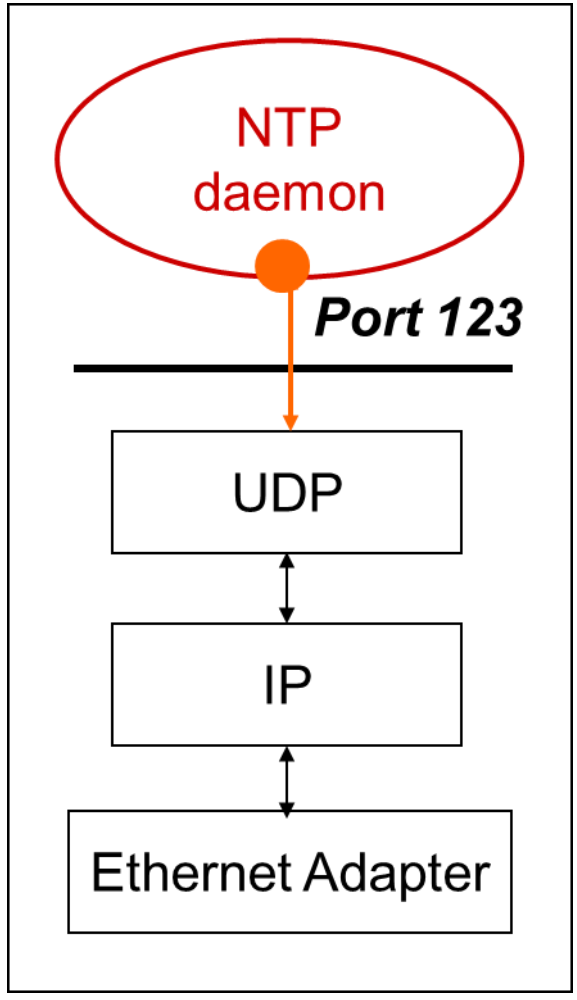


UDP Socket Programming



UDP: Server

- What does a UDP server need to do so that a UDP client can connect to it?



Example: NTP daemon

Create a socket

- The UDP server must create a datagram socket
  - SOCK_DGRAM: selects the UDP protocol

```
int fd;             /* socket descriptor */

if((fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
      perror("socket");
      exit(1);
}
```

Bind the socket

```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;          /* used by bind() */

/* create the socket */

/* bind: use the Internet address family */
srv.sin_family = AF_INET;

/* bind: socket 'fd' to port 80*/
srv.sin_port = htons(80);

/* bind: a client may connect to any of my addresses */
srv.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
      perror("bind"); exit(1);
}
```

Read Data

- read() does not provide the client's address to the UDP server

```
ssize_t recvfrom(int sockfd, void *buf, size_t len,
int flags, struct sockaddr *src_addr, socklen_t
*addrlen);
```

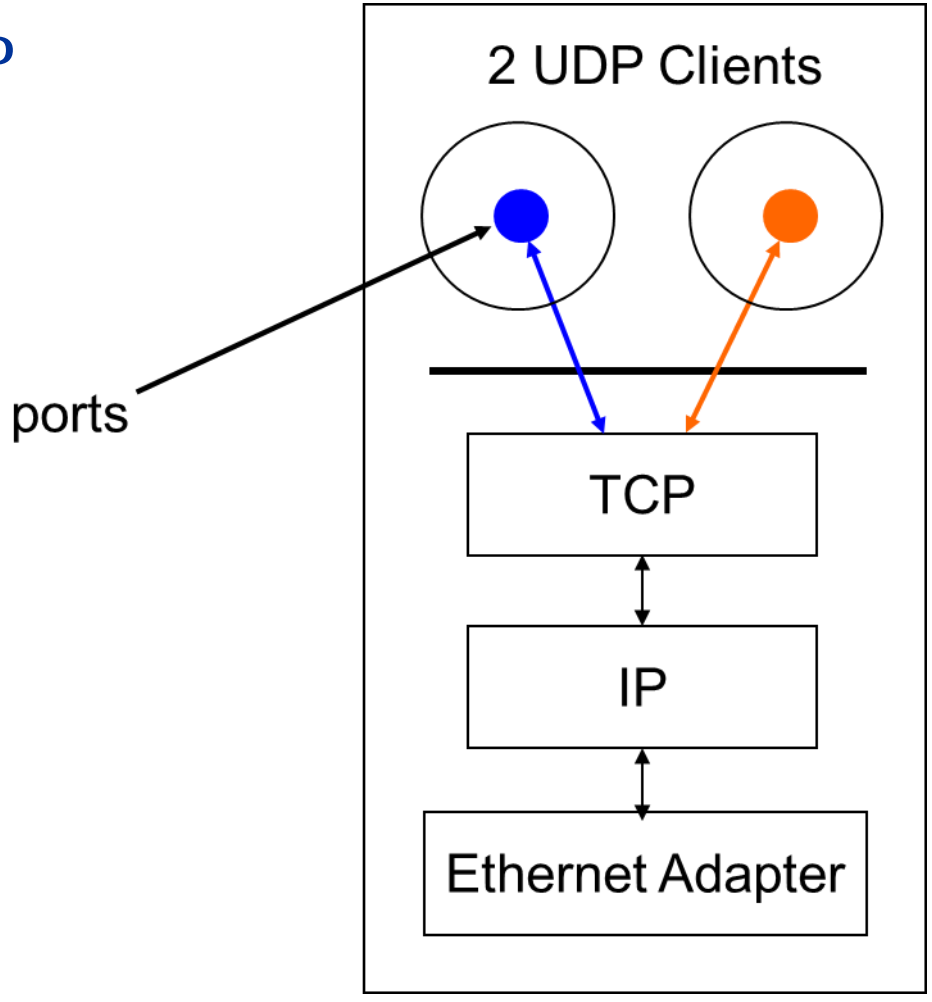
```
int fd;                                  /* socket descriptor */
struct sockaddr_in srv;                  /* used by bind() */
struct sockaddr_in cli;                  /* used by recvfrom() */
char buf[512];                           /* used by recvfrom() */
int cli_len = sizeof(cli);               /* used by recvfrom() */
int nbytes;                              /* used by recvfrom() */

/* 1) create the socket */
/* 2) bind to the socket */

nbytes = recvfrom(fd, buf, sizeof(buf), 0, (struct
            sockaddr*) &cli, &cli_len);
if(nbytes < 0) {
     perror("recvfrom"); exit(1);
}
```

Read Data

- nbytes = recvfrom(fd, buf, sizeof(buf), 0, (struct sockaddr*) &cli, &cli_len);
- The actions performed by recvfrom
  - returns the number of bytes read (nbytes)
  - copies nbytes of data into buf
  - returns the address of the client (cli)
  - returns the length of cli (cli_len)

UDP: Client

- How does a UDP client communicate with a UDP server?



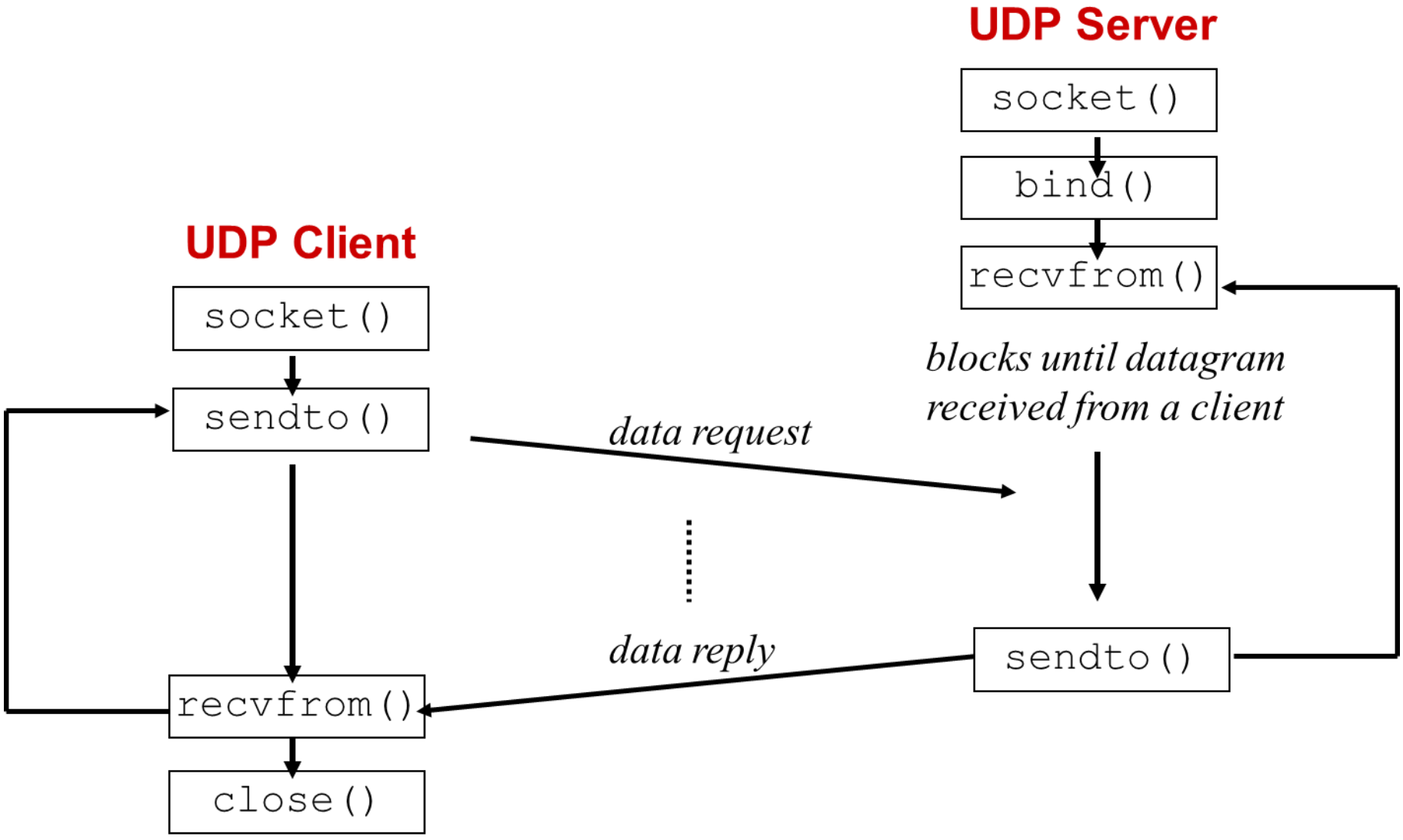
Send Data

- UDP client does not bind a port number: dynamically assigned when the first sendto() is called

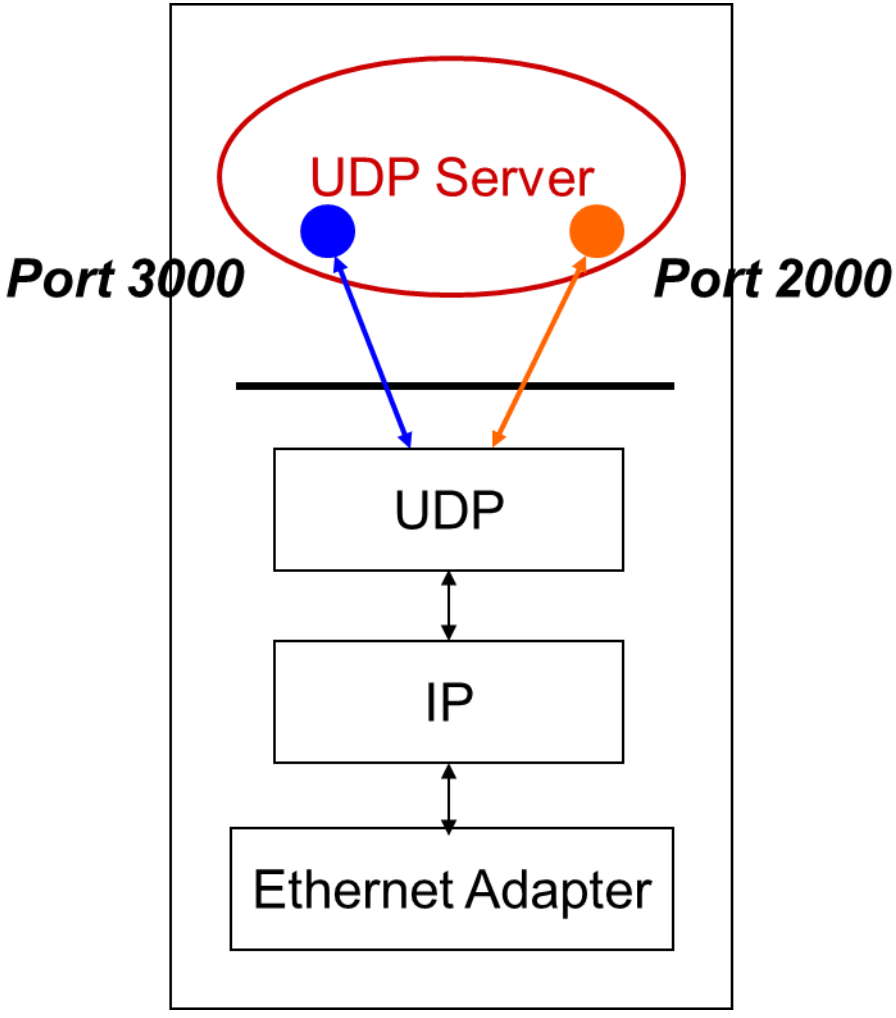
```
ssize_t sendto(int sockfd, const void *buf, size_t
len, int flags, const struct sockaddr *dest_addr,
socklen_t addrlen);
```

```
int fd;                              /* socket descriptor */
struct sockaddr_in srv;              /* used by sendto() */

/* 1) create the socket */

/* sendto: send data to IP Address "128.2.35.50" port 80 */
srv.sin_family = AF_INET;
srv.sin_port = htons(80);
srv.sin_addr.s_addr = inet_addr("128.2.35.50");

nbytes = sendto(fd, buf, sizeof(buf), 0 /* flags */,
                (struct sockaddr*) &srv, sizeof(srv));
if(nbytes < 0) {
      perror("sendto"); exit(1);
}
```

UDP Socket Programming



UDP Server: Multiple Connections



UDP Server: Multiple Connections


```
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?

UDP Server: Multiple Connections


```
int select(int maxfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

FD_CLR(int fd, fd_set *fds);   /* clear the bit for fd in fds */
FD_ISSET(int fd, fd_set *fds); /* is the bit for fd in fds? */
FD_SET(int fd, fd_set *fds);   /* turn on the bit for fd in fds */
FD_ZERO(fd_set *fds);          /* clear all bits in fds */
```

- maxfds: number of descriptors to be tested
  - descriptors (0, 1, ... maxfds-1) will be tested
- readfds: a set of fds we want to check if data is available
  - returns a set of fds ready to read
  - if input argument is NULL, not interested in that condition
- writefds: returns a set of fds ready to write
- exceptfds: returns a set of fds with exception conditions

UDP Server: Multiple Connections


```
int select(int maxfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

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

- timeout
  - if NULL, wait forever and return only when one of the descriptors is ready for I/O
  - otherwise, wait up to a fixed amount of time specified by timeout
- if we don't want to wait at all, create a timeout structure with timer value equal to 0

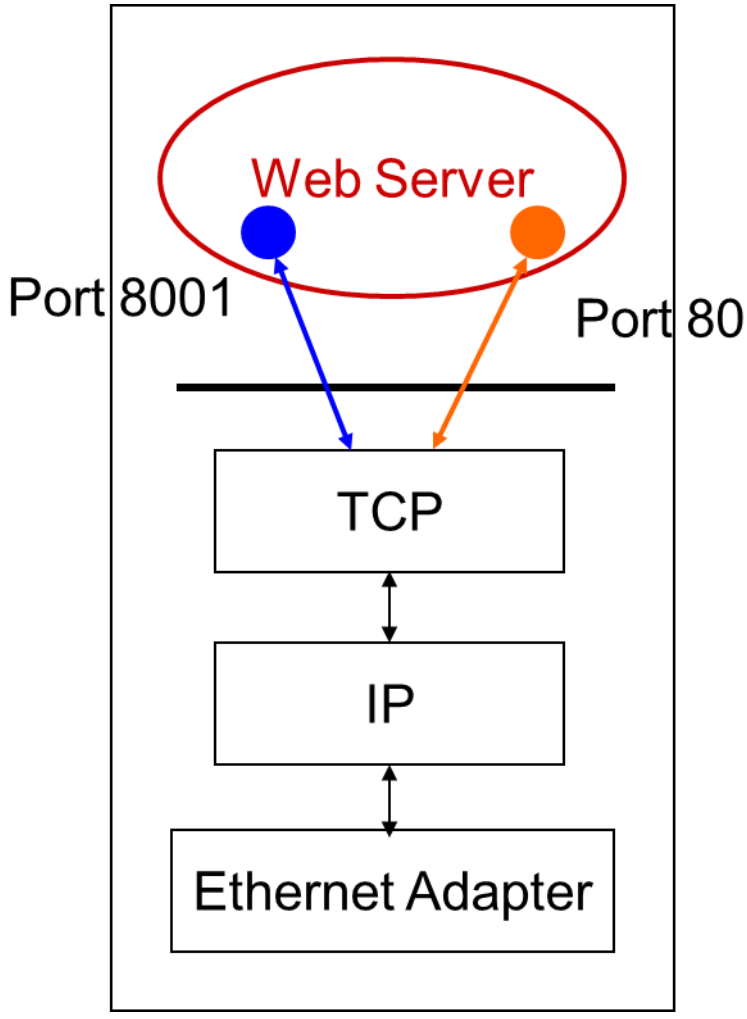
UDP Server: Multiple Connections

- Use select() for synchronous I/O multiplexing

```
int s1, s2;                 /* socket descriptors */
fd_set readfds;             /* used by select() */

/* create and bind s1 and s2 */
while(1) {
      FD_ZERO(&readfds);        /* initialize the fd set */
      FD_SET(s1, &readfds);     /* add s1 to the fd set */
      FD_SET(s2, &readfds);     /* add s2 to the fd set */

      if(select(s2+1, &readfds, 0, 0, 0) < 0) {
            perror("select");
            exit(1);
      }
      if(FD_ISSET(s1, &readfds)) {
            recvfrom(s1, buf, sizeof(buf), ...);
            /* process buf */
      }
      /* do the same for s2 */
}
```

Web Servers

- How can a web server manage multiple connections simultaneously?



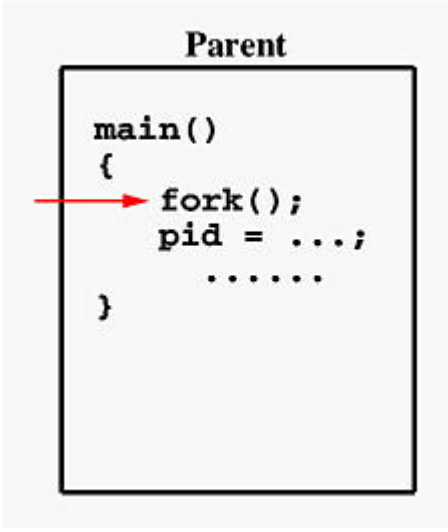
Web Server: Multiple Connections

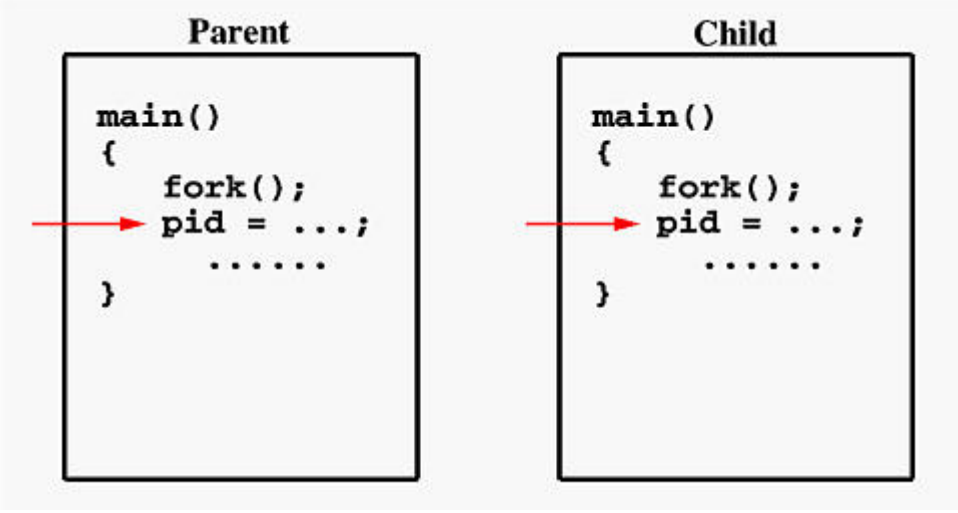
- Use select() for synchronous I/O multiplexing

```
int fd, next=0;                    /* original socket */
int newfd[10];                     /* new socket descriptors */
while(1) {
      fd_set readfds;
      FD_ZERO(&readfds); FD_SET(fd, &readfds);

      /* Now use FD_SET to initialize other newfd's
         that have already been returned by accept() */

      select(maxfd+1, &readfds, 0, 0, 0);
      if(FD_ISSET(fd, &readfds)) {
            newfd[next++] = accept(fd, ...);
      }
      /* do the following for each descriptor newfd[n] */
      if(FD_ISSET(newfd[n], &readfds)) {
            read(newfd[n], buf, sizeof(buf));
            /* process data */
      }
}
```

Concurrent Servers

- Use fork() for creating new (identical) processes





Concurrent Servers

- The fork() system call:

```
pid_t fork(void);
```

  - If fork() returns a negative value, the creation of a child process was unsuccessful.
  - fork() returns a zero to the newly created child process.
  - fork() returns a positive value, the process ID of the child process, to the parent.
  - The returned process ID is of type pid_t defined in sys/types.h.

Server Outline

```
pid_t pid;
int listenfd, connfd;

listenfd = socket(. . .);
bind(. . .);
listen(. . .);

while(1) {
      connfd = accept(listenfd, . . .);

      if ((pid = fork()) == 0) {
            close(listenfd);
            ...
            close(connfd);
            exit(0);
      }

      close(connfd);
}
```

Socket Programming References

- Man page
  - usage: man <function name>
- Textbook (Kurose and Ross)
  - Section 2.7
  - demo programs written in Python
- Unix Network Programming : Networking APIs: Sockets and XTI (Volume 1)
  - ultimate socket programming reference

TCP State Diagram