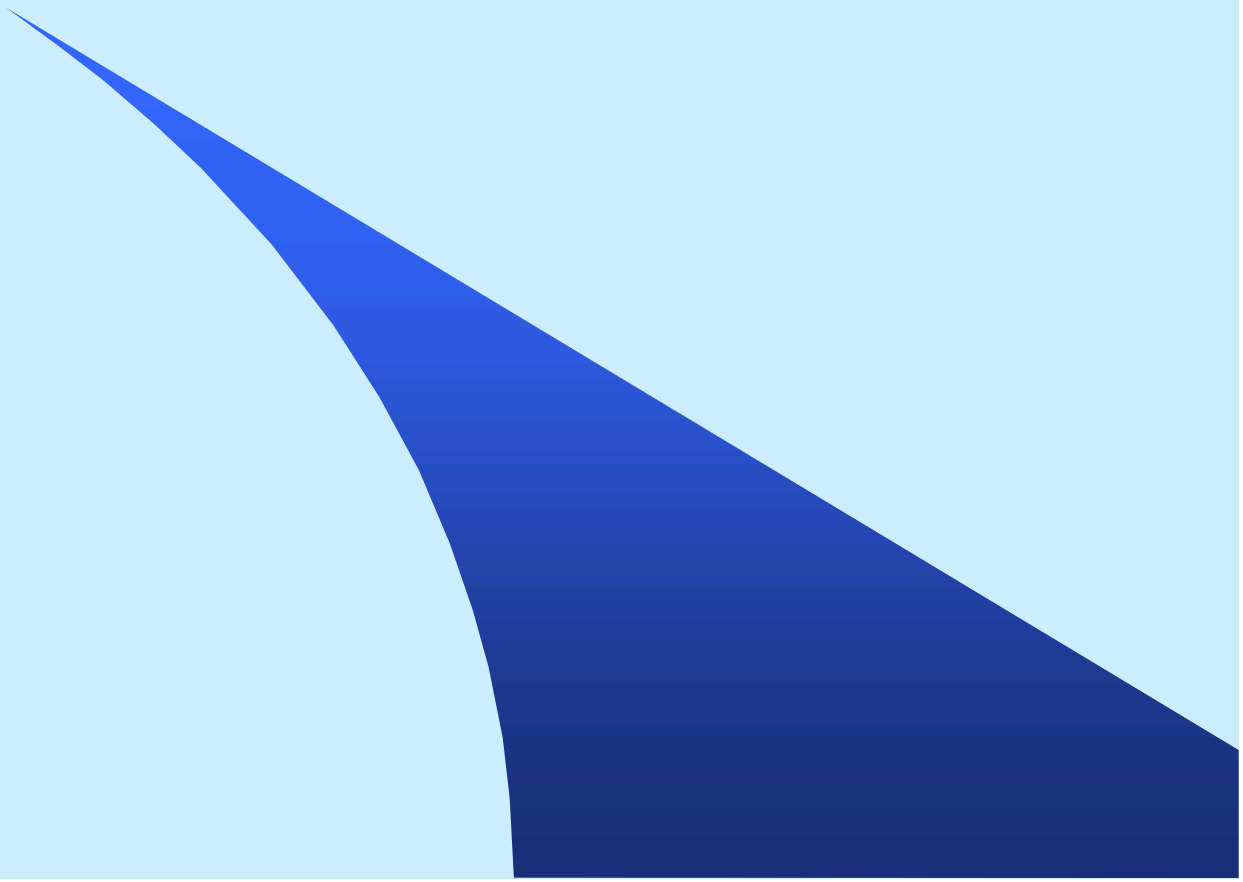
# UNIX System Programming

## Socket Programming



# Socket Programming

- What is a socket?
- Using sockets
  - Types (Protocols)
  - Associated functions
  - Styles

# What is a socket

## Socket API

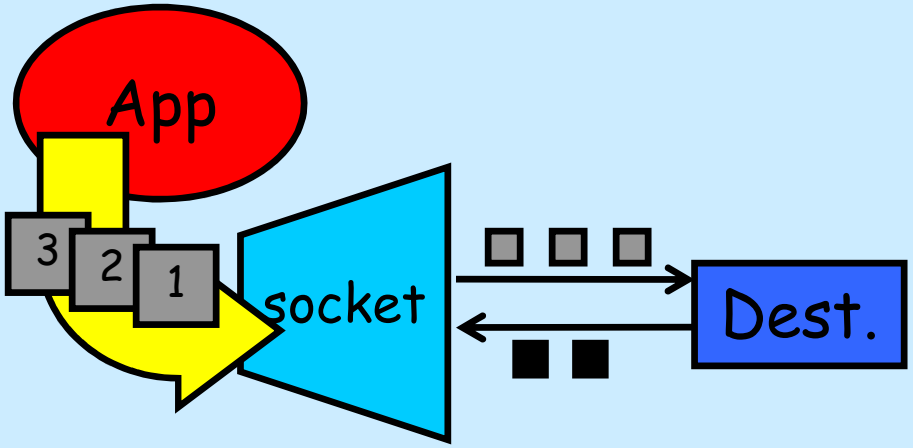
- introduced in BSD4.1 UNIX, 1981
- Two types of sockets
  - —connection-oriented
  - —connectionless

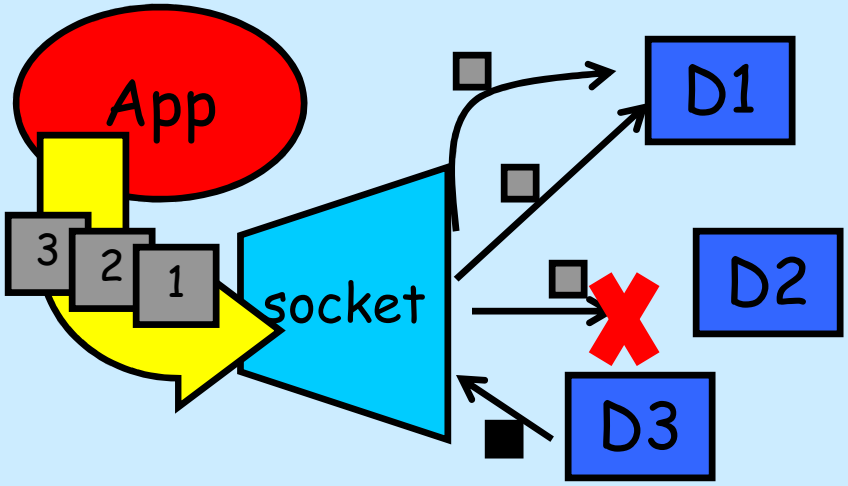
**socket**

an interface (a "door") into which one application process can both send and receive messages to/from another (remote or local) application process

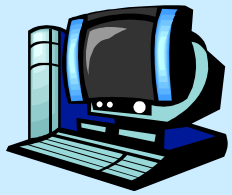
# Two essential types of sockets

- SOCK_STREAM
  - — a.k.a. TCP
  - — reliable delivery
  - — in-order guaranteed
  - — connection-oriented
  - — bidirectional
- SOCK_DGRAM
  - — a.k.a. UDP
  - — unreliable delivery
  - — no order guarantees
  - — no notion of "connection" – App indicates destination for each packet
  - — can send or receive



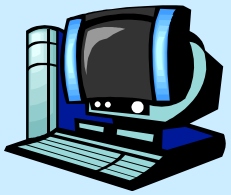



# A Socket-eye view of the Internet



loki.trentu.ca

(192.197.151.116)



www.google.com

(74.125.226.145)

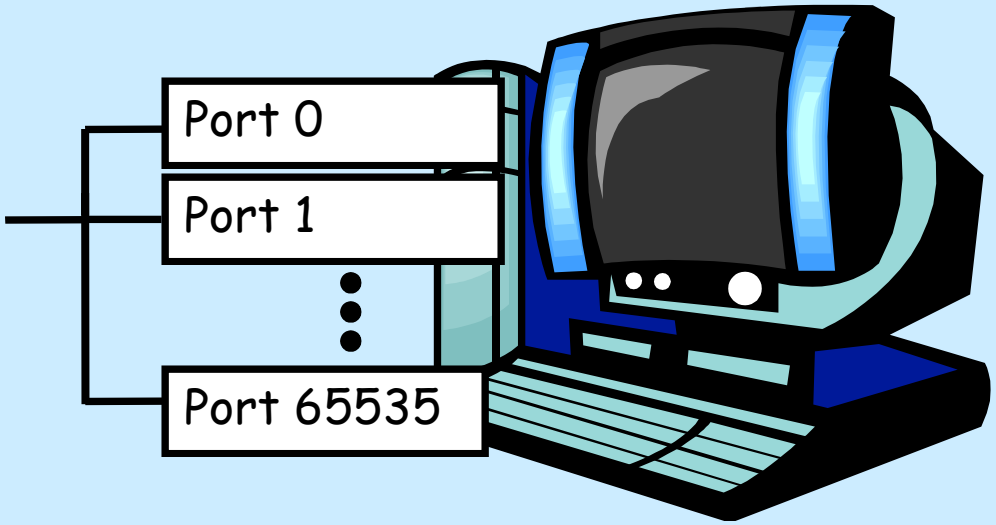
church.cse.ogi.edu

(129.95.50.2, 129.95.40.2)

- Each host machine has an IP address
- When a packet arrives at a host it contains information on which port it belongs to

# Ports

- Each host has 65,536 ports
- Some ports are *reserved for specific apps*
  - 20 & 21: FTP
  - 22: SSH
  - 23: Telnet
  - 25: SMTP
  - 53: DNS
  - 80: HTTP
  - 110: POP3
  - 143: IMAP
  - 443: HTTPS
  - 465: SMTPS




- A socket provides an interface to send data to/from the network through a port

# Addresses, Ports and Sockets

- Like apartments and mailboxes
  - You are the Application
  - Your home address is the IP address
  - Your mailbox is the port
  - The Post Office is the network
- Q: How do you choose which port to connects a socket?

# Socket Creation in C: socket

- int s = socket(domain, type, protocol);
  - s: socket descriptor, an integer (like a file descriptor)
  - domain: integer, communication domain
    - AF_INET (IPv4 protocol) – typically used
    - AF_INET6 (IPv6 protocol)
    - AF_UNIX or AF_LOCAL – intra-machine communication
  - type: communication type
    - SOCK_STREAM: reliable, 2-way, connection-based service
    - SOCK_DGRAM: unreliable, connectionless,
    - other values: need root permission, rarely used, or obsolete
  - protocol: specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family, in which case protocol can be specified as 0.
- NOTE: socket call does not specify where data will be coming from, nor where it will be going to – it just creates the interface!

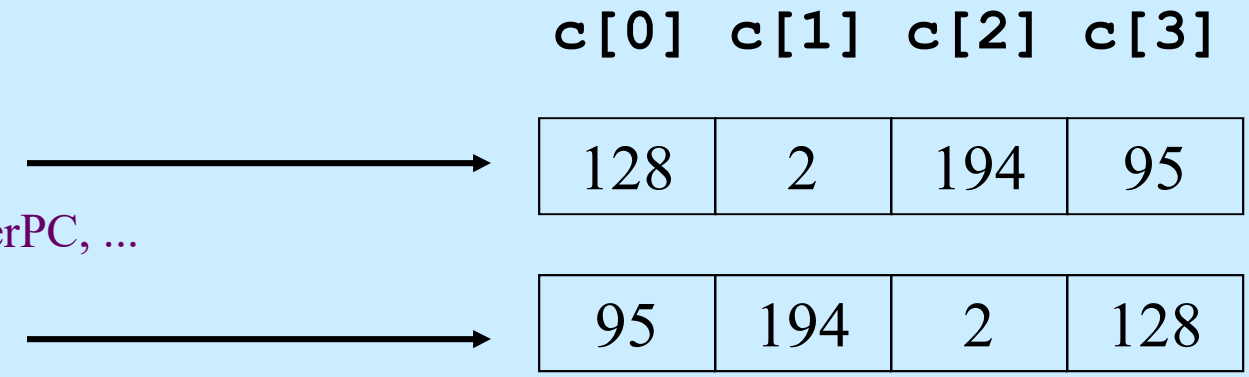
# Internet Addressing Data Structure

```
#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

```
union {
    u_int32_t addr;   /* 4 bytes address */
    char c[4];
} un;
/* 128.2.194.95 */
un.addr = 0x8002c25f;
/* c[0] = ? */
```



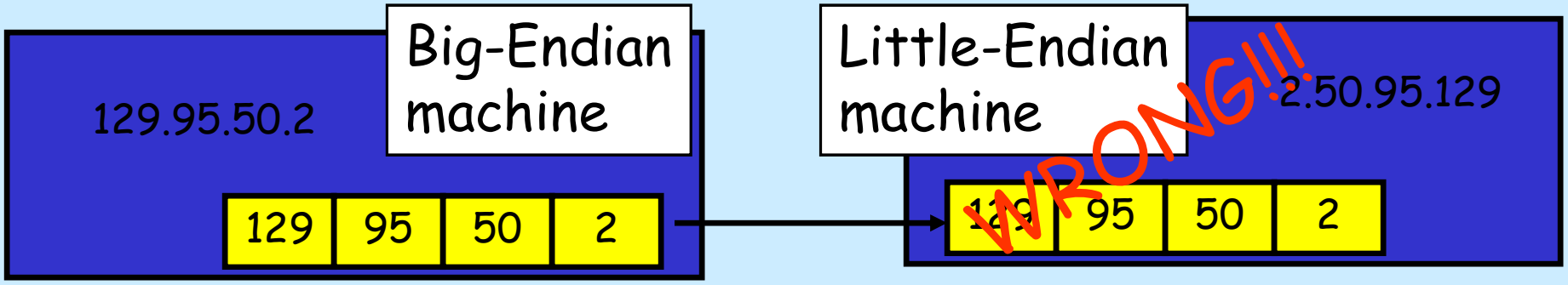

- Big Endian
  - —Sun Solaris, PowerPC, ...
- Little Endian
  - —i386, alpha, ...
- Network byte order = Big Endian

# Address and port byte-ordering

- Address and port are stored as integers
  - u_short sin_port; (16 bit)
  - in_addr sin_addr; (32 bit)

```
struct in_addr {
  u_long s_addr;
};
```

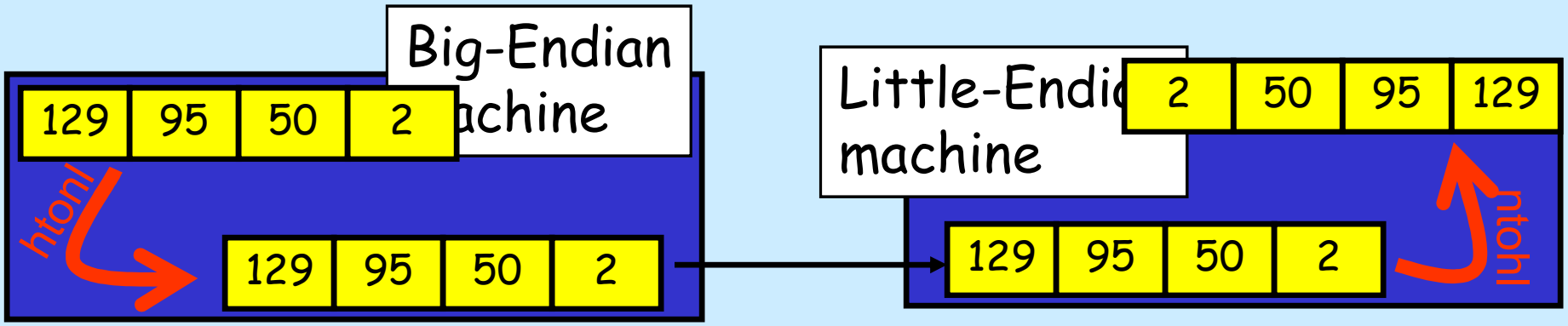
- Problem:
  - different machines / OS's use different word orderings
    - little-endian: lower bytes first
    - big-endian: higher bytes first
  - these machines may communicate with one another over the network



# UNIX's byte-ordering funcs

- u_long htonl(u_long x);
- u_short htons(u_short x);
- u_long ntohl(u_long x);
- u_short ntohs(u_short x);

- On big-endian machines, these routines do nothing
- On little-endian machines, they reverse the byte order



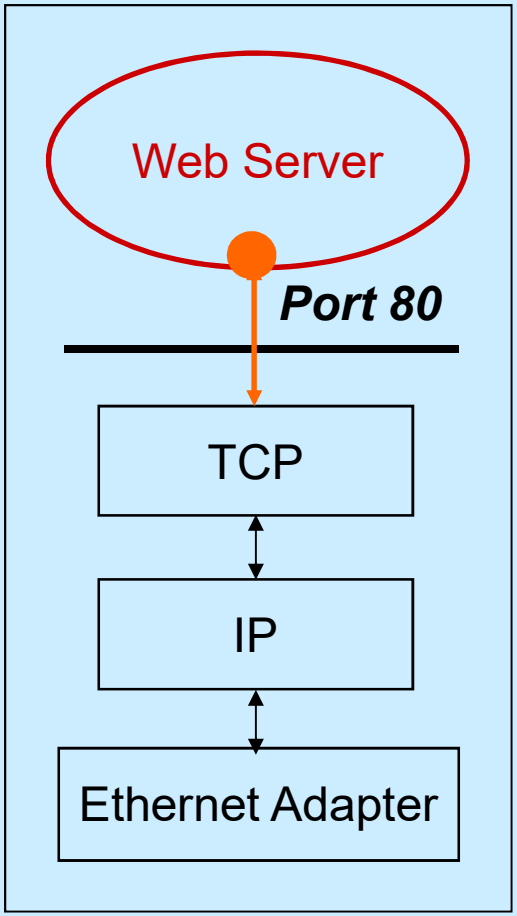

- Same code would have worked regardless of endian-ness of the two machines

# Byte Ordering Functions

- 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);
```

# TCP Server




- For example: web server
- **What does a *web server* need to do so that a *web client* can connect to it?**

# Socket I/O: socket()

- Since web traffic uses TCP, the web server must create a socket of type SOCK_STREAM (connection-oriented)

```
int socketd;              /* socket descriptor */

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

- ***socket*** returns an integer (**socket descriptor**)
  - —***socket*** < 0 indicates that an error occurred
- **AF_INET** associates a socket with the Internet protocol family
- **SOCK_STREAM selects the TCP protocol**
- **In some Unixes, need to compile with –lsocket -lnsl to link in socket libraries**

# Socket I/O: bind()

- A ***socket*** can be bound to a ***port***

```
int socketd;                          /* 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 'socketd' to port 80*/

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

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

- **Still not quite ready to communicate with a client...**

# Socket I/O: listen()

- ***listen*** indicates that the server will accept a connection

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

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

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

- **Still not quite ready to communicate with a client...**

# Socket I/O: accept()

- ***accept*** blocks waiting for a connection

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

- ***accept*** returns a new socket (***newfd***) with the same properties as the original socket (`socketd`)
  - —***newfd*** < 0 indicates that an error occurred

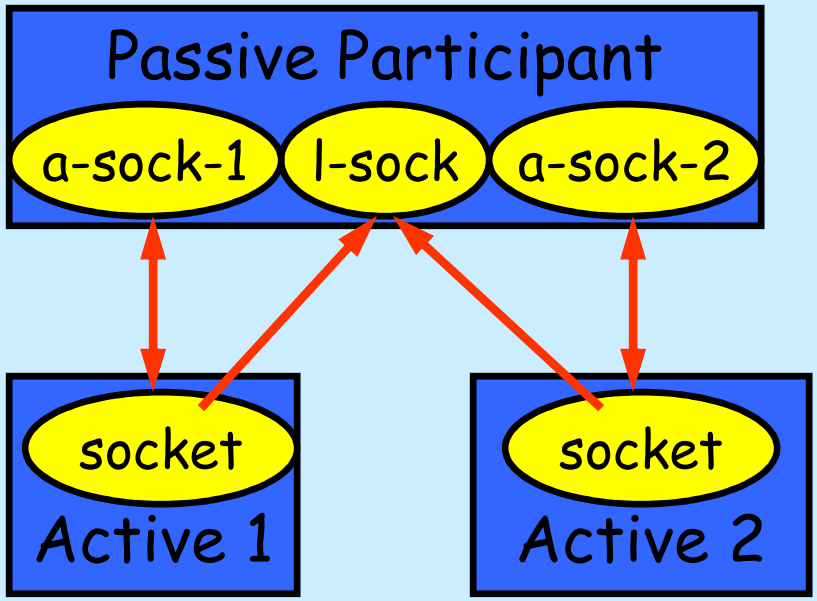
# Socket I/O: accept() continued...

```
struct sockaddr_in cli;          /* used by accept() */
int newfd;                       /* returned by accept() */
int cli_len = sizeof(cli);       /* used by accept() */

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

- How does the server know which client it is?
  - —**cli.sin_addr.s_addr** contains the client's ***IP address***
  - —**cli.sin_port** contains the client's ***port number***
- Now the server can exchange data with the client by using ***read*** and ***write*** on the descriptor ***newfd***.
- Why does ***accept*** need to return a new descriptor?

# Connection setup

- Passive participant - server
  - step 1: listen (for incoming requests)
  - step 3: accept (a request)
  - step 4: data transfer
- Active participant - clients
  - step 2: request & establish connection
  - step 4: data transfer
- The accepted connection is on a new socket
- The old socket continues to listen for other active participants



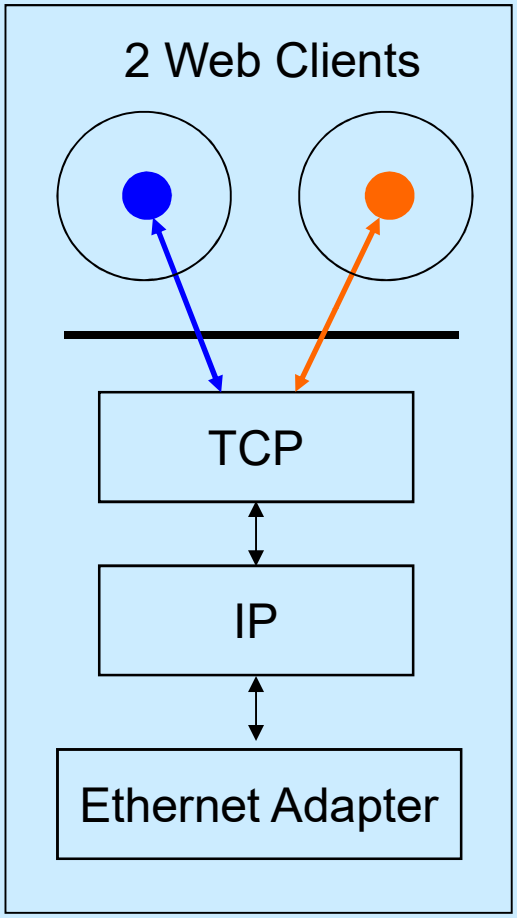
# Socket I/O: read()

- ***read*** can be used with a socket
- ***read*** **blocks waiting for data from the client but does not guarantee that sizeof(buf) is read**

```
int socketd;                          /* 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);
}
```

# TCP Client

- For example: web client
- **How does a *web client* connect to a *web server*?**



# Dealing with IP Addresses

- IP Addresses are commonly written as strings (“128.192.35.50”), but programs deal with IP addresses as integers.

**Converting strings to numerical address:**

```
struct sockaddr_in srv;

srv.sin_addr.s_addr = inet_addr(“128.192.35.50”);
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);
}
```

# Translating Names to Addresses

- `Gethostbyname` provides interface to DNS
- Additional useful calls
  - `Gethostbyaddr` – returns `hostent` given sockaddr_in
  - `Getservbyname`
    - Used to get service description (typically port number)
    - Returns `servent` based on name

```
#include <netdb.h>

struct hostent *hp; /*ptr to host info for remote*/
struct sockaddr_in peeraddr;
char *name = "odin.trentu.ca";

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

# Socket I/O: connect()

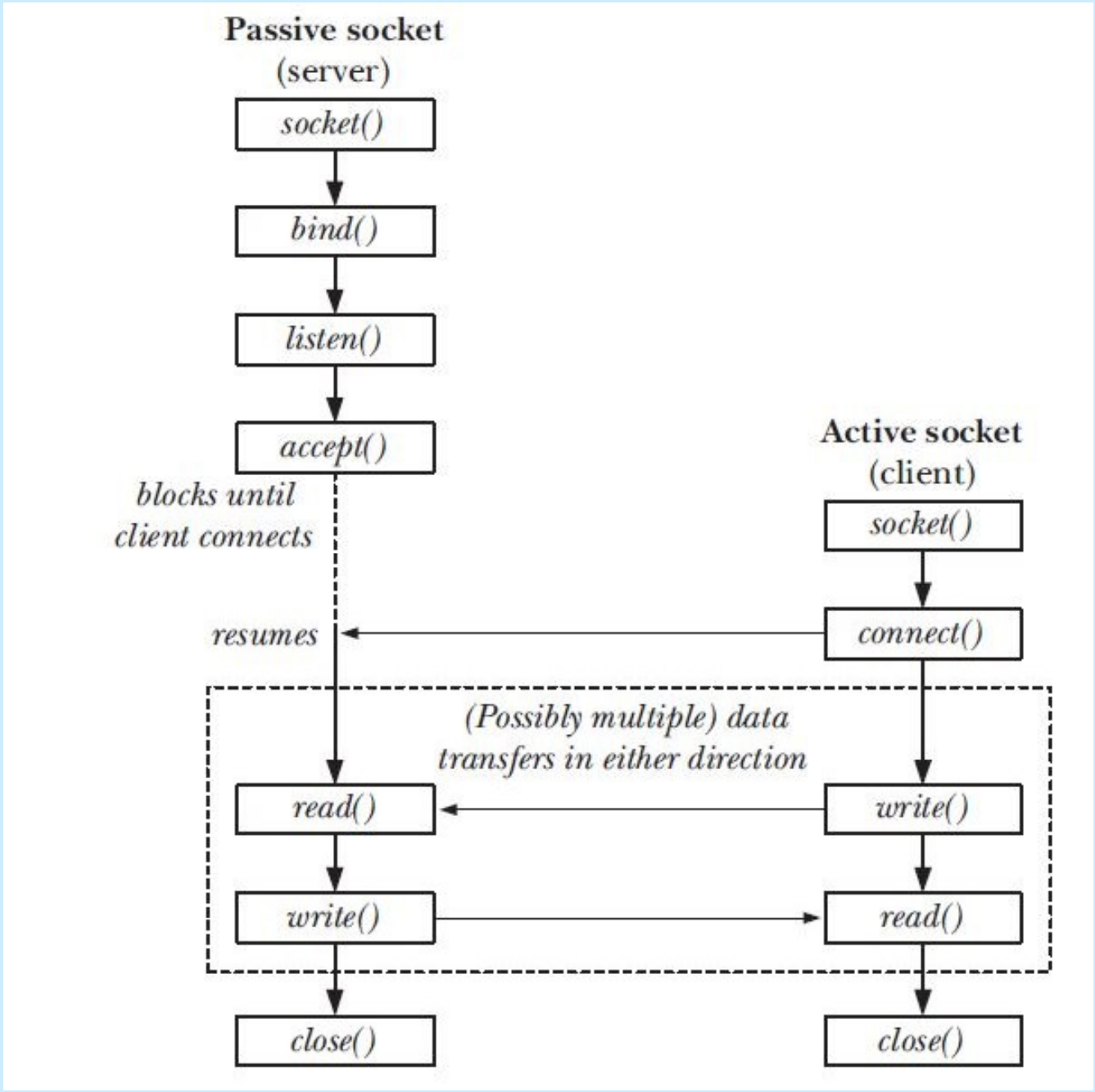
- ***connect*** allows a client to connect to a server...

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

/* create the socket */

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

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

/* connect: connect to IP Address "192.197.151.70" */
srv.sin_addr.s_addr = inet_addr("192.197.151.70");

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

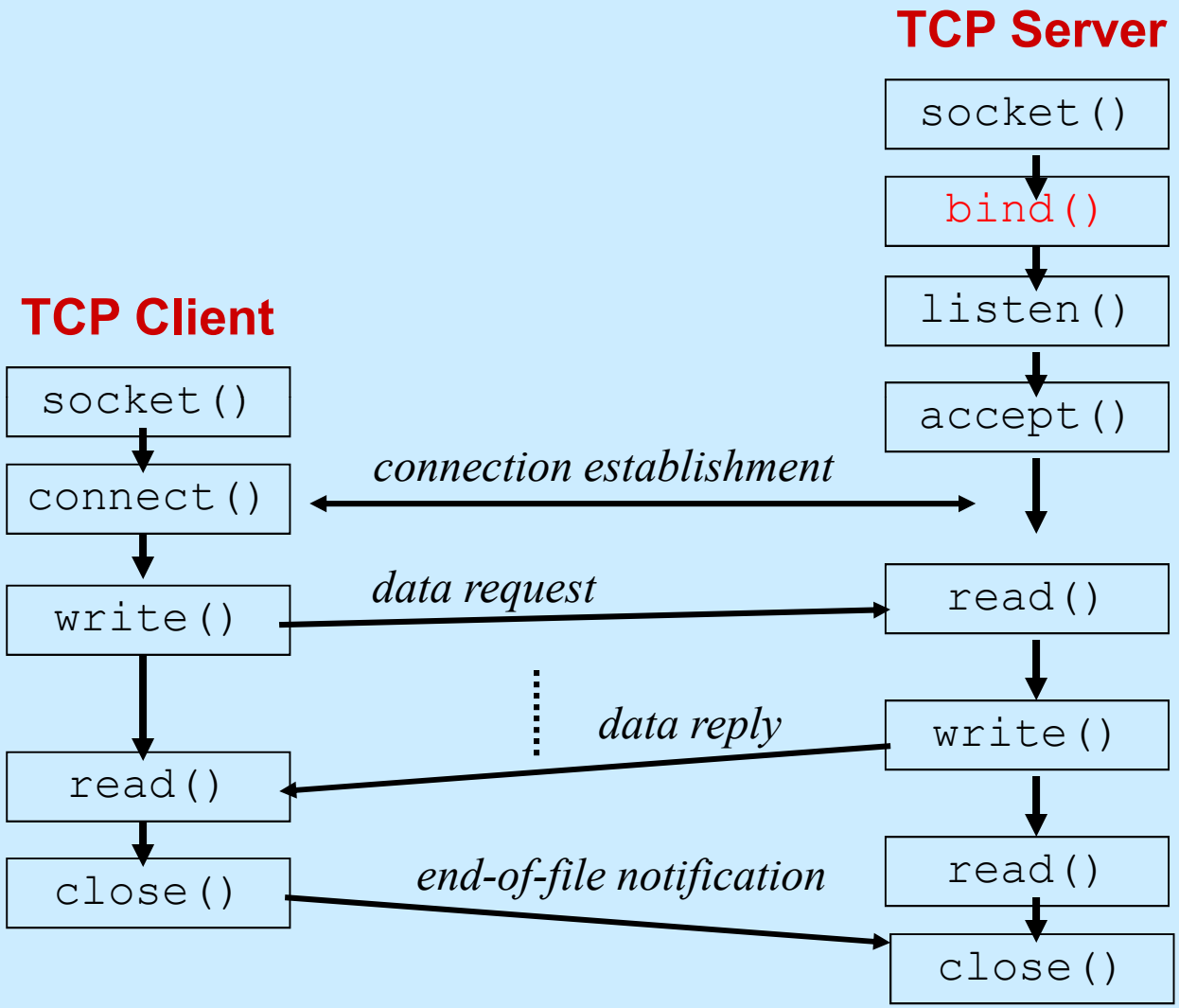
# Socket I/O: write()

- ***write*** can be used with a socket

```
int socketd;                    /* 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(socketd, buf, sizeof(buf))) < 0) {
      perror("write");
      exit(1);
}
```


Passive socket
(server)
socket()
bind()
listen()
accept()
blocks until
client connects
resumes
Active socket
(client)
socket()
connect()
(Possibly multiple) data
transfers in either direction
read()
write()
write()
read()
close()
close()

# Review: TCP Client-Server Interaction



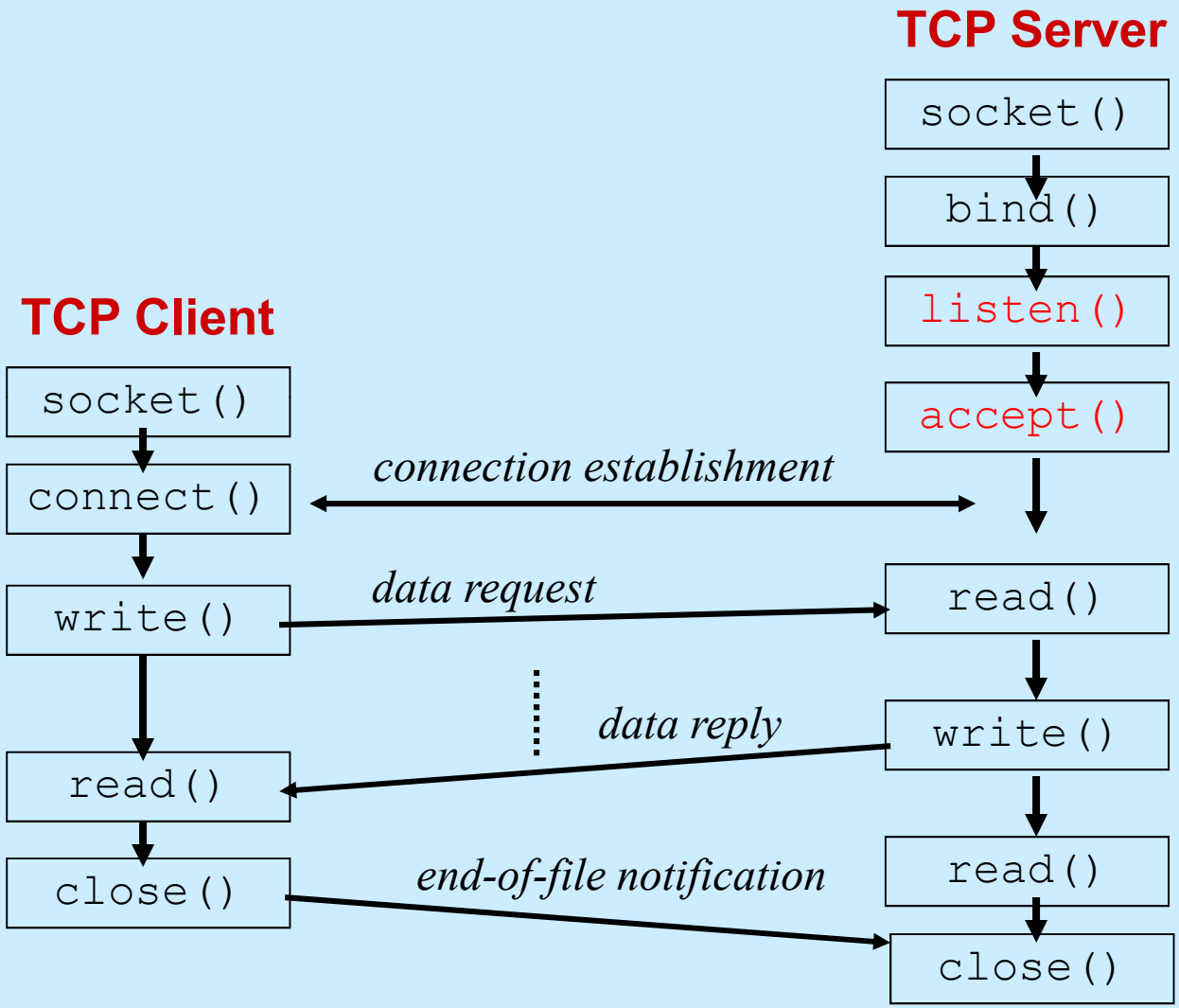
# The *bind* function

- associates and (can exclusively) reserves a port for use by the socket
- int status = bind(sockid, &addrport, size);
  - status: error status, = -1 if bind failed
  - sockid: integer, socket descriptor
  - addrport: struct sockaddr, the (IP) address and port of the machine (address usually set to INADDR_ANY – chooses a local address)
  - size: the size (in bytes) of the addrport structure
- bind can be skipped for both types of sockets.

# Skipping the *bind*

- SOCK_DGRAM:
  - if only sending, no need to bind. The OS finds a port each time the socket sends a packet
  - if receiving, need to bind
- SOCK_STREAM:
  - destination determined during connection setup
  - don't need to know port sending from (during connection setup, receiving end is informed of port)

# Review: TCP Client-Server Interaction

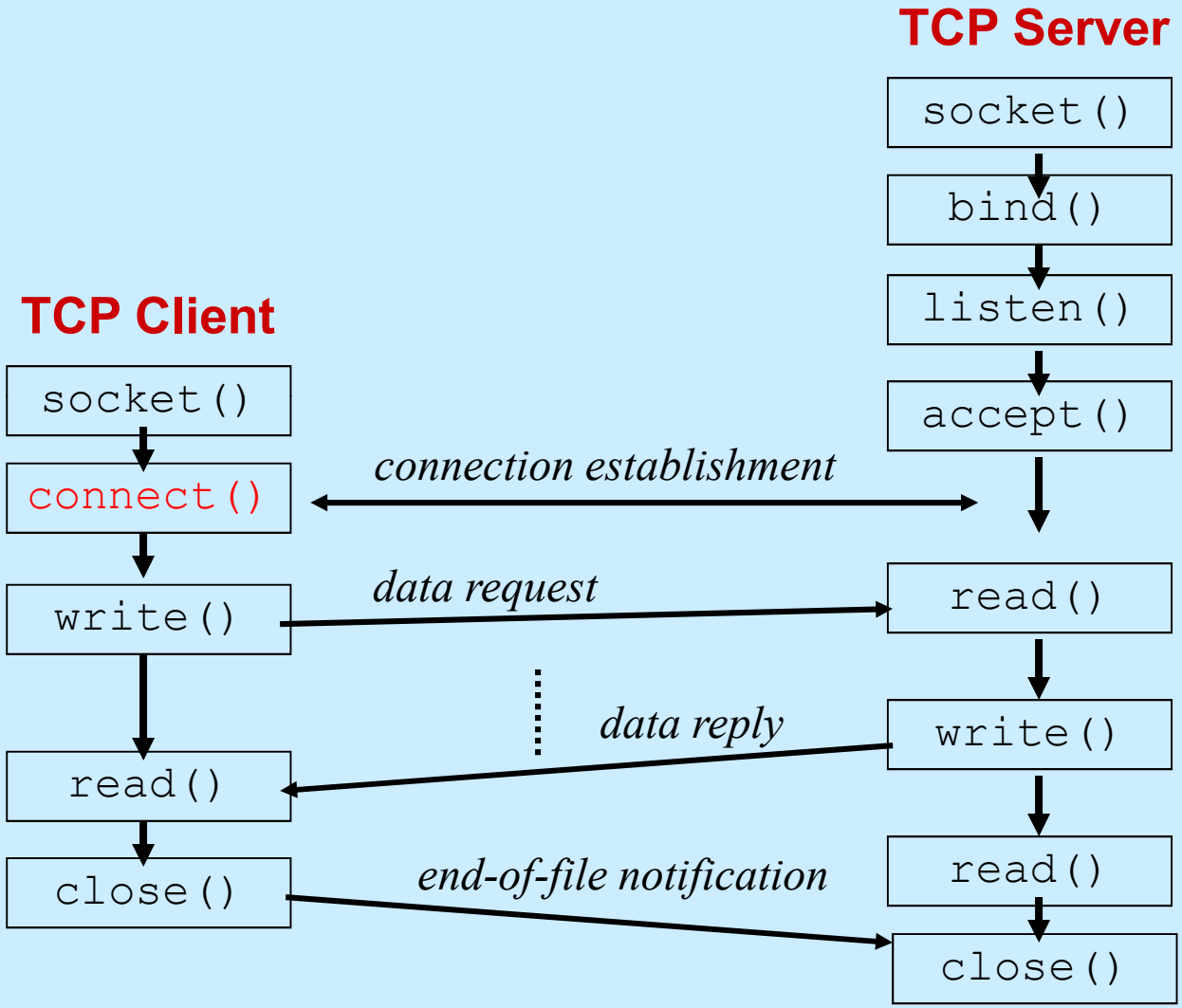


# Connection setup: *listen & accept*

- Called by passive participant
- int status = listen(sock, queuelen);
  - status: 0 if listening, -1 if error
  - sock: integer, socket descriptor
  - queuelen: integer, # of active participants that can "wait" for a connection
  - listen is **non-blocking**: returns immediately

- int s = accept(sock, &name, &namelen);
  - s: integer, the new socket (used for data-transfer)
  - sock: integer, the orig. socket (being listened on)
  - name: struct sockaddr, address of the active participant
  - namelen: sizeof(name): value/result parameter
    - must be set appropriately before call
    - adjusted by OS upon return
  - accept is **blocking**: waits for connection before returning

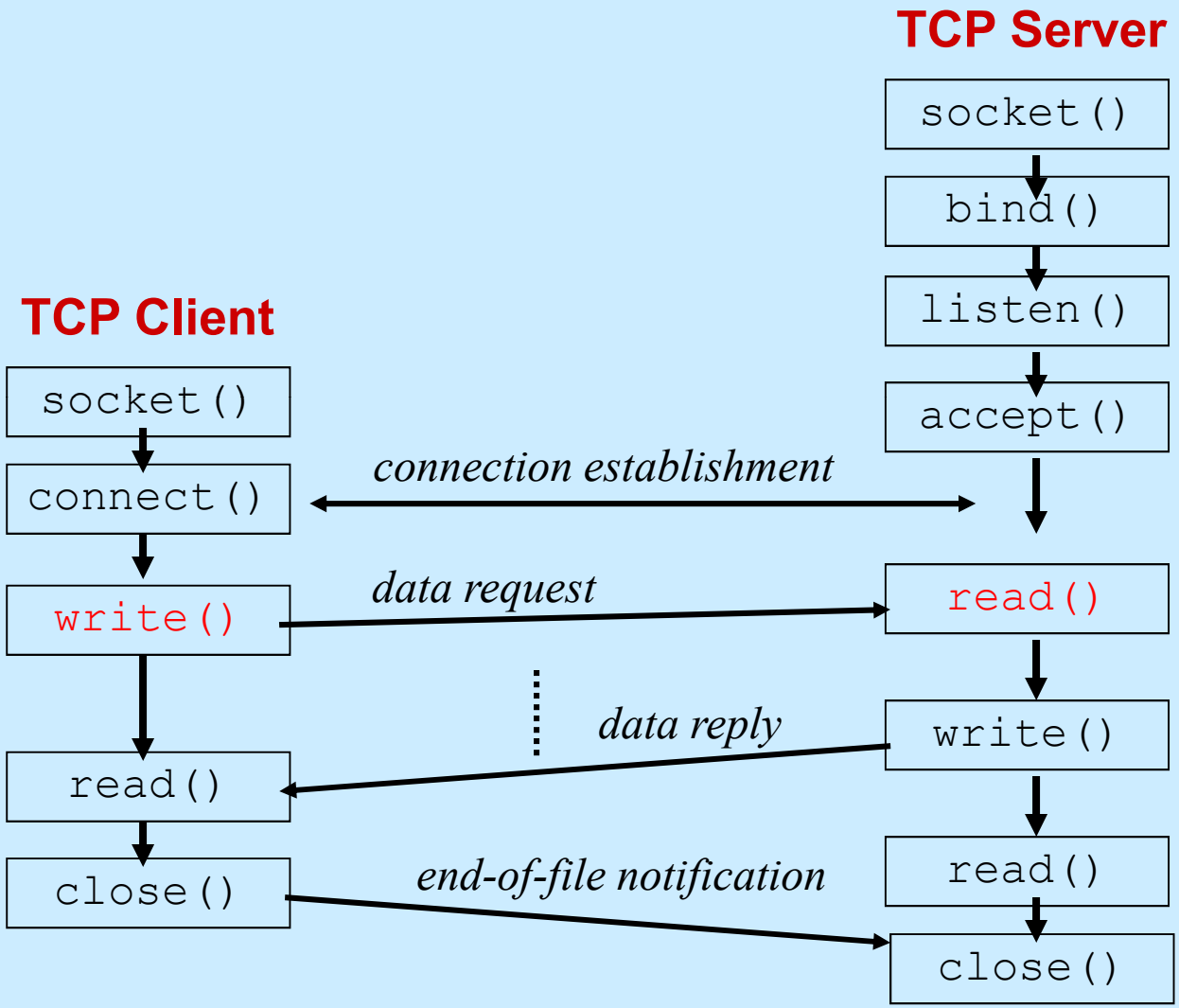
# Review: TCP Client-Server Interaction



# *connect* call

- int status = connect(sock, &name, namelen);
  - status: 0 if successful connect, -1 otherwise
  - sock: integer, socket to be used in connection
  - name: struct sockaddr: address of passive participant
  - namelen: integer, sizeof(name)
- connect is **blocking**

# Review: TCP Client-Server Interaction



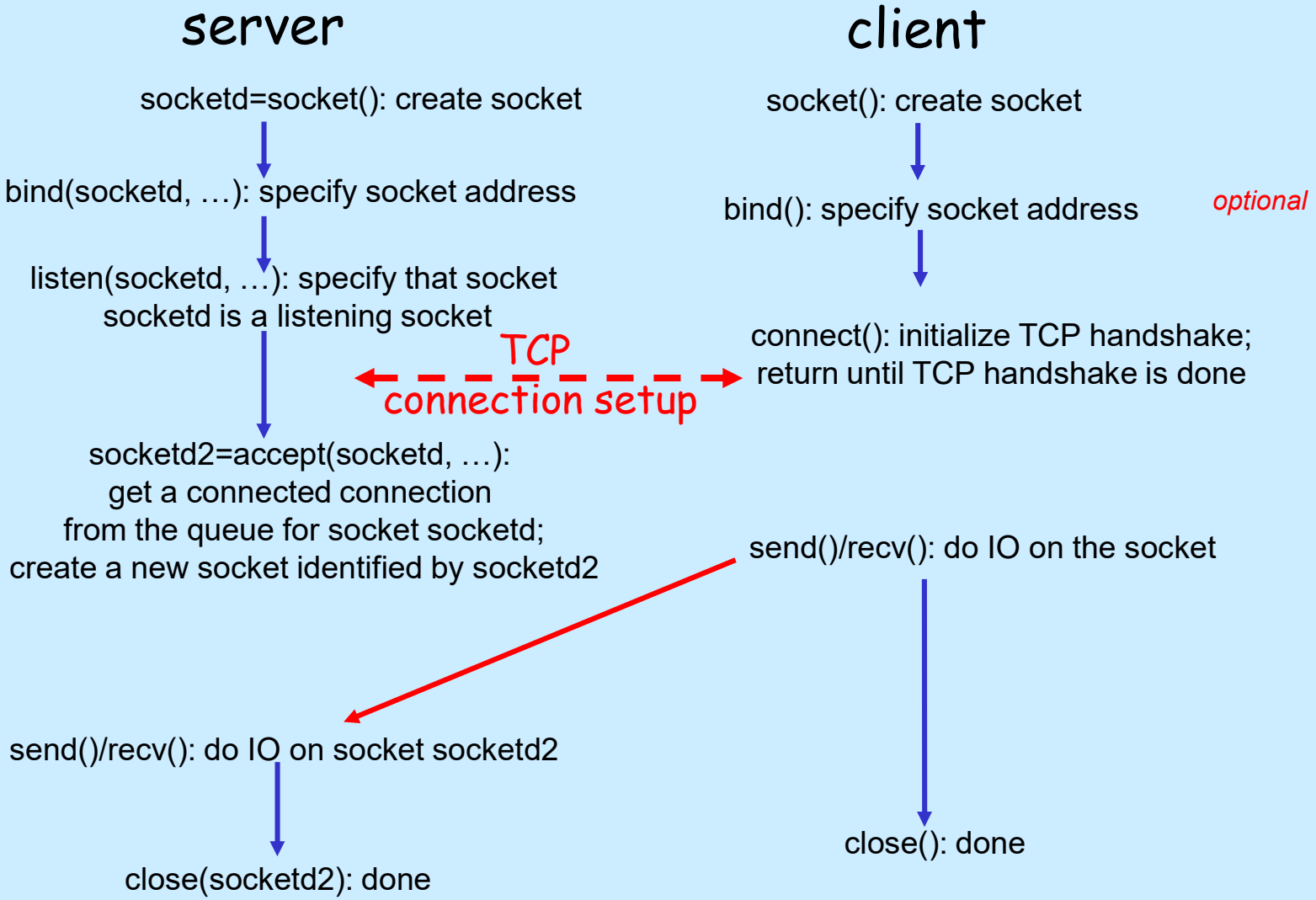
# Sending / Receiving Data

- With a connection (SOCK_STREAM):
  - Use send()/recv() instead of read()/write()
  - int count = send(sock, &buf, len, flags);
    - count: # bytes transmitted (-1 if error)
    - buf: char[], buffer to be transmitted
    - len: integer, length of buffer (in bytes) to transmit
    - flags: integer, special options, usually just 0
  - int count = recv(sock, &buf, len, flags);
    - count: # bytes received (-1 if error)
    - buf: void[], stores received bytes
    - len: # bytes received
    - flags: integer, special options, usually just 0
  - Calls are **blocking** [returns only after data is sent (to socket buf) / received]

# *close*

- When finished using a socket, the socket should be closed:
- status = close(s);
  - status: 0 if successful, -1 if error
  - s: the file descriptor (socket being closed)
- Closing a socket
  - closes a connection (for SOCK_STREAM)
  - frees up the port used by the socket

# Connection-oriented: Big Picture



# Connection Example - server

```
#include <sys/socket.h> /* for socket(), connect(), send(), and recv() */
#include <netinet/in.h>
#include <signal.h>
#include <ctype.h>
void catcher(int sig);
int newfd;
main()
  {
  int socketd;                          /* socket descriptor */
  struct sockaddr_in srv;          /* used by bind() */
  struct sockaddr_in cli;          /* used by accept() */
  int cli_len = sizeof(cli);       /* used by accept() */
  int not_done = 1;
  char c;

  signal(SIGPIPE, catcher);

  /* 1) create the socket */
  if((socketd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
      perror("socket call failed");
      exit(1);  }
```

# Connection Example - server

```
/* 2) bind the socket to a port */
   srv.sin_family = AF_INET; /* use the Internet addr family */
   srv.sin_port = htons(1700); /* bind socket 'socketd' to port 1700 */

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


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

# Connection example – server con't

```
/* 3) listen on the socket */
  if(listen(socketd, 5) < 0) {
      perror("listen call failed");
      exit(1);
      }


  /* loop looking for messages */
  while (not_done)
    {
        /* 4) accept the incoming connection */
       newfd = accept(socketd, (struct sockaddr*) &cli, &cli_len);
       if(newfd < 0) {
          perror("accept call failed");
          not_done = 0;
        }  // endif
```

# Connection example – server con't

```
   /* spawn a child to deal with this connection */
    if ( fork() == 0)
      {
            while(recv(newfd, &c, 1, 0) > 0) /* could use read as well */
             {
                  c = toupper(c);
                  send(newfd, &c, 1, 0); /* could use write as well */
             }
          /* when client is no longer sending, close socket and child */
         close(newfd);
         exit(0);
      }
     else /* parent */   {   close(newfd);    }
  }  //  While not done

/* signal handler in case socket becomes disconnected */
void catcher(int sig)
  {
     signal(SIGPIPE, catcher);
     close(newfd);
     exit(0);
  }
```

# Connection Example - client

```
#include <sys/socket.h> /* for socket(), connect(), send(), and recv() */
#include <arpa/inet.h> /* for sockaddr_in and inet_addr() */
#include <netinet/in.h>
#include <signal.h>
main()
  {
  int socketd;                          /* socket descriptor */
  struct sockaddr_in srv;           /* used by connect() */
  char c, rc;
  int more_data = 1;



  /* 1) create the socket */

  if((socketd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
      perror("socket call failed");
      exit(1);
      }
```

# Connection Example - client

```
/* 2) connect() to the server */


  srv.sin_family = AF_INET; /* connect: use the Internet address family */
  srv.sin_port = htons(1700); /* connect: socket 'socketd' to port 1700 */
  srv.sin_addr.s_addr = inet_addr("192.197.151.70");  /* connect: connect to IP Address
                                                         '192.197.151.70' - odin */

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

# Connection example – client con't

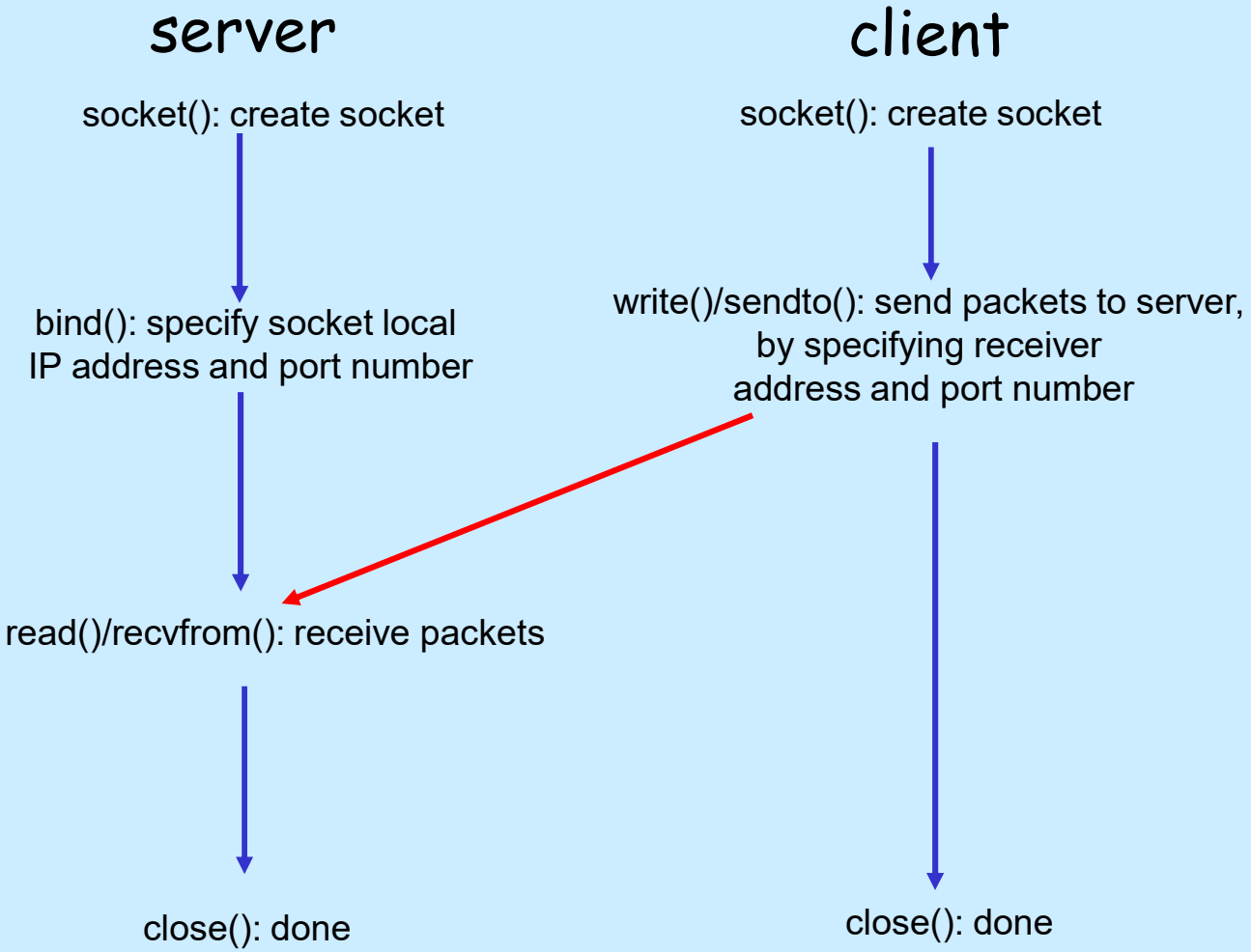
```
/*  send and receive information with the server */
  while (more_data)
    {
        if(c != '\n') /* ignore the enter at the end of the input */
        printf("Input a lower case letter (or 0 to stop) => ");
        c = getchar();

       if(c != '0')
        {
            if(c != '\n') /* ignore the enter for processing */
              {
                 send(socketd, &c, 1, 0); /* could use write as well */
                 if(recv(socketd, &rc, 1, 0) > 0) /* could use read as well */
                        printf("%c\n", rc);
                 else {
                      printf("Server has died\n");
                     close(socketd);
                     exit(1);
                  }  // recv
              } // if not NewLine
           }  // if not a ZERO
        else
       more_data = 0;
    }   // end of while
  exit(0);
  }
```

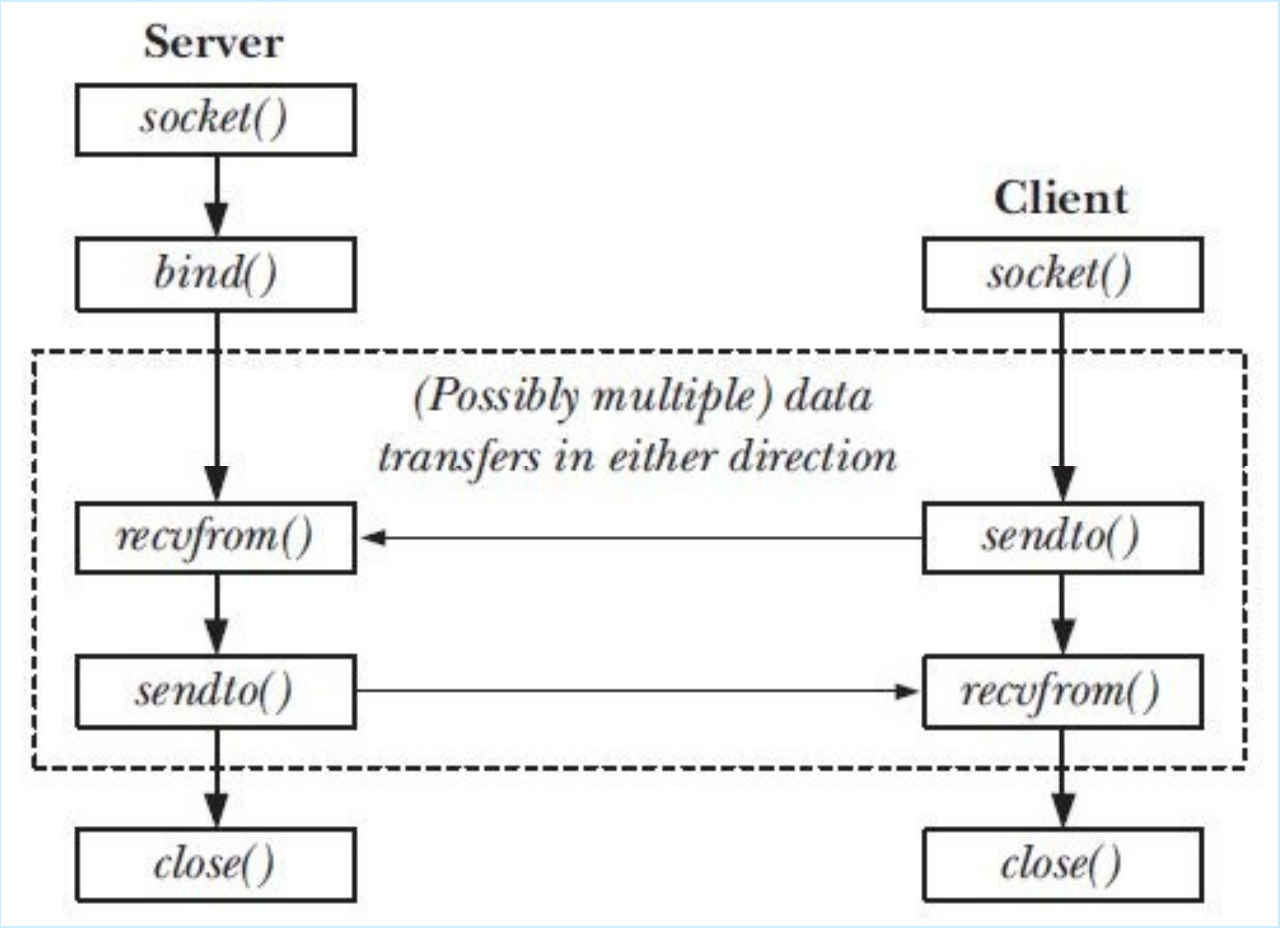
# Connection Example

- To compile use
  - gcc –o server server.c // some versions need –lsocket –lnsl
  - gcc –o client client.c // some versions need –lsocket –lnsl
- To run, start the server first in the background and then the client(s)

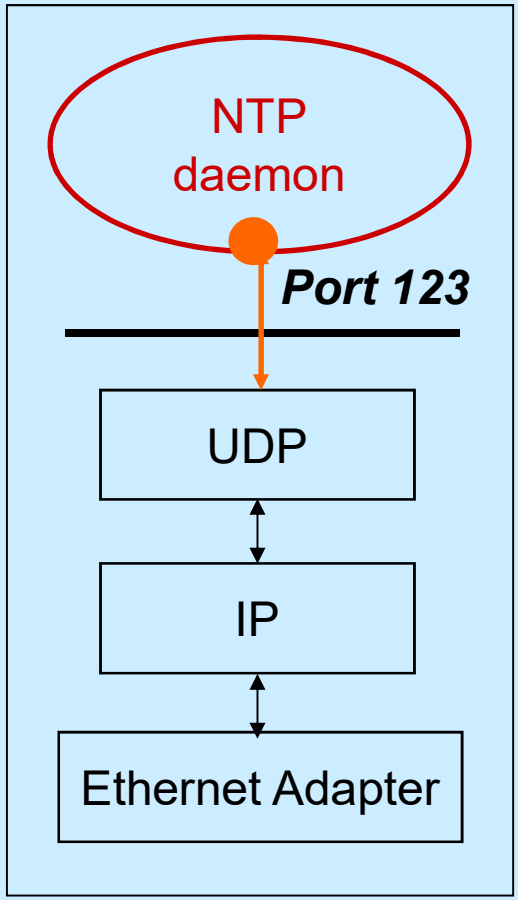
```
$ server &
$ client
Input a lower case letter (or 0 to stop) => r
R
Input a lower case letter (or 0 to stop) => i
I
Input a lower case letter (or 0 to stop) => c
C
Input a lower case letter (or 0 to stop) => h
H
Input a lower case letter (or 0 to stop) => 0
$ kill %1
[1]    Terminated   server
```

# Connectionless: Big Picture




Server
socket()
bind()
recvfrom()
sendto()
close()
Client
socket()
sendto()
recvfrom()
close()
(Possibly multiple) data
transfers in either direction

# UDP Server Example




- For example: NTP daemon
- **What does a *UDP server* need to do so that a *UDP client* can connect to it?**

# Socket I/O: socket()

- The UDP server must create a **datagram** socket…

```
int socketd;        /* socket descriptor */

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

- ***socket*** returns an integer (**socket descriptor**)
  - —***socketd*** < 0 indicates that an error occurred

- AF_INET: associates a socket with the Internet protocol family
- **SOCK_DGRAM: selects the UDP protocol**

# Socket I/O: bind()

- A ***socket*** can be bound to a ***port***

```
int socketd;                          /* 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 'socketd' 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(socketd, (struct sockaddr*) &srv, sizeof(srv)) < 0)
{
        perror("bind"); exit(1);
}
```

- **Now the UDP server  is ready to accept packets…**

# Socket I/O: recvfrom()

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

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

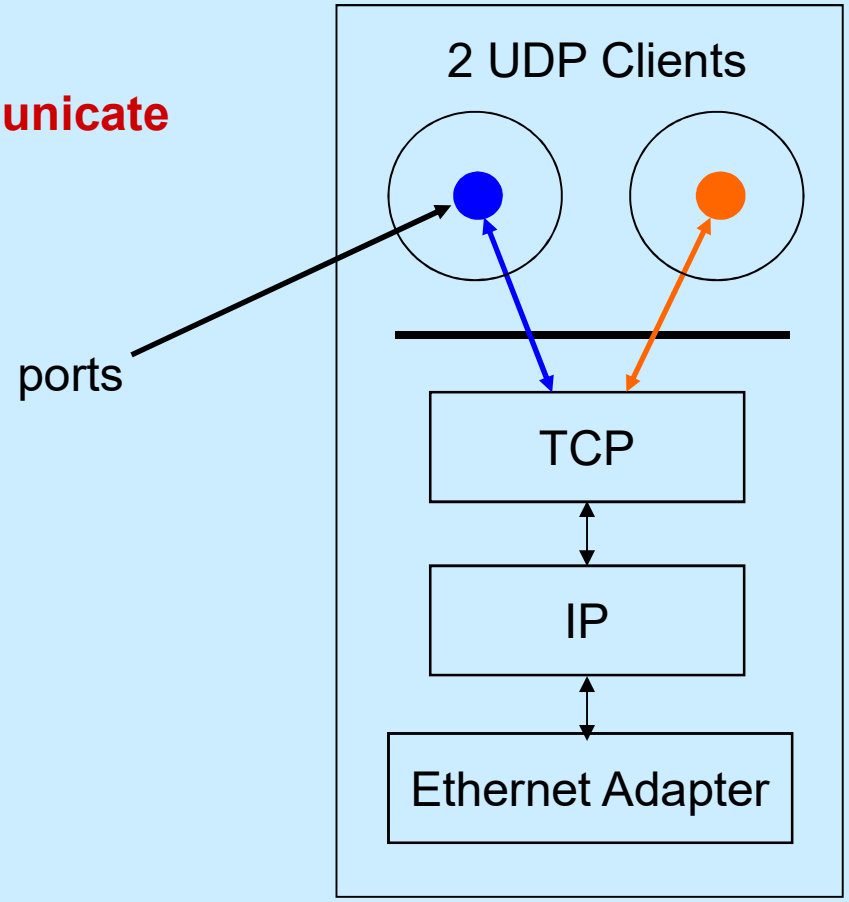
# Socket I/O: recvfrom() continued...

```
nbytes = recvfrom(socketd, buf, sizeof(buf), 0 /* flags */,
                  (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***)
  - —don't worry about flags

# UDP Client Example

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

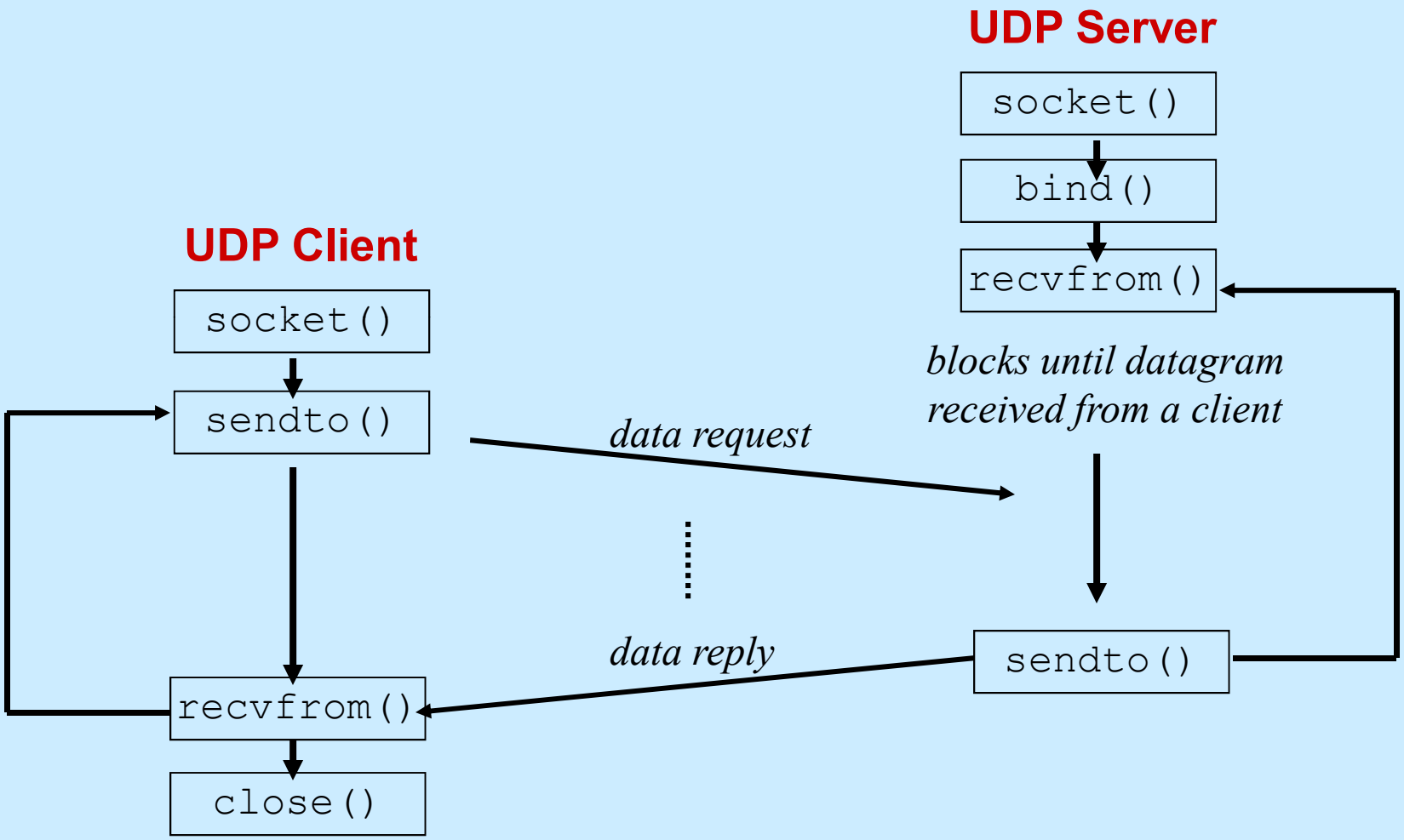


# Socket I/O: sendto()

- ***write*** is not allowed
- Notice that the UDP client does not ***bind*** a port number
  - a port number is **dynamically assigned** when the first ***sendto*** is called

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

/* 1) create the socket */

/* sendto: send data to IP Address "192.197.151.70" port 80 */
srv.sin_family = AF_INET;
srv.sin_port = htons(80);
srv.sin_addr.s_addr = inet_addr("192.197.151.70");

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

# Review: UDP Client-Server Interaction



# Connectionless example - server

```
#include <sys/socket.h> /* for socket(), connect(), send(), and recv() */
#include <netinet/in.h>
#include <signal.h>
#include <ctype.h>
main()
  {
  int socketd;                          /* socket descriptor */
  struct sockaddr_in srv;          /* used by bind() */
  struct sockaddr_in cli;           /* used by sendto(), recvfrom() */
  int cli_len = sizeof(cli);          /* used by sendto(), recvfrom() */
  int not_done = 1;
  char c;
  /* 1) create the socket */
  if((socketd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
      perror("socket call failed");
      exit(1);
      }
```

# Connectionless example - server

```
/* 2) bind the socket to a port */
  srv.sin_family = AF_INET; /* use the Internet addr family */

  srv.sin_port = htons(1700); /* bind socket 'socketd' to port 1700 */

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


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

# Connectionless – server con't

```
/* loop looking for messages */
  while(not_done)
    {
       /* 3) receive a message */
       if(recvfrom(socketd, &c, 1, 0, (struct sockaddr *) &cli, &cli_len) < 0)
         {
            perror("Server: receiving");
            not_done = 0;
           }

      c = toupper(c);

    /* send the message back to where it came from */
       if(sendto(socketd, &c, 1, 0, (struct sockaddr *) &cli, cli_len) < 0)
        {
          perror("Server: sending");
          not_done = 0;
         }
    } // while not done
  exit(0);
  }
```

# Connectionless example - client

```
#include <sys/socket.h> /* for socket(), connect(), send(), and recv() */
#include <arpa/inet.h> /* for sockaddr_in and inet_addr() */
#include <netinet/in.h>
#include <signal.h>
main()
  {
  int socketd;                          /* socket descriptor */
  struct sockaddr_in srv;               /* used by sendto() */
  int srv_len = sizeof(srv);
  char c;
  int more_data = 1;
  /* 1) create the socket */
  if((socketd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
      perror("socket call failed");
      exit(1);
      }
```

# Connectionless example - client

```
/* set up the server information for the sendto */
  srv.sin_family = AF_INET;
  /* port number */
  srv.sin_port = htons(1700);
  /* IP Address '192.197.151.70' - odin */
  srv.sin_addr.s_addr = inet_addr("192.197.151.70");
```

# Connectionless – client con't

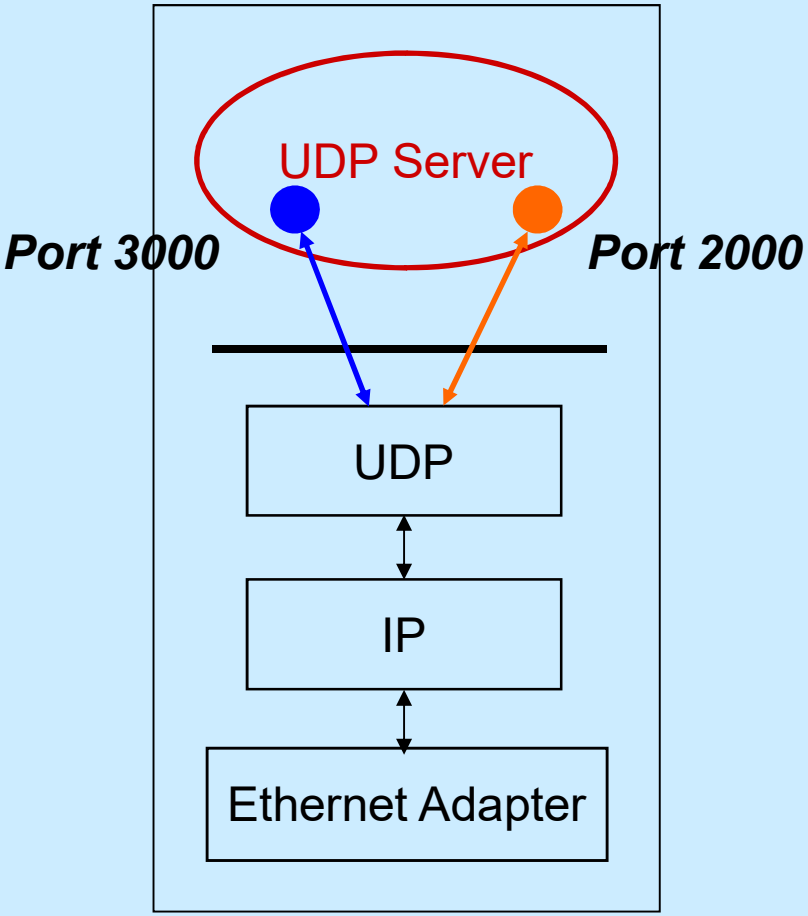
```
/*  send and receive information with the server */
 while (more_data)     {
        if(c != '\n') /* ignore the enter at the end of the input */
        printf("Input a lower case letter (or 0 to stop) => ");
        c = getchar();
       if(c != '0')     {
            if(c != '\n') { /* ignore the enter for processing */

           if (sendto(socketd, &c, 1, 0, (struct sockaddr *) &srv, srv_len) < 0)  {
              perror("Client: sending");
              exit(1);
             }
          if (recvfrom(socketd, &c, 1, 0, (struct sockaddr *) &srv, &srv_len) < 0) {
            perror("Client: receiving");
            exit(1);
            }
        printf("%c\n", c);
        }  //  c!= new line
     }  // c !=0
   else
     more_data = 0;
   }
exit(0);
}
```

# Connectionless Example

- To compile use
  - gcc –o server1 server1.c // some versions need –lsocket –lnsl
  - gcc –o client1 client1.c // some versions need –lsocket –lnsl
- To run, start the server first in the background and then the client(s)

```
$ server1 &
$ client1
Input a lower case letter (or 0 to stop) => r
R
Input a lower case letter (or 0 to stop) => i
I
Input a lower case letter (or 0 to stop) => c
C
Input a lower case letter (or 0 to stop) => h
H
Input a lower case letter (or 0 to stop) => 0
$ kill %1
[1]    Terminated   server1
```

# The UDP Server




- **How can the *UDP server* service multiple ports simultaneously?**

# UDP Server: Servicing Two Ports

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

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

while(not_done) {
      recvfrom(s1, buf, sizeof(buf), ...);
      /* process buf */

      recvfrom(s2, buf, sizeof(buf), ...);
      /* process buf */
}
```

- **What problems does this code have?**

# Dealing with blocking calls

- Many of the functions we saw block until a certain event
  - accept: until a connection comes in
  - connect: until the connection is established
  - recv, recvfrom: until a packet (of data) is received
  - send, sendto: until data is pushed into socket's buffer
- For simple programs, blocking is convenient
- What about more complex programs?
  - multiple connections
  - simultaneous sends and receives
  - simultaneously doing non-networking processing

# Dealing w/ blocking (cont'd)

- Options:
  - create multi-process or multi-threaded code
  - turn off the blocking feature (e.g., using the fcntl file-descriptor control function)
  - use the select function call.

- What does select() do?
  - can be permanent blocking, time-limited blocking or non-blocking
  - input: a set of file-descriptors
  - output: info on the file-descriptors' status
  - i.e., can identify sockets that are "ready for use": calls involving that socket will return immediately

# select function call

- int status = select(nfds, &readfds, &writefds, &exceptfds, &timeout);
  - status: # of ready objects, -1 if error
  - nfds: 1 + largest file descriptor to check
  - readfds: list of descriptors to check if read-ready
  - writefds: list of descriptors to check if write-ready
  - exceptfds: list of descriptors to check if an exception is registered
  - timeout: time after which select returns, even if nothing ready - can be 0 or ∞

    (point timeout parameter to NULL for ∞)

# To be used with select:

- Recall select uses a structure, struct fd_set
  - it is just a bit-vector
  - if bit *i* is set in [readfds, writefds, exceptfds], select will check if file descriptor (i.e. socket) *i* is ready for [reading, writing, exception]
- Before calling select:
  - FD_ZERO(&fdvar): clears the structure
  - FD_SET(i, &fdvar): to check file desc. *i*
- After calling select:
  - int FD_ISSET(i, &fdvar): boolean returns TRUE iff *i* is "ready"

# Socket I/O: select()

```
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

# Socket I/O: select()

```
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

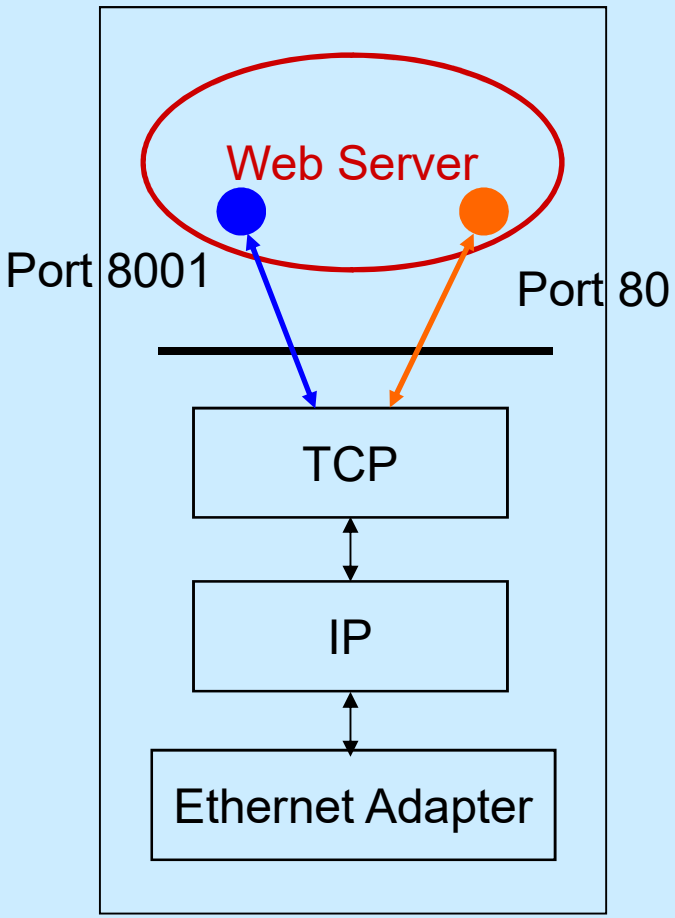
- Refer to the man page for more information

# Socket I/O: select()

- ***select*** allows synchronous I/O multiplexing

```
int s1, s2;                          /* socket descriptors */
fd_set readfds;                      /* used by select() */
int not_done = 1;

/* create and bind s1 and s2 */
while(not_done) {
        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 */
}
```

# More Details About a Web Server




**How can a a *web server* manage multiple connections simultaneously?**

# Socket I/O: select()

```
int fd, next=0;                          /* original socket */
int newfd[10];                           /* new socket descriptors */
int not_done = 1;

while(not_done) {
        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 */
        }
}
```

- **Now the web server can support multiple connections...**

# Other useful functions

- bzero(char* c, int n): 0's n bytes starting at c
- gethostname(char *name, int len): gets the name of the current host
- gethostbyaddr(char *addr, int len, int type): converts IP hostname to structure containing long integer
- inet_addr(const char *cp): converts dotted-decimal char-string to long integer
- inet_ntoa(const struct in_addr in): converts long to dotted-decimal notation

- Warning: check function assumptions about byte-ordering (host or network). Often, they assume parameters / return solutions in network byte-order