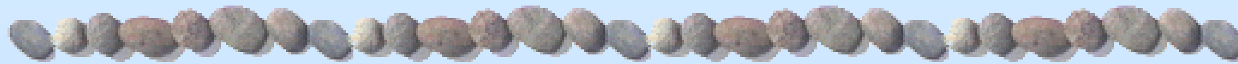
Sockets



UNIX-style IPC

Silberschatz, Galvin and Gagne ©2005

Msc. Ivan A. Escobar Broitman © 2007

Introduction to Sockets

- A socket is one of the most fundamental technologies of computer networking.
- The socket is the BSD method for accomplishing interprocess communication (IPC).
- What this means is a socket is used to allow one process to speak to another, very much like the telephone is used to allow one person to speak to another.
- Many of today's most popular software packages -- including Web Browsers, Instant Messaging and File Sharing -- rely on sockets.

The Socket Interface

- Funded by ARPA (Advanced Research Projects Agency) in 1980.
- Developed at UC Berkeley
- Objective: to transport TCP/IP software to UNIX
- The socket interface has become a de facto standard.

History of Sockets

- Sockets were introduced in 1981 as the Unix BSD 4.2 generic interface for Unix to Unix communications over networks.
- In 1985, SunOS introduced NFS and RPC over sockets.
- In 1986, AT&T introduced the Transport Layer Interface (TLI) with socket-like functionality but more network independent.

Sockets and TLI

- Unix after SVR4 includes both TLI and Sockets.
- TLI (Transport Layer Interface): was the networking API provided by AT&T Unix System V counterpart to BSD socket.
- The two are very similar from a programmers perspective. TLI is just a cleaner version of sockets that is, in theory, stack independent.
- TLI has about 25 API calls.

TCP/IP Network Standard

- The Windows socket API, *Winsock*, is a multi-vendor specification to standardize the use of TCP/IP under Windows. It is based on the Berkeley sockets interface.
- In BSD Unix, Sockets are part of the kernel and provide standalone and networked IPC services.
- MS-DOS, Windows, Mac OS, and OS/2 provide sockets in the form of libraries.

Socket Types

- There are 3 types of sockets:
  - Stream sockets interface to the TCP (transmission control protocol).
  - Datagram sockets interface to the UDP (user datagram protocol).
  - Raw sockets interface to the IP (Internet protocol).

TCP vs UDP

- TCP used for services with a large data capacity, and a persistent connection, while UDP is more commonly used for quick lookups, and single use query-reply actions.
- Some common examples of TCP and UDP with their default ports:

DNS lookup	UDP	53
FTP	TCP	21
HTTP	TCP	80
POP3	TCP	110
Windows shared printer name lookup	UDP	137
Telnet	TCP	23

Connection Oriented Protocols

- Connection-oriented protocols operate in three phases.
  - The first phase is the *connection setup* phase, during which the corresponding entities establish the connection and negotiate the parameters defining the connection.
  - The second phase is the *data transfer* phase, during which the corresponding entities exchange messages under the auspices of the connection.
  - Finally, the *connection release* phase is when the correspondents "tear down" the connection because it is no longer needed.

TCP vs UDP

- TCP (Transmission Control Protocol)
  - Connection-oriented
  - Reliability in delivery of messages
  - Splitting messages into datagrams
  - keep track of order (or sequence)
  - Use checksums for detecting errors

TCP vs UDP

- UDP (User Datagram Protocols)
  - Connectionless
  - No attempt to fragment messages
  - No reassembly and synchronization
  - In case of error, message is retransmitted
  - No acknowledgment

Socket Declaration

Socket Address

- A socket address on the TCP/IP internet consists of two parts:
- An **internet** (IP) **address**, a 32 bit number usually represented by 4 decimal number separated by dots. It is a unique identifier for a network interface card within an administered AF_INET domain. A TCP/IP Host may have as many addresses as it has network interfaces. *(Newer IP addresses have 6 decimal numbers)*
- A 16 bit **port number**, which is an entry point to an application that resides on a host. Port define entry points for services provided by server applications. Important commercial applications such as Oracle have their own well known ports.

Socket Call

- Means by which an application attached to the network
- int socket(int family, int type, int protocol)
- *Family*: address family (protocol family)
  - AF_UNIX, AF_INET, AF_NS, AF_IMPLINK
- *Type*: semantics of communication
  - SOCK_STREAM, SOCK_DGRAM, SOCK_RAW
  - Not all combinations of family and type are valid
- *Protocol*: Usually set to 0 but can be set to specific value.
  - Family and type usually imply the protocol
- Return value is a *handle* for new socket

Bind Call

- Binds a newly created socket to the specified address
- Int bind(int socket, struct sockaddr *address, int addr_len)
- *Socket*: newly created socket handle
- *Address*: data structure of address of *local* system
  - IP address and port number (demux keys)
  - Same operation for both connection-oriented and connectionless servers
    - Can use well known port or unique port

Listen Call

- Used by connection-oriented servers to indicate an application is willing to receive connections
- Int(int socket, int backlog)
- *Socket*: handle of newly creates socket
- *Backlog*: number of connection requests that can be queued by the system while waiting for server to execute accept call.

Accept Call

- After executing *listen,* the accept call carries out a *passive open* (server prepared to accept connects).
- Int accept(int socket, struct sockaddr *address, int addr_len)
- It blocks until a remote client carries out a connection request.
- When it does return, it returns with a *new* socket that corresponds with new connection and the address contains the clients address

But What is a socket?

- A communication endpoint
- An OS data structure that can be created, manipulated and used for communication using system calls
- Created using the socket() system call
  - fd = socket(AF_INET, SOCK_STREAM, 0)

File descriptor

domain

protocol (normally 0)

socket type
- SOCK_STREAM: reliable byte stream, connection-oriented (TCP)
- SOCK_DGRAM: unreliable, message-based, connectionless (UDP)
- SOCK_RAW: raw socket that bypasses transport layer

- ***socket*** returns an integer (socket descriptor)
  - fd < 0 indicates that an error occurred
  - socket descriptors are similar to file descriptorsct to it

Socket Initialization

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

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

Returns: non-negative socket descriptor if OK, –1 on error
```

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

- AF_INET: associates a socket with the Internet protocol family
- Family specifies address or protocol family. Can be AF_INET, AF_INET6, AF_LOCAL, etc
- Type specifies socket type. Can be SOCK_DGRAM, SOCK_STREAM, SOCK_RAW
- Protocol argument is normally left to zero except for raw sockets.

Socket Connection Overview

- Server:
  1. socket()
  2. bind()
  3. listen()
  4. accept()
- accept() returns a new file descriptor to be used to communicate with incoming requester
- old (listening) socket remains

- Client:
  1. socket()
  2. connect()
- “Unpleasant” calls (use weird data structures as arguments and may need casts)

Connect Call

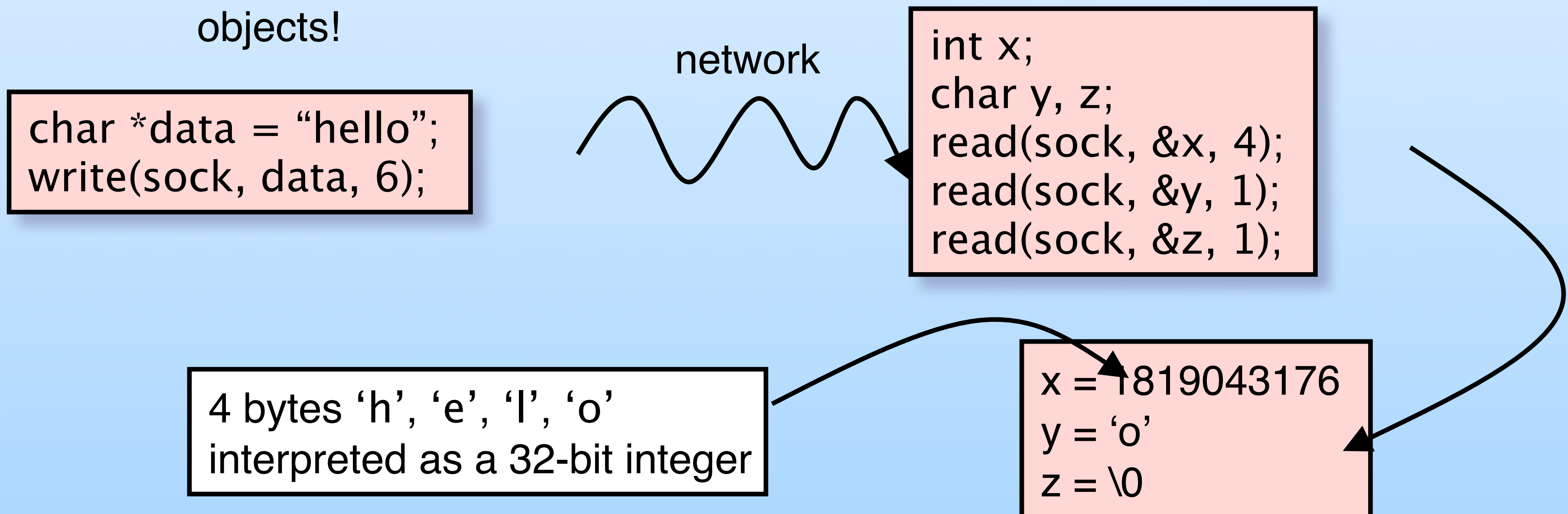
- Client executes an *active open* of a connection
- Int connect(int socket, struct sockaddr *address, int addr_len)
- Call does not return until the three-way handshake (TCP) is complete
- Address field contains remote system’s address
- Client OS usually selects random, unused port

Send(to), Receive (from)

- After connection has been made, application uses send/ recv to data
- Int send(int socket, char *message, int msg_len, int flags)
  - Send specified message using specified socket
- Int recv(int scoket, char *buffer, int buf_len, int flags)
  - Receive message from specified socket into specified buffer

Socket Connection Overview (cont'd)

- Once a connection is established, communication is just like reading/writing a file
- Sockets give us file descriptors
  - To send data: write() system call
  - To receive data: read() system call
- Can convert them to FILE * to use with fprintf() or fscanf() for formatted messages
- Note: read() and write() read/write bytes, not ints, floats, or objects!



Sending objects over the network

- To convert between the network byte order and the host byte order
  - ntohl(): Network to host (byte order) long (32 bits)
  - ntohs(): Network to host (byte order) short (16 bits)
  - htonl(): Host to network (byte order) long (32 bits)
  - htons(): Host to network (byte order) short (16 bits)

  - On Intel 80x86 host byte order is Least Significant Byte first, Internet byte order is Most Significant byte first, so make sure you use these calls!
  - Remember: a port number is a short (16 bits) so use to print port of incoming connection (for example)
  - Do "man ntohl" etc
- For more complicated objects (C arrays, structures, even with pointers in them): use XDR (eXternal Data Representation) calls ("man xdr")
  - Introduced by Sun Microsystems

Address conversion routines

- An Internet address can be written in a numerical form such as: 148.241.1.2 or by a set of characters such as *pandora.cem.itesm.mx*, but when stored for network transfer it is stored as a integer type variable.
- Certain functions let us conver from one format to another as shown below:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

unsigned long inet_addr(char *ptr);

converts a string character notation into a 32 bit decimal notation

char *inet_ntoa(struct in_addr inaddr);

converts a decimal notation into a string of characters notation

Byte Operations

- Below are some useful BSD functions used in conjunctions with socket operations.

bcopy(char *src, char *dest, int nbytes);

moves the number of specified bytes from src to dest

bzero(char *dest, int nbytes);

writes the specified number of null bytes into dest

int bcmp(char *ptr1, char *ptr2, int nbytes);

compares 2 bytes of string

returns zero if both strings are identical, if not it returns a non zero value

Closing a connection: close()

- ***int close(sockfd)***

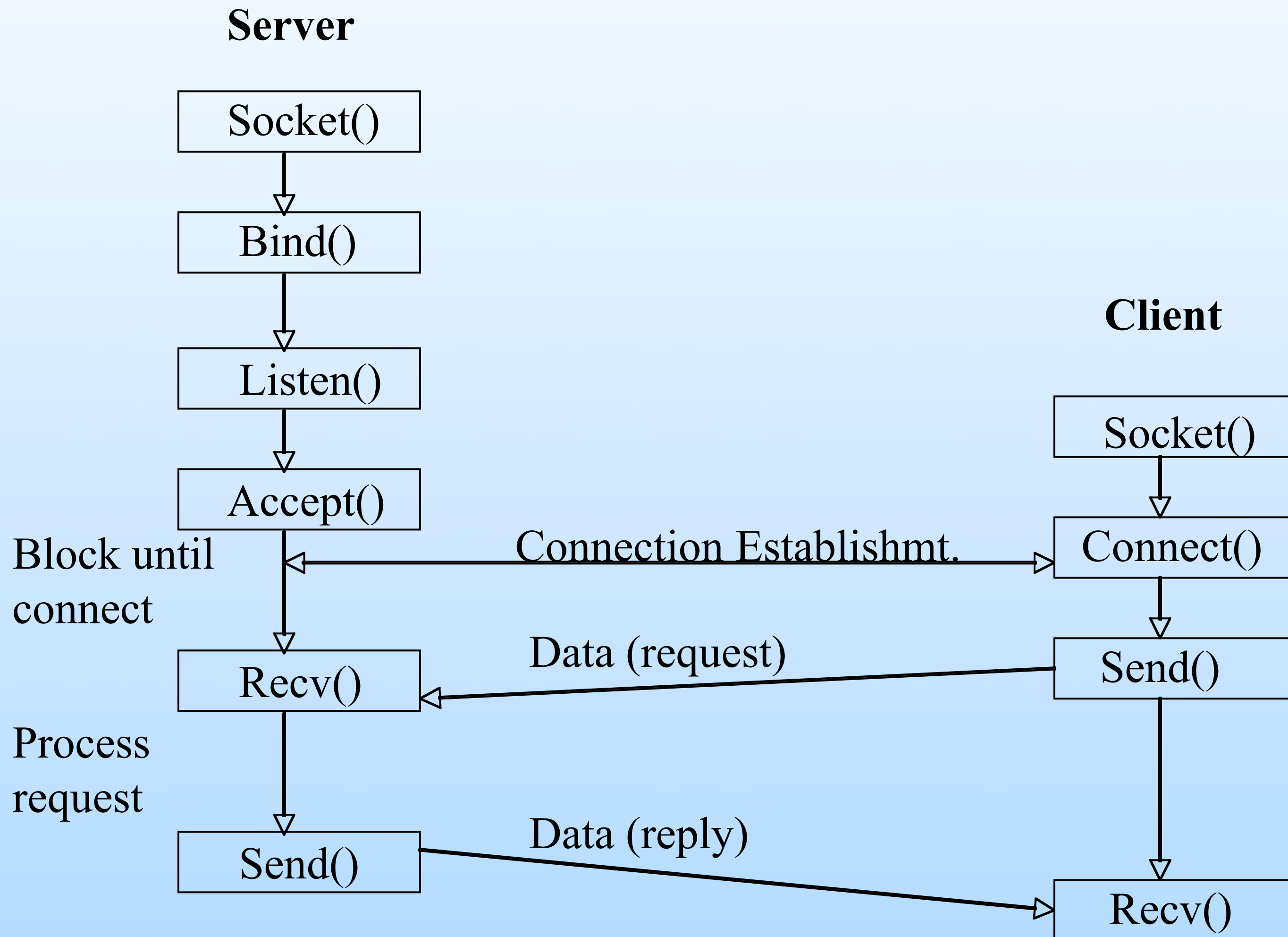
  Closes the socket
  It is similar to closing a file.

  int sockfd
  socket descriptor

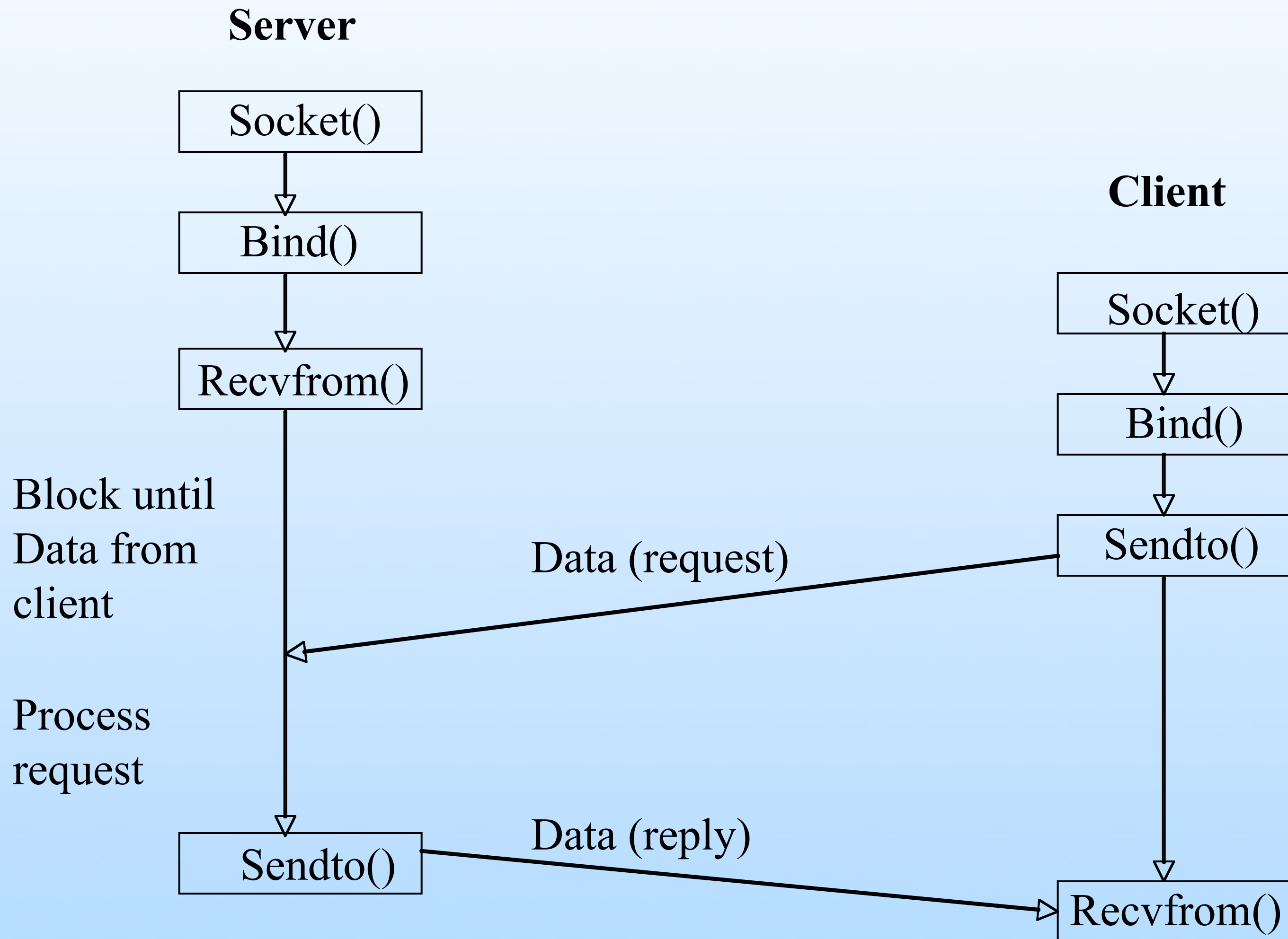
  Example:

```
int sock
        :
  sock = socket(…………..);
        :
        :
  close(sock);
```

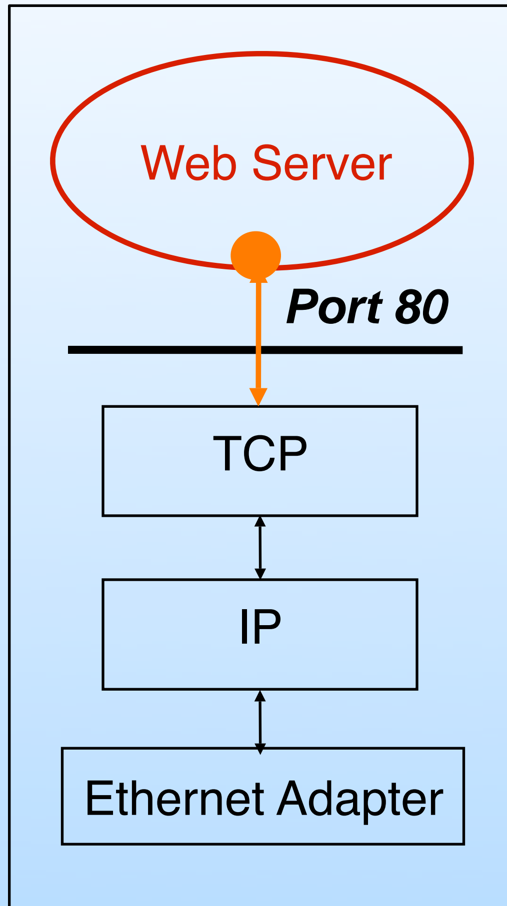
Connection Oriented Example (TCP)



Connectionless Example (UDP)



Examples: 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

```
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
- **AF_INET** associates a socket with the Internet protocol family
- **SOCK_STREAM selects the TCP protocol**

Socket I/O : bind()

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

```
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);
}
```

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

Socket I/O: listen()

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

```
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);
}
```

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

Socket I/O: accept()

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

```
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);
}
```

- ***accept*** returns a new socket (***newfd***) with the same properties as the original socket (***fd***)
  - ***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(fd, (struct sockaddr*) &cli, &cli_len);
if(newfd < 0) {
     perror("accept");
     exit(1);
}
```

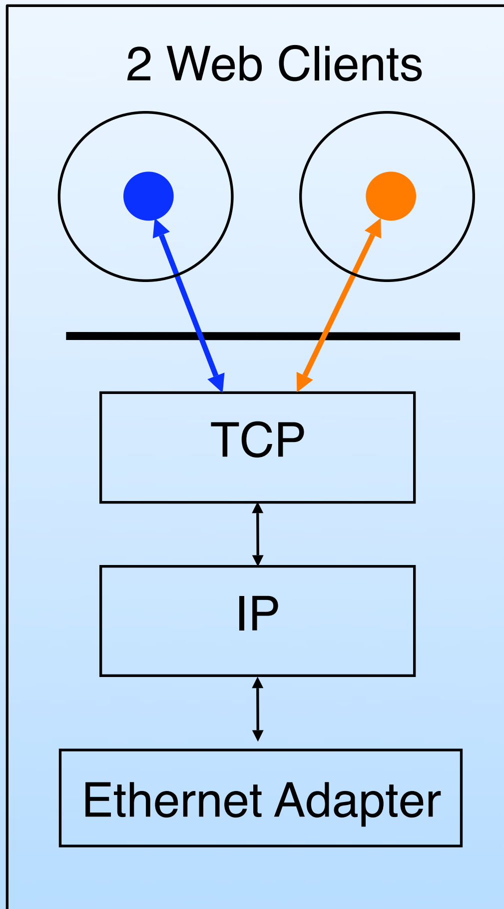
- How does the server know which client it is?
  - **cli.sin_addr.s_addr** contains the client's ***IP address***
  - **cli.sin_port** contains the client's ***port number***
- Now the server can exchange data with the client by using ***read*** and ***write*** on the descriptor ***newfd***.
-

Socket I/O: read()

```
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);
}
```

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

Examples: TCP Client



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

Dealing with IP Addresses

- IP Addresses are commonly written as strings (“128.2.35.50”), but programs deal with IP addresses as integers.

Converting strings to numerical address:

```
struct sockaddr_in srv;

srv.sin_addr.s_addr = inet_addr(“128.2.35.50”);
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);
}
```

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 = "www.cs.cmu.edu";

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...

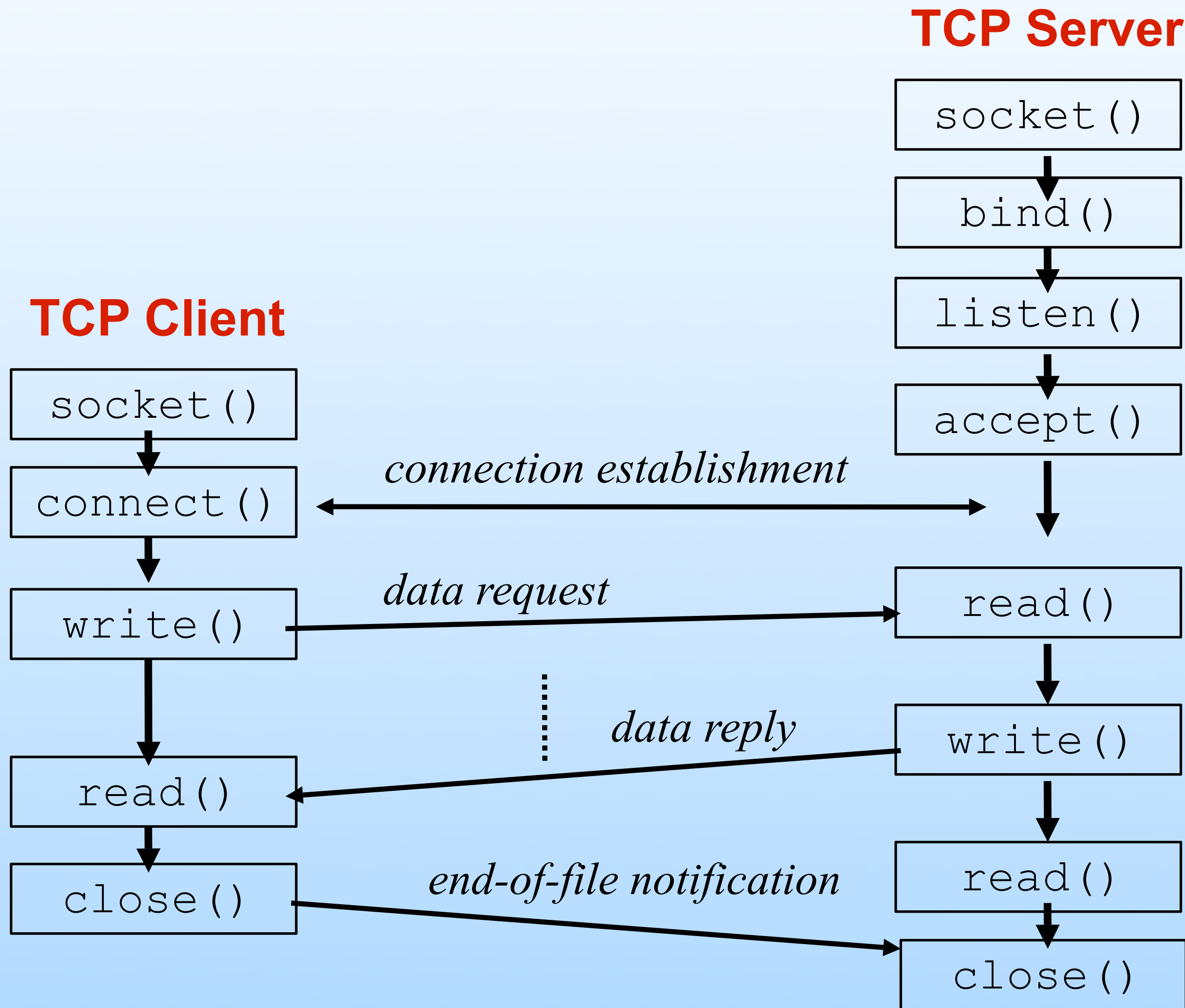
```
int fd;                               /* socket descriptor */
struct sockaddr_in srv;               /* used by connect() */

/* create the socket */

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

/* connect: socket ‘fd’ to port 80 */
srv.sin_port = htons(80);

/* connect: connect to IP Address “128.2.35.50” */
srv.sin_addr.s_addr = inet_addr(“128.2.35.50”);

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

Socket I/O: write()

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

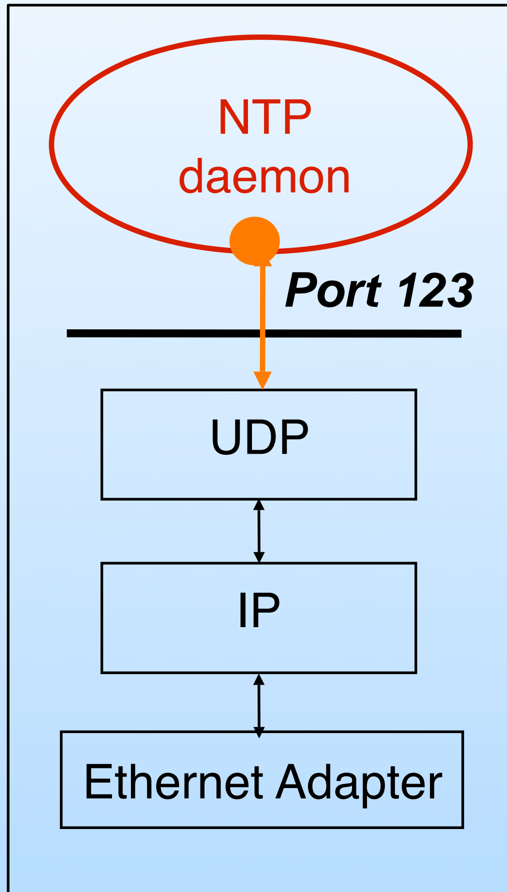
```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;          /* used by connect() */
char buf[512];                   /* used by write() */
int nbytes;                      /* used by write() */

/* 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);
}
```

Review TCP Client/Server Interaction



from UNIX Network Programming Volume 1, figure 4.1

Examples: UDP Server



- For example: NTP demon
- **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 fd;                 /* socket descriptor */

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

- ***socket*** returns an integer (**socket descriptor**)
  - ***fd*** < 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 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);
}
```

- **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

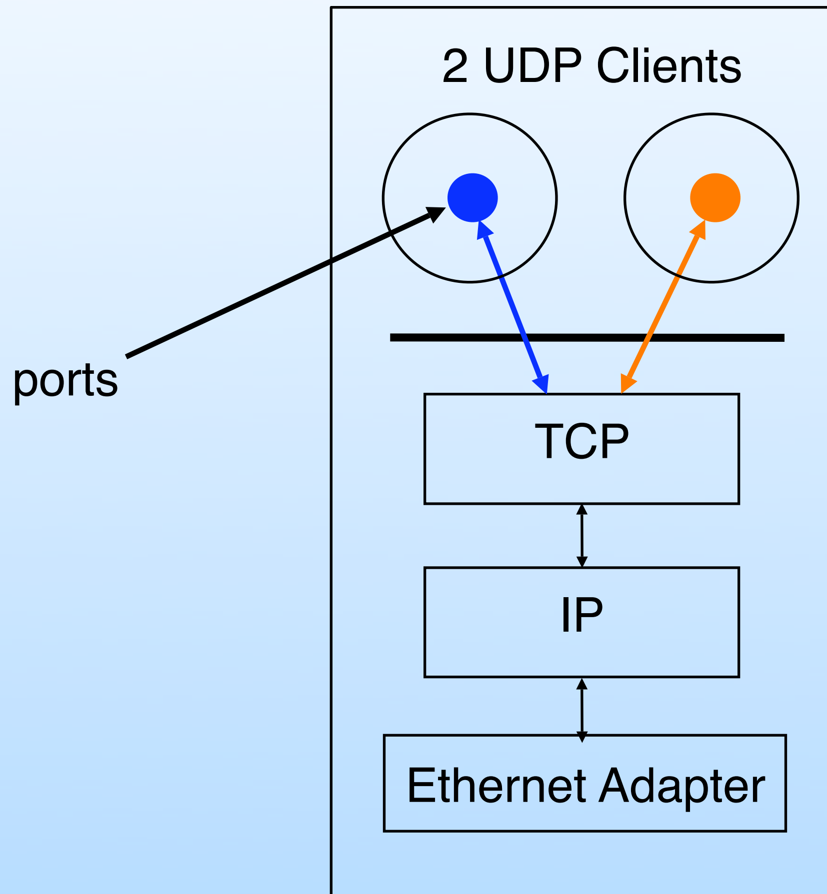
```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;          /* used by bind() */
struct sockaddr_in cli;          /* used by recvfrom() */
char buf[512];                   /* used by recvfrom() */
int cli_len = sizeof(cli);       /* used by recvfrom() */
int nbytes;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


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


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

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

```
nbytes = recvfrom(fd, 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

Examples: UDP Client



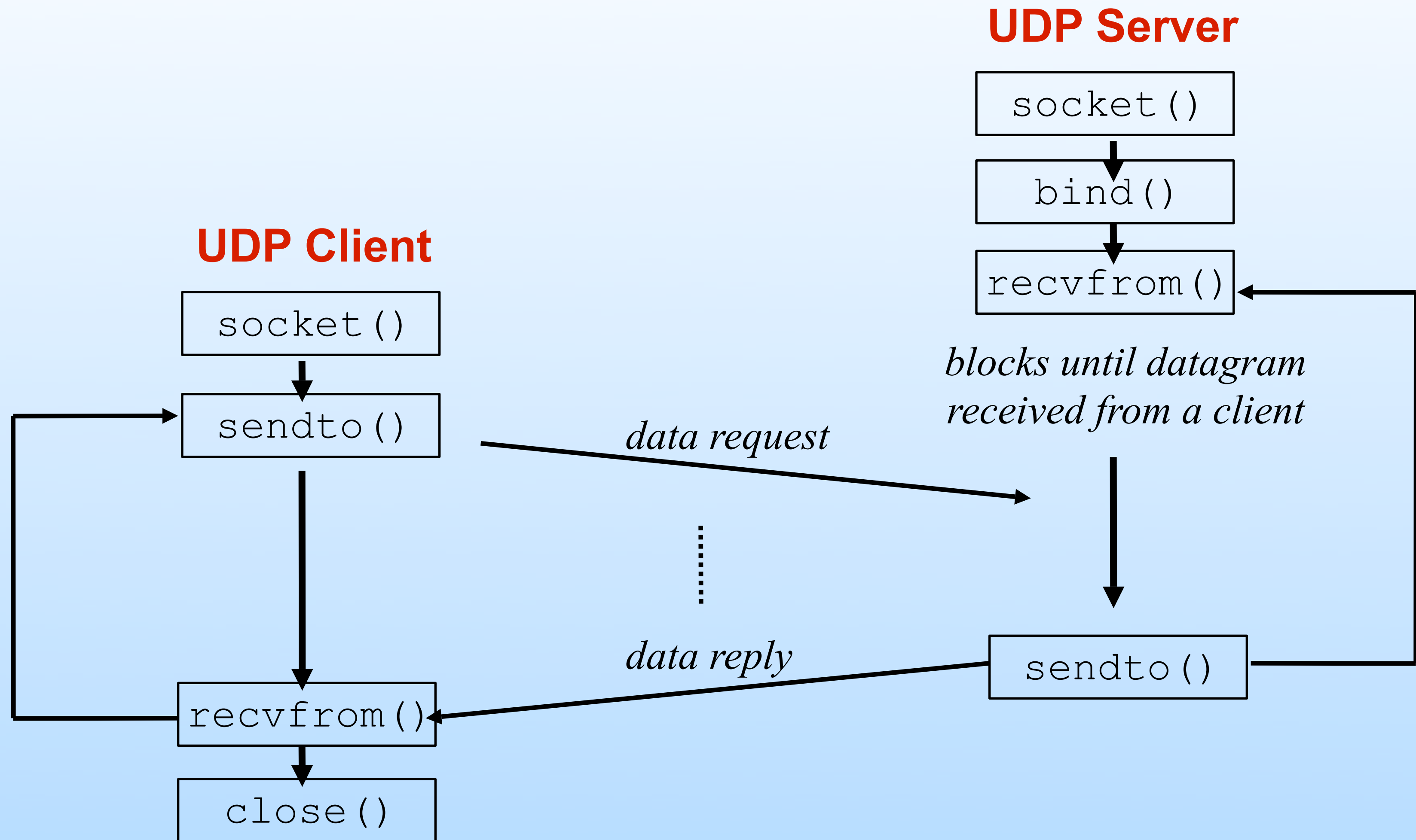
- **How does a UDP client communicate with a UDP server?**

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 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);
}
```

Review: UDP Client-Server Interaction



from UNIX Network Programming Volume 1, figure 8.1

Example TCP/IP Server Code

```
/*
 *   Simple TCP/IP socket server.
 */

#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <iostream.h>
#include <string.h>
#include <sys/time.h>
#include <sys/types.h>

/* Creates a socket that listens for connections.
 * Returns: the file descriptor of the listener on success,
 *          -1 on failure.
 */
int MakeListener()
{
  /*  Create a socket (i.e., communication endpoint). */
  int listener = socket(AF_INET, SOCK_STREAM, 0);
  if (listener < 0) {
    cerr << "Couldn't create socket\n";
    return -1;
  }
```

```
/*  Name the socket
 *  (required before receiving connections)
 */
struct sockaddr_in s1;
bzero((char *) &s1, sizeof(s1));  /* They say to do this */
s1.sin_family = AF_INET;
s1.sin_addr.s_addr = INADDR_ANY; /* Use any of host's addresses. */
s1.sin_port = 0;  /* Have a port number assigned to us. */
if (bind(listener, (sockaddr *) &s1, sizeof(s1)) < 0) {
  cerr << "Couldn't bind address to socket\n";
  return -1;
}

/*  Get the host name. */
char hostname[48];
gethostname(hostname, 48);
```

```
/*  Get the name of the socket.
 *  We only care about the port number, so that
 *  the clients know how to connect to our socket.
 */
size_t length;
length = sizeof(s1);
getsockname(listener, (sockaddr *) &s1, &length);

cout << "\nListening on host: " << hostname;
cout << ", port: " << ntohs(s1.sin_port) << "\n\n";

/*  Start listening for connections. */
if (listen(listener, 1) < 0) {
  cerr << "Couldn't listen().\n";
  return -1;
}

cout << "Ready for incoming connections\n";
return listener;
}
```

```
int main()
{
  int listener = MakeListener();
  if (listener < 0) return -1;
  for (;;) {
    /*  Wait for, and then accept an incoming connection. */
    cout << "Server waiting for connections\n";
    struct sockaddr_in s2;
    size_t length = sizeof(s2);
    int conn = accept(listener, (sockaddr *) &s2, &length);

    /*  We now have a connection to a client via
     *  file descriptor "conn".
     */

    cout << "Server accepted connection\n";

    /*  Get a message from the client. */
    char data[128];
    int msglen = read(conn, data, 128);
    cout << "Server got " << msglen << " byte message: " << data << "\n";
```

```
    /*  Send a quick acknowledgement to the client
     *  (the number of bytes we received).
     */
    write(conn, &msglen, sizeof(msglen));

    /*  Close the connection on this end. */
    close(conn);
  }
  cout << "How did we get here?\n";
  return 0;
}
```

Example TCP/IP Client Code

```
/*
 *  Simple TCP/IP socket client.
 */
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <iostream.h>

int ServerConnect()
/*  Establishes a TCP/IP connection with the server.
 *  The user is prompted for the hostname and port number.
 *  Returns: the file descriptor of the socket on success,
 *           -1 on failure
 */
{
  char server_host[80];
  u_short server_port;

  cout << "Enter the hostname the server is running on\n";
  cin.getline(server_host, 80);
  cout << "Enter the port number the server is listening on\n";
  cin >> server_port;
  cin.ignore(1,'\n');
```

```
/*  Create a socket (i.e., communication endpoint). */
int sock;
sock = socket(AF_INET, SOCK_STREAM, 0);

/*  Convert (host, port) into the required form
 */
struct sockaddr_in dest;
bzero((char *) &dest, sizeof(dest));  /* They say to do this */

/*  Get info about the host. */
struct hostent *hostptr = gethostbyname(server_host);
if (NULL == hostptr) {
  cerr << "Error looking up host " << server_host << "\n";
  return -1;
}
dest.sin_family = AF_INET;
bcopy(hostptr->h_addr, (char *) &dest.sin_addr, hostptr->h_length);
dest.sin_port = htons(server_port);
```

```
  /*  The address is set up, we're ready to connect. */
  cout << "Trying to connect\n";
  if (connect(sock, (sockaddr *) &dest, sizeof(dest))) {
    cout << "Couldn't connect\n";
    return -1;
  }

  cout << "Connection established\n";
  return sock;
}
```

```
int main()
{
  int conn = ServerConnect();
  if (conn < 0) return -1;

  char message[128];
  int ack;
  cout << "Enter message for server:\n";
  cin.getline(message, 128);

  /*  Send the message to the server. */
  write(conn, message, 1+strlen(message));

  /*  Get the ack from the server. */
  read(conn, &ack, sizeof(ack));

  cout << "The server got " << ack << " bytes\n";

  /*  Close the connection on this end. */
  close(conn);
  return 0;
}
```

pyrite-n2 : ~

```
dmarg@pyrite-n2 [~]% ./tcp-server

Listening on host: pyrite-n2.lab.cs.iastate.edu  port: 24533

Ready for incoming connections
Server waiting for connections
Server accepted connection
Server got 13 byte message: hello there!
Server waiting for connections
Server accepted connection
Server got 15 byte message: how do you do?
Server waiting for connections
```

pyrite-n2 : ~

```
dmarg@pyrite-n2 [~]% ./tcp-client
Enter the hostname the server is running on
pyrite-n2.lab.cs.iastate.edu
Enter the port number the server is listening on
24533
Trying to connect
Connection established
Enter message for server:
hello there!
The server got 13 bytes
dmarg@pyrite-n2 [~]% ./tcp-client
Enter the hostname the server is running on
pyrite-n2.lab.cs.iastate.edu
Enter the port number the server is listening on
24533
Trying to connect
Connection established
Enter message for server:
how do you do?
The server got 15 bytes
dmarg@pyrite-n2 [~]%
```