TCP/IP Programming Interface

- The network API for TCP/IP is the set of system calls provided by the operating system for the application to interface with the protocol software.

- Up to now, we have concentrated on the principles and concepts underlying the **TCP/IP** protocol suite without giving details on how applications use the protocols.

- Two interfaces were developed to provide UNIX interprocess communication and access to TCP/IP:

  System V Streams and the TLI (Transport Layer Interface)
  Berkley Sockets

- We will discuss the Berkley Socket API implementation. We can view the socket as a kernel data structure which can be accessed by the application program to perform I/O operations.

- The idea is to have two processes set up a pair of "**sockets**" to create a **communications channel** between them.

- In that sense it is similar to the **IPC** methods we have seen to date.

- The socket method however is a considerably more powerful because it not only allows inter-process communications on the same machine, but also on machines which are geographically separated.

- The kernel structures referenced by the file descriptor (the open file table entry and the v-node table entry) need to contain unique information for network communication protocols and hence need to look different from their local file and device counterparts.

- To the application programmer the sockets mechanism is accessed via a number of functions. These are:

  socket() - create a socket
  bind() - associate a socket with a network address
  connect() - connect a socket to a remote network address
  listen() - wait for incoming connection attempts
  accept() - accept incoming connection attempts

- In addition the functions **setsockopt()**, **getsockopt**(), **fcntl()** and **ioctl()** may be used to manipulate the properties of a socket.

- The **select()** function may be used to identify sockets with specific communication statuses.
- The function **close()** may be used to close a socket liaison.
- Data can be written to a socket using any of the functions **write(), writev(), send(), sendto()** and **sendmsg().**
- Data can be read from a socket using any of the functions **read(), readv(), recv(), recvfrom()** and **recvmsg().**

Sockets

- Dating back to the original **UNIX**, applications use a **common framework** to access both devices and files.
- Before an application performs **input** or **output** (**I/O**) operations, it calls **open()**, passing the file or device to be used as an argument.
- The open call returns a **file descriptor** which is used for future operations such as **read()** and **write()**. Finally **close()** is used to terminate the exchange.
- Because network protocols are more complicated, the file system model of I/O is not general enough to support network operations.
- For example, the read/write model assumes that all data is **stream data**. In a network environment, applications may want to send **variable-size datagrams**.
- **Sockets** are a generalization of the **UNIX file** access system designed to incorporate **network protocols**.
- One important difference between sockets and files is that the operating system **binds** file descriptors to a file or device when the open call creates the file descriptor.
- With sockets, applications can **specify the destination** each time they use the socket.
- Sockets permit applications to choose among several different network protocol families (e.g., Internet or Unix) as well as among different protocols within the same family (e.g., **UDP** or **TCP**).
- An application calls the socket system call to create a new socket. Arguments specify the protocol family and the type of service desired.
- An application desiring a socket for TCP/IP will request the protocol family **PF_INET**. The type of service will either **SOCK_STREAM** for **TCP** (reliable service) or **SOCK_DGRAM** for **UDP** (unreliable service).
- Formally a socket is defined by a group of four numbers, these are
    - The remote host identification number or address
    - The remote host port number
    - The local host identification number or address
    - The local host port number
- Internet applications are normally aware of all except the local port number, this is allocated when connection is established and is almost entirely arbitrary unlike the well known port numbers associated with popular applications.

- Applications issue calls to the **socket()** function to create sockets. The call has the form:

  s = socket (family, type, protocol);

  - The **family** argument specifies the **protocol family** (e.g., **PF_INET, PF_UNIX)**
  - The **type** argument specifies the abstract type of the communication desired. Valid types include **stream** and **datagram** (**SOCK_STREAM, SOCK_DGRAM**)
  - The **protocol** field specifies the **specific protocol** desired (e.g., **TCP** or **UDP**).

- The return value from is an integer value that may be used to refer to a socket in subsequent calls. It is called the **socket descriptor** or handle and is analogous to a file descriptor.
- All **three arguments** to the socket routine are **integers**. We will see how applications map high-level names to their integer counterparts.

Addressing

- To support multiple protocol families, UNIX represents all addresses within a **common framework**. Many of the API calls use these structures to specify and obtain address information.

- The generic **sockaddr** structure consists of two parts: a **sa_family** field that identifies the **protocol family** of the address and a **sa_data** field that contains the **actual address** encoded in a format understood by the specific protocol family:

```
struct sockaddr
{
        u_short sa_family;    /* address family */
        char sa_data[14];     /* up to 14 octets of address */
};
```

- In the **Internet family**, transport addresses are **6 octets** long: **4 octets** for the **Internet address** and **2 octets** for the **port number**. The appropriate structures to use for the Internet family are defined in **<netinet/in.h>**:

```
struct in_addr
{
    u_long s_addr;          /* Network byte order 32-bit IP Address */
}
```

```
struct sockaddr_in
{
  short int          sin_family;    /* Address family (AF_INET)            */
  unsigned short  int sin_port;     /* 16-bit Port number, network byte order     */
  struct in_addr     sin_addr;      /* 32-bit Internet address, network byte order */
  char               sin_zero[8];   /* unused                              */
};
```

- Here is a useful construct:

```
union sock
{
      struct sockaddr s;
      struct sockaddr_in i;
} sock;
```

- Applications issue calls to **gethostbyname()** to find the **Internet address** of a **host**. The call is as follows:

```
hostent = gethostbyname (name);
```

  - The argument **name** is a **string** containing the name of the machine.

- **gethostbyname()** maps the request into a **DNS** (**Domain Name System**) query that it sends to the **resolver** running on the local machine.

- The call returns a **pointer** to a **hostent structure** that contains the requested addresses. The fields of the **hostent** structure are as follows:

```
struct hostent
{
        char *h_name;          /* official name of host */
        char **h_alias;        /* list of aliases */
        int h_addrtype;        /* address type */
        int h_length;          /* length of address */
        char **h_addr_list;    /* list of addresses */
};
```

  - The **h_name** field is the official, fully qualified **domain name**.
  - The **h_alias** field is a null-terminated list of **alternate names** for the machine.
  - The **h_addrtype** field specifies what **protocol family** the address belongs to.
  - The **h_length** field specifies the **length** of an address.
  - The **h_addr_list** field is a null-terminated **array of addresses** for the **host**.

- After this returns the relevant fields need to be copied into an object of type **struct sockaddr_in** as shown below:

```
union sock
{
      struct  sockaddr s;
      struct  sockaddr_in i;
} sock;

struct in_addr      internet_address;
struct hostnet     *hp;

hp = gethostbyname(remote address);
memcpy(&internet_address, *(hp->h_addr_list),sizeof(struct in_addr);
```

- The relevant components of the **struct sockaddr_in** can now be filled in:

```
sock.i.sin_family = AF_INET;
sock.i.sin_port = required port number;
sock.i.sin_addr = internet_address;
```

In practice the final line of code above may be written thus

```
memcpy(&(sock.i.sin_addr),*(hp->h_addr_list),sizeof(struct in_addr));
```

- To determine the reason for a failed request, applications must inspect the **global variable h_error.**

- Upon return from the call, **h_error** indicates whether the request failed due to **server timeouts**, or whether the **DNS** returned an authoritative answer that the name **does not exist**.

- The related function **gethostbyaddr()** takes an **Internet address** as an argument and returns a corresponding **hostent** structure.

- Sometimes, it is necessary to use **Internet addresses** rather than **machine names**. **UNIX** supplies a set of routines for manipulating Internet addresses.

Address Manipulation Functions

- Several functions are provided for converting addresses between a 32-bit format and the dotted-decimal notation.

The function **inet-addr** takes a host address in dotted-decimal notation and returns the corresponding 32-bit IP address in network byte order. This function is obsolete.

- The **inet_aton()** and **inet_network()** routines take strings representing **Internet addresses** in **dotted notation** and return **numbers** suitable for use in **sockaddr_in** structures.

- The related routine **inet_ntoa()** takes an 32-bit IP address in **network byte order** Internet address and returns the corresponding address in dotted-decimal notation.

- Applications issue calls to **getservbyname()** to find the **integer port number** of the service. The call is as follows:

  servent = getservbyname (family, service);

  - The argument **family** is a **string** which specifies the **protocol family** (e.g., **TCP** or **UDP**).

  - **service** is a **string** which specifies the **name** of the service (e.g., **SMTP** or **TELNET**).

- The function call returns a **pointer** to a **servent structure** that contains the desired **port number**.

- The fields of the **servent** structure are as follows:

```
struct servent
{
        char *s_name; /* official name of service */
        char **s_aliases;       /* list of aliases */
        int s_port;             /* 16-bit port number */
        char *s_proto;          /* protocol family */
};
```

  - **s_name** is the **official name** of the service.
  - **s_aliases** is a null-terminated list of alternate names for the service.
  - **s_port** is the service's **16-bit port number**.
  - **s_proto** is the name of the **protocol family**.

- The related function **getservbyport()** takes a **port number** and **protocol family** as **arguments** and returns a **servent** structure.

- The **getprotobyname()** function takes the string form as a **protocol type** (e.g., **TCP**) and returns a **pointer to a structure** that contains the low-level **integer encoding** of the protocol.

- The call is as follows:

```
protoent = getprotobyname (protocol);
```

- The **protoent** structure is as follows:

```
struct protoent
{
        char *p_name;  /* official name of protocol */
        char **p_aliases;       /* list of aliases */
        int p_prot;     /* protocol type */
};
```

  - **p_name** is the official name for the protocol.
  - **p_aliases** is a null-terminated list of alternate names.
  - **p_proto** is the integer encoding for the **protocol type** that can be used as the third argument to the **socket()** call.

- The related routine **getprotobynumber()** takes the low-level **protocol number** as an argument and returns its corresponding **protoent** structure.

- Different computer architectures may store a multi-byte word in different orders. If the least significant byte is stored first, it is known as **little endian.**

- If the most significant byte is stored first, it is known as **big endian.**

- For any two computers to be able to communicate, they must use a common data format while transferring the multi-byte words.

- The Internet adopts the big-endian format. This representation is known as **network byte order** in contrast to the representation adopted by the host, which is called **host byte order.**

- It is important to remember that the values of **sin_port** and **sin_addr** in the **sockaddr_in** structure must be in the **network byte order**, since these values are communicated across the network.

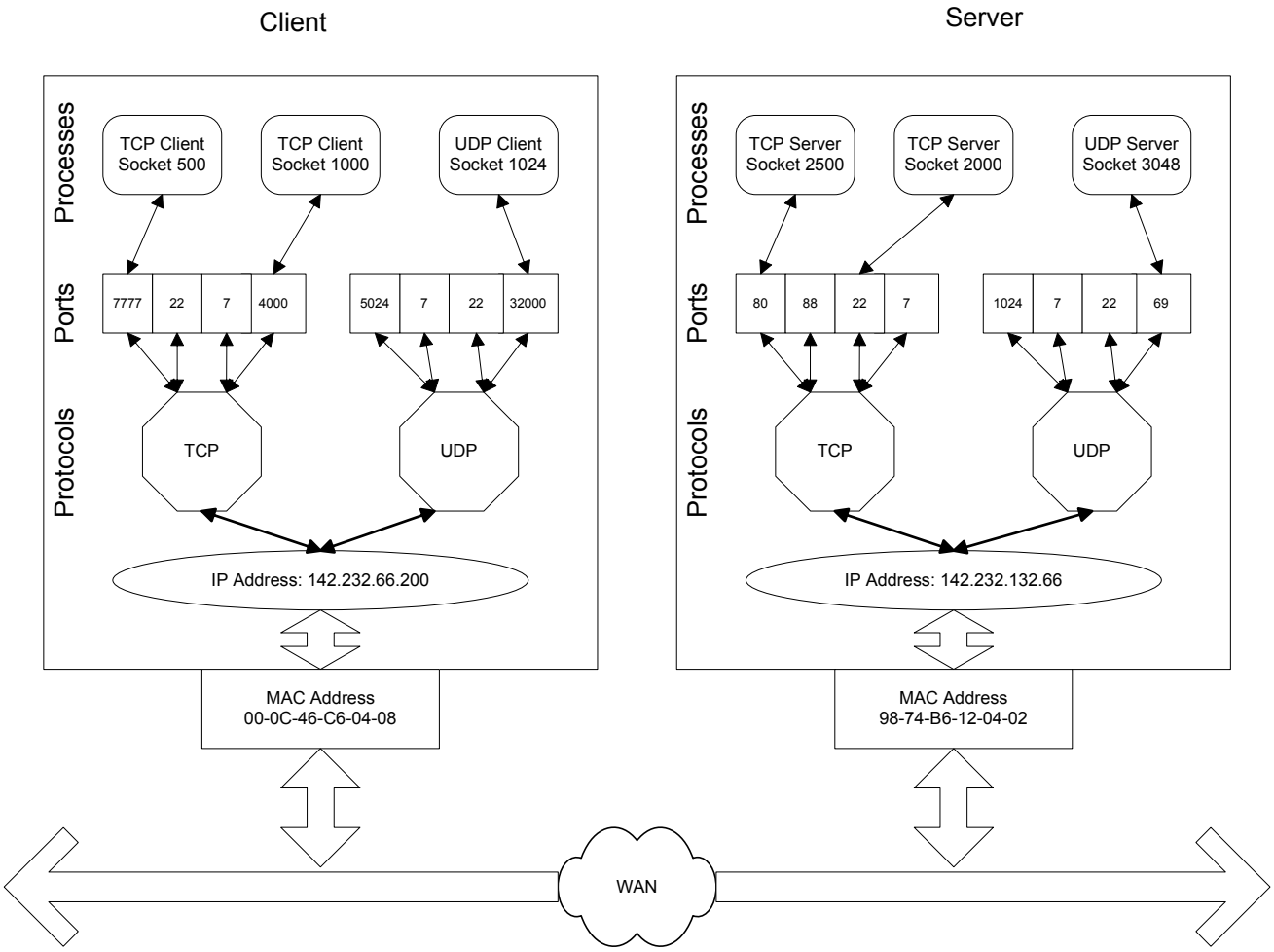
- Four functions are available to convert between the host and network byte order conveniently.

- Functions **htons** and **htonl** convert an unsigned short and an unsigned long, respectively, from the **host** to **network byte order**.

- Functions **ntohs** and **ntohl** convert an unsigned short and an unsigned long, respectively, from the **network** to **host byte order**.

- The appropriate prototypes are as follows:

    - **u_long htonl(u_long hostlong);**
    - **u-short htons(u_short hostshort);**
    - **u_long ntohl(u-long netlong);**
    - **u_short ntohs(u_short netshort);**



Communication System Calls

- The **connect()** function is used by a client program to establish communication with a remote entity. The prototype is

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

  int connect(int s, struct sockaddr *name, int namelen)

  - **s specifies the socket**

  - **name** points to an area of memory containing the address information and **namelen** gives the length of the address information block. This is done because addresses in some addressing domains are far longer than in others. connect() is normally only used for SOCK_STREAM sockets.

- The form of the **sockaddr** structure is as defined earlier.

- If the AF_INET domain is being used this will be the address of an object of type struct sockaddr_in defined earlier.
- The following is an example of how connect is called (after the sockaddr structure has been filled):

  connect(socket number, &sock.s, sizeof(struct sockaddr));

- When an application initially creates a **socket**, the **socket** is called **unbound** because it has no **addresses** associated with it.

- Communication cannot take place with an **unbound socket** because it is impossible to name the **process** that owns the socket.

- The **bind()** function **binds** an address to the **local end** of the socket as follows:

  result = bind (socket, sockaddr, sockaddrlen);

  - **socket** is the socket being bound.
  - **sockaddr** is a **pointer** to the **address** to which the socket should be bound.
  - **sockaddrlen** is the **size** of the **address**.

- Note that the related function **connect()** takes the same arguments but **binds** an address to the **remote end** of the socket.

- The prototype of **bind()** is

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

  int bind(int s, struct sockaddr *name, int namelen)

- This is similar to the **connect()** function except that, when binding in the **AF_INET** domain, the components of the **struct sockaddr_in** are filled in differently

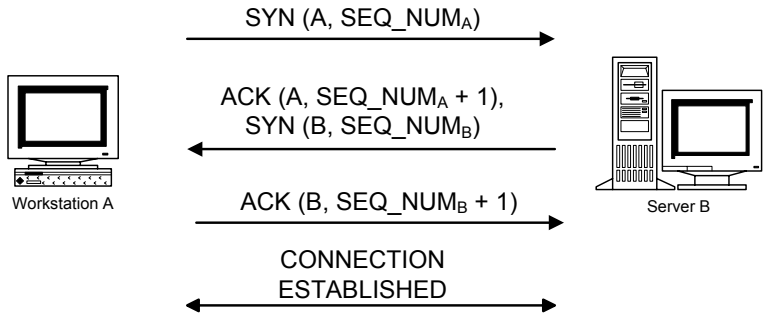
```
union sock
{
        struct sockaddr s;
        struct sockaddr_in i;
} sock;

sock.i.sin_family = AF_INET;
sock.i.sin_port = port number;
sock.i.sin_addr.s_addr = INADDR_ANY;
```

- The final value simply means that connections will be accepted from any remote host.

- If the socket is for a **TCP connection**, **connect()** initiates a three-way **handshake**.

TCP Connection Establishment



TCP 3-Way Handshake

- For a **connectionless** protocol such as **UDP**, the **operating system** simply records the **destination address** in a **control block** associated with the socket.

- **Severs** accept connections from **remote clients** but cannot use connect because they do not usually know the address of the remote client until **after** the client initiates a connection (see diagram).

- Applications use the **listen()** and **accept()** system calls to perform **passive opens**.
- Once an address has been bound to a socket it is then necessary to indicate the socket is to be listened to for incoming connection requests.
- This is done using the **listen()** function. Its prototype is:

  int listen(int s, int backlog)

  - **s specifies the socket.**
  - **backlog** specifies how long the queue of **unprocessed connection requests** can become before additional connection requests are discarded.
- listen() can only be associated with **SOCK_STREAM** or **SOCK_SEQPACKET** type sockets.
- Applications call **listen()** to indicate that they wish to accept **incoming connections**. The call is as follows:

  result = listen (socket, backlog);

- **listen()** only indicates that an application is willing to **accept connection** requests; applications call **accept()** to accept them.

- Once the **listen()** call has returned the **accept()** call should be issued, this will block until a connection request is received from a remote host.
- The prototype is

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

  int accept (int s, struct sockaddr *addr, int *addrlen)

  - **s** is the socket number
  - **addr** points to a struct sockaddr that will be filled in with the address of the remote (calling) system.
  - **addrlen** points to a location that will be filled in with the amount of significant information in the remote address, initially it should specify the size of the space set aside for the incoming address.
- The return value of the call is the number of a new socket descriptor that should be used for all subsequent communication with the remote host.
- You can, and should, carry on listening on the original socket number. There is no way of rejecting a connection request, you must accept it then close it.

- Here is an example of a call to accept:

  new_socket = accept (socket, from, sizeof(from));

  - **from** is a pointer to a **sockaddr** structure.
  - **fromlength** is a **pointer** to an **integer** containing the **length** of **from**.

- When the call to **accept()** returns, **from** contains the **address** of the **remote end** of the socket, and **new_socket** is in a connected state.

- It is useful to remember that if no connection requests are pending, **accept** will **block** until a connection request arrives.

Sending Data

- **UNIX** supplies **three ways** to send data to a **socket**.

- The **send()** call is similar to the **write()** call used to write to files, but has one **additional argument**:

  result = send (socket, message, length, flags);

  - **message** is a **pointer** to the data to be written.
  - **length** indicates the **data length**.
  - Applications use the **flags** field to send **flags** to the **underlying protocol**.
  - flags may be formed by ORing MSG_OOB and MSG_DONTROUTE. The latter is only useful for debugging.

- This call may only be used with **SOCK_STREAM** type sockets.

- Applications use **sendto()** when they want to specify the **remote destination** for each send as follows:

  result = sendto (socket, message, length, flags, dest, destlength);

- The first **four arguments** are the same as for the **send()** call.

  - **dest** is a **pointer** to the **sockaddr** structure.
  - **destlength** is the **length** of the structure.

- Applications use **sendto()** when they answer **datagram queries** sent by **arbitrary clients** because successive replies are sent to different destinations.

- **sendto()** can be used to send messages to sockets of **any type**.

- Applications use **sendmsg()** when the data they are sending does not reside in **contiguous memory locations** as follows:

  result = sendmsg (socket, messagestruct, flags);

  - **messagestruct** is a **structure** containing the **destination address** and describes where the actual data is **located**.

- Note that **write()** may be used in exactly the same way as it is used to write to files. This call may only be used with **SOCK_STREAM** type sockets.

Receiving Data

- Along with the functions for **sending** messages, **UNIX** also provides three functions for **receiving** data.
- Applications call **recv()** to read data from a **connected socket** as follows:

  result = recv (socket, message, length, flags);

- The arguments are the same as for the **send()** call; **UNIX** places data into the buffer pointed to by **message**.
- This call may only be used with **SOCK_STREAM** type sockets.
- Note that **read()** may be used in the exactly the same way as for reading from files. There are some complications with non-blocking reads.
- Applications call **recvfrom()** to receive messages on an **unconnected socket**.
- Because unconnected sockets do not have a **remote address** bound to them, the call also returns the **address** of the remote application that sent the message.
- The call is as follows:

  result = recvfrom (socket, message, length, from, fromlength);

- The **two additional** arguments **from** and **fromlength** are **pointers** to a **sockaddr** structure and its **length**.
- The **recvmsg()** function is analogous to **sendmsg()**. The call is as follows:

  result = rcvmsg (socket, messagestruct, flags);

  - Argument **messagestruct** is a **structure** that describes where **UNIX** should place the received data.

- **recvfrom()** and **recvmsg()** may be used with all types of sockets.
- **recvfrom()** is similar to **recv()** with extra parameters specifying the remote system address and **recvmsg()** receives into a message structure.
- The primary advantage of **rcvmsg()** is that an application can specify that data be placed in **noncontiguous memory locations**, possibly avoiding future **copies**.

Caveat Emptor

- It is a common beginner's error to think that data sent using a single write to a socket can be read at the other end using a single read.
- There is no guarantee that this will happen, data may well arrive in "dribs and drabs". If the application is a network server, it is not possible to impose any constraint on clients to write the data using a single call to write.
- Because TCP is stream oriented, the received data may come in multiple pieces of byte streams independent of how the data was sent at the other end.
- For example, when a transmitter sends 100 bytes of data in a single write call, the receiver may receive the data in two pieces-80 bytes and 20 bytes-or in three pieces-10 bytes, 50 bytes, and 40 bytes-or in any other combination.
- Thus the program has to make **repeated calls** to read until all the data has been received.
- The following code shows how to receive a message whose termination is indicated by the character pair CR+LF (a common Internet convention).

```
char    buff[BUFSIZ+1];        /* +1 for string terminator */
char    *bptr;
int     i;

bptr = buff;
while (TRUE)
{
        i = read (sd, bptr, BUFSIZ-(bptr-buff));
        if (i<=0)
                break;
        bptr += i;
        *bptr = '\0';   /* make what we've got into a string */
        if ((cp = strstr (buff, "\r\n"))
        {
                *cp = '\0';
                break;
        }
        if (bptr >= buff+BUFSIZ)
        {
                /* message too long */
        }
}
```

- The code also converts the received message into a string. If part of the next message is received in the same "chunk" of data as the end of the current message, this code is likely to loose the initial part of the next message.

Closing Connections

- Applications call **shutdown()** when they want to shut down all or part of a **full-duplex connection**. The call has the form:

  result = shutdown (socket, how);

  - **how** specifies which **direction** of the connection should be shutdown:

    0 - additional receives are shutdown.
    1 - additional sends are disallowed.
    2 - additional sends and receives will be disabled.

- Applications issue **close()** calls to close a socket as follows:

  result = close (socket);

Multiple Server Sessions

- In a server application it is important to ensure that multiple simultaneous sessions are handled correctly.

- Once a connection request has been accepted the program will be engaged in handling the associated dialogue, further connection requests will be held until the listener program gets round to issuing the accept() call again, thus only one client can be handled at a time.

- There are several ways of handling this issue. Under Unix the simplest solution is to fork a separate process to handle the client/server dialogue once a connection request has been received.

- Here is a code fragment that illustrates this idea:

```
listen (sd, 5);   /* at most 5 pending connections */
while (TRUE)
{
        nsd = accept (sd, &(work.s), &addrlen);
        pid = fork();
        if (pid == 0)
        {
              /* Child process handles the dialogue
                 using descriptor 'new_sd'
              */
```

```
            close(new_sd);
            exit(0);        /* end of child process */
        }
        else    close (new_sd);     /* parent won't be using 'new_sd' */
    }
```

- The overhead associated with forking a separate process every time a client connects can be significant. Alternative approaches use either multiplexing or multithreading.
- Multiplexing requires that all the code be capable of talking on several different sockets simultaneously.
- Typically there is be an array of descriptors and it is the responsibility of the program to keep track of the state of the dialogue associated with each active socket.
- Then poll() can be used to check for new connection requests. Multiplexing is the most complex and most efficient way of handling multiple dialogues.
- Multithreading is in many ways similar to forking a separate process for each dialogue with the exception that thread creation overhead is much lower than process creation overhead.
- In addition synchronization between threads is usually easier than synchronization between processes.
- The code provided shows an implementation of a simple **client/server** models using **TCP/IP**.

Summary

- The order of socket calls made by a (connection oriented) **server** is:
- **socket**

  Create the socket and identify the protocol and type of service
- **bind**

  Associate a server-side endpoint with the socket using well-known port
- **listen**

  Notify the kernel that the socket is to be placed in passive mode and specify a queue length
- **accept**

  get a connection (a new descriptor is returned), note that this is a blocking call.
- **send** and **receive**

  Receive clients request from the socket and send responses to the socket. Repeat until done.
- **close**

  When communication is complete, close the descriptor obtained from accept.
- The socket calls made by the (connection oriented) **client** are more simple.
- Bind isn't necessary, since the client need not advertise a port to be reached on instead, the endpoint is passed to the server when a connection is attempted.
- **socket**

  same as for server
- **connect**

  Request for a connection on the active socket passes the endpoint address to the remote host puts it in the data structure associated with the new descriptor given to the server.
- **send** and **receive**

Send requests to the socket and read responses from the server. Loop as necessary to complete communication.

- **close**

  Send an eof to server and close the socket.

TCP Client Algorithm

- Find IP address and protocol port number of server: library functions make it easy to look up server domain names and ports, as well as protocol numbers.

- When specifying the address of a server, the client must use the structure **sockaddr_in**, which requires the form of a 32-bit binary address. **inet_aton** and **gethostbyname** can do this for you (for dotted decimals or domain names, respectively).

- The latter returns the address of a structure called **hostent** that contains, among other things, the address in proper form.

- For the server port number look-up use **getservbyname**, which returns the port value in the proper form for **sockaddr_in**.

- To find a protocol number, use **getprotobyname**, which returns a **protoent** structrure containing the needed information.

- Allocate a socket: **socket** requires three arguments: the **protocol family**, the **service** to be used, and the particular **protocol**.

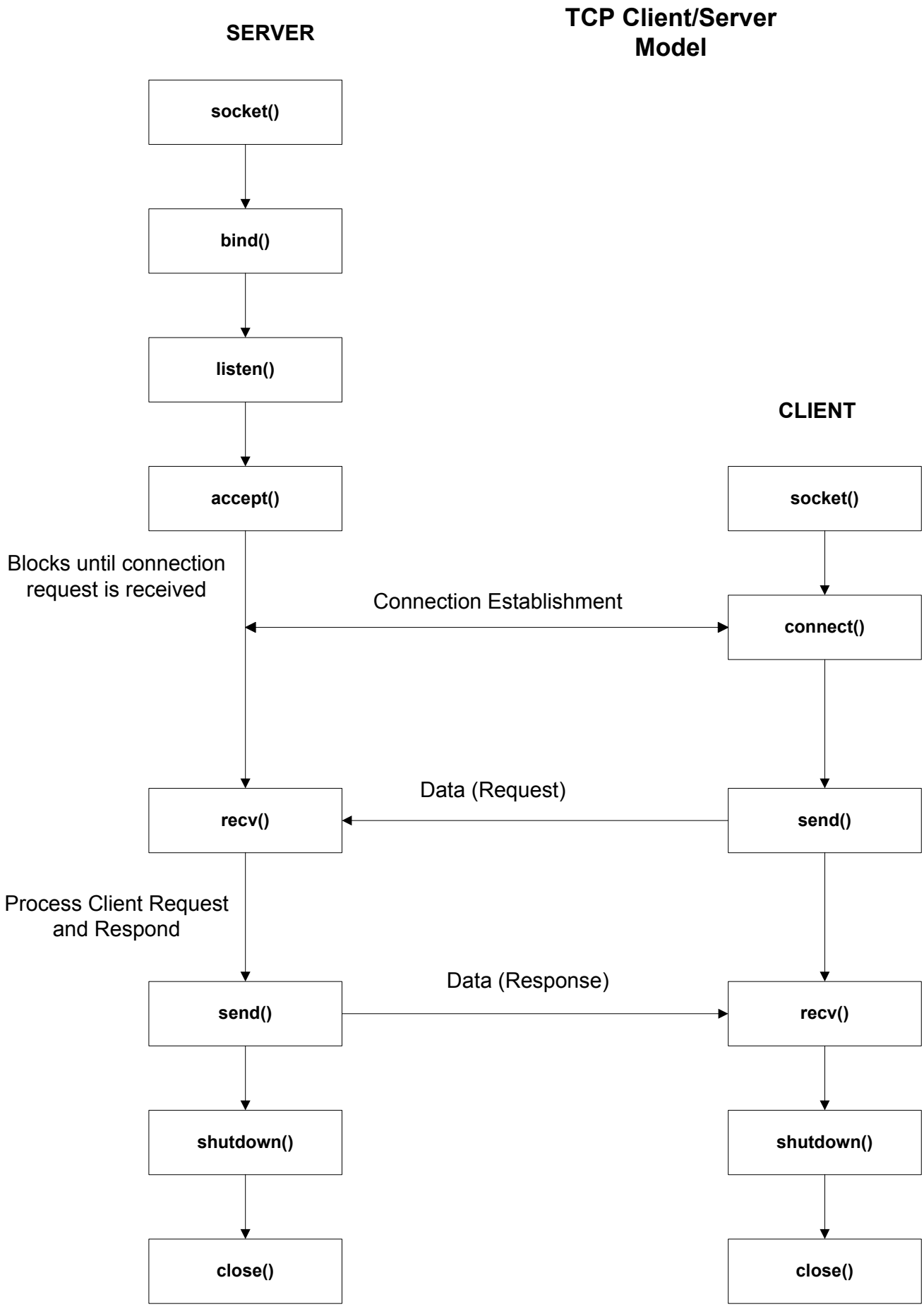
- Connect the socket to the server: use **connect**, which takes a socket descriptor, the address of a **sockaddr_in** structure (specifies remote endpoint), and the byte-length of that structure for arguments.

- The connect call makes sure that the socket is valid , fills in the remote endpoint address from **sock_addr**, chooses the local IP address and port # (if not already specified), and initiates TCP connection
- Send and receive data: this consists of a series of writes and reads. For each write, there should be several calls to read, since the information from the server may come in pieces.

- Close the connection. Servers can't close when they're done sending in case the client has more requests, and clients and can't close when they're done requesting in case the server has not sent all of its data.

- That is where **shutdown** comes into play. It takes two arguments: the socket descriptor and the direction of the close (0 for no more input, 1 for no more output, 2 for both).

- A partial close on one end eventually sends an "I'm done" signal to the other end so it knows to close when finished with the current operations.


TCP Client/Server Model
SERVER
CLIENT
socket()
bind()
listen()
accept()
socket()
Blocks until connection request is received
Connection Establishment
connect()
Data (Request)
recv()
send()
Process Client Request and Respond
Data (Response)
send()
recv()
shutdown()
shutdown()
close()
close()

UDP Client Algorithm

- The steps up to **connect** are the same for TCP above.
- Specify the server to which messages must be sent. UDP clients can either be connected or unconnected.
- Connected mode simply means that the socket is connected to a server so it can send and receive messages like TCP clients can.
- In unconnected mode, the client must specify a server every time it sends a message, which is helpful in simplifying communication with multiple servers.
- UDP is a connection-less protocol because messages can be lost, incomplete, or out of order. When using the **connect** call, unlike with TCP, no packet is exchanged is initiated and no validity tests are done when using **SOCK_DGRAM**.
- Even when the call succeeds, there is no guarantee that the remote endpoint address is valid or that the server is even reachable.
- Communicate with server: unlike with TCP, each call to read or write sends complete messages in one big packet. Therefore, only one call to read is necessary.
- Close the connection: close and shutdown can be used for this, but neither side actually informs the other application in the "connection" that this is going on. It is therefore important to design the client so that the remote end knows when to quit.
- Note: UDP is unreliable, so certain steps must be taken to improve on this. Ways to do this include packet sequencing, acknowledgements, timeouts, and re-transmission.

UDP Client/Server Model