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.

- 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.

- 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

- 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

- **Sockets** are a generalization of the **UNIX file** access system designed to incorporate **network protocols**.

- 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

- 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**).

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.

- 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                              */
};
```

- **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;
```

- 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

- 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.

- 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.

- 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));

- 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)

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

- 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.

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**.

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);

- 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**.

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);