Assignment No: 06

Title: UNIX Sockets

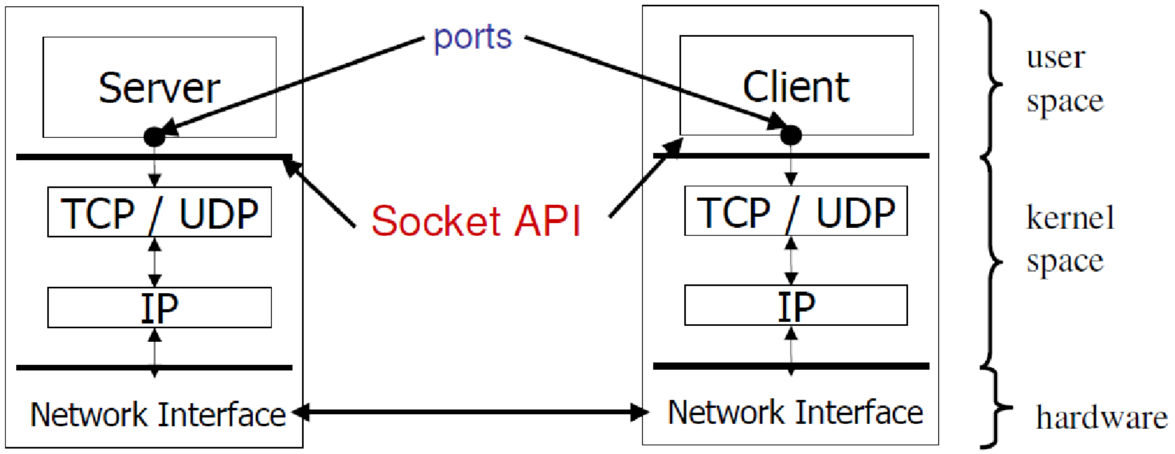
Problem Statement:.

UNIX Sockets: WAP program in C/C++ sockets API

a. TCP sockets
b. UDP sockets

Theory:

- Sockets are used for interprocess communication.
- Most of the interprocess communication follow a Client-Server
- Model, where client and server are two separate processes in itself.
- Server and Client exchange messages over the network through a common Socket API



* Port numbers: 1,024 -- 65,535 (2^{16} - 1)

Server Examples

• Web server (port 80)
• FTP server (20, 21)
• Telnet server (23)
• Mail server (25)

Client Examples

• Examples of client programs
– Web browsers, ftp, telnet, ssh

How does a client find the server?

- The IP address in the server socket address identifies the host
- The (well-known) port in the server socket address identifies the service, and thus implicitly identifies the server process that performs that service.

Examples of well know ports

- Port 7: Echo server
- Port 23: Telnet server
- Port 25: Mail server
- Port 80: Web server

What is an API ?

API expands as Application Programming Interface.
A set of routines that an application uses to request and carry out lower-level services performed by a computer's operating system.

What is a socket?

- An interface between application and network which is used for communication between processes
- Once configured the application can
  - pass data to the socket for network transmission
  - receive data from the socket (transmitted through the network by some other host)
- To the kernel, a socket is an endpoint of communication.
- To an application, a socket is a file descriptor that lets the application read/write from/to the network.
- Clients and servers communicate with each by reading from and writing to socket descriptors.
- Remember: All Unix I/O devices, including networks, are modeled as files.

Two essential types of sockets

SOCK_STREAM

- TCP
- connection-oriented
- reliable delivery
- in-order guaranteed
- bidirectional

SOCK_DGRAM

- UDP
- no notion of "connection" – app indicates dest. for each packet
- unreliable delivery
- no order guarantees
- can send or receive

Socket Primitives

Primitive	Meaning
SOCKET	Create a new communication end point
BIND	Attach a local address to a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Block the caller until a connection attempt arrives
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

socket()

The function *socket()* creates an endpoint for communication and returns a file descriptor for the socket. *socket()* takes three arguments:

- *domain*, which specifies the protocol family of the created socket. For example:
    - *AF_INET* for network protocol IPv4 or
    - *AF_INET6* for IPv6.
    - *AF_UNIX* for local socket (using a file).
- *type*, one of:
    - *SOCK_STREAM* (reliable stream-oriented service or Stream Sockets)
    - *SOCK_DGRAM* (datagram service or Datagram Sockets)
    - *SOCK_SEQPACKET* (reliable sequenced packet service), or
    - *SOCK_RAW* (raw protocols atop the network layer).
- *protocol* specifying the actual transport protocol to use. The most common are IPPROTO_TCP, IPPROTO_SCTP, IPPROTO_UDP, IPPROTO_DCCP. These protocols are specified in file *netinet/in.h*. The value *0* may be used to select a default protocol from the selected domain and type.

The function returns -1 if an error occurred. Otherwise, it returns an integer representing the newly assigned descriptor.

Prototype:

- int socket(int domain, int type, int protocol)

bind()

bind() assigns a socket to an address. When a socket is created using *socket()*, it is only given a protocol family, but not assigned an address. This association with an address must be performed with the bind() system call before the socket can accept connections to other hosts. *bind()* takes three arguments:

- *sockfd*, a descriptor representing the socket to perform the bind on.
- *my_addr*, a pointer to a *sockaddr* structure representing the address to bind to.
- *addrlen*, a *socklen_t* field specifying the size of the *sockaddr* structure.

Bind() returns 0 on success and -1 if an error occurs.

Prototype:

- int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen);

listen()

After a socket has been associated with an address, *listen()* prepares it for incoming connections. However, this is only necessary for the stream-oriented (connection-oriented) data modes, i.e., for socket types (*SOCK_STREAM*, *SOCK_SEQPACKET*). listen() requires two arguments:

- *sockfd*, a valid socket descriptor.
- *backlog*, an integer representing the number of pending connections that can be queued up at any one time. The operating system usually places a cap on this value.

Once a connection is accepted, it is dequeued. On success, 0 is returned. If an error occurs, -1 is returned.

Prototype:

- int listen(int sockfd, int backlog);

accept()

When an application is listening for stream-oriented connections from other hosts, it is notified of such events (cf. select() function) and must initialize the connection using the *accept()* function. The accept() function creates a new socket for each connection and removes the connection from the listen queue. It takes the following arguments:

- *sockfd*, the descriptor of the listening socket that has the connection queued.
- *cliaddr*, a pointer to a sockaddr structure to receive the client's address information.
- *addrlen*, a pointer to a *socklen_t* location that specifies the size of the client address structure passed to accept(). When *accept()* returns, this location indicates how many bytes of the structure were actually used.

The accept() function returns the new socket descriptor for the accepted connection, or -1 if an error occurs. All further communication with the remote host now occurs via this new socket.

Datagram sockets do not require processing by accept() since the receiver may immediately respond to the request using the listening socket.

Prototype:

- int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen)

connect()

The *connect()* system call *connects* a socket, identified by its file descriptor, to a remote host specified by that host's address in the argument list.

Certain types of sockets are *connectionless*, most commonly user datagram protocol sockets. For these sockets, connect takes on a special meaning: the default target for sending and receiving data gets set to the given address, allowing the use of functions such as send() and recv() on connectionless sockets.

connect() returns an integer representing the error code: 0 represents success, while -1 represents an error. Historically, in the BSD-derived systems, the state of a socket descriptor is undefined if the call to connect() fails (as it is specified in the Single Unix Specification), thus, portable applications should close the socket descriptor immediately and obtain a new descriptor with socket(), in the case the call to connect() fails.[3]

Prototype:

- int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen)

gethostbyname() and gethostbyaddr()

The *gethostbyname()* and *gethostbyaddr()* functions are used to resolve host names and addresses in the domain name system or the local host's other resolver mechanisms (e.g., /etc/hosts lookup). They return a pointer to an object of type *struct hostent*, which describes an Internet Protocol host. The functions take the following arguments:

- *name* specifies the name of the host. For example: www.wikipedia.org
- *addr* specifies a pointer to a *struct in_addr* containing the address of the host.
- *len* specifies the length, in bytes, of *addr*.
- *type* specifies the address family type (e.g., AF_INET) of the host address.

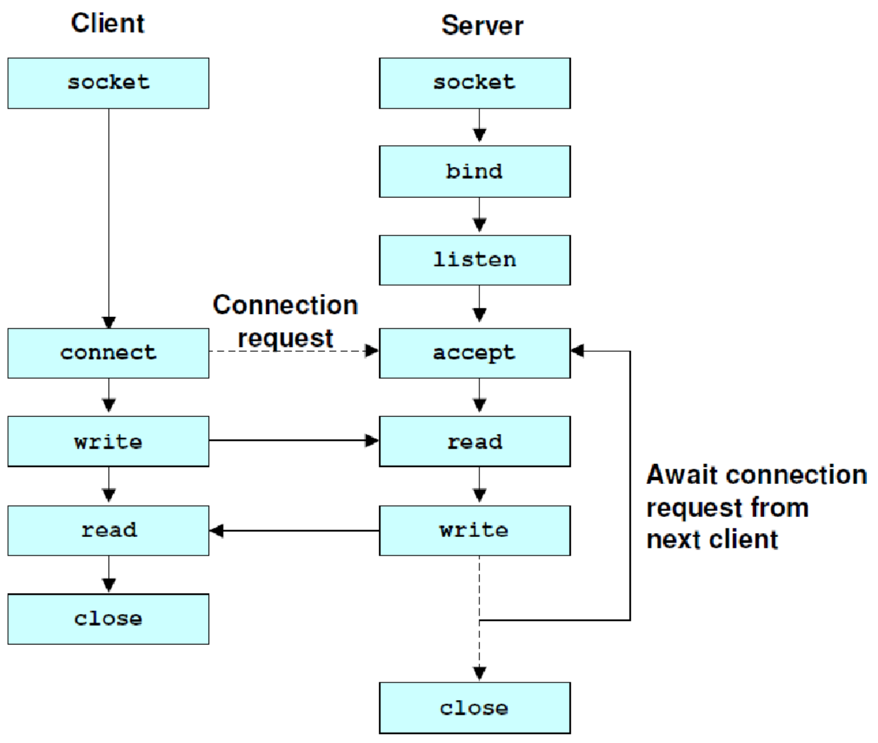
The functions return a NULL pointer in case of error, in which case the external integer *h_errno* may be checked to see whether this is a temporary failure or an invalid or unknown host. Otherwise a valid *struct hostent ** is returned.

These functions are not strictly a component of the BSD socket API, but are often used in conjunction with the API functions. Furthermore, these functions are now considered legacy interfaces for querying the domain name system. New functions that are completely protocol-agnostic (supporting IPv6) have been defined. These new function are getaddrinfo() and getnameinfo(), and are based on a new *addrinfo* data structure.

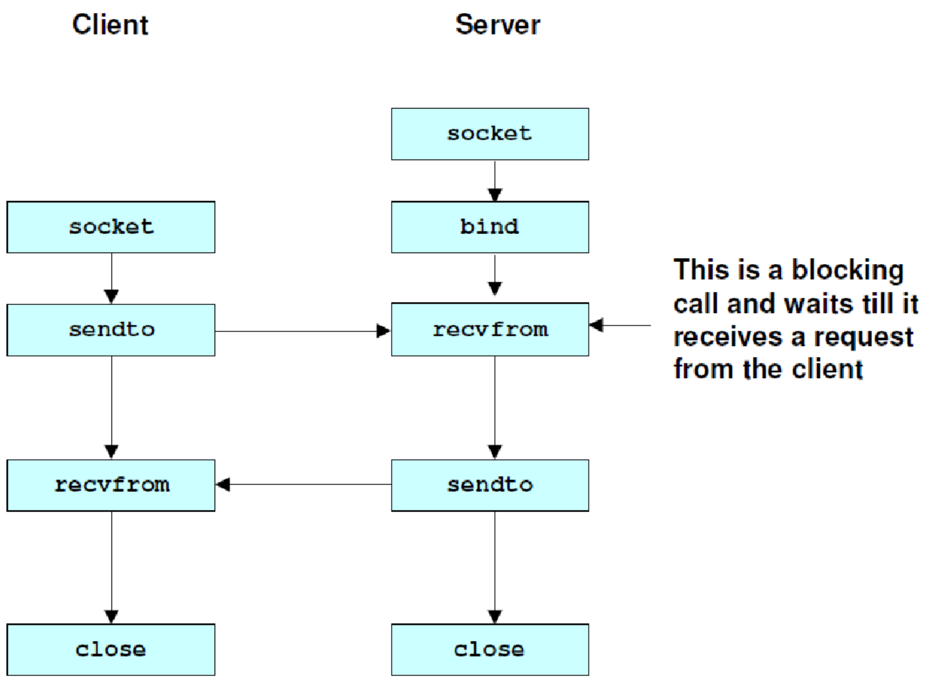
Prototypes:

- struct hostent *gethostbyname(const char *name)
- struct hostent *gethostbyaddr(const void *addr, int len, int type)

Socket programming *with TCP*



Socket programming *with UDP*


Client
Server
socket
bind
socket
sendto
recvfrom
This is a blocking call and waits till it receives a request from the client
recvfrom
sendto
close
close

Conclusion: TCP & UDP socket programs are studied and executed.

/**Mclient.c**/

```
#include"stdio.h"
#include"stdlib.h"
#include"sys/types.h"
#include"sys/socket.h"
#include"string.h"
#include"netinet/in.h"
#include"netdb.h"

#define PORT 5561
#define BUF_SIZE 2000

int main(int argc, char**argv) {
 struct sockaddr_in addr, cl_addr;
 int sockfd, ret;
 char buffer[BUF_SIZE];
 struct hostent * server;
 char * serverAddr;

 if (argc < 2) {
  printf("usage: client < ip address >\n");
  exit(1);
 }

 serverAddr = argv[1];
 sockfd = socket(AF_INET, SOCK_STREAM, 0);
 if (sockfd < 0) {
  printf("Error creating socket!\n");
  exit(1);
 }
 printf("Socket created...\n");

 memset(&addr, 0, sizeof(addr));
 addr.sin_family = AF_INET;
 addr.sin_addr.s_addr = inet_addr(serverAddr);
 addr.sin_port = PORT;

 ret = connect(sockfd, (struct sockaddr *) &addr, sizeof(addr));
 if (ret < 0) {
  printf("Error connecting to the server! : : %d\n",ret);
  exit(1);
 }
 printf("Connected to the server @  %s\n",serverAddr);

 memset(buffer, 0, BUF_SIZE);
 printf("Enter your message(s): ");

 while (fgets(buffer, BUF_SIZE, stdin) != NULL) {
  ret = sendto(sockfd, buffer, BUF_SIZE, 0, (struct sockaddr *) &addr,
sizeof(addr));
  if (ret < 0) {
   printf("Error sending data!\n\t-%s", buffer);
  }
  ret = recvfrom(sockfd, buffer, BUF_SIZE, 0, NULL, NULL);
  if (ret < 0) {
   printf("Error receiving data!\n");
  } else {
   printf("Received: ");
   fputs(buffer, stdout);
   printf("\n");
  }
 }

 return 0;
}
```

/**Mserver.c**/

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <errno.h>

#define PORT 5561
#define BUF_SIZE 2000
#define CLADDR_LEN 100

char *itoaa(int val, int base);

int main()
{
      struct sockaddr_in addr, cl_addr;
       int sockfd, len, ret, newsockfd;
       char buffer[BUF_SIZE];
       pid_t childpid;
       char clientAddr[CLADDR_LEN];
       int num, rem, sum;
       char *str;

       sockfd = socket(AF_INET, SOCK_STREAM, 0);

       if (sockfd < 0)
       {
             printf("Error creating socket!\n");
             exit(1);
       }

       printf("Socket created...\n");

       memset(&addr, 0, sizeof(addr));
       addr.sin_family = AF_INET;
       addr.sin_addr.s_addr = INADDR_ANY;
       addr.sin_port = PORT;

       ret = bind(sockfd, (struct sockaddr *) &addr, sizeof(addr));
       if (ret < 0)
       {
             printf("Error binding!\n");
             exit(1);
       }
       else
            printf("Binding done...\n");

       printf("Waiting for a connection...@ port no : %d\n",PORT );

       listen(sockfd, 5);
       for (;;)  //infinite loop
       {
             len = sizeof(struct sockaddr_in);
             newsockfd = accept(sockfd, (struct sockaddr
*)&cl_addr,(socklen_t *)&len);
             if (newsockfd < 0)
             {
                  printf("Error accepting connection!\n");
                  exit(1);
             }
             else
                  printf("Connection accepted from ");

             inet_ntop(AF_INET, &(cl_addr.sin_addr), clientAddr,
CLADDR_LEN);


             printf("Port %d of %s
Client\n",ntohs(cl_addr.sin_port),inet_ntoa(cl_addr.sin_addr));

             if ((childpid = fork()) == 0)  //creating a child process
             {
                     close(sockfd);
                  //stop listening for new connections by the main process.
                  //the child will continue to listen.
                  //the main process now handles the connected client.

                   for (;;)
                   {
                        memset(buffer, 0, BUF_SIZE);

                        ret = recvfrom(newsockfd, buffer, BUF_SIZE, 0,
(struct sockaddr *) &cl_addr, (socklen_t *)&len);

                        if(ret < 0)
                        {
                             printf("Error receiving data!\n");
                             exit(1);
                        }
                        else
                             printf("Received data from Port No %d of
Client %s : %s\n ", ntohs(cl_addr.sin_port),clientAddr, buffer);

                        num=atoi(buffer);
                        sum=0;
                        while(num>0)
                        {
                             sum = sum + (num % 10);
                             num = num / 10;
                        }

                        strcat(buffer,"            = sum of digits = ");
                        str=itoaa(sum,10);
                        strcat(buffer,str);

                        ret = sendto(newsockfd, buffer, BUF_SIZE, 0,
(struct sockaddr *) &cl_addr, len);

                        if (ret < 0)
                        {
                             printf("Error sending data!\n");
                               exit(1);
                        }
                        else
                             printf("\tSent data to %s on Port No %d :
%s\n", clientAddr,ntohs(cl_addr.sin_port), buffer);

                        printf("----------------------------------------
----------------------------------------------------------------\n");
                  }
            }
            close(newsockfd);
       }
      return(0);
}

char *itoaa(int val, int base)
{
      static char buf[32] = {0};

      int i = 30;

      for( ; val && i ; --i, val /= base)
           buf[i]="0123456789abcdef"[val % base];

      return &buf[i+1];
}
```