Lecture 7: Introduction to Sockets

References for Lecture 7:

API (Application Programming Interface) – the interface available to the programmer for using the communication protocols. The API is depend to the OS + the programming language.

In this course, we discuss the socket API. With sockets, the network connection can be used as a file. Network I/O is, however, more complicated than file I/O because:

- Asymmetric. The connection requires the program to know which process it is, the client or the server.
- A network connection that is connection-oriented is somewhat like opening a file. A connectionless protocol doesn’t have anything like an open.
- A network application needs additional information to maintain protections, for example, of the other process.
- There are more parameters required to specify a network connection than the file I/O. The parameter have different formats for different protocols.
- The network interface must support different protocols. These protocols may use different-size variable for addresses and other fields.

Socket Address
Because different protocols can use sockets, a generic socket address structure is defined. The socket address structure of the protocol is recast as the generic one when calling socket functions.

/* Generic Socket Address Structure, length=16*/
<sys/socket.h>
struct socketaddr {
    unit8_t sa_len;
    sa_family_t sa_family; /* address family: AF_XXX value */
    char sa_data[14]; /*up to 14 types of protocol-specific address */
};

/* Ipv4 Socket Address Structure, length=16*/
<netinet/in.h>
struct in_addr {
    in_addr_t s_addr; /* 32-bit IPv4 address, network byte ordered */
};
struct sockaddr_in{
    unit8_t sin_len; /* length of structure (16 byte) */
    sa_family_t sin_family; /*AF_INET*/
    in_port_t sin_port; /* 16-bit TCP or UDP port number, network byte ordered */
    struct in_addr sin_addr; /* 32-bit IPv4 address, network byte ordered */
    char sin_zero[8]; /* unused – initialize to all zeroes */
};

/* Ipv6 Socket Address Structure, length=24*/
<stdio/in.h>
struct in6_addr {
    unit8_t s6_addr[16]; /* 128-bit IPv6 address, network byte ordered */
} ;
#define SIN6_LEN /* required for compile-time tests */
struct sockaddr_in6{
    unit8_t sin6_len; /* length of this structure (24 byte) */
    sa_family_t sin6_family; /*AF_INET6*/
    in_port_t sin6_port; /* 16-bit TCP or UDP port number, network byte ordered */
    unit32_t sin6_flowinfor; /* priority + flow label, network byte ordered */
    struct in6_addr sin6_addr; /* IPv6 address, network byte ordered */
};

/*Xerox NS Socket Address Structure*/
struct sockaddr_ns{
    u_short sns_family /* AF_NS */
    struct ns_addr sns_addr /* the 12_byte XNS address */
    char sns_xer0[2] /* unused */
};

/* Unix Domain Socket Address Structure, variable length, you need to pass the actual length as an argument */
struct sockaddr_un {
    short sun_family; /*AF_UNIX*/
    char sun_path[108]; /* pathname */
};

Different protocols have different Socket Address Structure with different length. Unix has one set of network system calls to support such protocols. So, the socket address structure of the specific protocol is recast as the generic one when using Unix network system calls. That is also why a generic socket address structure is defined. ANSI C has a simple solution, by using void *, but the socket functions (announced in 1982) predate ANSI C.

**Byte Ordering Routines**
The network protocols use big-endian format. The host processor many not, so there are functions to convert between the two:
<netinet/in.h>

uint16_t htons(uint16_t host16val); -- convert 16-bit value from host to network order, used for the port number.
uint32_t htonl(uint32_t host32val); -- convert 32-bit value from host to network order, used for the IPv4 address.
uint16_t ntohs(uint16_t network16val); -- convert 16-bit value from network to host order.
uint32_t ntohl(uint32_t network32val); -- convert 32-bit value from network to host order.

Note1: These functions works equally well for signed integer.

2: These functions don’t do anything on systems where the host byte order is big-endian, however, they should be used for portability.

Byte Manipulation Routines
The following functions can be used to initialize or compare memory locations:

System V:
void *memset(void *dest, int c, size_t nbytes); -- writes value c (converted into an unsigned char) into nbytes bytes of dest. Returns dest.
void *memcpy(void *dest, const void *src, size_t nbytes); -- copies nbytes of src to dest. Returns dest.
int memcmp(const void *ptr1, const void *ptr2, size_t nbytes); -- compare nbytes of the two strings, Returns 0 if they match, >0 if ptr1 > ptr2, < 0 if ptr1 < ptr2.

BSD 4.3:
bzero(char *dest, int nbytes); -- writes NULL into nbytes bytes of dest.
bcopy(char *src, char *dest, int nbytes); -- similar with memcpy.
int bcmp(char *ptr1, char *ptr2, int nbytes); -- similar with memcmp.

Notice
1) the difference between the Byte Manipulation Functions and the string functions with name beginning with str, such as strcpy() and strcmp. 2) the meaning of void *dest 3) the meaning of const void *src 4) return value of memset() or memcpy()is the same as the first argument.

Address Conversion Routines
An internet address is written as: “192. 43. 234.1”, saved as a character string. The functions require their number as a 32-bit binary value. To convert between the two, some functions are available:
<arpa/inet.h>

int inet_aton(const char *strptr, struct in_addr *addrptr);
-- returns 1, 0 on error. Converts from a dotted-decimal string to a network address.
unsigned long inet_addr(char *strptr);
-- Converts from a dotted-decimal string to 32-bit integer as return value.
char *inet_ntoa(struct in_addr addrptr);
-- Returns a pointer to a dotted-decimal string, given a valid network address.

Notice: inet_addr() and inet_aton are similar. Use inet_aton in your code.

This can also be done with two functions that support both IPv4 and IPv6 protocol:
int inet_pton(int family, const char *strptr, void *addrptr);
-- returns 1 if OK, 0 if invalid input format, -1 on error.

Const char *inet_ntop(int family, const void *addrptr, char *strptr, size_t len);
-- returns pointer to result if OK, NULL on error.

**Procedure of Socket Programming**

In order to communicate between two processes, the two processes must provide the formation used by TCP/IP (or UDP/IP) to exchange data. This information is the 5-tupe:

{protocol, local-addr, local-process, foreign-addr, foreign-process}.
Several network systems calls are used to specify this information and use the socket.

![Socket Diagram](image)

**Connection-Orieted Protocol (TCP)**

**Questions:**

1) Why is there no bind() for client?
2) Why does server need listen()?
3) Why can UDP not use read() or write()?
Connectionless Protocol (UDP)

Notice the difference between TCP and UDP!

The fields in the 5-tuple are set by:

<table>
<thead>
<tr>
<th>Connection-oriented server</th>
<th>Protocol</th>
<th>Local-addr</th>
<th>Local-port</th>
<th>Foreign-addr</th>
<th>Foreign-port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection-oriented client</td>
<td>socket()</td>
<td>bind()</td>
<td></td>
<td>accept()</td>
<td></td>
</tr>
<tr>
<td>Connectionless server</td>
<td>socket()</td>
<td>bind()</td>
<td></td>
<td>recvfrom()</td>
<td></td>
</tr>
<tr>
<td>Connectionless client</td>
<td>socket()</td>
<td>bind()</td>
<td></td>
<td>sendto()</td>
<td></td>
</tr>
</tbody>
</table>

Socket Function Calls:

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

int socket(int family,  int type,  int protocol); -- returns a socket descriptor, -1 on error.
```

- **family** – AF_INET = Ipv4 protocol
  - AF_INET6 = Ipv6 protocol
  - AF_LOCAL = Unix domain protocol
  - AF_NS = Xerox NS protocol
  - AF_ROUTE = Access routing table information in kernel
  - AF_KEY = Access cryptographic/security key table in kernel

- **type** – SOCK_STREAM = stream socket (TCP)
  - SOCK_DGRAM = datagram socket (UDP)
  - SOCK_RAW = raw socket (IP, also used with AF_ROUTE + AF_KEY)

- **protocol** – usually set to 0 and protocol is chosen based on family and type fields.

Returns – socket descriptor, equivalent to 5-tuple. For your convenience, always associate a 5-tuple association with a socket descriptor.

```
int connect(int sockfd,  const struct sockaddr *servaddr,  socklen-t addrlen);
```

- **sockfd** – socket descriptor, value returned by socket( ).
- **servaddr** – server’s socket, contains the IP address and port number of the server.
**addrlen** – length of the structure sockaddr, to support different protocols.

Notes:
1) **connect()** sets the local and foreign addresses and port numbers.
2) It initializes an active request for connection between the client and the server.
3) It blocks until the connection is established.
4) Mainly used connection-oriented protocols, e.g., TCP.
5) Connect() can also be used for connectionless protocols. In this case,
   5.1) it doesn’t establish a connection.
   5.2) returns immediately.
   5.3) when sending data, the destination socket address are not specified explicitly. Use send() or write() instead of sendto().
   5.4) datagrams from only the designated address can be received. Use read() or recv() instead of recvfrom().
   5.5) connected UDP socket is much faster than unconnected UDP socket.
6) The socket can late be connected to another destination (or become unconnected) by calling connect() again. Use family=AF_UNSPEC to unconnect the socket.

```c
int bind(int sockfd, const struct sockaddr *myaddr, socklen-t addrlen); -- returns 0 if OK, -1 on error.
```

**sockfd** – value returned by socket().

**myaddr** – protocol-specific address. Values depend on usage.

**addrlen** – size of the address structure. In order to support different protocols.

Notes:
1) bind assigns the local IP address and port number to the socket.
2) Servers call bind() to their well-known port to the socket. If a server omits the call to bind(), the kernel selects an ephemeral port.
3) Normally a TCP client doesn’t bind an IP address and port to its socket. The kernel chooses the IP address and port number when connect is called. A UDP client uses bind() so that incoming packets can be delivered to it by the kernel.
4) Using a wildcard, INADDR_ANY, for the IP address allows the kernel to select the IP address.
5) Using 0 for the port allows the kernel to choose the port; you can call getsockname() to know a kernel assigned port number—see example 1.
6) Reserved ports: See RFC 1700 for the full list of well-known ports or visit our FAQs. See /etc/services or \WINNT\system32\drivers\etc\services for local definition of reserved ports.

<table>
<thead>
<tr>
<th>Reserved ports</th>
<th>Internet Port Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Internet Applications: FTP, TELNET, TFTP, SMTP</td>
<td>1--255</td>
</tr>
<tr>
<td>Reserved for future</td>
<td>256-511</td>
</tr>
<tr>
<td>Ports assigned by *<em>int rresvport(int <em>aport)</em></em></td>
<td>512--1023</td>
</tr>
<tr>
<td>Let system assign port number</td>
<td>0</td>
</tr>
<tr>
<td>Ports automatically assigned by system</td>
<td>1024--5000</td>
</tr>
<tr>
<td>User-developed Server</td>
<td>&gt; 5000</td>
</tr>
</tbody>
</table>
int listen(int sockfd, int backlog); -- returns 0 if OK; -1 on error.
sockfd – value returned by socket().
backlog – number of connections the kernel should queue for the socket. Queued while waiting for an accept().
Notes:
1) listen( ) converts the socket into a passive socket, which allows a server to accept incoming connection requests when the server is processing the present request.
2) It has been tested on my computer that listen( ) must be used for connection-oriented server. See example 1.

int accept(int sockfd, stuct sockaddr *cliaddr, int * addrlen); -- returns a new sockfd, -1 on error.
sockfd – value returned by socket().
cliaddr – returns the socket address of the client.
addrlen – a value-result argument. The calling process sets a value. The function uses this value and then updates it. In this case the server passes the size of the sockaddr structure and the number of bytes placed in cliaddr is returned.
Notes:
1) accept() blocks until a connection request is received from a client. sockfd is unchanged but a new sockfd is returned that specifies the connection to this client. Why ?
2) 3 return values: new socket fd, the client’s IP address and port number are placed in cliaddr, length of the client’s socket address stored in addrlen.
3) In example 1, accept(sk, 0, 0) means that “I don’t want to know the client’s socket address”.

int close(int sockfd); -- returns 0 if OK; -1 on error. Closes the socket. Data already written to the socket that hasn’t been sent yet will be sent and then the connection will be terminated. The call to close() returns immediately, even if unacknowledged data remains.

Miscellaneous Socket Functions:
Int shutdown(int sockfd, int howto) – returns 0 if OK, -1 on error.
Unlike close, shutdown allows the system to close the socket without waiting for the other process to also close the socket. Operations are:
howto = SHUT_RD(0) – receive on more data on the socket. Buffered data is discarded.
SHUT_WR(1) – send no more data on the socket.
SHUT_RDWR(2) – closes both halves, cannot send or receive data

Besides using read and write on connected sockets, special functions are available for sending and receiving data over a socket.
For connection-oriented:
ssize_t recv(int sockfd, void *buff, size_t nbytes, int flags);   
ssize_t send(int sockfd, const void *buff, size_t nbytes, int flags);
For connectionless:
ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags struct sockaddr *from, socklen_t *addrlen);
ssize_t sendto(int sockfd, void *buff, size_t nbytes, int flags struct sockaddr *to, socklen_t addrlen);
-- returns number of bytes read or written, -1 on error.
-- possible flags are:
MSG_DONTROUT – bypass the routing table lookup. Can be used for LANs.
MSG_DONTWAIT – converts the functions into a nonblocking call.
MSG_OOB – send or receive out-of-band data as opposed to normal data. TCP allows only one byte of OOB data.
MSG_PEEK – receives data, but does not remove it.
MSG_WAITALL – doesn’t return from a recv() or recvfrom() until the specified number of bytes have been read or an error occurs.

int getpeername(int sockfd, struct sockaddr *peer, socklen_t *addrlen); – returns 0 if OK, -1 on error.
peer – obtains the opposite process socket address: IP address and port number.

int getsockname(int sockfd, struct sockaddr *localaddr, socklen_t *addrlen) – returns 0 if OK, -1 on error.
localaddr – obtains IP address and port number of the local process. If bind() uses port number 0, getsockname() will get a new assigned port number – see Note 4 of bind().

#include <unistd.h>
int gethostname(char *name, size_t namelen); – returns 0 if OK, -1 on error.
name -- places the name of a host in name, For example, “gigastar”.

Given a machine name or host name, we can obtain the IP address and vice versa.
#include <netdb.h>
struct hostent *gethostbyname(const char *hostname); – returns NULL on error. gethostbyname() returns a hostent structure containing the hostname’s IP addresses and nicknames from local hosts file /etc/hosts or /etc/inet/hosts.
struct hostent *gethostbyaddr(const char *addr, size_t leng, int family); – returns NULL on error.
-- addr is pointer to a struct in_addr or struct in6_addr.

struct hostent{
  char *h_name; /* official name of the host, such as ece, gigastar.eng.wayne.edu*/
  char **h_aliases; /* an array of aliases (names) such as ece.eng.wayne.edu*/
  int h_addrtype; /* host address type: AF_INET or AF_INET6 */
  int h_length; /* length of the address*/
  char **h_addr_list; /* pointer to a list of IP addresses*/
#define h_addr h_addr_list[0] /* first address in the list, for backward compatibility */
}
The array of pointers h_addr_list[0], h_addr_list[1], and so on, are not pointers to characters but are pointers to structures of type in_addr which is defined in <netinet/in.h>.

%hostname --- get the host name of your computer
%ifconfig –a4 --- get the IP address of your computer
%ping –a gigastar --- check if a remote host “gigastar” is reachable. –a :get the IP address of the host.

See local hosts file /etc/hosts or /etc/inet/hosts or \WINNT\system32\drivers\etc\hosts for local DNS. Do you know how to speed up your surfing speed?
Two types of server

**Concurrent server** – forks a new process, so multiple clients can be handled at the same time.
**Iterative server** – the server processes one request before accepting the next.

**Concurrent Server**

```c
listenfd = socket(...);
bind(listenfd,...);
listen(listenfd,...)
for ( ; ; ) {
    connfd = accept(listenfd, ...);
    If (( pid = fork()) == 0) {    /* child*/
        close(listenfd);
        /* process the request */
        close(connfd);
        exit(0);
    }
    close(connfd);    /* parent*/
}
```

**Iterative Server**

```c
listenfd = socket(...);
bind(listenfd,...);
listen(listenfd,...)
for ( ; ; ) {
    connfd = accept(listenfd, ...);
    /* process the request */
    close(connfd);
}
```

**Client**

```c
sockfd = socket(...);
connect(sockfd,...)
/* process the request */
close(sockfd);
```
Summary:

1) Difference between TCP and UDP.
2) Difference between stream and datagram.
3) Difference between concurrent server and iterative server.
4) Socket fd = 5-tuple.
5) Socket address = IP address + port number.
6) Difference between port number and process number (pid).
7) How provide protocols: socket().
8) How many typical protocols: socket(family, type, protocol).
9) How to provide local socket address: bind() or connect().
10) How to provide opposite/peer socket address: sendto() or connect().
11) How to get opposite socket address: accept, recvfrom().
12) How to check local or opposite socket address: getsockname(), getpeername().
13) How to know a host name and its IP address: gethostname(), gethostbyname(), gethostbyaddr().

1) TCP \(\iff\) Connection oriented \(\iff\) stream
2) UDP \(\iff\) Connectionless \(\iff\) datagram
Example 1: A concurrent server echoes what client users type on keyboard, using connection-oriented TCP stream

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

/* This program creates a well-known Internet domain socket. Each time it accepts a connection, it forks a child to print out messages from the client. It then goes on accepting new connections itself. */

main()
{
    struct sockaddr_in local;
    int sk, len=sizeof(local), rsk;

    /* Create an Internet domain stream socket */
    sk=socket(AF_INET, SOCK_STREAM, 0);

    /* Construct and bind the name using default values */
    local.sin_family=AF_INET;
    local.sin_addr.s_addr=INADDR_ANY;
    local.sin_port=0; /* Let the kernel to assign a port; see Note 6 for bind( ) */
    bind(sk, &local, sizeof(local));

    /* Find out and publish the assigned name */
    getsockname(sk, &local, &len);
    printf("Socket has port %d\n", local.sin_port);

    /* Start accepting connections */
    listen(sk, 5) /* Declare the willingness to accept connection. Tested that it is necessary for connection-oriented server */
    while (1) {
        rsk=accept(sk, 0, 0); /* Accept new request for connection */
        if (fork()==0) { /* Create one child to serve each client */
            dup2(rsk, 0); /* Connect the new connection to "stdin" */
            execl("recho", "recho", 0); }
        else
            close(rsk);
    }
}
```

Can you find some errors in server.c?  Hint: child process without exit(0) will fork grand child process.
# Process "recho". This program implements the service provided by the server. It is a child of the server and receives messages from the peer through "stdin" */

main()
{
    char buf[BUFSIZ];
    while (read(0, buf, BUFSIZ) > 0) printf("%s", buf);
    printf("***** ending connection *****\n");
}

/* This program creates a socket and initiates a connection with the server socket given in the command line. Messages are read from the standard input until EOF (ctrl-d) is encountered. */

main(argc, argv)
int argc;
char *argv[];
{
    char buf[BUFSIZ], *gets();
    struct sockaddr_in remote;
    int sk;
    struct hostent *hp, *gethostbyname();

    sk=socket(AF_INET, SOCK_STREAM, 0);

    remote.sin_family=AF_INET;
    hp=gethostbyname(argv[1]); /* Get host number from its symbolic name */
    bcopy(hp->h_addr, (char *)&remote.sin_addr, hp->h_length);
    remote.sin_port=atoi(argv[2]);
    connect(sk, &remote, sizeof(remote)); /* Connect to the remote server */

    /* Read input from "stdin" and send to the server */
    while (read(0, buf, BUFSIZ) > 0)
        write(sk, buf, strlen(buf));
    close(sk);
}

Question: Why does the length field, sin_len, never need to be filled?
Example 2: For connectionless UDP datagrams.

------------------------------------------------------------------sender.c------------------------------------------------------------------
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define MSG "ECE 5650 is a great course!!!"

main(argc, argv)
int argc;
char *argv[];
{  int sk;
    char buf[BUFSIZ];
    struct sockaddr_in remote;
    struct hostent *hp, *gethostbyname();

    sk=socket(AF_INET, SOCK_DGRAM, 0);
    remote.sin_family=AF_INET;
    hp = gethostbyname(argv[1]);
    bcopy(hp->h_addr, &remote.sin_addr.s_addr, hp->h_length);
    remote.sin_port=atoi(argv[2]);
    sendto(sk, MSG, strlen(MSG), 0, &remote, sizeof(remote));
    read(sk, buf, BUFSIZ);
    printf("%s\n", buf);
    close(sk);
}

------------------------------------------------------------------replier.c------------------------------------------------------------------
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define MSG "Are you sure? I doubt!!!"

main()
{ struct sockaddr_in local, remote;
    int sk, rlen=sizeof(remote), len=sizeof(local);
    char buf[BUFSIZ];
    sk=socket(AF_INET, SOCK_DGRAM, 0);
    local.sin_family=AF_INET;
    local.sin_addr.s_addr=htonl(INADDR_ANY);
    local.sin_port=htons(0);
    bind(sk, &local, sizeof(local));
    getsockname(sk, &local, &len);
    printf("socket has port %d\n", local.sin_port);
    recvfrom(sk, buf, BUFSIZ, 0, &remote, &rlen);
    printf("%s\n", buf);
    sendto(sk, MSG, strlen(MSG), 0, &remote, sizeof(remote));
    close(sk);
}
Example 3: Unix Domain Streams (without name)

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>

/* This program accepts several numbers from the command line and forks a child to compute the square and cube of those numbers. Results are returned back to the parent and printed out. */
main(argc, argv)
int argc;
char *argv[];
{
    int i, cc, pid, sk[2];
    int magic;
    struct {float d, h;} num;

    /* Create a socketpair */
    socketpair(AF_UNIX, SOCK_STREAM, 0, sk);

    if (fork()==0) {   /* This is the child process */
        close(sk[0]);
        pid=getpid();
        /* Read the number and return the results until the parent ended */
        while (read(sk[1], &magic, sizeof(magic))>0) {
            printf("Child[%d]: the magic number is %d\n", pid, magic);
            num.d=magic * magic;
            num.h=magic * num.d;
            write(sk[1], &num, sizeof(num));
        }
        close(sk[1]);
        exit(1); }

    /* This is the parent process */
    close(sk[1]);
    for (i=1; i<argc; i++) {  /* Write the numbers to children */
        magic=atoi(argv[i]);
        write(sk[0], &magic, sizeof(magic));
    }
    for (i=1; i<argc; i++) {  /* Read the results from children */
        if (read(sk[0], &num, sizeof(num))<0)
            break;
        printf("parent: square=%f, cub=%f\n", num.d, num.h);
    }
    close(sk[0]);
}

Note: Unix domain unnamed stream is similar to PIPE, but is bi-directional. Remember all sockets are full-duplex.
```
Example 4: Unix Domain Datagram (with name)

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

/* This program creates a UNIX domain datagram socket, binds a name to it, then reads from the socket. */
#define NAME "name-socket-number"   /* define the name of the socket */

main()
{
    struct sockaddr_un local;
    int sk;
    char buf[BUFSIZ];

    sk=socket(AF_UNIX, SOCK_DGRAM, 0);  /* Create an UNIX domain datagram socket from which to read */
    local.sun_family=AF_UNIX;       /* Define the socket domain */
    strcpy(local.sun_path, NAME);     /* Define the socket name */
    bind(sk, &local, strlen(NAME)+2);  /* Bind the name to the socket, the actual length of socket address is passed*/
    read(sk, buf, BUFSIZ);            /* Read from the socket */
    printf("%s\n", buf);                /* Print the message */
    unlink(NAME);                  /* Delete the i-node bound to the socket */
    close(sk);
}
```

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

/* This program sends a message in datagram to the receiver. */
#define MSG "ECE 5650 is a great course!!!"
#define NAME "name-socket-number"

main()
{
    int sk;
    struct sockaddr_un remote;
    char buf[BUFSIZ];

    sk=socket(AF_UNIX, SOCK_DGRAM, 0);  /* Create an UNIX domain datagram socket on which to send */
    remote.sun_family=AF_UNIX;            /* Construct the name of socket to send to */
    strcpy(remote.sun_path, NAME);
    sendto(sk, MSG, strlen(MSG), 0, &remote, strlen(NAME)+2);  /* send message */
    close(sk);
}
```

Note: Datagram client doesn’t need to use bind().
Example 5 Client-Server Example:
/* Example of server using TCP protocol. */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#define SERV_UDP_PORT 6000
#define SERV_TCP_PORT 6000
#define SERV_HOST_ADDR "192.43.235.6" /* host addr for server */

char*pname;
main(argc, argv)
int argc;
char*argv[];
{
    int sockfd, newsockfd, clilen, childpid;
    struct sockaddr_in cli_addr, serv_addr;
   pname = argv[0];
    /*  Open a TCP socket (an Internet stream socket).   */
    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        err_dump("server: can't open stream socket");
    /*  * Bind our local address so that the client can send to us.  */
    bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(SERV_TCP_PORT);
    if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
        err_dump("server: can't bind local address");
    listen(sockfd, 5);
    for ( ; ; ) /* Wait for a connection from a client process. This is an example of a concurrent server. */
        clilen = sizeof(cli_addr);
        newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
        if (newsockfd < 0) err_dump("server: accept error");
        if ( (childpid = fork()) < 0)err_dump("server: fork error");
        else if (childpid == 0) { /* child process */
            close(sockfd); /* close original socket */
            str_echo(newsockfd); /* process the request */
            exit(0);
        }
        close(newsockfd); /* parent process */
    }
}
/* Example of client using TCP protocol. */

#include "inet.h"

main(argc, argv)
int  argc;
char* argv[];
{
    int sockfd;
    struct sockaddr_in serv_addr;
    pname = argv[0];

    /* Fill in the structure "serv_addr" with the address of the server that we want to connect with. */
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
    serv_addr.sin_port        = htons(SERV_TCP_PORT);

    /* Open a TCP socket (an Internet stream socket). */
    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        err_sys("client: can't open stream socket");

    /* Connect to the server. */
    if (connect(sockfd, (struct sockaddr *) &serv_addr,
                sizeof(serv_addr)) < 0)
        err_sys("client: can't connect to server");

    str_cli(stdin, sockfd);   /* do it all */
    close(sockfd);
    exit(0);
}
/* * Read a stream socket one line at a time, and write each line back to the sender. Return when the connection
is terminated. */

#define MAXLINE 512
str_echo(sockfd)
int sockfd;
{
  int n;
  char line[MAXLINE];
  for ( ; ; ) {
    n = readline(sockfd, line, MAXLINE);
    if (n == 0) return; /* connection terminated */
    else if (n < 0)
      err_dump("str_echo: readline error");
    if (writen(sockfd, line, n) != n)
      err_dump("str_echo: writen error");
  }
}

/* * Read the contents of the FILE *fp, write each line to the stream socket (to the server process), then read a
line back from the socket and write it to the standard output. Return to caller when an EOF is encountered on
the input file. */

#include <stdio.h>
#define MAXLINE 512
str_cli(fp, sockfd)
register FILE *fp;
register int sockfd;
{
  int n;
  char sendline[MAXLINE], recvline[MAXLINE + 1];
  while (fgets(sendline, MAXLINE, fp) != NULL) {
    n = strlen(sendline);
    if (writen(sockfd, sendline, n) != n)
      err_sys("str_cli: writen error on socket");

    /* Now read a line from the socket and write it to our standard output. */
    n = readline(sockfd, recvline, MAXLINE);
    if (n < 0)  err_dump("str_cli: readline error");
    recvline[n] = 0; /* null terminate */
    fputs(recvline, stdout);
  }
  if (ferror(fp))
    err_sys("str_cli: error reading file");
}