Lecture #12 – Network Sockets (Chapter 16)

- Overview

A socket is a file descriptor for a network connection.

By using sockets, we can open a connection to any machine on the network, and transmit data over that network by using normal file semantics (read, and write).

There are many different types of sockets, but the 2 most popular are Internet sockets (AF_INET or AF_INET6), and local or UNIX sockets (AF_UNIX).

Internet sockets account for well over 95% of all sockets, so we will concentrate on them.

There are 3 basic types of Internet sockets:

1. Stream sockets (SOCK_STREAM)
2. Datagram sockets (SOCK_DGRAM)
3. Raw sockets (SOCK_RAW)

Stream sockets:

These are reliable two-way connected communication streams.
Based on the TCP protocol (i.e. connection oriented).

Data sent over these sockets will arrive at their destination error free and in the same order as they were sent.

Many socket applications use stream sockets (including telnet, http, etc).

Datagram sockets:

These are connectionless sockets that are considered unreliable.
Based on the UDP protocol (i.e. connectionless).
If you send a datagram, it may arrive, or it may not.
It may arrive in a different order than it was transmitted.
Datagram sockets do not require an established "open" connection.

Raw sockets:

Raw sockets are a powerful version of sockets outside the scope of our discussion.

- socket

  "socket" creates a socket descriptor based on the arguments given.

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

  "domain" should be AF_INET
  "type" should be SOCK_STREAM or SOCK_DGRAM

  "protocol" should just be 0 to allow the system to choose based on "type"
  "protocol" can also be obtained using "getprotobyname()"

  "socket" returns to you a socket descriptor that can be used in later system calls like a file descriptor. Therefore, this is the closet equivalent to an "open" for network connections.

- shutdown

  int shutdown(int sockfd, int how);

  "how" is one of:

SHUT_RD	read disabled
SHUT_WR	write disabled
SHUT_RDWR	read and write disabled

  Can be used in conjunction with "close". Allows us to shutdown traffic in one or both directions regardless of how many other references exist to this socket (i.e. via dup).

- Byte Ordering

  There are two byte orderings in use by systems today: most significant byte first, or least significant byte first (i.e. Big Endian versus Little Endian).

  Network byte order stores the most significant byte first.

  Some machines use this technique for internal storage while others do not.

  Some things need to be in Network Byte Order to function properly (i.e. port number, IP address, etc).

  There are functions to provide conversions from Host Byte Order to Network Byte Order.

  htons() - Host to Network Short
  htonl() - Host to Network Long
  ntohs - Network to Host Short
  ntohl - Network to Host Long

- Address Formats

  Address structure:

```
struct sockaddr {
        sa_family_t     sa_family;      //  address family like AF_INET
        char            sa_data[14];    //  14 bytes of protocol address
};
```

  Internet address structure;

```
struct sockaddr_in {
        sa_family_t     sin_family;     //  address family
        in_port_t       sin_port;       //  port number
        struct in_addr  sin_addr;       //  Internet address
};
```

  Note: Sometimes includes a sin_zero field which should be all zeroes
  Note: We can interchange struct sockaddr_in * with struct sockaddr *
  Note: sin_port and sin_addr must be in Network Byte Order

```
struct in_addr {
        in_addr_t       s_addr;
};
```

  Internet address structure (IPV6):

```
struct sockaddr_in6 {
        sa_family_t     sin6_family;
        in_port_t       sin6_port;
        uint32_t        sin6_flowinfo;
        struct in6_addr sin6_addr;
        uint32_t        sin6_scope_id;
};

struct in6_addr {
        uint8_t         s6_addr[16];
};
```

- IP Addresses

  There are a bunch of functions to help manipulate IP addresses into the "sin_addr" field.

  For example:

```
a.sin_addr.s_addr = inet_addr("10.1.50.12");
```

Another:

```
int     inet_aton(const char *cp, struct in_addr *inp);     //  ascii to network

inet_aton("10.1.50.12", &(my_addr.sin_addr));
```

Reverse:

```
printf("%s\n", inet_ntoa(ina.sin_addr));
```

IPV6 safe routines:

```
//  Converts a binary address in network byte order into a text string
char    *inet_ntop(int domain, const void *addr, char *str, socklen_t size);

//  Converts a text string into a binary address in network byte order
int     inet_pton(int domain, char *str, void *addr);

Note:  INET_ADDRSTRLEN is big enough to hold IPV4 text string
Note:  INET6_ADDRSTRLEN is large enough to hold IPV6 text string
```

- bind

"bind" allows you to associate your socket with a port on your local machine. This is useful if you are going to be the "server" side of the connection.

```
int     bind(int fd, struct sockaddr *addr, int addrlen);
```

"fd" is the socket descriptor returned by 'socket'
"addr" contains information about the port and IP addresses
"addrlen" is the size of the "addr" structure

Example:

```
#define PORTNUM   9713

int                     s;
struct sockaddr_in      addr;

s = socket(AF_INET, SOCK_STREAM, 0);

addr.sin_family         = AF_INET;
addr.sin_port           = htons(PORTNUM);
addr.sin_addr.s_addr    = INADDR_ANY;
memset (&(addr.sin_zero), '\0', 8);
```

```
bind(s, (struct sockaddr *) &addr, sizeof(struct sockaddr));
```

Note: Port numbers < 1024 are reserved for root
Note: Port numbers must be less than 65535

Example: (to avoid "address already in use")

```
int     yes = 1;

if (setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1) {
        perror("setsockopt");
        exit(1);
}
```

- connect

"connect" takes a socket descriptor and connects to a remote IP / port

int connect(int s, struct sockaddr *addr, int addrlen);

Example:

```
#define DEST_IP     "10.1.1.50"
#define DEST_PORT 23

int                 s;
struct sockaddr_in  dest;

if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0)      {
        perror("socket");
        exit(1);
}
dest.sin_family         = AF_INET;
dest.sin_port           = htnos(DEST_PORT);
dest.sin_addr.s_addr    = inet_addr(DEST_IP);
memset(&(dest.sin_zero), '\0', 8);

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

- listen

  "listen" is used to specify that we want to wait for connections on a "bound" port.

  int listen(int s, int backlog);

  "s" is the open socket descriptor
  "backlog" is the number of connections allowed on the incoming queue

  Note: Most systems silently limit the "backlog" to something like 20 or 30.

- accept

  "accept" is used to wait for an incoming "connect" from a client

  int accept(int s, void *addr, int *addrlen);

  "s" is the open socket descriptor
  "addr" will be a pointer to a local struct sockaddr_in (to store info about the connection0
  "addrlen" is the length of struct sockaddr_in

  Example: (some error checking left out to simplify)

```
int                 s, cs;
struct sockaddr_in  myaddr;
struct sockaddr_in  clientaddr;
int                 sin_size;

s = socket(AF_INET, SOCK_STREAM, 0);

myaddr.sin_family = AF_INET;
myaddr.sin_port = htons(MYPORT);
myaddr.sin_addr.s_addr = INADDR_ANY;
memset(&myaddr.sin_zero, '\0', 8);

bind(s, (struct sockaddr *) &myaddr, sizeof(struct sockaddr));
listen(s, 35);

sin_size = sizeof(struct sockaddr_in);
cs = accept(s, (struct sockaddr *) & client_addr, &sin_size);
```

- Sample server (echo)

```
int main(int argc, char **argv)
{
        int                     listenfd, connfd;
        pid_t                   pid;
        socklen_t               clen;
        struct sockaddr_in      caddr, saddr;

        listenfd = socket(AF_INET, SOCK_STREAM, 0);

        bzero(&saddr, sizeof(saddr));

        saddr.sin_family                = AF_INET;
        saddr.sin_addr.s_addr           = htonl(INADDR_ANY);
        saddr.sin_port                  = htons(SERV_PORT);

        bind(listenfd, (struct sockaddr *) &saddr, sizeof(saddr));
        listen(listenfd, 35);

        while (1)
        {
                clen = sizeof(caddr);

                connfd = accept(listenfd, (struct sockaddr *) &caddr, &clen);

                if ((pid = fork()) == 0)        /*  child */
                {
                        close(listenfd);
                        str_echo(connfd);
                        exit(0);
                }
                else if (pid < 0)
                        perror("fork failed");

                close(connfd);                          /*  parent */
        }
}

void str_echo(int fd)
{
        ssize_t         n;
        char            line[MAXLINE];

        while (1)
        {
```

```
                if ((n = readline(fd, line, MAXLINE)) == 0)
                        return;

                fputs(line, stdout);
                writen(fd, line, n);
        }
}
```

- Sample client (echo)

```
int main(int argc, char **argv)
{
        int                     sockfd;
        struct sockaddr_in      saddr;

        if (argc != 2)
        {
                fprintf(stderr, "usage: %s <server IP>\n", argv[0]);
                exit(1);
        }

        sockfd = socket(AF_INET, SOCK_STREAM, 0);

        bzero(&saddr, sizeof(saddr));

        saddr.sin_family = AF_INET;
        saddr.sin_port = htons(SERV_PORT);
        inet_pton(AF_INET, argv[1], &saddr.sin_addr);

        connect(sockfd, (struct sockaddr *) &saddr, sizeof(saddr));

        str_client(stdin, sockfd);

        exit(0);
}

void str_client(FILE *fp, int sockfd)
{
        char    sendline[MAXLINE];
        char    recvline[MAXLINE];

        while (fgets(sendline, MAXLINE, fp) != NULL)
        {
                writen(sockfd, sendline, strlen(sendline));

                if (readline(sockfd, recvline, MAXLINE) == 0)
```

```
                {
                        fprintf(stderr, "server quit\n");
                        exit(1);
                }

                fputs(recvline, stdout);
        }
}
```

- Common Routines for echo server

```
ssize_t readn(int fd, void *vptr, size_t n)
{
        size_t          nleft;
        ssize_t         nread;
        char            *ptr;

        ptr = vptr;
        nleft = n;

        while (nleft > 0)       {
                if ((nread = read(fd, ptr, nleft)) < 0)   {
                        if (errno == EINTR)
                                nread = 0;
                        else
                                return(-1);
                }
                else if (nread == 0)
                        break;

                nleft -= nread;
                ptr += nread;
        }

        return (n - nleft);
}

ssize_t writen(int fd, const void *vptr, size_t n)
{
        size_t          nleft;
        ssize_t         nwritten;
        const char      *ptr;

        ptr = vptr;
        nleft = n;
```

```
        while (nleft > 0)  {
                if ((nwritten = write(fd, ptr, nleft)) <= 0)  {
                        if (errno == EINTR)
                                nwritten = 0;
                        else
                                return(-1);
                }

                nleft -= nwritten;
                ptr   += nwritten;
        }

        return n;
}

static ssize_t myread(int fd, char *ptr)
{
        static int      read_cnt = 0;
        static char     *read_ptr;
        static char     read_buf[MAXLINE];

        if (read_cnt <= 0)  {
                again: if ((read_cnt = read(fd, read_buf, sizeof(read_buf))) < 0)  {
                        if (errno == EINTR)
                                goto again;

                        return(-1);
                }
                else if (read_cnt == 0)
                        return(0);

                read_ptr = read_buf;
        }

        read_cnt--;     *ptr = *read_ptr++;
        return(1);
}

ssize_t readline(int fd, void *vptr, size_t maxlen)
{
        size_t  n, rc;
        char    c, *ptr;

        ptr = vptr;

        for (n = 1; n < maxlen; n++)
```

```
        {
                if ((rc = myread(fd, &c)) == 1)
                {
                        *ptr++ = c;

                        if (c == '\n')
                                break;

                }


                else if (rc == 0)
                {
                        if (n == 1)
                                return(0);
                        else
                                break;
                }
                else
                        return(-1);
        }

        *ptr = 0;
        return(n);
}
```