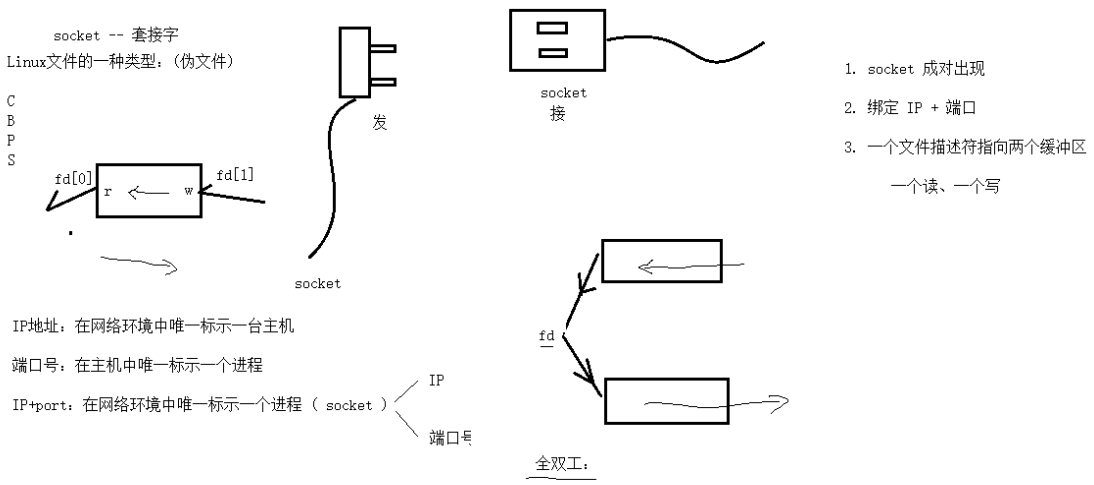
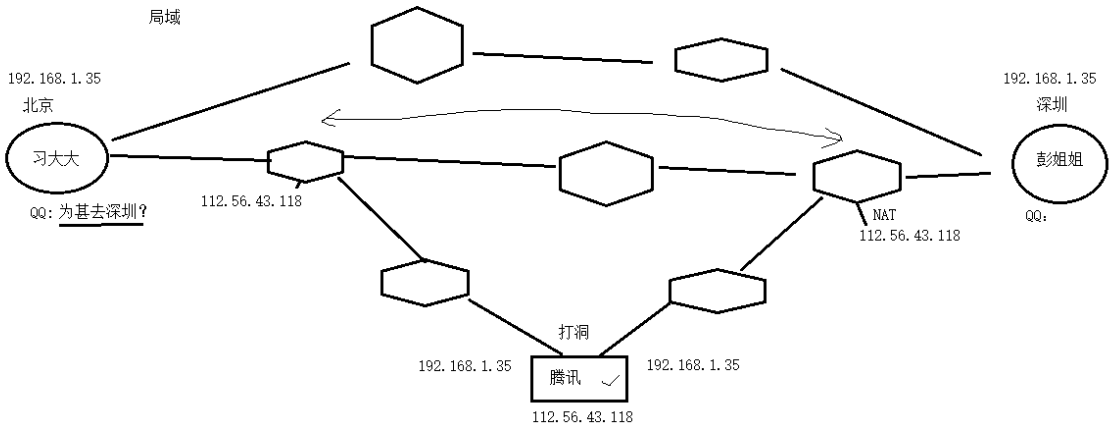
socket



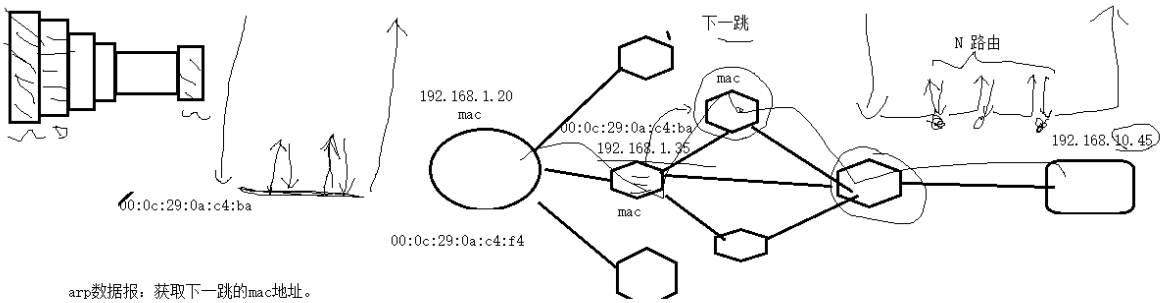
有两个缓冲区

打洞

公-公： 直接访问
公-私： NAT映射
私-公： NAT映射
私-私： NAT映射 、打洞机制



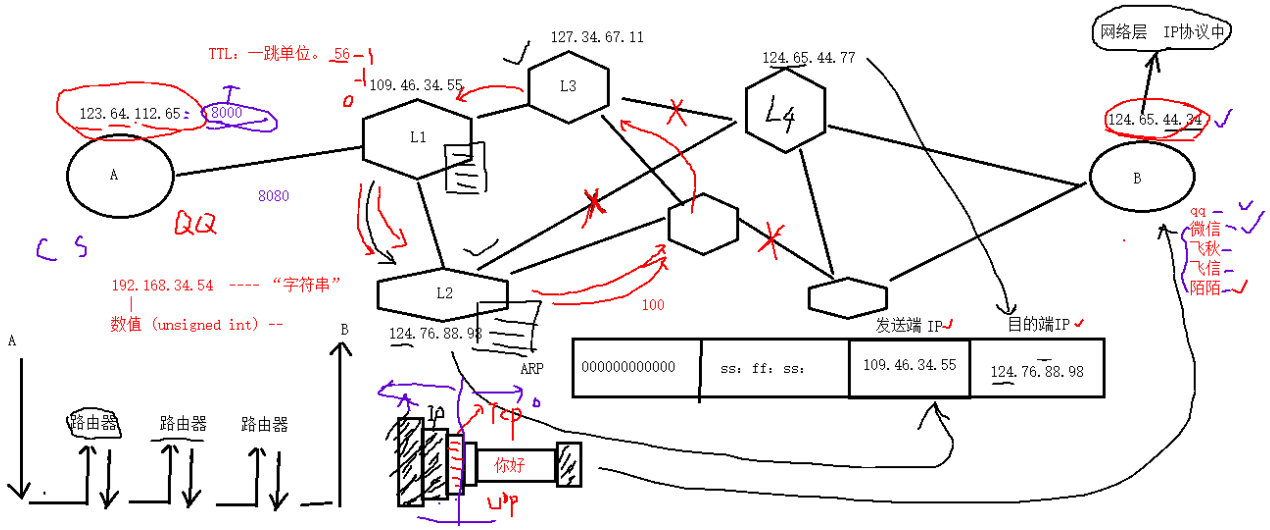
以太网帧



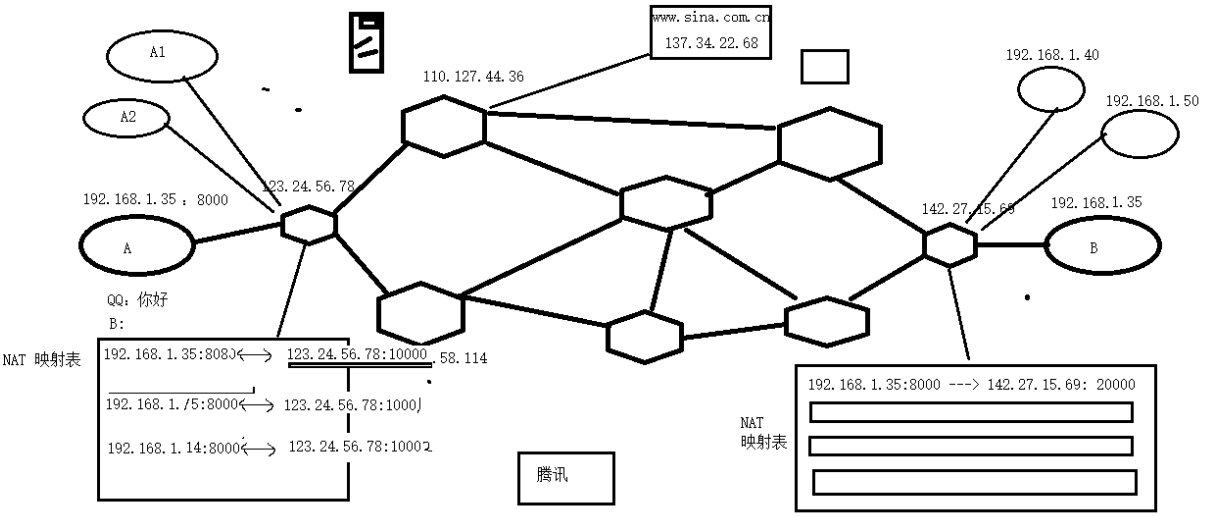
arp数据报：获取下一跳的mac地址。

	目的mac	源mac	8	发送端mac	发送端ip	接收端mac	接收端的ip
arp1	00:00:00:00:00:00	00:0c:29:0a:c4:f4	0806	。。。。。	00:0c:29:0a:c4:f4	192.168.1.20	00:00:00:00:00:00	192.168.1.35
arp2	目的mac	源mac	8	发送端mac	发送端ip	接收端mac	接收端的ip
	00:0c:29:0a:c4:f4	00:0c:29:0a:c4:ba	0806	。。。。。	00:0c:29:0a:c4:ba	192.168.1.35	00:0c:29:0a:c4:f4	192.168.1.20

数据包寻路再分析

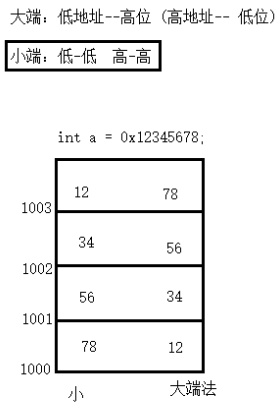


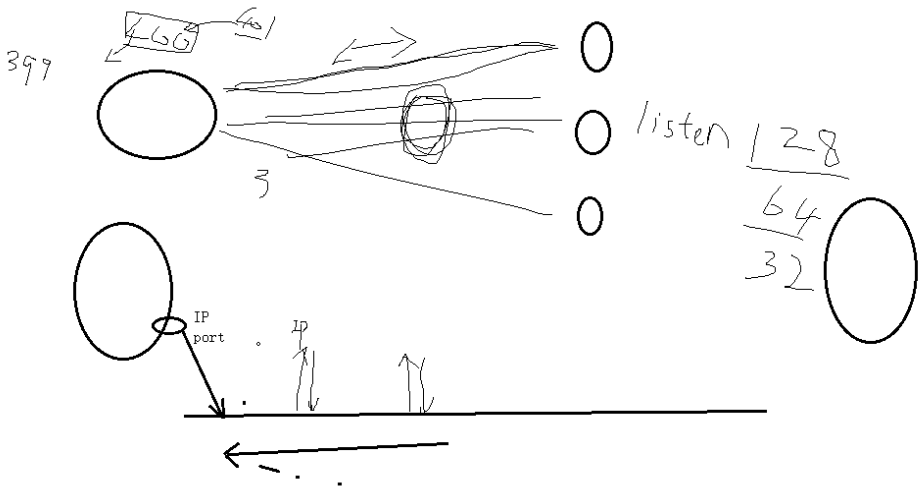
NAT映射



socket编程

网络字节序





为使网络程序具有可移植性，使同样的C代码在大端和小端计算机上编译后都能正常运行，可以调用以下库函数做网络字节序和主机字节序的转换。

```
#include <arpa/inet.h>
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

h表示host，n表示network，l表示32位长整数，s表示16位短整数。

如果主机是小端字节序，这些函数将参数做相应的大小端转换然后返回，如果主机是大端字节序，这些函数不做转换，将参数原封不动地返回。

ip地址转换函数

早期：

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

#include <netinet/in.h>

#include <arpa/inet.h>

int inet_aton(const char *cp, struct in_addr *inp);

in_addr_t inet_addr(const char *cp);

char *inet_ntoa(struct in_addr in);
```

只能处理IPv4的ip地址

不可重入函数

注意参数是struct in_addr

现在：

```
#include <arpa/inet.h>
    int inet_pton(int af, const char *src, void*dst);
    const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
```

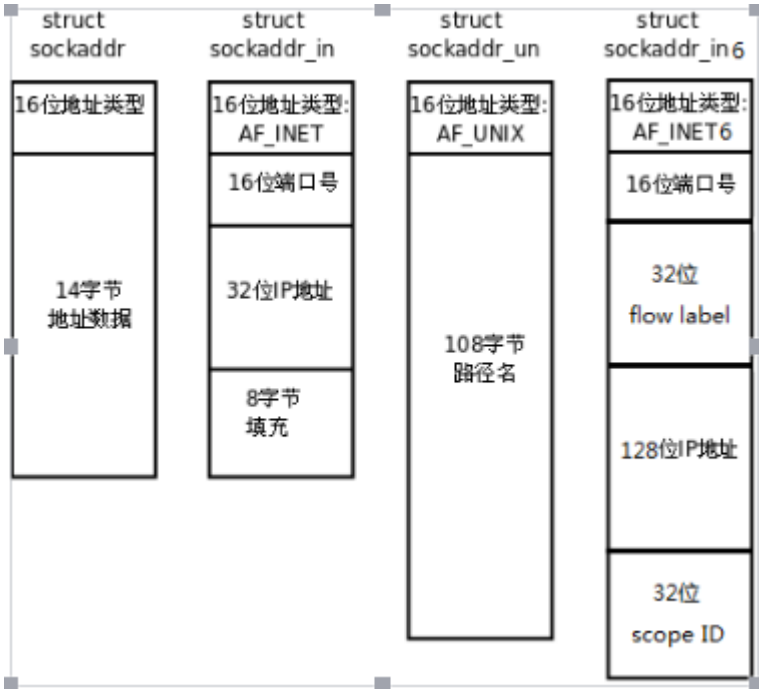
支持IPv4和IPv6

可重入函数

其中inet_pton和inet_ntop不仅可以转换IPv4的in_addr，还可以转换IPv6的in6_addr。

因此函数接口是void *addrptr。

sockaddr数据结构




```
struct sockaddr {

    sa_family_t  sa_family;      /* address family, AF_xxx */

    char  sa_data[14];           /* 14 bytes of protocol address */

};
```

使用 sudo grep -r "struct sockaddr_in {" /usr 命令可查看到struct sockaddr_in结构体的定义。一般其默认的存储位置：/usr/include/linux/in.h 文件中。 也可以使用man 7 ip来查看

```
struct sockaddr_in {

    __kernel_sa_family_t  sin_family;            /* Address family */     地址结构类型

    __be16 sin_port;                             /* Port number */        端口号

    struct in_addr sin_addr;                     /* Internet address */   IP地址

    /* Pad to size of `struct sockaddr'. */

    unsigned char pad[SOCK_SIZE__ - sizeof(short int) -

    sizeof(unsigned short int) - sizeof(struct in_addr)];

};



struct in_addr {                         /* Internet address. */
```

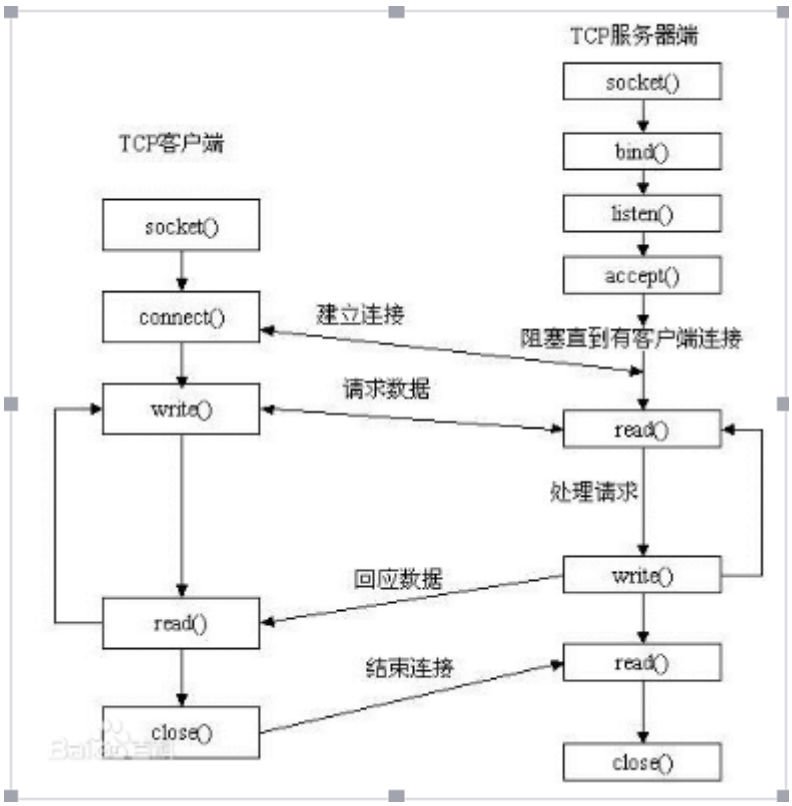
```
    __be32 s_addr;
};

struct sockaddr_in6 {
    unsigned short int sin6_family;         /* AF_INET6 */
    __be16 sin6_port;                   /* Transport layer port # */
    __be32 sin6_flowinfo;               /* IPv6 flow information */
    struct in6_addr sin6_addr;          /* IPv6 address */
    __u32 sin6_scope_id;                /* scope id (new in RFC2553) */
};

struct in6_addr {
    union {
        __u8 u6_addr8[16];
        __be16 u6_addr16[8];
        __be32 u6_addr32[4];
    } in6_u;
    #define s6_addr          in6_u.u6_addr8
    #define s6_addr16    in6_u.u6_addr16
    #define s6_addr32        in6_u.u6_addr32
};

#define UNIX_PATH_MAX 108
    struct sockaddr_un {
    __kernel_sa_family_t  sun_family;    /* AF_UNIX */
    char  sun_path[UNIX_PATH_MAX];   /* pathname */
```

```
};
```

Pv4和IPv6的地址格式定义在netinet/in.h中，IPv4地址用sockaddr_in结构体表示，包括16位端口号和32位IP地址，IPv6地址用sockaddr_in6结构体表示，包括16位端口号、128位IP地址和一些控制字段。UNIX Domain Socket的地址格式定义在sys/un.h中，用sock-addr_un结构体表示。各种socket地址结构体的开头都是相同的，前16位表示整个结构体的长度（并不是所有UNIX的实现都有长度字段，如Linux就没有），后16位表示地址类型。IPv4、IPv6和Unix Domain Socket的地址类型分别定义为常数AF_INET、AF_INET6、AF_UNIX。这样，只要取得某种sockaddr结构体的首地址，不需要知道具体是哪种类型的sockaddr结构体，就可以根据地址类型字段确定结构体中的内容。因此，socket API可以接受各种类型的sockaddr结构体指针做参数，例如bind、accept、connect等函数，这些函数的参数应该设计成void *类型以便接受各种类型的指针，但是sock API的实现早于ANSI C标准化，那时还没有void *类型，因此这些函数的参数都用struct sockaddr *类型表示，在传递参数之前要强制类型转换一下，例如：

```
struct sockaddr_in servaddr;

bind(listen_fd, (struct sockaddr *)&servaddr, sizeof(servaddr));     / initialize
servaddr */
```

socket模型创建流程图



socket编程函数

#include /* See NOTES */

#include

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

domain:

AF_INET 这是大多数用来产生socket的协议，使用TCP或UDP来传输，用IPv4的地址

AF_INET6 与上面类似，不过是来用IPv6的地址

AF_UNIX 本地协议，使用在Unix和Linux系统上，一般都是当客户端和服务器在同一台及其上的时候使用

type:

SOCK_STREAM 这个协议是按照顺序的、可靠的、数据完整的基于字节流的连接。这是一个使用最多的socket类型，这个socket是使用TCP来进行传输。

SOCK_DGRAM 这个协议是无连接的、固定长度的传输调用。该协议是不可靠的，使用UDP来进行它的连接。

SOCK_SEQPACKET该协议是双线路的、可靠的连接，发送固定长度的数据包进行传输。必须把这个包完整的接受才能进行读取。

SOCK_RAW socket类型提供单一的网络访问，这个socket类型使用ICMP公共协议。（ping、traceroute使用该协议）

SOCK_RDM 这个类型是很少使用的，在大部分的操作系统上没有实现，它是提供给数据链路层使用，不保证数据包的顺序

protocol:

传0 表示使用默认协议。

返回值：

成功：返回指向新创建的socket的文件描述符，失败：返回-1，设置errno

socket()打开一个网络通讯端口，如果成功的话，就像open()一样返回一个文件描述符，应用程序可以像读写文件一样用read/write在网络上收发数据，如果socket()调用出错则返回-1。对于IPv4，domain参数指定为AF_INET。对于TCP协议，type参数指定为SOCK_STREAM，表示面向流的传输协议。如果是UDP协议，则type参数指定为SOCK_DGRAM，表示面向数据报的传输协议。protocol参数的介绍从略，指定为0即可。

bind()函数

bind函数****

#include /* See NOTES */

#include

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

sockfd：

socket文件描述符

addr:

构造出IP地址加端口号

addrlen:

sizeof(addr)长度

返回值：

成功返回0，失败返回-1, 设置errno

服务器程序所监听的网络地址和端口号通常是固定不变的，客户端程序得知服务器程序的地址和端口号后就可以向服务器发起连接，因此服务器需要调用bind绑定一个固定的网络地址和端口号。

bind()的作用是将参数sockfd和addr绑定在一起，使sockfd这个用于网络通讯的文件描述符监听addr所描述的地址和端口号。前面讲过，struct sockaddr *是一个通用指针类型，addr参数实际上可以接受多种协议的sockaddr结构体，而它们的长度各不相同，所以需要第三个参数addrlen指定结构体的长度。如：

struct sockaddr_in servaddr;

bzero(&servaddr, sizeof(servaddr));

servaddr.sin_family = AF_INET;

servaddr.sin_addr.s_addr = htonl(INADDR_ANY);

servaddr.sin_port = htons(6666);

首先将整个结构体清零，然后设置地址类型为AF_INET，网络地址为**INADDR_ANY**，这个宏表示本地的任意**IP**地址，因为服务器可能有多个网卡，每个网卡也可能绑定多个IP地址，这样设置可以在所有的IP地址上监听，直到与某个客户端建立了连接时才确定下来到底用哪个IP地址，端口号为6666。

listen()函数

listen函数****

#include /* See NOTES */

#include

int listen(int sockfd, int backlog);

sockfd:

socket文件描述符

backlog:

排队建立3次握手队列和刚刚建立3次握手队列的链接数和

查看系统默认backlog

cat /proc/sys/net/ipv4/tcp_max_syn_backlog

典型的服务器程序可以同时服务于多个客户端，当有客户端发起连接时，服务器调用的accept()返回并接受这个连接，如果有大量的客户端发起连接而服务器来不及处理，尚未accept的客户端就处于连接等待状态，listen()声明sockfd处于监听状态，并且最多允许有backlog个客户端处于连接待状态，如果接收到更多的连接请求就忽略。listen()成功返回0，失败返回-1。

accept函数

accept函数****

#include /* See NOTES */

#include

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

sockdf:

socket文件描述符

addr:

传出参数，返回链接客户端地址信息，含IP地址和端口号

addrlen:

传入传出参数（值-结果）,传入sizeof(addr)大小，函数返回时返回真正接收到地址结构体的大小

返回值：

成功返回一个新的socket文件描述符，用于和客户端通信，失败返回-1，设置errno

三方握手完成后，服务器调用accept()接受连接，如果服务器调用accept()时还没有客户端的连接请求，就阻塞等待直到有客户端连接上来。addr是一个传出参数，accept()返回时传出客户端的地址和端口号。addrlen参数是一个传入传出参数（value-result argument），传入的是调用者提供的缓冲区addr的长度以避免缓冲区溢出问题，传出的是客户端地址结构体的实际长度（有可能没有占满调用者提供的缓冲区）。如果给addr参数传NULL，表示不关心客户端的地址。

我们的服务器程序结构是这样的：

while (1) {

cliaddr_len = sizeof(cliaddr);

connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len);

n = read(connfd, buf, MAXLINE);

.....

close(connfd);

}

整个是一个while死循环，每次循环处理一个客户端连接。由于cliaddr_len是传入传出参数，每次调用accept()之前应该重新赋初值。accept()的参数listenfd是先前的监听文件描述符，而accept()的返回值是另外一个文件描述符connfd，之后与客户端之间就通过这个connfd通讯，最后关闭connfd断开连接，而不关闭listenfd，再次回到循环开头listenfd仍然用作accept的参数。accept()成功返回一个文件描述符，出错返回-1。

connect函数

connect函数****

#include /* See NOTES */

#include

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

sockdf:

socket文件描述符

addr:

传入参数，指定服务器端地址信息，含IP地址和端口号

addrlen:

传入参数,传入sizeof(addr)大小

返回值：

成功返回0，失败返回-1，设置errno

客户端需要调用connect()连接服务器，connect和bind的参数形式一致，区别在于bind的参数是自己的地址，而connect的参数是对方的地址。connect()成功返回0，出错返回-1。

简单的服务器和客户端的通信例子

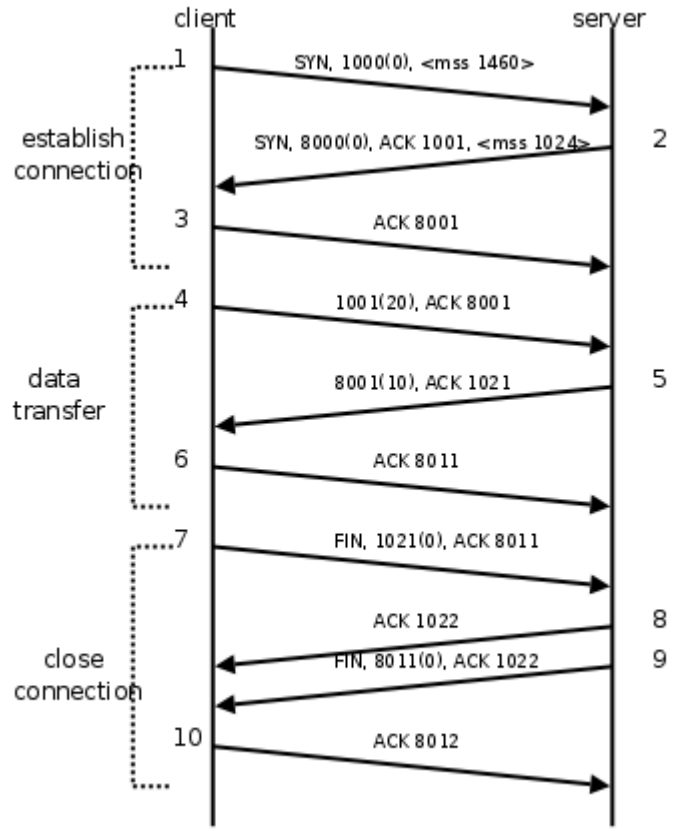
```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <strings.h>
#include <string.h>
#include <ctype.h>
#include <arpa/inet.h>

#define SERV_PORT 9527

int main(void)
{
    int sfd, cfd;
    int len, i;
    char buf[BUFSIZ], clie_IP[BUFSIZ];

    struct sockaddr_in serv_addr, clie_addr;
    socklen_t clie_addr_len;

    /*创建一个socket 指定IPv4协议族 TCP协议*/
    sfd = socket(AF_INET, SOCK_STREAM, 0);

    /*初始化一个地址结构 man 7 ip 查看对应信息*/
    bzero(&serv_addr, sizeof(serv_addr));           //将整个结构体清零
    serv_addr.sin_family = AF_INET;                 //选择协议族为IPv4

    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);  //监听本地所有IP地址
```

```
    serv_addr.sin_port = htons(SERV_PORT);          //绑定端口号

    /*绑定服务器地址结构*/
    bind(sfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

    /*设定链接上限,注意此处不阻塞*/
    listen(sfd, 64);                                //同一时刻允许向服务器发起链接请求的数量

    printf("wait for client connect ...\n");

    /*获取客户端地址结构大小*/
    clie_addr_len = sizeof(clie_addr_len);
    /*参数1是sfd; 参2传出参数, 参3传入传入参数, 全部是client端的参数*/
    cfd = accept(sfd, (struct sockaddr *)&clie_addr, &clie_addr_len);           /*监听
客户端链接, 会阻塞*/

    printf("client IP:%s\tport:%d\n",
           inet_ntop(AF_INET, &clie_addr.sin_addr.s_addr, clie_IP, sizeof(clie_IP)),
           ntohs(clie_addr.sin_port));

    while (1) {
        /*读取客户端发送数据*/
        len = read(cfd, buf, sizeof(buf));
        write(STDOUT_FILENO, buf, len);

        /*处理客户端数据*/
        for (i = 0; i < len; i++)
            buf[i] = toupper(buf[i]);

        /*处理完数据回写给客户端*/
        write(cfd, buf, len);
    }

    /*关闭链接*/
    close(sfd);
    close(cfd);

    return 0;
}
```

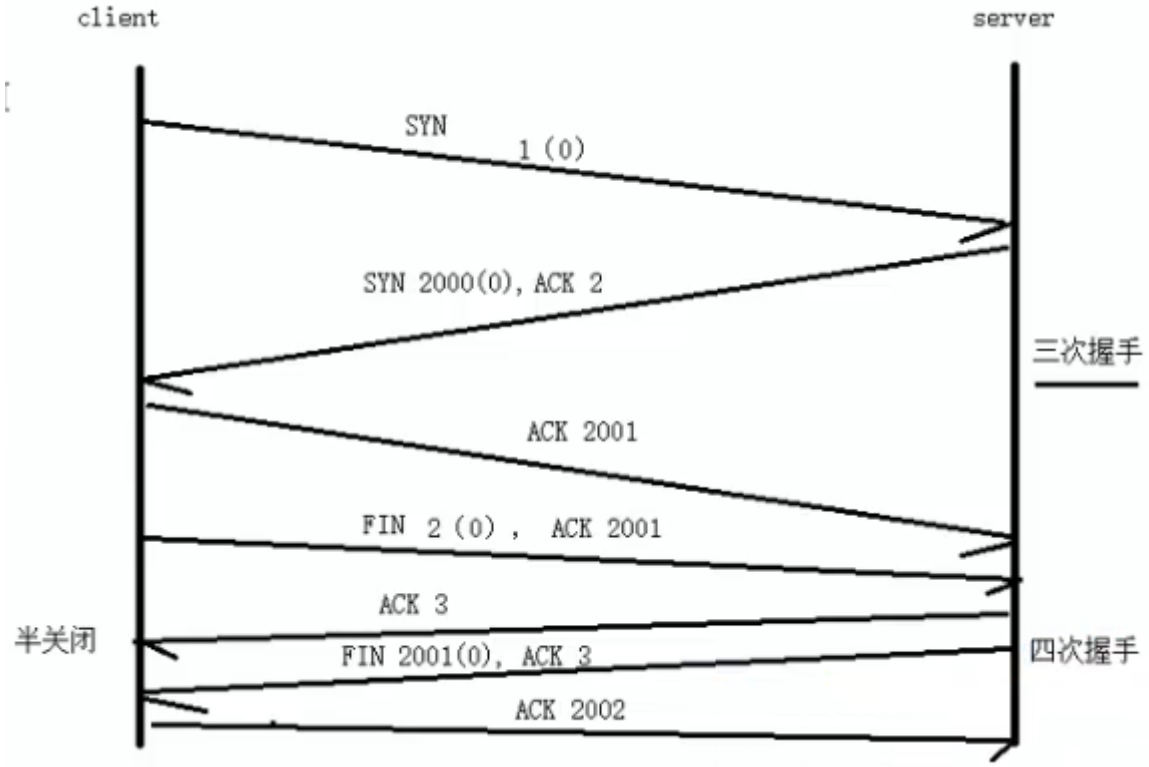
```
//client.c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>

#define SERV_IP "127.0.0.1"
#define SERV_PORT 9527

int main(void)
{
    int sfd, len;
```

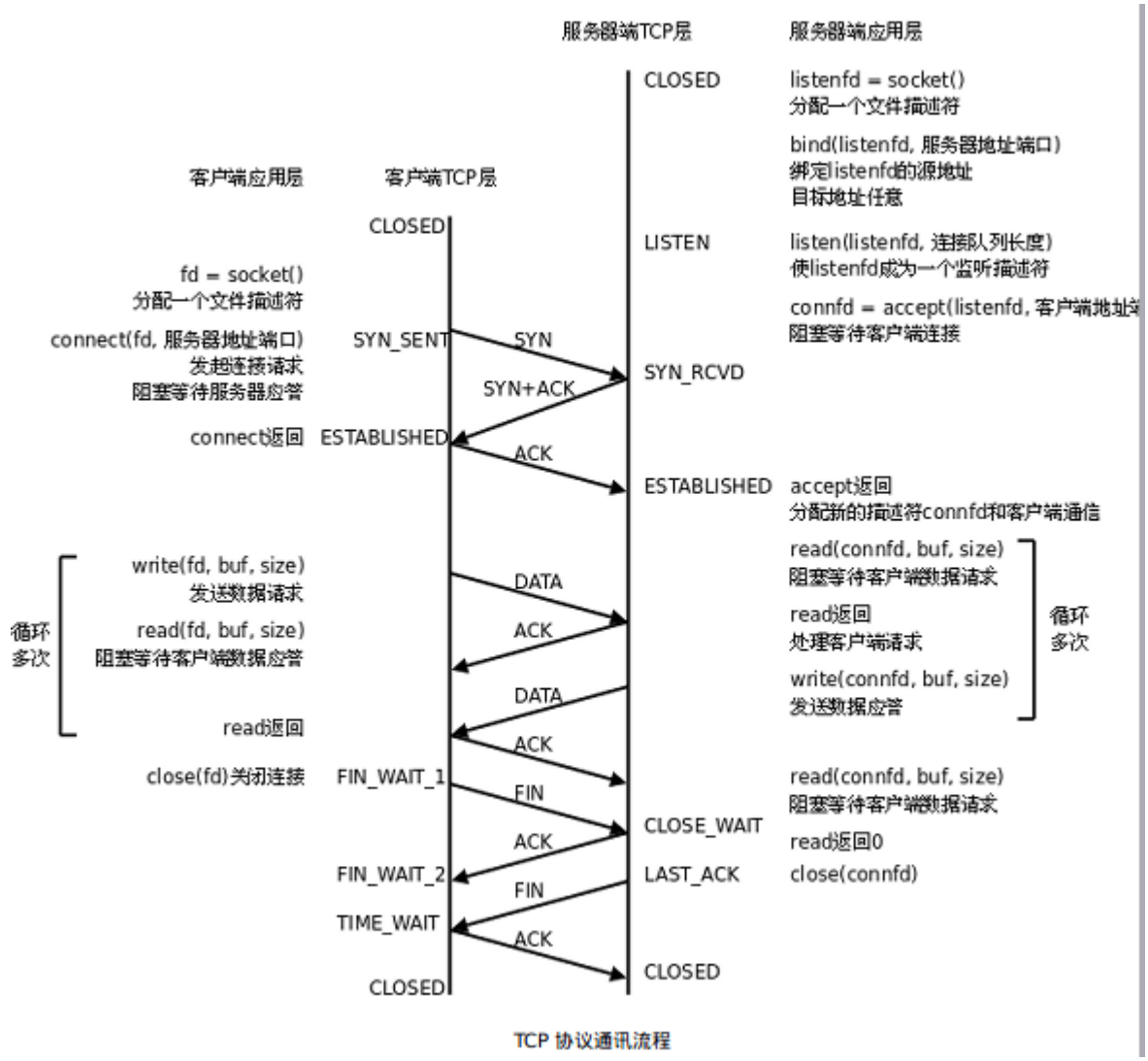
```
    struct sockaddr_in serv_addr;
    char buf[BUFSIZ];

    /*创建一个socket 指定IPv4 TCP*/
    sfd = socket(AF_INET, SOCK_STREAM, 0);

    /*初始化一个地址结构:*/
    bzero(&serv_addr, sizeof(serv_addr));                       //清零
    serv_addr.sin_family = AF_INET;                             //IPv4协议族
    inet_pton(AF_INET, SERV_IP, &serv_addr.sin_addr.s_addr);    //指定IP 字符串类型转换为网络字节序 参3:传出参数
    serv_addr.sin_port = htons(SERV_PORT);                      //指定端口 本地转网络字节序

    /*根据地址结构链接指定服务器进程*/
    connect(sfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

    while (1) {
        /*从标准输入获取数据*/
        fgets(buf, sizeof(buf), stdin);
        /*将数据写给服务器*/
        write(sfd, buf, strlen(buf));       //写个服务器
        /*从服务器读回转换后数据*/
        len = read(sfd, buf, sizeof(buf));
        /*写至标准输出*/
        write(STDOUT_FILENO, buf, len);
    }

    /*关闭链接*/
    close(sfd);

    return 0;
}
```

三次握手和四次挥手


client
server
1
SYN, 1000(0), <mss 1460>
establish
connection
SYN, 8000(0), ACK 1001, <mss 1024>
2
3
ACK 8001
4
1001(20), ACK 8001
data
transfer
8001(10), ACK 1021
5
6
ACK 8011
7
FIN, 1021(0), ACK 8011
ACK 1022
8
close
connection
FIN, 8011(0), ACK 1022
9
10
ACK 8012

TCP 通讯时序


client
server
SYN
1 (0)
SYN 2000(0), ACK 2
三次握手
ACK 2001
FIN 2 (0) , ACK 2001
ACK 3
半关闭
FIN 2001(0), ACK 3
四次握手
ACK 2002



TCP 协议通讯流程

tcp的不是不会丢包,而是丢包了之后,那是会重传的.