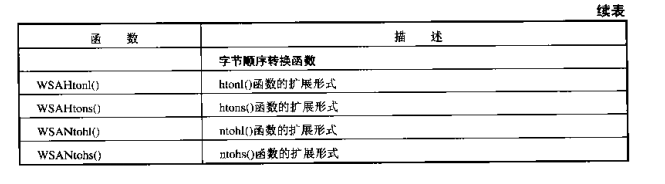
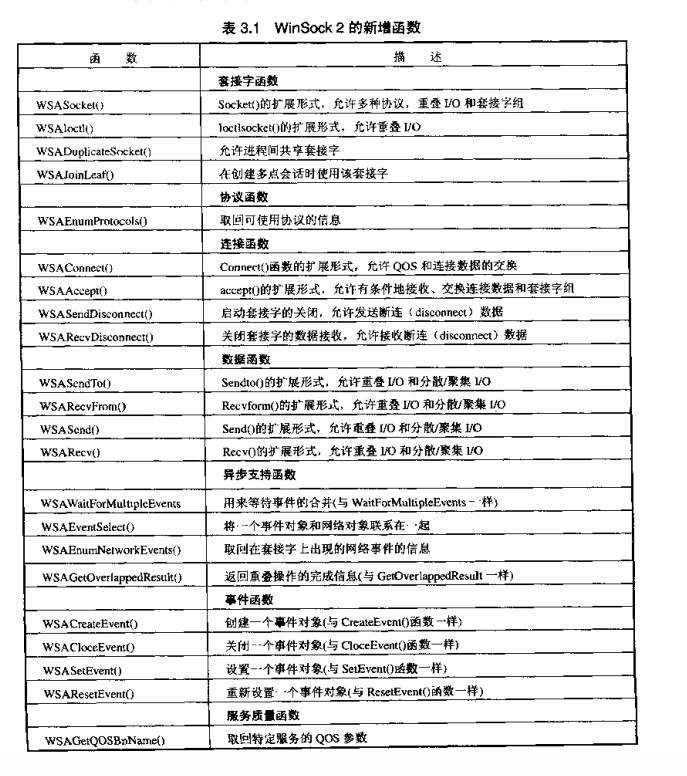
HP-Socket 是一套通用的高性能 TCP/UDP 通信框架（应用程序不必关注通信层的任何细节），包含服务端组件、客户端组件和Agent组件，广泛适用于各种不同应用场景的 TCP/UDP 通信系统。HP-Socket 对通信层实现完全封装，应用程序不必关注通信层的任何细节。支持Agent组件（Agent组件实质上是Multi-Client组件，与Server组件采用相同的技术架构。一个Agent组件对象可同时建立和高效处理大规模Socket连接，类似于网关的汇总）！网络传输过程中，数据包超过某长度会MTU分片，那么服务器接收到的一个包，可能是一个完整包、半个包或者一个半包，还好贴心HP-Socket作者已经帮我们考虑好，有3种接收模型可选：PACK、PULL和PUSH模型（全自动、半自动和手工解决粘包问题）。

-1.感觉win socket1编程里面的函数名字都是小写，类型就大写了

Winsock在常见的Windows平台上有两个主要的版本，即Winsock1和Winsock2。编写与Winsock1兼容的程序你需要引用头文件WINSOCK.H，如果编写使用Winsock2的程序，则需要引用WINSOCK2.H。此外还有一个MSWSOCK.H头文件，它是专门用来支持在Windows平台上高性能网络程序扩展功能的。使用WINSOCK.H头文件时，同时需要库文件WSOCK32.LIB，使用WINSOCK2.H时，则需要WS2\_32.LIB，如果使用MSWSOCK.H中的扩展API，则需要MSWSOCK.LIB。正确引用了头文件，并链接了对应的库文件，你就构建起编写WINSOCK网络程序的环境了。

socket没有bind()就直接使用的话，那么系统会替你把创建的sock对象

套接字I/O模型共有五种类型，如下：

  select、poll、epoll（选择）   
  WSAAsyncSelect（异步选择） 设置套接字属性为非阻塞，并使用windows消息机制实现信息传递  
  WSAEventSelect（事件选择）  
  overlapped（重叠）  
  completion port（完成端口）

0.socket其实是一个序号，表示其在句柄表中的位置。句柄，其实是代表了系统中的某些特定的对象，用于在各种函数中作为参数传入，以对特定对象进行操作。

socket 结构体的定义如下： 所以tcp socket可以认为是源地址、源端口、目的地址、目的端口、协议号来确定唯一性，只要其中任何一个改变，同一个进程同一个端口号还是可以建立连接的。一个套接字不能连接两次，并不代表一个本地地址不能用两次，描述套接字是5元信息。udp socket是希望去请求的ip+port 因为udp的无连接特性。

struct socket

{

socket\_state state;

unsigned long flags;

const struct proto\_ops \*ops;

struct fasync\_struct \*fasync\_list;

struct file \*file; //这里有一个指向文件的指针

struct sock \*sk;

wait\_queue\_head\_t wait;

short type;

};

其中，struct sock 包含有一个 sock\_common 结构体，而sock\_common结构体又包含有struct inet\_sock 结构体，而struct inet\_sock 结构体的部分定义如下：

struct inet\_sock

{

struct sock sk;

#if defined(CONFIG\_IPV6) || defined(CONFIG\_IPV6\_MODULE)

struct ipv6\_pinfo \*pinet6;

#endif

\_\_u32 daddr; //IPv4的目的地址。

\_\_u32 rcv\_saddr; //IPv4的本地接收地址。

\_\_u16 dport; //目的端口。

\_\_u16 num; //本地端口（主机字节序）。

…………

}

由此，我们清楚了，C语言socket结构体不仅仅记录了本地的IP和端口号，还记录了目的IP和端口。所以同一个进程关系树里面可以有多个使用本机同一个端口的socket，或者多个设置了允许端口复用的进程REUSEADDR。多个socket绑定同一个端口，叫做端口复用。

1.WSAStartup的WSA是Windows Sockets Asynchronous

2.UDP的英文是User Datagram Protocol 用户包协议

3.TCP和UDP的端口各有65536个。

5.先初始化winsock2库：int WSAStartup(WORD wVersionRequested, LPWSADATA lpWSAData);

TCP函数调用过程：

服务器端：socket()->bind()->listen()->accept()->send()/recv()->closesocket()//话说为什么不把listen和accept合在一起。回应客户端的建立连接报文是TCP协议栈负责的，而应用程序的accept只是从系统协议栈已连接的socket队列中拿走一个使用，listen之后没有accept，客户端的connect也能成功。

客户端：socket()->connect()->send()/recv()->closesocket()

UDP函数调用过程：

服务器端：socket()->bind()->sendto()/recvfrom()->closesocket()

客户端：socket()->sendto()/recvfrom()->closesocket()

再释放winsock库：int WSACleanup(void);

UDT：UDP版TCP

UDP也提供了connec方法

1. 将对端信息存入udp socket，sendto/recvfrom改用send/recv

2. 限定了这个socket的使用范围： 只允许从这个指定的SocketAddress 上获得数据包和向这个指定的SocketAddress 发送数据包， 当你一旦通过这个socket向别的地址发送了数据包，或者接收到了不是这个地址发送来的数据包，那么程序就会抛出IllegalArgumentException 异常， 特殊的是如果指定的这个SocketAddress 是个多播地址或者广播地址，那么只允许向这个地址发送数据包，不允许从这个地址接收数据包

在winsock2.h中#define PF\_INET AF\_INET。不过这只是windows暂时的情况，为了以后先预留下，让地址族和协议族先一样，所以为了保持良好的兼容性，在使用地址族的时候就不要使用PF\_INET。AF就是Address Family。PF就是Protocol Family。在socket函数中应该使用的是PF\_INET.

AF\_INET、AF\_IPX和AF\_NETBIOS等分别用于IP寻址、IPX寻址和NETBIOS寻址等

不存在cwinsock，使用winsock，windows与linux都不需要链接？windows的连接库不是socket，而就是ws2\_32。

socket(int af, int type, int protocol)创建套接字。af表示地址族(感觉这个参数没必要)，就是PF\_INET，表示不明所以；type三个：SOCK\_STREAM，SOCK\_DGRAM和SOCK\_RAM，分别表示流套接字、数据报套接字和原始协议接口；protocol用来指定应用程序使用的通信协议：IPPROTO\_TCP，IPPROTO\_UDP，IPPROTO\_ICMP等。第三个参数受到第二个参数影响：如果第二个参数是SOCK\_STREAM那就应该IPPROTO\_TCP，是SOCK\_DGRAM那就应该IPPROTO\_UDP，是SOCK\_DGRAM或者SOCK\_STREAM那可以默认为0，是SOCK\_RAM那就不能使用0这个值，可以是IPPROTO、IPPROTO\_IP、IPPROTO\_ICMP、IPPROTO\_TCP、IPPROTO\_RAM等，使用前四种系统就会自动为数据加上IP首部，除非有IP\_HDRINCL选项才不会自动添加，使用IPPROTO\_RAM那就需要自己写上IP首部的代码。不管哪个，接收的时候IP首部都是要自己的程序处理的。如果socket调用失败那么就会返回INVALID\_SOCKET，这时候想要知道失败的原因你就WSAGetLastError()得到错误码，再用VC的一个工具Error Lookup去看看。这个函数应该与closesocket()成对存在。其实我觉的第二个参数可以用第三个参数推出来，不知道第二个参数的存在意义。

int closesocket(SOCKET s);

int bind(SOCKET s, const struct sockaddr FAR \*name, int namelen);第二个参数是一个16字节的结构:

struct sockaddr {

u\_short sa\_family;

char sa\_data[14];

};

struct sockaddr\_in {

//这个结构体也是十六字节，我们先构造好这个，然后再把这个结构体的地址传给\* sockaddr，还有着FAR感觉没有什么用，读的时候就是忽略它

short sin\_family;

u\_short sin\_port;

struct in\_addr sin\_addr;

char sin\_zero[8];//sin\_zero用来将sockaddr\_in结构填充到与struct sockaddr同样的长度，可以用bzero()或memset()函数将其置为零。

};

struct in\_addr {//用于存放不同类型的IPv4地址

union {

struct { u\_char s\_b1, s\_b2, s\_b3, s\_b4; } S\_un\_b;//b是byte的意思

struct { u\_shhort s\_w1, s\_w2;} S\_un\_w;//w是word的意思

u\_long S\_addr;

}S\_un;//un是union的意思

};//如果对服务器的地址指定为INADDR\_ANY宏（0.0.0.0 这个通用地址），这表示任意地址，那么服务器会根据网络配置自动选择一个IP与客户端沟通

设置 SO\_REUSEPORT 可以让占用的port，继续被另外一个tcp/udp socket绑定，注意每一个socket都要设置。对于tcp，每一个进程有一个独立的监听socket，并且bind相同的ip:port，独立的listen()和accept()提高接收连接的能力。各个进程中accept socket fd不一样，有新连接建立时，内核只会唤醒一个进程来accept，并且保证唤醒的均衡性，避免了应用层多线程或者进程监听同一ip:port的“惊群效应”。例如nginx多进程同时监听同一个ip:port。

SO\_REUSEADDR 重用地址

unsigned long inet\_addr(const char FAR\* cp);将标准的Ipv4转化成为一个unsigned long格式，然后就赋值给sockaddr\_in对象中的sin\_addr.S\_un.S\_addr。他还说sockaddr\_in里面没有提供用来存储点分十进制的IPv4。将IP地址转为网上传播的套接字内容。

char FAR\* inet\_ntoa(struct in\_addr in);首先不知道为什么参数没有const。这个函数名就是net to address，把套机字里面的IP转化为本机可以使用的点分十进制IPv4。

bing函数使用举例：

//创建套接字

SOCKET sLisent =socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

//填充sockadd\_in对象

struct sockaddr\_in SeverAddr；

SeverAddr.sin\_family =AF\_INET;

SeverAddr.sin\_port =htons(4321);

SeverAddr.sin\_addr.S\_un.S\_addr = inet\_addr("114.114.114.114");

//绑定套接字与地址信息

bind(sLient, (SOCKADDR \*)&ServerAddr, sizeof(ServerAddr));

int listen(SOCKET s, int limit);s就是要监听的套接字描述符，limit就是允许进入请求连接队列的个数。在winsock2.h中，其最大值由SOMAXCONN表示, #define SOMAXCONN 0×7fffffff。第二个参数可以是SOMAXCONN。设置并且监听。

SOCKET accept(SOCKET s, struct sockaddr FAR \*addr, int FAR\* addrlen);本函数从s的等待连接队列中抽取第一个连接，创建一个与s同类的新的套接口并返回句柄。如果队列中无等待连接，且套接口为阻塞方式，则accept()阻塞调用进程直至新的连接出现。如果套接口为非阻塞方式且队列中无等待连接，则accept()返回一错误代码。（百度百科说第二个参数与第三个参数是可选的，这两个参数就是为了表示的连接来的远程套接字内的信息）。返回值估计是自己这边新建的同类套接字。这个函数不可以省略，估计就是它参与完成了tcp的“三次”连接的第二次应答。

注意accept第三个参数，不同于bind这个函数。和UDP的recvfrom一样，最后一个参数同样的作用，都是指针。

int connect(SOCKET s, struct sockaddr FAR\* addr, int addrlen);奇怪这里的函数怎么都用int当返回值而不用Bool，难道是返回值有多种失败可能。这个函数应该就是用于TCP三次握手。默认设置的超时返回时间有点长。

connect和accept都是对套接字里面结构体的继续设置

客户端连接成功后,如果服务器sleep(),客户端发送send(),是否可以成功？可以，因为机器会用缓冲区接受的，只是程序拒绝取。如果sleep久了，缓冲区被填满，就会导致另外一段发送失败或者陷入阻塞。

int send(SOCKET s, const char FAR\* buf, int len, int flags);第四个参数默认为0

recv起到的是数据的转移，从缓冲区复制出去（所以返回值也就是复制的字节数），并不是说把数据接收到缓冲区当中。

不论是客户还是服务器应用程序都用recv函数从TCP连接的另一端接收数据。该函数的第一个参数指定接收端套接字描述符； 第二个参数指明一个缓冲区，该缓冲区用来存放recv函数接收到的数据； 第三个参数指明buf的长度； 第四个参数一般置0。

第四个参数就是TCP的flags。

每一个除send外的Socket函数在执 行的最开始总要先等待套接字的发送缓冲中的数据被协议传送完毕才能继续，如果在等待时出现网络错误，那么该Socket函数就返回 SOCKET\_ERROR，因为send函数的具体作用是将数据从用户缓冲区发送到协议的缓冲区，然后如果没有设置为阻塞就会返回，否则的话估计还需要等待发送完毕从而获取到底发送了多少个字节作为返回值。这么做的原因估计就是因为不能够直接控制实现真正发送到链路上，所以通过这种等待的做法来确保刚刚send实现，不然系统发送的过程中改变了socket的配制就难受了，而刚刚的send就应该使用刚刚的配置。

这里只描述同步Socket的recv函数的执行流程。当应用程序调用recv函数时，

（1）recv先等待s的发送缓冲中的数据被协议传送完毕，如果协议在传送s的发送缓冲中的数据时出现网络错误，那么recv函数返回SOCKET\_ERROR。

（2）如果s的发送缓冲中没有数据或者数据被协议成功发送完毕后，recv先检查套接字s的接收缓冲区，如果s接收缓冲区中有数据或者协议正在接收数 据，那么recv就一直等待，直到协议把数据接收完毕。当协议把数据接收完毕，recv函数就把s的接收缓冲中的数据copy到buf中（注意协议接收到的数据可能大于buf的长度，所以 在这种情况下要调用几次recv函数才能把s的接收缓冲中的数据copy完。recv函数仅仅是copy数据，真正的接收数据是协议来完成的）， recv函数返回其实际copy的字节数。如果recv在copy时出错，那么它返回SOCKET\_ERROR；如果recv函数在等待协议接收数据时网络中断了，那么它返回0。 默认 socket 是阻塞的 解阻塞与非阻塞recv返回值没有区分，都是 <0 出错 =0 连接关闭 >0 接收到数据大小，

SOCKET\_ERROR的数值就是-1。

在Intel X86的架构下，数值存储方式默认是大端方式（又称大尾方式字节序，内存高位放数据低位字节，相当于是现代人读取的顺序），但是TCP/UDP都是小端方式也就是（小尾字节序，0x01020304也就是以04 03 02 01存起来）。Unix是多以小端存储。在大端存储的地方我们要用转换函数转一下：

u\_short htons(u\_short hostshort);

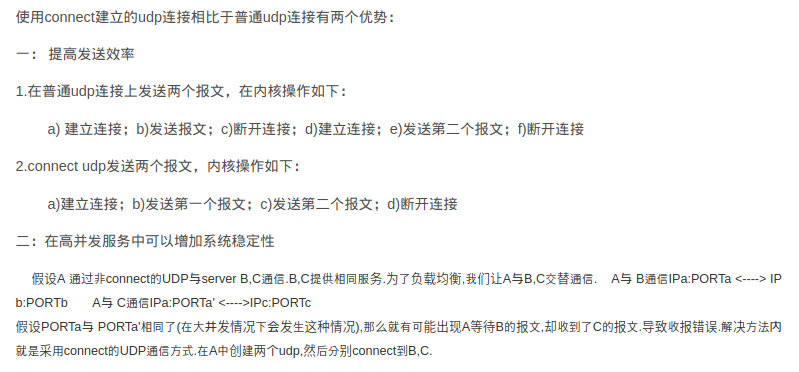
u\_long htonl(u\_long hostshort);//估计u\_long等价与unsigned long

其逆函数就是：

ntohs和ntohl //n表示network h表示host 最后一个字母表示转换的级别

端口和IP都是按照大端方式存储，但是IP的转换函数不好用。

udp没有调用connect那只能调用sendto recvfrom 不可以调用send recv，允许许多次调用connect函数来改变发送目标（支持随时断开连接，tcp不支持）。



对于UDP连接：

允

发送和接收函数多了两个末尾的参数：

int sendto(SOCKET s. const char FAR\* buf. int len. int flags, const struct sockaddr FAR \*to, int tolen);第五个参数给出接收消息的地址信息。第6个参数就是表示大小的。没有connect()那就只好在发送的时候设置目标。

int recvfrom(SOCKET s, const char FAR\* buf, int len, int flags, const struct sockaddr FAR \*from, int FAR\* fromlen)注意第六个参数是指针类型，我也不知道为什么。第五个参数存储来源位置,起的就是accept的作用，所以最后一个参数是指针形式。

对于非阻塞模式:有多种将套接字进行设置使之变成非阻塞模式。书上提到的是WSAAsyncSelect()函数：

int WSAAsyncSelect() {

SOCKET s,//准备改为非阻塞的套接字

hwnd hWnd,//接收消息的窗口

unsigned int wMsg;//自定义消息，是在WM\_USER的基础上加一个数值

long lEvent;//指定应用程序感兴趣的通知码

}//这个函数的使用一般就是刚好在使用socket函数初始化SOCKET对象后面，将套接字和窗口绑定，相当于靠窗口的消息机制结合来实现非阻塞，然后在mfc中增加消息的映射关系（在类定义中增加消息调用函数afx\_msg void OnSock(WPARAM wParam， LPARAM lParam)；在类实现增加void OnSock(WPARAM wParam， LPARAM lParam)的函数体；添加消息映射ON\_MESSAGE(UM\_SERVER， OnSock））在OnSock函数里面在用switch对通知码进行不同的处理

#define UM\_SERVER (WM\_USER + 200)这就是个自定义消息，WM\_USER就是数值最大的预定义消息了

lEvent:在不同的情况下，会有部分通知码是无意义的

FD\_READ 套接字收到对端发来的数据包

FD\_ACCEPT 监听中的套接字有连接请求

FD\_CONNECT 套接字成功连接到对方

FD\_CLOSE 套接字的连接被关闭

udp因为无连接可以简单快速地写出并发，但终究是单端口单buffer处理，因此对于大并发也可以epoll。写法与tcp基本一致，将listener加入epoll ，通过connect创建专用udp socket。

#include <stdio.h>  
#include <stdlib.h>  
#include <errno.h>  
#include <string.h>  
#include <sys/types.h>  
#include <netinet/in.h>  
#include <sys/socket.h>  
#include <sys/wait.h>  
#include <unistd.h>  
#include <arpa/inet.h>  
#include <fcntl.h>  
#include <sys/epoll.h>  
#include <sys/time.h>  
#include <sys/resource.h>  
#include <pthread.h>  
#include <assert.h>  
  
#define SO\_REUSEPORT 15  
  
#define MAXBUF 10240  
#define MAXEPOLLSIZE 100  
  
int flag = 0;  
  
int read\_data(int sd)  
{  
 char recvbuf[MAXBUF + 1];  
 int ret;  
 struct sockaddr\_in client\_addr;  
 socklen\_t cli\_len=sizeof(client\_addr);  
  
 bzero(recvbuf, MAXBUF + 1);  
   
 ret = recvfrom(sd, recvbuf, MAXBUF, 0, (struct sockaddr \*)&client\_addr, &cli\_len);  
 if (ret > 0) {  
 printf("read[%d]: %s from %d\n", ret, recvbuf, sd);  
 } else {  
 printf("read err:%s %d\n", strerror(errno), ret);  
   
 }  
 fflush(stdout);  
}  
  
int udp\_accept(int sd, struct sockaddr\_in my\_addr)  
{  
 int new\_sd = -1;  
 int ret = 0;  
 int reuse = 1;  
 char buf[16];  
 struct sockaddr\_in peer\_addr;  
 socklen\_t cli\_len = sizeof(peer\_addr);  
  
 ret = recvfrom(sd, buf, 16, 0, (struct sockaddr \*)&peer\_addr, &cli\_len);  
 if (ret > 0) {  
 }  
  
 if ((new\_sd = socket(PF\_INET, SOCK\_DGRAM, 0)) == -1) {  
 perror("child socket");  
 exit(1);  
 } else {  
 printf("parent:%d new:%d\n", sd, new\_sd);  
 }  
  
 ret = setsockopt(new\_sd, SOL\_SOCKET, SO\_REUSEADDR, &reuse,sizeof(reuse));  
 if (ret) {  
 exit(1);  
 }  
  
 ret = setsockopt(new\_sd, SOL\_SOCKET, SO\_REUSEPORT, &reuse, sizeof(reuse));  
 if (ret) {  
 exit(1);  
 }  
  
 ret = bind(new\_sd, (struct sockaddr \*) &my\_addr, sizeof(struct sockaddr));  
 if (ret){  
 perror("chid bind");  
 exit(1);  
 } else {  
 }  
  
 peer\_addr.sin\_family = PF\_INET;  
 printf("aaa:%s\n", inet\_ntoa(peer\_addr.sin\_addr));  
 if (connect(new\_sd, (struct sockaddr \*) &peer\_addr, sizeof(struct sockaddr)) == -1) {  
 perror("chid connect");  
 exit(1);  
 } else {  
 }  
  
out:  
 return new\_sd;  
}  
  
int main(int argc, char \*\*argv)  
{  
 int listener, kdpfd, nfds, n, curfds;  
 socklen\_t len;  
 struct sockaddr\_in my\_addr, their\_addr;  
 unsigned int port;  
 struct epoll\_event ev;  
 struct epoll\_event events[MAXEPOLLSIZE];  
 int opt = 1;;  
 int ret = 0;  
  
 port = 1234;  
   
 if ((listener = socket(PF\_INET, SOCK\_DGRAM, 0)) == -1) {  
 perror("socket");  
 exit(1);  
 } else {  
 printf("socket OK\n");  
 }  
  
 ret = setsockopt(listener,SOL\_SOCKET,SO\_REUSEADDR,&opt,sizeof(opt));  
 if (ret) {  
 exit(1);  
 }  
  
 ret = setsockopt(listener, SOL\_SOCKET, SO\_REUSEPORT, &opt, sizeof(opt));  
 if (ret) {  
 exit(1);  
 }  
   
 bzero(&my\_addr, sizeof(my\_addr));  
 my\_addr.sin\_family = PF\_INET;  
 my\_addr.sin\_port = htons(port);  
 my\_addr.sin\_addr.s\_addr = INADDR\_ANY;  
 if (bind(listener, (struct sockaddr \*) &my\_addr, sizeof(struct sockaddr)) == -1) {  
 perror("bind");  
 exit(1);  
 } else {  
 printf("IP bind OK\n");  
 }  
   
 kdpfd = epoll\_create(MAXEPOLLSIZE);  
  
 ev.events = EPOLLIN|EPOLLET;  
 ev.data.fd = listener;  
  
 if (epoll\_ctl(kdpfd, EPOLL\_CTL\_ADD, listener, &ev) < 0) {  
 fprintf(stderr, "epoll set insertion error: fd=%dn", listener);  
 return -1;  
 } else {  
 printf("ep add OK\n");  
 }  
   
 while (1) {  
   
 nfds = epoll\_wait(kdpfd, events, 10000, -1);  
 if (nfds == -1) {  
 perror("epoll\_wait");  
 break;  
 }  
   
 for (n = 0; n < nfds; ++n) {  
 if (events[n].data.fd == listener) {  
 printf("listener:%d\n", n);  
 int new\_sd;   
 struct epoll\_event child\_ev;  
  
 new\_sd = udp\_accept(listener, my\_addr);  
 child\_ev.events = EPOLLIN;  
 child\_ev.data.fd = new\_sd;  
 if (epoll\_ctl(kdpfd, EPOLL\_CTL\_ADD, new\_sd, &child\_ev) < 0) {  
 fprintf(stderr, "epoll set insertion error: fd=%dn", new\_sd);  
 return -1;  
 }  
 } else {  
 read\_data(events[n].data.fd);  
 }  
 }  
 }  
 close(listener);  
 return 0;  
}

6.主机的小大端和操作系统的架构有关系。判断方法除了制造连续字节存储然后看输出外，还可用这些转换函数，看转了还一不一样。

7.#pragma comment(lib, "ws2\_32.lib")如果你加了这句，编译当前的单元时，linker会静态链接该preproccessor指定的lib文件，这样等于就告诉编译器有这样一个lib文件可以去链接。那个.lib似乎可以没有。gnu不支持#pragma，但不影响编译，所以有没有还是一样，关键是编译时链接上dll库。

如果你用mfc做开发，你可以点击菜单project-〉setting-〉link-〉object/library中添加这个静态库。或通过 #pragma comment（）；命令来连接静态库。gnu -lws2\_32

ws2\_32.lib是socket lib库。

8.MAKEWORD是将两个byte型合并成一个word型，一个在高8位(b)，一个在低8位(a)

9.[Linker error] undefined reference to `WSAStartup@8'.在工程里已#include <winsock2.h>了结果还是报错（h文件里面只包含了函数声明信息），其实是没有链接上ws2\_32.lib，对于VC6.0要在porject/settings的link页面中objec；ry modules中添ws2\_32.lib就可以了。

10.socket没有bind()就直接使用的话，那么系统会替你把创建的sock对象填充、自动给你分配一个端口号。

11.windows防火墙会拦住一些发来的包，它的开启可能是导致实现失败的原因，如果你指定的本机接收端口比较特殊的话。开启windows 自带 ftp就会受到防火墙的影响。

12.如果连接没有建立，你就send，那么send就会返回错误标志SOCKET\_ERROR。send和recv都是错误返回SOCKET\_ERROR。

14.默认，服务器在accept的时候就会一直阻塞，等待连接。阻塞模式下的recv也是这样子，会等待信息的到来。

15.sokit 是TCP/UDP 数据包收发测试(调试)工具。

16.near指针和far指针？

汇编跳转里面出现过这两个修饰符

在DOS下（实模式）地址是分段的，每一段的长度为64K字节，刚好是16位（二进制的十六位）。

near指针的长度是16位的，所以可指向的地址范围是64K字节，通常说near指针的寻址范围是64K。

far指针的长度是32位，含有一个16位的基地址和16位的偏移量，将基地址乘以16后再与偏移量相加，（所以实际上far指针是20位的长度。）即可得到far指针的1M字节的偏移量。所以far指针的寻址范围是1M字节，超过了一个段64K的容量。例如一个far指针的段地址为0x7000，偏移量为0x1244，则该指针指向地址0x71224.如果一个far指针的函数。段地址是0x7122，偏移量为0x0004，则该指针也指向地址0x71224。

如果没有指定一个指针是near或far，那么默认是near。所以far指针要显式指定。far指针工作起来要慢一些，因为每次访问一个far指针时，都要将数据段或程序段的数据交换出来。另外，far指针的运算也比较反常，例如上面讲到的far指针指向同一个地址，但是比较的结果却不相同（认为这两个指针指向的地址不同）。

什么时候使用far指针？

当使用小代码或小数据存储模式时，不能编译一个有很多代码或数据的程序。因为在64K的一个段中，不能放下所有的代码与数据。为了解决这个问题，需要指定以far函数或far指针来使用这部分的空间（64K以外的空间）。许多库函数就是显式地指定为far函数的形式。far指针通常和farmalloc()这样的内存分配函数一起使用

17. Winsock API 函数运行由WS2\_32.DLL支持，我们上面的编程通过#program实现静态加载dll，当然也可以实现动态加载从而加快程序开启速度

lib是编译时需要的，dll是运行时需要的

18.

WS2\_32.DLL 初始化：

int WSAStartup(WORD wVersionRequested,LPWSADATA lpWSAData);

该函数第一个参数为WS2\_32.DLL版本，通常设为MAKEWORD(2,0)，第二个参数为一个指向WSADATA的指针，用于返回WS2\_32.DLL的配置信息。Socket编程前需要调用该函数。

WS2\_32.DLL 释放：

int WSACleanup( );

该函数用于释放WS2\_32.DLL，不需要Winsock功能时调用该函数。

19.

clientfd = socket(AF\_INET, SOCK\_STREAM, 0);

/\*如果创建失败\*/

if (clientfd == INVALID\_SOCKET)

看来创建失败会返回INVALID\_SOCK。

20.

char str[] ="This message from server";

strlen(str)+sizeof(char)

21.#define SOMAXCONN 0x7fffffff

22.数组的总大小不能超过0x7fffffff,否则VS就会报错数组太大。比如说：

char \*data[MAXSIZE\_T];

#define MAXSIZE\_T ((SIZE\_T)~((SIZE\_T)0)) //很好地利用了~

数组应该定义为char \*data[MAXBYTE];其中MAXBYTE是

#define MAXBYTE 0xff也就是255，正是一个字节的最大大小。

23.一开始为了初始化socket库而创建的WSADATA的类型名是全大写，如果写成了WSAData那么只有在g++下面才是“有效的”（我觉得不是有效，是他根本没有发现错误）。在gcc下面才发现这是个错误。

struct sockaddr\_in注意下名字，不是socketaddr\_in。可以去掉struct（仅仅是在C++中），不过最好是SOCKADDR\_IN。

24

//ICMP计算校验和

/\* 计算校验和：按2个字节(16位)对齐进行反码加运算，然后放入校验和字段(16位)。如何有进位，进位要加到最低位上，相当于循环加了\*/

直接视为反码

unsigned short chsum(struct icmp\_header \*picmp, int len) {

long sum = 0;

unsigned short \*pusicmp = (unsigned short \*)picmp;

while (len > 1) {

sum += \*(pusicmp++);

if (sum & x080000000)

sum = (sum & 0xffff) + (sum >> 16);

len -= 2;

}

if (len)

sum += (unsigned \*)(char \*)pusicmp;

while (sum >> 16)

sum = (sum & 0xffff) + (sum >> 16);

return ~(short)sum;

}

25.

ICMP是:Internet 控制信息协议（ICMP）是 IP 组的一个整合部分。通过 IP 包传送的 ICMP 信息主要用于涉及网络操作或错误操作的不可达信息。 ICMP 包发送是不可靠的，所以主机不能依靠接收 ICMP 包解决任何网络问题。ICMP不象TCP或UDP有端口，但它确实含有两个域：类型(type)和代码(code)。而且这些域的作用和端口也完全不同。

Ping用到的是ICMP协议。不是端口，还没有到第四层。

包的构造是8位类型码+8位代码+16位反码和的校验+内容（对于ping包还要以16位icmp\_id+16位icmp\_sequence+32位icmp\_timestamp时间戳）：

struct icmp\_header {

unsigned char icmp\_type;

unsigned char icmp\_code;

unsigned short icmp\_check;

unsigned short icmp\_id;//书上的ping程序使用当前进程id。(USHORT)::GetCurrentProcessId();

unsigned short icmp\_sequence;//序列号，也就是用于分辨是第几个包

unsigned long icmp\_timestamp;//书上的ping程序这里写了0

}

对于ping包，其值分别是ICMP\_ECHO\_REQUEST，

26.在VS2010以上版本使用inet\_addr和inet\_ntoa函数就会有SDL报警告，他认为你这个函数有问题最好不要使用，给出

error C4996: 'inet\_ntoa': Use inet\_ntop() or InetNtop() instead or define \_WINSOCK\_DEPRECATED\_NO\_WARNINGS to disable deprecated API warnings。也就是说使用这个更新的函数inet\_ntop去代替inet\_ntoa，当然没准是InetNtop函数，或者使用特殊的#define关闭掉这方面警告的提醒（不过我怎么关闭无效。。。）。

有效的关闭警告手段是关闭SDL检查：在“项目”-“属性”-“配置属性”-“C/C++”-“常规”下，有一项“SDL检查”，把“是 (/sdl)”改为“否 (/sdl-)”就可以了。但这个做法可移植性不好，而且用这个函数本身有些限制还不支持IPv6。还有在一开始的项目选择中有是否选择SDL检查。

windsock2.h中的定义：

char FAR \*

WSAAPI

inet\_ntoa(

\_In\_ struct in\_addr in//此处不该有分号

);所以他的参数就是from\_addr.sin\_addr。然而inet\_addr函数的返回值是给dest\_addr.sin\_addr.S\_un.S\_addr。

in\_addr\_t inet\_addr(const char\* strptr);果然类型不一样。所以这两个函数也就是功能开起来对称。

Windows下WS2tcpip.h中的定义：

（linux下需要

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>）

inet\_pton和inet\_ntop这两函数的简单使用举例：

int main (void)

{

char IPdotdec[20]; //存放点分十进制IP地址

struct in\_addr s; // IPv4地址结构体

// 输入IP地址

printf("Please input IP address: ");

scanf("%s", IPdotdec);

// 转换

inet\_pton(AF\_INET, IPdotdec, (void \*)&s);//尤其注意第三个参数是IPv4地址结构体，而不是第二个参数的大小范围

// 反转换

inet\_ntop(AF\_INET, (void \*)&s, IPdotdec, 16); //ipv4 字符串长度加上最后的空字符

}

28.#define MAXBYTE 255已经在winsock2.h中定义。

29.inet\_addr("192.168.65.1");这是对的，但是如果192前面有一个空格就会导致程序运行闪退。

30.与虚拟机的沟通：主机是windows虚拟是linux

主机直接ipconfig，然后可以看到：

以太网适配器 VMware Network Adapter VMnet1:

连接特定的 DNS 后缀 . . . . . . . :

本地链接 IPv6 地址. . . . . . . . : fe80:ec77:b608:8187:e684

IPv4 地址 . . . . . . . . . . . . : 192.168.224.1//这就是主机看到的虚拟机ip地址吗？不是！.1结尾明显是网关地址，VMware创建的虚拟网卡持有这个IP作为网关

子网掩码 . . . . . . . . . . . . : 255.255.255.0

默认网关. . . . . . . . . . . . . :

虚拟机使用ifconfig可以看到虚拟机的ip地址是192.168.56.128，这个才是。

本机IP是在：

C:\Users\Administrator>ipconfig

Windows IP 配置

PPP 适配器 宽带连接:

连接特定的 DNS 后缀 . . . . . . . :

IPv4 地址 . . . . . . . . . . . . : 121.48.204.6//这就是本机iP，在默认的NAT模式下，虚拟机访问网页是这个ip

子网掩码 . . . . . . . . . . . . : 255.255.255.255

默认网关. . . . . . . . . . . . . : 0.0.0.0

31.端口可以用来接收数据的同时也发送数据。

32.IP、ICMP、UDP和TCP报文头部都有校验和字段，大小都是16bit，算法也基本一样。在发送数据时，为了计算数据包的校验和。应该按如下步骤：

　　（1）把校验和字段置为0；

　　（2）把需校验的数据看成以16位为单位的数字组成，依次进行二进制反码求和；

　　（3）把得到的结果存入校验和字段中。

在接收数据时，计算数据包的校验和相对简单，按如下步骤：

　　（1）把首部看成以16位为单位的数字组成，依次进行二进制反码求和，包括校验和字段；

　　（2）检查计算出的校验和的结果是否为0；

　　（3）如果等于0，说明被整除，校验是和正确。否则，校验和就是错误的，协议栈要抛弃这个数据包。

反码求和：

long sum =0;

sum += \*picmp++;//这个比用索引一个个加快

sum = (sum & 0xffff) + (sum >> 16);//为了能保存下这个进位，sum至少要是17位的

不过要注意由于我们是按照16位来处理的，所以包如果是奇数大小的话，之后还要处理一个字节的数据。

最后返回反码：我们按照无符号数计算，所以要有一次取反

return (unsigned short)~sum;

当然也可以先去反后取和。

unsigned short CheckSum(struct icmp\_header \*picmp, int len){

long sum =0;

unsigned short \*shortPIcmp= (unsigned short \*)picmp;//在这里强制转换，之后就不用一个个来了

while(len > 1) {

sum += \*shortPIcmp++;

if(sum & 0x80000000)//这个验证条件其实可以不要

sum = (sum & 0xffff) + (sum >> 16);

--len;

}

//当包的大小为奇数的时候

if(len) {

sum += \*shortPIcmp;

}

//进行进位的循环处理

while(sum >> 16) {

sum = (sum & 0xffff) + (sum >> 16);

}

return (unsigned short)~sum;//可以相加以后再取反

}

33.ping 127.0.0.1在wireshark中抓不到包，因为抓的网卡不对，应该抓回环地址的网卡。

ping里面的Data填入的内容与操作系统协议栈的实现有关系。

Frame 161319: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface 0

Ethernet II, Src: AsrockIn\_9a:26:31 (d0:50:99:9a:26:31), Dst: XiaomiCo\_d0:f5:e2 (f0:b4:29:d0:f5:e2)

Internet Protocol Version 4, Src: 192.168.1.147, Dst: 119.75.217.109

Internet Control Message Protocol

Type: 8 (Echo (ping) request)

Code: 0

Checksum: 0x4ced [correct]

[Checksum Status: Good]

Identifier (BE): 1 (0x0001)

Identifier (LE): 256 (0x0100)

Sequence number (BE): 110 (0x006e)

Sequence number (LE): 28160 (0x6e00)

[Response frame: 161344]

Data (32 bytes)

Data: 6162636465666768696a6b6c6d6e6f707172737475767761...（ASCII字母不完全循环）

[Length: 32]

34.wsaData在把地址给了WSAStartup函数以后，有以下用法：

wsaData.wVersion 这个值即MAKEWORD(2,2)即LOBYTE(wsaData.wVersion)==2 && HIBYTE(wsaData.wVersion)==2

35.用scanf来制定读取包的某一个部分是很好地做法，就是scanf("bao%d内容%d内容", &a, &b) bao即使包的一大串字符串

36.AF\_INET──指定为IPv4协议，AF\_INET6──指定为IPv6，AF\_LOCAL──指定为UNIX 协议域

37.在TCP/IP协议栈里发送和接收是不同的Buffer.Socket是全双工的，可以同时进行发送和接收！

38.可以使用setsockeopt可以把本机缓冲区大小改了。

closesocket（一般不会立即关闭而经历TIME\_WAIT的过程）后想继续重用该socket：

BOOL bReuseaddr=TRUE;

setsockopt(s,SOL\_SOCKET ,SO\_REUSEADDR,(const char\*)&bReuseaddr,sizeof(BOOL));

设置重用之后，TIME\_WAIT也能够迅速恢复

SO\_LINGER设置关闭时延，close之后还需要等待数据发送完成，不能直接关闭，所以存在关闭时延

如果要已经处于连接状态的soket在调用closesocket后强制关闭，不经历

TIME\_WAIT的过程：

BOOL bDontLinger = FALSE;

setsockopt(s,SOL\_SOCKET,SO\_DONTLINGER,(const char\*)&bDontLinger,sizeof(BOOL)); 但大多数系统不提供SO\_DONTLINGER

有效的办法是设置：/etc/sysctl.conf 具体见linux.txt

在send(),recv()过程中有时由于网络状况等原因，发收不能预期进行,而设置收发时限：

int nNetTimeout=1000;//1秒

//发送时限

setsockopt(socket，SOL\_SOCKET,SO\_SNDTIMEO，(char \*)&nNetTimeout,sizeof(int));

//接收时限

setsockopt(socket，SOL\_SOCKET,SO\_RCVTIMEO，(char \*)&nNetTimeout,sizeof(int));

在send()的时候，返回的是实际发送出去的字节(同步)或发送到socket缓冲区的字节

(异步);系统默认的状态发送和接收一次为8688字节(约为8.5K)；在实际的过程中发送数据

和接收数据量比较大，可以设置socket缓冲区，而避免了send(),recv()不断的循环收发：

// 接收缓冲区

int nRecvBuf=32\*1024;//设置为32K

setsockopt(s,SOL\_SOCKET,SO\_RCVBUF,(const char\*)&nRecvBuf,sizeof(int));

//发送缓冲区

int nSendBuf=32\*1024;//设置为32K

setsockopt(s,SOL\_SOCKET,SO\_SNDBUF,(const char\*)&nSendBuf,sizeof(int));

如果在发送数据的时，希望不经历由系统缓冲区到socket缓冲区的拷贝而影响

程序的性能： 应是直接指定了缓冲区

int nZero=0;

setsockopt(socket，SOL\_S0CKET,SO\_SNDBUF，(char \*)&nZero,sizeof(nZero));

同上在recv()完成上述功能(默认情况是将socket缓冲区的内容拷贝到系统缓冲区)：

int nZero=0;

setsockopt(socket，SOL\_S0CKET,SO\_RCVBUF，(char \*)&nZero,sizeof(int));

在client连接服务器过程中，如果处于非阻塞模式下的socket在connect()的过程中可

以设置connect()延时,直到accpet()被呼叫(本函数设置只有在非阻塞的过程中有显著的

作用，在阻塞的函数调用中作用不大)

BOOL bConditionalAccept=TRUE;

setsockopt(s,SOL\_SOCKET,SO\_CONDITIONAL\_ACCEPT,(const char\*)&bConditionalAccept,sizeof(BOOL));

39.套接字混杂模式，可以监听到ACK包。使用阻塞send()来发送倒是自己处理了。IP层面上的rawsocket。于是捕捉到的信息据说包含了协议头。

得到目标网卡句柄-》设置过滤器-》定位、初始化设备-》设置为混杂模式 由于rawsocket不支持对网卡的指定，socket是一个5元组不能用来抓包，所以这时候仅可使用winpcap，使用了PCAP\_OPENFLAG\_PROMISCUOUS标志。

winpcap

适配器信息：

pcap\_findalldev\_ex 获得本机或者远程的网络接口的列表

pcap\_open 获取适配器列表

pcap\_freealldevs 结束释放接口列表

过滤器的三个主要函数

pcap\_compile\_nopcap 对过滤器字符串的离线检查而不影响到当前在运行的抓包函数

pcap\_compile 编译过滤规则

pcap\_setfilter 设置BFP过滤规则

流量统计函数：

pcap\_setmode 设置为统计模式

pcap\_open第四个参数 定义统计的时间间隔

pcap\_loop 返回时间戳，用来计算时间间隔，如果还有到设定的时间间隔就会自动进行循环

dispatcher\_handler 获取每秒的比特数(bps)以及每秒的数据包数量(pps)

pcap\_dump 输出\*.cap流量抓捕文件

pcap\_open\_offline() 打开记录的流量文件

pcap\_next\_ex() 可以对打开的流量文件进行定位到需要分析的包

第一种使用pcap\_dispatch()或pcap\_loop()函数将会开始捕捉数据包，其中，pcap\_dispatch()将会在超时后直接返回，pcap\_loop()则一定要等到一定数量的数包被处理了以后才会返回。第二种使用pcap\_next\_ex()函数来获得缓冲区内的下一个数据包 它包含一个网络适配器的描述符和两个可以初始化和返回给用户的指针 (一个指向 pcap\_pkthdr 结构体，另一个指向数据报数据的缓冲)

40.本机上的程序只是把数据放到socket发送缓冲区中,什么时候发送给对方由协议决定。对方缓冲区收到数据就会自动返回一个ACK包，这种层面iptable可以控制。send()的返回值正常也就是成功copy到缓冲区的字节数，或者SOCK\_ERROR。在Unix系统下，如果send在等待协议传送数据时网络断开的话，调用send的进程会接收到一个SIGPIPE信号，进程对该信号的默认处理是进程终止。通过测试发现，异步socket的send函数在网络刚刚断开时还能发送返回相应的字节数，同时使用select检测也是可写的，但是过几秒钟之后，再send就会出错了，返回-1。select也不能检测出可写了

异步connect是否成功，是靠检查socket状态比如select中是否可以写入。

41.如果让同一个进程依次派生执行，那么这个多进程侦听同一个套接口是一定无法实现的，因为在bind系统调用会返回端口被占用错误，所以此时就需要由一个父进程来完成这个同一个的bind+listen动作

进程对端口的占用是唯一的，不然bind的时候就会出错

同一个端口可以tcp连接多个目标端口

42.linux的内核微调：这一点其实不算epoll的优点了，而是整个linux平台的优点。也许你可以怀疑linux平台，但是你无法回避linux平台赋予你微调内核的能力。比如，内核TCP/IP协议栈使用内存池管理sk\_buff结构，那么可以在运行时期动态调整这个内存pool（skb\_head\_pool）的大小--- 通过echo XXXX>/proc/sys/net/core/hot\_list\_length完成。再比如listen函数的第2个参数（TCP完成3次握手的数据包队列长度），也可以根据你平台内存大小动态调整。更甚至在一个数据包个数目巨大但同时每个数据包本身大小却很小的特殊系统上尝试最新的NAPI网卡驱动架构。

43.TCP四次挥手以后，那么一端调用read必然可读，且返回0(read returns 0)

read默认是阻塞的

44.套接字可读写条件：

socket可读可写条件，经常做为面试题被问，因为它考察被面试者对网络编程的基础了解的是不是够深入。

要了解socket可读可写条件，我们先了解几个概念：

1.接收缓存区低水位标记（用于读）和发送缓存区低水位标记（用于写）：

每个套接字有一个接收低水位和一个发送低水位。他们由select函数使用。

接收低水位标记是让select返回"可读"时套接字接收缓冲区中所需的数据量。对于TCP,其默认值为1。

发送低水位标记是让select返回"可写"时套接字发送缓冲区中所需的可用空间。对于TCP，其默认值常为2048.

通俗的解释一下，缓存区我们当成一个大小为 n bytes的空间，那么：

接收区缓存的作用就是，接收对面的数据放在缓存区，供应用程序读。当然了，只有当缓存区可读的数据量(接收低水位标记)到达一定程度（eg:1）的时候，我们才能读到数据，不然不就读不到数据了吗。

发送区缓存的作用就是，发送应用程序的数据到缓存区，然后一起发给对面。当然了，只有当缓存区剩余一定空间(发送低水位标记)（eg:2048）,你才能写数据进去，不然可能导致空间不够。

2.FIN: (结束标志,Finish)用来结束一个TCP回话.但对应端口仍处于开放状态,准备接收后续数据.

下列四个条件中的任何一个满足时,socket准备好读:

1. socket的接收缓冲区中的数据字节大于等于该socket的接收缓冲区低水位标记的当前大小。对这样的socket的读操作将不阻塞并返回一个大于0的值(也就是返回准备好读入的数据)。我们可以用SO\_RCVLOWATsocket选项来设置该socket的低水位标记。对于TCP和UDP .socket而言，其缺省值为1.

2. 该连接的读这一半关闭(也就是接收了FIN的TCP连接)。对这样的socket的读操作将不阻塞并返回0

3.socket是一个用于监听的socket,并且已经完成的连接数为非0.这样的soocket处于可读状态,是因为socket收到了对方的connect请求,执行了三次握手的第一步:对方发送SYN请求过来,使监听socket处于可读状态;正常情况下,这样的socket上的accept操作不会阻塞;

4.有一个socket有异常错误条件待处理.对于这样的socket的读操作将不会阻塞,并且返回一个错误(-1),errno则设置成明确的错误条件.这些待处理的错误也可通过指定socket选项SO\_ERROR调用getsockopt来取得并清除;

下列三个条件中的任何一个满足时,socket准备好写:

1. socket的发送缓冲区中的数据字节大于等于该socket的发送缓冲区低水位标记的当前大小。对这样的socket的写操作将不阻塞并返回一个大于0的值(也就是返回准备好写入的数据)。我们可以用SO\_SNDLOWAT socket选项来设置该socket的低水位标记。对于TCP和UDPsocket而言，其缺省值为2048

2. 该连接的写这一半关闭。对这样的socket的写操作将产生SIGPIPE信号，该信号的缺省行为是终止进程。

3.有一个socket异常错误条件待处理.对于这样的socket的写操作将不会阻塞并且返回一个错误(-1),errno则设置成明确的错误条件.这些待处理的错误也可以通过指定socket选项SO\_ERROR调用getsockopt函数来取得并清除;

45.setsockopt就用于bind()之前设置socket。在这里面可以设置端口被多个进程复用的允许REUSEADDR。

46.一个程序创建了一个socket，并让其监听80端口，其实是向TCP/IP协议栈声明了其对80端口的占有。以后，所有目标是80端口的TCP数据包都会转发给该程序（这里的程序，因为使用的是Socket编程接口，所以首先由Socket层来处理）。所谓accept函数，其实抽象的是TCP的连接建立过程。accept函数返回的新socket其实指代的是本次创建的连接，而一个连接是包括两部分信息的，一个是源IP和源端口，另一个是宿IP和宿端口。所以，accept可以产生多个不同的socket，而这些socket里包含的宿IP和宿端口是不变的，变化的只是源IP和源端口。这样的话，这些socket宿端口就可以都是80，而Socket层还是能根据源/宿对来准确地分辨出IP包和socket的归属关系，从而完成对TCP/IP协议的操作封装！而同时，放火墙的对IP包

47.第二种设置非阻塞方式，这种似乎可以不必在bind函数之前：

#include <fcntl.h>

#define exit\_if(r, ...) if(r) {printf(\_\_VA\_ARGS\_\_); printf("%s:%d error no: %d error msg %s\n", \_\_FILE\_\_, \_\_LINE\_\_, errno, strerror(errno)); exit(1);}

void setNonBlock(int fd) {

int flags = fcntl(fd, F\_GETFL, 0);

exit\_if(flags<0, "fcntl failed");

int r = fcntl(fd, F\_SETFL, flags | O\_NONBLOCK);

exit\_if(r<0, "fcntl failed");

}

48.套接字是用来描述连接的一端和另外一端的，而它们都是模拟端口，因为可能多个套接字占用一个端口。在linux下，套接字是文件描述符。

49.connect只是发送了SYN包，三次握手之后的步骤都是协议栈自己完成的。如果是非阻塞方式，connect返回的错误码是EINPROGRESS，是否连接成功就要靠是否可读可写来判断。超时时间在连接前设置，设置连接超时和设置发送超时是一样的 ：timeo = sock\_sndtimeo(sk, flags & O\_NONBLOCK); 从代码可以看到发送后会启用重传定时器，直到应答或者超时,每次重传的超时时间采用指数退避的方式。

如果发送syn超时没有响应，重传次数sysctl\_tcp\_sys\_retries，这个值是在tcp的系统参数设置，使用 sysctl 查看，默认设置为5

sysctl net.ipv4.tcp\_syn\_retries

kali中tcp栈设定里面syn重传次数设置是6.

短连接在后端服务异常时大量产生的TIME\_WAIT状态导致创建文件描述符失败，不能处理请求。这种情况通常的处理建议是打开tcp\_tw\_recycle 或者tcp\_tw\_reuse 选项。linux下的网络栈的规则是可控的，比如说TIME\_WAIT是有个等待时间的，但是现在可以设置不等待。。。我记得有tcp连接重用等等。这个有待研究。

50.nagle算法是以他的发明人John Nagle的名字命名的，它用于自动连接许多的小缓冲器消息；这一过程（称为nagling）通过减少必须发送包的个数来增加网络软件系统的效率。xshell或telnet这样的应用，每次键盘输入发送包含一个字符的包，却要耗费40字节的包头（tcp头加ip头），为了改进这种情况，Nagle算法的做法是先把第一个小包发送出去，后面的小包都缓存起来，直到收到前一个数据段的ack，或者缓存数据长度已经达到mss大小才发送

满足下列条件之一，需要立即发送ack，否则进入延迟确认模式：

1、收到大于mss的包且有能力接收数据

2、满足快速确认模式

3、有乱序的数据，需要对端重传

发送ARP包：iphlpapi提供的函数

#include <windows.h>

#include <stdio.h>

#include <winsock.h>

#include "iphlpapi.h"

#pragma comment ( lib, "ws2\_32.lib" )

#pragma comment ( lib, "Iphlpapi.lib" )

void main( int argc, char \*\* argv )

{

int numberOfHost = 1;

struct hostent \*remoteHostent;

//处理命令行参数

if ( argc == 3 )

numberOfHost = atoi( argv[2] );

if ( ( argc >3 ) || ( argc < 2 ) )

{

printf( "RmtHost v0.2 - Get remote HostName /MacAddress\n" );

printf( "by ShotgunLabs ( Shotgun@xici.net )\n\n" );

printf( "Usage　:\n\tRmtHost.exe [RemoteIP] \n\n" );

printf( "Example:\n\tRmtHost.exe 192.168.0.3\n" );

printf( "\tRmtHost.exe 192.168.0.3 255\n\n" );

exit( 0 );

}

//初始化SOCKET

WSADATA wsaData;

int iRet = WSAStartup(MAKEWORD(2,1), &wsaData);

if ( iRet != 0 ) {

printf( "WSAStartup Error:%d\n", GetLastError() );

exit( 0 );

}

int nRemoteAddr = inet\_addr( argv[1] );

remoteHostent= (struct hostent\*)malloc( sizeof(struct hostent ));

struct in\_addr sa;

for ( int i = 0; i < numberOfHost; i ++ )

{

//获取远程机器名

sa.s\_addr = nRemoteAddr;

printf( "\nIpAddress : %s\n", inet\_ntoa( sa ) );

remoteHostent = gethostbyaddr( (char\*)&nRemoteAddr,4, AF\_INET );

if ( remoteHostent )

printf( "HostName　: %s\n",remoteHostent->h\_name );

else

printf( "gethostbyaddr Error:%d\n",GetLastError() );

//发送ARP查询包获得远程MAC地址

unsigned char macAddress[6];

ULONG macAddLen = 6;

iRet=SendARP(nRemoteAddr, (unsigned long)NULL,(PULONG)&macAddress, &macAddLen);

if ( iRet == NO\_ERROR )

{

printf( "MacAddress: " );

for( int i =0; i<6; i++ )

{

printf( "%.2x", macAddress[i] );

if ( i<5 ) printf( "-" );

}

printf( "\n" );

}else {

printf( "SendARP Error:%d\n", GetLastError());

nRemoteAddr = htonl( ntohl( nRemoteAddr ) + 1 );

}

}

51.LSP就是TCP/IP等协议的接口．LSP用在正途上可以方便程序员们编写监视系统网络通讯情况的Sniffer，可是现在常见的LSP都被用于浏览器劫持。Winsock LSP全称Windows Socket Layered Service Provider(分层服务提供商)，它是Windows底层网络Socker通信需要经过的大门。一些流氓软件可以把自已加进去，就可以截取、访问、修改网络的数据包，可以随意添加广告，还能获取你的浏览习惯

在修复LSP以后需要进行netsh winsock reset的重启。

52.socket(AF\_INET, SOCK\_RAW, IPPROTO\_TCP|IPPROTO\_UDP|IPPROTO\_ICMP)发送接收ip数据包，不能用IPPROTO\_IP，因为如果是用了IPPROTO\_IP，系统根本就不知道该用什么协议。

53.从参数s中提取sockaddr信息存入到struct sockaddr\_in中。

int getpeername(int s, struct sockaddr \*name, socklen\_t \*namelen);描述

获取socket的对方地址

54.如果要想使用模拟 UDP原理的tracert程序： 在写程序的时候需要进行特殊的处理

http://blog.csdn.net/mrpre/article/details/43451775

1. 一个IP:Port 可以同时与多个目标维持连接，hadoop的9000端口

56.重叠IO：程序的处理代码与IO操作同时进行

利用套接字重叠IO模型，应用程序能一次投递一个或多个IO请求，当系统完成IO操作后通过事件通知应用程序。之所以可以做到是同时进行，是因为将IO操作的数据在两个态中的传送交给了操作系统，操作系统处理完成之后通知程序。

从发送和接收的角度来看，重叠IO模型与前面介绍的Select模型、WSAAsyncSelect模型和WSAEventSelect模型都不同。因为在这三个模型中IO操作还是同步的，例如：在应用程序调用recv函数时，都会在recv函数内阻塞，直到接收数据完毕后才返回。而重叠IO模型会在调用recv后立即返回。等数据准备好后再通知应用程序。

57.

===================================================================================================================================================

linux socket：

socket 是一个“文件”，函数就像是对“文件”的读写。本质是socket的描述结构体，与文件的描述结构体一毛一样。

通过端口获取服务：

#include<netdb.h> 这些头文件都是可以使用man手册查阅的

struct servent\* sp=getservbyport(htons(start\_port),"tcp");

sp->s\_name 就是服务对应的名字

linux中socket就像文件一样,可以使用socket实现lock file,,从而防止程序被多次启动,下面是通过python去编写:

def acquire\_lock():

\_\_socket = socket.socket(

socket.AF\_UNIX, socket.SOCK\_DGRAM)

try:

\_\_socket.bind('\0'+'dynpaper')

except socket.error:

print('Already running.')

exit(-1)

return \_\_socket