计 算 机 网 络
课程设计指导书

(2021 级本部，潇湘各专业**)**

湖南科技大学计算机科学与工程学院

2023 年 **12** 月

一、课程设计目的

1．加深对计算机网络的工作原理的理解

通过编写计算机程序实现、模拟网络的某些功能，将书本上抽象的概念与具体实现技术结合起来，理解并掌握计算机网络的基本工作原理及工作过程。

2．实现应用进程跨越网络的通信

了解系统调用和应用编程接口基本知识，理解应用程序和操作系统之间传递控制权的机制，掌握套接字的创建和运用，通过 socket 系统调用实现跨网通信。

3．提高网络编程和应用的能力

提高实际编程能力和灵活运用所学知识解决问题的能力。培养调查研究、查阅技术文献、资料、手册以及编写技术文档的能力，理论应用于实践的能力。

二、课程设计要求

1. 正确理解题意，仔细分析每一个实验的具体内容、步骤和要求，按设计要求完成任务。

2. 使用 C 和 C++语言，用基础 SOCKET 编程方法实现设计功能。

3. 程序设计需要完整的程序流程图、说明文档和源程序清单，设计者需要清楚每个模块的功能和原理，并有良好的编程规范和适当的注释。

4. 完成程序的编写、编译、执行和测试，每人原则上要完成 4 个题目的设计工作（第一题必做），各指导教师可以根据情况酌情增减。

5. 提交课程设计报告（含括课程设计封面（附录 6）、课程设计题目、课程设计内容、课程设计步骤、调试过程、课程设计结果及结果分析、心得体会）、程序源文件、可执行文件各一份。

三、课程设计考核方式

1. 测试中演示所设计的程序，占总成绩 20％；

2. 测试中回答指导老师所提出的问题，占总成绩 20％；

3. 设计报告，占总成绩 40％；

4. 考勤情况，占总成绩 20％。

四、课程设计题目

1、网络聊天程序的设计与实现

参照附录 1，了解 Socket 通信的原理，在此基础上编写一个聊天程序。

2、Tracert 与 Ping 程序设计与实现

参照附录 2，了解 Tracert 程序的实现原理，并调试通过。然后参考 Tracert 程序和教材 4.4.2 节，

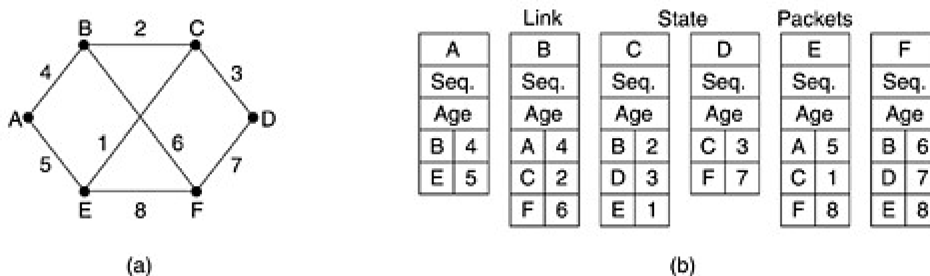
编写一个 Ping 程序，并能测试本局域网的所有机器是否在线，运行界面如下图所示的 QuickPing 程序。

3、滑动窗口协议仿真

滑动窗口协议以基于分组的数据传输协议为特征，该协议适用于在数据链路层以及传输层中对按顺序传送分组的可靠性要求较高的环境。在长管道传输过程（特别是无线环境）中，相应的滑动窗口协议可实现高效的重传恢复。附录 3 给出了一个选择性重传的滑动窗口协议的简单实现，以此为参考，设计并实现一个滑动窗口协议的仿真，显示数据传送过程中的各项具体数据，双方帧的个数变化，帧序号，发送和接受速度，重传提示等。

4、OSPF 路由协议原型系统设计与实现

参考附录 4 及教材 164 页 OSPF 路由协议工作原理，在此基础上，实现一个简单的原型系统。主要完成工作有：路由节点泛洪发布本地节点的链路信息，其它节点接收信息，构造网络拓扑，然后利用 Dijkstra（或 Floyd）算法计算出到其它节点的最短路径，最后生成本节点的路由表。设计可以以下图为例，其中 a 图是一个 6 节点的网络，每个节点生成自己的链路状态包（图 b），然后将其扩散至其他节点。



5、基于 IP 多播的网络会议程序

参照附录 5 的局域网 IP 多播程序，设计一个图形界面的网络会议程序（实现文本多播方式即可）。

6、编程模拟 NAT 网络地址转换

参考教材 188 页内容，模拟 NAT 路由器的工作过程，主要有 2 个步骤的工作：1、将收到的来自内网报文中的私有源 IP 地址转换为 NAT 的外部合法 IP 地址，同时将传输层源端口号转换为 NAT 路由器分配的端口号，建立转换映射表；2、将收到的来自外网的应答报文提取其目的 IP 地址及端口号，查找映射表，找到其对应的内网机器的 IP 地址及端口号并替换。 转换表如下图所示：

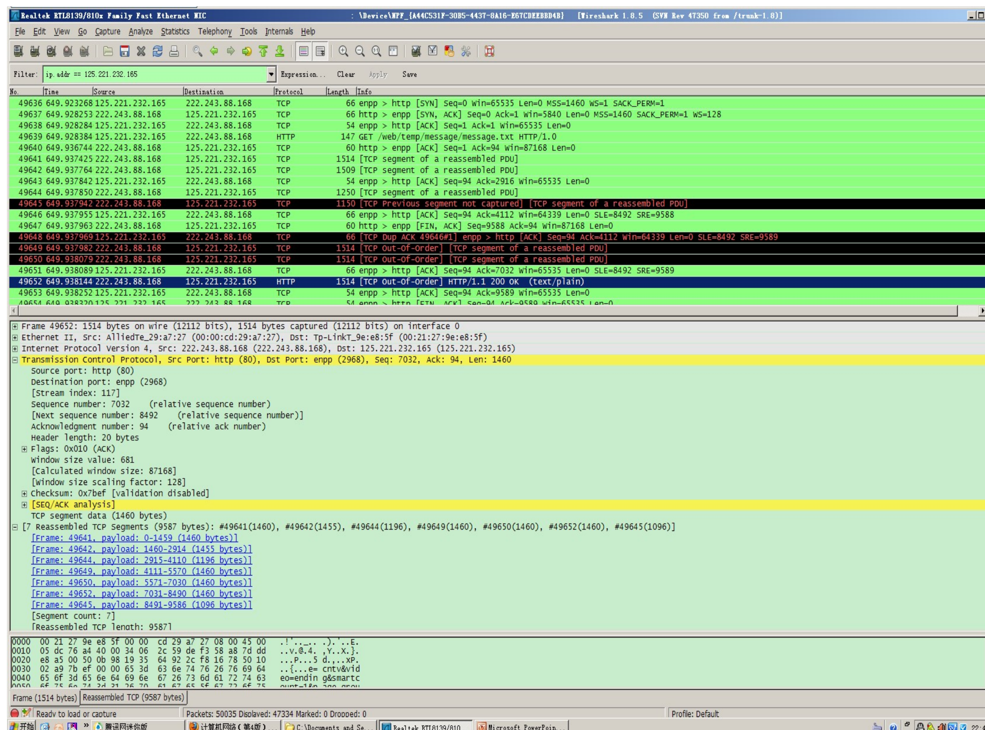
方向	字段	旧的IP地址和端口号	新的IP地址和端口号
出	源IP地址:TCP源端口	192.168.0.3:30000	172.38.1.5:40001
出	源IP地址:TCP源端口	192.168.0.4:30000	172.38.1.5:40002
入	目的IP地址:TCP目的端口	172.38.1.5:40001	192.168.0.3:30000
入	目的IP地址:TCP目的端口	172.38.1.5:40002	192.168.0.4:30000

7、网络嗅探器的设计与实现

参照附录 6 raw socket 编程例子，设计一个可以监视网络的状态、数据流动情况以及网络上传输的信息的网络嗅探器。

8、网络报文分析程序的设计与实现

在上一题的基础上，参照教材中各层报文的头部结构，结合使用 wireshark 软件（下载地址 https://www.wireshark.org/download.html#releases）观察网络各层报文捕获，解析和分析的过程（如下图所示），尝试编写一个网络报文的解析程序。



9、简单 Web Server 程序的设计与实现

Web 服务是 Internet 最方便与受用户欢迎的服务类型，它的影响力也远远超出了专业技术范畴，已广泛应用于电子商务、远程教育、远程医疗与信息服务等领域，并且有继续扩大的趋势。目前很多的 Internet 应用都是基于 Web 技术的，因此掌握 Web 环境的软件编程技术对软件人员是至关重要的。

编写简单的 Web Server 有助于读者了解 Web Server 的工作流程，掌握超文本传送协议(HTTP)基本原理，掌握 Windows 环境中用 socket 实现 C/S 结构程序的编程方法。附录 7 介绍了一个简单 Web Server 的程序设计过程。

10、 路由器查表过程模拟

参考教材 140 页 4.3 节内容，编程模拟路由器查找路由表的过程，用（目的地址 掩码 下一跳）的 IP 路由表以及目的地址作为输入，为目的地址查找路由表，找出正确的下一跳并输出结果。

五、推荐参考文献

[1] 谢希仁，计算机网络(第 8 版)，电子工业出版社，2021.

[2] Andrew S.Tanenbaum.计算机网络（第四版），北京:清华大学出版社,2004.

[3] W. Richard Stevens，UNIX 网络编程（第 1 卷）套接口 API（第 3 版），清华大学出版社，2006.
[4] 刘琰，罗军勇，常斌，Windows 网络编程课程设计，机械工业出版社，2014.
[5] 蒋清明，C 语言程序设计，中国矿业大学出版社，2011.
[6] 陈维兴，林小茶，C++面向对象程序设计(第二版), 中国铁道出版社, 2009.
[7] 中国 Linux 论坛：http://www.linuxforum.net/ .

六、附录

附录 1：Windows Socket 编程简介
附录 2：Tracert 程序源代码
附录 3：选择性重传协议
附录 4：OSPF 相关技术实现
附录 5：用 Visual C++实现局域网 IP 多播
附录 6：raw socket 编程例子(基于 LINUX 操作系统)
附录 7：简单 Web Server 程序

附录 1、Windows Socket 编程简介

使用 WinSock API 的编程，应该了解 TCP/IP 的基础知识。虽然你可以直接使用 WinSock API 来写网络应用程序，但是，要写出优秀的网络应用程序，还是必须对 TCP/IP 协议有一些了解的。

1. TCP/IP 协议与 WinSock 网络编程接口的关系

WinSock 并不是一种网络协议，它只是一个网络编程接口，也就是说，它不是协议，但是它可以访问很多种网络协议，你可以把它当作一些协议的封装。现在的 WinSock 已经基本上实现了与协议无关。你可以使用 WinSock 来调用多种协议的功能。那么，WinSock 和 TCP/IP 协议到底是什么关系呢？实际上，WinSock 就是 TCP/IP 协议的一种封装，你可以通过调用 WinSock 的接口函数来调用 TCP/IP 的各种功能.例如我想用 TCP/IP 协议发送数据，你就可以使用 WinSock 的接口函数 Send()来调用 TCP/IP 的发送数据功能，至于具体怎么发送数据，WinSock 已经帮你封装好了这种功能。

2、TCP/IP 协议介绍

TCP/IP 协议包含的范围非常的广，它是一种四层协议，包含了各种硬件、软件需求的定义。TCP/IP 协议确切的说法应该是 TCP/UDP/IP 协议。UDP 协议(User Datagram Protocol 用户数据报协议)，是一种保护消息边界的，不保障可靠数据的传输。TCP 协议(Transmission Control Protocol 传输控制协议)，是一种流传输的协议。他提供可靠的、有序的、双向的、面向连接的传输。

保护消息边界，就是指传输协议把数据当作一条独立的消息在网上传输，接收端只能接收独立的消息。也就是说存在保护消息边界，接收端一次只能接收发送端发出的一个数据包。

而面向流则是指无保护消息边界的，如果发送端连续发送数据，接收端有可能在一次接收动作中，会接收两个或者更多的数据包。

举例来说，假如，我们连续发送三个数据包，大小分别是 2k、4k、8k，这三个数据包都已经到达了接收端的网络堆栈中，如果使用 UDP 协议，不管我们使用多大的接收缓冲区去接收数据，我们必须有三次接收动作，才能够把所有的数据包接收完。而使用 TCP 协议，我们只要把接收的缓冲区大小设置在 14k 以上，我们就能够一次把所有的数据包接收下来，只需要有一次接收动作。

这就是因为 UDP 协议的保护消息边界使得每一个消息都是独立的。而流传输，却把数据当作一串数据流，它不认为数据是一个一个的消息。所以有很多人在使用 TCP 协议通讯的时候，并不清楚 TCP 是基于流的传输，当连续发送数据的时候，他们时常会认为 TCP 会丢包。其实不然，因为当它们使用的缓冲区足够大时，它们有可能会一次接收到两个甚至更多的数据包，而很多人往往会忽视这一点，只解析检查了第一个数据包，而已经接收的其它据包却被忽略了。

3. WinSock 编程简单流程

WinSock 编程分为服务器端和客户端两部分，TCP 服务器端的大体流程如下：

对于任何基于 WinSock 的编程首先必须要初始化 WinSock DLL 库。

int WSAStarup(WORD wVersionRequested,LPWSADATA lpWsAData)。

wVersionRequested 是我们要求使用的 WinSock 的版本。

调用这个接口函数可以初始化 WinSock 。

然后必须创建一个套接字(Socket)。

SOCKET Socket(int af,int type,int protocol);

套接字可以说是 WinSock 通讯的核心。WinSock 通讯的所有数据传输，都是通过套接字来完成的，套接字包含了两个信息，一个是 IP 地址，一个是 Port 端口号，使用这两个信息，就可以确定网络中的任何一个通讯节点。

当调用了 Socket()接口函数创建了一个套接字后，必须把套接字与你需要进行通讯的地址建立联

系，可以通过绑定函数 bind 来实现这种联系。

int bind(SOCKET s,const struct sockaddr FAR* name,int namelen) ;

struct sockaddr_in{

short sin_family ;

u_short sin_port;

struct in_addr sin_addr ;

char sin_sero[8] ;

}

就包含了需要建立连接的本地的地址，包括地址族、IP 和端口信息。sin_family 字段必须把它设为 AF_INET，这是告诉 WinSock 使用的是 IP 地址族。sin_port 就是要用来通讯的端口号。sin_addr 就是要用来通讯的 IP 地址信息。

在这里，必须还得提一下有关'大头(big-endian)'小头(little-endian)'。因为各种不同的计算机处理数据时的方法是不一样的，Intel X86 处理器上是用'小头'形式来表示多字节的编号，就是把低字节放在前面，把高字节放在后面，而互联网标准却正好相反，所以，必须把主机字节转换成网络字节的顺序。WinSock API 提供了几个函数。

把主机字节转化成网络字节的函数;

u_long htonl(u_long hostlong);

u_short htons(u_short hostshort);

把网络字节转化成主机字节的函数;

u_long ntohl(u_long netlong);

u_short ntohs(u_short netshort) ;

这样，设置 IP 地址和 port 端口时，就必须把主机字节转化成网络字节后，才能用 Bind()函数来绑定套接字和地址。

当绑定完成之后，服务器端必须建立一个监听的队列来接收客户端的连接请求。

int listen(SOCKET s,int backlog);

这个函数可以把套接字转成监听模式。

如果客户端有了连接请求，我们还必须使用

int accept(SOCKET s,struct sockaddr FAR* addr,int FAR* addrlen);

来接受客户端的请求。

现在基本上已经完成了一个服务器的建立，而客户端的建立的流程则是初始化 WinSock，然后创建 Socket 套接字，再使用

int connect(SOCKET s,const struct sockaddr FAR* name,int namelen) ;

来连接服务端。

下面是一个最简单的创建服务器端和客户端的例子：

服务器端的创建：

```
WSADATA wsd;
SOCKET sListen;
SOCKET sclient;
UINT port = 800;
int iAddrSize;
struct sockaddr_in local , client;
WSAStartup( 0x11 , &wsd );
sListen = Socket ( AF_INET , SOCK_STREAM , IPPOTO_IP );
local.sin_family = AF_INET;
```

```
local.sin_addr = htonl( INADDR_ANY );
local.sin_port = htons( port );
bind( sListen , (struct sockaddr*)&local , sizeof( local ) );
listen( sListen , 5 );
sClient = accept( sListen , (struct sockaddr*)&client , &iAddrSize );
```

客户端的创建：

```
WSADATA wsd;
SOCKET sClient;
UINT port = 800;
char szIp[] = "127.0.0.1";
int iAddrSize;
struct sockaddr_in server;
WSAStartup( 0x11 , &wsd );
sClient = Socket ( AF_INET , SOCK_STREAM , IPPOTO_IP );
server.sin_family = AF_INET;
server.sin_addr = inet_addr( szIp );
server.sin_port = htons( port );
connect( sClient , (struct sockaddr*)&server , sizeof( server ) );
```

当服务器端和客户端建立连接以后，无论是客户端，还是服务器端都可以使用

```
int send( SOCKET s,const char FAR* buf,int len,int flags);
int recv( SOCKET s,char FAR* buf,int len,int flags);
```

函数来接收和发送数据，因为，TCP 连接是双向的。

当要关闭通讯连接的时候，任何一方都可以调用

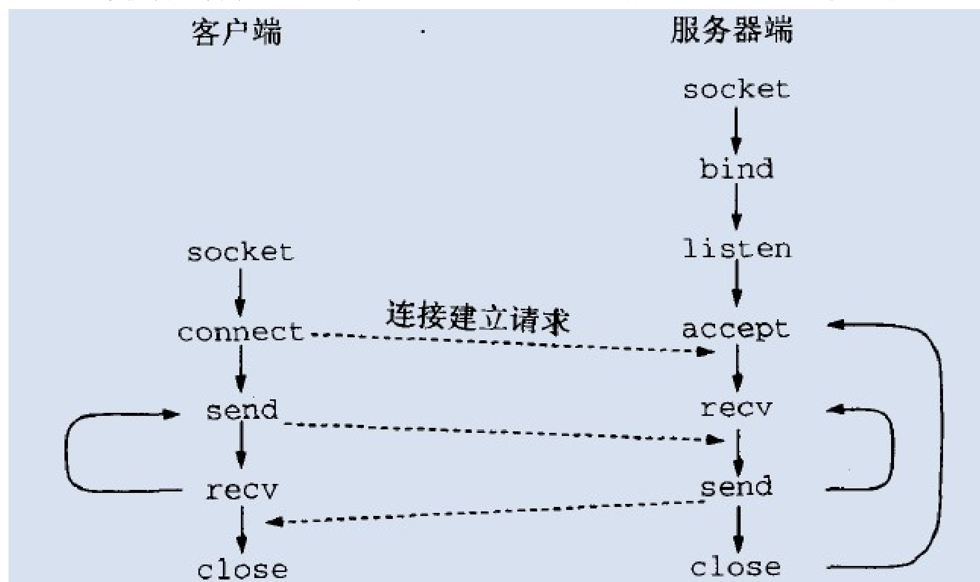
```
int shutdown(SOCKET s,int how);
```

来关闭套接字的指定功能，再调用

```
int closeSocket(SOCKET s) ;
```

来关闭套接字句柄，这样一个通讯过程就算完成了。

可以参考教材计算机网络（第 6 版）295 页图 6-32 所示的系统调用使用顺序：



注意：上面的代码没有任何检查函数返回值，如果你作网络编程就一定要检查任何一个 WinSock API 函数的调用结果，因为很多时候函数调用并不一定成功。上面介绍的函数，返回值类型是 int 的话，

如果函数调用失败的话，返回的都是 SOCKET_ERROR。

4．VC 中 socket 编程步骤

sockets（套接字）编程有三种，流式套接字（SOCK_STREAM），数据报套接字（SOCK_DGRAM），原始套接字（SOCK_RAW）；基于 TCP 的 socket 编程是采用的流式套接字。在这个程序中，将两个工程添加到一个工作区。**要链接一个 ws2_32.lib 的库文件(#pragma comment(lib,"ws2_32"))**。

服务器端编程的步骤：

1：加载套接字库，创建套接字(WSAStartup()/socket())；

2：绑定套接字到一个 IP 地址和一个端口上(bind())；

3：将套接字设置为监听模式等待连接请求(listen())；

4：请求到来后，接受连接请求，返回一个新的对应于此次连接的套接字(accept())；

5：用返回的套接字和客户端进行通信(send()/recv())；

6：返回，等待另一连接请求；

7：关闭套接字，关闭加载的套接字库(closesocket()/WSACleanup())。

服务器端代码如下：

```
#include <stdio.h>
#include <Winsock2.h>
void main()
{
    WORD wVersionRequested;
    WSADATA wsaData;
    int err;

    wVersionRequested = MAKEWORD( 1, 1 );

    err = WSAStartup( wVersionRequested, &wsaData );
    if ( err != 0 ) {
        return;
    }

    if ( LOBYTE( wsaData.wVersion ) != 1 || HIBYTE( wsaData.wVersion ) != 1 ) {
        WSACleanup( );
        return;
    }
    SOCKET sockSrv=socket(AF_INET,SOCK_STREAM,0);

    SOCKADDR_IN addrSrv;
    addrSrv.sin_addr.S_un.S_addr=htonl(INADDR_ANY);
    addrSrv.sin_family=AF_INET;
    addrSrv.sin_port=htons(6000);

    bind(sockSrv,(SOCKADDR*)&addrSrv,sizeof(SOCKADDR));

    listen(sockSrv,5);
```

```
    SOCKADDR_IN addrClient;
    int len=sizeof(SOCKADDR);
    while(1){
        SOCKET sockConn=accept(sockSrv,(SOCKADDR*)&addrClient,&len);
        char sendBuf[50];
        sprintf(sendBuf,"Welcome %s to here!",inet_ntoa(addrClient.sin_addr));
        send(sockConn,sendBuf,strlen(sendBuf)+1,0);
        char recvBuf[50];
        recv(sockConn,recvBuf,50,0);
        printf("%s\n",recvBuf);
        closesocket(sockConn);
    }
}
```

客户端编程的步骤：

1：加载套接字库，创建套接字(WSAStartup()/socket())；

2：向服务器发出连接请求(connect())；

3：和服务器端进行通信(send()/recv())；

4：关闭套接字，关闭加载的套接字库(closesocket()/WSACleanup())。

客户端的代码如下：

```
#include <stdio.h>
#include <Winsock2.h>
void main()
{
    WORD wVersionRequested;
    WSADATA wsaData;
    int err;

    wVersionRequested = MAKEWORD( 1, 1 );

    err = WSAStartup( wVersionRequested, &wsaData );
    if ( err != 0 ) {
        return;
    }
    if ( LOBYTE( wsaData.wVersion ) != 1 || HIBYTE( wsaData.wVersion ) != 1 ) {
        WSACleanup( );
        return;
    }
    SOCKET sockClient=socket(AF_INET,SOCK_STREAM,0);

    SOCKADDR_IN addrSrv;
    addrSrv.sin_addr.S_un.S_addr=inet_addr("127.0.0.1");
    addrSrv.sin_family=AF_INET;
    addrSrv.sin_port=htons(6000);
```

```
    connect(sockClient,(SOCKADDR*)&addrSrv,sizeof(SOCKADDR));
    send(sockClient,"hello",strlen("hello")+1,0);
    char recvBuf[50];
    recv(sockClient,recvBuf,50,0);
    printf("%s\n",recvBuf);
    closesocket(sockClient);
    WSACleanup();
}
```

附录 2、Tracert 程序

/*　程序名称：路由追踪(Tracert)程序

实现原理：Tracert 程序关键是对 IP 头部生存时间(time to live)TTL 字段的使用,程序实现时是向目地主机发送一个 ICMP 回显请求消息，初始时 TTL 等于 1，这样当该数据报抵达途中的第一个路由器时，TTL 的值就被减为 0，导致发生超时错误，因此该路由生成一份 ICMP 超时差错报文返回给源主机。随后，主机将数据报的 TTL 值递增 1，以便 IP 报能传送到下一个路由器，并由下一个路由器生成 ICMP 超时差错报文返回给源主机。不断重复这个过程，直到数据报达到最终的目地主机，此时目地主机将返回 ICMP 回显应答消息。这样，源主机只需对返回的每一份 ICMP 报文进行解析处理，就可以掌握数据报从源主机到达目地主机途中所经过的路由信息。

*/

```
#include <iostream>
#include <winsock2.h>
#include <ws2tcpip.h>
using namespace std;
#pragma comment(lib, "Ws2_32.lib")

//IP 报头
typedef struct
{
    unsigned char hdr_len:4;          //4 位头部长度
    unsigned char version:4;          //4 位版本号
    unsigned char tos;                //8 位服务类型
    unsigned short total_len;         //16 位总长度
    unsigned short identifier;        //16 位标识符
    unsigned short frag_and_flags;    //3 位标志加 13 位片偏移
    unsigned char ttl;                //8 位生存时间
    unsigned char protocol;           //8 位上层协议号
    unsigned short checksum;          //16 位校验和
    unsigned long sourceIP;           //32 位源 IP 地址
    unsigned long destIP;             //32 位目的 IP 地址
} IP_HEADER;

//ICMP 报头
typedef struct
{
```

```
    BYTE type;       //8 位类型字段
    BYTE code;       //8 位代码字段
    USHORT cksum;       //16 位校验和
    USHORT id;       //16 位标识符
    USHORT seq;       //16 位序列号

} ICMP_HEADER;

//报文解码结构
typedef struct
{
    USHORT usSeqNo;              //序列号
    DWORD dwRoundTripTime;       //往返时间
    in_addr dwIPaddr;              //返回报文的 IP 地址
}DECODE_RESULT;

//计算网际校验和函数
USHORT checksum(USHORT *pBuf,int iSize)
{
    unsigned long cksum=0;
    while(iSize>1)
    {
        cksum+=*pBuf++;
        iSize-=sizeof(USHORT);
    }
    if(iSize)
    {
        cksum+=*(UCHAR *)pBuf;
    }
    cksum=(cksum>>16)+(cksum&0xffff);
    cksum+=(cksum>>16);
    return (USHORT)(~cksum);
}

//对数据包进行解码
```

```
BOOL DecodeIcmpResponse(char * pBuf,int iPacketSize,DECODE_RESULT &DecodeResult,BYTE
ICMP_ECHO_REPLY,BYTE    ICMP_TIMEOUT)
{
    //检查数据报大小的合法性
    IP_HEADER* pIpHdr = (IP_HEADER*)pBuf;
    int iIpHdrLen = pIpHdr->hdr_len * 4;
    if (iPacketSize < (int)(iIpHdrLen+sizeof(ICMP_HEADER)))
        return FALSE;
    //根据 ICMP 报文类型提取 ID 字段和序列号字段
    ICMP_HEADER *pIcmpHdr=(ICMP_HEADER *)(pBuf+iIpHdrLen);
    USHORT usID,usSquNo;
    if(pIcmpHdr->type==ICMP_ECHO_REPLY)       //ICMP 回显应答报文
    {
        usID=pIcmpHdr->id;              //报文 ID
        usSquNo=pIcmpHdr->seq;       //报文序列号
    }
    else if(pIcmpHdr->type==ICMP_TIMEOUT)//ICMP 超时差错报文
    {
        char * pInnerIpHdr=pBuf+iIpHdrLen+sizeof(ICMP_HEADER); //载荷中的 IP 头
        int iInnerIPHdrLen=((IP_HEADER *)pInnerIpHdr)->hdr_len*4; //载荷中的 IP 头长
        ICMP_HEADER * pInnerIcmpHdr=(ICMP_HEADER *)(pInnerIpHdr+iInnerIPHdrLen);//载
荷中的 ICMP 头
        usID=pInnerIcmpHdr->id;              //报文 ID
        usSquNo=pInnerIcmpHdr->seq;       //序列号
    }
    else
    {
        return false;
    }

    //检查 ID 和序列号以确定收到期待数据报
    if(usID!=(USHORT)GetCurrentProcessId()||usSquNo!=DecodeResult.usSeqNo)
    {
        return false;
    }
```

```
        //记录 IP 地址并计算往返时间
        DecodeResult.dwIPaddr.s_addr=pIpHdr->sourceIP;
        DecodeResult.dwRoundTripTime=GetTickCount()-DecodeResult.dwRoundTripTime;

        //处理正确收到的 ICMP 数据报
        if (pIcmpHdr->type == ICMP_ECHO_REPLY ||pIcmpHdr->type == ICMP_TIMEOUT)
        {
            //输出往返时间信息
            if(DecodeResult.dwRoundTripTime)
                cout<<"         "<<DecodeResult.dwRoundTripTime<<"ms"<<flush;
            else
                cout<<"         "<<"<1ms"<<flush;
        }
        return true;
    }

    void main()
    {
        //初始化 Windows sockets 网络环境
        WSADATA wsa;
        WSAStartup(MAKEWORD(2,2),&wsa);
        char IpAddress[255];
        cout<<"请输入一个 IP 地址或域名：";
        cin>>IpAddress;
        //得到 IP 地址
        u_long ulDestIP=inet_addr(IpAddress);
        //转换不成功时按域名解析
        if(ulDestIP==INADDR_NONE)
        {
            hostent * pHostent=gethostbyname(IpAddress);
            if(pHostent)
            {
                ulDestIP=(*(in_addr*)pHostent->h_addr).s_addr;
            }
            else
```

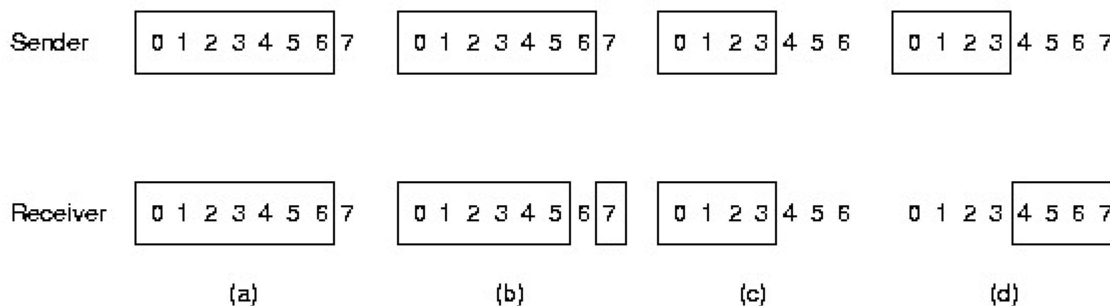
```
        {
            cout<<"输入的 IP 地址或域名无效!"<<endl;
            WSACleanup();
            return;
        }
    }
    cout<<"Tracing route to "<<IpAddress<<" with a maximum of 30 hops.\n"<<endl;
    //填充目地端 socket 地址
    sockaddr_in destSockAddr;
    ZeroMemory(&destSockAddr,sizeof(sockaddr_in));
    destSockAddr.sin_family=AF_INET;
    destSockAddr.sin_addr.s_addr=ulDestIP;
    //创建原始套接字
    SOCKET sockRaw=WSASocket(AF_INET,SOCK_RAW,IPPROTO_ICMP,NULL,0,
WSA_FLAG_OVERLAPPED);
    //超时时间
    int iTimeout=3000;
    //接收超时
    setsockopt(sockRaw,SOL_SOCKET,SO_RCVTIMEO,(char *)&iTimeout,sizeof(iTimeout));
    //发送超时
    setsockopt(sockRaw,SOL_SOCKET,SO_SNDTIMEO,(char *)&iTimeout,sizeof(iTimeout));

    //构造 ICMP 回显请求消息，并以 TTL 递增的顺序发送报文
    //ICMP 类型字段
    const BYTE ICMP_ECHO_REQUEST=8;     //请求回显
    const BYTE ICMP_ECHO_REPLY=0;       //回显应答
    const BYTE ICMP_TIMEOUT=11;             //传输超时

    //其他常量定义
    const int DEF_ICMP_DATA_SIZE=32;       //ICMP 报文默认数据字段长度
    const int MAX_ICMP_PACKET_SIZE=1024;//ICMP 报文最大长度（包括报头）
    const DWORD DEF_ICMP_TIMEOUT=3000;      //回显应答超时时间
    const int DEF_MAX_HOP=30;                  //最大跳站数

    //填充 ICMP 报文中每次发送时不变的字段
```

```
char IcmpSendBuf[sizeof(ICMP_HEADER)+DEF_ICMP_DATA_SIZE];//发送缓冲区
memset(IcmpSendBuf, 0, sizeof(IcmpSendBuf));          //初始化发送缓冲区
char IcmpRecvBuf[MAX_ICMP_PACKET_SIZE];          //接收缓冲区
memset(IcmpRecvBuf, 0, sizeof(IcmpRecvBuf));          //初始化接收缓冲区

ICMP_HEADER * pIcmpHeader=(ICMP_HEADER*)IcmpSendBuf;
pIcmpHeader->type=ICMP_ECHO_REQUEST;          //类型为请求回显
pIcmpHeader->code=0;          //代码字段为 0
pIcmpHeader->id=(USHORT)GetCurrentProcessId();     //ID 字段为当前进程号
memset(IcmpSendBuf+sizeof(ICMP_HEADER),'E',DEF_ICMP_DATA_SIZE);//数据字段

USHORT usSeqNo=0;          //ICMP 报文序列号
int iTTL=1;          //TTL 初始值为 1
BOOL bReachDestHost=FALSE;     //循环退出标志
int iMaxHot=DEF_MAX_HOP;     //循环的最大次数
DECODE_RESULT DecodeResult;     //传递给报文解码函数的结构化参数
while(!bReachDestHost&&iMaxHot--)
{
    //设置 IP 报头的 TTL 字段
    setsockopt(sockRaw,IPPROTO_IP,IP_TTL,(char *)&iTTL,sizeof(iTTL));
    cout<<iTTL<<flush;     //输出当前序号
    //填充 ICMP 报文中每次发送变化的字段
    ((ICMP_HEADER *)IcmpSendBuf)->cksum=0;     //校验和先置为 0
    ((ICMP_HEADER *)IcmpSendBuf)->seq=htons(usSeqNo++);     //填充序列号
    ((ICMP_HEADER *)IcmpSendBuf)->cksum=checksum((USHORT *)IcmpSendBuf,
sizeof(ICMP_HEADER)+DEF_ICMP_DATA_SIZE); //计算校验和

    //记录序列号和当前时间
    DecodeResult.usSeqNo=((ICMP_HEADER*)IcmpSendBuf)->seq;     //当前序号
    DecodeResult.dwRoundTripTime=GetTickCount();          //当前时间
    //发送 TCP 回显请求信息
  sendto(sockRaw,IcmpSendBuf,sizeof(IcmpSendBuf),0,(sockaddr*)&destSockAddr,sizeof(destSockAdd
r));
    //接收 ICMP 差错报文并进行解析处理
    sockaddr_in from;          //对端 socket 地址
```

```
int iFromLen=sizeof(from);      //地址结构大小
int iReadDataLen;               //接收数据长度
while(1)
{
    //接收数据
iReadDataLen=recvfrom(sockRaw,IcmpRecvBuf,MAX_ICMP_PACKET_SIZE,0,(sockaddr*)&from,&
iFromLen);
    if(iReadDataLen!=SOCKET_ERROR)//有数据到达
    {
        //对数据包进行解码
if(DecodeIcmpResponse(IcmpRecvBuf,iReadDataLen,DecodeResult,ICMP_ECHO_REPLY,ICMP_TI
MEOUT))
        {
            //到达目的地，退出循环
            if(DecodeResult.dwIPaddr.s_addr==destSockAddr.sin_addr.s_addr)
                bReachDestHost=true;
            //输出 IP 地址
            cout<<'\t'<<inet_ntoa(DecodeResult.dwIPaddr)<<endl;
            break;
        }
    }
    else if(WSAGetLastError()==WSAETIMEDOUT)      //接收超时，输出*号
    {
        cout<<"              *"<<'\t'<<"Request timed out."<<endl;
        break;
    }
    else
    {
        break;
    }
}
iTTL++;      //递增 TTL 值
}
}
```

附录 3、选择性重传协议

链路层的滑动窗口协议大致分为停等协议，后退 n 帧协议，以及选择性重传协议。停等协议效率较低，后退 n 帧协议在链路状态较差的时候效率会急剧下降。选择性重传协议在一定程度上可以弥补这个缺陷。其原理在于接收端会设置多个缓存来接收所有到达帧，当某个帧出现错误时，只会要求重传这一个帧，只有当某个序号后的所有帧都正确收到后，才会一起提交给高层应用。需要注意的是，当采用 n 比特来标识缓存窗口时，发送端一次最多只能发送 2^{n-1} 个帧。如超过该限制，则会使协议发生错误，如下图 a，b 所示：假设发送窗口和接收窗口大小都为 7，发送方发送连续的 0～6 号帧，接收方全部接收，返回 Ack 0～Ack 6，同时接收方移动窗口为 7,0,1,2,3,4,5；接收方返回的 Ack 0～Ack 6 全部丢失；发送方重发 0～6 号帧；接收方判断 0,1,2,3,4,5 号帧落入接收窗口，将 0~5 帧接收下来(此时协议已经出现错误)，但由于 7 号帧没有收到，接收返回的确认帧一直为 Ack 6；发送方移动发送窗口，发送 7,0,1,2,3,4,5 号帧；接收方将 7 号帧接收，丢弃新的 0,1,2,3,4,5 号帧(协议出错)，并将 7,0,1,2,3,4,5 号帧(其中 0,1,2,3,4,5 号帧为重复帧)组装好交给网络层，并依次返回 Ack 0,1,2,3,4,5 帧，从而造成协议失败。其原因在于，接收方在移动窗口后，新旧窗口有重叠，导致接收方在接收到一批帧时不能分辨这些帧是新帧还是重复帧。如果发送方的窗口是 2^{n-1} 的话（发送 0,1,2,3），接收方接受后将滑动至 4,5,6,7，就没有重复的可能了，如下图 c，d 所示。




```
/* Protocol 6 (nonsequential receive) accepts frames out of order, but passes packets to the
   network layer in order. Associated with each outstanding frame is a timer. When the timer
   goes off, only that frame is retransmitted, not all the outstanding frames, as in protocol 5. */

#define MAX_SEQ 7 /* should be 2^n - 1 */
#define NR_BUFS ((MAX_SEQ + 1)/2)
typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready, ack_timeout} event_type;
#include "protocol.h"
boolean no_nak = true;/* no nak has been sent yet */
seq_nr oldest_frame = MAX_SEQ+1;    /* init value is for the simulator */

static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
/* Same as between in protocol5, but shorter and more obscure. */
  return ((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a));
```

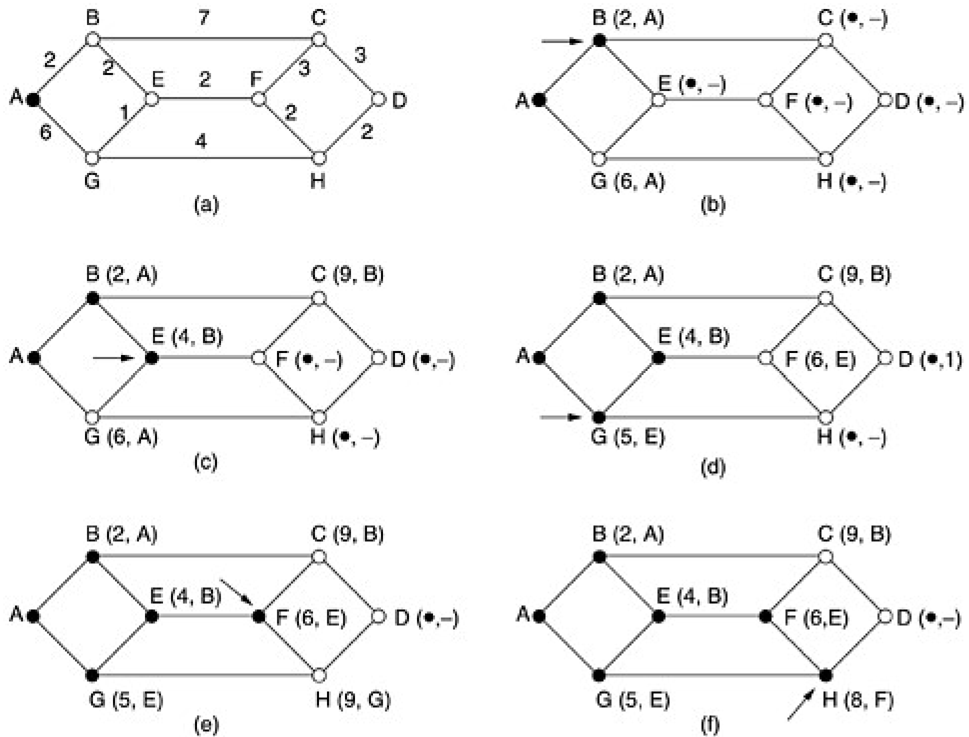
```
}

static void send_frame(frame_kind fk, seq_nr frame_nr, seq_nr frame_expected, packet buffer[])
{
/* Construct and send a data, ack, or nak frame. */
  frame s;    /* scratch variable */

  s.kind = fk;    /* kind == data, ack, or nak */
  if (fk == data) s.info = buffer[frame_nr % NR_BUFS];
  s.seq = frame_nr;    /* only meaningful for data frames */
  s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
  if (fk == nak) no_nak = false;    /* one nak per frame, please */
  to_physical_layer(&s);   /* transmit the frame */
  if (fk == data) start_timer(frame_nr % NR_BUFS);
  stop_ack_timer();    /* no need for separate ack frame */
}

void protocol6(void)
{
  seq_nr ack_expected;/* lower edge of sender's window */
  seq_nr next_frame_to_send;    /* upper edge of sender's window + 1 */
  seq_nr frame_expected;   /* lower edge of receiver's window */
  seq_nr too_far; /* upper edge of receiver's window + 1 */
  int i; /* index into buffer pool */
  frame r;    /* scratch variable */
  packet out_buf[NR_BUFS]; /* buffers for the outbound stream */
  packet in_buf[NR_BUFS];  /* buffers for the inbound stream */
  boolean arrived[NR_BUFS];     /* inbound bit map */
  seq_nr nbuffered;    /* how many output buffers currently used */
  event_type event;

  enable_network_layer();  /* initialize */
  ack_expected = 0;    /* next ack expected on the inbound stream */
  next_frame_to_send = 0;  /* number of next outgoing frame */
  frame_expected = 0; /* frame number expected */
  too_far = NR_BUFS;   /* receiver's upper window + 1 */
  nbuffered = 0;  /* initially no packets are buffered */

  for (i = 0; i < NR_BUFS; i++) arrived[i] = false;
  while (true) {
     wait_for_event(&event);    /* five possibilities: see event_type above */
     switch(event) {
        case network_layer_ready:   /* accept, save, and transmit a new frame */
                nbuffered = nbuffered + 1;   /* expand the window */
```

```
                from_network_layer(&out_buf[next_frame_to_send % NR_BUFS]); /* fetch new packet */
                send_frame(data, next_frame_to_send, frame_expected, out_buf); /* transmit the frame
*/
                inc(next_frame_to_send);    /* advance upper window edge */
                break;

        case frame_arrival:    /* a data or control frame has arrived */
                from_physical_layer(&r);    /* fetch incoming frame from physical layer */
                if (r.kind == data) {
                        /* An undamaged frame has arrived. */
                        if ((r.seq != frame_expected) && no_nak)
                            send_frame(nak, 0, frame_expected, out_buf); else start_ack_timer();
                        if (between(frame_expected, r.seq, too_far) && (arrived[r.seq%NR_BUFS] ==
false)) {
                                /* Frames may be accepted in any order. */
                                arrived[r.seq % NR_BUFS] = true;  /* mark buffer as full */
                                in_buf[r.seq % NR_BUFS] = r.info; /* insert data into buffer */
                                while (arrived[frame_expected % NR_BUFS]) {
                                        /* Pass frames and advance window. */
                                        to_network_layer(&in_buf[frame_expected % NR_BUFS]);
                                        no_nak = true;
                                        arrived[frame_expected % NR_BUFS] = false;
                                        inc(frame_expected);  /* advance lower edge of receiver's
window */
                                        inc(too_far);/* advance upper edge of receiver's window */
                                        start_ack_timer();    /* to see if (a separate ack is needed
*/
                                }
                        }
                }
                if((r.kind==nak) && between(ack_expected,(r.ack+1)%(MAX_SEQ+1),next_frame_to_send))
                        send_frame(data, (r.ack+1) % (MAX_SEQ + 1), frame_expected, out_buf);

                while (between(ack_expected, r.ack, next_frame_to_send)) {
                        nbuffered = nbuffered - 1;   /* handle piggybacked ack */
                        stop_timer(ack_expected % NR_BUFS);    /* frame arrived intact */
                        inc(ack_expected);  /* advance lower edge of sender's window */
                }
                break;

        case cksum_err: if (no_nak) send_frame(nak, 0, frame_expected, out_buf); break; /* damaged
frame */
        case timeout: send_frame(data, oldest_frame, frame_expected, out_buf); break;   /* we timed
out */
```

```
        case ack_timeout: send_frame(ack,0,frame_expected, out_buf);   /* ack timer expired; send ack
*/
     }

     if (nbuffered < NR_BUFS) enable_network_layer(); else disable_network_layer();
  }
}
```

附录 4、 OSPF 相关技术实现

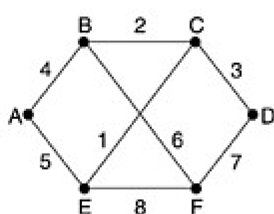
下图 a-f 显示了 Dijkstra 算法的运行过程，节点 A 寻找一条到节点 D 的最短路径。



算法的代码如下图示：

```
#define MAX_NODES 1024                  /* maximum number of nodes */
#define INFINITY 1000000000             /* a number larger than every maximum path */
int n, dist[MAX_NODES][MAX_NODES];/* dist[i][j] is the distance from i to j */

void shortest_path(int s, int t, int path[])
{ struct state {                        /* the path being worked on */
      int predecessor;                  /* previous node */
      int length;                       /* length from source to this node */
      enum {permanent, tentative} label; /* label state */
  } state[MAX_NODES];

  int i, k, min;
  struct state *p;

  for (p = &state[0]; p < &state[n]; p++) { /* initialize state */
      p->predecessor = -1;
      p->length = INFINITY;
      p->label = tentative;
  }
  state[t].length = 0;  state[t].label = permanent;
  k = t;                                /* k is the initial working node */
  do {                                  /* Is there a better path from k? */
      for (i = 0; i < n; i++)           /* this graph has n nodes */
          if (dist[k][i] != 0 && state[i].label == tentative) {
              if (state[k].length + dist[k][i] < state[i].length) {
                  state[i].predecessor = k;
                  state[i].length = state[k].length + dist[k][i];
              }
          }

      /* Find the tentatively labeled node with the smallest label. */
      k = 0; min = INFINITY;
      for (i = 0; i < n; i++)
          if (state[i].label == tentative && state[i].length < min) {
              min = state[i].length;
              k = i;
          }
      state[k].label = permanent;
  } while (k != s);

  /* Copy the path into the output array. */
  i = 0;  k = s;
  do {path[i++] = k; k = state[k].predecessor; } while (k >= 0);
}
```

OSPF 算法的链路状态包如下图所示，如果简化处理，构造状态包时可以不考虑 seq 及 Age，只考虑邻居信息。



(a)

Link State Packets

A		B		C		D		E		F	
Seq.		Seq.		Seq.		Seq.		Seq.		Seq.	
Age		Age		Age		Age		Age		Age	
B	4	A	4	B	2	C	3	A	5	B	6
E	5	C	2	D	3	F	7	C	1	D	7
		F	6	E	1			F	8	E	8

(b)

附录5、 用Visual C++实现局域网IP多播

在局域网中，管理员常常需要将某条信息发送给一组用户。如果使用一对一的发送方法，虽然是可行的，但是过于麻烦，也常会出现漏发、错发。为了更有效的解决这种组通信问题，出现了一种多播技术（也常称为组播通信），它是基于IP层的通信技术。为了帮助读者理解，下面将简要的介绍一下多播的概念。

众所周知，普通IP通信是在一个发送者和一个接收者之间进行的，我们常把它称为点对点的通信，但对于有些应用，这种点对点的通信模式不能有效地满足实际应用的需求。例如：一个数字电话会议系统由多个会场组成，当在其中一个会场的参会人发言时，要求其它会场都能即时的得到此发言的内容，这是一个典型的一对多的通信应用，通常把这种一对多的通信称为多播通信。采用多播通信技术，不仅可以实现一个发送者和多个接收者之间进行通信的功能，而且可以有效减轻网络通信的负担，避免资源的无谓浪费。

广播也是一种实现一对多数据通信的模式，但广播与多播在实现方式上有所不同。广播是将数据从一个工作站发出，局域网内的其他所有工作站都能收到它。这一特征适用于无连接协议，因为LAN上的所有机器都可获得并处理广播消息。使用广播消息的不利之处是每台机器都必须对该消息进行处理。多播通信则不同，数据从一个工作站发出后，如果在其它 LAN 上的机器上面运行的进程表示对这些数据"有兴趣"，多播数据才会发给它们。

本实例由Sender和Receiver两个程序组成，Sender用户从控制台上输入多播发送数据，Receiver端都要求加入同一个多播组，完成接收Sender发送的多播数据。

一、实现方法

1、 协议支持

并不是所有的协议都支持多播通信，对 Win32 平台而言，仅两种可从 WinSock 内访问的协议（IP/ATM）才提供了对多播通信的支持。因通常通信应用都建立在 TCP/IP 协议之上的，所以本文只针对IP协议来探讨多播通信技术。

支持多播通信的平台包括Windows CE 2.1、Windows 95、Windows 98、Windows NT 4、Windows 2000 和 WindowsXP。自 2.1 版开始，Windows CE 才开始实现对 IP 多播的支持。本文实例建立在WindowsXP专业版平台上。

2、多播地址

IP采用D类地址来支持多播。每个D类地址代表一组主机。共有28位可用来标识小组。所以可以同时有多达 25 亿个小组。当一个进程向一个 D 类地址发送分组时，会尽最大的努力将它送给小组的所有成员，但不能保证全部送到。有些成员可能收不到这个分组。举个例子来说，假定五个节点都想通过 I P 多播，实现彼此间的通信，它们便可加入同一个组地址。全部加入之后，由一个节点发出的任何数据均会一模一样地复制一份，发给组内的每个成员，甚至包括始发数据的那个节点。D类I P地址范围在224.0.0.0到239.255.255.255之间。它分为两类：永久地址和临时地址。永久地址是为特殊用途而保留的。比如，224.0.0.0根本没有使用（也不能使用），224.0.0.1代表子网内的所有系统（主机），而224.0.0.2代表子网内的所有路由器。在 RFC 1700文件中，提供了所有保留地址的一个详细清单。该文件是为特殊用途保留的所有资源的一个列表，大家可以找来作为参考。"Internet分配数字专家组"（I A N A）负责着这个列表的维护。在表1中，我们总结了目前标定为"保留"的一些地址。临时组地址在使用前必须先创建，一个进程可以要求其主机加入特定的组，它也能要求其主机脱离该组。当主机上的最后一个进程脱离某个组后，该组地址就不再在这台主机中出现。每个主机都要记录它的进程当前属于哪个组。 表1 部分永久地址说明

地 址 说 明

224.0.0.0 基本地址（保留）

224.0.0.1 子网上的所有系统

224.0.0.2 子网上的所有路由器

224.0.0.5 子网上所有 OSPF 路由器

224.0.0.6 子网上所有指定的 OSPF 路由器

224.0.0.9 RIP 第 2 版本组地址

224.0.1.1 网络时间协议

224.0.1.24 WINS 服务器组地址

3、 多播路由器

多播由特殊的多播路由器来实现，多播路由器同时也可以是普通路由器。各个多播路由器每分钟发送一个硬件多播信息给子网上的主机(目的地址为 224.0.0.1)，要求它们报告其进程当前所属的是哪一组，各主机将它感兴趣的 D 类地址返回。这些询问和响应分组使用 IGMP（Internet group management protocol），它大致类似于 ICMP。它只有两种分组：询问和响应，都有一个简单的固定格式，其中有效载荷字段的第一个字段是一些控制信息，第二字段是一个 D 类地址，在 RFC1112 中有详细说明。

多播路由器的选择是通过生成树实现的，每个多播路由器采用修改过的距离矢量协议和其邻居交换信息，以便向每个路由器为每一组构造一个覆盖所有组员的生成树。在修剪生成树及删除无关路由器和网络时，用到了很多优化方法。

4．库支持

WinSock 提供了实现多播通信的 API 函数调用。针对 IP 多播，WinSock 提供了两种不同的实现方法，具体取决于使用的是哪个版本的 WinSock。第一种方法是 WinSock1 提供的，要求通过套接字选项来加入一个组；另一种方法是 WinSock2 提供的，它是引入一个新函数，专门负责多播组的加入，这个函数便是 WSAJoinLeaf，它是基层协议无关的。本文将通过一个多播通信的实例的实现过程，来讲叙多播实现的主要步骤。因为 Window98 以后版本都安装了 Winsock2.0 以上版本，所以本文实例在 WinSock2.0 平台上开发的，但在其中对 WinSock1 实现不同的地方加以说明。

二、编程步骤

1、启动 Visual C++6.0，创建一个控制台项目工程 MultiCast。在此项目工程中添加 Sender 和 Receiver 两个项目。

Receiver 项目实现步骤：

(1)、创建一个 SOCK_DGRAM 类型的 Socket。

(2)、将此 Socket 绑定到本地的一个端口上，为了接收服务器端发送的多播数据。

(3)、加入多播组。

①、 WinSock2 中引入一个 WSAJoinLeaf，此函数原型如下：

SOCKET WSAJoinLeaf(SOCKET s, const struct sockaddr FAR *name, int namelen, LPWSABUF lpCallerData, LPWSABUF lpCalleeData, LPQOS lpSQOS, LPQOS lpGQOS, DWORD dwFlags);

其中，第一个参数 s 代表一个套接字句柄，是自 WSASocket 返回的。传递进来的这个套接字必须使用恰当的多播标志进行创建；否则的话 WSAJoinLeaf 就会失败，并返回错误 WSAEINVAL。第二个参数是 SOCKADDR（套接字地址）结构，具体内容由当前采用的协议决定，对于 IP 协议来说，这个地址指定的是主机打算加入的那个多播组。第三个参数 namelen（名字长度）是用于指定 name 参数的长度，以字节为单位。第四个参数 lpCallerData（呼叫者数据）的作用是在会话建立之后，将一个数据缓冲区传输给自己通信的对方。第五个参数 lpCalleeData（被叫者数据）用于初始化一个缓冲区，在会话建好之后，接收来自对方的数据。注意在当前的 Windows 平台上，lpCallerData 和 lpCalleeData 这两个参数并未真正实现，所以均应设为 NULL。LpSQOS 和 lpGQOS 这两个参数是有关 Qos（服务质量）的设置，通常也设为 NULL，有关 Qos 内容请参阅 MSDN 或有关书籍。最后一个参数 dwFlags 指出该

主机是发送数据、接收数据或收发兼并。该参数可选值分别是：JL_SENDER_ONLY、JL_RECEIVER_ONLY 或者 JL_BOTH。

②、在 WinSock1 平台上加入多播组需要调用 setsockopt 函数，同时设置 IP_ADD_MEMBERSHIP 选项，指定想加入的那个组的地址结构。具体实现代码将在下面代码注释列出。

(4)、接收多播数据。

Sender 实现步骤：

(1)、创建一个 SOCK_DGRAM 类型的 Socket。

(2)、加入多播组。

(3)、发送多播数据。

3、编译两个项目，在局域网中按如下步骤测试：

（1）、将 Sender.exe 拷贝到发送多播数据的ＰＣ上。

（2）、将 Receiver.exe 拷贝到多个要求接收多播数据的ＰＣ上。

（3）、各自运行相应的程序。

（4）、在 Sender PC 上输入多播数据后，你就可以在 Receiver PC 上看到输入的多播数据。

```
//sender.cpp
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>
#define MCASTADDR "233.0.0.1" //本例使用的多播组地址。
#define MCASTPORT 5150 //本地端口号。
#define BUFSIZE 1024 //发送数据缓冲大小。
#pragma comment(lib,"ws2_32")
int main( int argc,char ** argv)
{
    WSADATA wsd;
    struct sockaddr_in remote;
    SOCKET sock,sockM;
    TCHAR sendbuf[BUFSIZE];
    int len = sizeof( struct sockaddr_in);
    //初始化 WinSock2.2
    if( WSAStartup( MAKEWORD(2,2),&wsd) != 0 )
    {
        printf("WSAStartup() failed\n");
        return -1;
    }
    if((sock=WSASocket(AF_INET,SOCK_DGRAM,0,NULL,0,
        WSA_FLAG_MULTIPOINT_C_LEAF|WSA_FLAG_MULTIPOINT_D_LEAF|
        WSA_FLAG_OVERLAPPED)) == INVALID_SOCKET)
    {
        printf("socket failed with:%d\n",WSAGetLastError());
        WSACleanup();
        return -1;
    }
```

```
        //加入多播组
        remote.sin_family = AF_INET;
        remote.sin_port = htons(MCASTPORT);
        remote.sin_addr.s_addr = inet_addr( MCASTADDR );
        if(( sockM = WSAJoinLeaf(sock,(SOCKADDR*)&remote,
            sizeof(remote),NULL,NULL,NULL,NULL,
            JL_BOTH)) == INVALID_SOCKET)
        {
            printf("WSAJoinLeaf() failed:%d\n",WSAGetLastError());
            closesocket(sock);
            WSACleanup();
            return -1;
        }

        //发送多播数据，当用户在控制台输入"QUIT"时退出。
        while(1)
        {
            printf("SEND : ");
            scanf("%s",sendbuf);
            if( sendto(sockM,(char*)sendbuf,strlen(sendbuf),0,(struct sockaddr*)

                &remote,sizeof(remote))==SOCKET_ERROR)
            {
                printf("sendto failed with: %d\n",WSAGetLastError());
                closesocket(sockM);
                closesocket(sock);
                WSACleanup();
                return -1;
            }
            if(strcmp(sendbuf,"QUIT")==0) break;
            Sleep(500);
        }

        closesocket(sockM);
        closesocket(sock);
        WSACleanup();
        return 0;
    }

    //receiver.cpp
    #include <winsock2.h>
    #include <ws2tcpip.h>
    #include <stdio.h>
    #include <stdlib.h>
```

```
#define MCASTADDR "233.0.0.1" //本例使用的多播组地址。
#define MCASTPORT 5150 //绑定的本地端口号。
#define BUFSIZE 1024 //接收数据缓冲大小。
#pragma comment(lib,"ws2_32")
int main( int argc,char ** argv)
{
    WSADATA wsd;
    struct sockaddr_in local,remote,from;
    SOCKET sock,sockM;
    TCHAR recvbuf[BUFSIZE];
    /*struct ip_mreq mcast; // Winsock1.0 */

    int len = sizeof( struct sockaddr_in);
    int ret;
    //初始化 WinSock2.2
    if( WSAStartup( MAKEWORD(2,2),&wsd) != 0 )
    {
        printf("WSAStartup() failed\n");
        return -1;
    }
    /*
    创建一个 SOCK_DGRAM 类型的 SOCKET
    其中,WSA_FLAG_MULTIPOINT_C_LEAF 表示 IP 多播在控制面层上属于
    "无根"类型;
    WSA_FLAG_MULTIPOINT_D_LEAF 表示 IP 多播在数据面层上属于"无根",
    有关控制面层和
    数据面层有关概念请参阅 MSDN 说明。
    */
    if((sock=WSASocket(AF_INET,SOCK_DGRAM,0,NULL,0,
        WSA_FLAG_MULTIPOINT_C_LEAF|WSA_FLAG_MULTIPOINT_D_LEAF|
        WSA_FLAG_OVERLAPPED)) == INVALID_SOCKET)
    {
        printf("socket failed with:%d\n",WSAGetLastError());
        WSACleanup();
        return -1;
    }
    //将 sock 绑定到本机某端口上。
    local.sin_family = AF_INET;
    local.sin_port = htons(MCASTPORT);
    local.sin_addr.s_addr = INADDR_ANY;
    if( bind(sock,(struct sockaddr*)&local,sizeof(local)) == SOCKET_ERROR )
    {
        printf( "bind failed with:%d \n",WSAGetLastError());
        closesocket(sock);
```

```
        WSACleanup();
        return -1;
    }
    //加入多播组
    remote.sin_family = AF_INET;
    remote.sin_port = htons(MCASTPORT);
    remote.sin_addr.s_addr = inet_addr( MCASTADDR );
    /* Winsock1.0 */
    /*
    mcast.imr_multiaddr.s_addr = inet_addr(MCASTADDR);
    mcast.imr_interface.s_addr = INADDR_ANY;
    if( setsockopt(sockM,IPPROTO_IP,IP_ADD_MEMBERSHIP,(char*)&mcast,

      sizeof(mcast)) == SOCKET_ERROR)
      {
      printf("setsockopt(IP_ADD_MEMBERSHIP) failed:%d\n",WSAGetLastError());
      closesocket(sockM);
      WSACleanup();
      return -1;
      }
    */
    /* Winsock2.0*/
    if(( sockM = WSAJoinLeaf(sock,(SOCKADDR*)&remote,sizeof(remote),

        NULL,NULL,NULL,NULL,
        JL_BOTH)) == INVALID_SOCKET)
    {
        printf("WSAJoinLeaf() failed:%d\n",WSAGetLastError());
        closesocket(sock);
        WSACleanup();
        return -1;
    }
    //接收多播数据，当接收到的数据为"QUIT"时退出。
    while(1)
    {
        if(( ret = recvfrom(sock,recvbuf,BUFSIZE,0,

            (struct sockaddr*)&from,&len)) == SOCKET_ERROR)
        {
            printf("recvfrom failed with:%d\n",WSAGetLastError());
            closesocket(sockM);
            closesocket(sock);
            WSACleanup();
            return -1;
```

```
            }
            if( strcmp(recvbuf,"QUIT") == 0 ) break;
            else {
                recvbuf[ret] = '\0';
                printf("RECV:' %s ' FROM <%s> \n",recvbuf,inet_ntoa(from.sin_addr));
            }
        }

        closesocket(sockM);
        closesocket(sock);
        WSACleanup();
        return 0;
}
```

附录 6、raw socket 编程例子

1 原始套接字工作原理与规则

原始套接字是一种不同于 SOCK_STREAM 和 SOCK_DGRAM 的套接字，它实现于系统核心。它的创建方式跟 TCP/UDP 创建方法几乎是一模一样，例如，通过

int sockfd;

sockfd=socket(AF_INET，SOCK_RAW，IPPROTO_ICMP);

这两句程序你就可以创建一个原始套接字。这种类型套接字的功能与 TCP 或者 UDP 类型套接字的功能有很大的不同：TCP/UDP 类型的套接字只能够访问传输层以及传输层以上的数据，因为当 IP 层把数据传递给传输层时，下层的数据包头已经被丢掉了。而原始套接字却可以访问传输层以下的数据，所以使用 raw 套接字你可以实现上至应用层的数据操作，也可以实现下至链路层的数据操作。比如：通过

sock=socket(PF_PACKET，SOCK_RAW，htons(ETH_P_IP))

方式创建的 rawsocket 就能直接读取链路层的数据。

1)使用原始套接字时应该注意的问题(参考<<unix 网络编程>>以及网上的优秀文档)

(1)：对于 UDP/TCP 产生的 IP 数据包，内核不将它传递给任何原始套接字，而只是将这些数据交给对应的 UDP/TCP 数据处理句柄(所以，如果你想要通过原始套接字来访问 TCP/UDP 或者其它类型的数据，调用 socket 函数创建原始套接字第三个参数应该指定为 htons(ETH_P_IP)，也就是通过直接访问数据链路层来实现。(我们后面的密码窃取器就是基于这种类型的)。

(2)：对于 ICMP 和 EGP 等使用 IP 数据包承载数据但又在传输层之下的协议类型的 IP 数据包，内核不管是否已经有注册了的句柄来处理这些数据，都会将这些 IP 数据包复制一份传递给协议类型匹配的原始套接字。

(3)：对于不能识别协议类型的数据包，内核进行必要的校验，然后会查看是否有类型匹配的原始套接字负责处理这些数据，如果有的话，就会将这些 IP 数据包复制一份传递给匹配的原始套接字，否则，内核将会丢弃这个 IP 数据包，并返回一个 ICMP 主机不可达的消息给源主机。

(4)：如果原始套接字 bind 绑定了一个地址，核心只将目的地址为本机 IP 地址的数据包传递给原始套接字，如果某个原始套接字没有 bind 地址，核心就会把收到的所有 IP 数据包发给这个原始套接字。

(5)：如果原始套接字调用了 connect 函数，则核心只将源地址为 connect 连接的 IP 地址的 IP 数据包传递给这个原始套接字。

(6)：如果原始套接字没有调用 bind 和 connect 函数，则核心会将所有协议匹配的 IP 数据包传递给这个原始套接字。

2)编程选项

原始套接字是直接使用 IP 协议的非面向连接的套接字，在这个套接字上可以调用 bind 和 connect 函数进行地址绑定。说明如下：

(1)bind 函数：调用 bind 函数后，发送数据包的源 IP 地址将是 bind 函数指定的地址。如果不调用 bind，则内核将以发送接口的主 IP 地址填充 IP 头。如果使用 setsockopt 设置了 IP_HDRINCL(headerincluding)选项，就必须手工填充每个要发送的数据包的源 IP 地址，否则，内核将自动创建 IP 首部。

(2)connetc 函数：调用 connect 函数后，就可以使用 write 和 send 函数来发送数据包，而且内核将会用这个绑定的地址填充 IP 数据包的目的 IP 地址，否则的话，则应使用 sendto 或 sendmsg 函数来发送数据包，并且要在函数参数中指定对方的 IP 地址。

综合以上种种功能和特点，我们可以使用原始套接字来实现很多功能，比如最基本的数据包分析，

主机嗅探等。其实也可以使用原始套接字作一个自定义的传输层协议。

2 一个简单的应用

下面的代码创建一个直接读取链路层数据包的原始套接字，并从中分析出源 MAC 地址和目的 MAC 地址，源 IP 和目的 IP，以及对应的传输层协议，如果是 TCP/UDP 协议的话，打印其目的和源端口。为了方便阅读，程序中避免了使用任何与协议有关的数据结构，如 structether_header，structiphdr 等，当然，要完全理解代码，你需要关于指针以及位运算的知识。

```
/***************SimpelSniffer.c*************/
/*注意：本代码为 LINUX 操作系统下的源代码*/
//author:duanjigang@2006s
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <linux/if_ether.h>
#include <linux/in.h>
#define BUFFER_MAX 2048

int main(int argc, char *argv[])
{

    int sock, n_read, proto;
    char buffer[BUFFER_MAX];
    char    *ethhead, *iphead, *tcphead,
        *udphead, *icmphead, *p;

    if((sock = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_IP))) < 0)
    {
        fprintf(stdout, "create socket error\n");
        exit(0);
    }

    while(1)
    {
        n_read = recvfrom(sock, buffer, 2048, 0, NULL, NULL);
        /*
        14      6(dest)+6(source)+2(type or length)
        +
        20      ip header
        +
        8      icmp,tcp or udp header
        = 42
        */
        if(n_read < 42)
```

```
        {
            fprintf(stdout, "Incomplete header, packet corrupt\n");
            continue;
        }

        ethhead = buffer;
        p = ethhead;
        int n = 0XFF;
        printf("MAC: %.2X:%02X:%02X:%02X:%02X:%02X==>"
            "%.2X:%.2X:%.2X:%.2X:%.2X:%.2X\n",
            p[6]&n, p[7]&n, p[8]&n, p[9]&n, p[10]&n, p[11]&n,
            p[0]&n, p[1]&n, p[2]&n,p[3]&n, p[4]&n, p[5]&n);

        iphead = ethhead + 14;
        p = iphead + 12;

        printf("IP: %d.%d.%d.%d => %d.%d.%d.%d\n",
            p[0]&0XFF, p[1]&0XFF, p[2]&0XFF, p[3]&0XFF,
            p[4]&0XFF, p[5]&0XFF, p[6]&0XFF, p[7]&0XFF);
        proto = (iphead + 9)[0];
        p = iphead + 20;
        printf("Protocol: ");
        switch(proto)
        {
        case IPPROTO_ICMP: printf("ICMP\n");break;
        case IPPROTO_IGMP: printf("IGMP\n");break;
        case IPPROTO_IPIP: printf("IPIP\n");break;
        case IPPROTO_TCP :
        case IPPROTO_UDP :
            printf("%s,", proto == IPPROTO_TCP ? "TCP": "UDP");
            printf("source port: %u,",(p[0]<<8)&0XFF00 |    p[1]&0XFF);
            printf("dest port: %u\n", (p[2]<<8)&0XFF00 | p[3]&0XFF);
            break;
        case IPPROTO_RAW : printf("RAW\n");break;
        default:printf("Unkown, please query in include/linux/in.h\n");
        }
    }
}
```

附录 7、简单 Web Server 程序

实现一个简单的 Web Server，能够响应客户端的请求将指定目录下的 HTML 或 text 件通过指定的 TCP 端发送给客户端。具体编程要求是：

(1)服务器启动时可指定服务端口，默认为 8000。

(2)可指定 Web Server 的根目录。

(3)服务器应能够并发处理多个请求。要求至少能支持 Get 命令。鼓励增强 Web Server 的功能，如支持 Head、Post 以及 Delete 等命令。

(4)统计 Web Server 接收和发送的流量。

(5)在 Windows 平台实现，要求使用图形界面显示服务器的各种信息。

(6)不允许使用已有的 HTTP 库。

(7)编写必要的客户端测试程序用于发送 HTTP 请求并显示返回结果，也可使用一般的 Web 浏览器测试。

1 相关知识

Web 技术的独特之处是采用超链接和多媒体信息。Web 服务器使用超文本标记语言（HTML-HyperText Marked Language）描述网络的资源，创建网页，以供 Web 浏览器阅读。HTML 文档的特点是交互性。不管是一般文本还是图形，都能通过文挡中的链接连接到服务器上的其他文档，从而使客户快速地搜寻他们想要的资料。HTML 网页还可提供表单供用户填写并通过服务器应用程序提交给数据库。这种数据库一般是支持多媒体数据类型的。

Web 浏览器（Web Browser）是一个用于文档检索和显示的客户应用程序，并通过超文本传输协议 Http（Hyper Text Transfer Protocol）与 Web 服务器相连。通用的、低成本的浏览器节省了两层结构的 C/S 模式客户端软件的开发和维护费用。

1.1 HTTP 的工作过程

HTTP 协议工作流程：

1.首先客户机与服务器需要建立连接。只要单击某个超级链接，HTTP 的工作就开始了。

2.建立连接后，客户机发送一个请求给服务器，请求方式的格式为：统一资源标识符（URL）、协议版本号，后边是 MIME 信息：包括请求修饰符、客户机信息和可能的内容。

3.服务器接到请求后，给予相应的响应信息，其格式为一个状态行，包括信息的协议版本号、一个成功或错误的代码，后边是 MIME 信息包括服务器信息、实体信息和可能的内容。

4.客户端接收服务器所返回的信息通过浏览器显示在用户的显示屏上，然后客户机与服务器断开连接。

5.1.2 HTTP 支持的有关方法

标准的万维网传输协议是超文本传输协议 Http，每一次交互一个 ASCII 码请求，跟着一个与

RFC822 类似的 Mime 的应答组成。Http 协议由两个相当明显的项组成：从浏览器到服务器的请求集和从另一方向来的应答集，下面依次分析它们：

所有较新的 Http（Http/1.1）都支持两种请求：简单请求和完全请求。简单请求只是一行声明了所需页面的 get 行，而没有协议版本(get/default.htm)。应答仅是原始的页面，没有头部，没有 Mime，没有编码。表示完全请求的语句是 get 请求行出现协议版本(get/default.html Http/1.0)，请求可以有好几行组成，后跟一个空行标示请求结束，完全请求的第一行包括一个命令(get 仅是其中一个)，所需要的页面和协议/版本，接下来的行包含 RFC822 头部。下表为内置的 Http 方法：

内置的 Http 有关方法

方法	描 述
get	请求读一个万维网页
head	请求读一个万维网页的头
put	请求存储一个万维网页
post	附加一个命名的资源
delete	删除万维网页
link	连接两个已有资源
unlink	切断两个已有资源间的连接

(1) Get 方法请求服务器发送一页面（更一般的情况指一个对象），页面以 Mime 合适的编码。如果 get 请求后面跟有一个 If-Modified-Since 头部，则仅当数据在提供的日期以后被修改过时服务器才发送它。使用这一机制，当要求浏览器显示一缓存的页面时，能带有条件的向服务器请求这一页，并给出与这一页相关的修改时间。如果缓存的页面还有效，服务器仅发送回一状态行宣布这一事实，这样便消除了重传一页的开销。

(2) head 方法仅要求消息的头部，而非真实的一页。这一方法可用来得到该页上的一次修改时间，以便为索引的目的收集或测试 URL 的有效性。没有带条件的 head 请求。

(3) put 方法与 get 相反：不是读取一页，而是写入一页。这一方法使得建立位于远程服务器上的万维网页面的集合成为可能，请求体中包含这一页面，它可以用 MIME 编码。这种情况下，put 后的一行可能包含 ContentType 和鉴别头部，以证明调用者的确允许执行请求操作。

(4) 与 put 相似的是 post 方法，它也带有 url，但不是替换已存在的数据，通常是将新的数据附加在它的后面，象滚动新闻和公告板等。

(5) delete 所做的是删除页面。如同 put，鉴别和许可在这是重要的角色。 delete 的成功与否并没有保证，因为即使远端的 Http 服务器想要删除页面，但该文件可能有一模式禁止 Http 服务器修改或删除它。

(6) link 和 unlink 方法允许在已存在的页面或其它资源之间建立链接。每一个请求得到一个由

状态行和可能的附加消息（如：全部或部分页面）组成的应答。状态行可能包含代码 200(ok)，或任一个各种各样的错误代码，如：304 Not Modified,400 Bad Request,401 Unauthorized，403 Forbidden 等。

2 Web 服务器/浏览器如何实现通信

2.1 分析请求

(1) Http 请求的格式参考教材 255 页图 6-12(a)。在 Http 请求中，第一行必须是一个请求行（request line），用来说明请求类型、要访问的资源以及使用的 Http 版本。紧接着是一个首部（header）小节，用来说明服务器要使用的附加信息。在首部之后是一个空行，再此之后可以添加任意的其他数据。

只要在 Web 浏览器上输入一个 url，浏览器就将基于该 url 向服务器发送一个 get 请求，以告诉服务器获取并返回什么资源。对于 www.hnust.edu.cn 的 get 请求如下所示：

get / Http/1.1

Host: www.hnust.edu.cn

User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.6)

Gecko/20050225 Firefox/1.0.1

Connection: Keep-Alive

请求行的第一部分说明了该请求是 get 请求。该行的第二部分是一个斜杠（/），用来说明请求的是该域名的根目录。该行的最后一部分说明使用的是 Http 1.1 版本（另一个可选项是 1.0）。那么请求发到哪里去呢？这就是第二行的内容。第 2 行是请求的第一个首部 host。首部 host 将指出请求的目的地。结合 HOST 和上一行中的斜杠（/），可以通知服务器请求的是 www.hnust.edu.cn/（Http 1.1 才需要使用首部 host，而原来的 1.0 版本则不需要使用）。第三行中包含的是首部 User-Agent，服务器端和客户端脚本都能够访问它，它是浏览器类型检测逻辑的重要基础。该信息由你使用的浏览器来定义，并且在每个请求中将自动发送。最后一行是首部 Connection，通常将浏览器操作设置为 Keep-Alive（当然也可以设置为其他值）。注意，在最后一个首部之后有一个空行。即使不存在请求主体，这个空行也是必需的。

2.2 构造响应

Http 响应的格式参考教材 255 页图 6-12(b)，它与请求的格式十分类似。在响应中唯一真正的区别在于第一行中用状态信息代替了请求信息。状态行（status line）通过提供一个状态码来说明所请求的资源情况。以下就是一个 Http 响应的例子：

Http/1.1 200 OK

Date: Sat, 31 Dec 2005 23:59:59 GMT

Content-Type: text/html;charset=ISO-8859-1

Content-Length: 122

在本例中，状态行给出的 Http 状态代码是 200，以及消息 ok。状态行始终包含的是状态码和相应的简短消息，以避免混乱。下表为最常用的状态码：

Http 常用状态码

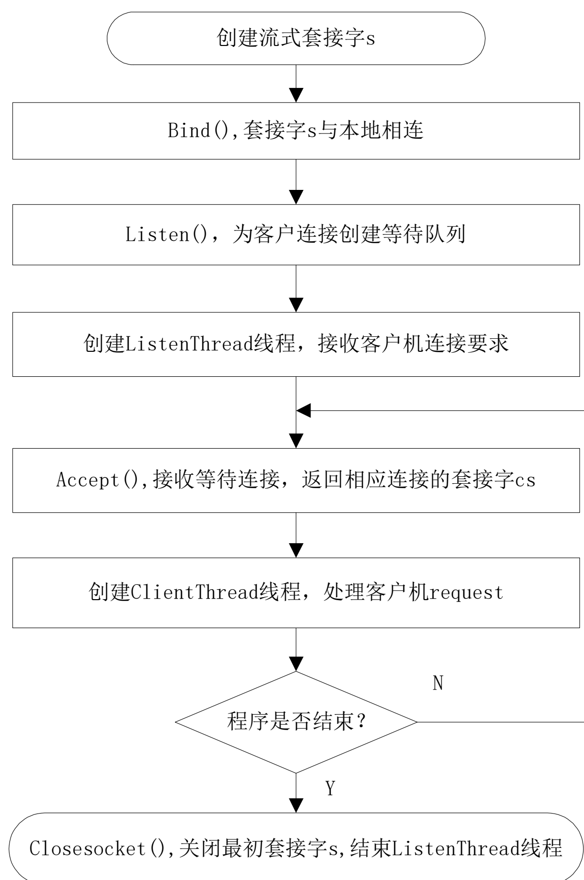
200(ok)	找到了该资源，并且一切正常。
304(not modified)	该资源在上次请求之后没有任何修改。这通常用于浏览器的缓存机制。
401(unauthorized)	客户端无权访问该资源。这通常会使得浏览器要求用户输入用户名和密码，以登录到服务器。
403 (forbidden)	客户端未能获得授权。这通常是在 401 之后输入了不正确的用户名或密码。
404 (not found)	在指定的位置不存在所申请的资源。

在状态行之后是一些首部。通常，服务器会返回一个名为 Date 的首部，用来说明响应生成的日期和时间（服务器通常还会返回一些关于其自身的信息，尽管并非是必需的）。接下来的两个首部大家应该熟悉，就是与 post 请求中一样的 Content-Type 和 Content-Length。在本例中，首部 Content-Type 指定了 Mime 类型 Html（text/html），其编码类型是 ISO-8859-1（这是针对美国英语资源的编码标准）。响应主体所包含的就是所请求资源的 Html 源文件。浏览器将把这些数据显示给用户。注意，这里并没有指明针对该响应的请求类型，不过这对于服务器并不重要。客户端知道每种类型的请求将返回什么类型的数据，并决定如何使用这些数据。

3 程序设计和实现

Web Server 可以分为两个组成模块：客户请求处理模块和响应生成发送模块。其中客户请求处理模块的任务就是负责接收客户的连接，它监听系统的端口，以获取客户机到达本服务器的连接，在程序中线程 ListenThread 实现该部分功能。当获得一个连接请求时，就为该连接建立一个客户请求处理线程来处理这个请求，求该处理线程就是响应生成发送模块负责分析请求中的各个协议参数，根据客户请求的分析结果查找资源，生成响应和发送响应，在程序中线程 ClientThread 实现该部分功能。

程序的主要流程如下图所示。



程序流程图

下面分别介绍客户请求处理模块和响应生成发送模块的代码实现。

1. 客户请求处理模块实现

在接收客户请求之前，首先应该创建套接字，在某一特定端口侦听客户端的请求。实现该部分功能的代码如下：

```
bool CHttpProtocol::StartHttpSrv()
{
    WORD wVersionRequested = WINSOCK_VERSION;
    WSADATA wsaData;
    int nRet;
    // 启动 Winsock
    nRet = WSAStartup(wVersionRequested, &wsaData);        // 加载成功返回 0
    if (nRet)
    {
        // 错误处理
        AfxMessageBox("Initialize WinSock Failed");
```

```
        return false;
    }
    // 检测版本
    if (wsaData.wVersion != wVersionRequested)
    {
        // 错误处理
        AfxMessageBox("Wrong WinSock Version");
        return false;
    }
    m_hExit = CreateEvent(NULL, TRUE, FALSE, NULL);
    if (m_hExit == NULL)
    {
        return false;
    }
    //创建套接字
    m_listenSocket = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, NULL, 0,
WSA_FLAG_OVERLAPPED);
    if (m_listenSocket == INVALID_SOCKET)
    {
        // 异常处理
        CString *pStr = new CString;
        *pStr = "Could not create listen socket";
        SendMessage(m_hwndDlg, LOG_MSG, (UINT)pStr, NULL);
        return false;
    }
    SOCKADDR_IN sockAddr;
    LPSERVENT  lpServEnt;
    if (m_nPort != 0)
    {
        // 从主机字节顺序转为网络字节顺序赋给 sin_port
        sockAddr.sin_port = htons(m_nPort);
    }
    else
    {
        // 获取已知 http 服务的端口，该服务在 tcp 协议下注册
```

```
        lpServEnt = getservbyname("http", "tcp");
        if (lpServEnt != NULL)
        {
            sockAddr.sin_port = lpServEnt->s_port;
        }
        else
        {
            sockAddr.sin_port = htons(HTTPPORT);    // 默认端口 HTTPPORT＝80
        }
    }
    sockAddr.sin_family = AF_INET;
    sockAddr.sin_addr.s_addr = INADDR_ANY;   // 指定端口号上面的默认 IP 接口
    // 初始化 content-type 和文件后缀对应关系的 map
    CreateTypeMap();

    // 套接字绑定
    nRet = bind(m_listenSocket, (LPSOCKADDR)&sockAddr, sizeof(struct sockaddr));
    if (nRet == SOCKET_ERROR)
    {
        // 绑定发生错误
        CString *pStr = new CString;
        *pStr = "bind() error";
        SendMessage(m_hwndDlg, LOG_MSG, (UINT)pStr, NULL);
        closesocket(m_listenSocket); // 断开链接
        return false;
    }
    // 套接字监听。为客户连接创建等待队列,队列最大长度 SOMAXCONN 在 windows sockets
头文件中定义
    nRet = listen(m_listenSocket, SOMAXCONN);
    if (nRet == SOCKET_ERROR)
    {
        // 异常处理
        CString *pStr = new CString;
        *pStr = "listen() error";
        SendMessage(m_hwndDlg, LOG_MSG, (UINT)pStr, NULL);
```

```
        closesocket(m_listenSocket); // 断开链接
        return false;
    }
    // 创建 listening 进程，接受客户机连接要求
    m_pListenThread = AfxBeginThread(ListenThread, this);
    if (!m_pListenThread)
    {
        // 线程创建失败
        CString *pStr = new CString;
        *pStr = "Could not create listening thread" ;
        SendMessage(m_hwndDlg, LOG_MSG, (UINT)pStr, NULL);
        closesocket(m_listenSocket); // 断开链接
        return false;
    }
    CString strIP, strTemp;
    char hostname[255];
    PHOSTENT hostinfo;
    // 获取计算机名
    if(gethostname(hostname, sizeof(hostname))==0) // 得到主机名
    {
        // 由主机名得到主机的其他信息
        hostinfo = gethostbyname(hostname);
        if(hostinfo != NULL)
        {
            strIP = inet_ntoa(*(struct in_addr*)*(hostinfo->h_addr_list));
        }
    }
    // 显示 web 服务器正在启动
    CString *pStr = new CString;
    *pStr = "****** My WebServer is Starting now! *******";
    SendMessage(m_hwndDlg, LOG_MSG, (UINT)pStr, NULL);
    // 显示 web 服务器的信息，包括主机名，IP 以及端口号
    CString *pStr1 = new CString;
    pStr1->Format("%s", hostname);
    *pStr1 = *pStr1 + "[" + strIP + "]" + "      Port ";
```

```
	strTemp.Format("%d", htons(sockAddr.sin_port));
	*pStr1 = *pStr1 + strTemp;
	SendMessage(m_hwndDlg, LOG_MSG, (UINT)pStr1, NULL);
	return true;
}
// 连接的 Client 的信息
typedef struct REQUEST
{
	HANDLE		hExit;
	SOCKET		Socket;			// 请求的 socket
	int		nMethod;		// 请求的使用方法：GET 或 HEAD
	DWORD		dwRecv;			// 收到的字节数
	DWORD		dwSend;			// 发送的字节数
	HANDLE		hFile;			// 请求连接的文件
	char		szFileName[_MAX_PATH]; // 文件的相对路径
	char		postfix[10];		// 存储扩展名
	char		StatuCodeReason[100];	// 头部的 status cod 以及 reason-phrase
	bool		permitted;		// 用户权限判断
	char *		authority;		// 用户提供的认证信息
	char		key[1024];		// 正确认证信息
	void* pHttpProtocol;			// 指向类 CHttpProtocol 的指针
}REQUEST, *PREQUEST;

UINT CHttpProtocol::ListenThread(LPVOID param)
{
	CHttpProtocol *pHttpProtocol = (CHttpProtocol *)param;
	SOCKET		socketClient;
	CWinThread* pClientThread;
	SOCKADDR_IN	SockAddr;
	PREQUEST	pReq;
	int		nLen;
	DWORD		dwRet;
	// 初始化 ClientNum，创建"no client"事件对象
	HANDLE		hNoClients;
	hNoClients = pHttpProtocol->InitClientCount();
```

```
while(1)  // 循环等待,如有客户连接请求,则接受客户机连接要求
{
    nLen = sizeof(SOCKADDR_IN);
    // 套接字等待链接,返回对应已接受的客户机连接的套接字
    socketClient   =   accept(pHttpProtocol->m_listenSocket,   (LPSOCKADDR)&SockAddr, 
&nLen);
    if (socketClient == INVALID_SOCKET)
    {
        break;
    }
    // 将客户端网络地址转换为用点分割的 IP 地址
    CString *pstr = new CString;
    pstr->Format("%s Connecting on socket:%d", inet_ntoa(SockAddr.sin_addr), socketClient);
    SendMessage(pHttpProtocol->m_hwndDlg, LOG_MSG, (UINT)pstr, NULL);
    pReq = new REQUEST;
    if (pReq == NULL)
    {
        // 处理错误
        CString *pStr = new CString;
        *pStr = "No memory for request";
        SendMessage(pHttpProtocol->m_hwndDlg, LOG_MSG, (UINT)pStr, NULL);
        continue;
    }
    pReq->hExit   = pHttpProtocol->m_hExit;
    pReq->Socket = socketClient;
    pReq->hFile = INVALID_HANDLE_VALUE;
    pReq->dwRecv = 0;
    pReq->dwSend = 0;
    pReq->pHttpProtocol = pHttpProtocol;
    // 创建 client 进程，处理 request
    pClientThread = AfxBeginThread(ClientThread, pReq);
    if (!pClientThread)
    {
        // 线程创建失败,错误处理
```

```
                CString *pStr = new CString;
                *pStr = "Couldn't start client thread";
                SendMessage(pHttpProtocol->m_hwndDlg, LOG_MSG, (UINT)pStr, NULL);
                delete pReq;
            }
        } //while
        // 等待线程结束
        WaitForSingleObject((HANDLE)pHttpProtocol->m_hExit, INFINITE);
        // 等待所有 client 进程结束
        dwRet = WaitForSingleObject(hNoClients, 5000);
        if (dwRet == WAIT_TIMEOUT)
        {
            // 超时返回，并且同步对象未退出
            CString *pStr = new CString;
            *pStr = "One or more client threads did not exit";
            SendMessage(pHttpProtocol->m_hwndDlg, LOG_MSG, (UINT)pStr, NULL);
        }
        pHttpProtocol->DeleteClientCount();
        return 0;
    }
```

2、 响应生成发送模块实现

ClientThread 负责分析客户请求中的各个协议参数，根据对客户请求的分析结果查找资源，生成响应和发送响应。

```
UINT CHttpProtocol::ClientThread(LPVOID param)
{
    int nRet;
    BYTE buf[1024];
    PREQUEST pReq = (PREQUEST)param;
    CHttpProtocol *pHttpProtocol = (CHttpProtocol *)pReq->pHttpProtocol;
    pHttpProtocol->CountUp();            // 记数
    // 接收 request data
     if (!pHttpProtocol->RecvRequest(pReq, buf, sizeof(buf)))
    {
        pHttpProtocol->Disconnect(pReq);
```

```
        delete pReq;
        pHttpProtocol->CountDown();
        return 0;
    }
    // 分析 request 信息
    nRet = pHttpProtocol->Analyze(pReq, buf);
    if (nRet)
    {
        // 处理错误
        CString *pStr = new CString;
        *pStr = "Error occurs when analyzing client request";
        SendMessage(pHttpProtocol->m_hwndDlg, LOG_MSG, (UINT)pStr, NULL);
        pHttpProtocol->Disconnect(pReq);
        delete pReq;
        pHttpProtocol->CountDown();
        return 0;
    }
    // 生成并返回头部
    pHttpProtocol->SendHeader(pReq);
    // 向 client 传送数据
    if(pReq->nMethod == METHOD_GET)
    {
        pHttpProtocol->SendFile(pReq);
    }
    pHttpProtocol->Disconnect(pReq);
    delete pReq;
    pHttpProtocol->CountDown();     // client 数量减 1
    return 0;
}

// 分析 request 信息
int  CHttpProtocol::Analyze(PREQUEST pReq, LPBYTE pBuf)
{
    // 分析接收到的信息
    char szSeps[] = " \n";
```

```
char *cpToken;
// 防止非法请求
if (strstr((const char *)pBuf, "..") != NULL)
{
    strcpy(pReq->StatuCodeReason, HTTP_STATUS_BADREQUEST);
    return 1;
}
// 判断 ruquest 的 mothed
cpToken = strtok((char *)pBuf, szSeps);      // 缓存中字符串分解为一组标记串。
if (!_stricmp(cpToken, "GET"))               // GET 命令
{
    pReq->nMethod = METHOD_GET;
}
else if (!_stricmp(cpToken, "HEAD"))  // HEAD 命令
{
    pReq->nMethod = METHOD_HEAD;
}
else
{
    strcpy(pReq->StatuCodeReason, HTTP_STATUS_NOTIMPLEMENTED);
    return 1;
}
// 获取 Request-URI
cpToken = strtok(NULL, szSeps);
if (cpToken == NULL)
{
    strcpy(pReq->StatuCodeReason, HTTP_STATUS_BADREQUEST);
    return 1;
}
strcpy(pReq->szFileName, m_strRootDir);
if (strlen(cpToken) > 1)
{
    strcat(pReq->szFileName, cpToken);   // 把该文件名添加到结尾处形成路径
}
else
```

```
        {
            strcat(pReq->szFileName, "/index.html");
        }
        return 0;
    }

    // 发送头部
    void CHttpProtocol::SendHeader(PREQUEST pReq)
    {
        int n = FileExist(pReq);
        if(!n)          // 文件不存在，则返回
        {
            return;
        }
        char Header[2048] = " ";
        char curTime[50] = " ";
        GetCurentTime((char*)curTime);
        // 取得文件长度
        DWORD length;
        length = GetFileSize(pReq->hFile, NULL);
        // 取得文件的 last-modified 时间
        char last_modified[60] = " ";
        GetLastModified(pReq->hFile, (char*)last_modified);
        // 取得文件的类型
        char ContenType[50] = " ";
        GetContenType(pReq, (char*)ContenType);
        sprintf((char*)Header,
"HTTP/1.0 %s\r\nDate: %s\r\nServer: %s\r\nContent-Type: %s\r\nContent-Length: %d\r\nLast-Modified: %s
\r\n\r\n",
                    HTTP_STATUS_OK,
                    curTime,              // Date
                    "My Http Server",     // Server
                    ContenType,           // Content-Type
                    length,               // Content-length
                    last_modified);       // Last-Modified
```

```
        // 发送头部
        send(pReq->Socket, Header, strlen(Header), 0);
}

// 发送文件
void CHttpProtocol::SendFile(PREQUEST pReq)
{
        int n = FileExist(pReq);
        if(!n)                  // 文件不存在，则返回
        {
                return;
        }
        CString *pStr = new CString;
        *pStr = *pStr + &pReq->szFileName[strlen(m_strRootDir)];
        SendMessage(m_hwndDlg, LOG_MSG, UINT(pStr), NULL);
        static BYTE buf[2048];
        DWORD   dwRead;
        BOOL    fRet;
        int flag = 1;
        // 读写数据直到完成
        while(1)
        {
                // 从 file 中读入到 buffer 中
                fRet = ReadFile(pReq->hFile, buf, sizeof(buf), &dwRead, NULL);
                if (!fRet)
                {
                        static char szMsg[512];
                        wsprintf(szMsg, "%s", HTTP_STATUS_SERVERERROR);
                        // 向客户端发送出错信息
                        send(pReq->Socket, szMsg, strlen(szMsg), 0);
                        break;
                }
                // 完成
                if (dwRead == 0)
                {
```

```
                break;
            }
            // 将 buffer 内容传送给 client
            if (!SendBuffer(pReq, buf, dwRead))
            {
                break;
            }
            pReq->dwSend += dwRead;
        }
        // 关闭文件
        if (CloseHandle(pReq->hFile))
        {
            pReq->hFile = INVALID_HANDLE_VALUE;
        }
        else
        {
            CString *pStr = new CString;
            *pStr = "Error occurs when closing file";
            SendMessage(m_hwndDlg, LOG_MSG, (UINT)pStr, NULL);
        }
    }
```

湖南科技大学计算机科学与工程学院

________________课程设计报告

专业班级：______________________________

姓　　名：______________________________

学　　号：______________________________

指导教师：______________________________

时　　间：______________________________

地　　点：______________________________

指导教师评语：

成绩：　　　　　等级：

签名：________________

年　　月　　日

一、实验题目

二、实验目的

三、总体设计（含背景知识或基本原理与算法、或模块介绍、设计步骤等）

四、详细设计（含主要的数据结构、程序流程图、关键代码等）

五、实验结果与分析

六、小结与心得体会

湖南科技大学潇湘学院计算机科学与工程系

________________课程设计报告

专业班级：______________________________

姓　　名：______________________________

学　　号：______________________________

指导教师：______________________________

时　　间：______________________________

地　　点：______________________________

指导教师评语：

成绩：　　　　等级：

签名：____________

年　　月　　日

一、实验题目

二、实验目的

三、总体设计（含背景知识或基本原理与算法、或模块介绍、设计步骤等）

四、详细设计（含主要的数据结构、程序流程图、关键代码等）

五、实验结果与分析

六、小结与心得体会