一、目的

在分析 ping 命令实现实例代码基础上，理解该命令的实现原理；通过构造 ICMP ECHO 请求报文，在目标计算机上对 ICMP ECHO 请求报文进行接收和解析，深刻理解 ICMP 协议的工作原理。

通过设计和实现一个 FTP 客户端系统，深刻理解 FTP 协议工作原理，重点掌握协议与现实中控制连接和数据连接建立过程，两个连接通信模式特点，以及多进程通信编程方法。

二、要求

实验内容 1：

1）分析 ping 命令实现的实例代码。

2）在发送端构造 ICMP ECHO 请求报文并发送给接收端。

3）在接收端接收 ICMP ECHO 请求报文并解析与显示首部各个字段的值。

实验内容 2：

1）设计和实现一个 FTP 客户端系统

2）在设计实现过程中深刻理解 FTP 协议工作原理

3）通过对模型的建立，重点掌握协议与显示中控制连接和数据的建立过程。

4）了解两个连接通信模式特点（主动模式和被动模式）

三、相关知识

实验内容 1：

ping 用于确定本地主机是否能与另一台主机成功交换(发送与接收)数据包，再根据返回的信息，就可以推断 TCP/IP 参数是否设置正确，以及运行是否正常、网络是否通畅等。

设计原理：网络上的机器都有唯一确定的 IP 地址，我们给目标 IP 地址发送一个数据包，对方就要返回一个同样大小的数据包，根据返回的数据包我们可以确定目标主机的存在，可以初步判断目标主机的操作系统等。

PING 命令：

PING （Packet Internet Groper），因特网包探索器，用于测试网络连接量的程序 。Ping 是工作在 TCP/IP 网络体系结构中应用层的一个服务命令， 主要是向特定的目的主机发送 ICMP（Internet Control Message Protocol 因特网报文控制协议）Echo 请求报文，测试目的站是否可达及了解其有关状态.

ICMP 协议：

ICMP（Internet Control Message Protocol）Internet 控制报文协议。它是 TCP/IP 协议簇的一个子协议，用于在 IP 主机、路由器之间传递控制消息。控制消息是指网络通不通、主机是否可达、路由是否可用等网络本身的消息。这些控制消息虽然并不传输用户数据，但是对于用户数据的传递起着重要的作用。

实验内容 2：

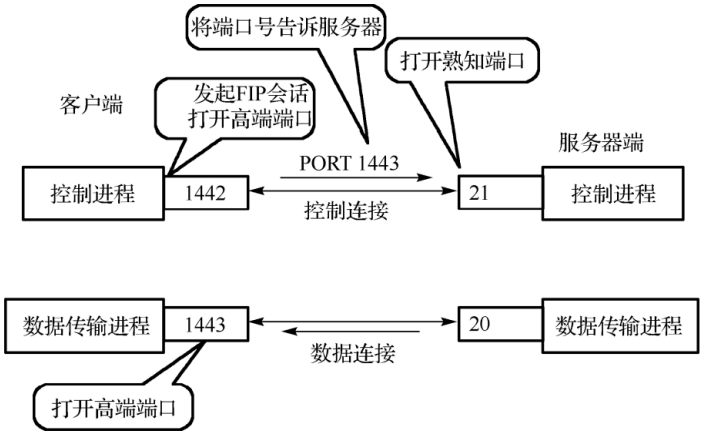
FTP 协议

该协议是 Internet 文件传送的基础，它由一系列规格说明文档组成，目标是提高文件的共享性，提供非直接使用远程计算机，使存储介质对用户透明和可靠高效地传送数据。简单的说，FTP 就是完成两台计算机之间的拷贝，从远程计算机拷贝文件至自己的计算机上，称之为“下载 （download）”文件。若将文件从自己计算机中拷贝至远程计算机上，则称之为“上载（upload）”文件。在 TCP/IP 协议中，FTP 标准命令 TCP 端口号为 21，Port 方式数据端口为 20。

FTP 工作模式：

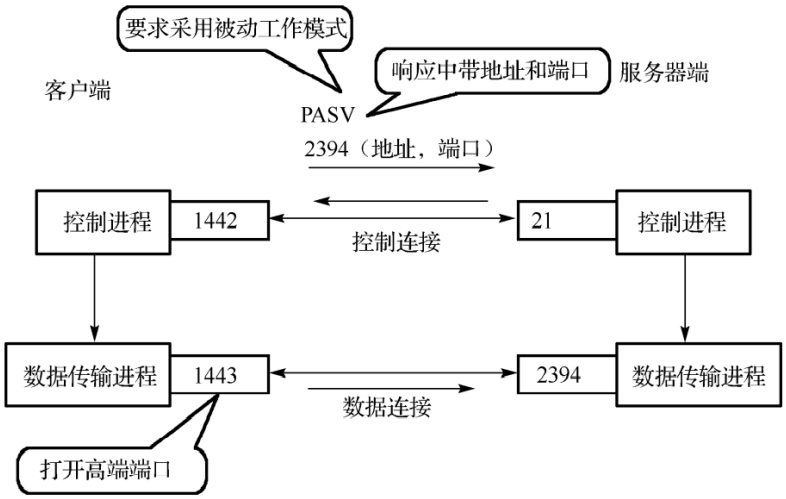
- **Standard 模式（主动模式）**

  FTP 客户端首先和 FTP Server 的 TCP 21 端口建立连接，通过这个通道发送命令，客户端需要接收数据的时候在这个通道上发送 PORT 命令。 PORT 命令包含了客户端用什么端口接收数据。在传送数据的时候，服务器端通过自己的 TCP 20 端口发送数据。 FTP server 必须和客户端建立一个新的连接用来传送数据。



- **Passive 模式（被动模式）**

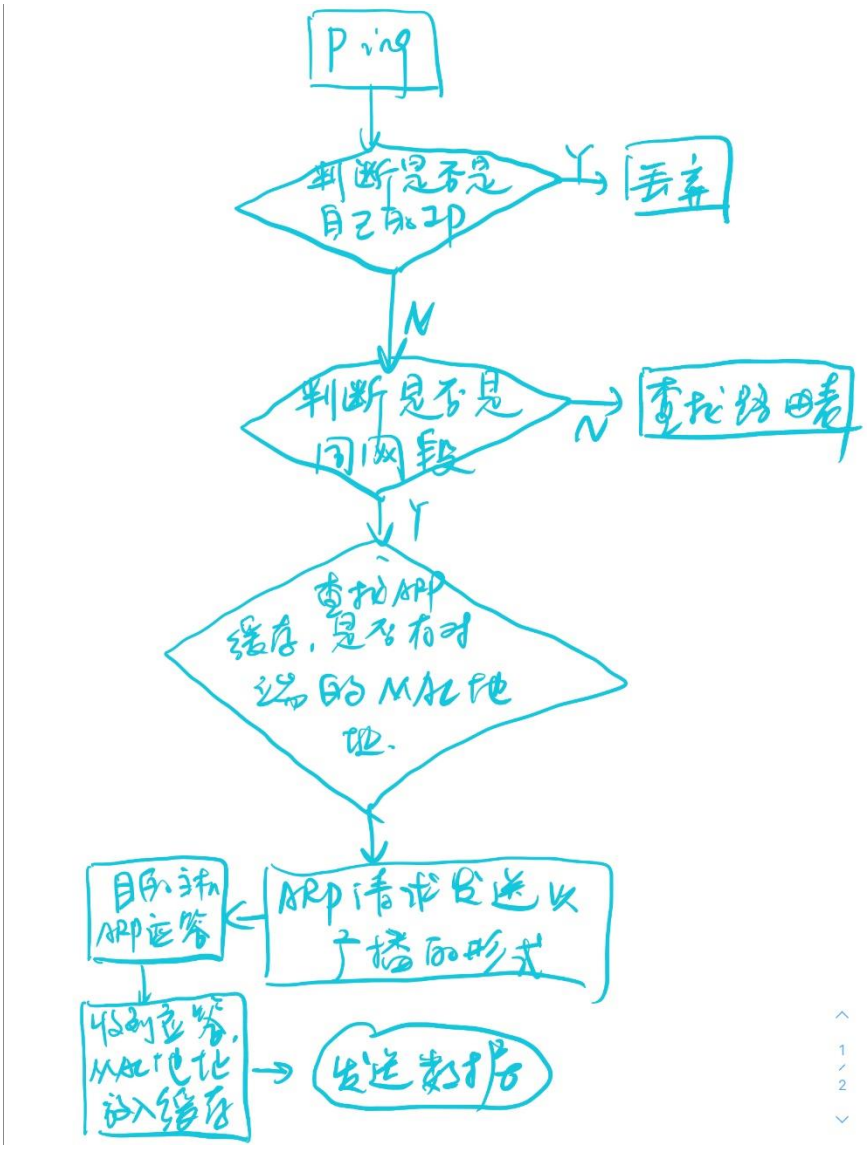
  在建立控制通道的时候和 Standard 模式类似，当客户端通过这个通道发送 PASV 命令的时候，FTP server 打开一个位于 1024 和 5000 之间的随机端口并且通知客户端在这个端口上传送数据的请求，然后 FTP server 将通过这个端口进行数据的传送，这个时候 FTP server 不再需要建立一个新的和客户端之间的连接。



四、实现原理及流程图

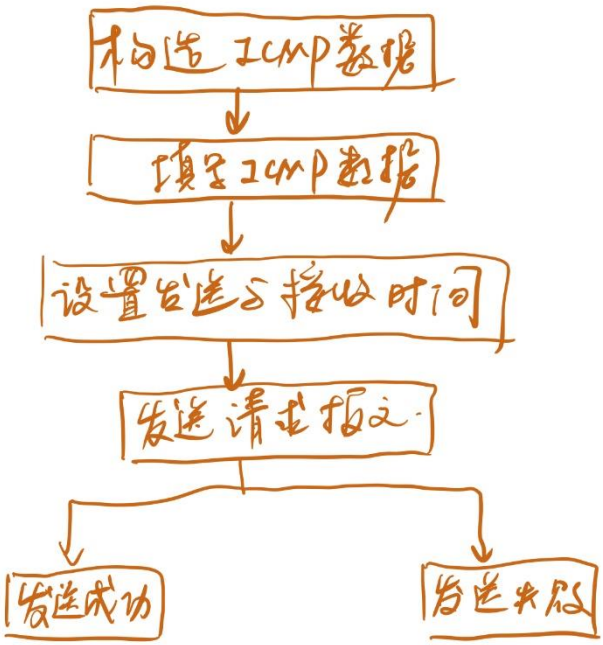
实验内容 1：

工作流程实例图：



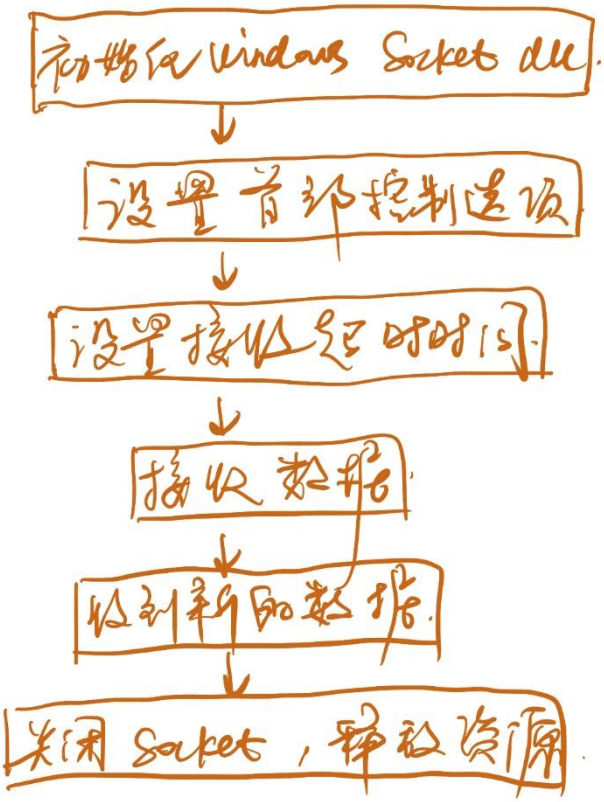
ICMP ECHO 请求报文发送端的构造和发送流程

构造：前 32bits 为 8bits 类型字段、8bits 代码字段、16bits 校验和字段



ICMP ECHO 请求报文接收端的接收和解析流程

构造：前 8bits 是发送序号，用于获得端口号



实验内容 2：

FTP 的设计和工作流程：

1) 首先 FTP 客户端与服务器之间建立控制连接。

2) FTP 客户端通过控制连接向服务器发送账号信息（用户名+密码），进行身份认证。

3) FTP 客户端通过控制连接向服务器发送 passiv 命令，说明采用被动模式建立数据连接。

4) FTP 客户端与服务器之间的被动模式建立数据连接。

5). FTP 客户端向服务器发送 dir 命令，服务器对该命令进行处理，并向客户端发送处理结果。

6) FTP 客户端接收服务器发送过来的处理结果（获得服务器当前目录下的列表信息），并在屏幕上显示。

7) 释放数据连接。

8) FTP 客户端向服务器发送 quit 命令，并释放控制连接。

9) 通信结束。

五、程序代码

实验内容 1：

```
#include <stdio.h>
#include <winsock2.h>
#pragma comment(lib, "ws2_32.lib")
#define ICMP_TYPE_ECHO 8
#define ICMP_TYPE_ECHO_REPLY 0
#define ICMP_MIN_LEN 8
#define ICMP_DEF_COUNT 4
#define ICMP_DEF_SIZE 32
#define ICMP_DEF_TIMEOUT 1000
#define ICMP_MAX_SIZE 65500
struct ip_hdr
{
    unsigned char vers_len;
    unsigned char tos;
    unsigned short total_len;
    unsigned short id;
    unsigned short frag;
    unsigned char ttl;
    unsigned char proto;
    unsigned short checksum;
    unsigned int sour;
    unsigned int dest;
};
struct icmp_hdr
{
    unsigned char type;
    unsigned char code;
    unsigned short checksum;
    unsigned short id;
    unsigned short seq;
```

```
    unsigned long timestamp;
};
struct icmp_user_opt
{
    unsigned int persist;
    unsigned int count;
    unsigned int size;
    unsigned int timeout;
    char *host;
    unsigned int send;
    unsigned int recv;
    unsigned int min_t;
    unsigned int max_t;
    unsigned int total_t;
};
const char icmp_rand_data[] = "abcdefghigklmnopqrstuvwxyz0123456789"
                              "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
struct icmp_user_opt user_opt_g = {
    0, ICMP_DEF_COUNT, ICMP_DEF_SIZE, ICMP_DEF_TIMEOUT, NULL,
    0, 0, 0xFFFF, 0};
unsigned short ip_checksum(unsigned short *buf, int buf_len);
void icmp_make_data(char *icmp_data, int data_size, int sequence)
{
    struct icmp_hdr *icmp_hdr;
    char *data_buf;
    int data_len;
    int fill_count = sizeof(icmp_rand_data) / sizeof(icmp_rand_data[0]);
    data_buf = icmp_data + sizeof(struct icmp_hdr);
    data_len = data_size - sizeof(struct icmp_hdr);
    while (data_len > fill_count)
    {
        memcpy(data_buf, icmp_rand_data, fill_count);
        data_len -= fill_count;
    }
    if (data_len > 0)
        memcpy(data_buf, icmp_rand_data, data_len);
    icmp_hdr = (struct icmp_hdr *)icmp_data;
    icmp_hdr->type = ICMP_TYPE_ECHO;
    icmp_hdr->code = 0;
    icmp_hdr->id = (unsigned short)GetCurrentProcessId();
    icmp_hdr->checksum = 0;
    icmp_hdr->seq = sequence;
    icmp_hdr->timestamp = GetTickCount();
    icmp_hdr->checksum = ip_checksum((unsigned short *)icmp_data, data_size);
```

```
}
int icmp_parse_reply(char *buf, int buf_len, struct sockaddr_in *from)
{
    struct ip_hdr *ip_hdr;
    struct icmp_hdr *icmp_hdr;
    unsigned short hdr_len;
    int icmp_len;
    unsigned long trip_t;
    ip_hdr = (struct ip_hdr *)buf;
    hdr_len = (ip_hdr->vers_len & 0xf) << 2;
    if (buf_len < hdr_len + ICMP_MIN_LEN)
    {
        printf("[Ping] Too few bytes from %s\n", inet_ntoa(from->sin_addr));
        return -1;
    }
    icmp_hdr = (struct icmp_hdr *)(buf + hdr_len);
    icmp_len = ntohs(ip_hdr->total_len) - hdr_len;
    if (ip_checksum((unsigned short *)icmp_hdr, icmp_len))
    {
        printf("[Ping] icmp checksum error!\n");
        return -1;
    }
    if (icmp_hdr->type != ICMP_TYPE_ECHO_REPLY)
    {
        printf("[Ping] not echo reply : %d\n", icmp_hdr->type);
        return -1;
    }
    if (icmp_hdr->id != (unsigned short)GetCurrentProcessId())
    {
        printf("[Ping] someone else's message!\n");
        return -1;
    }
    trip_t = GetTickCount() - icmp_hdr->timestamp;
    buf_len = ntohs(ip_hdr->total_len) - hdr_len - ICMP_MIN_LEN;
    printf("%d bytes from %s:", buf_len, inet_ntoa(from->sin_addr));
    printf(" icmp_seq = %d  time: %d ms\n", icmp_hdr->seq, trip_t);
    user_opt_g.recv++;
    user_opt_g.total_t += trip_t;
    if (user_opt_g.min_t > trip_t)
        user_opt_g.min_t = trip_t;
    if (user_opt_g.max_t < trip_t)
        user_opt_g.max_t = trip_t;
    return 0;
}
```

```
int icmp_process_reply(SOCKET icmp_soc)
{
    struct sockaddr_in from_addr;
    int result, data_size = user_opt_g.size;
    int from_len = sizeof(from_addr);
    char *recv_buf;
    data_size += sizeof(struct ip_hdr) + sizeof(struct icmp_hdr);
    recv_buf = malloc(data_size);
    result = recvfrom(icmp_soc, recv_buf, data_size, 0,
                      (struct sockaddr *)&from_addr, &from_len);
    if (result == SOCKET_ERROR)
    {
        if (WSAGetLastError() == WSAETIMEDOUT)
            printf("timed out\n");
        else
            printf("[PING] recvfrom_ failed: %d\n", WSAGetLastError());
        return -1;
    }
    result = icmp_parse_reply(recv_buf, result, &from_addr);
    free(recv_buf);
    return result;
}
void icmp_help(char *prog_name)
{
    char *file_name;
    file_name = strrchr(prog_name, '\\');
    if (file_name != NULL)
        file_name++;
    else
        file_name = prog_name;
    printf(" usage:     %s host_address [-t] [-n count] [-l size] "
           "[-w timeout]\n",
           file_name);
    printf(" -t         Ping the host until stopped.\n");
    printf(" -n count   the count to send ECHO\n");
    printf(" -l size    the size to send data\n");
    printf(" -w timeout timeout to wait the reply\n");
    exit(1);
}
void icmp_parse_param(int argc, char **argv)
{
    int i;
    for (i = 1; i < argc; i++)
    {
```

```
        if ((argv[i][0] != '-') && (argv[i][0] != '/'))
        {
            if (user_opt_g.host)
                icmp_help(argv[0]);
            else
            {
                user_opt_g.host = argv[i];
                continue;
            }
        }
        switch (tolower(argv[i][1]))
        {
        case 't':
            user_opt_g.persist = 1;
            break;
        case 'n':
            i++;
            user_opt_g.count = atoi(argv[i]);
            break;
        case 'l':
            i++;
            user_opt_g.size = atoi(argv[i]);
            if (user_opt_g.size > ICMP_MAX_SIZE)
                user_opt_g.size = ICMP_MAX_SIZE;
            break;
        case 'w':
            i++;
            user_opt_g.timeout = atoi(argv[i]);
            break;
        default:
            icmp_help(argv[0]);
            break;
        }
    }
}
int main(int argc, char **argv)
{
    WSADATA wsaData;
    SOCKET icmp_soc;
    struct sockaddr_in dest_addr;
    struct hostent *host_ent = NULL;
    int result, data_size, send_len;
    unsigned int i, timeout, lost;
    char *icmp_data;
```

```
    unsigned int ip_addr = 0;
    unsigned short seq_no = 0;
    if (argc < 2)
        icmp_help(argv[0]);
    icmp_parse_param(argc, argv);
    WSAStartup(MAKEWORD(2, 0), &wsaData);
    user_opt_g.host = "192.168.9.83";
    ip_addr = inet_addr(user_opt_g.host);
    if (ip_addr == INADDR_NONE)
    {
        host_ent = gethostbyname(user_opt_g.host);
        if (!host_ent)
        {
            printf("[PING] Fail to resolve %s\n", user_opt_g.host);
            return -1;
        }
        memcpy(&ip_addr, host_ent->h_addr_list[0], host_ent->h_length);
    }
    icmp_soc = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
    if (icmp_soc == INVALID_SOCKET)
    {
        printf("[PING] socket() failed: %d\n", WSAGetLastError());
        return -1;
    }
    timeout = user_opt_g.timeout;
    result = setsockopt(icmp_soc, SOL_SOCKET, SO_RCVTIMEO,
                        (char *)&timeout, sizeof(timeout));
    timeout = 1000;
    result = setsockopt(icmp_soc, SOL_SOCKET, SO_SNDTIMEO,
                        (char *)&timeout, sizeof(timeout));
    memset(&dest_addr, 0, sizeof(dest_addr));
    dest_addr.sin_family = AF_INET;
    dest_addr.sin_addr.s_addr = ip_addr;
    data_size = user_opt_g.size + sizeof(struct icmp_hdr) - sizeof(long);
    icmp_data = malloc(data_size);
    if (host_ent)
        printf("Ping %s [%s] with %d bytes data\n", user_opt_g.host,
               inet_ntoa(dest_addr.sin_addr), user_opt_g.size);
    else
        printf("Ping [%s] with %d bytes data\n", inet_ntoa(dest_addr.sin_addr),
               user_opt_g.size);
    for (i = 0; i < user_opt_g.count; i++)
    {
        icmp_make_data(icmp_data, data_size, seq_no++);
```

```
        send_len = sendto(icmp_soc, icmp_data, data_size, 0,
                          (struct sockaddr *)&dest_addr, sizeof(dest_addr));
        if (send_len == SOCKET_ERROR)
        {
            if (WSAGetLastError() == WSAETIMEDOUT)
            {
                printf("[PING] sendto is timeout\n");
                continue;
            }
            printf("[PING] sendto failed: %d\n", WSAGetLastError());
            break;
        }
        user_opt_g.send++;
        result = icmp_process_reply(icmp_soc);
        user_opt_g.persist ? i-- : i;
        Sleep(1000);
    }
    lost = user_opt_g.send - user_opt_g.recv;
    printf("\nStatistic :\n");
    printf("    Packet : sent = %d, recv = %d, lost = %d (%3.f%% lost)\n",
           user_opt_g.send, user_opt_g.recv, lost, (float)lost * 100 / user_opt_g
.send);
    if (user_opt_g.recv > 0)
    {
        printf("Roundtrip time (ms)\n");
        printf("    min = %d ms, max = %d ms, avg = %d ms\n", user_opt_g.min_t,
               user_opt_g.max_t, user_opt_g.total_t / user_opt_g.recv);
    }
    free(icmp_data);
    closesocket(icmp_soc);
    WSACleanup();
    return 0;
}
unsigned short ip_checksum(unsigned short *buf, int buf_len)
{
    unsigned long checksum = 0;
    while (buf_len > 1)
    {
        checksum += *buf++;
        buf_len -= sizeof(unsigned short);
    }
    if (buf_len)
    {
        checksum += *(unsigned char *)buf;
```

```
    }
    checksum = (checksum >> 16) + (checksum & 0xffff);
    checksum += (checksum >> 16);
    return (unsigned short)(~checksum);
}
```

实验内容 2：

```
#include "conio.h"
#include "iostream"
#include "stdio.h"
#include "string.h"
#include "winsock2.h"
#pragma comment(lib, "ws2_32.lib")
#define MAX_SiZE 4096
char cmdBuf[MAX_SiZE];
char command[MAX_SiZE];
char ReplyMsg[MAX_SiZE];
int nReplyCode;
bool bConnected = false;
SOCKET SocketControl;
SOCKET SocketDATA;

bool RecvReply(SOCKET sock)
{
    int nRecv = 0;
    memset(ReplyMsg, 0, MAX_SiZE);
    nRecv = recv(sock, ReplyMsg, MAX_SiZE, 0);
    if (nRecv == SOCKET_ERROR)
    {
        //cout<<endl << "socket recv failed!"<<endl;
        printf("socket recv failed!\n");
        closesocket(sock);
        return false;
    }
    if (nRecv > 4)
    {
        char *ReplyCodes = new char[3];
        memset(ReplyCodes, 0, 3);
        memcpy(ReplyCodes, ReplyMsg, 3);
        nReplyCode = atoi(ReplyCodes);
    }
    return true;
}
bool Sendcommand()
```

```
{
    int nSend;
    nSend = send(SocketControl, command, strlen(command), 0);
    if (nSend == SOCKET_ERROR)
    {
        printf("SocketControl create error:%d\n", WSAGetLastError());
        return false;
    }
    return true;
}
bool DataConnect(char *ServeripAddr)
{
    //向FTP 服务器发送PASV 命令
    memset(command, 0, MAX_SiZE);
    memcpy(command, "PASV", strlen("PASV"));
    memcpy(command + strlen("PASV"), "\r\n", 2);
    if (!Sendcommand())
    {
        return false;
    }
    //获得PASV 命令的应答消息
    if (RecvReply(SocketControl))
    {
        if (nReplyCode != 227)
        {
            printf("PASV 命令应答错误！");
            closesocket(SocketControl);
            return false;
        }
    }
    printf("%s\n",ReplyMsg);
    //解析PASV 命令和应答消息
    char *part[6];
    if (strtok(ReplyMsg, "("))
    {
        for (int i = 0; i < 5; i++)
        {
            part[i] = strtok(NULL, ",");
            if (!part[i])
            {
                return false;
            }
        }
        part[5] = strtok(NULL, ")");
```

```
        if (!part[5])
            return false;
    }
    else
        return false;
    //获得 FTP 服务器的数据端口号
    unsigned short serverPort;
    serverPort = (unsigned short)((atoi(part[4]) << 8) + atoi(part[5]));
    SocketDATA = socket(AF_INET, SOCK_STREAM, 0); //创建数据 SOCKET
    if (SocketDATA == INVALID_SOCKET)
    {
        printf("data socket creat error: %d", WSAGetLastError());
        return false;
    }
    SOCKADDR_IN server_addr;
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(serverPort);
    //server_addr.sin_addr.s_addr = inet_addr(ServeripAddr);
    server_addr.sin_addr.S_un.S_addr = inet_addr(ServeripAddr);
    //与 FTP 服务器之间建立数据 TCP 连接
    int nConnect = connect(SocketDATA, (sockaddr *)&server_addr, sizeof(server_ad
dr));
    if (nConnect == SOCKET_ERROR)
    {
        printf("create data TCP connection error : %d\n", WSAGetLastError());
        return false;
    }
    return true;
}
int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        printf("please input param as the following: ftpclient ftpiPaddr\n");
        return false;
    }
    if (bConnected == true)
    {
        printf("client has established the TCP control connection with server\n")
;
        closesocket(SocketControl);
    }
    WSADATA WSAData;
```

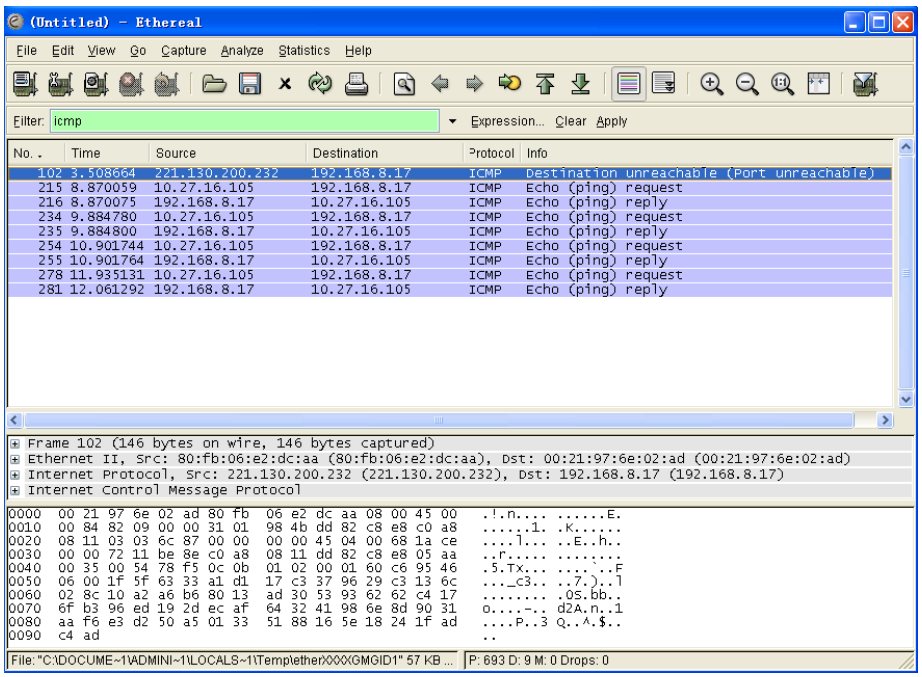
```
    WSAStartup(MAKEWORD(2, 2), &WSAData);
    SocketControl = socket(AF_INET, SOCK_STREAM, 0);
    SOCKADDR_IN server_addr;
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(21);
    server_addr.sin_addr.S_un.S_addr = inet_addr(argv[1]);
    int nConnect = connect(SocketControl, (sockaddr *)&server_addr, sizeof(server
_addr));
    if (nConnect == SOCKET_ERROR)
    {
        printf("client could not establish the FTP control connection with server
\n");
        return false;
    }
    //获取 Connect 应答消息
    if (RecvReply(SocketControl))
    {
        printf("%d\n", nReplyCode);
        if (nReplyCode == 220) //判断应答 Code
            printf("%s \n", ReplyMsg);
        else
        {
            printf("the reply msg is error\n");
            closesocket(SocketControl);
            return false;
        }
    }
    //向服务器发送 USER 命令
    printf("FTP->USER:");
    memset(cmdBuf, 0, MAX_SiZE);
    gets(cmdBuf); //输入用户名并保存
    memset(command, 0, MAX_SiZE);
    memcpy(command, "USER ", strlen("USER "));
    memcpy(command + strlen("USER "), cmdBuf, strlen(cmdBuf));
    memcpy(command + strlen("USER ") + strlen(cmdBuf), "\r\n", 2);
    if (!Sendcommand())
        return 0;
    //获得 USER 命令的应答信息
    if (RecvReply(SocketControl))
    {
        //printf("%d\n", nReplyCode);
        if (nReplyCode == 220 || nReplyCode == 331)
            printf("%s", ReplyMsg);
```

```
        else
        {
            printf("USER 命令应答错误\n");
            closesocket(SocketControl);
            return false;
        }
    }
    if (nReplyCode == 331)
    {
        //向 FTP 服务器发送 PASS 命令
        printf("FTP-> PASS:");
        memset(cmdBuf, 0, MAX_SiZE);
        gets(cmdBuf);
        memset(command, 0, MAX_SiZE);
        memcpy(command, "PASS ", strlen("PASS "));
        memcpy(command + strlen("PASS "), cmdBuf, strlen(cmdBuf));
        memcpy(command + strlen("PASS ") + strlen(cmdBuf), "\r\n", 2);
        //获得 PASV 命令的应答信息
        if (!Sendcommand())
            return 0;
        //printf("%s\n",command);
        if (RecvReply(SocketControl))
        {
            Sleep(1000);
            printf("%d\n", nReplyCode);
            if (nReplyCode == 230)
                printf("%s", ReplyMsg);
            else
            {
                printf("PASS 命令应答错误\n");
                closesocket(SocketControl);
                return false;
            }
        }
    }
    //
    DataConnect(argv[1]);
    //
    printf("FTP->LIST:\n");
    memset(command, 0, MAX_SiZE);
    memcpy(command, "LIST /", strlen("LIST /")); // /为参数
    memcpy(command + strlen("LIST /"), "\r\n", 2);
    if (!Sendcommand())
        return false;
```

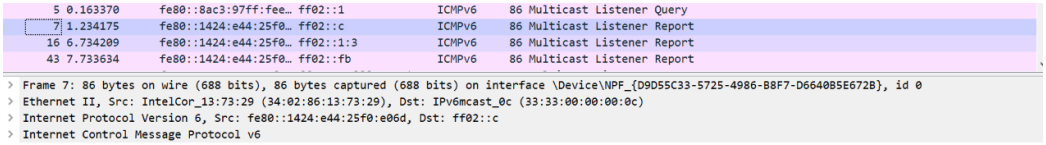
```
    //获得 quit 命令的应答信息
    if (RecvReply(SocketControl))
    {
        printf("%s", ReplyMsg);
    }
    if (RecvReply(SocketDATA))
    {
        printf("%s", ReplyMsg);
    }
    if (RecvReply(SocketControl))
    {
        printf("%s", ReplyMsg);
    }
    //Sleep(1000);
    //向 FTP 服务器发送 quit 命令
    printf("FTP->QUIT:\n");
    memset(command, 0, MAX_SiZE);
    memcpy(command, "QUIT", strlen("QUIT"));
    memcpy(command + strlen("QUIT"), "\r\n", 2);
    if (!Sendcommand())
        return false;
    //获得 quit 命令的应答信息
    if (RecvReply(SocketControl))
    {
        printf("%s", ReplyMsg);
        if (nReplyCode == 221)
        {
            printf("%s", ReplyMsg);
            bConnected = false;
            closesocket(SocketControl);
            return true;
        }
        else
        {
            printf("QUiT 命令应答错误\n");
            closesocket(SocketControl);
            return false;
        }
    }
    WSACleanup();
    return 0;
}
```

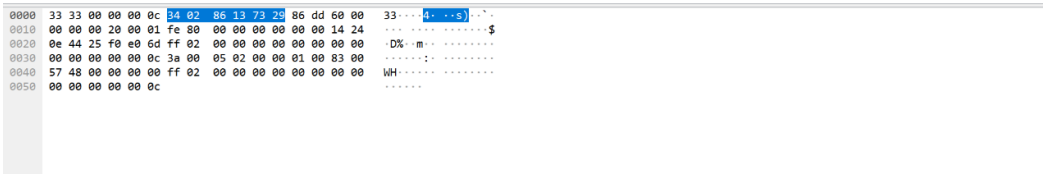
六、运行结果与分析

实验内容 1：

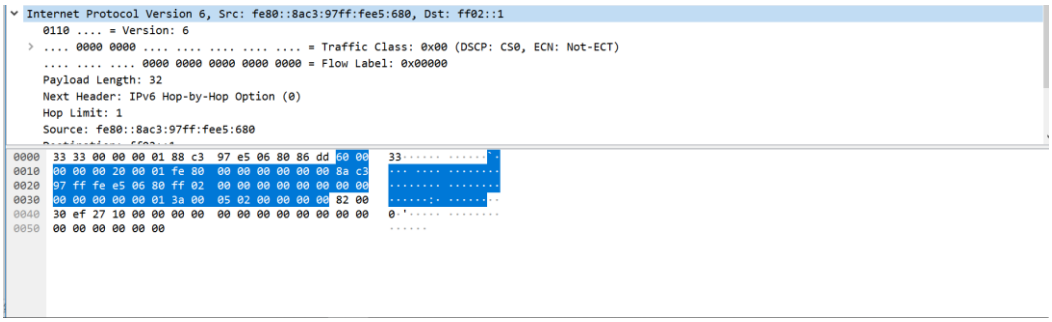


机房电脑截图如上，本机分析如下：

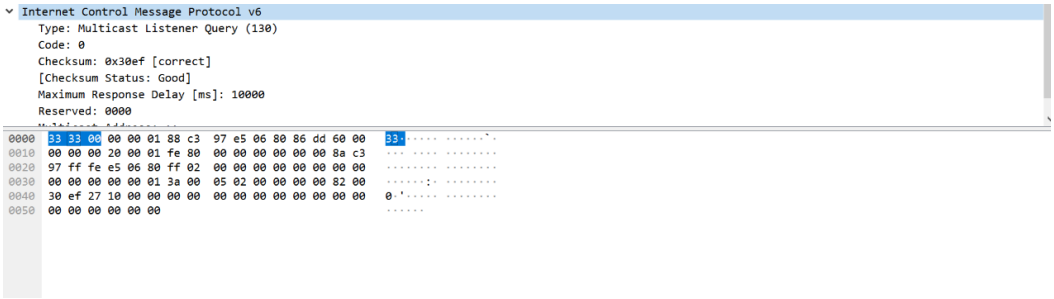




IP 首部：



ICMP 首部：

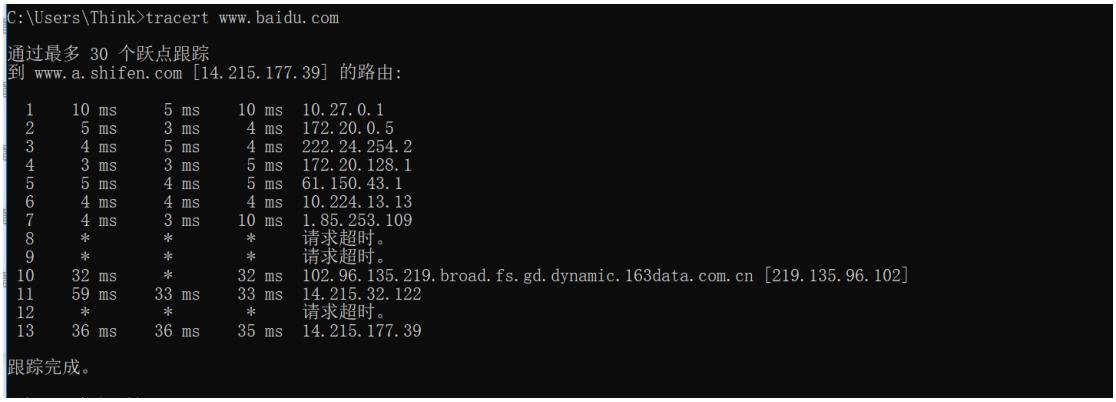


对 ping 命令实现的实例代码进行功能拓展具体代码如下：

```
static int icmp_unpack(char *buf, int len)
{
    int i, iphdrlen;
    struct ip *ip = NULL;
    struct icmp *icmp = NULL;
    int rtt;
    ip = (struct ip *)buf;
    iphdrlen = ip->ip_hl * 4;
    icmp = (struct icmp *)(buf + iphdrlen);
    len -= iphdrlen;
    if (len < 8)
    {
        printf("ICMP packets\'s length is less than 8\n");
        return -1;
    }
    if ((icmp->icmp_type == ICMP_ECHOREPLY) && (icmp->icmp_id == pid))
    {
        struct timeval tv_interval, tv_recv, tv_send;
        pingm_packet *packet = icmp_findpacket(icmp->icmp_seq);
        if (packet == NULL)
            return -1;
        packet->flag = 0;
        tv_send = packet->tv_begin;
        gettimeofday(&tv_recv, NULL);
        tv_interval = icmp_tvsub(tv_recv, tv_send);
        rtt = tv_interval.tv_sec * 1000 + tv_interval.tv_usec / 1000;
        printf("%d byte from %s: icmp_seq=%u ttl=%d rtt=%d ms\n"),
        len,inet_ntoa(ip->ip_src),icmp->icmp_seq,ip->ip_ttl,rtt);
        packet_recv++;
    }
    else
    {
        return -1;
    }
}
```

以上是对 ping 命令解压包并打印信息的拓展功能。

思考题：设计一个 tracert 命令（分析设计原理），根据 IP 数据报通信的特点，分析利用该命令获取的发送端到目的端的路径信息是否正确。若利用该命令可以获取发送端网络的网关 IP 地址（可通过抓包获取发送端网络网关的 MAC 地址），则分析利用该命令是否可以获取目的端所在网络的网关的 IP 和 MAC 地址，并解释原因。

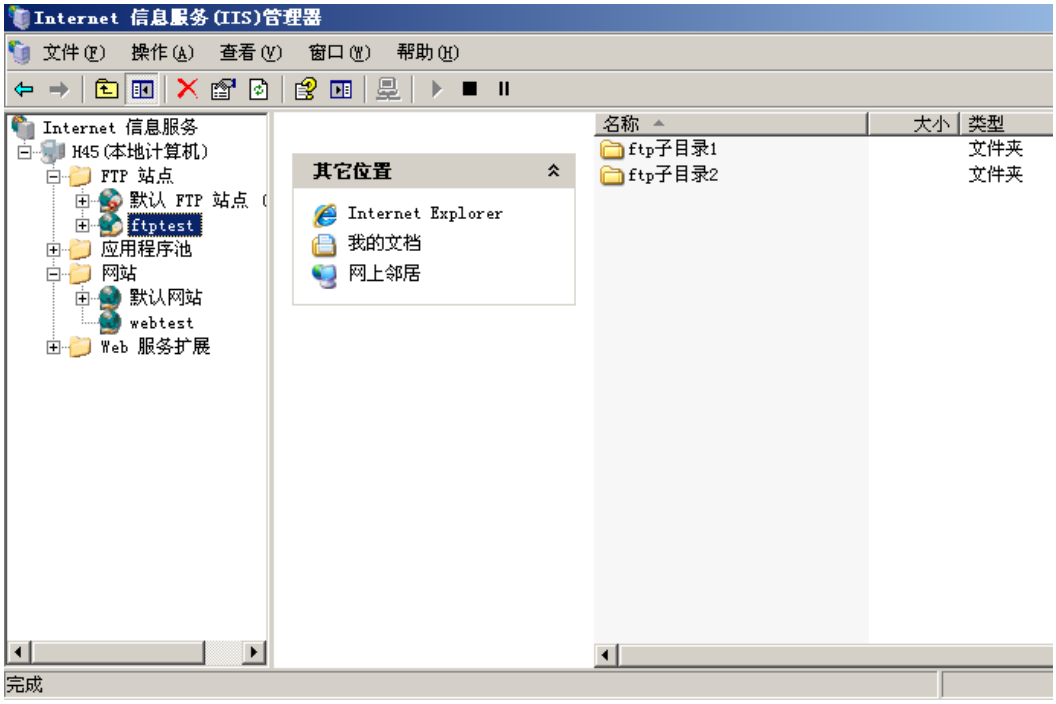


```
C:\Users\Think>tracert www.baidu.com

通过最多 30 个跃点跟踪
到 www.a.shifen.com [14.215.177.39] 的路由:

  1    10 ms     5 ms    10 ms  10.27.0.1
  2     5 ms     3 ms     4 ms  172.20.0.5
  3     4 ms     5 ms     4 ms  222.24.254.2
  4     3 ms     3 ms     5 ms  172.20.128.1
  5     5 ms     4 ms     5 ms  61.150.43.1
  6     4 ms     4 ms     4 ms  10.224.13.13
  7     4 ms     3 ms    10 ms  1.85.253.109
  8     *        *        *     请求超时。
  9     *        *        *     请求超时。
 10    32 ms     *       32 ms  102.96.135.219.broad.fs.gd.dynamic.163data.com.cn [219.135.96.102]
 11    59 ms    33 ms    33 ms  14.215.32.122
 12     *        *        *     请求超时。
 13    36 ms    36 ms    35 ms  14.215.177.39

跟踪完成。
```

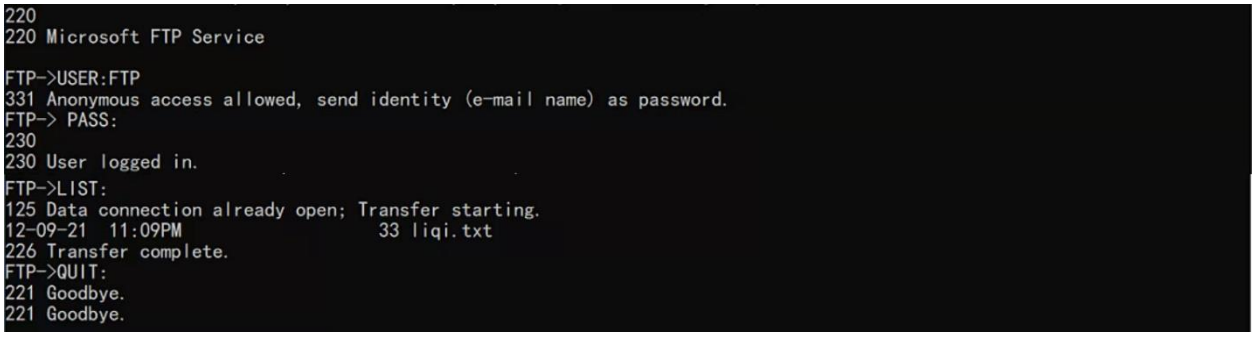
Tracert 命令用 IP 生存时间 (TTL) 字段和 ICMP 错误消息来确定从一个主机到网络上其他主机的路由。首先，tracert 送出一个 TTL 是 1 的 IP 数据包到目的地，当路径上的第一个路由器收到这个数据包时，它将 TTL 减 1。此时，TTL 变为 0，所以该路由器会将此数据包丢掉，并送回一个「ICMP time exceeded」消息（包括发 IP 包的源地址，IP 包的所有内容及路由器的 IP 地址），tracert 收到这个消息后，便知道这个路由器存在于这个路径上，接着 tracert 再送出另一个 TTL 是 2 的数据包，发现第 2 个路由器..... tracert 每次将送出的数据包的 TTL 加 1 来发现另一个路由器，这个重复的动作一直持续到某个数据包 抵达目的地。当数据包到达目的地后，该主机则不会送回 ICMP time exceeded 消息，一旦到达目的地，由于 tracert 通过 UDP 数据包向不常见端口(30000 以上)发送数据包，因此会收到「ICMP port unreachable」消息，故可判断到达目的地。

实验内容 2：

我们首先建立了一个 FTP 服务器，并且设置为可匿名登陆，然后通过可执行程序作为客户端连接 FTP 服务器。



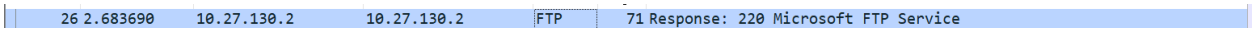
可以看到，我们成功登陆到 ftp 服务器，并接收到了服务器的消息。



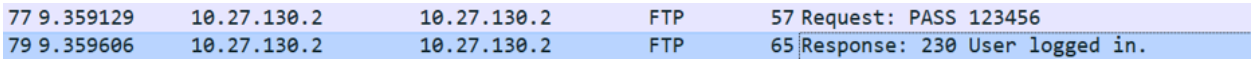
如图所示，登陆 ftp 账号，即可成功接入 ftp 并且能够查看目录下的文件。

以及 dir 命令的结果，也会显示目录下的文件，所以看到了两次文件目录。

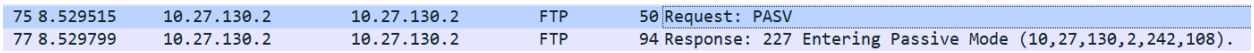
1.首先 FTP 客户端(本机用户)和服务器(本地创建的 ftp 站点)之间建立控制连接。



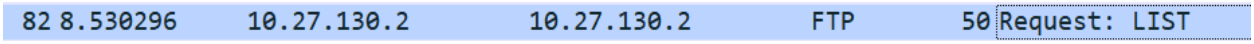
2.FTP 客户端通过控制连接向服务器发送账号信息（用户名+密码），进行身份认证。



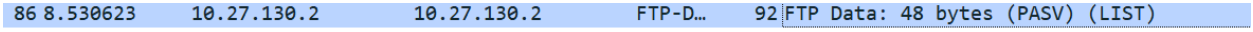
3.FTP 客户端通过控制连接向服务器发送 passiv 命令，说明用被动模式建立数据连接。



4.FTP 客户端与服务器之间通过被动模式建立数据连接。



5.FTP 客户端向服务器发送 dir 命令，服务器对该命令进行处理，并向客户端发送处理结果；



6.FTP 客户端接收服务器发送的处理结果（获得服务器当前目录下的列表信息），并在屏幕上显示。

84	8.530576	10.27.130.2	10.27.130.2	FTP	98 Response: 125 Data connection already open; Transfer starting.
90	8.530830	10.27.130.2	10.27.130.2	FTP	68 Response: 226 Transfer complete.

7.释放数据连接。

8.FTP 客户端向服务器发送 quit 命令，并释放控制连接。

9.通信结束。

94	8.531066	10.27.130.2	10.27.130.2	FTP	50 Request: QUIT
96	8.531137	10.27.130.2	10.27.130.2	FTP	58 Response: 221 Goodbye.

分析指导书上的问题：

（2）理解 FTP 协议中数据连接建立两种方式（被动模式和主动模式）的区别。

主动模式：客户端发送自己的 IP 地址和端口号给 FTP 服务器。服务器收到后，创建连接，去连接客户端发送来的 IP 地址和端口。

被动模式：。客户端发送 PASV 到服务器端，服务器开放接口，返回给客户端 IP 地址和端口号，客户端创建连接，和服务器进行连接。

（3）掌握控制连接和数据连接的建立方法和通信特点。

建立方法：客户端首先和服务器的 21 端口建立控制连接，在这个通道发送命令。客户端接收数据时，首先在这个通道上发送 PORT 命令，其中包含了客户端使用哪个端口接收数据。在传送数据的时候，服务器端通过自己的 TCP 20 端口发送数据。

特点：控制连接在服务器和客户端之间传输控制信息，如用户标识，命令等等，使用的是端口 21。数据连接用于传输文件数据等，使用的是端口 20。

（4）掌握多进程编程方法。

多进程指的是多任务同时正确地并发运行。

FTP 通信中，服务器在建立连接后需要一个循环来与客户端循环的收发数据，但服务器并不知道客户端什么时候会发来数据，导致服务器进入了一个等待状态，此时其他客户端也无法连接服务器。所以使用多进程编程可以让一个程序中的的多个任务可以同时被处理。

七、参考文献

《计算机网络原理实验分析与实践》 姚烨、朱怡安