华中科技大学
网络空间安全学院

《计算机通信与网络》实验报告

姓　　名_____

班　　级_____

学　　号_____

联系方式_____

分　　数_____

实验报告及代码和设计评分细则

评 分 项 目		满分	得分	备注
文档格式（段落、行间距、缩进、图表、编号等）		10		
感想（含思政）		10		
意见和建议		10		
验收时间		10		
Socket 编程	代码可读性	10		
	注释	10		
	软件体系结构	30		
	问题描述及解决方案	10		
实验报告总分		100		
教师签名			日　期	

目　　录

Socket 编程实验

一、 实验概述

1.1 实验名称

Socket 编程实验。

1.2 实验目的

通过 socket 程序的编写、调试，了解计算机网络可靠传输协议，熟悉基于 UDP 协议的 socket 编程方法，掌握如何开发基于 TCP/UDP 的网络应用。

1.3 实验环境

操作系统：Windows/Linux
编程语言：C，C++

1.4 实验内容

完成一个 TFTP 协议客户端程序，实现一下要求：
（1） 严格按照 TFTP 协议与标准 TFTP 服务器通信；
（2） 能够实现两种不同的传输模式 netascii 和 octet；
（3） 能够将文件上传到 TFTP 服务器；
（4） 能够从 TFTP 服务器下载指定文件；
（5） 能够向用户展现文件操作的结果：文件传输成功/传输失败；
（6） 针对传输失败的文件，能够提示失败的具体原因；
（7） 能够显示文件上传与下载的吞吐量；
（8） 能够记录日志，对于用户操作、传输成功，传输失败，超时重传等行为记录日志；
（9） 人机交互友好（图形界面/命令行界面均可）；
（10） 额外功能的实现，将视具体情况予以一定加分。

1.5 实验要求

（1） 必须基于 Socket 编程，不能直接借用任何现成的组件、封装的库等；
（2） 提交实验设计报告和源代码；实验设计报告必须包括程序流程图，源代码必须加详细

注释。

（3） 实验设计报告需提交纸质档和电子档，源代码、编译说明需提交电子档。

（4） 基于自己的实验设计报告，通过实验课的上机试验，将源代码编译成功，运行演示给实验指导教师检查。

二、 实验过程

2.1 系统结构设计

吸取了上次课设的教训，这次在开始写代码之前，我决定先完全想清楚设计的模型和模块功能划分，免得一次次重构。

而设计我的系统模块之前，当然是要先去学习一下类似的工程的设计思路。我参考了 Github 上面几个类似的工程，以及一个看起来似乎是某个标准协议的写法。最终大概确定了我的写法思路，那就是数据（主要是数据包）的控制和对传输的控制尽可能分离，而且把可能在 Client 和 Server 都用得到的部分提到一个单独的模块里，这样既方便检查，也为了编写 Server 提供了便利。

我的写法很大程度上参考了一个学长留下的代码，学长的代码给了我很大的启发，尤其是在写的规范上，比如使用 namespace、常量定义分离、变量名的标准以及模块分离的标准。也添加了几个辅助用的模块，包括常量以及全局变量的模块，输出调试信息的模块，还有一个 Github 上的命令行控制公开项目 AnyOption。但这个公开控制项目也有个问题，好像没办法来控制单个-？指令加上很多的字符串，并由此来完成多线程的输入，因此我额外设计了一个控制台方法来完成多线程的部分。

下面一张图说明我的模块模型，首先是模块名，第二栏标识具体功能，最后一栏标识对外的接口。因为这个模块图是我按照写成之后的程序来编写的，因此具有一些 C++中的类，函数之类的描述。

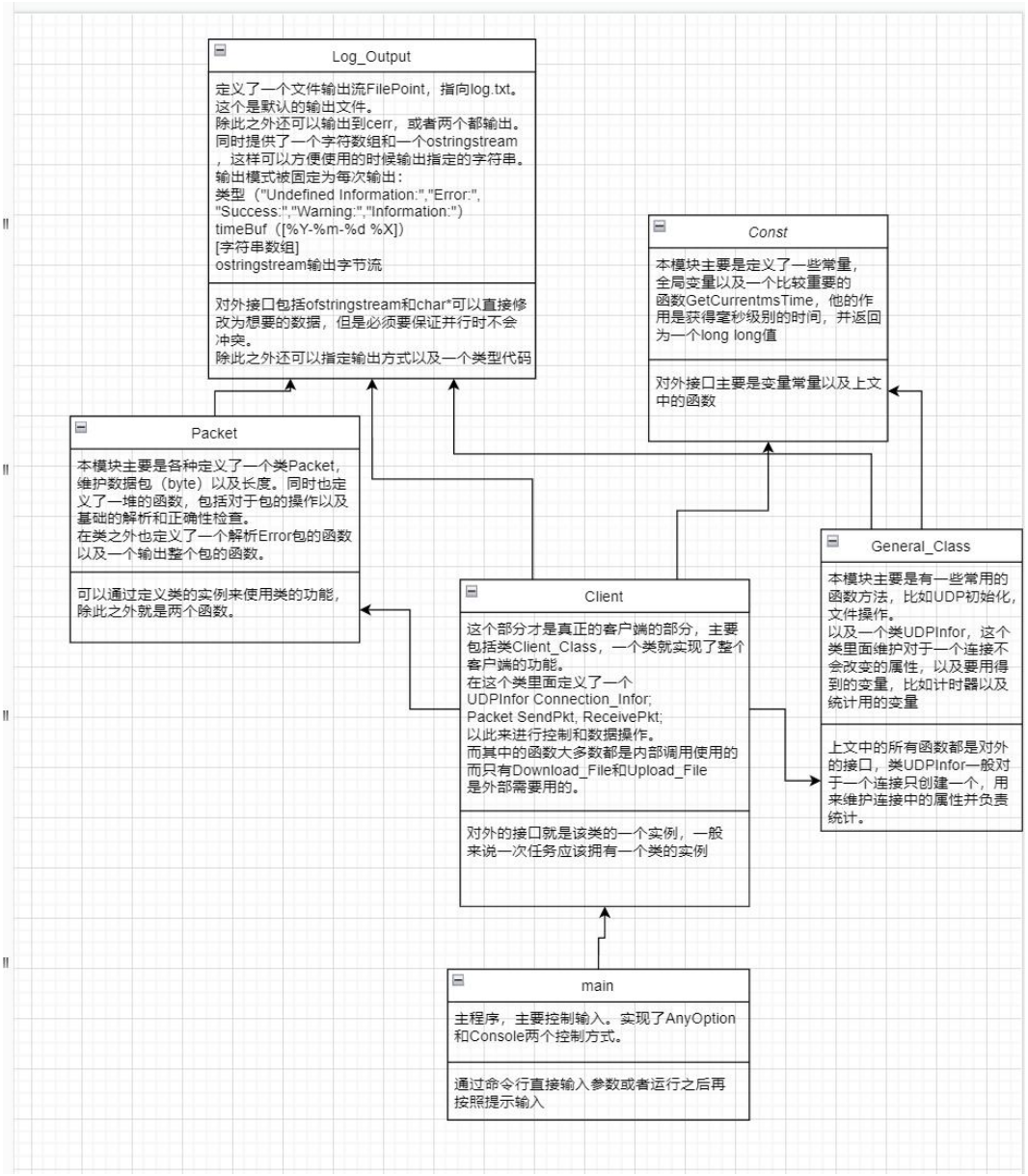

图 2-1 模块介绍图

根据本图容易发现我的模块设计结果是类似于一个层次图的结构，Const 和 Log_Output 构成最底层，Packet 和 General_Class 构成第二层，Client（和或许可以有的 Server）是第三层，main 构成最顶层。每一层只会调用更底层和同层中的功能，而向更高层提供服务。服务上主要是通过直接使用函数，或者通过类的实例来完成的。

而数据的处理流程，就是从下到上，逐步调用。

如此设计的主要优点有 1.结构清晰，避免了局部调用。2.整体问题局部化，有利于保证正确性。3.有利于系统的维护、扩充、移植。

2.2 详细设计

分开模块进行介绍,为了简单起见，直接分层介绍。

2.2.1 第一层 Const 和 Log_Output

```cpp
namespace TFTP{
    using std::cout;
    using std::cerr;
    using std::endl;
    using std::string;
    using std::ifstream;
    using std::ofstream;
    using std::ostringstream;
    using byte = unsigned char;
    using uint16 = UINT16;
    using uint = unsigned int;
    using uint64 = unsigned long long;
    using ll = long long;

    constexpr uint DefIp = 0x0100007f;  //默认IP(127.0.0.1)，待修改
    constexpr ll DefTimeOut = 200;      //默认超时时间
    constexpr uint DefRetries = 6;      //默认重传次数
    constexpr uint DefSleepTime = 10;      //默认没过20ms查看一次套接字缓存
    constexpr uint16 DefPort = 69;  //第一次连接的服务器端口号
    constexpr uint16 OpRRQ = 1;         //读请求操作码
    constexpr uint16 OpWRQ = 2;         //写请求操作码
    constexpr uint16 OpDATA = 3;        //数据操作码
    constexpr uint16 OpACK = 4;         //确认操作码
    constexpr uint16 OpERROR = 5;       //错误操作码
    constexpr uint16 LogUndefinedCode = 0;          //输出日志时未定义信息的开头标识
    constexpr uint16 LogErrCode = 1;        //输出日志时错误信息的开头标识
    constexpr uint16 LogSuccessCode = 2;     //输出日志时成功信息的开头标识
    constexpr uint16 LogWarnCode = 3;        //输出日志时警告信息的开头标识
    constexpr uint16 LogInforCode = 4;         //输出日志时信息的开头标识

    constexpr char RQMode[][15] = {"octet","netascii"};   //octet,netascii模式
    constexpr int DefBufSize = 1024;    //默认缓冲区大小
    constexpr int DataMaxSize = 512;    //最大数据大小
    //默认错误信息
    constexpr char ErrMsg[][34] = {
        "Undefined error code","File not found","Access violation",
        "Disk full or allocation exceeded","Illegal TFTP operation",
        "Unknown transfer ID","File already exists","No such user"
    };
    //获得UDPSocket的时候的默认错误信息
    constexpr char ErrMsg_getUDPSocket[][50] = {
        "底层网络子系统没有准备好。","Winsock 版本信息号不支持。","阻塞式 Winsock1.1 存在于进程中。",
        "已经达到 Winsock 使用量的上限。","lpWSAData 不是一个有效的指针"
    };
    //输出日志的5中Title
    constexpr char Log_Output_Title[][35] = {
        "Undefined Information:","Error:","Success:","Warning:","Information:"
    };
    //constexpr int ProcessWorkType = 0;      //整个进程的工作模式 0->控制台模式 1->命令行模式
    constexpr bool ClearLog = true;      //是否在初始化时清空Log文件
    extern bool ReadInforFromConfiguration;       //是否直接从配置文件中读取IP地址，下载模式、文件名等等  只有在Console模式才生效，当为true时，不会输出提示信息
    extern bool showInfo;   //是否展示传输细节标识
    extern int SktAddrLen;  //sockaddr大小
    extern bool EchoInputPara;   //是否回显输入参数


    long long GetCurrentmsTime();   //获取当前的时间 毫秒级
}
```

图 2-2 Const 模块

这个模块定义了所有可能需要用到的常数以及控制用的变量。因为本段我想不到不直接引用代码怎么方便的展示功能，因此直接贴上代码。而最后的一个 GetCurrentmsTime 函数将在下一节介绍。

而 Log_Output 主要是控制输出的，因为涉及到多线程，所以有必要对输出进行控制，减少冲突。

本模块主要是提供对输出信息的标准化封装，使得每一次输出都具有类型+时间+信息；同时也能够指定 cerr 和 log.txt 两个输出位置。

2.2.2 第二层 Packet 和 General_Class

Packet 模块顾名思义，就是为了处理数据包的，因此其核心功能就是维护一个表示一个数据包的结构，即一个类 Packet。在这个类中提供了修改数据块、获取数据块信息以及进行数据块检查的功能。

General_Class 在内容的角度来讲，完全可以放到 Client 模块中，但独立出来是因为这里涉及到的连接属性，对于客户端和服务器端都是可以使用的，也就是说这里定义了一个更通用的类型。而在本模块中进行的功能主要是 UDP 初始化，本地文件指针的初始化以及统计用变量的初始化。

2.2.3 第三层 Client

本模块是整个 TFTP 协议的最核心内容，他控制了整个传输过程。而在这个模块的内部，我采用的是进一步的进行封装，包括接收发送过程、包的封装、更主要的是定义了一个专门负责进行一次 TFTP 意义上的尽力而为的发送 DATA 包的函数，以及一个尽力而为的接收 DATA 包的函数。除此之外还有建立连接的函数，最外层则是直接面向外面的接口 Download_File 和 Upload_File。因此整个过程大概构成了下面这样的一个函数流程图：

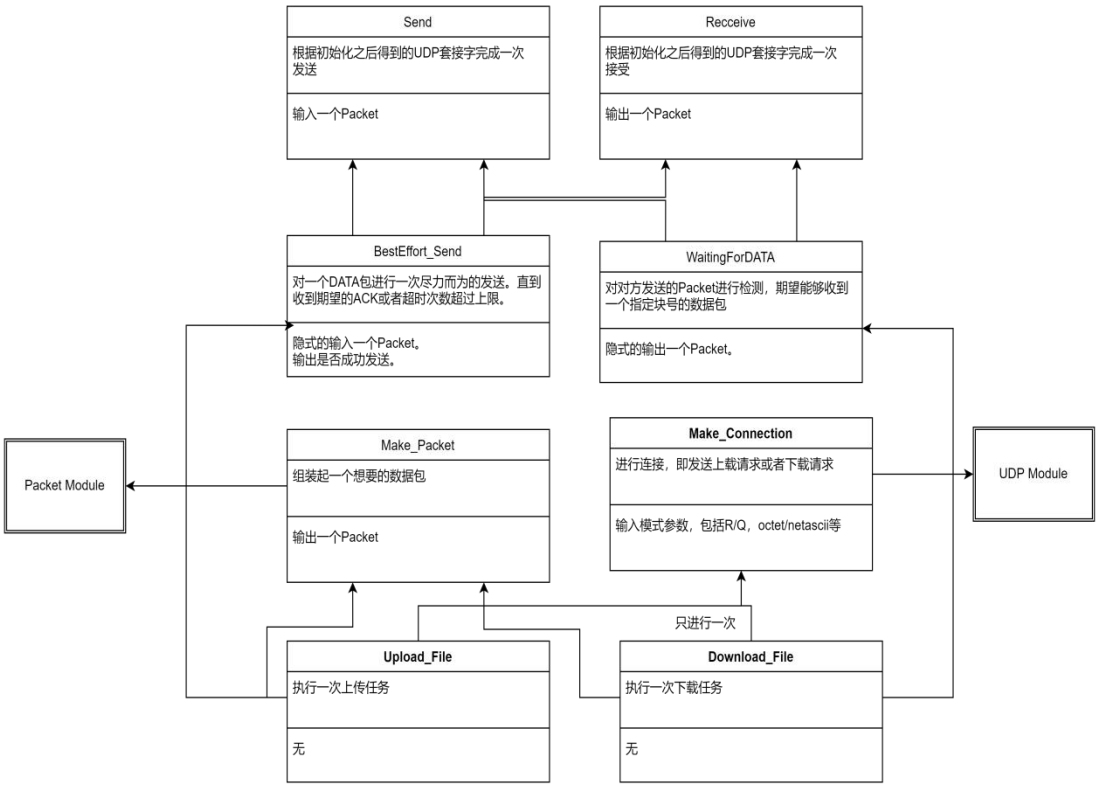

图 2-3 Client 模块

除此之外，还需要额外注意一个问题，就是丢包率到底应该怎么计算。从直觉出发，似乎很容易想到丢包率应该表示为 $1-\frac{\text{SuccessPacketNum}}{\text{TotPacketNum}}$，但实际上这个是不对的。因为实际上一次成功的传输，需要两次不丢包才行，也就是说

$$\text{TrueSuccessRate}^2 = \text{MeasuredSuccessRate}$$

$$\text{TrueLossRate} = 1-\sqrt{\frac{\text{SuccessPacketNum}}{\text{TotPacketNum}}}$$

应当按照这个来计算丢包率。这样修正之后，明显测量结果更接近实际结果了，具体可见第三部分系统测试。

2.2.4 第四层 main

本层是面向用户的界面，包括了 AnyOption 和 Console 两种模式。因为在更底层中已经完成了整个 TFTP 的方法。因此在这个模块中要做的就是构造一个方法（类）的实例，并且将参数（包括 IP 地址，文件名，模式等）传入到 Client 中定义的类中，之后指定工作类型，就可以全部交给下层去做了。

除此之外，因为每个任务化为了一个实例，因此这种写法可以支持多线程是显而易见的。而为了支持多线程，应当在本模块中实现多线程的创建，在生成之后仍然是交给下层。

2.3 代码实现

按照上一部分的方式，仍然分层来叙述。

2.3.1 Const 和 Log_Output

在 Const 库中，我写了一个函数 GetCurrentmsTime()，这看起来似乎违背了我希望该模块只定义常数的规定。但实际上因为 C++中如果想要一个获得毫秒级别的时间这个代码会非常长，加上后面经常使用这个功能，因此做了封装。之所以写在 Const 库中，是因为那位同层的 Log_Output 以及上层的模块都有使用这个模块的行为，本着分层编写，上层调用下层的原则，才把这个函数加到了这里。

```
long long TFTP::GetCurrentmsTime()
{
    return std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::system_clock::now().time_since_epoch()).count();
}
```

图 2-4 GetCurrentmsTime 函数实现

```
namespace TFTP {

    class Log_Output {
        //日志文件
        static ofstream FilePoint;    //输出流
        static char timeBuf[100];        //获取时间之后得到的字符串
        Log_Output();   //初始化输出流的地址，清空字节流
        ~Log_Output();   //关闭输出流
    public:
        static ostringstream Log_Msg;//输出字节流
        static void logNewLine();   //记录空行到Log_Msg
        static void GetNowTime();   //获得当前的时间到timeBuf数组中
        static void ClearLogMsg();   //清空Log_Msg和timeBuf
        static const char* FindOutput_Msg(int Infor_Type);          //找到当前type对应的字符串地址 [1-5]  或者 0 未定义

        //输出函数当Output_Msg==NULL的时候输出字节流
        static void OutputtoLog(const int Infor_Type,const char* Output_Msg);  //记录日志并输出到控制台
        static void OutputtoCerr(const int Infor_Type,const char* Output_Msg);   //输出到控制台
        static void OutputtoBoth(const int Infor_Type,const char* Output_Msg);   //两个都输出
    };


}
```

图 2-5 Log_Output 类定义

因为所用的文件指针唯一，所以实际上使用的全部都是静态方法，这样可以直接使用::来调用其中的函数。但造成的问题主要是在多线程的时候可能会冲突。

2.3.2 第二层 Packet 和 General_Class

```
class Packet {
public:
    byte buf[DefBufSize];   //数据包缓冲区   1024
    int Packetlen;  //数据包有效长度   要实时更新的
    Packet() :buf(""), Packetlen(0) {}   //构造函数，初始化

    void setPacketLen(const int l = 0);  //设置数据包有效长度
    int getDataLen() const;    //获取DATA包中数据长度  （总长度-4）

    void addByte(byte add_val);         //添加一个字节，会更新Packetlen
    void addUint16(uint16 add_val);     //添加两个字节 （传入变量没有进行字节翻转）   会更新Packetlen
    void addChars(const char* str);         //添加字符串，会加入\0   会更新Packetlen
    void addBytes(const byte* str, int strLenth);          //添加字节串，需要指定长度  会更新Packetlen
    void setOp(uint16 Opcode);    //设置数据包的操作码（传入变量没有进行字节翻转）


    byte* GetByteAddr(int Aimpls);  //获得buf+Aimpls处的地址（byte*）
    const byte* getErrMsg() const;   //获得ErrMsg信息  仅当Packet为ERRORPacket时调用
    const byte* getData() const;  //获取DATA包的数据(基地址)    仅当Packet为DATAPacket时调用

    uint16 ExtractUint16(int base = 0) const;      //解析base位置开始的两个字节成16位无符号数（传出变量已翻转为正常顺序）
    uint16 ExtractOpCode();     //解析Opcode
    uint16 ExtractBlockNo();   //解析BlockNo    仅当Packet为DATA/ACKPacket时调用
    uint16 ExtractErrCode();   //解析ErrCode    仅当Packet为ERRORPacket时调用


    bool PackRQ(uint16 OpRQ, const char* Filename, const int Modetype); //opRQ表示R/Q种类   FileName为请求的文件名   typ为0 传输二进制，否则netascii码
    bool PackDATA(uint16 blockNo, const char* data, const int Lenth); //封装DATA包 blockNo表示包的编号，data为数据
    void PackACK(uint16 blockNo);   //封装一个ACK包
    void PackERROR(uint16 errCode, const char* msg = nullptr);   //封装一个ERROR包  errCode为1-7，否则输入msg，再否则回复未定义错误

    uint16 CheckPacket();       //核查数据包类型以及基本的合法性判断,需要预先获取包以及Packetlen    对于一个合法包，返回值为op值
    //否则为  -1->Too Short  -2->Too Long  -3->Illegal String  -4->Illegal ErrCode  0->Unknown Opcode
};
```

图 2-6 Packet 类定义

里面大多数函数都是易于实现的，只有一个 CheckPacket 需要展开。大概如下：

Algorithm 1 CheckPacket

```
function "CHECKPACKET"
    if "Packet_Lenth <This_OpCode_Min_Lenth" then return "TooShort"
    end if
    if "Packet_Lenth <This_OpCode_Max_Lenth" then return "TooLong"
    end if
    if OpCode == "RRQ" and OpCode == "WRQ" then
        if RQMode! = "octet" and RQMode! = "netascii" then return "WrongMod"
        end if
    end if
    return "OK"
end function
```

```
class UDPInfor {
public:
    sockaddr_in addr;       //对方的socketaddr
    SOCKET Local_Socket;    //本地的socet
    FILE* Local_FilePointer;    //本地打开文件的指针
    int FunctionType;           //表示现在的功能类型 0表示未知  1表示下载文件  2表示上载文件
    char FilePath[100];         //文件路径字符串，当为客户端时，存储目标文件的名字
    long long Begin_Time;          //连接开始的时间
    long long ResendTimer;          //重传计时器
    int RemainResendNum;        //剩余的重传次数
    int SuccessBytes;           //已经传输成功的字节数，ACK后才会被计算
    int SuccessPacketNum;       //传输成功的数据包数
    int TotPacketNum;          //总共传输的数据包数
    sockaddr_in Received_addr;      //接收包的sockaddr  初始时悬空？

    UDPInfor();
    UDPInfor(const char* ip, uint16 port);    //从一个字符串初始化IP地址以及端口号
    UDPInfor(byte a, byte b, byte c, byte d, uint16 port);  //从四个byte初始化IP 地址 以及端口号
    ~UDPInfor();

    uint16 ChangePort(uint16 NewPort);      //改变ip地址 输入的应当为正常字节序
    uint64 ChangeIP(const char* ip);       //改变端口号  正常字节序输入
    sockaddr* getSktAddr();    //获取sockaddr型指针
};
```

图 2-7 UDPInfor 类定义

其中所有函数和变量定义都有注释，而实现也都是比较简单的。

2.3.3 第三层 Client

根据上一节中的描述可以发现，可以清晰的看到函数之间的调用过程。下面使用伪代码来近似描述实现过程，因为不涉及具体语言所以里面的很多过程都是用了简略的描述带过。一半来说，省略的内容都是比较容易实现的几行代码，不涉及非常复杂的操作。

Algorithm 2 Client

```
1: function "BestEffort_Send"
2:     Initialization
3:     error_Code=SendPacket()
4:     if corrupt(error) then return −1
5:     end if
6:     while "true" do
7:         error_Code=ReceivePacket()
8:         if corrupt(error) then
9:             if timeout and RemainResendNum==0 then return −1
10:            else if timeout then
11:                error_Code=SendPacket()
12:                if corrupt(error) then return −1
13:                end if
14:                Restart(timer)
15:            else
16:                sleep()
17:            end if
18:        end if
19:        Prase_Packet
20:        if Op_Code == Error_Code then
21:            Prase_Error_Packet
22:            return −1
23:        end if
24:        if Op_Code == neededPacketType and (Don't have Block Num or Block_Num == needed-
   BlockNum) then
25:            Update_statistics
26:            return 0
27:        else
28:            Drop_Packet
29:            Decrease the timer by half
30:        end if
31:    end while
32:    return −1
```

```
end function


function "WaitingForDATA"
    Initialization
    while "true" do
        error_Code=ReceivePacket()
        if corrupt(error) then Sleep()
        end if
        Prase_Packet
        if Op_Code == Error_Code then
            Prase_Error_Packet
            return -1
        end if
        if Op_Code == neededPacketType and (Don't have Block Num or Block_Num == neededBlockNum) then
            Update_statistics
            return 0
        else
            Drop_Packet Sleep()
        end if
    end while
    return -1
end function


function "Make_Connection"
    Initialization
    AssemblyRQPacket
    if BestEffort_Send then
        print Connection_Error return -1
    end if
    Update_statistics
    Change_Aim_Port
    return 0
end function


function "Upload_File"
    Initialization
    if Make_Connection then
        print Upload_Error return -1
    end if
    lenthofdata=512
    while "lenthofdata==512" do
        lenthofdata=File_Read
        Pack_Data
        if BestEffort_Send then
            print Connection_Error return -1
        end if
```

```
82:         Update_statistics
83:     end while
84:     return 0
85: end function
86:
87:
88: function "DOWNLOAD_CONNECTION"
89:     Initialization
90:     if Make_Connection then
91:         print Connection_Error return -1
92:     end if
93:     lenthofdata=ReceivePkt.lenth
94:     File_Write
95:     while "lenthofdata==512" do
96:         if WaitingForDATA then
97:             print Download_Error return -1
98:         end if
99:         lenthofdata=ReceivePkt.lenth
100:        File_Write
101:        SendACK
102:        Update_statistics
103:    end while
104:    return 0
105: end function
```

2.3.4 第四层 main

本层主要是实现一个 Windows 下的简易用户界面，可以通过 Console 和 AnyOption（使用命令行参数）两种方式进行控制。

核心思路就是从控制台输入/命令行输入中找出需要的数据参数，首先判断是否合法，如果合法，就将其赋予一个 Client 类实例中的某一个变量。

对于多线程部分，则是这样实现的（直接引用 C++代码）：

图 2-8 多线程的创建 1

```
unsigned int __stdcall BeginMultiThread(void* Para) {
    Client_Class* NowTask = (Client_Class *)Para;
    if (NowTask->Connection_Infor.FunctionType == 1) {
        NowTask->Download_File();
        //_endthreadex
    }
    else if (NowTask->Connection_Infor.FunctionType == 2) {
        NowTask->Upload_File();
    }
    else Log_Output::OutputtoBoth(2, "Wrong FunctionType!!!");
    _endthreadex(0);
    return 0;
}
```

图 2-9 多线程的创建 2

用_beginthreadex 来建立线程比直接使用 CreateThread 要更加的安全，因此使用这种方法。

在线程最后结束的时候，要加上_endthreadex(0)。

三、 实验测试与分析

3.1 系统测试及结果说明

如果需要使用 AnyOption 模式运行可执行程序并输入-h 可以查看到如下的信息。按照提示进行输入即可。

```
Help:
    If you wish to use multithreaded transmission ,please use the console method.
    In order to use this method please run the program without entering any parameters.And follow the instructions

    Otherwise,please enter the parameters according to the following BNF
    Socket_Programming.exe - i <serverip> -u|d <filepath> [-m <mode>] [-s]
    -h  --help      Print usage information and exit
    -i  --ip        Server ip address xxx.xxx.xxx.xxx xxx in [0,255]
    -u  --upload    Upload file to server ,follows the file name
    -d  --download  Download file from server ,follows the file name
    -m  --mode      TFTP modes of transfer: octet(b) or netascii(t).Default is octet
    -s  --show      Show detail about packet Default is false
```

图 3-1 AnyOption 实现结果

否则如果执行后不跟参数，则自动进入 Console 模式。按照提示输入即可。

考虑到如果要全部测试，需要测试的太多。尤其是错误测试，如果想要测试所有的错误类型，可能需要几百个不同的输入。因此采用部分测试。尽可能覆盖到所有功能。以下是四个测试的情况

3.1.1 octet 模式上传文本文件

图 3-2 传输指令和回显信息

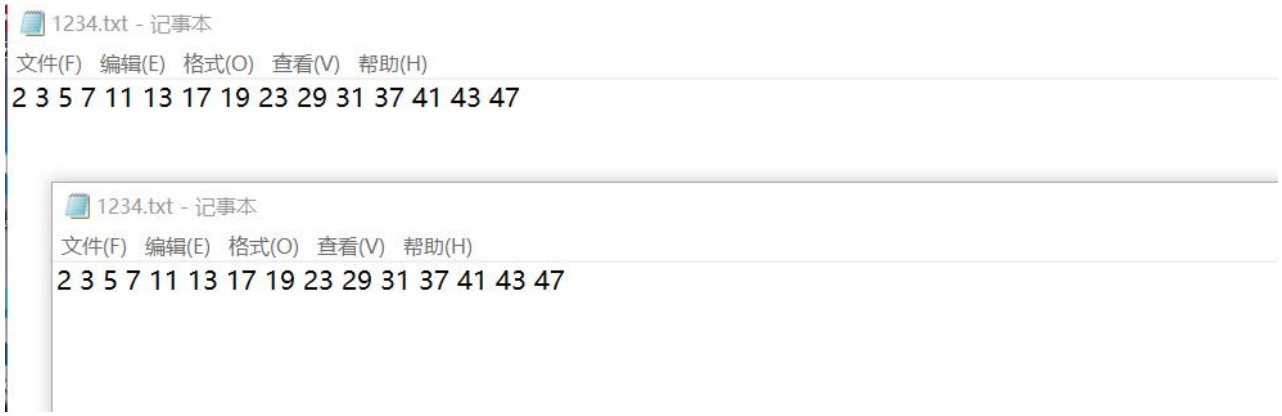

图 3-3 传输结果

图 3-4 完成后统计信息显示

图 3-5 日志文件输出

在开启详细显示的情况下，log 显示和控制台输出一致，所以后面只展示控制台输出。而在不开启详细现实的模式下，log 会输出详细信息，而控制台只输出简报。

3.1.2 netascii 模式下载图片文件

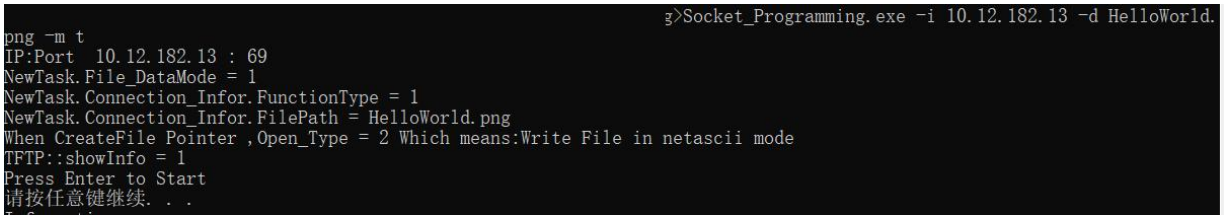

图 3-6 传输指令和回显信息

图 3-7 传输结果

图 3-8 完成后统计信息显示

意料之中的出现了错误。因为图片文件并不是按 ascii 字符串存储的。

3.1.3 丢包开启时的异常情况测试

图 3-9 传输指令和统计丢包率

测得丢包率为 0.095466 非常接近给定的 0.1，或许是因为使用的开根号的算法（具体请见详细设计一节）。

之所以采用小的丢包率进行测试，是因为如果想要测试准确，一定要传输大文件，而对于一个大文件，传输的数据包数量非常多，如果丢包率开的过大，很容易出现传输中断的情况。

3.1.4 多线程上传

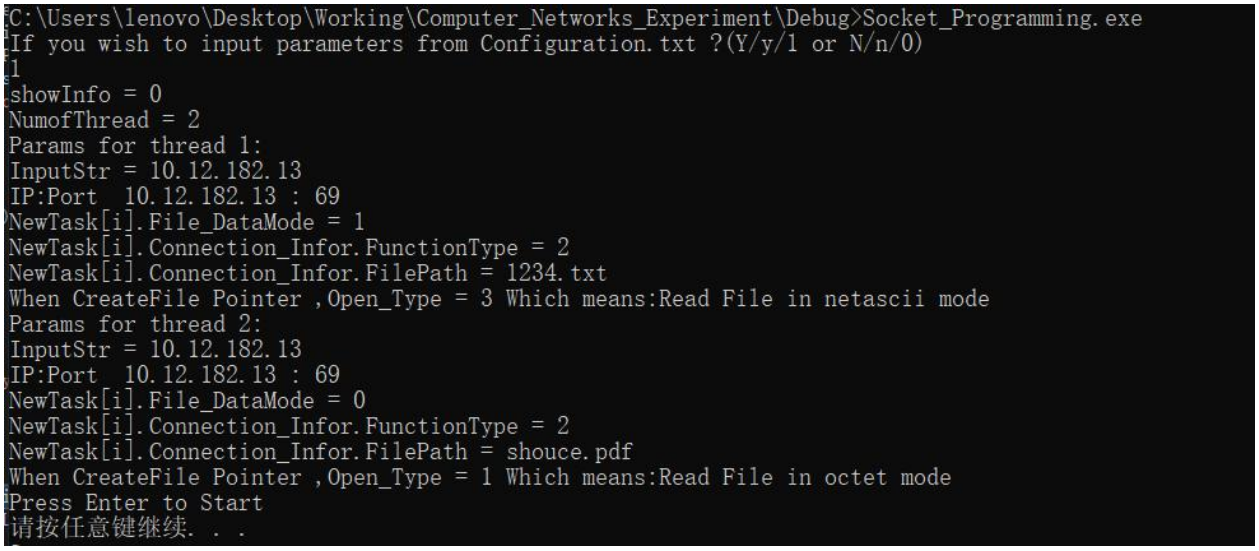

图 3-10 传输指令和回显信息

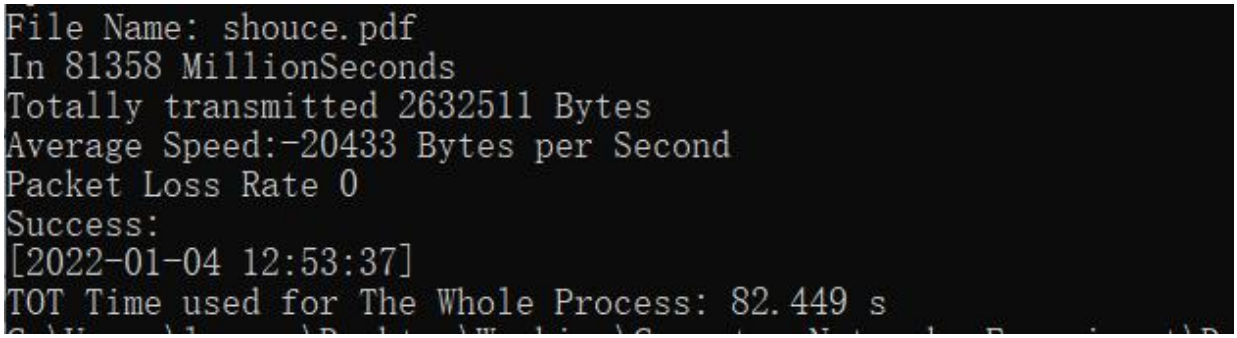

图 3-11 完成后统计信息显示

另一个很早就传输完了，已经被刷到屏幕外面了。

图 3-12 传输结果

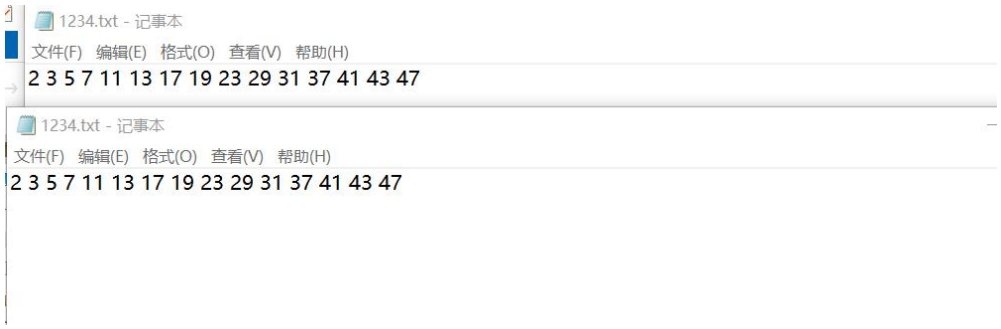

图 3-13 传输结果

3.2 遇到的问题及解决方法

遇到的问题那可太多了，从设计到实现，再到测试到处都是问题。

大多数技术上的问题都解决了，但是有些在设计上就没法弥补的缺陷最终还是没有解决，关于这些在下一节中陈述，本节主要是一些在当时编写程序的时候出现的问题。

1. 引用库可能会重复引用，导致错误。

   解决：

```
#ifndef _??_H
#define _??_H
#endif
```

2. 在.h 中定义的函数原型，在 cpp 中实现的时候，不能直接包在 namespace 里面。

   解决：

```
long long TFTP::? ? ()
{
}
```

3. 时间精度不够。

   解决

```
1. std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::system_clock::now().time_since_epoch()).cou
nt();
```

4. 当文件恰好可以发送为整数个包的时候。
   解决： 再发送一个 data 大小为 0 的包
5. 当文件过大，导致 Blk_No 超过 65535.
   解决：从 0 重新循环
6. 多线程用到的函数无法正常调用
   解决：添加__stdcall
7. 因为文件中读取到的字符串出现’\0’导致中断。
   解决：使用 memcpy 而不是 strcpy
8. 丢包率计算误差较大
   解决：使用正确的估计公式。
9. 有的时候和同学们无法互传，但是过一段时间又可以了。
   解决:怀疑是校园网的问题

3.3 设计方案存在的不足

我觉得如果有什么不足的话，最主要的是没有一个 GUI，但是这个黑色的控制台界面其实使用的话已经很方便了。但是毕竟少了界面总是让人感觉有点遗憾。

我在设计输出的时候采用的使用 static 来完成的。虽然这样子不会出现多个进程同时进入输出这个函数，但仍然避免不了所有多线程的输出全都堆在一起，很难分辨出谁是谁。这是因为我并没有对线程编号。在创造线程的时候直接把类实例的指针穿了进去，所以也不知道到底是第几号的线程。

还有我的传输速度移植非常的慢，即使关闭了输出传输速度依然超不过 10MB/s 的速度，我也不太清除是因为什么原因，我怀疑可能是我睡眠时间设置有些问题。

最后时我的这个设计方案是考虑了如果增加上 Server 部分的方便性的，所以才把共同的部分提取出来。但是一直到现在我也没腾出时间来编写 Server，实现一个完整的 TFTP，有点遗憾。

四、 实验总结

4.1 实验感想

实验一明显是需要花最长时间的一次实验，在这个实验里，按照要求我们复现了 TFTP 协议的客户端部分，我觉得这种从 0 开始（指应用层）对着一个规则来一边揣测制造者的想法，一边自己造轮子对于深刻的学习这个知识是是非常有意义的。但是 TFTP 的官方协议（RFC）说明属实有点简单，而且对于很多特殊情况也没有很明确的说明，包括对于一些奇怪情况的处理，也包括了一些参数的选择方法。因此最重要的检测自己做的对不对的方法反而是直接去和 Server 端交互。反过来说，如果真的让我们实现一个极其复杂，要求及其细致的协议，恐怕我们也很难复现出来。所以综上我觉得这个课设的选题还是很不错的。

而且这次试验我也使用了分层编写的思路，在此再次感谢学长的工程代码，他的代码写法非常的标准，无论是使用命名空间或者这种面向对象的构造程序方法都让我学到了很多。

第二次实验中自己制做网线确实挺有意思，可惜最终也没有弄成一个 8 根线全部通了的成品。不过这个过程还是蛮有意思的。后面使用 eNsP 的部分，因为实际上上课内容还没有到这里，所以做的也是云里雾里的，反正大概就是照着 ppt 或者问别人然后直接写代码这样过去的。后来在学习完理论知识之后，再回想实验的时候，才能够比较清晰的理解这个过程。但是仔细想想，我还是觉得先做实验后学理论可能在做实验的时候还是收获可能会小一点，或许先学理论要好一些。

在第二次实物连接实验的时候，我们组意外的选用了一台坏了无线路由器的设备组做实验。结果就是我们尝试了各种网上搜来的办法，就差直接把路由器给拆了，但是还是无法成功。最后还是 psk 同学提醒我们或许是设备坏了，让我们换一台设备进行试验，最终我们把保持下面的不动，然后用一根长线连接上后面一排的无线路由器，才算是通过了实验。那天晚上一直做到十点多才完成。助教也一直等着我们检查，感谢助教陪我们到这么晚。

第三次实验我反而觉得是做的最顺畅的，基本上就一直照着做，中间偶尔卡一下问问同学就解决了，最后检查也比较轻松的通过了。或许就是因为做的太顺了，所以第三次实验我反而觉得可能收获到的知识比较少，因为像我这种摸鱼人，只有当完全照做会出现问题的时候，我才会去研究原理尝试修补。

总的来说三次实验都是比较有意思的，难度和任务量也比较适中，而且确实对于我理解计算机网络的理论知识起到了很好的作用，理论指导了实践，实践又反哺了理论学习，因此我觉得这次实验我收获很大。

最后还是要感谢实验过程中老师同学们的帮助。

4.2 意见和建议

希望在提供一份简洁易懂的任务书之外，可以提供一份更详细完备的资料。这次虽然也提供了不少的资料，但是看完之后仍然有疑惑。

网线实验的时候，希望能多给一些时间（X。而且就我知道的成功率来说，没让大家用自己搓的网线来做实物连接实验真是太正确了。

虽然我不太清楚 eNsP 的内部实现机制是不是比较复杂，但这个东西实在是太慢了，启动慢，关闭也慢。而且听其他同学说保存文件有时也出问题。

建议老师可以先筛选一下有硬件问题的设备。
这个实验和理论课程的同步或许还可以再优化一下。

原创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

已阅读并同意以下内容。

判定为不合格的一些情形：

（1） 请人代做或冒名顶替者；

（2） 替人做且不听劝告者；

（3） 实验报告内容抄袭或雷同者；

（4） 实验报告内容与实际实验内容不一致者；

（5） 实验代码抄袭者。

作者签名：