合肥工業大學

计算机与信息学院

《计算机网络课程设计》报告

设计题目：网络文件传输

学生姓名：曲艺
学　　号：2020218037

专业班级：计算机 4 班

2023 年 6 月

一、任务描述及设计要求

1、任务描述

设计一个网络文件传输系统，可以通过 UDP 或 TCP 实现文件传输功能。系统要求实现单进程（线程）的文件传输，包括传输 ASCII 文本文件和二进制文件（如图片、音视频）。系统还需要通过 UDP 实现大型音频或视频文件的传输，并验证接收端能够正确播放接收到的文件。此外，系统需要具备异常控制功能，提高程序的鲁棒性。最后，要求了解如何提高套接字传输的速率和稳定性。

2、设计要求

①实现单进程（线程）文件传输功能，包括 ASCII 文本文件和二进制文件。

②验证接收端打开的文本文件内容与发送端是否一致。

③使用 UDP 实现大型音乐或视频文件（大于 15M）的传输，并验证文件的完整性和正确播放。

④通过修改代码观察函数出错时的错误图片并进行分析。

⑤观察发送长度为 65535 字节的数据的结果并进行分析。

⑥观察接收端开辟的接收缓冲区小于发送端发送的数据长度时的结果并进行分析。

⑦利用 TCP 多线程技术实现大文件的网络多线程传输，并分析在固定网络带宽情况下线程数量与传输速率的关系。

⑧加入异常控制以增强程序的鲁棒性。

⑨学习如何提高套接字传输的速率和稳定性。

二、开发环境与工具

操作系统：Windows 11 家庭中文版（版本号：21H2）

开发工具：Visual Studio 2022

编程语言：C/C++

框架与库：MFC（Microsoft Foundation Classes）

网络通信：Winsock

互联网协议：IPV6

三、设计原理

在我的课程设计中，使用了 Winsock 来实现网络文件传输功能。下面是所用到的计算机网络相关知识和原理的概述：

1、计算机网络基础：

我们知道，计算机网络是将多台计算机通过通信链路连接起来，实现数据传输和资源共享的系统。在网络通信中，我们需要了解数据传输介质、通信协议、网络拓扑结构等基本要素。

2、套接字（Socket）编程：

套接字是网络编程中用于实现网络通信的接口，它提供了一组函数和数据结构，用于创建、连接、发送和接收数据。在我的设计中，我使用了 Winsock，这是 Windows 操作系统上的套接字编程接口，通过它可以实现网络通信功能。

3、TCP/IP 协议族：

TCP/IP 协议族是互联网的核心协议，包括 TCP（传输控制协议）和 IP（互联网协议）等协议。TCP 协议提供可靠的、面向连接的数据传输，确保数据的完整性和顺序性。而 IP 协议则负责将数据包从源主机传输到目标主机。

4、UDP 协议：

我在设计中使用了 UDP（用户数据报协议），它是一种无连接的、不可靠的传输协议，适用于实时性要求较高、数据丢失可以容忍的应用场景。UDP 提供了简单的数据传输机制，没有建立连接和数据重传的开销，但无法保证数据的可靠性和顺序性。

5、文件传输：

在我的设计中，我实现了 ASCII 文本文件传输和二进制文件传输。

ASCII 文本文件由字符组成，可以使用 TCP 或 UDP 进行传输。而二进制文件（如图片、音视频等）需要进行特殊处理，一般使用 UDP 进行传输。

6、异常控制和鲁棒性：

在网络通信过程中，可能会出现各种异常情况，例如网络延迟、丢包、连接中断等。为了增强程序的鲁棒性，我进行了异常控制，处理异常情况，以保证程序的稳定性和可靠性。

四、系统功能描述及软件模块划分

1、实现的主要功能

①能够通过 ipv6 实现公网 TCP 文件传输

②多线程文件网络传输，包括但不限于文本、音频和视频

③大文件网络传输

2、软件模块划分

本次课程设计所实现的文件传输软件，我将文件传输的客户端和服务端封装成了一个程序，这个也就意味着用户既可以主动和其他用户进行连接，也可以被动的等待其他用户的连接，在用户看来，他与其他用户之间是等价的，这样也不会依赖服务器进行文件转发，避免被他人窃取。

（1）服务端

①端口监听模块。该模块会在程序启动时自动初始化，监听自己的端口以响应客户端的连接。该端口的连接用作“协商”，通过“协商连接”与客户端进行协商，协商的内容包括“文件名”、“文件长度”和“需要的线程数”，协商完成后创建指定数量个线程用于文件接收，向线程传入参数“文件路径”。

②文件接收模块。该模块首先会监听当前线程的端口，等待客户端“发送线程”的连接；连接成功之后，首先接收发送端传来的“字节索引”并给出回应，然后开始接收客户端发来的文件内容向指定文件中的指定索引位置写入。

（2）客户端

①连接模块。该模块仅用于建立“协商”连接，允许用户手动输入 IP 地址进行连接，并在连接按钮的上方显示连接状态。该模块会在用户点击连接之后，与服务端建立“协商”连接，该链接用户二者协商“文件名”、“文件长度”和“需要的线程数”。

②文件选择模块。用户可以通过该模块来可视化的选择本地电脑中的文件，通过“发送”按钮，可以实现文件的发送。

③文件发送模块。当用户点击“发送”时，客户端开始向服务端发送 “文件名”、“文件长度”以及协商“需要的线程数”；当协商完成后启动多个

线程向服务器发送 “传输的起始字节索引”，等待服务器回应后，开始按序读取并发送文件内容。

④详细内容显示模块。通过该模块向用户显示文件传输的情况。

五、设计步骤

使用 winscok 实现的网络文件传输，同时也需要对网络连接的安全和有效进行保障，代码中该部分出错处理代码较为重复，因此只展示了部分核心代码，对“出错处理的代码”和“建立连接的代码”进行了省略。

（1）服务端

①端口监听模块（主要模块）

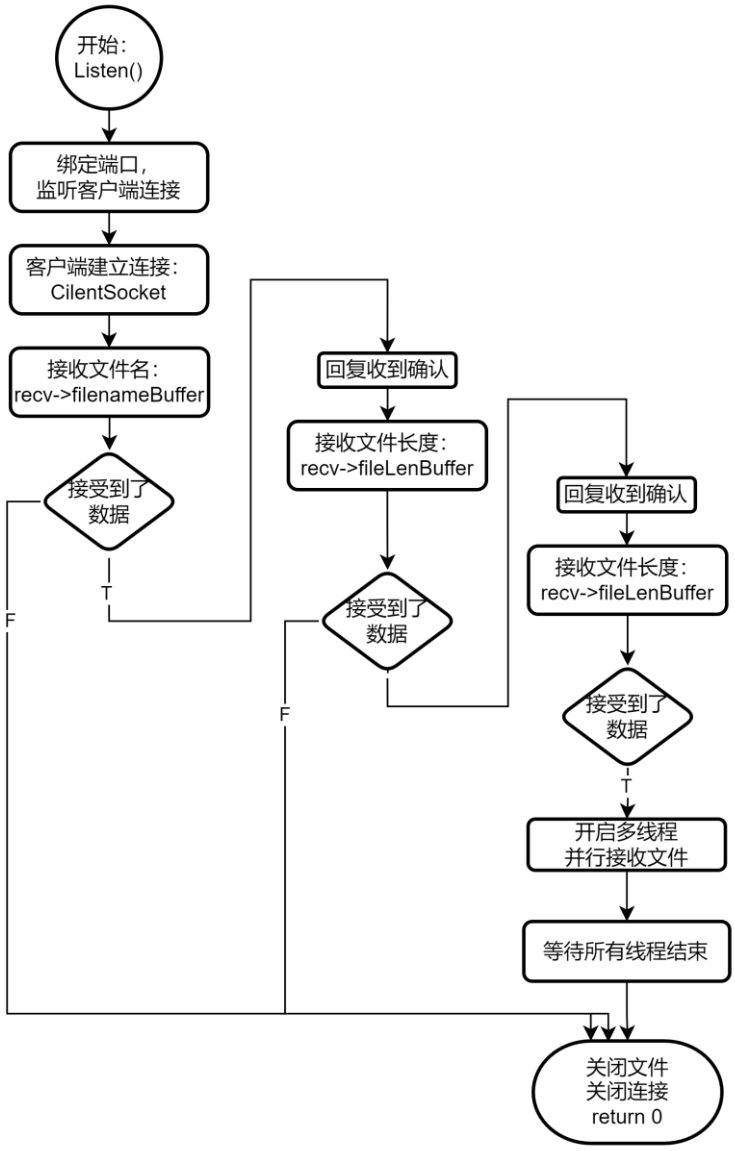

图 1 端口监听模块流程图

核心代码：

```
int FileTransfer::Listen()
{
    int iResult = 0;
    SOCKET ListenSocket = INVALID_SOCKET;
    SOCKET ClientSocket = INVALID_SOCKET;
    //监听并等待用户连接
    ……
    //用户已经连接
    ……
    //得到了 ClientSocket 连接

    //连接建立之后应该开始协商=================
    //char recvbuf[DEFAULT_BUFLEN] = { 0 };
    char filenameBuffer[DEFAULT_BUFLEN] = { 0 };
    char fileLenBuffer[DEFAULT_BUFLEN] = { 0 };
    char threadNumBuffer[DEFAULT_BUFLEN] = { 0 };
    char* recvStr = "hjy";
    int recvbuflen = DEFAULT_BUFLEN;
    //接收文件名
    iResult = recv(ClientSocket, filenameBuffer, recvbuflen, 0);
    if (iResult > 0) {
        send(ClientSocket, recvStr, (int)strlen(recvStr), 0);//同步
        //接收文件长度
        iResult = recv(ClientSocket, fileLenBuffer, recvbuflen, 0);
        if (iResult > 0) {
            send(ClientSocket, recvStr, (int)strlen(recvStr), 0);//同步
            //接收线程数
            if (iResult > 0) {
                int threadNum = std::stoi(threadNumBuffer);
                // 将字符串转换为整数
                //开始接收
                writeToLogFile("server:等待接收...TODO");
                FILE* fp = NULL; // 打开输入文件
                if (fopen_s(&fp, filenameBuffer, "wb") != 0) {
                    // 打开文件失败
                    writeToLogFile("无法打开文件\n");
                    return 1;
                }
                else {
                    // 将文件指针移动到指定位置
                    long fileLen = atol(fileLenBuffer); // 转换为长整数
                    if (fseek(fp, fileLen - 1, SEEK_SET) != 0) {
                        printf("无法移动文件指针\n");
                        fclose(fp);
                        return 1;
                    }
                    // 写入一个字节，将文件长度扩展到指定长度
                    if (fwrite("", 1, 1, fp) != 1) {
```

```
                    printf("无法写入文件\n");
                    fclose(fp);
                    return 1;
                }
                fclose(fp);
            }
            std::vector<std::thread> threadQueue;//线程队列
            // 启动线程
            for (int i = 0; i < threadNum; i++) {
                threadQueue.emplace_back(&FileTransfer::receive,
                            this, filenameBuffer, portNum[i]);
            }
            // 同步
            send(ClientSocket, recvStr, (int)strlen(recvStr), 0);
            writeToLogFile("server:已同步");
            // 等待所有线程执行完毕
            for (auto& threadObj : threadQueue) {
                if (threadObj.joinable()) {
                    threadObj.join();
                }
            }
            fclose(fp);
        }
    }
}
return 0;
}
```

②文件接收模块（主要模块）

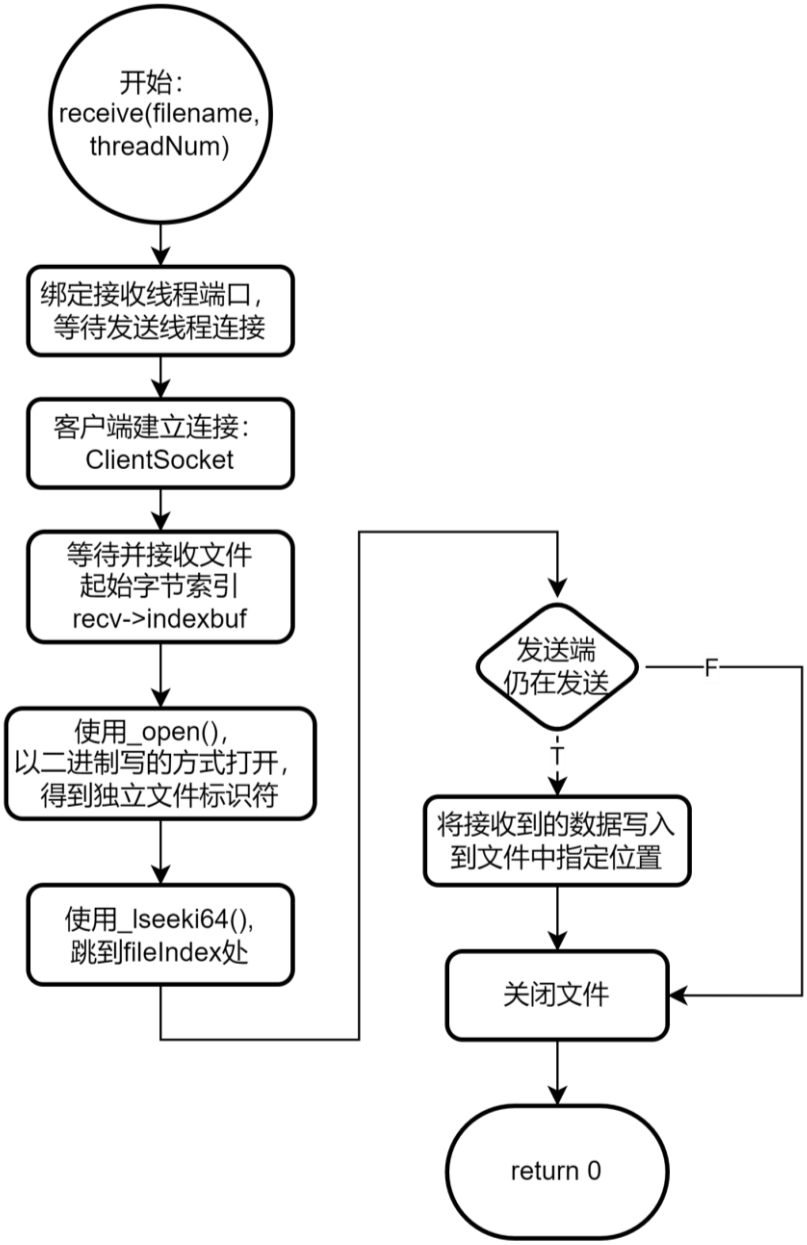

图 2 文件接收模块流程图

```
int FileTransfer::receive(const char* filename,int port)//接收
{
    SOCKET ListenSocket = INVALID_SOCKET;
    SOCKET ClientSocket = INVALID_SOCKET;
    //监听并等待发送线程连接
    ……
    //发送线程已经连接
    ……
    //关闭监听连接

    char recvbuf[DEFAULT_BUFLEN] = { 0 };
```

```
    char indexbuf[DEFAULT_BUFLEN] = { 0 };
    int recvbuflen = DEFAULT_BUFLEN;
    int fileLen = 0;
    int haveRecvLen = 0;
    // server:等待接收文件字节下标..
    recv(ClientSocket, indexbuf, recvbuflen, 0);//接收文件下标
    removeNewline(indexbuf);
    // server:接收到了文件字节下标
    send(ClientSocket, "hjy", (int)strlen("hjy"), 0);//同步
   //处理接收到的起始字节索引
    long int fileIndex = atoi(indexbuf);
    //打开文件，循环接收并写入
    //得到独立文件描述符，指向同一文件的指针之间不会相互影响
    int fd = _open(filename, _O_WRONLY | _O_BINARY);
    // 设置文件指针位置为偏移地址
    __int64 newPosition = _lseeki64(fd, fileIndex, SEEK_SET);
    if (fd == -1) {
        // 处理文件打开失败的情况
    }
    else {//文件可以打开
        int i = 0;
         while ((iResult = recv(ClientSocket, recvbuf, recvbuflen, 0)) > 0)
        {
            haveRecvLen += iResult;//记录接收到的文件长度
            // server:TCP 接收到数据!
             //开始写入
            int bytesWritten = _write(fd, recvbuf, iResult);
        }
        // 关闭文件描述符
        _close(fd);
        if (iResult == 0)
                //正常关闭连接
        else {
            //错误处理
            closesocket(ClientSocket);
            WSACleanup();
            return 1;
        }
    }
     return 0;
}
```

（2）客户端

①连接模块（辅助模块）

该模块属于辅助模块，仅用于建立协商连接，建立之后的工作交给其他

模块进行处理。

②文件选择模块（辅助模块）

该模块属于辅助模块。仅用于文件的选择，方便用户可视化的选择需要发送的文件。

③文件发送模块（主要模块）

包括两个函数，一个是用于协商的 muliThreadTrans，另一个是用于文件传输的 sendData。

i.函数 1：muliThreadTrans

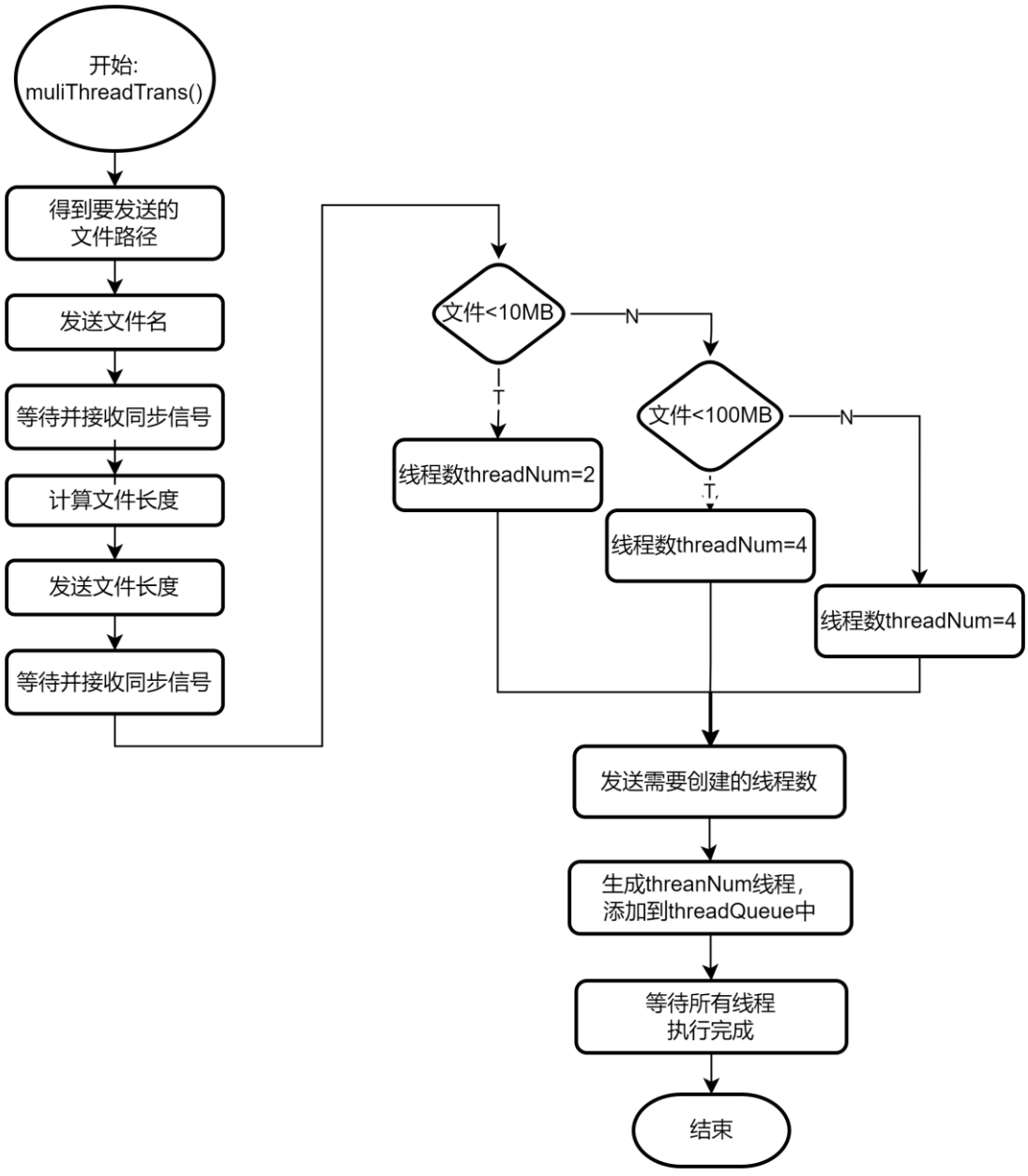

图 3 函数 muliThreadTrans 流程图

```
void FileTransfer::muliThreadTrans(char* filePath) {
int iResult = 0;
char* sendbuf = NULL;

//处理得到文件路径
……放到 sendbuf 中
char recvbuf[DEFAULT_BUFLEN] = { 0 };
int recvbuflen = DEFAULT_BUFLEN;

//发送文件名----------------------------------------------------
iResult = send(ConnectSocket, sendbuf, (int)strlen(sendbuf), 0);
//接收同步信号
iResult = recv(ConnectSocket, recvbuf, recvbuflen, 0);
//发送文件名 end------------------------------------------------

//发送文件长度--------------------------------------------------
//打开文件，循环读取并发送
FILE* fp;//文件指针变量
// 打开文件并获取文件描述符
if (fopen_s(&fp, filePath, "rb") != 0) {
    pStaticText->SetWindowText(CString("无法打开文件。"));//可视化显示
}
FILE* fplen = fp;//文件指针
long int fileSize = getFileSize(fplen);
this->totalLen = fileSize;//文件总长度
fclose(fp);
char fileLength[512]; // 假设目标字符串的长度不超过 20 个字符
sprintf_s(fileLength, sizeof(fileLength), "%ld", fileSize);
//发送文件长度
send(ConnectSocket, fileLength, sizeof(fileLength), MSG_OOB);
//接收同步信号
iResult = recv(ConnectSocket, recvbuf, recvbuflen, 0);
//发送文件长度 end----------------------------------------------


//发送需要的线程数----------------------------------------------
int threadNum = 2;
//根据文件大小选择 threadNum 大小
long int  tenMb = std::stol("10485760");
long int threentyMb = std::stol("31457280");
if (fileSize < tenMb) {
    threadNum = 2;
}
else if (fileSize > threentyMb) {
    threadNum = 8;
}
else {
    threadNum = 4;
}
```

```
    char threadNumStr[20]; //足够存储整数转换后的字符串
    sprintf_s(threadNumStr, sizeof(threadNumStr), "%d", threadNum);
    addNewline(threadNumStr);//末尾加上'\n'
    //发送线程数
    send(ConnectSocket, threadNumStr, sizeof(threadNumStr), MSG_OOB);
    //发送需要的线程数 end----------------------------------------------


    //计算文件指针的相对位置
    int threadSize = fileSize / threadNum;//每个线程需要发送的字节数
    //最后一个线程需要发送的字节数 8 + 2 = 10
    int threadSizeLast = threadSize + fileSize % threadNum;
    int threadSizeLastIndex = threadNum - 1;//最后一个线程的索引
    //最后一个线程的起始位置
    int threadSizeLastIndexStart = threadSizeLastIndex * threadSize;
    std::vector<std::thread> threadQueue;
    for (int i = 0; i < threadNum; i++) {
        int len = (i == threadSizeLastIndex) ? threadSizeLast : threadSize;
        int startIdx = i * threadSize;
        // 发送，创建一个新线程并添加到线程队列
        threadQueue.emplace_back(&FileTransfer::sendData, this,
                                 filePath, portNum[i], startIdx, len);
    }
    // 等待所有线程执行完毕
    for (auto& threadObj : threadQueue) {
        if (threadObj.joinable()) {
            threadObj.join();
        }
    }
}
```

ii. 函数 2：sendData

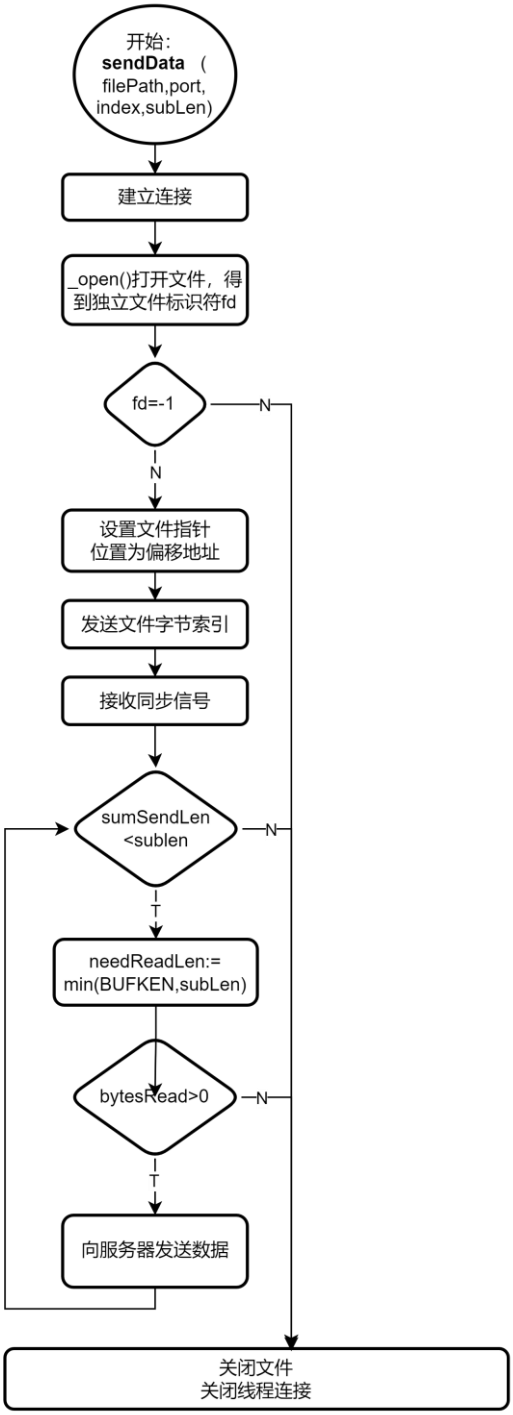

图 4 函数 sendData 流程图

```
int FileTransfer::sendData(const char* filePath,int port,int index,int
subLen)
{
    //建立连接--------------------------------
    SOCKET SendtSocket = SOCKET_ERROR;//已建立的连接
    ……
```

```
    //连接已经建立

    int fd = _open(filePath, _O_RDONLY | _O_BINARY);
   if (fd == -1) {}
    else {//打开成功
        // 设置文件指针位置为偏移地址
        __int64 newPosition = _lseeki64(fd, index, SEEK_SET);
        char fileIndex[DEFAULT_BUFLEN] = { 0 };
        char buffertmp[DEFAULT_BUFLEN] = { 0 };
        sprintf_s(fileIndex, sizeof(fileIndex), "%ld", index);
        //发送文件字节索引
        send(SendtSocket, fileIndex, sizeof(fileIndex), MSG_OOB);
        //接收文件字节索引--同步
        recv(SendtSocket, buffertmp, DEFAULT_BUFLEN, 0);
        unsigned char buffer[DEFAULT_BUFLEN];
        size_t bytesRead;
        int sum = 0;
        int needReadLen = 0;
        while (sum < subLen)//未结束
        {
            needReadLen = (DEFAULT_BUFLEN < subLen) ? DEFAULT_BUFLEN :
                                                                subLen;
            bytesRead = _read(fd, buffer, sizeof(buffer));

            if (bytesRead > 0) {
            //读到了内容就向服务器发送数据
            iResult = send(SendtSocket, (const char*)buffer, bytesRead, 0);
            }
            else {
                break;
            }
        }
        // 关闭文件描述符
        _close(fd);
        //fclose(fp);//关闭文件
        writeToLogFile("client:发送结束");
        //结束发送
    }
    closesocket(SendtSocket);
    return 0;
}
```

④详细内容显示模块（辅助模块）

仅用于文件传输信息的显示

六、关键问题及其解决方法

关键问题 1：多线程文件传输通信双方如何确定所需要的线程数量？

这里采用的是“预先划分端口”+“运行时动态划分”的方法，我们预先按照一定的顺序约定好我们可以使用哪些端口，然后再发送文件时根据文件的大小，来决定使用几个端口来进行文件传输。

但是更好的做法应该是预先约定一个起始端口，例如“9999”端口，然后由发送方根据文件大小决定所需要的线程数，然后与服务器每次动态的协商一个端口，服务器返回该端口的可用状态同时启动新的线程监听当前端口， 客户端得到可用端口后就记录下来，不可用就将查询的端口号加一，继续向服务端查询该端口是否可用，直到得到了所需要的端口数，就开始与服务器建立连接，从而传输数据。

```
//发送需要的线程数-----------------------------------------------------
    int threadNum = 2;
    //TODO:根据文件大小选择 threadNum 大小
    long int  tenMb = std::stol("10485760");
    long int threentyMb = std::stol("31457280");
    if (fileSize < tenMb) {
        threadNum = 2;
    }
    else if (fileSize > threentyMb) {
        threadNum = 8;
    }
    else {
        threadNum = 4;
    }

    char threadNumStr[20]; // 假设足够存储整数转换后的字符串

    sprintf_s(threadNumStr, sizeof(threadNumStr), "%d", threadNum);
    //发送线程数
    send(ConnectSocket, threadNumStr, sizeof(threadNumStr), MSG_OOB);
    ------------------------------------------------------
```

关键问题 2：通信双方协商时发送的消息明明及时的刷新了，为什么对方还是会一起接收？

这个问题其实很简单，虽然发送方及时的把信息发送了出去，但是由于接收方处理的速度和时间存在差异，会导致上一次的协商信息还没有读取，后一次的协商信息就已经到来了，这样就会导致信息在缓冲区拼接到一起，导致读取出现问题。

我们对于这个问题的解决方法是让服务端每次接收到消息并读取之后给客户端发送一个“同步确认”的回应，客户端接收到这个确认之后再发送下一个消息，通过这样的方式解决了协商上信息错误拼接的问题。

服务端：

```
//接收文件名
iResult = recv(ClientSocket, filenameBuffer, recvbuflen, 0);
if (iResult > 0) {
    //回应同步消息
    send(ClientSocket, recvStr, (int)strlen(recvStr), 0);
    //接收文件长度
    iResult = recv(ClientSocket, fileLenBuffer, recvbuflen, 0);
    if (iResult > 0) {
        //回应同步消息
        send(ClientSocket, recvStr, (int)strlen(recvStr), 0);
        ……
        其他处理代码
    }
}
```

关键问题 3：如果解决多进程对同一个文件的读（写）问题？

对于发送端来说是如何解决对同一个文件的读问题，对于发送端来说是如何解决对同一个文件的写问题。

我在第一次实现的时候使用的是 c 语言的 FILE 类型的文件指针来对文件进行读、写，这时出现了文件发送了，但是缺少了部分内容，经过调查发现多个 FILE 文件指针共享的同一个文件描述符，共享同一个结构体，在我使用 fseek 函数定位 FILE 指针的时候其他指针也会发生变化，这就解释了为什么会导致多线程发送时文件缺少内容。

为了解决这个问题我使用了_open()函数来打开文件，这样可以得到独立的文件描述符，这时再指定位置读写就不会影响其他指针了，能够实现多线程的读写操作。

发送时读文件：

```
int fd = _open(filePath, _O_RDONLY | _O_BINARY);
if (fd == -1) {

}
else {//打开成功
    // 设置文件指针位置为偏移地址
```

```
    __int64 newPosition = _lseeki64(fd, index, SEEK_SET);
      后续读文件、发送操作
}
```

接收时写文件：

```
//打开文件，循环接收并写入
int fd = _open(filename, _O_WRONLY | _O_BINARY);
// 设置文件指针位置为偏移地址
__int64 newPosition = _lseeki64(fd, fileIndex, SEEK_SET);
if (fd == -1) {
    // 处理文件打开失败的情况
}
else {//文件可以打开
    后续操作
}
```

七、设计结果

（一）设计结果

1、能够实现单线程、多线程文件传输文本、图片、音频和视频等。

（1）文本传输

①选择文件界面：

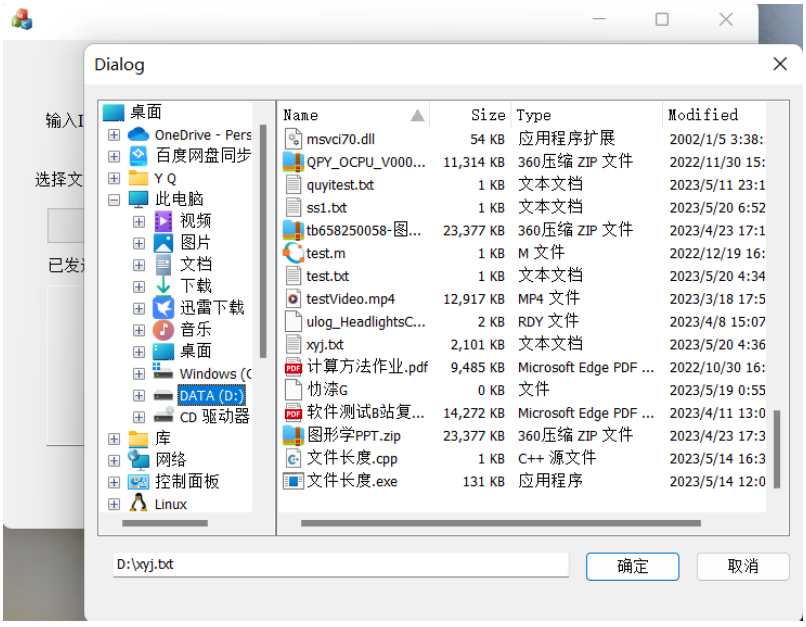

图 5 文件选择界面

②完成发送

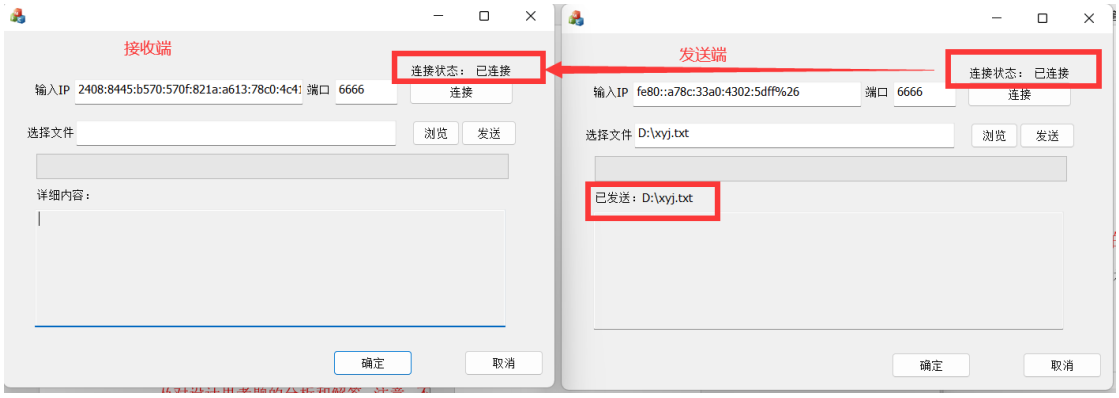

图 6 完成发送界面

③查看文件字节数一致：

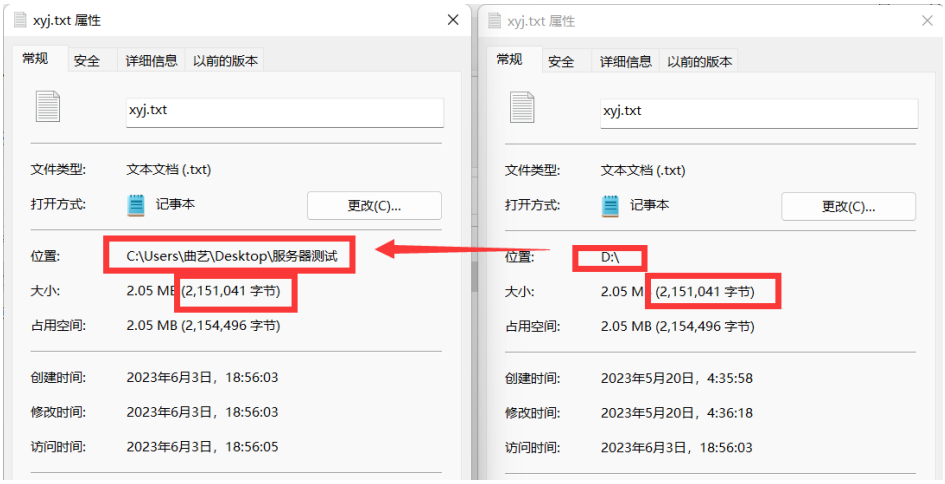

图 7 文件对比结果

④内容正确无误：

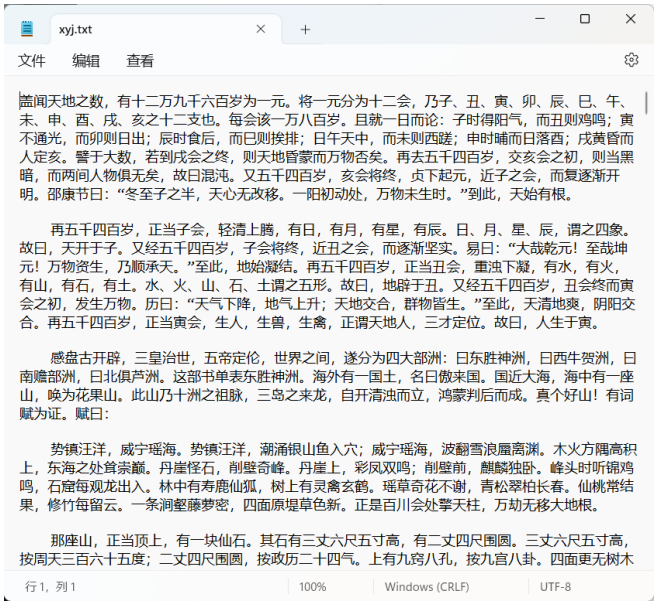

图 8 内容验证

（2）视频传输

①选择文件

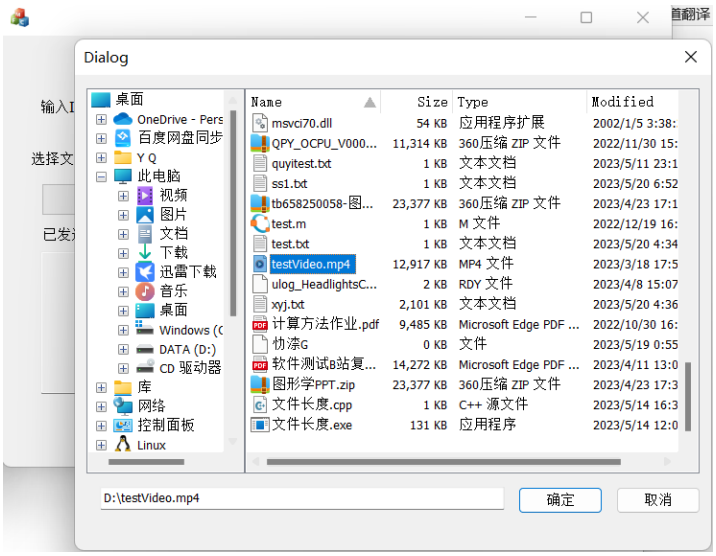

图 9 选择文件 testVideo.mp4

②正在发送

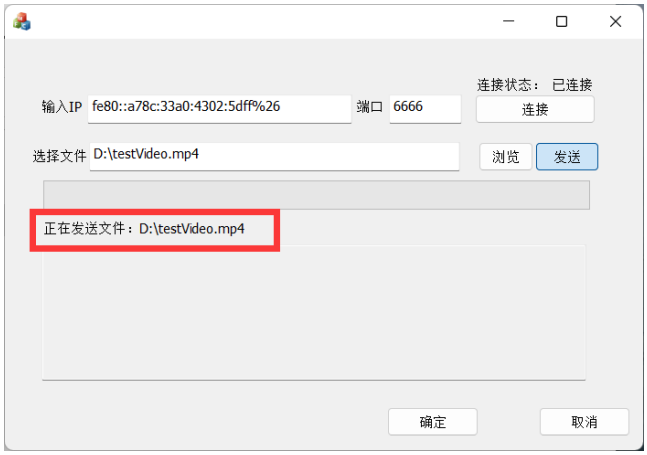

图 10 正在发送文件

③已发送

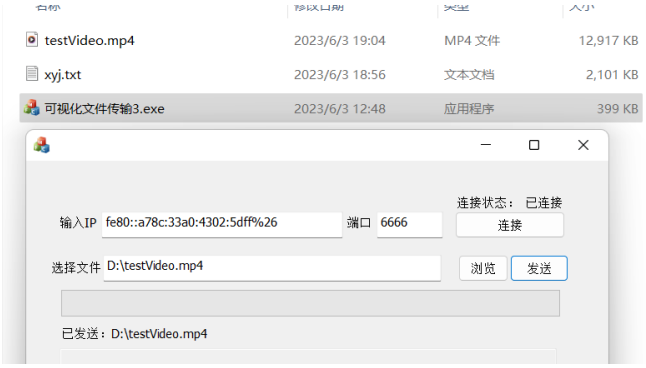

图 11 已发送

④对比接收到的文件字节数，文件接收正确

图 12　testVide.mp4 文件对比

⑤能够正常播放

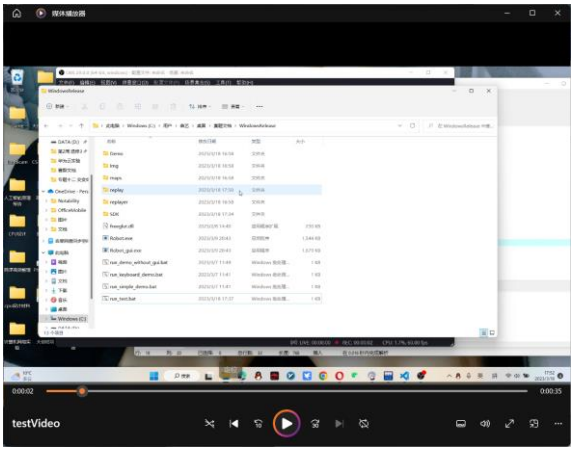

图 13　视频播放验证

（3）大文件传输验证

图 14　文件属性

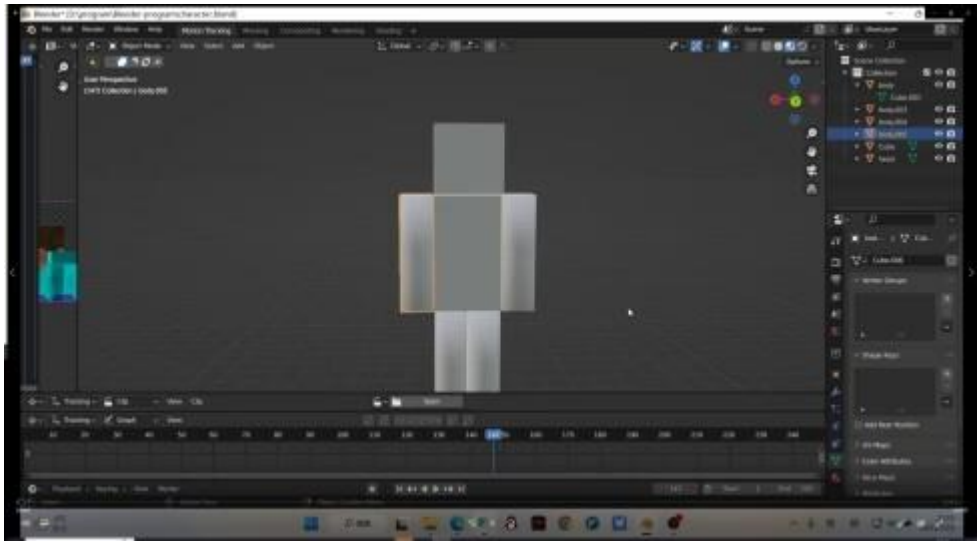

图 15 大文件正确性验证

2、UDP 文件传输测试

（1）传输后的文本文件打开乱码，音视频无法正常打开，原因是 UDP 是一个不保证一定到达的协议，这种协议对于实时性较高的音视频传输比较有优势，但是对于文件传输要保证文件的完整性，就需要在接收方进行检测，根据发来数据的字节索引和长度，统计文件还有那些部分是缺失的，向客户端发送重发消息，在软件的层面来保证文件的完整性。

（2）接收端缓冲池小于发送端缓冲池会导致改数据包丢失，如果没有拥塞控制的话，发送端继续使用原来大小的报文发送会导致接收端永远叶无法接收到数据，导致程序无法继续执行。

3、特定带宽下线程与速率的关系

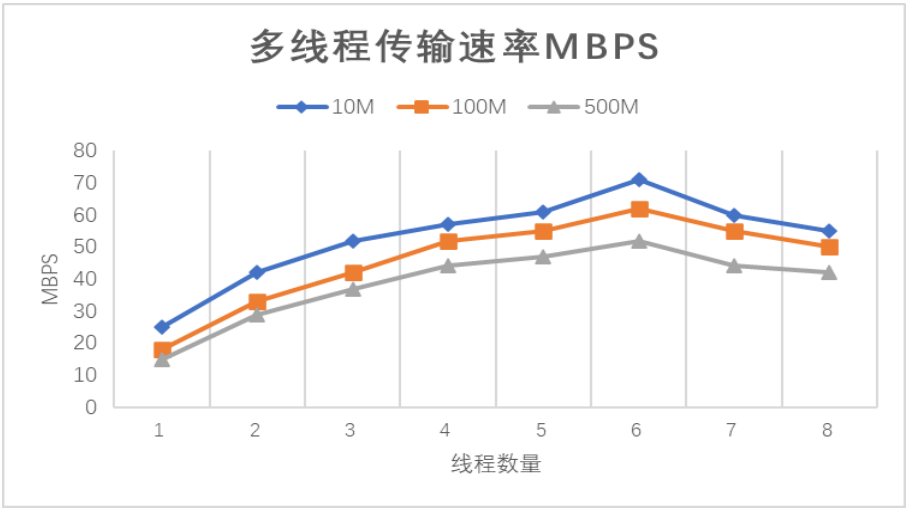

图 16 特定带宽下线程与速率的关系

4、异常控制

（1）用户操作的鲁棒性

当两个用户之间已经建立了连接之后，被动连接的这一端的用户再次点击“连接”按钮，将不会触发连接时间，也不会向新的地址发送连接请求，同时发送方也是如此，发送方在连接建立之后再次点击连接不会重复连接

（2）连接端口选择的鲁棒性

预先约定一个起始端口，例如“9999”端口，然后由发送方根据文件大小决定所需要的线程数，然后与服务器每次动态的协商一个端口，服务器返回该端口的可用状态同时启动新的线程监听当前端口， 客户端得到可用端口后就记录下来，不可用就将查询的端口号加一，继续向服务端查询该端口是否可用，直到得到了所需要的端口数，就开始与服务器建立连接，从而传输数据。

4、如何提高套接字传输的速率，以及如何加强传输的稳定性。

对于提升套接字传输速率这一方面我们可以在传输大文件之前先启动一个测试程序，检测出在当前带宽下使用某一个线程数进行数据传输可以达到最高的传输速率；检测完了之后才进行大文件传输，但是对于小一点的文件，就不需要进程检测，可能检测的时间就足以使用默认的线程数将文件传送完了。

对于稳定性提升，可以通过加入断点保存机制，在连接中断的情况下保存文件接收情况，并等待客户端上线进行断点续传。

（二）思考题

1、套接字有基于阻塞和非阻塞的工作方式，试问你编写的程序是基于阻塞还是非阻塞的？各有什么优缺点？

答：

我编写的程序在用户发送文件的时候是阻塞式工作方式，会在用户点击发送之后等待文件发送结束之后，用户才能够进行其他的操作，由于我做的程序是文件传输，用户在等待文件发送的时候也没有其他事情可以做，所以可以理解。

我程序中的套接字使用的是阻塞的工作方式，单个线程中如何发送缓冲池满了，就会阻塞等待，等待缓冲池发送出去清空之后再继续发送。这样的好处是比较直观，容易理解，但是它的缺点也很明显，就是会被阻塞，效率不如非阻塞式。

对于非阻塞套接字在网络和数据没有准备好的情况下，也不影响程序的正常执行，能够提高程序的响应性和资源利用率，同时有利于并发处理。

2、如何将上述通信改为非阻塞，避免阻塞？

答：

可以按照下述方案进行修改：

①设置套接字为非阻塞模式：在创建套接字后，使用 fcntl()函数将套接字设置为非阻塞模式。这样可以确保在没有数据可读或写时，套接字操作不会阻塞程序执行。

②使用循环或事件驱动机制：在传输文件的过程中，使用循环或事件驱动的方式来检查套接字的状态，包括是否可读和是否可写。

③检查可读性：使用 select()或 poll()等函数来检查套接字是否有可读数据。如果可读，可以调用 recv()函数读取数据并进行处理。

④检查可写性：使用 select()或 poll()等函数来检查套接字是否可写。如果可写，可以调用 send()函数将数据发送到套接字。

⑤处理非阻塞状态：在非阻塞模式下，当套接字操作返回 EAGAIN 或 EWOULDBLOCK 错误时，表示当前操作会阻塞，此时可以选择等待一段时间后重新尝试，或者进行其他操作。

⑥完成文件传输：通过循环或事件驱动机制，直到文件传输完成或出现错误，可以根据具体需求进行相应的处理。

3、在传输前能否先将要传输的文件的相关属性现行报告给对方，以便对方判断是否接受该文件的传输？

答：

这个是需要的，我们这里已经实现先传输文件属性，传输的文件属性是文件大小，我们还可以传输一些其他的属性，例如文件类型、文件

的修改日期、文件的创建者等，用户可以根据自己的需要来决定这个文件是不是自己想要的，如果不是，用户有权利拒绝接收该文件。

4、了解并熟悉多线程工作原理，试编写基于多线程的网络文件传输程序。

答:

我们已经实现了多线程文件传输，能够基于 IP6 网络进行各类文件传输。

八、软件使用说明

本程序仅支持英文路径的文件传输

1、首先将程序拷贝一份，作为服务端用户

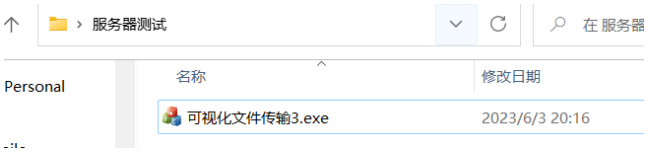

图 17 拷贝服务端

2、然后启动服务端

图 18 启动服务端

3、启动客户端。

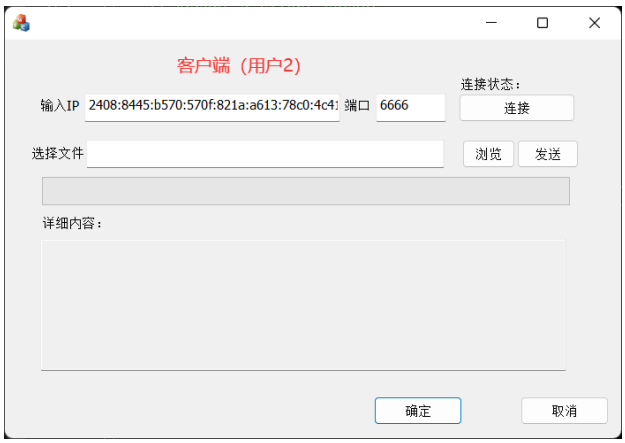

图 19 启动客户端

4、在 IP 地址栏中输入自己的 IPV6 地址，点击连接。

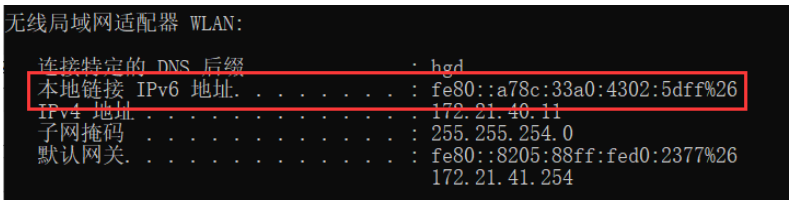

图 20 获取服务器 IPV6 地址

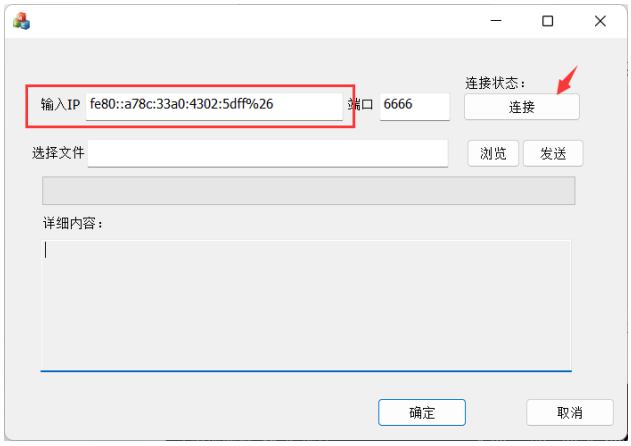

图 21 填入服务器 IPV6 地址

5、连接成功两个用户都会显示“已连接”。

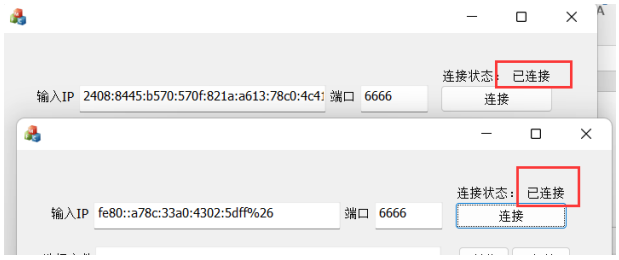

图 22 连接状态验证

6、客户端点击“浏览”进入文件选择窗口，可以在窗口左侧逐级查看本机文件。

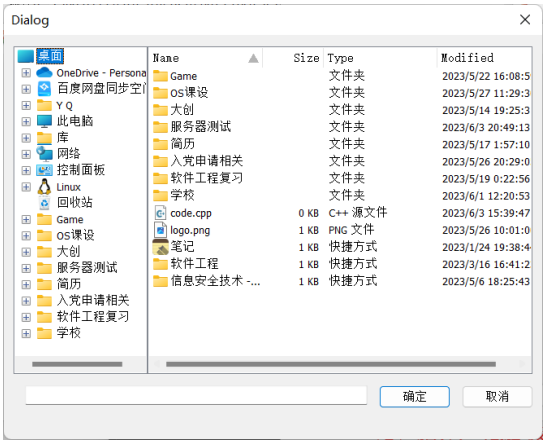

图 23 浏览文件

7、在右侧会显示当前文件夹下的文件内容，在右侧窗口点击文件会在输入框中显示选择的文件路径，点击“确定”按钮即选择的该文件，会自动返回主界面

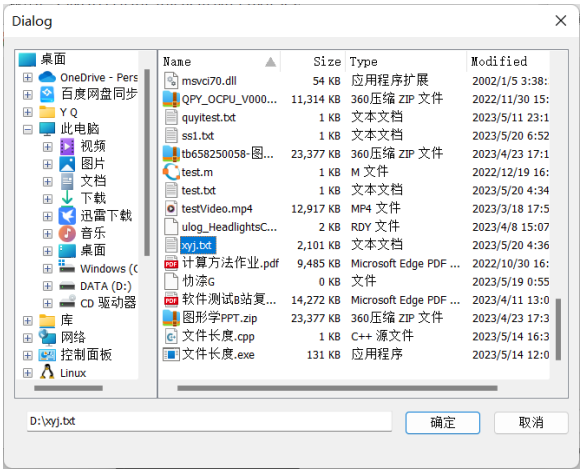

图 24 选择文件

8、点击“发送”按钮开始发送文件。下方会显示文件的发送状态，如图所示已完成文件传输。

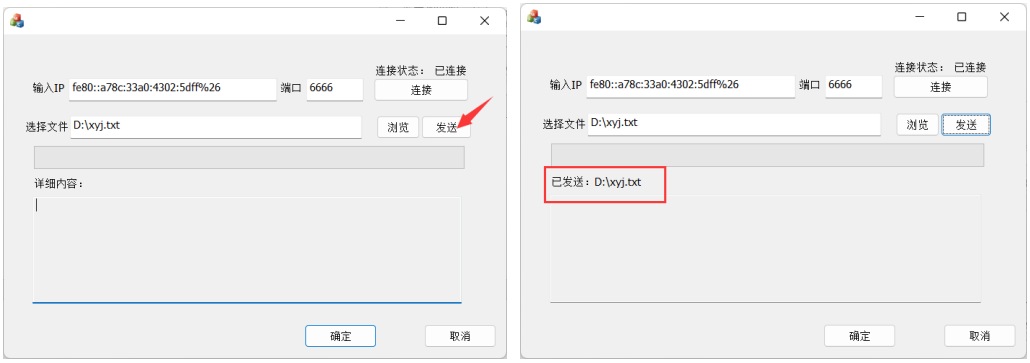

图 25 文件已发送

服务器接收到到文件就放在自己的同级文件夹下：

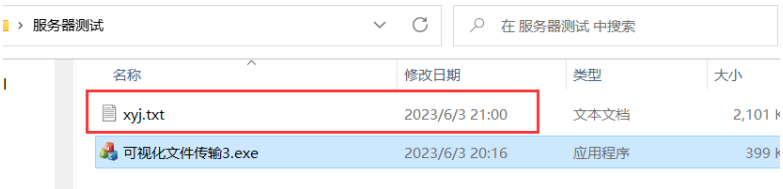

图 26 服务端接收到的文件

九、参考资料

[1] Microsoft Corporation. "Getting Started with Winsock." Microsoft Documentation.

[2] Microsoft Corporation. "MFC (Microsoft Foundation Class Library)

Documentation." Microsoft Docs.
[3] Beej. "Beej's Guide to Network Programming".
[4] Douglas E. Comer. "Internetworking with TCP/IP, Vol. 1: Principles, Protocols, and Architecture (6th Edition)". Pearson, 2013.
[5] W. Richard Stevens, Bill Fenner, Andrew M. Rudoff. "UNIX Network Programming, Volume 1: The Sockets Networking API (3rd Edition)". Addison-Wesley Professional, 2003.

十、验收时间及验收情况

验收时间：2023.5.20

（1）小组每人所做的工作。

我和队友在开始编写代码之前我们商量了为了尽可能的模拟真实的网络情况，模拟不同的主机和程序之间的通信，所以我们选择使用 IPV6 实现在真实的公网传输数据，而我们两个则是每人各自实现一个文件传输程序，我们首先设计了程序之间基于网络的通信流程，然后使用相同的传输协议和沟通方式进行实现，但是我使用 C/C++语言进行开发，他使用 JAVA 进行开发，我们两个都各自实现了一个可以正常工作的文件传输软件，能够实现同种语言软件之间的文件传输，也能够实现两种语言编写的程序之间的文件传输，还能够使用 IPV6 在真实的网络环境中实现文件传输，并且保证文件的完整性。

（2）验收时的演示情况

①正确的演示了对文本、音频、视频的相互传输

②正确的演示了大文件的传输

③演示了我们两个程序之间能够进行正常的文件传输

④我们各自演示了所编写程序的运行过程

（3）与老师的交流情况

我们首先向老师讲述了我们两个的分工和实现思路，然后进行了上述演示，然后老师向我们提问，问我们多线程文件传输是如何实现的。

老师首先让我回答，我回答说我们的实现方式首先是向服务器发送文件的名字和文件的大小，根据文件的大小来与服务器进行连接端口的协商，然后建立连接，建立连接之后建立多个发送文件的线程，在线程中中使用_open()函数来得到独立的文件描述符，根据线程创建时传进来的字节索引位置开始读取字节并发送，发送端同样也是建立多个线程进行连接接收，接受的时候也是使用_open()来得到独立的文件描述符，然后跳到指定的位置开始写入，每个线程写入的位置

不会重叠，这也就保证了多线程直接不会出现同时读和写入同一个内存。

然后我的队友回答这个问题，因为我们是一起完成的项目，因此回答的内容基本一致。

然后老师问我你是怎么保证文件可以做到多线程的读和写的，我回答说我使用 C/C++采用的时独立文件描述符，与 FILE 指针不同，直接之间不会相互影响，多线程每一个线程都会各自得到一个独立的文件描述符，从而保证了文件读、发送、写的并行处理。

十一、设计体会

通过本次课程设计加深了我对计算机网络的认识和理解，也算是初步接触并完成了第一个基于网络的软件，并且真的能够在公网上进行文件传输，虽然传输速度没有很快，但是要比 QQ 的龟速要快，可以和同学之间传输一些文件，有一些使用价值。

另外，通过本次课程设计，我学会了 windows 平台基于 winsock 的网络编程，我在学习的过程中发现很多 CSDN 的内容讲的都不够详细，不够准确，因此我主要参考的时微软的 winsock 官方文档，在程序设计的时候使用 C/C++进行开发，虽然语言比较低层，没有那么多已经封装好的库，需要自己来造轮子，但也正是这个造轮子的过程，让我对 TCP/UDP 真正在应用的时候是什么样的、应该使用什么样的方式来保证它正常通信都有了更加深入的理解，=。

明白了在实现基于网络编程的软件的时候，不仅仅要考虑程序自身的健壮性，还要考虑通信双方在存在延迟的时候出现的各种问题，既要保障信息的正确到达，也要保证信息的正确读取，就像我们最开始协商时遇到的信息拼接问题，通信双方的程序都没有问题，问题出现在了基于网络传输没有考虑处理的发送的延迟，这才导致了信息的错误拼接，于是借鉴人类沟通都需要一问一答的形式，采用了每次协商都需要回应才能继续进行的方式，这样就保证了信息的正确到达。

最后在多线程文件传输功能的开发中，遇到了文件指针共享文件标识符的问题，我们在最开始也没有想到有这样的问题，经过了漫长的 Debug 终于发现问题的根源，于是更换了_open()获得独立文件标识符解决问题。

通过这次课程设计，我不仅学到了具体的网络编程技术，还培养了问题分析和解决的能力。我对计算机网络有了更深入的认识和理解，并且掌握了实际应用中的一些注意事项和技巧。这次经历让我受益匪浅，为我今后的学习和工作打下了坚实的基础。