Lab3-3 实验报告

UDP 可靠协议传输

1813265 李彦欣

目录

实验要求

一、具体要求：

利用数据报套接字在用户空间实现面向连接的可靠数据传输，功能包括：建立连接、差错检测、确认重传。流量控制采用停等机制，完成给定测试文件的传输。

在以上的基础上，将停等机制改成基于滑动窗口的流量控制机制，采用固定窗口大小，支持累积确认，完成给定测试文件的传输。

在以上的基础上，选择实现一种拥塞控制算法，也可以是改进的算法，完成给定测试文件的传输。

二、开发环境：

（1）集成开发环境：Visual Studio 2019

（2）开发语言：c++

（3）操作系统：Windows10

协议设计

一、主要协议内容设计：

- **数据内容说明：**

端口	数据报文

其中数据报文的格式：（序列号表示数据包的序号）

序列号	长度	校验和	数据

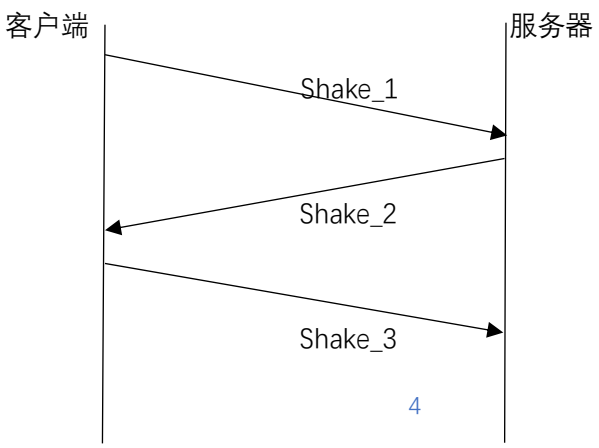
【数据包的具体长度说明】：

本次实验更改了一下协议设计，**增加端口号**，因为增加了路由器，原先的服务器和客户端就不能再是一样的端口了，应该有所区分，于是在发送数据报时，就先发送了一下端口号，对于是否经过路由程序进行区分。一个数据包的长度是有限制的，对于不是最后一个传输的数据包来说最多数据长度为 253 个字节，即数据报文为 256 个字节。对于传输段最后的不完整的数据来说则按具体长度发送。 对于非结尾数据包，标志位的倒数第 4 位为 1；对于结尾数据包，标志位的倒数 4，5 位为 1；因为文件传输期间，可能发送的整个数据段不止一个，因此序号位需要在连接未断开时不断递增，否则接收方可能会混淆不同的阶段的数据。

- **握手过程：**

  本次实验由于经过路由程序时，出现了丢失建立连接时所发送的数据包，于是在本次实验的程序中，删去了握手和挥手过程，直接绑定。还是对之前的握手和挥手过程进行展示说明。

  三次握手过程的大致流程图如下：



步骤一：客户端发送第一次握手信号 shake_1。

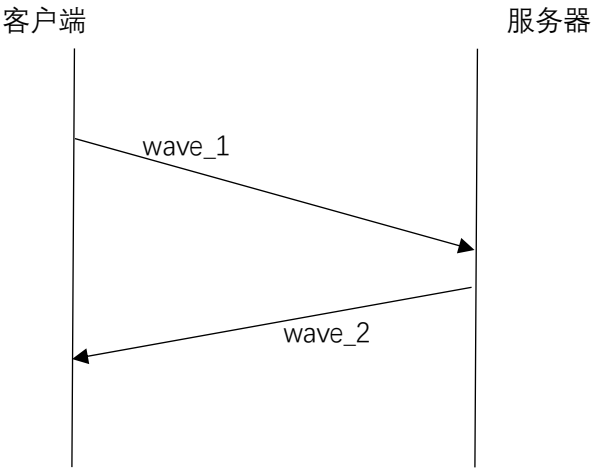
步骤二：服务器端接收后需计算校验和，若收到 shake_1 且校验和等于 0，则开始发送第二次握手信号 shake_2。

步骤三：客户端收到服务器端发来的 shake_2，并计算出校验和正确，开始发第三次握手 shake_3。

步骤四：服务器端接收到第三次握手信号，且计算出校验和正确，那么握手过程成功。

- **挥手过程：**

  两次挥手过程的大致流程图如下：



步骤一：客户端发送第一次挥手信号 wave_1。

步骤二：服务器端接收后需计算校验和，若收到 wave_1 且校验和等于 0，则开始发送第二次挥手信号 wave_2。

步骤三：客户端收到服务器端发来的 wave_2，并计算出校验和正确，则挥手过程成功。

- **校验和的计算：**

使用上课 PPT 中讲到过的 checksum 函数进行差错检测：

【发送端】：

- 产生伪首部，校验和域段清 0，将数据报用 0 补齐为 16 位整数倍

- 将伪首部和数据报一起看成 16 位整数序列
- 进行 16 位二进制反码求和运算，计算结果取反写入校验和域段

【接收端】:

- 产生伪首部，将数据报用 0 补齐为 16 为整数倍
- 按 16 位整数序列，采用 16 位二进制反码求和运算
- 如果计算结果位全 1，没有检测到错误；否则，说明数据报存在差错

在发送包（包括 ACK）的时候都需要进行校验和的计算，来确保发包是否出错。

注：本次 lab3-3 的协议部分跳转第九页

lab3-1 的功能实现的协议部分

➢ **确认重传:**

具体的机制意思是在发送一个数据包之后，就开启一个定时器，若是在这个时间内没有收到发送数据的 ACK 确认报文，则对该报文进行重传，在达到一定次数还没有成功时放弃并发送一个复位信号。主要包括累计确认、超时时间计算、快速重传等几个方面。在 3-1 中我采用的是超时重传。

【超时重传】:

对于当前数据包开启一个计时器，如果超时则进行重传，并且选择适当的序列号。实现出来就体现在当数据包丢包时就会进行超时重传；并且在 ACK/NAK 进行丢包的时候也能进行数据包的超时重传。

在文件传输过程中的超时的设置:

因为在服务端 recvfrom 是进行阻塞接受的，我们先使用库函数将其阻塞时间进行设置，在我们的 TIMEOUT 范围之内。

【出错重传】:

出错重传分为两种:

- 自身的数据包重传，当对方发来的的 NAK 数据包被接收到的时候，我们需要进行重传；
- 接收到的 ACK 或者 NAK 数据包校验有问题，无法恢复，则进行重传之前的数据包。

（为了和之前保持一致，重传数据包的的标志位也和之前的标志位相同。）

（如果在挥手和握手期间传递的数据包出错，则从头进行整个挥手握手过程，保证连接信息完整性。）

（握手挥手过程中的丢包因为无法确认和检测，只能进行重新进行握手挥手实现，并且在挥手中进行超时次数的设置，超过次数认为对方完全断网，自己则进行资源回收后断网。）

- **ACK 和 NAK 数据包:（3-2 不涉及 NAK，原因见累计确认）**

当服务器接收到数据包之后，将对于数据进行校验和计算，对于"校验和"和"数据报文"进行统一的校验操作，其中的 ACK 和 NAK 数据包长度均为 3 字节（包含差错检测位，标志位和序列号，序列号位为其对应的接收包的序号。

如果校验和计算之后的结果是 0，那么证明没有错误产生，则缓存接收到的数据报文。之后向客户端发送“ACK”数据包，ACK 数据包的标志位的后两位为 1，其他位都为 0(即 0x03);

如果校验和计算之后的结果不是 0，那么证明出错了，那么发送“NAK”数据包，NAK 数据包的标志位后 3 位为 1，其他位都为 0(即 0x07)，表示接受的数据包出错。

- **传输的具体内容：**

首先我们会将文件名作为一个单独的数据段发送过去，之后发送整个文件的数据段，在接收端接收到两端数据之后，就可以根据文件名进行数据解析工作，或者在本地存储数据。

lab3-2 的补充协议部分

- **滑动窗口：**

该协议允许发送方在停止并等待确认前可以连续发送多个分组。由于发送方不必每发一个分组就停下来等待确认，因此该协议可以加速数据的传输。但它也受限于在流水线中未确认的分组不能超过最大窗口数。不同于 3-1 实现的停等协议的是，之前是一个一个连续地发送，相当于滑动窗口的窗口大小等于 1。

在发送端，窗口内的分组序号对应的数据分组是可以连续发送的。窗口内的数据分组有：

- 已发送但尚未得到确认
- 未发送但可以连续发送
- 已发送且已得到确认，但窗口中本序号的前面还有未得到确认的数据分组

滑动窗口法要求各数据分组按顺序发送，但并不要求确认按序返回。一旦窗口前面部分的数据分组得到确认，则窗口向前滑动相应的位置，落入窗口中的后续分组又可以连续发送。

在本次程序中，对于发送方我们设定一个大小为 WINDOW_SIZE 的序号窗口，其中的数据为已经发送确还未确认的。如果窗口未满，则我们可以继续发送数据包；如果窗口满了，则需要等待 ACK 确认后窗口的减小，之后才能发送数据包。

对于一个数据段的结束，因为窗口此时不能继续扩大，因此我们需要做特殊的标记处理，此时只允许缩小窗口等待全部数据包确认，则可以退出发送函数。

- **GO——BACKN**

整体的 GBN 协议设计，采用 while 循环来回调换 recv 和 send 进行实现，避免了使用线程导致的繁琐。具体含义是对于每一个发送的数据包，记录了他发送出去的时间，存储在一个队列里 timer_list 中，之后根据当前 clock()-最初发送的时间来判断队首的包是否发送超时，若超时则需要进行回退重发，并且清空队列；如果未超时收到包，则需要弹出队列队首，即更新计时器。

- **累计确认：**

由于使用累计确认策略，我们将不会在接收方进行 NAK 包的设定。只有当当前序号之前的所有包都接收到了，发送这个序号对应的 ACK 包，否则即使接受到了之后的包，也只会发送之前连续接受到的正确的包的 ACK。

- **整体实现思路：**

对于上层应用程序发送来的数据段，首先进行各个发送包的封装，开始流水线的式的进行数据的发送，在发送窗口未填满时，边发送边进行接收检测，查看对方的是否有累计确认状态码发送过来。对于一个累计确认状态的序号高于当前的 base 序

号的包，则进行发送窗口的滑动，并且更新定时器，继续数据包发送。在对方 ACK 包丢失的情况下，我们需要处理队列 timer_list 的出队个数，我们通过创建一个 bool 型的数组来实现在队列中的序号这样可以快速地进行查找。

其中窗口大小和 ip 地址和文件名都可以进行输入。

- **发送窗口是否已满的判断：**

因为我们设定的序列号只有一个 char 型大小(8 比特)，因此存在序号重复的可能，使得 ACK 序号和发送方窗口是否已满的判断很难通过 base 和 nextpacknum 判断，因此我们可以利用前面提到的 timer_list 的长度和变换进行判断，使得此问题可以得到处理。

- **传输时间和平均吞吐率：**

从调用开始传包的函数开始计时，到调用结束也结束计时，定义 clock_t start, end,time;按照 time = (end - start) / CLOCKS_PER_SEC 计算出 time 得到具体的时间。

平均吞吐率根据公式：$x = \frac{C}{t}$

其中其中 C 为完成的任务总量，而 t 为完成这些任务所需要的总时间。

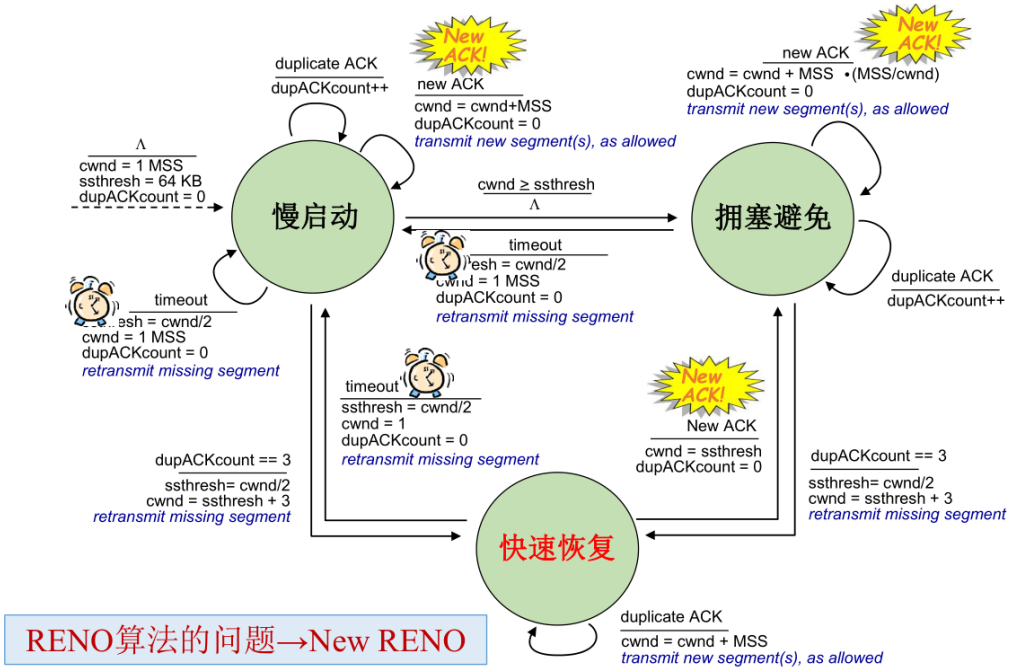
所以在本次程序中，平均吞吐率=length *8/ 1000 / time 单位是："kbps"（其中 length 为文件的长度）

lab3-3 的补充协议部分

- **拥塞控制算法**

对于拥塞控制算法主要是仿照于 TCP 的拥塞控制的一种算法进行实现，具体的算法就是 RENO 算法（详情见下图）。主要包含三个状态，以下将具体说明：

TCP拥塞控制： RENO算法状态机



状态一：慢启动状态，发送窗口初始设置为 1，然后每次进行进行增加，即收到对方一个合格的 ack 之后，发送方会连续发送两个数据段过去。这样就可以模拟慢启动状态的指数增长，即每次发送窗口会增加一倍。

如果期间有数据段包发生了超时重传，那么此时发送方应该进入，并且将下一轮的 ssthresh 设置为当前发送窗口的一半及 cwnd/2，之后将 cwnd 设置为 1，重新开始慢启动状态。

如果当前发送窗口 cwnd 超过了 ssthresh，则进入下一个状态————拥塞避免状态。

如果在当前出现了 3 个冗余 ACK 的情况，则计入另一个状态，快速回复状态。

状态二：拥塞避免状态，发送窗口初始为转移到这个状态时的样子，然后窗口的增长方式变为没个发送阶段增长 1 个 MSS，即下一个完整的发送窗口将会比当前完整的发送窗口大 1，进行线性增长，直到出现以下状况：

出现某个包的超时事件，表明这个包已经丢失。并且后面的包也会丢失，因此我们需要进行强烈的拥塞控制，将 ssthresh 设置为 cwnd/2，并且将 cwnd 设置为 1，重新进入慢启动状态。

如果出现了 3 次冗余 ACK，则证明只有一个包丢失，其余后面的包到达发送方，因此状态将走到下一个状态——快速恢复状态

状态三：快速恢复状态，当收到 3 个重复 ACK 时，把 ssthresh 设置为 cwnd 的一半，把 cwnd 设置为 ssthresh 的值加 3，然后重传丢失的报文段，加 3 的原因是因为收到 3 个重复的 ACK，表明有 3 个“老”的数据包离开了网络。若再收到重复的 ACK 时，拥塞窗口增加 1。当收到新的数据包的 ACK 时，把 cwnd 设置为第一步中的 ssthresh 的值。原因是因为该 ACK 确认了新的数据，说明从重复 ACK 时的数据都已收到，该恢复过程已经结束，可以回到恢复之前的状态了，也即再次进入拥塞避免状态。

其余在窗口内的发送与 3.2 基本一致。

- **传输时间和平均吞吐率：**

从调用开始传包的函数开始计时，到调用结束也结束计时，定义 clock_t start, end,time;按照 time = (end - start) / CLOCKS_PER_SEC 计算出 time 得到具体的时间。

平均吞吐率根据公式：$x = \frac{C}{t}$

其中其中 C 为完成的任务总量，而 t 为完成这些任务所需要的总时间。

所以在本次程序中，平均吞吐率=length *8/ 1000 / time 单位是："kbps"（其中 length 为文件的长度）

实验代码

【客户端】:

- 先定义一些常量和必要的转换函数，比如端口号和得到 number 的高八位和低八位等等。

```
#include<Winsock2.h>
#include<vector>
#include<stdio.h>
#include<string.h>
#include<iostream>
#include<algorithm>
#include<time.h>
#include<cstdlib>
#include <fstream>
#pragma comment(lib,"ws2_32")
using namespace std;
#define SERVER_PORT 4001//路由器转发的端口
#define CLIENT_PORT 6665
#define High(number) ((int)number&0xFF00)>>8
#define Low(number) ((int)number&0x00FF)
#define HighLow(h,l) ((((int)h<<8)&0xff00)|((int)l&0xff))
const int bufferSize = 4096;
const unsigned char SHAKE_1 = 0x01;
const unsigned char SHAKE_2 = 0x02;
const unsigned char SHAKE_3 = 0x04;
const unsigned char WAVE_1 = 0x80;
const unsigned char WAVE_2 = 0x40;
char buffer[200000000];
int len;
```

- 定义数据包的格式和一些函数:

```
struct MESSAGE {
    int server_port;//端口号
    int seq_num;//序号
    int length;//报文段二进制长度
    int check_sum;//校验和
    char* message;//报文段
    MESSAGE() {}
    //定义的数据包格式：
    MESSAGE(int server_port, int seq_num, int length, int check_sum, char* message) :
        server_port(server_port),
        seq_num(seq_num),
        length(length),
        check_sum(check_sum),
        message(message) {}
    void print() {
        printf("端口:%d\n数据包序列号:%d\n数据段长度:%d\n校验和:%d\n数据:%s\n",
            server_port, seq_num, length, check_sum, message);
    }
```

- 整理收到的报文：（其中有用到求校验和函数，在下面将进行说明）

```
//真正的报文，二进制串，保存到msg中
void send_message(char* msg) {
    //存的不是每一位，而是把8位作为一个字节，存到char的一个单位里!
    msg[0] = High(server_port);
    msg[1] = Low(server_port);//前两位存端口号
    msg[2] = High(seq_num);
    msg[3] = Low(seq_num);//之后存序列号
    msg[4] = High(length);
    msg[5] = Low(length);//之后是数据长度
    msg[6] = 0;
    msg[7] = 0;//校验和
    for (int i = 0; i < length; ++i) {
        msg[8 + i] = message[i];//之后都存数据
    }

    int a = checksum(msg);//存入数据之后再计算校验和
    msg[6] = High(a);
    msg[7] = Low(a);
}
```

- 计算校验和函数：

```
//计算校验和，每16位转为10进制，然后求和取反
int checksum(char* msg) {
    unsigned long sum = 0;
    for (int i = 0; i < 8 + length; i += 2) {//这里不能用strlen(msg)，因为如果中间有0，读
        sum += HighLow(msg[i], msg[i + 1]);
        sum = (sum >> 16) + (sum & 0xffff);
        // ...
    }
    return (~sum) & 0xffff;
}
};
```

- 解析报文的函数：解析收到的 udp 报文，并将它写入 message 中。

```
//解析udp报文msg，返回报文段，写入message
bool handle(char* msg, char* message) {
    int server_port = HighLow(msg[0], msg[1]);
    int seq_num = HighLow(msg[2], msg[3]);
    int length = HighLow(msg[4], msg[5]);
    int check_sum = HighLow(msg[6], msg[7]);
    for (int i = 0; i < length; i++) {
        message[i] = msg[8 + i];
    }
    //  MESSAGE m=MESSAGE(server_port,seq_num,length,check_sum,message);
    //  m.print();
    //直接计算校验和
    int sum = 0;
    for (int i = 0; i < length + 8; i += 2) {
        sum += HighLow(msg[i], msg[i + 1]);
        sum = (sum >> 16) + (sum & 0xffff);
    }
    if (sum == 0xffff) {//校验和正确，回复ACK
        return true;
    }
    else {
        return false;
    }
}
```

- 定义拥塞控制的一些变量：具体含义见注释。

```
double cwnd = 1.0;  //窗口大小
double ssthresh = 8.0;  //阈值，一旦达到阈值，则指数->线性
int dup_ack_cnt = 0;//冗余ack计数器
int last_ack_seq = 0;//上一次的ack序号，用于更新ack_cnt

SOCKET localSocket;
struct sockaddr_in serverAddr, clientAddr;//接收端的ip和端口号信息

DWORD WINAPI handlerRequest(LPVOID lpParam);//负责接收的线程

bool begin_recv = false;//可以开始接收
bool waiting = false; //等待发送
string temp;

int nextseqnum = 0;//序号，就是nxtseqnum
```

- 接收和发送信息函数：

```
//发送报文
void send_to(char* message) {
    MESSAGE u = MESSAGE(SERVER_PORT, nextseqnum, 1024 - 10, 0, message);//打包成udp
    //打包后的报文msg
    char msg[bufferSize];
    u.send_message(msg);
    //发送数据
    sendto(localSocket, msg, sizeof(msg), 0, (SOCKADDR*)&serverAddr, sizeof(SOCKADDR));//大小
}


//接收
void recv_from(char* message) {
    char msg[bufferSize];
    int size = sizeof(serverAddr);
    recvfrom(localSocket, msg, sizeof(msg), 0, (SOCKADDR*)&serverAddr, &size);
    handle(msg, message);
}
```

- 接下来是关键的文件发送函数：（其中包括了拥塞控制中的超时状态的处理）。

```
double dt;
clock_t start, end;
double TIME_OUT = 10000;//超时时间
//把文件分块存储在v中
vector<char*>v;//分片发送
char block[12000][1024];


clock_t all_start, all_end;
int base = 0;
void send_file_2(string path) {
    //先保存到数组里
    FILE* fin = fopen(path.c_str(), "rb");//先读文件
    int i = 0;
    while (!feof(fin)) {
        fread(block[i], 1, sizeof(block[i]) - 10, fin);
        v.push_back(block[i++]);//加入数组
    }
    fclose(fin);

    char* message;
    int n = v.size();

    all_start = clock();
    while (nextseqnum < n)
    {
        int end = clock();
        dt = (double)(end - start);
        //如果超时，未确认的全部重传
        if (dt >= TIME_OUT) {
            cout << "超时，超时时间为：" << dt << endl;
            ssthresh =cwnd/ 2;
            cwnd = 1;
            dup_ack_cnt = 0;
            nextseqnum = base;//重发，回退到base
            cout << "超时，窗口变化为：" << cwnd << endl;
            cout << "超时，阈值变化为：" << ssthresh << endl;
        }
```

```
        //当现在有空间，且不在停等状态时，允许发送报文
        if (nextseqnum < base + cwnd)
        {
            message = v[nextseqnum];
            send_to(message);
            start = clock();
            begin_recv = true;//***可以开始接收服务器端消息啦***
            //cout<<nextseqnum<<' '<<strlen(message)<<endl;
            nextseqnum++; //新的报文，序号变化
        }
    }
    all_end = clock();
    cout << "结束" << endl;
}
```

- 主函数部分：包含 socket 套接字的创建和连接建立和发包的调用主要过程。

  本次将 ip 地址进行绑定，端口号进行输入，这样才能将路由程序正确使用。

```
int main() {
    //初始化
    WSADATA wsaData;
    WORD wVersionRequested = MAKEWORD(2, 1);
    int error=WSAStartup(wVersionRequested, &wsaData);
    if (error) {
        printf("init error");
        return 0;
    }
    cout << "请输入目的端口号(路由器4001，服务器6666)：";
    int sendbuf;
    cin >> sendbuf;
    //去连接服务器的socket
    localSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

    //服务器的ip和端口号
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(sendbuf);
    serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");

    clientAddr.sin_family = AF_INET;
    clientAddr.sin_port = htons(6665);
    clientAddr.sin_addr.s_addr = inet_addr("127.0.0.1");

    HANDLE hThread = CreateThread(NULL, 0, handlerRequest, LPVOID(), 0, NULL);
```

实验 3-2 需要输出传输时间和平均吞吐率，具体计算说明见协议设计，计算过程见以下的代码中。

```
    string filename;
    while (1) {
        printf("请输入要发送的文件名：");
        cin >> filename;
        ifstream fin(filename.c_str(), ifstream::binary);
        if (!fin) {
            printf("文件未找到!\n");
            continue;
        }
        else {
            cout << "找到文件，开始发送..." << endl;
            unsigned char t = fin.get();
            while (fin) {
                buffer[len++] = t;
                t = fin.get();
            }
            fin.close();
            send_file_2(filename);
        }

        break;
    }
    //string sendbuf;
    //cin >> sendbuf;
    //send_test();
    double all_pass = (double)(all_end - all_start) / CLOCKS_PER_SEC;


    Sleep(10000);//一定要等一会，否则线程立即关掉，没法打印结果
    CloseHandle(hThread);
    closesocket(localSocket);
    WSACleanup();
    cout << "传输时间为：" << all_pass << "s" << endl;
    cout << "平均吞吐率为：" << len*8 /1000/ all_pass << "kbps" << endl;
    return 0;
}
```

- 负责接收的线程：其中包含拥塞控制算法的慢启动和拥塞避免和三个重复 ack 的情况下窗口和阈值的变化情况。

```
//负责接收的线程
DWORD WINAPI handlerRequest(LPVOID lpParam) {
    while (1) {
        Sleep(10);//改变begin_recv后，需要等一下再接收，不然会丢掉第一个ACK
        char msg[bufferSize], message[bufferSize];
        int size = sizeof(serverAddr);
        if (begin_recv == false) {
            continue;
        }
        recvfrom(localSocket, msg, sizeof(msg), 0, (SOCKADDR*)&serverAddr, &size);


        if (handle(msg, message)) {
            int new_ack_seq = HighLow(msg[2], msg[3]);
            cout << "ACK" << new_ack_seq << endl;
            base = new_ack_seq + 1;

            if (new_ack_seq != last_ack_seq) {//ack序号不冗余
                last_ack_seq = new_ack_seq;
                dup_ack_cnt = 0;
                //拥塞避免状态
                if (cwnd >= ssthresh) {
                    cwnd += 1 / cwnd;
                    cout << "发生拥塞避免，窗口变化为：" << cwnd << endl;
                    cout << "发生拥塞避免，阈值为：" << ssthresh << endl;
                }
                //慢启动状态，每收到一个ack，窗口+1，一轮过后就是*2
                else {
                    cwnd++;
                    cout << "慢启动状态，窗口为：" << cwnd << endl;
                    cout << "慢启动状态，阈值为：" << ssthresh << endl;
                }
            }
```

```
            else {//冗余
                dup_ack_cnt++;
                if (dup_ack_cnt == 3) {//3个冗余ack，则阈值=cwnd/2,cwnd=阈值+3
                    ssthresh = cwnd / 2;
                    cwnd = ssthresh + 3;
                    nextseqnum = base;//重传，回退到base
                    cout << "3个重复ack，窗口变化为：" << cwnd << endl;
                    cout << "3个重复ack，阈值为：" << ssthresh << endl;
                }
            }
            //当收到一个ack，则下一个包开始计时
            //cout << "窗口大小："<<cwnd << endl;
            start = clock();
        }
    }
```

【服务器端】:

其中校验和的计算函数、主函数和程序头部定义的一些常量，比如数得到数据高八位和低八位等和数据包的格式定义和处理等和客户端完全相同。因此不再阐述。

- 主要介绍接收文件函数：传入文件的路径，接收到数据计算校验和正确的话，若收到的序号是自己想要，则发送 ACK 回客户端，并显示接收成功，最后更新序列号。

```
int expected_seqnum = 0;//希望接收的序列号
void recv_file_2(string path) {
    char msg[bufferSize], message[bufferSize];
    FILE* fout = fopen(path.c_str(), "wb");//将文件写入具体路径
    while (1)
    {
        int size = sizeof(clientAddr);
        recvfrom(localSocket, msg, sizeof(msg), 0, (SOCKADDR*)&clientAddr, &size);
        //bool error_msg = random_loss();
        //if (error_msg == false)msg[1] /= 2;

        bool check = handle(msg, message);

        //一定概率丢掉ACK包
        //bool lost_ack = random_loss();
        //if (lost_ack == false)continue;

        int recv_seqnum = HighLow(msg[2], msg[3]);
        //如果校验和正确，并且收到的序号是想要的
        if (check == true && expected_seqnum == recv_seqnum) {
            cout << "接收成功" << endl;
            fwrite(message, 1, HighLow(msg[4], msg[5]), fout);
            send_to("ACK", expected_seqnum);
            expected_seqnum++;//更新想要的序号
        }
        else {
            //否则发送最近正确接收的序号
            cout << "接收失败" << endl;
            send_to("ACK", expected_seqnum - 1);
        }
        memset(message, 0, sizeof(message));
    }
    fclose(fout);
}
```

- 主函数部分：包括套接字的建立和连接的建立，定义接收的数据具体存入的位置。

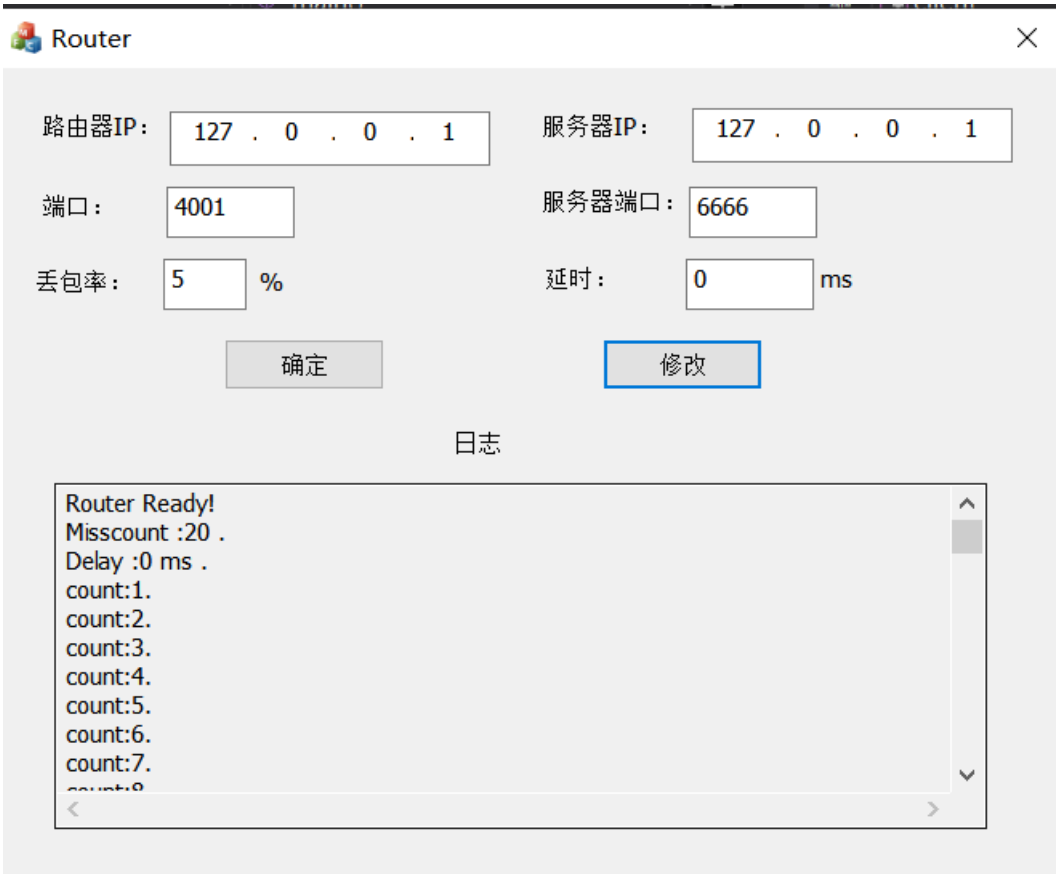
```
int main() {
    WSADATA wsaData;
    WORD wVersionRequested = MAKEWORD(2, 1);
    int nError = WSAStartup(wVersionRequested, &wsaData);
    if (nError) {
        printf("start error\n");
        return 0;
    }

    //本地socket，只负责接收
    localSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    //服务器的ip和端口号
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(6666);//htons把unsigned short类型从主机序转换到网络序
    serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");

    //绑定本地socket和端口号
    bind(localSocket, (SOCKADDR*)&serverAddr, sizeof(SOCKADDR));
    cout << "服务器端口为：" << ntohs(serverAddr.sin_port) << endl;//ntohs把unsigned short类
    printf("等待接入...\n");
    //wait_shake_hand();
    //printf("用户已接入。\n正在接收数据...\n");
    srand(time(0));
    recv_file_2("newmessage");
    closesocket(localSocket);
    WSACleanup();
    return 0;
}
```

实验结果

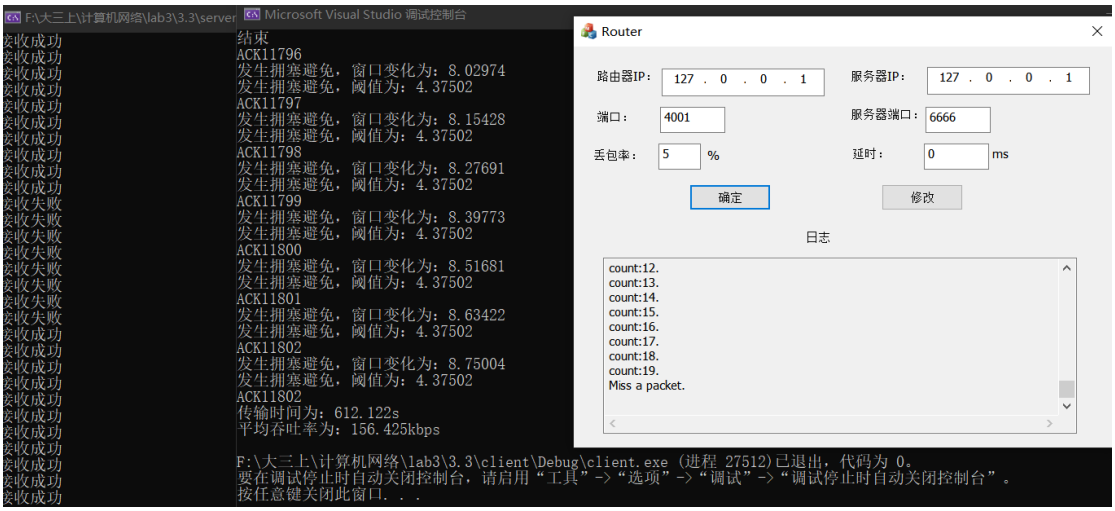
正确运行顺序：先打开服务器端的控制台，再打开路由器程序，填好具体的端口和 ip 之后点击确定，之后再打开客户端的控制台。

手动输入端口地址进行绑定，输入需要进行传输的文件路径，回车之后便开始传输。（例如：3.jpg 的传输截图如下）

路由程序的设定：



例如：3.jpg 的传输截图如下



打开接收到的文件在本地的路径，进行检查。在最后会显示出传输时间和平均吞吐率，进行一下传输性能的测试。