计算机网络第三次实验报告

Lab3-3 基于 UDP 服务设计可靠传输协议

2212422 孙启森

一、实验内容

实验3-3：在实验3-2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、协议设计

（一）报文格式

在本次实验中，仿照 TCP 协议的报文格式进行了数据报设计，其中整个报文包括报头段和数据段。报头包括源端口号、目的端口号、序列号、确认号、消息数据长度、标志位、校验值。其中标志位包括 FIN、PUSH、ACK、SYN 四位。数据段则是大小为size的数据。
具体的设置如下所示。

```
 |0            15|16           31|
 ---------------------------------
 |           sourcePort          |
 ---------------------------------
 |           destinationPort     |
 ---------------------------------
 |             Seqnum            |
 ---------------------------------
 |             acknum            |
 ---------------------------------
 |            Length             |
 ---------------------------------
 |      Flags    |      Checksum |
 ---------------------------------
 |              Data             |
 ---------------------------------
```

```
#define SYN 0b01
#define ACK 0b10
#define FIN 0b100
#define PUSH 0b1000
```

```
 ---------------------------------
 struct Header {
        uint16_t sourcePort;
        uint16_t destinationPort;
        uint32_t seqnum;
        uint32_t acknum;
        uint16_t Flags;
        uint16_t checksum;
        uint32_t length;
};
struct message
{
        Header head;
        char data[size];
        message() :head{ 0, 0, 0, 0, 0, 0,0 } {
                memset(data, 0, size);
        }
}
```

（二）消息传输机制

我们在lab3-2的基础上，实现了拥塞控制算法，采用RENO协议。

1. 建立连接——三次握手

我们仿照TCP协议设置了三次握手，首先由发送方向接收方发送连接请求，并将FLAG设置为SYN，然后，接收方会发送报文设置FLAG为SYN以及ACK，并相应的加上acknum。最后发送方再发送一个ACK的包，从而确认收到。实现三次握手，连接建立。

2. 差错检测

对于差错检测，我们采用了校验和的形式，针对报文报文内容计算校验和，并将其设置的报文中。然后由接收方在收到时进行校验和的确认。因为校验和计算时需要16位，因此我们发送时需要做到发送的message的大小为16位的倍数。

3拥塞控制

3.1 慢启动

在传输的初始阶段，我们设置cwnd为1个数据包，同时设置阈值ssthresh为64个数据包，重复计数为初始为0.
当收到新的ACK时，就对cwnd进行加一。同时将重复计数归零。当cwnd超过阈值ssthresh时，就会进入拥塞避免阶段。
而若是发生超时，则会将阈值变为cwnd/2，接着将cwnd重置为1.
当接收到重复的ACK时，则会将重复计数加1，当重计数达到3时，就会进入快速恢复阶段。同时将阈值变为cwnd/2，cwnd=ssthresh+3.

3.2拥塞避免

进入拥塞避免后，每当收到一个新的ACK，就会对cwnd加上1/cwnd.同时重计数清零。
当收到重复的ACK时，就会将重计数加一。重计数达到3时，就会进入快速恢复阶段。同时将阈值变为cwnd/2，cwnd=ssthresh+3.
当发生超时时，就会重新进入慢启动阶段，同时将阈值变为cwnd/2，cwnd归一。

3.3快速恢复

进入快速恢复后，当收到重复的ACK则将cwnd加一。
若收到新的ACK则将cwnd变为ssthresh，然后进入拥塞避免阶段。
若发生超时，也进入慢启动阶段，同时将阈值变为cwnd/2，cwnd归一。

4. 超时重传

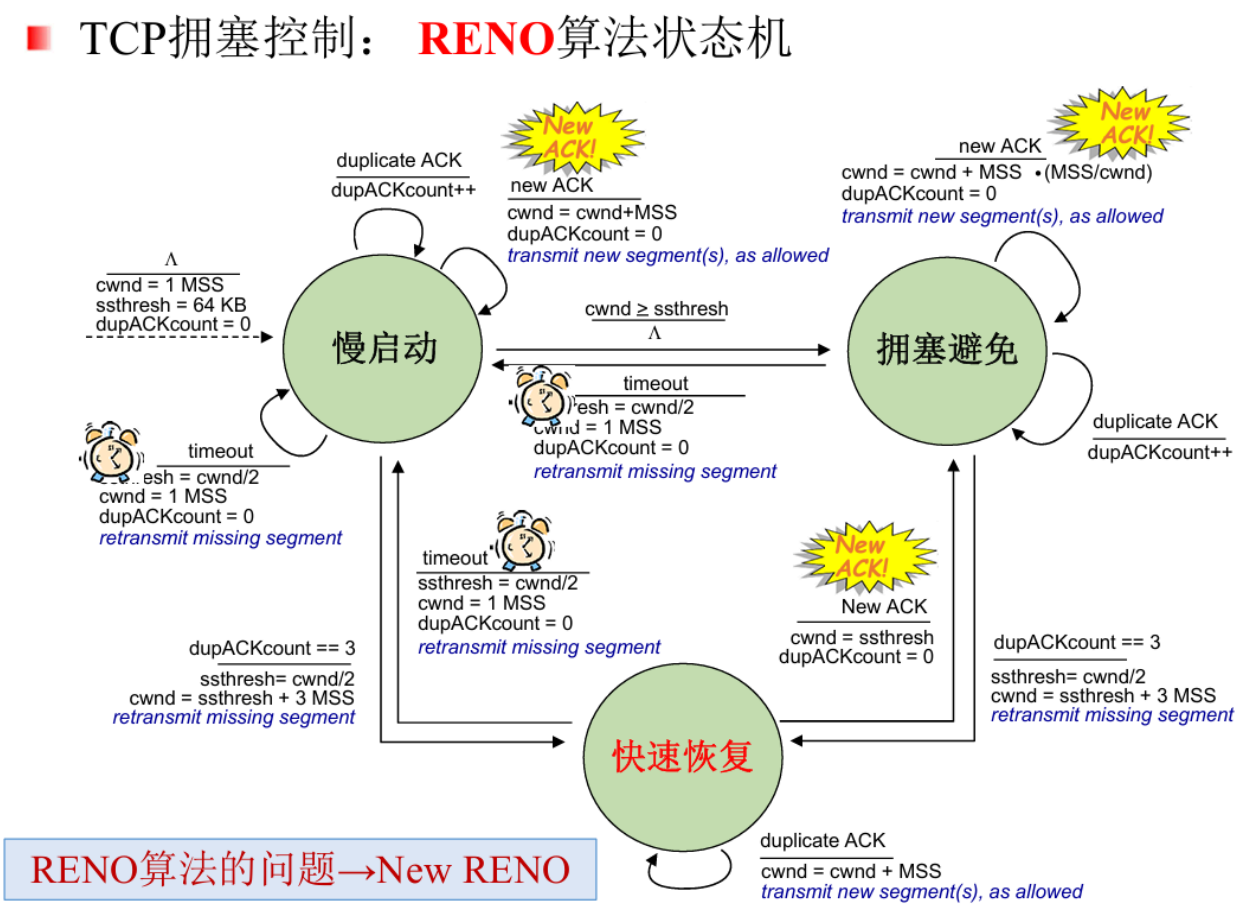
采用记时机制，每当收到正确的ack包时就重置计时器，若对应时间未能收到，则将从base_seq重新开始传输。

5. 断开连接——四次挥手

我们仿照TCP的四次挥手设计了断开连接的操作。当发送端发送FIN想要断开时，接收方先回复ACK，再发送FIN，最后由发起方发送ACK.接受方收到ACK后立即关闭。发起方等待一段时间没有信息则自动关闭。seq及acknum的变化与握手时相似，不再赘述。

6. 状态机

（1）发送端



这里状态机原理与先前所述一致，不再赘述。

三、代码实现

（一）协议设计

对应标志位

```
#define SYN 0b01
#define ACK 0b10
#define FIN 0b100
#define PUSH 0b1000
```

我们将message分为Header和data两个区域。Header作为每个包的信息，而data则是要传输的数据。

```
struct message
{
        Header head;
        char data[size];
        message() :head{ 0, 0, 0, 0, 0, 0,0 } {
                memset(data, 0, size);
        }
        void setchecksum();
        bool check();
        void setsyn() { this->head.Flags |= SYN; }
        void setack() { this->head.Flags |= ACK; }
        void setfin() { this->head.Flags |= FIN; }
        void setpush() { this->head.Flags |= PUSH; }
        bool ispush() { return (this->head.Flags & PUSH) != 0; }
        bool issyn(){ return (this->head.Flags & SYN) != 0; }
        bool isack() { return (this->head.Flags & ACK) != 0; }
        bool isfin() { return (this->head.Flags & FIN) != 0; }
        void print() {
                printf("----------------\n");
                printf("Source Port: %u\n",head.sourcePort);
                printf("Destination Port: %u\n",head.destinationPort);
                printf("Sequence Number: %u\n", head.seqnum);
                printf("Acknowledgment Number: %u\n",head.acknum);
                printf("Flags: ");
                if (issyn()) printf("SYN ");
                if (isfin()) printf("FIN ");
                if (head.Flags & ACK) printf("ACK ");
                if (head.Flags & PUSH) printf("PUSH ");
                printf("\n");
                printf("Checksum: %u\n", head.checksum);
                printf("Length: %u\n",head.length);
                printf("----------------\n");
        }

};
```

在结构体中，我们设置了相应的校验函数，标志位设置，以及输出的函数。标志位的设置和检验具体不再赘述。

（二）初始化

我们在发送端和接受端采用相同的初始化方式，除了发送端设置为非阻塞模式，而因为是单方向传输，所以在接收端采用阻塞模式。下面以发送方为例。

```
bool initial()
{
	serversocket = socket(AF_INET,SOCK_DGRAM, IPPROTO_UDP);
	if (serversocket == INVALID_SOCKET)
	{
		cout << "出错了" << ",错误请看" << WSAGetLastError();
		WSACleanup();
		return false;
	}
	u_long mode = 1; // 1 表示非阻塞模式
	if (ioctlsocket(serversocket, FIONBIO, &mode) != NO_ERROR) {
		std::cerr << "无法设置非阻塞模式: " << WSAGetLastError() << std::endl;
		closesocket(serversocket);
		WSACleanup();
		return false;
	}
	sockaddr_in addrSrv;
	addrSrv.sin_family = AF_INET;
	const char* ip = "127.0.0.1";
	if (inet_pton(AF_INET, ip, &addrSrv.sin_addr) <= 0) {
		std::cerr << "inet_pton失败了 " << WSAGetLastError() << std::endl;
		closesocket(serversocket);
		WSACleanup();
		return false;
	}
	addrSrv.sin_port = htons(serverport);
	if(bind(serversocket, (SOCKADDR*)&addrSrv, sizeof(addrSrv)) == SOCKET_ERROR)
	{
		return false;
	}


	clientaddr.sin_family = AF_INET;
	clientaddr.sin_port = htons(clientport);
	if (inet_pton(AF_INET, ip, &clientaddr.sin_addr) <= 0) {
		std::cerr << "inet_pton失败了 " << WSAGetLastError() << std::endl;
		closesocket(serversocket);
		WSACleanup();
		return false;
	}
	return true;
}
```

（三）建立连接——三次握手

1. 发送端

- 发送第一次握手消息，并开始计时，申请建立连接，然后等待接收第二次握手消息
  - 如果超时未收到，则重新发送，多次重传失败则握手失败。
- 收到正确的第二次握手消息后，发送第三次握手消息

```
bool connect()
{
        cout << "开始握手" << endl;
        message msg1;
        msg1.setsyn();
        msg1.head.seqnum = seq;
        int len = sizeof(clientaddr);
        if (send(msg1) == SOCKET_ERROR)
        {
                int error_code = WSAGetLastError();
                printf("sendto failed with error: %d\n", error_code);
                closesocket(serversocket); // 关闭套接字
                WSACleanup(); // 清理 Winsock
                return false;
        }
        cout << "发送第一次握手的报文" << endl;
        msg1.print();
        start_time = clock();
        int i = 0;
        message msg2;
        while (1)
        {
                if (recvfrom(serversocket, (char*)&msg2, sizeof(msg2), 0,
(SOCKADDR*)&clientaddr, &len) > 0)
                {
                        if (!msg2.check() || msg1.head.seqnum + 1 != msg2.head.acknum || !
(msg2.issyn() && msg2.isack()))
                        {
                                cout << "有错误";
                                return false;
                        }
                        else
                        {
                                cout << "收到第二次握手的报文，第二次握手成功" << endl;
                                msg2.print();

                        }
                        break;
                }
                if (i > 2)
                {
                        cout << "重传过多且失败,终止" << endl;
                        i = 0;
                        closesocket(serversocket); // 关闭套接字
                        WSACleanup(); // 清理 Winsock
                        return false;
```

```
                }
                end_time = clock();
                double elapsed_time = 1000.0 * (end_time - start_time) / CLOCKS_PER_SEC;
                if (elapsed_time > waittime)
                {
                        start_time = clock();
                        i++;
                        cout << "已超时，现在进行重传" << endl;
                        send(msg1);
                }

        }
        message msg3;
        seq = seq + 1;
        ack = msg2.head.seqnum + 1;
        msg3.head.seqnum = msg1.head.seqnum + 1;
        msg3.head.acknum = ack;
        msg3.setack();
        if (send(msg3) == SOCKET_ERROR)
        {
                int error_code = WSAGetLastError();
                printf("sendto failed with error: %d\n", error_code);
                closesocket(serversocket); // 关闭套接字
                WSACleanup(); // 清理 Winsock
                return false;
        }
        cout << "发送第三次握手的报文" << endl;
        msg3.print();
        cout << sizeof(msg3) << endl;
        return true;
}
```

2. 接收端

- 接收正确的第一次握手消息，发送第二次握手消息，并开始计时，等待接收第三次握手消息
- 接收到正确的第三次握手消息，连接成功建立
  与发送方相似，不再附上代码。

（四）数据传输

我们定义了send函数，用于简化操作。

```
int send(message &messg)
{
        messg.head.sourcePort = serverport;
        messg.head.destinationPort = clientport;
        messg.setchecksum();
        return sendto(serversocket, (char*)&messg, sizeof(messg), 0,
(SOCKADDR*)&clientaddr,sizeof(clientaddr));
}
```

1. 发送端

本次文件我们要进行文件传输，因此，我们需要找到对应的文件。这里采用输入文件名的形式，找到当前路径下的文件.接着根据文件的大小，设置相应大小的缓冲区。然后将文件的姓名储存到data中，大小存储到length中。设置包的标志位未PUSH，用来表明是文件信息包。对应的seq++。

```
        message file;
        FILE* file1 = fopen(name, "rb");
        if (!file1) {
                perror("无法打开文件");
                return false;
        }

        fseek(file1, 0, SEEK_END);
        long fileSize = ftell(file1);
        datasize = fileSize;
        fseek(file1, 0, SEEK_SET);
        strncpy(file.data, name, strlen(name));
        int num = fileSize / size + 1;
        if (fileSize % size > 0)
        {
                num++;
        }
        message* sendbuffer = new message[num];
        sendbuffer[0] = file;
        file.head.seqnum = seq;
        file.head.length = num;
        file.setpush();
        file.head.acknum = ack;
        seq += 1;
        sendbuffer[0] = file;
```

1.1发送端发送

发送时我们同样按照窗口进行发送，若next_seq小于窗口大小，则继续发送，超出窗口时则不进行发送操作。在这里我们同样也尝试了多线程操作，只需要将发送接收放到两个线程中，然后在对公共值修改时加锁即可。但其速度不如单线程，这里不再附上。

```
if (next_seq < base_seq + int(cwnd+1) && next_seq <= num - 1) {
                        send(sendbuffer[next_seq]);
                        cout << next_seq << "/" << num - 1 << endl;
                        cout << "window" << base_seq << "-" << base_seq + cwnd << endl;
                        cout << "window_size" << cwnd << endl;
                        sendbuffer[next_seq].print();
                        if (base_seq == next_seq) {
                                start_timer();
                        }
                        next_seq++;
                }
```

1.2发送端接收

为了确保reno各个状态的转换，我们采用state用来区分快速重传以及其他。重复计数达到3时，就会将state置为1，并将cwnd和ssthresh对应变化，这样在进行接收时就会进入快速重传状态。当接收到新的ack时就会将state变为0，从而改变状态。
当state为0时，会根据cwnd和ssthresh的大小关系来进行判断是慢启动还是拥塞避免。
若发生超时，则都会回归到慢启动状态。

```
if (is_timeout()) {
                        cout << "已超时，重传" << base_seq << "--" << next_seq << endl;
                        cout << "慢启动转换" << endl;
                        next_seq = base_seq;
                        state = 0;

                        ssthresh = max((cwnd + 1) / 2,2);
                        cwnd = 1;
                        cout << "cwnd  " << cwnd << endl;
                        cout << "ssthrsh  " << ssthresh << endl;
                        count = 0;
                        start_timer();
                }
                if (state == 0)
                {
                        if (recvfrom(serversocket, (char*)&ack_msg, sizeof(ack_msg), 0,
(SOCKADDR*)&clientaddr, &len) > 0) {
                                if (ack_msg.check() && ack_msg.head.acknum >= base_seq +
head_seq && ack_msg.isack()) {
                                        if (lastack != ack_msg.head.acknum)
                                        {
                                                count = 0;
                                                if (cwnd < ssthresh)
                                                {
                                                        cwnd += 1;
                                                }
                                                else if (cwnd >= ssthresh)
                                                {
                                                        cout << "cwnd  " << cwnd << endl;
                                                        cout << "ssthrsh  " << ssthresh <<
endl;

                                                        cout << "拥塞避免启动" << endl;
                                                        cwnd += 1 / cwnd;
                                                }
                                                lastack = ack_msg.head.acknum;
                                                start_timer();
                                                base_seq = ack_msg.head.acknum - head_seq;
                                                if (ack_msg.head.acknum == seq)
                                                {
                                                        return true;
                                                }
                                        }
                                        else if (lastack == ack_msg.head.acknum)
                                        {
                                                count++;
                                        }
```

```
                                        if (count == 3)
                                        {
                                                cout << "快速恢复启动" << endl;
                                                state = 1;
                                                ssthresh = max(cwnd / 2, 2);
                                                cwnd = ssthresh + 3;
                                        }
                                }
                        }
                }
                else if (state == 1)
                {
                        if (recvfrom(serversocket, (char*)&ack_msg, sizeof(ack_msg), 0,
(SOCKADDR*)&clientaddr, &len) > 0) {
                                if (ack_msg.check() && ack_msg.head.acknum >= base_seq +
head_seq && ack_msg.isack()) {
                                        if (lastack != ack_msg.head.acknum)
                                        {
                                                count = 0;
                                                cwnd = ssthresh;
                                                state = 0;
                                                lastack = ack_msg.head.acknum;
                                                start_timer();
                                                base_seq = ack_msg.head.acknum - head_seq;
                                                if (ack_msg.head.acknum == seq)
                                                {
                                                        return true;
                                                }
                                        }
                                        else if (lastack == ack_msg.head.acknum)
                                        {
                                                cwnd += 1;
                                        }

                                }
                        }
                }
```

2. 接收端

对于接收端，会根据发送报文的不同标志位进行处理。当收到的包为PUSH时，会根据内容创建新的的文件，同时存储文件大小。接着进入循环之中。当收到的文件完整时，关闭文件。如果收到FIN，则对应使用挥手函数。而对于那些不符合自己期望的包，则会进行重传上次一确认的最高的包。

```
void recfile()
{
        int len = sizeof(clientaddr);
        message msg;
        message sendmsg;
        ofstream outFile;
        long recived = 0;
        long filesize = 0;
        int i = 0;
        while (true)
        {

                recvfrom(serversocket, (char*)&msg, sizeof(msg), 0, (SOCKADDR*)&clientaddr,
&len);
                if (msg.ispush() && msg.check() && msg.head.acknum == seq && msg.head.seqnum
== ack)
                {
                        recived = 0;
                        if (outFile.is_open()) {
                                outFile.close();
                        }

                        outFile.open(msg.data, std::ios::out | std::ios::binary);
                        if (!outFile) {
                                perror("无法打开文件");
                                break;
                        }
                        cout << "new file";
                        filesize = msg.head.length;
                        cout << filesize << endl;
                        recived++;
                        ack += 1;
                        sendmsg.setack();
                        sendmsg.head.acknum = ack;
                        sendmsg.head.seqnum = seq;
                        send(sendmsg);
                        sendmsg.print();
                        cout << "wat for" << ack << endl;

                }
                else if (msg.check() && msg.head.acknum == seq && (recived +1) < filesize &&
msg.head.seqnum == ack)
                {
                        ack += 1;
                        recived++;
                        sendmsg.head.acknum = ack;
```

```
                        sendmsg.head.seqnum = seq;
                        sendmsg.setack();
                        send(sendmsg);
                        if (outFile.is_open())
                        {
                                outFile.write(msg.data, msg.head.length);
                        }
                        sendmsg.print();
                        cout << "wat for" << ack << endl;
                }
                else if (msg.check() && msg.head.acknum == seq && (recived +1) >= filesize
&& msg.head.seqnum == ack)
                {
                        recived ++;
                        ack += 1;
                        sendmsg.head.acknum = ack;
                        sendmsg.head.seqnum = seq;
                        sendmsg.setack();
                        send(sendmsg);
                        sendmsg.print();
                        if (outFile.is_open())
                        {
                                outFile.write(msg.data, msg.head.length);
                        }
                        outFile.close();
                        cout << "传输文件成功" << endl;
                }
                else if (msg.check() && msg.head.acknum == seq && msg.head.seqnum != ack)
                {
                        cout << "不是等待的包，重传最高已确认的" << endl;
                        send(sendmsg);
                }
                else if (msg.check() && msg.isfin())
                {
                        ack += 1;
                        cout << "收到第一次挥手的报文" << endl;
                        msg.print();
                        if (!saybye())
                        {
                                cout << "没能说再见" << endl;
                        }
                        else
                        {
                                cout << "bye bye" << endl;
                                closesocket(serversocket);
                                WSACleanup();
```

```
                                break;
                            }
                        }
                    }

}
```

（五）断开连接——四次挥手

1. 发送端

- 发送第一次挥手消息，并开始计时，提出断开连接，然后等待接收第二次挥手消息
  - 如果超时未收到，则重新发送
- 收到正确的第二次挥手消息后，等待接收第三次挥手消息
- 接收到正确的第三次挥手消息，输出日志，准备断开连接
- 再等待设定时间时间，确定没有消息传来，断开连接。

```
bool saybye()
{
        message msg1;
        msg1.setfin();
        msg1.head.seqnum = seq;
        seq++;
        int len = sizeof(clientaddr);
        if (send(msg1) == SOCKET_ERROR)
        {
                int error_code = WSAGetLastError();
                printf("sendto failed with error: %d\n", error_code);
                closesocket(serversocket); // 关闭套接字
                WSACleanup(); // 清理 Winsock
                return false;
        }
        cout << "发送第一次挥手的报文" << endl;
        msg1.print();
        start_time = clock();
        int i = 0;
        message msg2;
        while (1)
        {
                if (recvfrom(serversocket, (char*)&msg2, sizeof(msg2), 0,
(SOCKADDR*)&clientaddr, &len) > 0)
                {
                        cout << msg2.head.acknum << endl;
                        if (!(seq == msg2.head.acknum) || !msg2.isack() ||!msg2.check())
                        {
                                cout << "有错误";
                                return false;
                        }
                        cout << "收到第二次挥手的报文" << endl;
                        msg2.print();
                        break;

                }
                if (i > 2)
                {
                        cout << "重传过多且失败,终止" << endl;
                        i = 0;
                        closesocket(serversocket); // 关闭套接字
                        WSACleanup(); // 清理 Winsock
                        return false;
                }
                end_time = clock();
                double elapsed_time = 1000.0 * (end_time - start_time) / CLOCKS_PER_SEC;
```

```
                if (elapsed_time > waittime)
                {
                        start_time = clock();
                        i++;
                        cout << "已超时，现在进行重传" << endl;
                        send(msg1);
                }
        }
        ack++;
        message msg4;
        start_time = clock();
        while (1)
        {
                if (recvfrom(serversocket, (char*)&msg4, sizeof(msg4), 0,
(SOCKADDR*)&clientaddr, &len) > 0)
                {
                        if (!(seq == msg2.head.acknum) || !msg4.isack() || !msg4.check() ||
!msg4.isfin())
                        {
                                cout << "有错误";
                                return false;
                        }
                        cout << "收到第三次挥手的报文" << endl;
                        msg4.print();

                        break;
                }
                end_time = clock();
                double elapsed_time = 1000.0 * (end_time - start_time) / CLOCKS_PER_SEC;
                if (elapsed_time > 100)
                {
                        cout << "等待时间过长，自动关闭" << endl;
                        closesocket(serversocket); // 关闭套接字
                        WSACleanup(); // 清理 Winsock
                        return true;
                }

        }
        ack++;
        message msg3;;
        msg3.head.seqnum = seq;
        msg3.head.acknum = ack;
        msg3.setack();
        if (send(msg3) == SOCKET_ERROR)
        {
                int error_code = WSAGetLastError();
```

```
                printf("sendto failed with error: %d\n", error_code);
        }
        cout << "发送第四次挥手的报文" << endl;
        msg3.print();
        start_time = clock();
        while (1)
        {
                if (recvfrom(serversocket, (char*)&msg4, sizeof(msg4), 0,
(SOCKADDR*)&clientaddr, &len) > 0)
                {
                        cout << "挥手失败" << endl;
                        break;
                }
                end_time = clock();
                double elapsed_time = 1000.0 * (end_time - start_time) / CLOCKS_PER_SEC;
                if (elapsed_time > 200)
                {
                        cout << "等待时间已过，断开连接，挥手成功" << endl;
                        closesocket(serversocket); // 关闭套接字
                        WSACleanup(); // 清理 Winsock
                        return true;
                }

        }
        return true;
}
```

2. 接收端

- 接收正确第一次挥手消息，发送第二次挥手消息，同意断开连接
- 发送第三次挥手消息，然后等待接收第四次挥手消息
- 接收到正确的第四次挥手消息，输出日志，断开连接

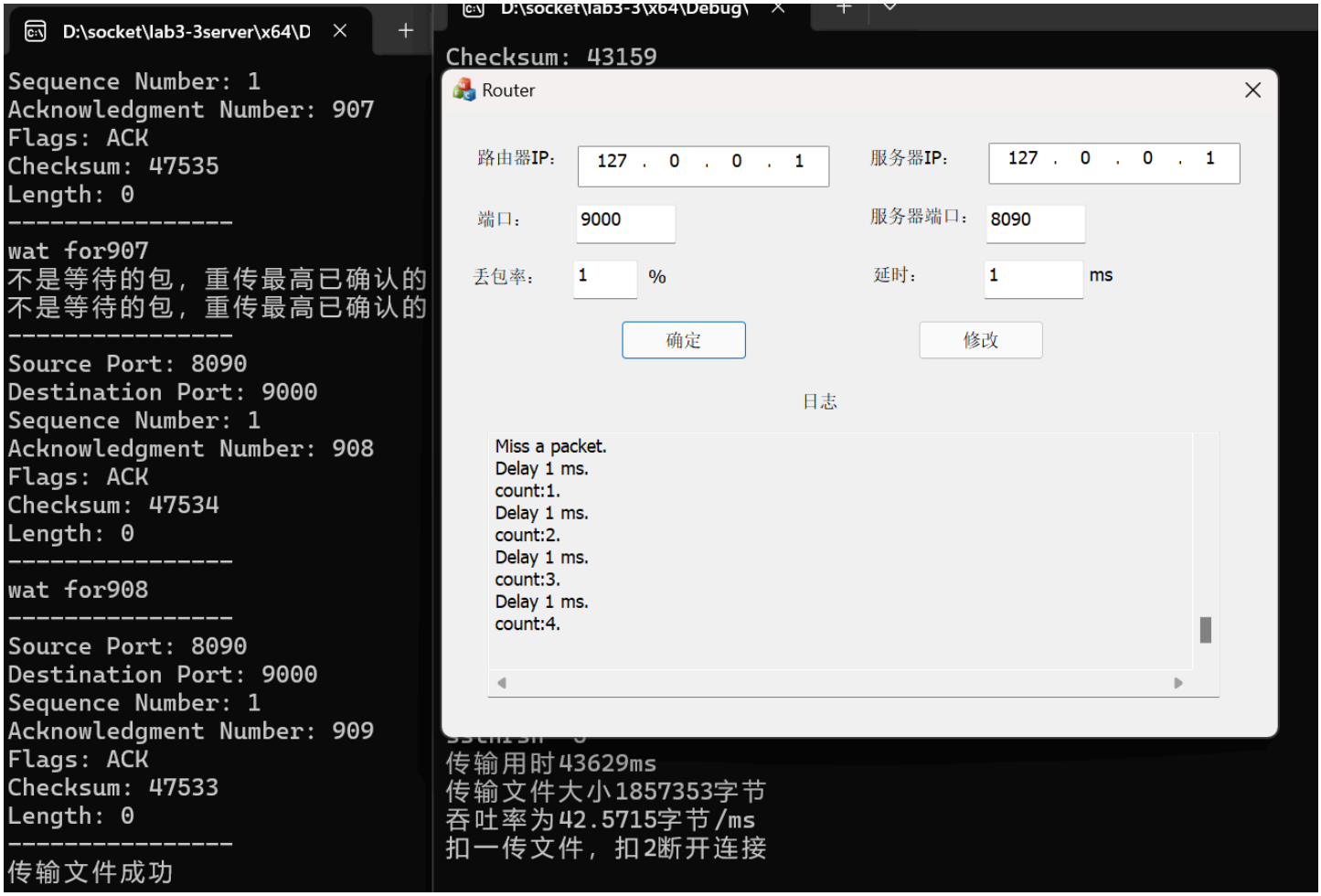
```
bool saybye()
{

        message msg[3];
        int len = sizeof(clientaddr);
        msg[0].setack();
        msg[0].head.seqnum = seq;
        msg[0].head.acknum = ack;
        if (send(msg[0]) == SOCKET_ERROR)
        {
                int error_code = WSAGetLastError();
                printf("sendto failed with error: %d\n", error_code);
        }
        cout << "发送第二次挥手的报文" << endl;
        msg[0].print();
        seq++;
        msg[1].setack();
        msg[1].setfin();
        msg[1].head.acknum = ack;
        msg[1].head.seqnum = seq;
        send(msg[1]);
        cout << "发送第三次挥手的报文" << endl;
        msg[1].print();
        seq++;
        while (1)
        {
                recvfrom(serversocket, (char*)&msg[2], sizeof(msg[2]), 0,
(SOCKADDR*)&clientaddr, &len);
                if (!msg[2].check() || !msg[2].isack() ||! (msg[2].head.acknum ==seq))
                {
                        cout << "校验错误" << endl;
                        return false;
                }
                break;
        }
        cout << "收到第四次挥手的报文" << endl;
        msg[2].print();
        cout << "终于说再见" << endl;

        return true;
}
```

四、传输测试与性能分析

（一）传输测试

1.传输测试

在这里测试了1.jpg的文件，设置丢包为1，延时为1ms.可以看到文件传输成功。在发送端输出了传输用时以及传输文件的大小。并输出了吞吐率。



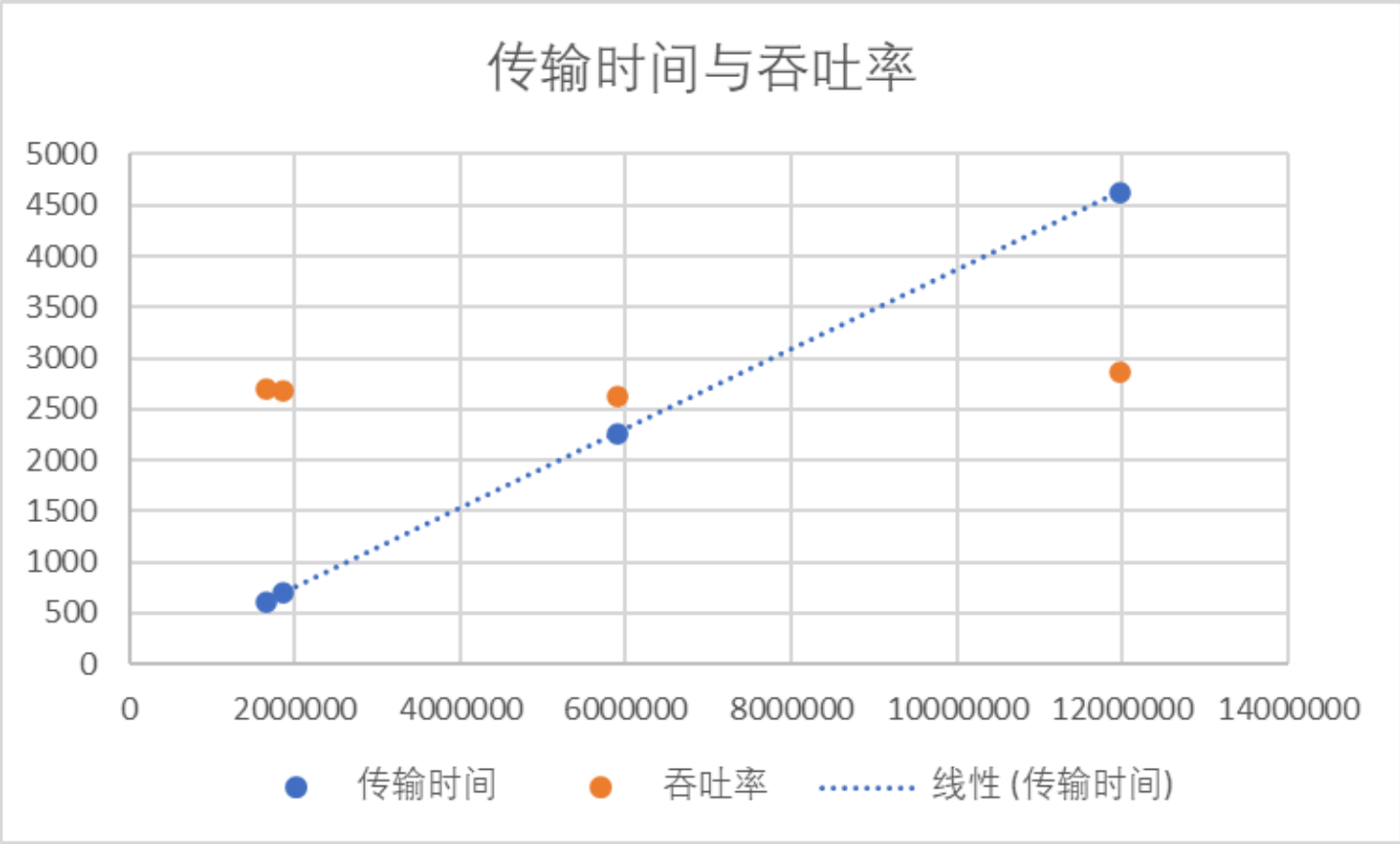
接着查看文件的属性。可以看到两个文件的大小以及文件名一致。传输无误。

接着进行抓包，可以看到对应的抓包

```
1208 228.000196   127.0.0.1   127.0.0.1   UDP   2100 8090 → 8080 Len=2068
1209 228.001169   127.0.0.1   127.0.0.1   UDP   2100 8080 → 8090 Len=2068
1210 228.001216   127.0.0.1   127.0.0.1   UDP   2100 8090 → 8080 Len=2068
1211 228.002449   127.0.0.1   127.0.0.1   UDP   2100 8080 → 8090 Len=2068
1212 228.002539   127.0.0.1   127.0.0.1   UDP   2100 8090 → 8080 Len=2068
1213 228.004116   127.0.0.1   127.0.0.1   UDP   2100 8080 → 8090 Len=2068
1214 228.004165   127.0.0.1   127.0.0.1   UDP   2100 8090 → 8080 Len=2068
1215 228.005283   127.0.0.1   127.0.0.1   UDP   2100 8080 → 8090 Len=2068
1216 228.005327   127.0.0.1   127.0.0.1   UDP   2100 8090 → 8080 Len=2068
1217 228.006350   127.0.0.1   127.0.0.1   UDP   2100 8080 → 8090 Len=2068
1218 228.006405   127.0.0.1   127.0.0.1   UDP   2100 8090 → 8080 Len=2068
1219 228.007348   127.0.0.1   127.0.0.1   UDP   2100 8080 → 8090 Len=2068
1220 228.007396   127.0.0.1   127.0.0.1   UDP   2100 8090 → 8080 Len=2068
1221 228.008312   127.0.0.1   127.0.0.1   UDP   2100 8080 → 8090 Len=2068
1222 228.008347   127.0.0.1   127.0.0.1   UDP   2100 8090 → 8080 Len=2068
1223 228.009327   127.0.0.1   127.0.0.1   UDP   2100 8080 → 8090 Len=2068
```

（二）性能分析

接下来我们统计了所给测试文件进行传输时的吞吐率以及传输时延。得到下面的对应图表。



可以看到随着文件大小，传输时间会对应增加。而对于吞吐率，则维持在一个相对稳定的范围。

五、问题反思

（一）是否设置多线程

在实现的过程中，先实现了单线程情况，然后采用了多线程，经过对比发现在将发送端的接收和发送在一个线程并不会影响。并且速度相差不大。因此采用了单线程。

（二）当发生丢包时，发生ssthresh连续下降的情况

当发生丢包时，窗口中后续的包仍然会被发出，因此将会多次快速恢复，同时此时易触发超时，所以有时会导致cwnd和ssthresh下降过多，因此我们将ssthresh的最小值设置为2.从而避免过低的情况，增加之后传输速率有所提高。

(三)重传机制

在起初，我们设置将整个窗口的包从base进行传输，但这会造成不必要的消耗。因此我们采用将next_seq挪到base_seq的方式，而不是再超时的时候直接传输。这样使得我们消耗减少，效率增加。