计算机网络第三次实验报告

Lab3-2 基于 UDP 服务设计可靠传输协议

2212422 孙启森

一、实验内容

实验3-2：在实验3-1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。

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lab3-1的基础上，实现了滑动窗口机制。采用GBN流水线协议，使用固定窗口进行传输，同时采用了累计确认。

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. 滑动窗口与累积确认

- 回退N：*Go-Back-N (GBN)*
  - 允许发送端发出N个未得到确认的分组
  - 需要增加序列号范围
    - 分组首部中增加k位的序列号，序列号空间为$[0, 2^k-1]$
  - 采用累积确认，只确认连续正确接收分组的最大序列号
    - 可能接收到重复的ACK
  - 发送端设置定时器，定时器超时时，重传所有未确认的分组

采用GBN协议，设置固定窗口大小，通过滑动窗口进行传输。

发送端

- 设置基序号和窗口大小，用来进行窗口的滑动。其中base_seq对应了已经发出并被确认的包，而next_seq则对应了发出的包。当接收到满足条件的包时，则将base_seq进行更新。对于next_seq，当其在窗口范围之内时，则将其发出，并进行更新。
- 发送端接收到接收端回应的ack时，会进行判断其是否是大于base_seq的包，若是则将base_seq进行更新。同时重置计时器。

- 若是一直没有收到符合条件的包，则会导致计时器超时，这是将会重传从base_seq 到next_seq间所有的包。

接收端

- 在接收端维护了一个waitack，用来进行累积确认，当收到的包是自己想要的包时，就会发送对应的ack包，若是不满足，则会重传按序正确接收的最高序号分组

4. 超时重传

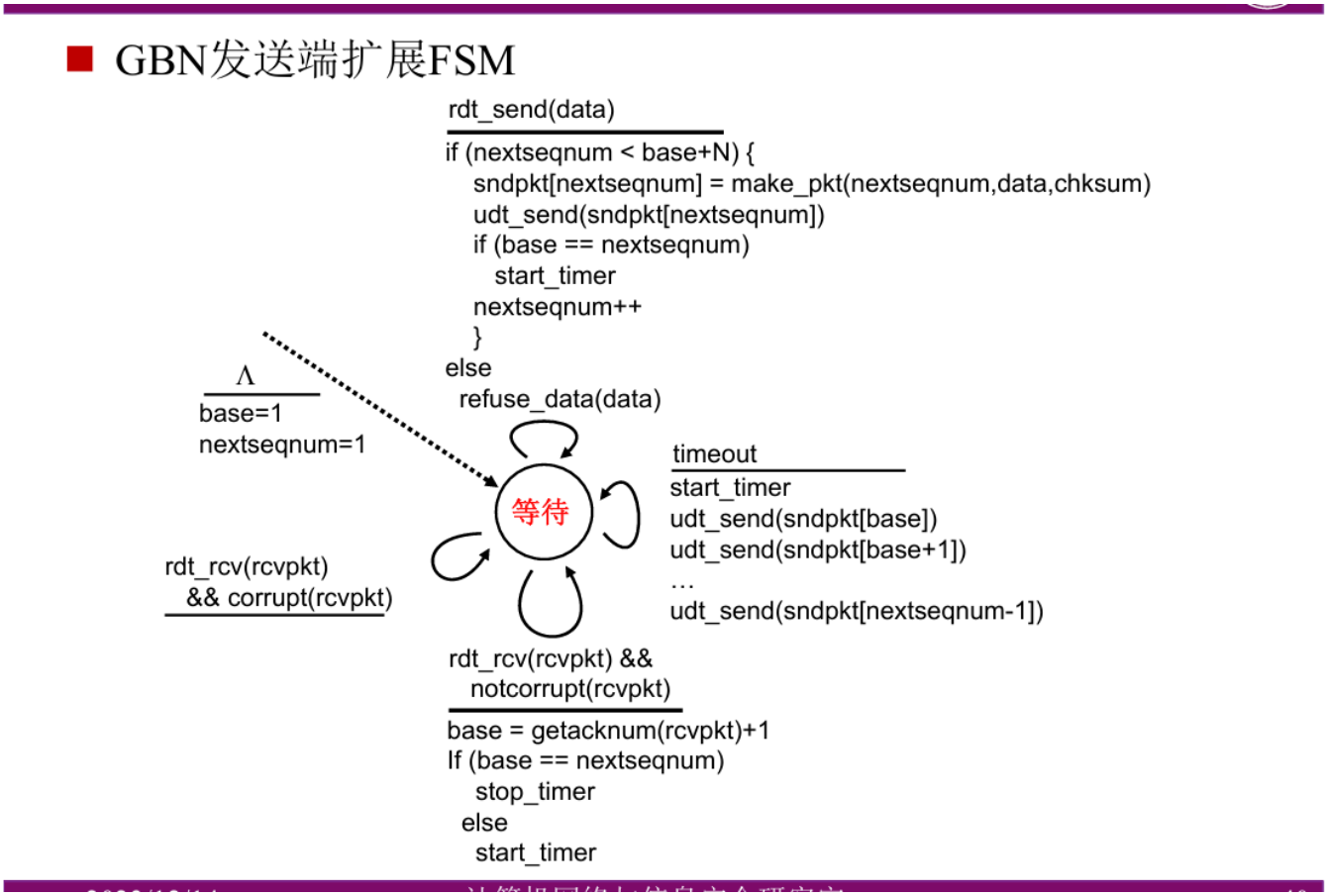
采用记时机制，每当收到正确的ack包时就重置计时器，若对应时间未能收到，则重传从base_seq到next_seq间所有的包。

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

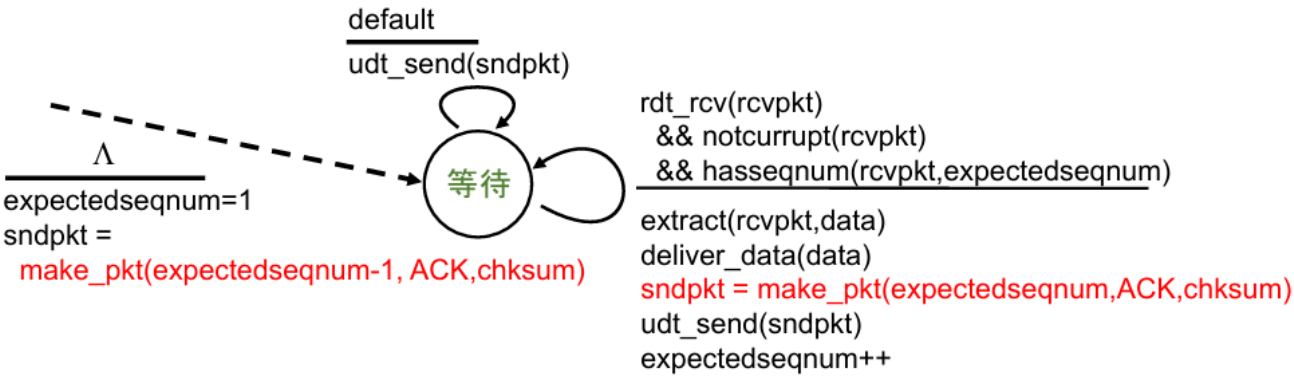
（1）发送端



这里状态机原理与先前所述一致，不再赘述。

（2）接收端

- GBN接收端扩展FSM
  - 只使用ACK，确认按序正确接收的最高序号分组
    - 会产生重复的ACK，需要保存希望接收的分组序号 (expectedseqnum)
  - 失序分组（未按序到达）处理
    - 不缓存、丢弃
    - 重发ACK，确认按序正确接收的最高序号分组



接收端同理。

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;
```

之后则将文件信息存放到对应的缓冲区中，然后使用滑动窗口进行传输。这里采用单线程实现接收和发送。也可采用多线程，但采用多线程时，发现反而有时因为加锁导致时间减慢，因此这里不再附上

多线程。因为采用了累积确认，因此当收到ack包时，滑动窗口即滑动到ack所确认的包，即baseseq=ack-1。因为此前的包都已经被确认完毕。

```
message ack_msg;
        int lastack = 0;
        int count = 0;
        filestart_time = clock();
        while (true) {
                int flag = 0;
                if (next_seq < base_seq + window && next_seq <= num - 1) {
                        send(sendbuffer[next_seq]);
                        cout << next_seq << "/" << num - 1 << endl;
                        cout << "window" << base_seq << "-" << base_seq + window << endl;
                        sendbuffer[next_seq].print();
                        if (base_seq == next_seq) {
                                start_timer();
                        }
                        next_seq++;
                }
                if (is_timeout()) {
                        cout << "已超时，重传" << base_seq << "--" << next_seq << endl;
                        for (int i = base_seq; i < next_seq; i++) {
                                send(sendbuffer[i]);
                        }
                        start_timer();
                }

                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)
                                {
                                        lastack = ack_msg.head.acknum;
                                        start_timer();
                                        base_seq = ack_msg.head.acknum - 1 - head_seq;
                                        if (ack_msg.head.acknum == seq)
                                        {
                                                return true;
                                        }
                                }
                        }
                }
        }
```

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

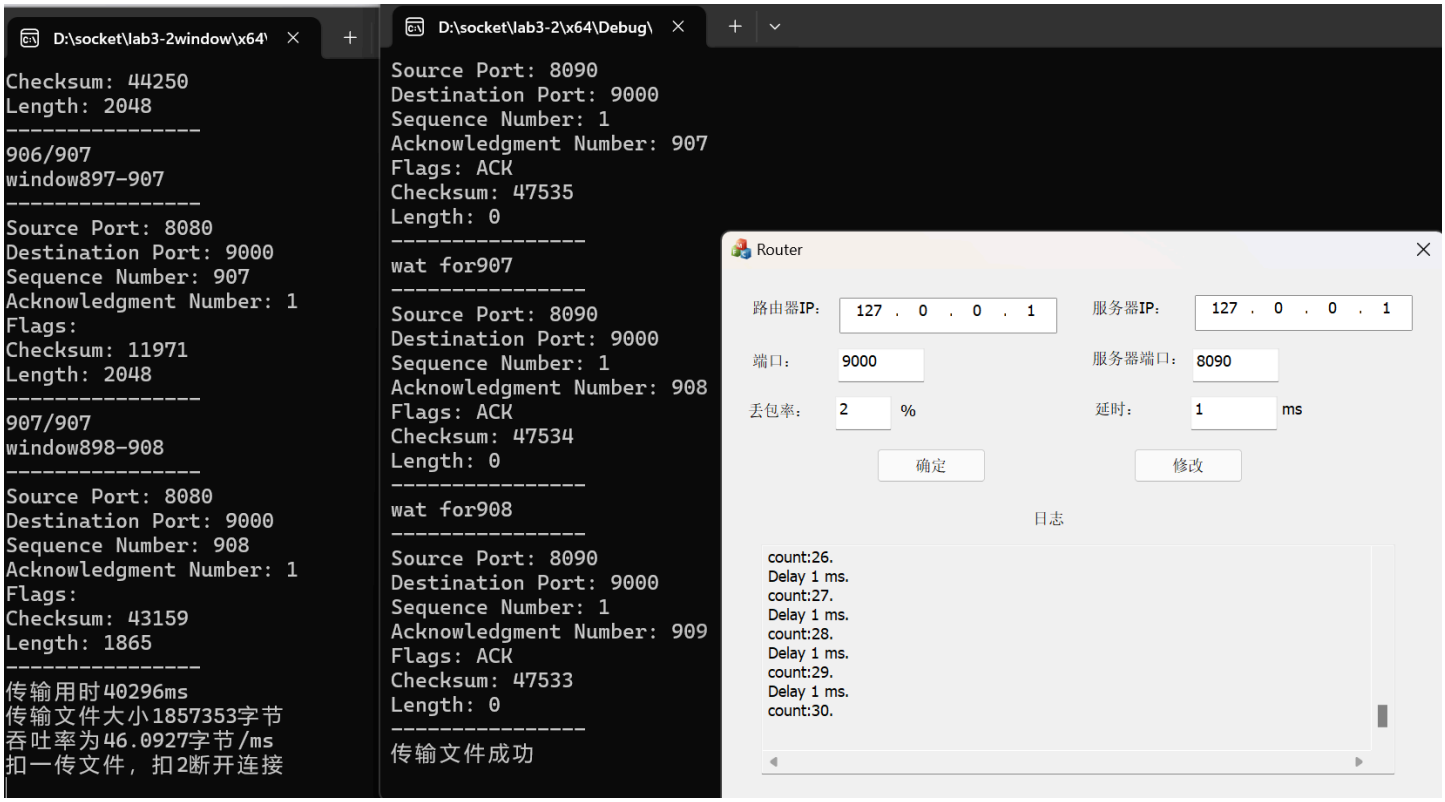
```
bool saybye()
{

        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

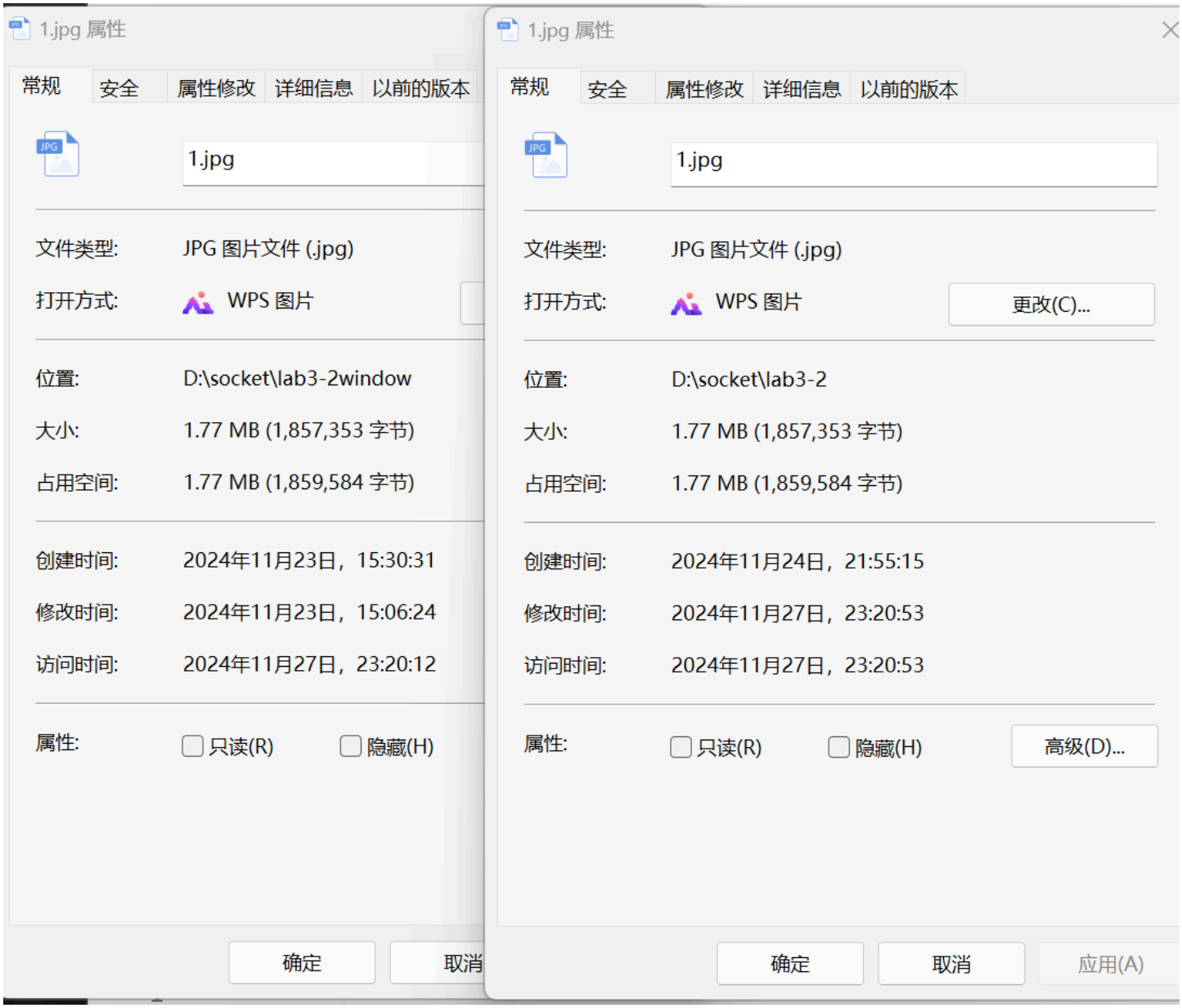
（一）传输测试

1.传输测试

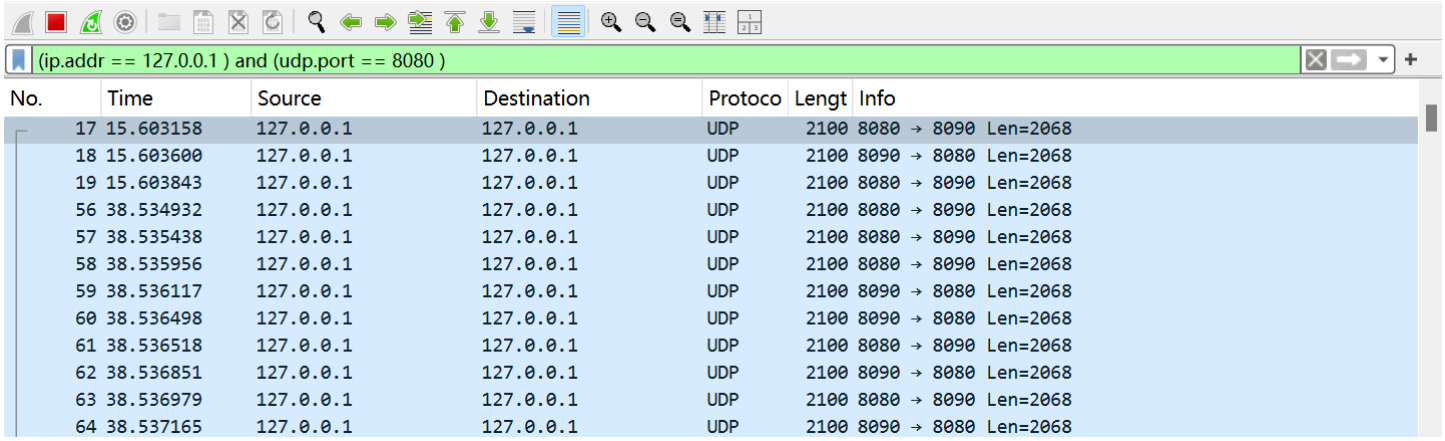
在这里测试了1.jpg的文件，设置丢包为1，延时为1ms.可以看到文件传输成功。在发送端输出了传输用时以及传输文件的大小。并输出了吞吐率。



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



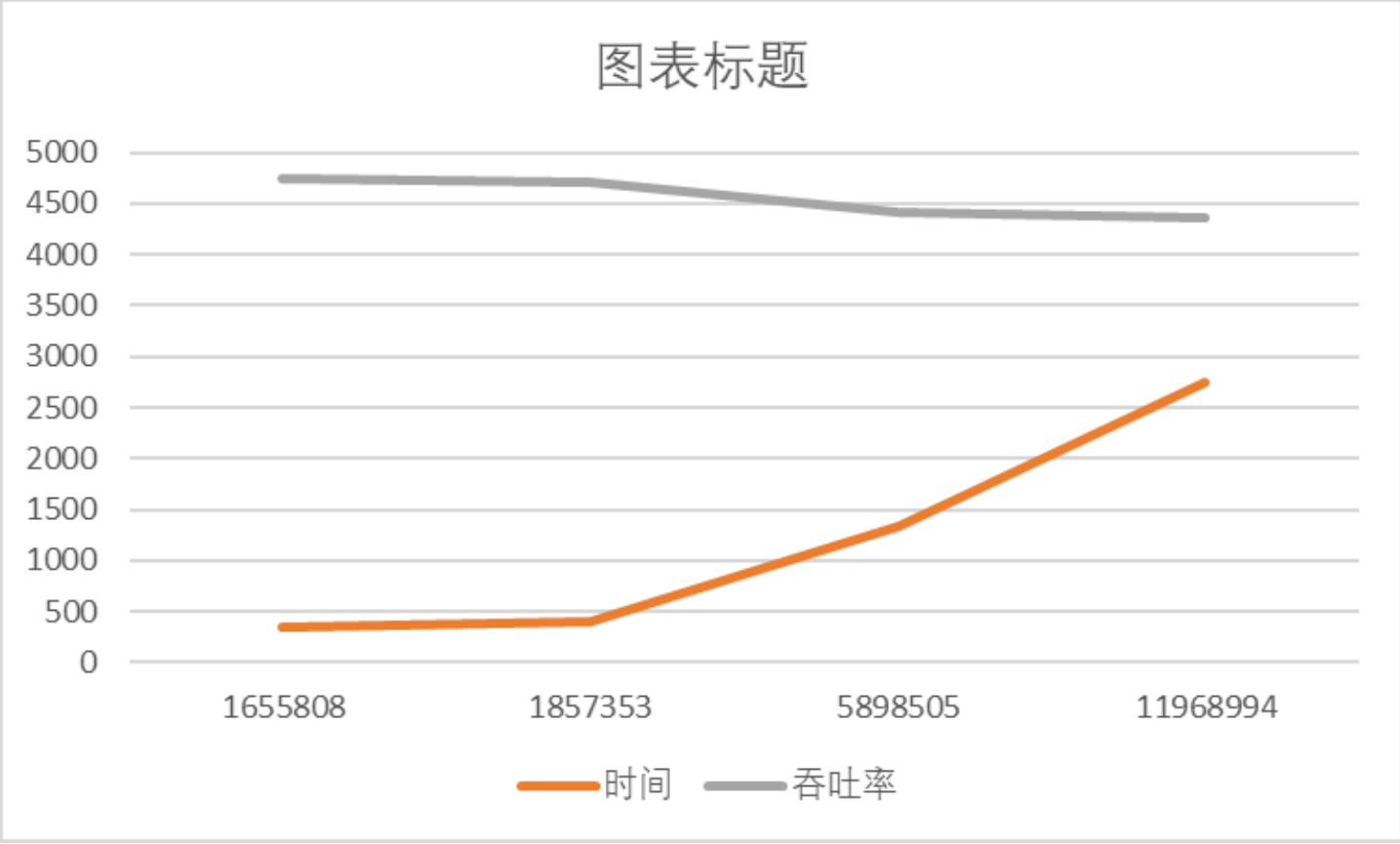
2.丢包重传测试

查看过程中发生的丢包，可以看到发送端会重传，窗口内base到nextseq之间的包。接收端会进行判断确认。

```
Source Port: 8080
Destination Port: 9000
Sequence Number: 896
Acknowledgment Number: 1
Flags:
Checksum: 46946
Length: 2048
----------------
已超时，重传886--896
896/907
window887-897
----------------
Source Port: 8080
Length: 0
----------------
wat for888
不是等待的包，重传最高已确认的
不是等待的包，重传最高已确认的
不是等待的包，重传最高已确认的
不是等待的包，重传最高已确认的
不是等待的包，重传最高已确认的
不是等待的包，重传最高已确认的
不是等待的包，重传最高已确认的
不是等待的包，重传最高已确认的
不是等待的包，重传最高已确认的
----------------
```

（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）对于如果服务器端发送的ACK丢失的情况

因为我们采用的累积确认，所以只要我们收到一个ACK，即证明此前所有的包都被正确接收，所以我们只需要将窗口的起点挪到最近接收到的最高已确认的前一个即可。