



南开大学
Nankai University

南 开 大 学

网 络 空 间 安 全 学 院

计算机网络实验报告

Lab3-3 选择确认实现可靠传输

2113946 刘国民

年级：2021 级

专业：信息安全

2023 年 12 月 14 日

目录

一、 实验内容	1
二、 协议设计	1
(一) 报文格式	1
(二) 差错检验	2
(三) 建立连接	3
(四) 传输数据	3
1. 接收端逻辑	3
2. 发送端逻辑	4
3. 情形分析	4
(五) 关闭连接	4
三、 代码实现	5
(一) 发送端	5
(二) 接收端	10
四、 实验结果	11
五、 遇到的问题 & 解决方法	14
六、 思考与总结	15

一、 实验内容

本次实验依旧采用滑动窗口机制，并且在实验 3-2 的基础上，引入选择确认。即接收端在接收到失序的数据包时，会缓存下该数据包，并通知发送端已经缓存下该包，后续超时重发的时候不必重发该包。选择确认能够有效避免因为一个包的丢失，导致重发整个窗口数据包的问题发生。

二、 协议设计

协议设计分为报文格式、差错检验、建立连接、数据传输和关闭连接五个部分。

(一) 报文格式

本次实验采用了两种不同的数据报格式，以发送端为例：

```
1 struct Send_Datagram {
2     bool ack, syn, fin, sack;
3     uint16_t checksum; // 16位校验和
4     long long seqnum, acknum, sacknum;
5     int DataLen; // DataLen <= MaxBufferSize
```

```
6     char data[MaxBufferSize] = { 0 };
7 }SendData;
8
9 struct Receive_Datagram {
10     bool ack, syn, fin, sack;
11     uint16_t checksum; //16位校验和
12     long long seqnum, acknum, sacknum;
13     // 不需要数据缓冲区
14 }ReceiveData;
```

实验中定义了两种不同的数据报结构体，即发送数据报和接收数据报。考虑到实验情景为一端发送数据，另一端只负责接收数据并返回 ack，所以在接收数据报结构体中，本次实验去除了数据缓冲区，只包括 ack,syn,fin,sack 等必要的标志位、校验和以及序列号。注意标志位部分引入了 sack，下面我们给出标志位以及序列号的含义：

- ack: 表示这是一个 ack 回复的数据包。接收端在回复确认报文时，将该标志位置为 true，发送端只在建立或者关闭连接时使用该标志位，实际传输时无需考虑。
- syn: 建立连接时使用的标志位，表示一方请求与另一方建立连接。
- fin: 关闭连接时使用的标志位，表示一方请求与另一方断开连接。
- sack: 收到失序数据包时使用的标志位。
- seqnum=x: 在传输过程中表示发送端发送了序号为 x 的数据包，这点与面向字节的 TCP 协议不同。和关闭连接中这个序号单独使用，后续相关部分会进行说明。
- acknum=x: 在传输过程中表示接收端成功接收序号小于或等于 x 的数据包，这点也与面向字节的 TCP 协议不同。建立和关闭连接中这个序号单独使用，后续相关部分会进行说明。
- sacknum=x: 当 sack=true 的时候才使用的序号，表示收到了序号为 x 的失序数据包。

序列号方面实验仍然采用类似于 TCP 协议的非循环序列号，直接用一个 long long 类型保存即可。

(二) 差错检验

差错检验使用以下算法实现：

1. 将数据报按 16bit 为一组进行分组，不足一组的用 0 补齐
2. 将 checksum 字段置 0
3. 按位累加所有组的值，每次相加如果最高位有溢出则补到 16 位数的最低位上
4. 将上述步骤的结果取反后填入 checksum 字段

实验中用 RECEIVE_CHECK 和 SEND_CHECK 两个宏来标识是否取反，发送的时候需要取反，接受时则不用取反，接收时验证算出的校验和与数据包的校验和字段相加是否为 0xffff 即可。同时由于实验定义了两种不同的数据报文，所以在代码实现中利用函数重载的形式，分别实现了对两种报文的差错检验。

(三) 建立连接

依旧采用以下方式建立连接：

1. 首先发送方发送一个空数据报给接收方（全为 0），仅将 syn 标志位置为 1，表示发送方请求建立连接。同时阻塞等待接收方的 syn+ack 包。
2. 接收方持续阻塞等待发送方的 syn 包，如果成功接收到 syn 标志位为 1 的数据包，则将一个空包的 syn、ack 和 acknum 位置为 1 后发回，告诉发送方成功收到 seqnum 为 0 的 syn 包。
3. 发送方成功收到 syn+ack 包后，发回一个 ack 包，同时设置 seqnum 和 acknum 为 1，表示成功收到 seqnum 为 0 的 syn+ack 包，同时序列号自增 1。至此连接建立完成，

建立连接的具体流程基本与 TCP 三次握手一致，在这里双方的初始序列号 ISN 均取 0。

(四) 传输数据



图 1: 滑动窗口

传输数据部分的基本逻辑与上次实验相同，类似于 GBN 协议，只是在此基础上扩展了支持选择确认的功能，这里不再详细说明原有协议本身，重点阐述如何将选择确认补充到原有协议设计中。

1. 接收端逻辑

在上次实验中，接收端维护一个期望序列号变量。如果收到的数据包序号等于期望序列号则写回文件，发回回复报文并让期望序列号加 1；如果收到的数据包序号大于或者小于期望序列号，则发回一个 acknum 为期望序列号-1 的回复报文，告诉发送端已经收到在此之前的，现在要期望序列号的数据包。本次实验中，我们对这个逻辑进行修改，即：

1. 如果接收端收到的数据包大于期望值，则缓存下该数据包，不再发回 ack 包，而是将 sack 置为 1，同时 sacknum 设置为收到数据包的数据号，即告诉发送端收到了序号为 sacknum 的失序包。
2. 如果接收端收到的数据包等于期望值，则存下数据包，同时期望值 +1。注意，这里还需要继续判断，如果期望值的数据包在之前已经缓存过了，则继续更新期望值，继续加 1，直到期望序列号的数据包没有缓存过。最后再 acknum 期望值-1，表示在此之前（包含该序号）的数据包都已收到或缓存。
3. 如果接收端收到的数据包小于期望值，与上次实验类似，直接丢弃即可。

2. 发送端逻辑

发送端的逻辑变化主要在于接收 ack 回复和重发窗口数据包的时候。在之前的实验中，发送端为 Base 数据包启动定时器，如果收到的 ack 包的回复序列号 $\text{acknum} \geq \text{Base}$ 则重置定时器，并且更新 Base 的值。本次实验由于新引入了 sack，所以逻辑也需要更改。即如果收到了 sack 包，则表示序号为 sacknum 的数据包已经被接收端成功缓存。后续涉及重发窗口的时候，遇到该序号的数据包直接跳过即可，不需要重发。注意收到 sack 的时候不能重置定时器，仍然需要为 Base 包的回复报文计时。

3. 情形分析

基于设计的协议，我们分析丢失、错误等各种情形下该协议是怎么运转的，遇到问题后能否正确恢复。

1. 发送数据包丢失/出错: 序号为 x 的数据包丢失后，之前的数据包都已成功收到 ack 的回复报文，则窗口右移到该包，此时 $\text{Base}=x$ ，开启定时器后，假设后续窗口中的所有数据包都成功抵达接收端。接收端收到后失序缓存下这些数据包，并发回 sack 报文，但发送端仍然未等到 Base 包的回复报文，最终超时重传窗口，而窗口中后续数据包都已成功抵达，发送端重发 Base 包即可，此时接收端成功收到后，由于该包填上了空缺，回复报文的 acknum 变为 $\text{Base}+\text{WindowSize}$ 。发送端收到该报文后则知道在此之前（包含该序号）的数据包都成功收到，直接更新 Base 包即可，继续传输。在这里我们考虑下更极端的情形，假如 sack 报文丢失或出错，那么发送端则不知道接收端已经成功缓存后续数据包，重发窗口时会再次发送这些数据包，根据协议设计，接收端仍然按照三种情况判断即可，不会影响传输；再假如填上空缺后接收端回复的 ack 报文丢失了怎么办？这里其实跟下面一种情况是一样的，直接看下面的逻辑即可。
2. ack 丢失/出错/提前超时: 假设接收端收到了 $\text{seqnum}=x$ 的数据包，此时发送一个 $\text{acknum}=x$ 的回复报文给发送端，并且期望值 $+1$ ，如果该包丢失、出错或者提前超时了，但接收端也收到了后续的数据包 ($x+1, x+2, \dots$)，之后的回复报文没有丢失、出错，则发送端收到的 $\text{acknum} > \text{Base}$ ，直接更新 Base，窗口右移即可；如果后续数据包丢失出错或者回复的 ack 丢失出错，则 Base 包超时，发送端重发窗口。这个过程中没有涉及选择确认和失序缓存，发送端发送后，接收端收到序号小于期望值的数据包，给出期望值-1 的回复报文，则发送端可以直接调整 Base，继续正常传输。

(五) 关闭连接

关闭连接也使用与 TCP 协议类似的流程，如下图所示：

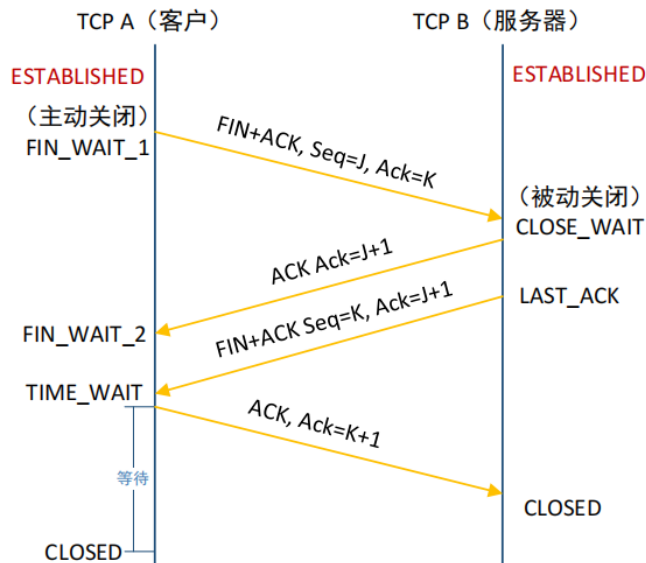


图 2: 关闭连接

如上图所示，但序列号 J 和 K 在此处跟三次握手类似，均取为 0。

三、 代码实现

代码实现部分主要分析传输数据和差错检验部分的代码，Socket 初始化、建立和关闭连接的代码与上次实验基本相同，不再赘述。建立连接时，我将发送数据报的总数放入了第一次握手 SYN 报文的空闲缓冲区，将文件名放入了第三次握手 ACK 报文的空闲缓冲区，一起发送给接收端。

(一) 发送端

与上次实验类似，发送端仍然使用三个线程，具体如下：

- 发送端用主线程 main 和两个创建的子线程 Resend 和 Timer 完成协议给出的功能和操作。
- 主线程 main 负责发送数据包，即一旦窗口左边界右移，主线程负责发送新的数据包给接收端。
- 子线程 Resend 负责持续检测是否“超时”，“超时”则重传，这里的“超时”指的是在指定时间内没有收到正确的 acknum（即 $\text{acknum} \geq \text{Base}$ ）。
- 子线程 Timer 负责为 Base 包开启定时器，一旦超时则通知线程 Resend，这里的“通知”是通过全局变量的设置来实现，稍后会在具体的代码实现中看到。如果收到 sack，则记录 sacknum 的值，重传的时候跳过该包，这里的“记录”通过一个布尔数组来实现。

首先给出差错检验函数：

```

1 uint16_t Checksum(Send_Datagram d, int flag) {
2     unsigned short* check = (unsigned short*)&d; // 两字节为一组来处理
3     int count = Send_DatagramLen / sizeof(short); // 一个数据报有 count 组
4     d.checksum = 0;

```

```

5     unsigned long sum = 0;
6     while (count--) {
7         sum += *check;
8         check++;
9         if (sum & 0xffff0000) {// 最高位有溢出
10            sum = sum & 0xffff;
11            sum++;
12        }
13    }
14    if (flag == SEND_CHECK) {
15        return (~sum) & 0xffff; //取反
16    }
17    else {
18        return sum & 0xffff;
19    }
20 }

21
22 // 利用函数重载
23 uint16_t Checksum(Receive_Datagram d, int flag) {
24     unsigned short* check = (unsigned short*)&d; //两字节为一组来处理
25     int count = Receive_DatagramLen / sizeof(short); // 一个数据报有 count 组
26     d.checksum = 0;
27     unsigned long sum = 0;
28     while (count--) {
29         sum += *check;
30         check++;
31         if (sum & 0xffff0000) {// 最高位有溢出
32            sum = sum & 0xffff;
33            sum++;
34        }
35    }
36    if (flag == SEND_CHECK) {
37        return (~sum) & 0xffff; //取反
38    }
39    else {
40        return sum & 0xffff;
41    }
42 }

```

代码基本按协议设计中给出的算法实现，两个函数逻辑相同，只是函数形参中数据报类型不同，直接使用的函数重载，没有使用 C++ 的模版特性。我们给出下列全局变量：

```

1 // 多线程共享数据，写数据的时候用互斥锁
2 int Flag_Resend = 0; // 表示recvfrom的数据是否超时
3 int Flag_Timer = 0; // 表示是否需要启用定时器服务
4 int SeqNumber = 0; // 记录一共有多少个数据包，seqnum:0--SeqNumber-1
5 long long Base = 0; // 初始序列号从0开始
6 long long NextSeq = 0;
7

```

```

8 vector<Send_Datagram> v; //用来保存数据包序列
9 bool* flag_cache; // 指示某个包是否被缓存

```

从注释中可以看到, Flag_Resend 用于表示收到的数据是否超时, 只要子线程 Timer 发现超时了就将该标志位置为 1, 这时候子线程 Resend 就可以检测到, 从而开始重发窗口中的数据包。Flag_Timer 用于表示是否需要开启定时器, 即主线程或者 Resend 线程发送 Base 包, 或者窗口右移更新 Base 值的时候, 就将该标志位置为 1, Timer 线程就会开始启用定时器, 持续接收可能的 ACK 回复。而 Base, NextSeq 和 SeqNumber 为全局变量, 为三个线程共同使用。

我们用一个 bool 数组来指示某个包是否被缓存, bool[i]=true 则说明序列号为 i 的数据包被缓存。再用一个 vector 容器来保存所有要传输的数据包, 在传输之前我们先把原始文件以 4096 字节为一组放入 vector 中, 具体代码如下:

```

1 // 移动文件指针到文件末尾
2 fseek(p, 0, SEEK_END);
3 long long FileLen = ftell(p);
4 fseek(p, 0, SEEK_SET);
5 long long temp = FileLen;
6 // 数据以4096字节为单位放入vector序列
7 while (temp>0) {
8     SendData = { 0 };
9     fread(SendData.data, MaxBufferSize, 1, p);
10    SendData.DataLen = temp < MaxBufferSize ? temp : MaxBufferSize;
11    SendData.seqnum = SeqNumber++;
12    v.push_back(SendData);
13    temp -= SendData.DataLen;
14 }

```

fread 将数据读取到发送数据包缓冲区中, 将相关信息封装好后放入容器, 之后涉及重传的时候, 直接从容器中拿取数据包重发即可, 对于窗口的滑动, 直接增加 Base, NextSeq 等变量后通过 vector 随机访问即可, SeqNumber 用来记录一共有多少个数据包。接下来我们逐个分析三个线程的具体代码。

```

1 while (true) {
2     if (NextSeq == SeqNumber) { // 发送完毕
3         break;
4     }
5     if (NextSeq < Base + WindowSize) {
6         SendData = v[NextSeq];
7         Send();
8         if (NextSeq == Base) { // 第一个包
9             lock_guard<mutex> lock(mtx);
10            Flag_Timer = 1;
11        }
12        {
13            lock_guard<mutex> lock(mtx);
14            NextSeq++;
15        }
16        Sleep(100); // 每次发送睡眠100毫秒, 避免短时间内发送大量数据包给接收方
17    }

```


18 }

上述代码是 main 函数中发送数据的核心代码，整体逻辑是持续检测 NextSeq 是否小于 Base+WindowSize，即发送的数据包是否到达右边界，如果没到达就持续发送并增加 NextSeq 的值。如果发送的是 Base 包则需要设置 Timer 标志位，通知子线程启动定时器。其中 v 即是上述提到的 vector 容器，需要注意在这个过程中，对多线程的共享数据（这里为 Flag_Timer）进行写操作时，需要使用互斥锁以阻塞子线程进行操作，避免多线程同时进行写操作从而引发错误。每次发送后，使用 sleep 函数睡眠 100 毫秒，避免短时间发送大量数据包给接收方。如果 NextSeq 等于 SeqNumber，则代表窗口右边界已经到达容器末尾，此时主线程直接 break 退出循环即可。后续可能涉及到的重传由 Resend 完成。

```

1 void Resend() { // 只要检测到 Flag_Resend 变为 1 就重发 Base--(NextSeq-1)
2     while (true) {
3         if (Flag_Resend == 1) {
4             BackCounter++;
5             int curr = 0;
6             for (int i = Base; i <= NextSeq - 1; i++) {
7                 // 首先根据失序数据包序列判断是否需要重发
8                 if (flag_cache[i]) continue;
9                 SendData = v[i];
10                Send();
11                if (i == Base) { // Base 包
12                    lock_guard<mutex> lock(mtx);
13                    Flag_Timer = 1; // 启动定时，通知 Receive 线程
14                }
15                Sleep(100); // 重发的时候也不要一股脑全发过去
16            }
17            {
18                lock_guard<mutex> lock(mtx);
19                Flag_Resend = 0;
20            }
21        }
22        if (Base == SeqNumber) { // 传输完毕
23            cout << "[提示]: ReSend 线程正确结束" << endl;
24            return; // 结束进程
25        }
26    }
27 }

```

上述代码是子线程 Resend 绑定的函数，核心逻辑就是 while 循环持续检测 Flag 标志位，如果为 1 就开始重传窗口，重传数据也是通过容器 v 获得，BackCounter 用于记录整个传输过程中的重传次数。与主线程类似，发送 Base 包需要启动定时器，遍历窗口数据包时，如果 flag_cache[i] 为 true，则说明该序号数据包已经被接收端失序缓存，通过 continue 语句跳过该包。写操作使用互斥锁。当 Base=SeqNumber，即左边界也到达文件末尾时，此时退出循环。

```

1 void Timer() { // 持续监测 Flag_Timer，只要 Flag_Timer=1 就对 Base 包启动定时器
2     while (true) {
3         if (Base == SeqNumber) { // 传输完毕
4             cout << "[提示]: Timer 线程正确结束" << endl;

```

```

5         return;// 结束进程
6     }
7     clock_t start, end;
8     if (Flag_Timer == 1) {
9         int len = sizeof(Receiver_addr);
10        start = clock();
11        while (true) {
12            int recv = recvfrom(SenderSocket, (char*)&ReceiveData,
13                                Receive_DatagramLen, 0, (struct sockaddr*)&Receiver_addr,
14                                &len);
15            int check = Checksum(ReceiveData, RECEIVE_CHECK);
16            // recvfrom非阻塞, 如果接收到的数据无误且ack大于Base,则移动窗口左边界, 用队列来实现
17            // 同时主线程监测到NextSeq<Base+WindowSize, 传输新增的右边界窗口, 从而实现滑动窗口的功能
18            if (recv != -1 && ReceiveData.acknum >= Base && (check ^ ReceiveData.checksum) == 0xffff && ReceiveData.ack == true) {
19                cout << "收到acknum为" << ReceiveData.acknum << "的回复" << endl;
20                lock_guard<mutex> lock(mtx);
21                Base = ReceiveData.acknum + 1;
22                break;//跳出内层循环后, 因为Flag_Timer仍然为1, 会为新的Base设置定时器
23            }
24            if (recv != -1 && (check ^ ReceiveData.checksum) == 0xffff && ReceiveData.sack == true) {
25                // 接收端将已经缓存的失序数据包通知给发送端
26                flag_cache[ReceiveData.sacknum] = 1;
27            }
28            end = clock();
29            if (end - start >= timeout) { // 超时
30                lock_guard<mutex> lock(mtx);
31                Flag_Resend = 1; // 写操作使用互斥锁, 通知Resend重传整个窗口
32                Flag_Timer = 0; // 重传的时候再设置为1
33                break; // 跳出内层循环, 此时因为Flag_Timer为0, 不需要再监测, 等待Resend重发Base包的时候设置Flag_Timer
34            }
35        }
36    }

```

与实验 3-1 类似, 本次实验在建立连接后将 `recvfrom` 设置为非阻塞, 在传输结束开始断开连接的时候恢复为阻塞模式。上述代码即用 `while` 循环持续检测 `Flag_Timer`, 如果为 1 启动定时, 首先先用 `start` 获取当前时间, 之后再用一个 `while` 循环持续 `recvfrom`, 如果成功收到正确的数据, 且 `acknum >= Base`, 则改变 `Base`, 这里需要注意不是 `Base+1`, 而是 `acknum+1`, 因为可能收

到的 acknum 大于 Base, 同时跳出内层循环。在这里没有将 Flag_Timer 置为 0, 因为 Base 增加, 需要为新的 Base 包启动定时器; 如果成功收到数据, 但是是 sack 报文, 则设置 flag_cache 标志位; 如果一直没有收到符合要求的数据包, end 不断获取时间, 当持续时间超过设置的时间时, 将 Flag_Resend 设为 1, Resend 线程就会立即进行相应操作。同时将 Flag_Timer 设为 0, 跳出内层循环。等到 Resend 发出第一个包时, 再将 Flag_Timer 置为 1, 继续开启定时器。

(二) 接收端

新增的全局变量如下:

```
1 Receive_Datagram* Buffer;// 用来缓存数据, 最后一次性写入文件中
2 bool* flag_cache;
```

接收端也定义了两种类型的报文, 并编写了类似的差错检验函数。建立连接时, 发送端已经将总共的数据包数量发送给接收端, 所以接收端定义了一个 Buffer 数组来存下所有的数据报。同时用 flag_chahe 来指示哪些序号的数据包被放入 Buffer 数组了。主函数的核心代码如下:

```
1 while (true) {
2     ReceiveData = { 0 };
3     bool recv = Receive();
4     if (recv && ReceiveData.fin == true && ReceiveData.ack == true) {// 第一次挥手
5         break;
6     }
7     if (recv) {// 数据无误
8         if (ReceiveData.seqnum == ExpectedNum) {
9             Buffer[ReceiveData.seqnum] = ReceiveData;
10            cout << "成功接收第" << ExpectedNum << "个数据包" << endl;
11            SendData = { 0 };
12            SendData.ack = true;
13            for (int i = ++ExpectedNum; flag_cache[i] && i < SeqNumber; i++)
14                {// 有可能刚好填补上空缺的一块, 这里需要连续增加ExpectedNum
15                    ExpectedNum++;
16                }
17            SendData.acknum = ExpectedNum - 1; //期望值-1
18            Send();
19        }
20        else if (ReceiveData.seqnum > ExpectedNum) {// 收到的不是期望的数据包, 缓存并回复ACK
21            Buffer[ReceiveData.seqnum] = ReceiveData; //这里即使重复收到, 也不需要判断, 直接覆盖即可
22            cout << "第" << ReceiveData.seqnum << "个数据包失序缓存" << endl;
23            flag_cache[ReceiveData.seqnum] = 1;
24            SendData = { 0 };
25            SendData.sack = true; //通知发送端这是失序回复报文
26            SendData.sacknum = ReceiveData.seqnum;
27            Send();
28        }
29        else {
```

```

29         cout << "收到重复数据包, seqnum为: " << ReceiveData.seqnum <<
        endl;
30         SendData = { 0 };
31         SendData.ack = true;
32         SendData.acknum = ExpectedNum - 1;
33         Send();
34     }
35 }
36
37 }
38 ... .. // 省略四次挥手的内容
39 for (int i = 0; i < SeqNumber; i++) {
40     fwrite(Buffer[i].data, Buffer[i].DataLen, 1, p); // 写回文件
41 }

```

主要逻辑就是 recv 正确的数据包, 分 seqnum 大于、等于或者小于期望序列号 ExpectedNum 三种情况进行处理。如果大于则缓存数据包, 把数据包放入 Buffer 数组中, 设置 flag_cache 标志位, 同时回复一个 sack 报文给发送端; 如果等于则是期望的数据包, 同样吧数据包放入 Buffer 数组中, 同时更新期望值 ExpectedNum, 注意可能后续数据包已经缓存, 所以需要通过一个循环来判断是否需要持续更新 ExpectedNum, 最后封装好数据包发给发送端; 如果小于则直接丢弃然后回复 ACK。等到检测到 fin 和 ack 标志时则退出循环, 说明发送端开始进行第一次挥手。完成四次挥手后, 接收端再遍历整个 Buffer 数组, 把每个数据包的数据依次写回文件中, 完成传输。

四、 实验结果

该部分在路由器的丢包率为 3%, 延时为 5ms, 窗口大小为 8 的条件下进行测试传输, 将传输 4 个给定文件。打开路由器, 配置端口、IP、丢包率以及延时, 随后打开 Receiver.exe 和 Sender.exe, 传输 helloworld 文件结果如下:

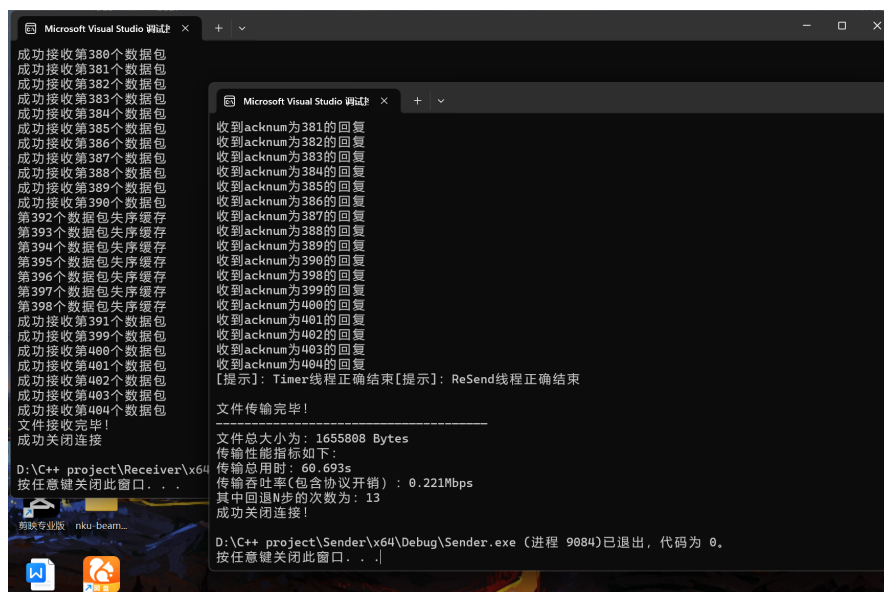
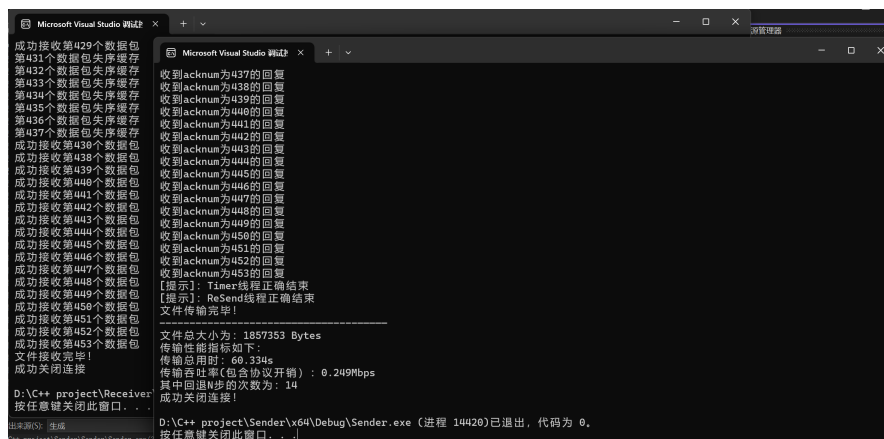


图 3: 传输 helloworld

可以看到，程序成功完成了本次传输并且正常关闭连接，在发送端目录下能看到传输的 hel-lowworld.txt 文件，大小与源文件大小相同。传输 1.jpg 文件结果如下：



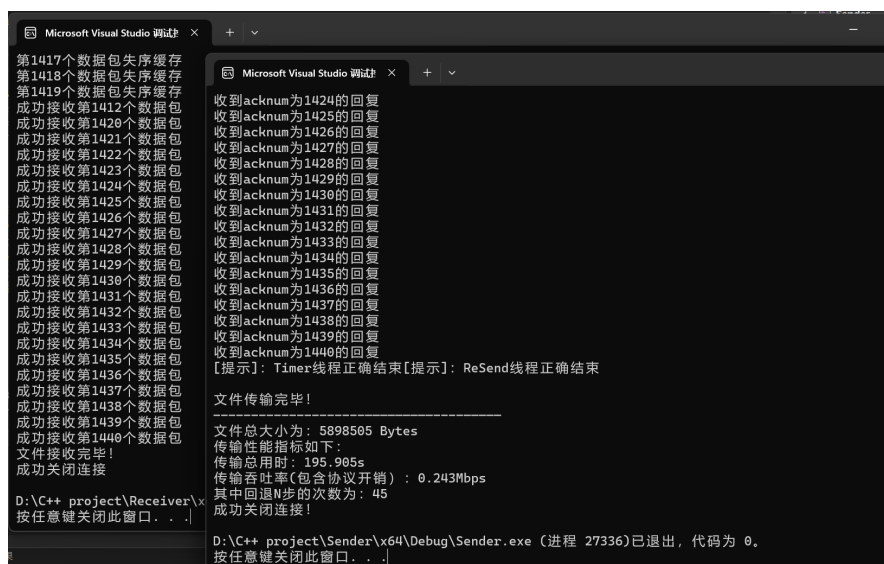
```
Microsoft Visual Studio 调试
成功接收第429个数据包
第431个数据包失序缓存
第432个数据包失序缓存
第433个数据包失序缓存
第434个数据包失序缓存
第435个数据包失序缓存
第436个数据包失序缓存
第437个数据包失序缓存
成功接收第438个数据包
成功接收第438个数据包
成功接收第439个数据包
成功接收第440个数据包
成功接收第441个数据包
成功接收第442个数据包
成功接收第443个数据包
成功接收第444个数据包
成功接收第445个数据包
成功接收第446个数据包
成功接收第447个数据包
成功接收第448个数据包
成功接收第449个数据包
成功接收第450个数据包
成功接收第451个数据包
成功接收第452个数据包
成功接收第453个数据包
文件接收完毕!
成功关闭连接
D:\C++ project\Receiver\
按任意键关闭此窗口...

Microsoft Visual Studio 调试
收到acknum为437的回复
收到acknum为438的回复
收到acknum为439的回复
收到acknum为440的回复
收到acknum为441的回复
收到acknum为442的回复
收到acknum为443的回复
收到acknum为444的回复
收到acknum为445的回复
收到acknum为446的回复
收到acknum为447的回复
收到acknum为448的回复
收到acknum为449的回复
收到acknum为450的回复
收到acknum为451的回复
收到acknum为452的回复
收到acknum为453的回复
【提示】: Timer线程正确结束
【提示】: ReSend线程正确结束
文件传输完毕!
文件总大小为: 1857353 Bytes
传输性能指标如下:
传输总用时: 60.334s
传输吞吐量(包含协议开销): 0.249Mbps
其中回退N步的次数为: 14
成功关闭连接!
D:\C++ project\Sender\x64\Debug\Sender.exe (进程 14426)已退出, 代码为 0。
按任意键关闭此窗口...

D:\C++ project\Receiver\
按任意键关闭此窗口...
D:\C++ project\Sender\x64\Debug\Sender.exe (进程 14426)已退出, 代码为 0。
按任意键关闭此窗口...
```

图 4: 传输图片 1

传输 2.jpg 文件结果如下：



```
Microsoft Visual Studio 调试
第1417个数据包失序缓存
第1418个数据包失序缓存
第1419个数据包失序缓存
成功接收第1412个数据包
成功接收第1420个数据包
成功接收第1421个数据包
成功接收第1422个数据包
成功接收第1423个数据包
成功接收第1424个数据包
成功接收第1425个数据包
成功接收第1426个数据包
成功接收第1427个数据包
成功接收第1428个数据包
成功接收第1429个数据包
成功接收第1430个数据包
成功接收第1431个数据包
成功接收第1432个数据包
成功接收第1433个数据包
成功接收第1434个数据包
成功接收第1435个数据包
成功接收第1436个数据包
成功接收第1437个数据包
成功接收第1438个数据包
成功接收第1439个数据包
成功接收第1440个数据包
文件接收完毕!
成功关闭连接
D:\C++ project\Receiver\
按任意键关闭此窗口...

Microsoft Visual Studio 调试
收到acknum为1424的回复
收到acknum为1425的回复
收到acknum为1426的回复
收到acknum为1427的回复
收到acknum为1428的回复
收到acknum为1429的回复
收到acknum为1430的回复
收到acknum为1431的回复
收到acknum为1432的回复
收到acknum为1433的回复
收到acknum为1434的回复
收到acknum为1435的回复
收到acknum为1436的回复
收到acknum为1437的回复
收到acknum为1438的回复
收到acknum为1439的回复
收到acknum为1440的回复
【提示】: Timer线程正确结束【提示】: ReSend线程正确结束
文件传输完毕!
文件总大小为: 5898505 Bytes
传输性能指标如下:
传输总用时: 195.905s
传输吞吐量(包含协议开销): 0.243Mbps
其中回退N步的次数为: 45
成功关闭连接!
D:\C++ project\Sender\x64\Debug\Sender.exe (进程 27336)已退出, 代码为 0。
按任意键关闭此窗口...
```

图 5: 传输图片 2

传输 3.jpg 文件结果如下：

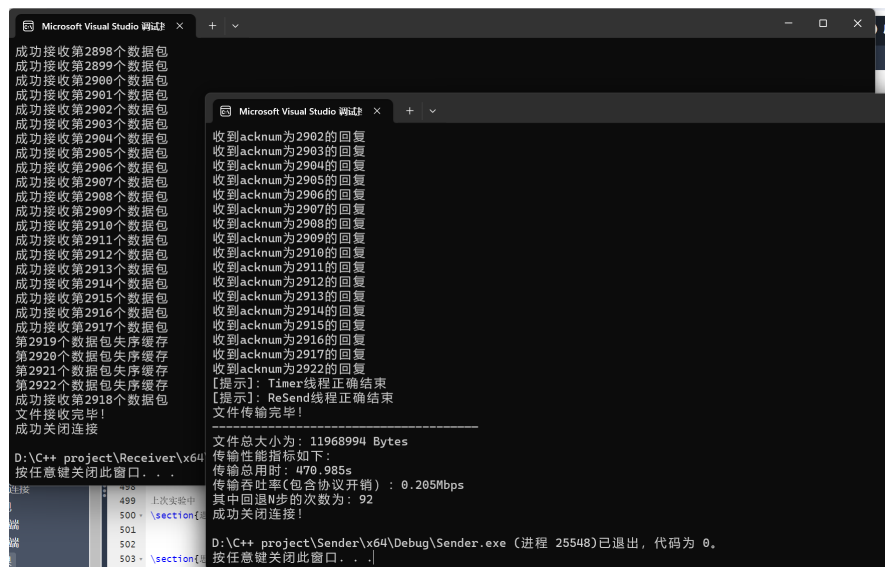


图 6: 传输图片 3

最后我们分别打开发送端和接收端的 VS 项目文件夹, 把传输的四个文件与源文件大小进行比对, 发现字节数均相同, 说明传输成功。(这里只展示 3.jpg 文件的字节数)

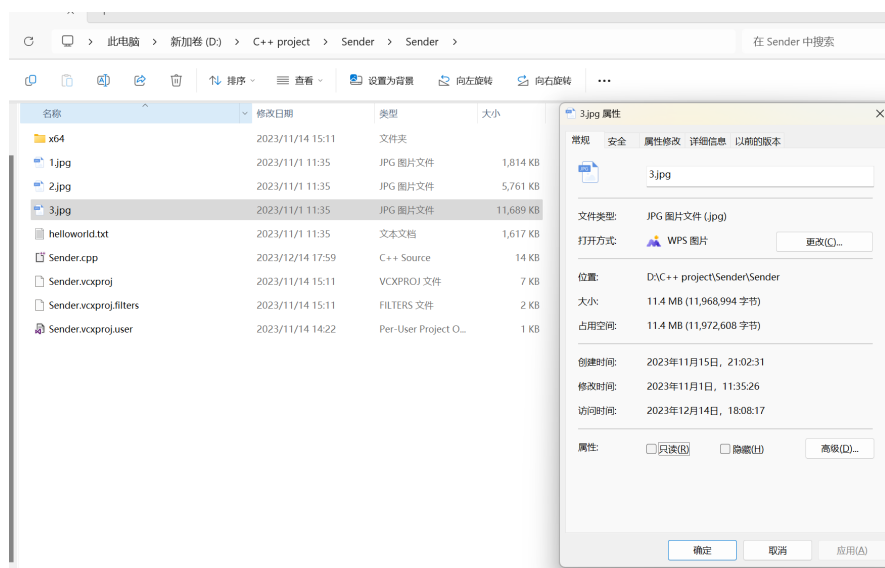


图 7: 发送端目录

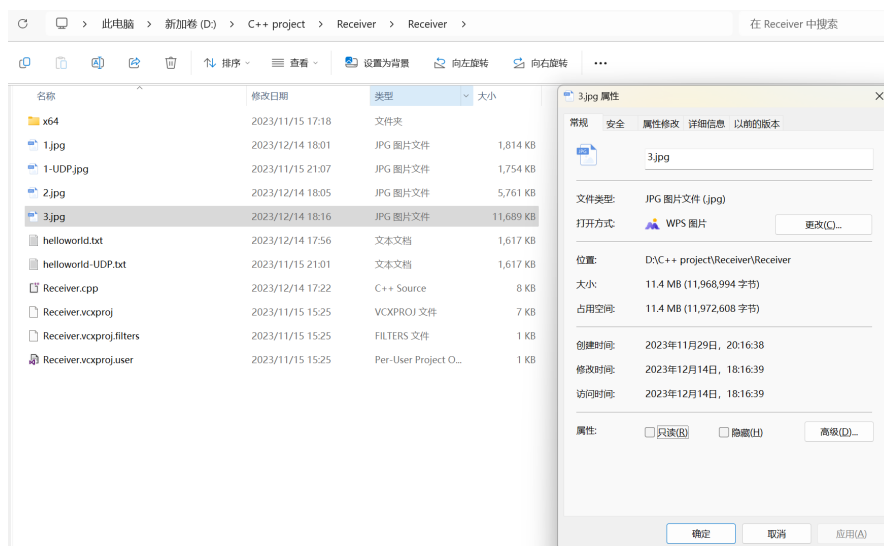


图 8: 接收端目录

另外我们发现这四次传输的吞吐率，相较于上一次采用累积确认传输的吞吐率有明显的提升，提升则是在于重传时无需传输整个窗口，在 3% 的丢包率情况下，选择确认甚至只需要单独重传丢失的那一个包即可。接下来以传输 helloworld.txt 为例，测试在不同丢包率环境中的传输吞吐率，并用 stata 软件绘制出折线统计图。给定窗口大小为 8，超时时间为 1.5s，延时为 5ms 的条件，结果如下（横坐标为丢包率，单位为%，纵坐标为吞吐率，单位为 Mbps）：

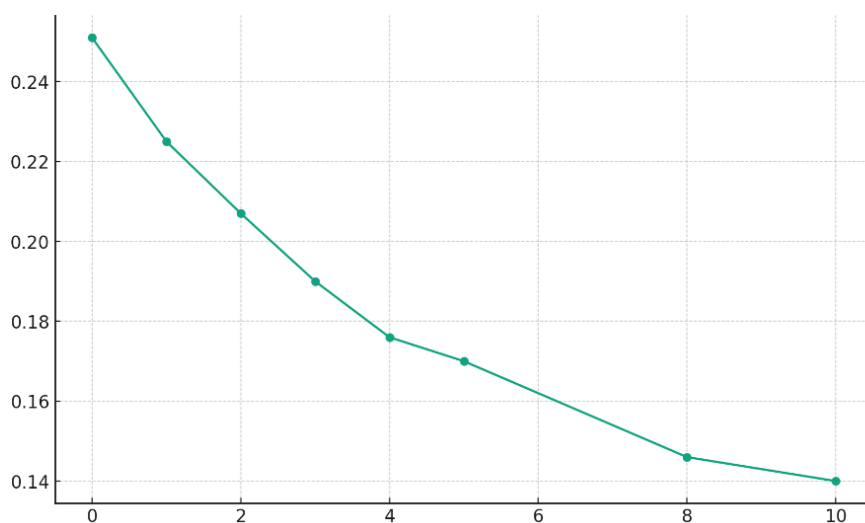


图 9: 折线图

五、 遇到的问题 & 解决方法

Q1: 选择确认需要接收端缓存下失序的数据包，数据包存在哪儿？

A: 在之前的实验中，接受端只需要维护一个期望序列号，如果相等则直接写回文件，写回文件的数据包自然有序。而选择确认意味着需要处理失序的数据包，但最后又需要按序交付给文件，这就需要用数据结构来保存数据包，按理说其实可以开一个发送端窗口大小的数据包数组来缓存数据，Expected 增加时则把此前的数据包写回文件，但代码实现起来相对复杂。实验中直

接开辟了一个大的数据来存所有收到的数据包，传输完毕后最后一次性写回文件即可，缺点是空间开销较大。

Q2: 接收端怎么告诉发送端哪些数据包已经缓存?

A: 本次实验实现选择确认最核心的即是双方的 `flag_cache` 数组，接收端在“填补”好缺失的数据包后，需要依据该数组来增加 `ExpectedNum`，从而更新 `acknum` 并回复报文。在缓存失序数据包的时候，这里的“缓存”其实跟直接存下来是同样的操作，都是直接放回 `Buffer` 数组中，只不过需要用 `sack` 报文单独只是哪个包缓存了，这个 `sack` 和 `sacknum` 或许可以称作“失序确认”和“失序确认号”，本质上和累积确认号是一个思路，都是通知发送端收到了某一个包，只不过一个是失序，一个是按序。发送端进而可以捕捉到这类型的数据包，并设置 `flag` 标志位，重传时则利用标志位跳过失序报文。

六、 思考与总结

三次基于 UDP 的可靠传输实验全部结束。通过三次实验，我对网络协议有了一个更深入和更全面的理解和认知。所谓协议，个人把它理解为一种“约定”，一种“标准”，即接收和发送双方共同遵守的规则。协议设计没有问题的情况下，双方都按照规则做事，则数据能够成功传输。但如果一方不遵守约定，传输也会出现错误。比如在实验中，双方实现约定好建立连接的时候，发送方把文件名放入空闲的数据缓冲区。那么接下来则是，发送方按照约定把文件名放进去，接收方按照约定从收到数据包的缓冲区中读取。如果其中有一方不按规则操作，那么最终文件名无法正确传输到接收方手中，这其实就是一种协议。网络这种东西，本质还是一个分布式的系统，没有一个中心化的机构来进行管理，而网络协议则定义了网络中的设备如何相互通信和交换数据，维系着这个互联网的运转。