南开大学
Nankai University

南　　开　　大　　学

计 算 机 学 院

计算机网络实验报告

基于 UDP 服务设计可靠传输协议 Part2

2011763 黄天昊

年级：2020 级

专业：计算机科学与技术

指导教师：徐敬东、张建忠

2022 年 11 月 25 日

摘要

在基于 UDP 实现可靠数据传输协议 Part2 的设计中，主要是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，完成了回退 N 帧 (GBN) 协议的实现，此外，还实现了**选择重传 (SR) 协议**。

关键字：UDP 可靠数据传输; 滑动窗口; 回退 N 帧协议; 选择重传协议

目录

一、 滑动窗口协议简述

滑动窗口协议是基于停止等待协议的优化版本。

停等协议的性能因为需要等待 ack 之后才能发送下一个帧，在传送的很长时间内信道一直在等待状态。滑动窗口则利用缓冲思想，允许连续发送 (未收到 ack 之前) 多个帧，以加强信道利用

滑动窗口的概念：其实就是缓冲帧的一个容器，将处理好的帧发送到缓冲到窗口，可以发送时就可以直接发送，借此优化性能。一个帧对应一个窗口。

基于滑动窗口协议，我们可以设计出回退 N 帧协议，进而还可以设计出选择重传协议。

二、 回退 N 帧 (GBN) 协议

GBN 是滑动窗口中的一种，其中发送窗口 > 1 , 接收窗口 $=1$ 因发送错误后需要退回到最后正确连续帧位置开始重发，故而得名。

控制方法：

- 发送端：在将发送窗口内的数据连续发送
- 接收端：收到一个之后向接收端发送累计确认的 ack
- 发送端：收到 ack 后窗口后移发送后面的数据

累计确认：累计确认允许接收端一段时间内发送一次 ack 而不是每一个帧都需要发送 ack。该确认方式确认代表其前面的帧都以正确接收到。

差错控制：发送帧丢失、ack 丢失、ack 迟到等处理方法基本和停等协议相同，不同的是采用累计确认恢复的方式，当前面的帧出错之后后面帧无论是否发送成功都要重传

优点：信道利用率高（利用窗口有增加发送端占用，并且减少 ack 回复次数）

缺点：累计确认使得该方法只接收正确顺序的帧，而不接受乱序的帧，错误重传浪费严重

下图展示了 GBN 协议可能出现的实际传输情况：

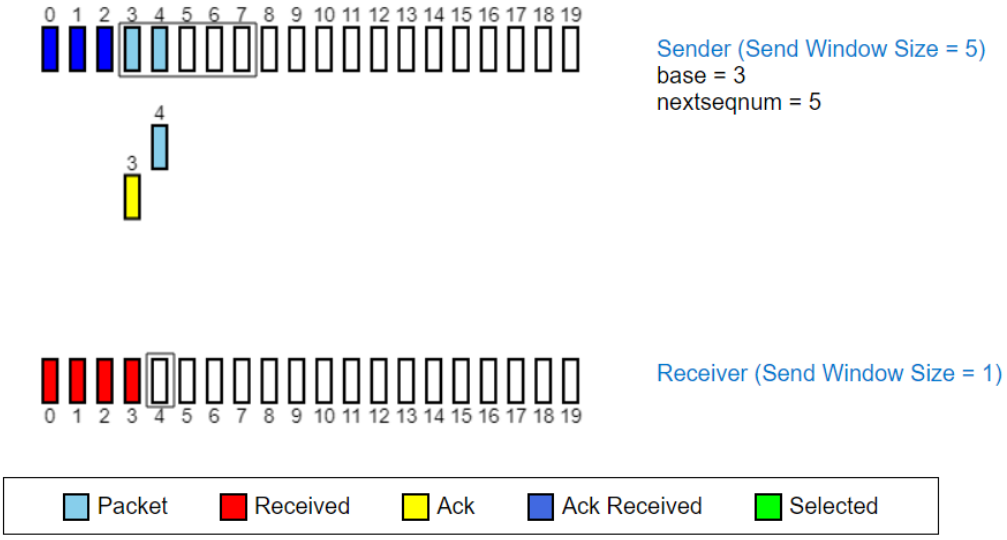

图 1: GBN 协议可能的实际传输情况

由于我在 GBN 的基础上又实现了选择重传，因此会在下一小节具体讲解代码实现。

三、 选择重传 (SR) 协议

(一) 选择重传协议简述

SR 协议可以说是 GBN 的 plus 版本，在 GBN 的基础上改回每一个帧都要确认的机制，解决了累计确认只接收顺序帧的弊端只需要重发错误帧。其中发送窗口 > 1 , 接收窗口 > 1 , 接收窗口 $>$ 发送窗口 (建议接收窗口 $=$ 发送窗口接收窗口少了溢出多了浪费)。

控制方法:

- 发送端：将窗口内的数据连续发送
- 接收端：收到一个帧就将该帧缓存到窗口中并回复一个 ack
- 接收端：接收到顺序帧后将数据提交给上层并接收窗口后移（若接收到的帧不是连续的顺序帧时接收窗口不移动）
- 发送端：接收到顺序帧的 ack 后发送窗口后移（同理发送窗口接收到的 ack 不连续也不移动）

差错控制：发送帧丢失、ack 丢失、ack 迟到三类处理方式仍然和停等协议相同，不同的是 SR 向上层提交的是多个连续帧，停等只提交一个帧（不连续的帧要等接收或重传完成后才会提交）

下图展示了 SR 协议可能出现的实际传输情况:

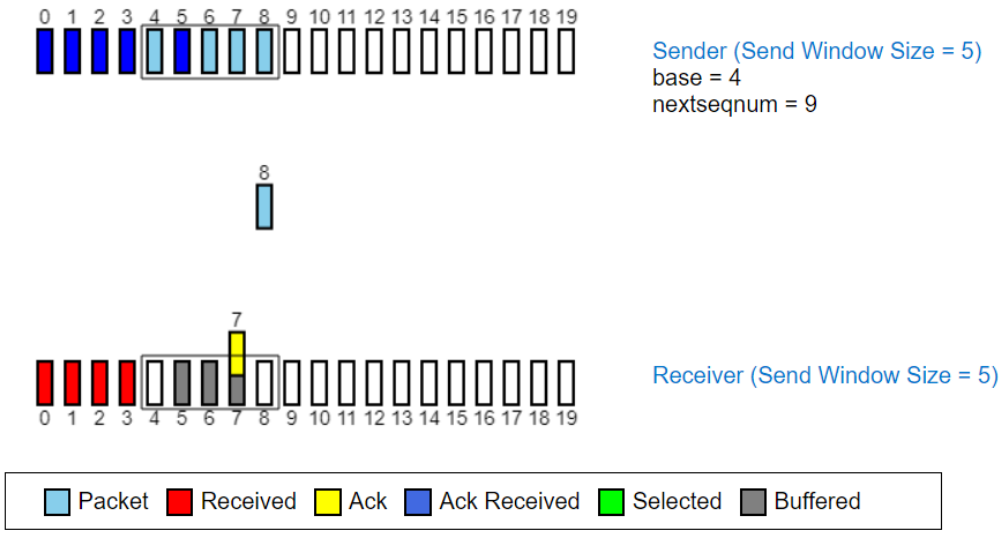

图 2: SR 协议可能的实际传输情况

(二) 代码实现

选择重传的具体代码实现中，发送端的实现较为复杂。

首先是基于滑动窗口发送数据包的机制的实现:

滑动窗口机制相关全局变量

```
#define DATA_AREA_SIZE 1024
#define BUFFER_SIZE sizeof(Packet)  // 缓冲区大小
#define WINDOW_SIZE 16  // 滑动窗口大小

unsigned int packetNum;  // 发送数据包的数量
unsigned int sendBase;  // 窗口基序号，指向已发送还未被确认的最小分组序号
unsigned int nextSeqNum;  // 指向下一个可用但还未发送的分组序号

char** selectiveRepeatBuffer;  // 选择重传缓冲区
```

基于滑动窗口发送数据包代码实现

```
while (sendBase < packetNum) {
        while (nextSeqNum < sendBase + WINDOW_SIZE && nextSeqNum < packetNum)
            {  // 只要下一个要发送的分组序号还在窗口内，就继续发
            if (nextSeqNum == packetNum - 1) {  // 如果是最后一个包
                    sendPkt->setTAIL();
                    sendPkt->fillData(nextSeqNum, fileSize - nextSeqNum *
                        DATA_AREA_SIZE, fileBuffer + nextSeqNum *
                        DATA_AREA_SIZE);
                    sendPkt->checksum = checksum((uint32_t*)sendPkt);
            }
            else {  // 正常的1024Bytes数据包
                    sendPkt->fillData(nextSeqNum, DATA_AREA_SIZE,
                        fileBuffer + nextSeqNum * DATA_AREA_SIZE);
                    sendPkt->checksum = checksum((uint32_t*)sendPkt);
            }
            memcpy(selectiveRepeatBuffer[nextSeqNum - sendBase], sendPkt,
                sizeof(Packet));  // 存入缓冲区
            sendPacket(sendPkt);
            timerID[nextSeqNum - sendBase] = SetTimer(NULL, 0, TIME_OUT,
                (TIMERPROC)resendPacket);  // 启动计时器，这里nIDEvent=0
                ，是因为窗口句柄设为NULL的情况下这里填什么都无所谓，反正
                它会重新分配，返回值才是真正的nIDEvent
            nextSeqNum++;
            Sleep(10);  // 睡眠10ms，包与包之间有一定的时间间隔
        }

        // 如果当前发送窗口已经用光了，就进入接收ACK阶段
        err = recvfrom(socketClient, (char*)recvPkt, BUFFER_SIZE, 0, (
            SOCKADDR*)&(socketAddr), &socketAddrLen);
        if (err > 0) {
                printReceivePacketMessage(recvPkt);
                ackHandler(recvPkt->ack);  // 处理ack
        }

        // 应该不能把监听消息的写在这里
```

```
        MSG msg;
        // GetMessage(&msg, NULL, 0, 0); // 阻塞式，一开始用的这个，被坑了，他不得到消息不会结束，所以一定会触发重传
        while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) { // 以查看的方式从系统中获取消息，可以不将消息从系统中移除，是非阻塞函数；当系统无消息时，返回FALSE，继续执行后续代码。
            if (msg.message == WM_TIMER) { // 定时器消息
                DispatchMessage(&msg);
            }
        }
}
```

这里我使用了和教材中一样的两个指针，sendBase 和 nextSeqNum，分别指向已发送还未确认的最小分组序号和下一个可用还未发送的分组序号。只要当前发送端的 sendBase 还不等于要发送的数据包数量，发送端就会继续处于循环发送数据包的状态。需要注意的是，每次发送一个数据包，需要将其数据进行缓存，以备超时重传，并且需要设置计时器来判断是否需要超时重传。

其中，如果当前发送窗口内的分组已全部发送，那么会进入接收 ACK 的阶段，如果收到了 ACK 数据包，将会对其进行处理。

处理发送端发来的 ACK 数据包

```
void ackHandler(unsigned int ack) { // 处理来自接收端的ACK
        // cout << endl << "ack " << ack << endl << endl;
        if (ack >= sendBase && ack < sendBase + WINDOW_SIZE) { // 如果ack分组序号在窗口内
            // cout << "KillTimer " << timerID[ack - sendBase] << endl << endl;
            KillTimer(NULL, timerID[ack - sendBase]);
            timerID[ack - sendBase] = 0; // timerID置零

            if (ack == sendBase) { // 如果恰好=sendBase，那么sendBase移动到具有最小序号的未确认分组处
                for (int i = 0; i < WINDOW_SIZE; i++) {
                    if (timerID[i]) break; // 遇到一个有计时器的停下来(如果当前timerID数组元素不为0，说明有计时器在使用)
                    sendBase++; // sendBase后移
                }
                int offset = sendBase - ack;
                for (int i = 0; i < WINDOW_SIZE - offset; i++) {
                    timerID[i] = timerID[i + offset]; // timerID也得平移，不然对不上了
                    timerID[i + offset] = 0;
                    memcpy(selectiveRepeatBuffer[i], selectiveRepeatBuffer[i + offset], sizeof(Packet)); // 缓冲区也要平移
                }
                for (int i = WINDOW_SIZE - offset; i < WINDOW_SIZE; i++) {
```

```
                                timerID[i] = 0;  // 这里一开始没想到，可能出现的bug是这样的，窗口内除了recvBase都收到了，再收到的recvBase的ack，这时offset就等于窗口大小，那么上一个循环将不会执行，那些值也不会置零了
                            }
                        }
                    }
                }
```

在处理 ACK 时，对于相对于当前发送窗口不同区域分组序号的数据包，发送端会采取不同的处理方法。具体而言：

- 如果 ACK 序号在 [sendBase, sendBase + N -1] 内，则发送方将该分组标记为已接收。特别的，如果该分组序号等于 sendBase，则 sendBase 向前移动到具有最小序号的未确认分组处。而且将相应确认过的计时器关闭。
- 如果窗口移动了并且有序号落到窗口内的未发送分组，则发送这些分组。

其次是超时重传机制的实现。

不同于 GBN 协议，SR 协议的发送端在传输数据包时，需要为每个数据包设置一个独立的计数器，这里我采用的方法是每次发送数据包时，都调用 SetTimer 函数来建立一个计时器，该函数会自动计时，到时会发送一个 WM_TIMER 消息到消息队列，通过 PeekMessage 函数去循环查看消息队列有无超时消息，如果有，就会将该消息派发给对应的处理函数，重传数据包

这部分源代码如下：

超时重传机制相关全局变量

```
#define TIME_OUT 0.5 * CLOCKS_PER_SEC  // 超时重传，这里暂时设为2s
/*
* 计时器SetTimer的返回值，即Timer的ID。
* 当窗口句柄为NULL时，系统会随机分配ID，因此如果该数组某个元素不为0，
* 就代表当前对应的分组有Timer在计时
* 所以这个数组有两个重要的作用：
* (1)数组元素的值对应计时器的ID
* (2)标识一个分组有没有对应的计时器，如果有(即数组元素不为0)，说明该分组未被ack
*/
int timerID[WINDOW_SIZE];
```

重传数据包函数

```
void resendPacket(HWND hwnd, UINT uMsg, UINT idEvent, DWORD dwTime) {  // 重传函数
    // cout << endl << "resend" << " Timer ID " << idEvent << endl << endl;
    unsigned int seq = 0;
    for (int i = 0; i < WINDOW_SIZE; i++) {  // 找到是哪个Timer超时了
        if (timerID[i] == idEvent && timerID[i] != 0) {
```

```
                    seq = i + sendBase;
                    break;
                }
            }
            cout << "第 " << seq << " 号数据包对应的计时器超时，重新发送" << endl;

            Packet* resendPkt = new Packet;
            memcpy(resendPkt, selectiveRepeatBuffer[seq - sendBase], sizeof(Packet));  // 从缓冲区直接取出来
            sendPacket(resendPkt);
            printSendPacketMessage(resendPkt);
            cout << endl;
            // timerID[seq - sendBase] = SetTimer(NULL, 0, TIME_OUT, (TIMERPROC)resendPacket);  // 重设Timer
}
```

值得一提的是，原本我想要实现的是单开一个线程来循环监听消息队列，如果有超时消息，将其发送给 resendPacket 重传函数，但是由于 C++ 的相关机制，如果单开线程，子线程和主线程的消息队列并不相通，需要一定的机制实现（如 PostMessage 函数，但是用这个函数去实现并不比我在代码中实现的方法简单，且性能更差），可能会涉及到底层的消息队列的机制问题，因此此处我只是在 sendFile 函数中每次循环监听一次消息队列，这里存在改进空间，在此特别说明。

相对而言，接收端的实现就简单一些，只需要维护一个 recvBase 变量，指向期待收到但尚未收到的最小分组序号。

接收端累计确认机制相关全局变量

```
#define DATA_AREA_SIZE 1024
#define BUFFER_SIZE sizeof(Packet)  // 缓冲区大小
#define WINDOW_SIZE 16  // 滑动窗口大小

unsigned int packetNum;  // 发送数据包的数量
unsigned int fileSize;  // 文件大小
unsigned int recvSize;  // 累积收到的文件位置
unsigned int recvBase;  // 期待收到但还未收到的最小分组序号
bool isCached[WINDOW_SIZE];  // 标识窗口内的分组有没有被缓存
char* fileBuffer;  // 读入文件缓冲区
char** receiveBuffer;  // 接收缓冲区
```

基于滑动窗口接收数据包代码实现

```
while (recvBase < packetNum) {  // 还没收完
        err = recvfrom(socketServer, (char*)recvPkt, BUFFER_SIZE, 0, (SOCKADDR*)&(socketAddr), &socketAddrLen);
        if (err > 0) {
                if (isCorrupt(recvPkt)) {  // 检测出数据包损坏
                        printTime();
                        cout << "收到一个损坏的数据包，不做任何处理" << endl << endl;
```

```
7                   }
8                   printReceivePacketMessage(recvPkt);
9                   printTime();
10                  cout << "收到第 " << recvPkt->seq << " 号数据包" << endl;
11                  if (randomNumber(randomEngine) >= 0.02) {  // 自主进行丢包测
                        试
12                      unsigned int seq = recvPkt->seq;
13                      if (seq >= recvBase && seq < recvBase + WINDOW_SIZE)
                            {  // 窗口内的分组被接收
14                          sendACK(recvPkt->seq);  // 必须发送ACK
15                          if (!isCached[seq - recvBase]) {  // 没有被缓
                                存，那就缓存下来
16                              cout << "首次收到该数据包，将其缓存
                                    ";
17                              memcpy(receiveBuffer[seq - recvBase],
                                     recvPkt->data, DATA_AREA_SIZE);
18                              isCached[seq - recvBase] = 1;
19                          }
20                          else {
21                              cout << "已经收到过该数据包！";
22                          }
23
24                          if (seq == recvBase) {  // 该分组序号等于基序
                                号
25                              for (int i = 0; i < WINDOW_SIZE; i++)
                                    {  // 窗口移动
26                                  if (!isCached[i]) break;  //
                                        遇到没有缓存的，停下来
27                                  recvBase++;
28                              }
29                              int offset = recvBase - seq;
30                              cout << "该数据包序号等于目前接收基序
                                    号，窗口移动 " << offset << " 个
                                    单位 ";
31                              for (int i = 0; i < offset; i++) {
                                    // 将这些分组交付上层
32                                  memcpy(fileBuffer + (seq + i)
                                         * DATA_AREA_SIZE,
                                        receiveBuffer[i], recvPkt
                                        ->len);
33                              }
34                              for (int i = 0; i < WINDOW_SIZE -
                                    offset; i++) {
35                                  isCached[i] = isCached[i +
                                        offset];  // 这里与发送端
                                        类似，都需要平移两个数组
                                        以对齐端口
36                                  isCached[i + offset] = 0;
```

```
37                                                  memcpy(receiveBuffer[i],
                                                        receiveBuffer[i + offset
                                                        ], DATA_AREA_SIZE); //
                                                        缓存区也平移，或许这里可
                                                        以使用取模运算避免大量的
                                                        内存拷贝，提高性能？（时
                                                        间和脑子不够用了，不想
                                                        了）
38                                              }
39                                              for (int i = WINDOW_SIZE - offset; i
                                                    < WINDOW_SIZE; i++) {
40                                                  isCached[i] = 0; // 这里一开
                                                        始没想到，可能出现的bug是
                                                        这样的，窗口内除了
                                                        recvBase都收到了，再收到
                                                        的recvBase的ack，这时
                                                        offset就等于窗口大小，那
                                                        么上一个循环将不会执行，
                                                        那些值也不会置零了
41                                              }
42                                      }
43                                      printWindow();
44                                      cout << endl;
45                              }
46                              else if (seq >= recvBase - WINDOW_SIZE && seq <
                                    recvBase) {
47                                      printWindow();
48                                      cout << "  该数据包分组序列号在[recv_base - N
                                            , recv_base - 1]，发送ACK，但不进行其他操
                                            作" << endl;
49                                      sendACK(recvPkt->seq); // 在这个范围内的分组
                                            必须发送ACK，但不做其他任何操作
50                              }
51                              else {
52                                      printWindow();
53                                      cout << "  分组序号 " << seq << " 不在正确范
                                            围，不做任何操作" << endl;
54                              }
55                      }
56                      else {
57                              cout << "主动丢包" << endl;
58                      }
59              }
60      }
```

在上述代码中可以看到，对于相对于当前接收窗口不同区域分组序号的数据包，接收端会采取不同的处理方法。具体而言：

- 如果序号在 [recvBase, recvBase + N -1] 内的分组被正确接收，接收方发送一个 ACK 给发

送方。如果该分组没有被收到过，则缓存该分组。特别的，如果该分组序号等于 recvBase，则该分组以及以前缓存的序号连续的（起始于 recvBase 的）分组交付给上层。

- 如果序号在 [recvBase - N, recvBase -1] 内的分组被正确接收，则接收端必须产生一个 ACK，但是不做其他任何操作。
- 其他情况，忽略该分组。

四、 可靠 UDP 传输流程展示

本节进行实现了选择重传协议的基于 UDP 的可靠传输协议实际收发流程的展示。
首先是三次握手的 log 信息：

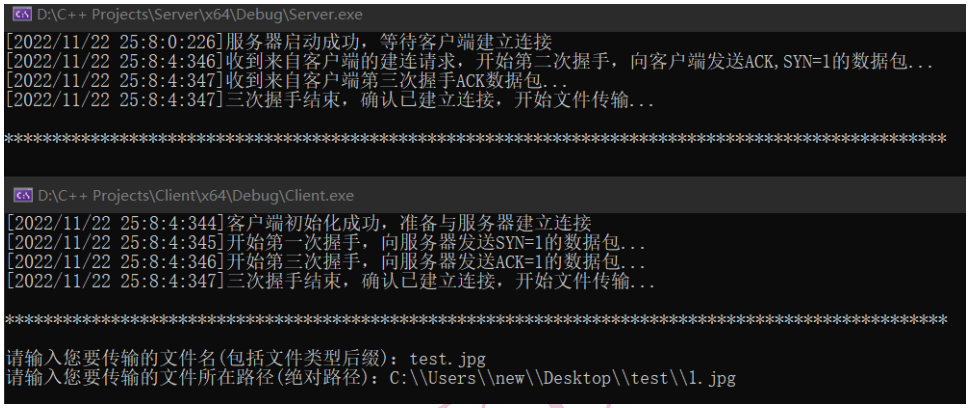

图 3: 三次握手建立连接

接下来是文件传输的过程：

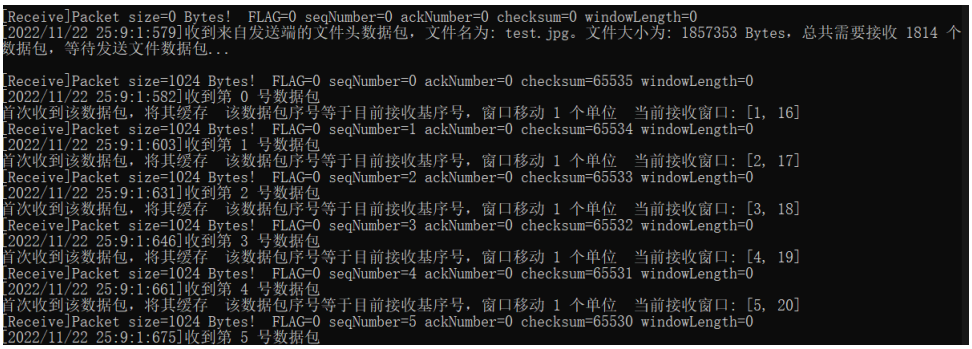

图 4: 服务器接收数据包部分过程 log 信息打印

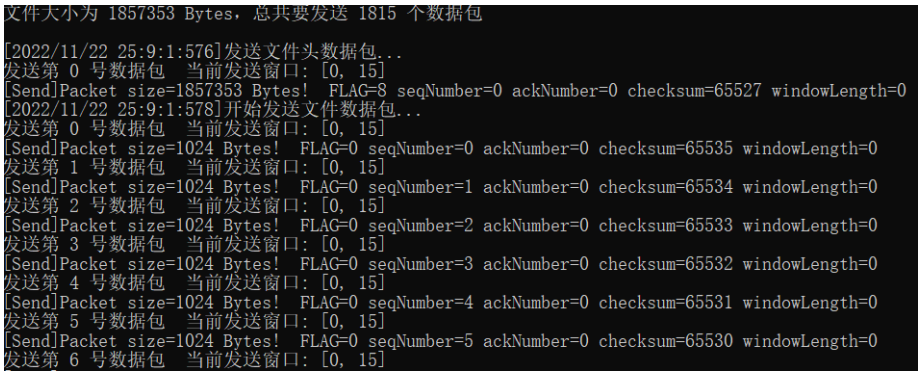

图 5: 客户端发送数据包部分过程 log 信息打印

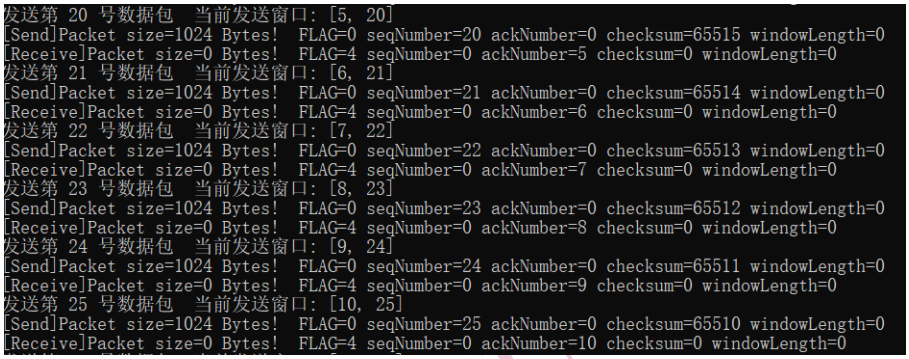

图 6: 客户端发送数据包部分过程 log 信息打印

如果某个数据包出现了丢失，则两方相应的动作如下:

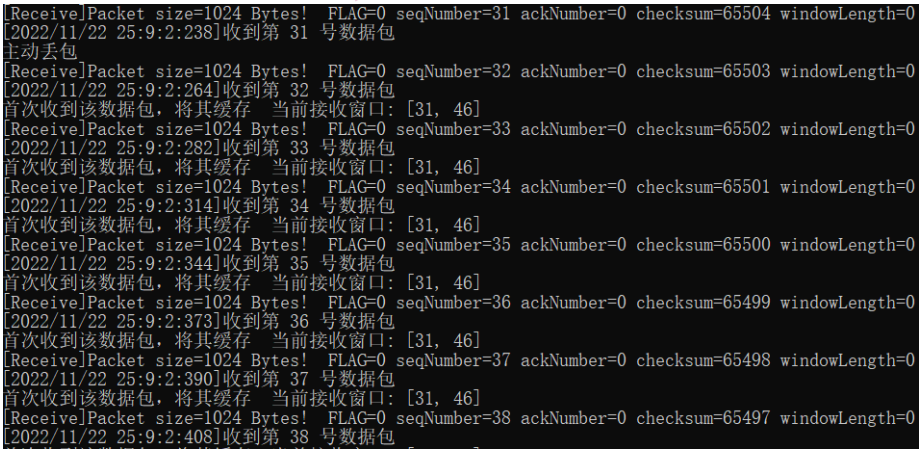

图 7: 某个数据包丢失后接收端动作

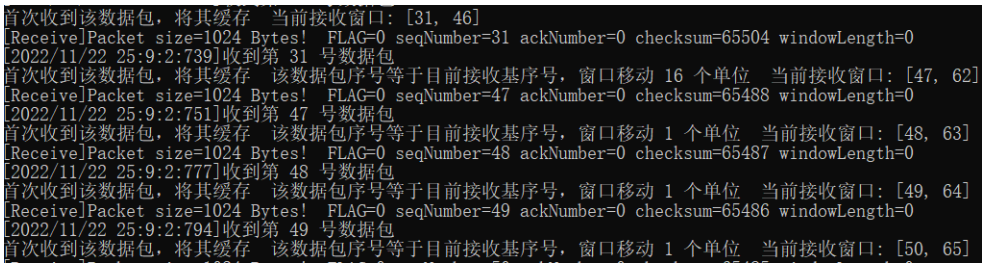

图 8: 某个数据包丢失后接收端动作

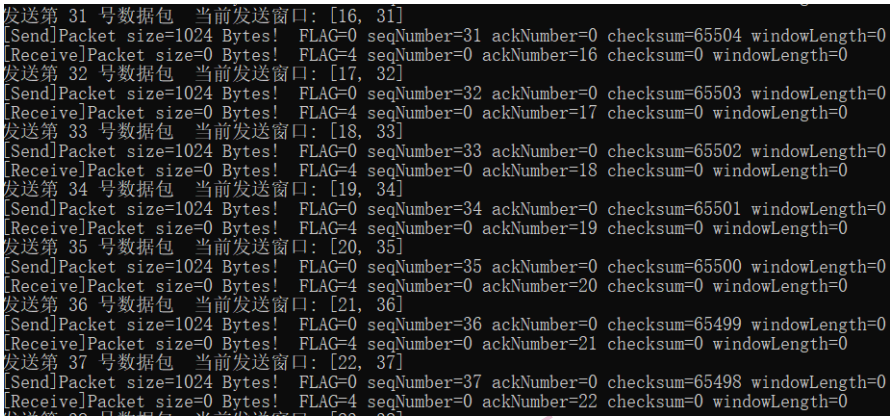

图 9: 某个数据包丢失后发送端动作

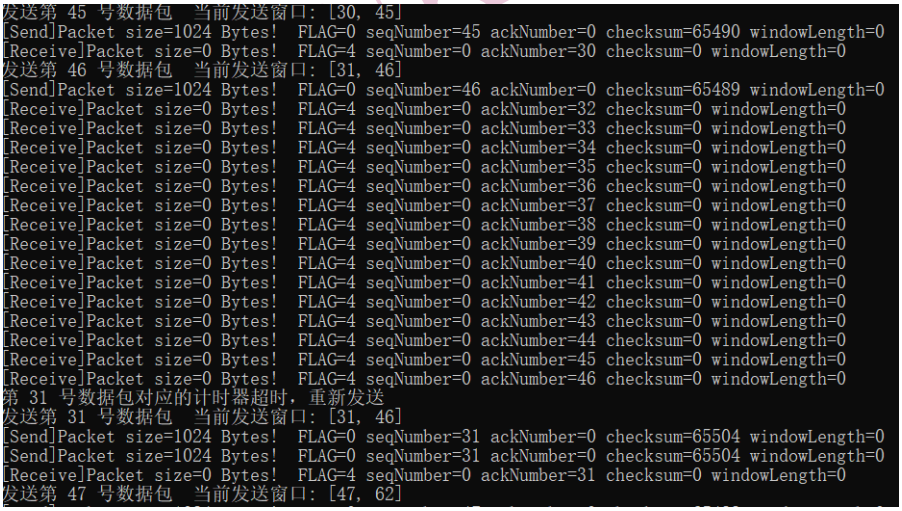

图 10: 某个数据包丢失后发送端动作

文件收发完成后，四次握手断开连接

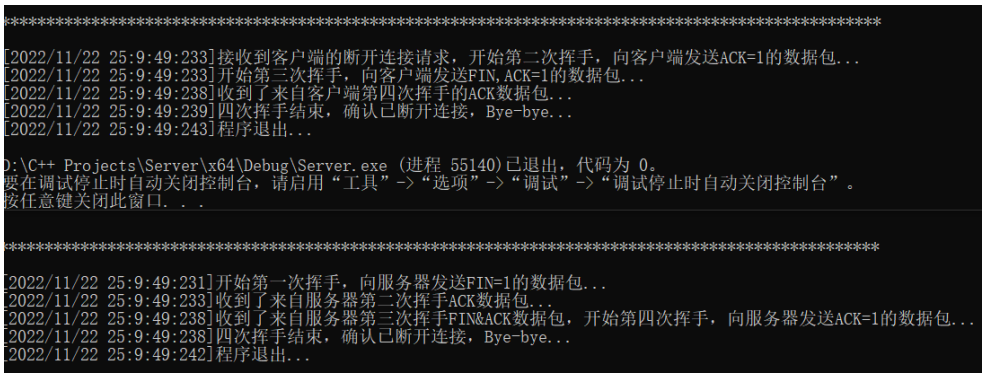

图 11: 四次挥手断开连接

至此，整个过程结束。

五、 总结

在本次实验之中，深入地了解了滑动窗口协议、后退 N 帧协议以及选择重传协议，并且对于选择重传协议进行了编程实现。在编写代码的过程中，对于线程以及消息队列的相关知识进行了深入了解，并且进一步理解和体会了计算机网络中传输协议的设计与实现。

个人 GitHub 仓库链接：https://github.com/Skyyyy0920/Computer-Network

参考文献

1、https://www.jianshu.com/p/1d6df6c61c3b
2、03-计算机网络-第三章-2022.pdf
3、上机作业 3 讲解-2022.pdf