

计算机网络第一——实验3-2

- 实验名称：基于UDP服务设计可靠传输协议并编程实现（实验3-2）
- 专业：物联网工程
- 姓名：秦泽斌
- 学号：2212005

一、实验要求

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、实验原理

1.滑动窗口

滑动窗口机制 是一种用于流量控制和可靠传输的技术，广泛应用于传输层协议（如TCP）中。它允许发送方在等待确认的同时，连续发送多个数据包，而不必等待每个数据包的确认。这种机制提高了数据传输的效率，减少了等待时间，从而增加了网络的吞吐量。

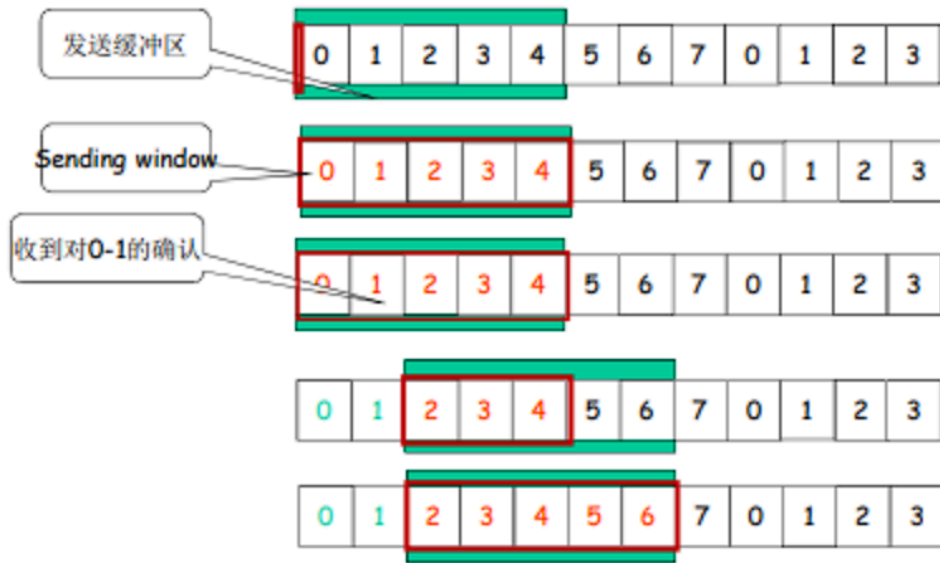
1. 发送缓冲区：

- 形式：内存中的一个区域，**落入缓冲区的分组可以发送**
- 功能：用于存放已发送，但是没有得到确认的分组
- 必要性：需要重发时可用
- 缓冲区大小：
 - 停等协议：大小为1
 - **流水线协议：大小大于1**，设置合理的值，不能很大，链路利用率不能够超100%
- **发送缓冲区的分组：**
 - **未发送的（缓冲区中，但不在滑动窗口内部的）**：落入发送缓冲区的分组，可以连续发送出去；
 - **已经发送出去的、等待对方确认的分组（滑动窗口内部的分组）**：发送缓冲区的分组只有得到确认才能删除
- **发送端的滑动窗口**：发送方维护一个窗口，这个窗口可以根据网络条件进行调整，**以实现流量控制**。其本质是**发送缓冲区内容的一个子集，那些已发送但是未经确认分组的序号构成的空间**。本次实验实现过程也是遵循这个概念，即我的发送缓冲区是有固定大小的，而滑动窗口大小是会变化的。
 - 发送窗口的最大值 \leq 发送缓冲区的值
 - 一开始：没有发送任何一个分组。即前沿=后沿，之间为发送窗口的尺寸=0。
 - **前沿移动**：每发送一个分组，前沿前移一个单位。发送窗口前沿移动的极限：不能够超过发送缓冲区。
 - **后沿移动**：
 - 条件：收到老分组的确认
 - 结果：发送缓冲区罩住新的分组，来了分组可以发送
 - 移动的极限：不能够超过前沿

3.5 滑动窗口(slide window)协议

□ 滑动窗口技术

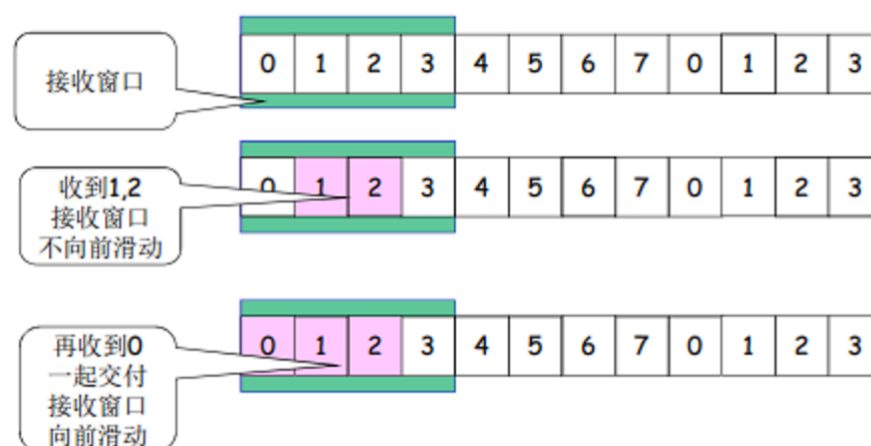
○ 发送窗口 (sending window)



2. 接收缓冲区:

- 接收端的滑动窗口: 对于接收者来说, 接收缓冲区就等于接收端的滑动窗口。即二者大小一致。
- 接收窗口用于控制哪些分组可以接收: 只有收到的分组序号落入接收窗口内才允许接收; 若序号在接收窗口之外, 则丢弃。
- 接收窗口尺寸 $W_r=1$, 则只能顺序接收 (停等机制与GBN); 接收窗口尺寸 $W_r>1$, 则可以乱序接收 (SR)。但提交给上层的分组, 要按序。
- 接收端滑动窗口的滑动和发送确认:
 - 滑动:
 - 低序号的分组到来, 接收窗口移动; (停等机制与GBN)
 - 高序号分组乱序到, 缓存但不交付 (因为要实现RDT, 不允许失序。即RDT的可靠要保证交付时的有序!), 不滑动 (SR)。
 - 发送确认:
 - 接收窗口尺寸=1; 发送连续收到的最大的分组确认 (累计确认)。也是我们本次实验要实现的确认方式。
 - 接收窗口尺寸>1; 收到分组, 发送那个分组的确认 (非累计确认)

滑动窗口(slide window)协议-接收窗口



2. GBN累计确认

GBN (Go-Back-N, 回退N) 是一种基于窗口的自动重传请求 (ARQ) 协议，常用于可靠数据传输。它通过累计确认 (Cumulative Acknowledgment) 机制来确保数据的可靠传输。在GBN协议中，发送方可以一次发送多个数据包，但接收方只发送一个确认帧，用来确认所有已接收到的数据包。累计确认机制是GBN的核心特点之一。

2.1 基本概念

1. 滑动窗口：

- 发送方和接收方都维护一个滑动窗口。发送方的窗口大小是N，意味着发送方可以在未收到确认的情况下连续发送N个数据包。
- 接收方的窗口大小是1，因为接收方按顺序接收数据包，只确认当前正确接收到的数据包。

2. 序列号：

- 每个数据包都有一个唯一的序列号。发送方按照顺序为每个数据包分配序列号。
- 发送方在窗口内可以同时发送多个数据包，但不要求按顺序确认它们。

3. 累计确认：

- 接收方发送一个确认帧 (ACK)，该确认帧包含一个序列号，表示接收方成功接收到所有序列号小于该确认号的数据包。
- 比如，接收方如果成功接收到序列号为3的数据包，那么接收方会返回一个确认帧，表示它已成功接收到所有小于或等于3的序列号的数据包。

2.2 工作流程

1. 发送方操作：

- 发送方维护一个窗口，窗口大小为N。当窗口中的数据包被发送出去时，发送方会启动一个定时器并等待确认。

- 如果超时，发送方会重新发送所有未确认的数据包，即使部分数据包已经被接收方正确接收。
2. 接收方操作：
- 接收方只接收按序到达的数据包。如果接收方发现一个数据包的序列号超出了期望的范围（比如接收到序列号为3的包，但它期望的是2），它会丢弃该数据包，并继续等待正确的包。
 - 一旦接收到按序排列的所有数据包，接收方发送一个累计确认帧，确认所有小于当前序列号的数据包已被接收。
3. 确认机制：
- 发送方会根据接收方的累计确认帧来确定已成功接收的最大数据包序列号，并将其从发送窗口中移除。
 - 例如，如果接收方返回确认帧ACK 3，表示它已经成功接收到所有序列号为0, 1, 2和3的数据包。发送方就可以将窗口向前滑动，并继续发送新的数据包。
4. 超时重传：
- 如果某个数据包的确认未在超时时间内收到，发送方会重新发送该数据包以及所有后续的数据包。这就是"回退N"的含义，发送方回退到未确认的最早数据包并重新发送。

2.3 GBN协议的优缺点

优点：

- 1. **实现简单**：GBN协议通过累计确认和回退N机制使得实现相对简单。
- 2. **高效的带宽利用**：由于发送方在等待确认时可以发送多个数据包，较高效地利用带宽。

缺点：

- 1. **带宽浪费**：如果出现丢包或超时，发送方会重新发送一个数据包后续的所有数据包。这可能导致带宽浪费，特别是当丢包发生时。
- 2. **性能问题**：当网络质量较差时，GBN的性能较差，特别是在大量数据包丢失时。

GBN与其他协议对比

特性	GBN协议	SR (Selective Repeat) 协议
确认机制	累计确认 (Cumulative ACK)	独立确认 (Individual ACK)
重传策略	回退N (回传所有未确认数据)	只重传丢失的数据包
窗口大小	发送窗口为N，接收窗口为1	发送窗口和接收窗口都为N
带宽效率	较低 (大量丢包时浪费带宽)	更高 (只重传丢失的数据包)

三、实验内容

1. 发送端滑动窗口设计

由于本实验的Part1中函数封装较好，本次Part2我们着重设计了 `send_packet_GBN()` 和 `send_file_GBN()` 这两个函数来实现**滑动窗口机制**。

首先是 `send_packet_GBN` 函数，该函数较为简单，目的就是传入的数据包进行发送，方便发送文件时调用。

```
1 void send_packet_GBN(my_udp& Packet, SOCKET& SendSocket, sockaddr_in&
  RecvAddr) {
2     int iResult;
3     int RecvAddrSize = sizeof(RecvAddr);
4     char* SendBuf = new char[UDP_LEN];
5
6     memcpy(SendBuf, &Packet, UDP_LEN);
7     iResult = sendto(SendSocket, SendBuf, UDP_LEN, 0, (SOCKADDR*)&RecvAddr,
  sizeof(RecvAddr));
8     if (iResult == SOCKET_ERROR) {
9         setConsoleColor(12);
10        cout << "sendto failed with error: " << WSAGetLastError() << endl;
11    }
12
13    delete[] SendBuf;
14 }
```

接下来的设计重点，也是发送端滑动窗口机制的核心，`send_file_GBN` 函数。该函数首先会读取文件信息并将文件分片，接下来发送第一个数据包（包含文件名）来启动发送，接下来会启动计时器，并创建接收线程，启动while循环阻塞。窗口大小设置为32。

- 关于 `while()` 循环，首先其内部还有一个 `while()` 循环，内循环用于检查现在是否符合发送数据包的条件（发送窗口内有未被发送的数据包），若符合条件则将数据包打包并调用 `send_packet_GBN` 函数发送。接下来是超时重传逻辑，当等待时间超时后，发送端会将缓存区队列中的所有数据包重新发送。
- 关于缓存区队列，本实验建立了一个 `queue` 队列用于维护已被发送窗口发送但尚未被ACK确认的数据包。
- 关于计时器，计时器的设置并启动实在进入 `while()` 循环之前，至于重置计时器，分别在已经完成一次超时重传的操作后和成功发送了一次数据包后进行。

```
1 // GBN的发送文件函数
2 void send_file_GBN(string filename, SOCKET& SendSocket, sockaddr_in&
  RecvAddr) {
3     GBN_init();
4     int RecvAddrSize = sizeof(RecvAddr);
5
6     // 读取文件数据
7     ifstream fin(filename.c_str(), ifstream::binary);
8     fin.seekg(0, std::ifstream::end);
9     long size = fin.tellg();
10    file_size = size;
11    fin.seekg(0);
12
13    char* binary_file_buf = new char[size];
14    setConsoleColor(1);
15    cout << " #####文件大小: " << size << " bytes" << endl;
16    fin.read(&binary_file_buf[0], size);
17    fin.close();
18
19    HEADER udp_header(filename.length(), 0, START, stream_seq_order, 0);
```

```

20     my_udp udp_packets(udp_header, filename.c_str());
21     uint16_t check = checksum((uint16_t*)&udp_packets, UDP_LEN); // 计算校验和
22     udp_packets.udp_header.cksum = check;
23
24     int packet_num = size / DEFAULT_BUFLen + 1;
25     setConsoleColor(1);
26     cout << " #####文件名校验和: " << check << endl;
27     cout << " #####送数据包的数量: " << packet_num << endl;
28     cout << " #####滑动窗口大小: " << N << endl;
29
30     // 正常发送第一个文件名数据包
31     send_packet(udp_packets, SendSocket, RecvAddr, packet_num);
32     //创建计时器
33     clock_t start = clock();
34     // 创建接收 ACK 的线程
35     thread ack_thread(receive_ACK_thread, ref(SendSocket), ref(RecvAddr),
36         ref(base), ref(ACK_index), ref(message_queue), ref(packet_num),ref(start));
37
38     // 主线程发送数据包
39     while (ACK_index < packet_num) {
40         lock_guard<mutex> lock(mtx); // 确保共享变量的安全访问
41
42         // 检查是否可以发送新的数据包
43         while (next_seqnum < base + N && next_seqnum <= packet_num) {
44             uint16_t remainder = next_seqnum % DEFAULT_SEQNUM;
45             if (next_seqnum == packet_num) { // 最后一个包
46                 udp_header.set_value(size - (next_seqnum - 1) *
47                     DEFAULT_BUFLen, 0, OVER, stream_seq_order, remainder);
48                 udp_packets.set_value(udp_header, binary_file_buf +
49                     (next_seqnum - 1) * DEFAULT_BUFLen, size - (next_seqnum - 1) *
50                     DEFAULT_BUFLen);
51             }
52             else { // 正常包
53                 udp_header.set_value(DEFAULT_BUFLen, 0, 0, stream_seq_order,
54                     remainder);
55                 udp_packets.set_value(udp_header, binary_file_buf +
56                     (next_seqnum - 1) * DEFAULT_BUFLen, DEFAULT_BUFLen);
57             }
58
59             udp_packets.udp_header.cksum = checksum((uint16_t*)&udp_packets,
60                 UDP_LEN);
61             send_packet_GBN(udp_packets, SendSocket, RecvAddr);
62             setConsoleColor(7);
63             cout << "[SEND] Packet SEQ=" << remainder << " Checksum = " <<
64                 udp_packets.udp_header.cksum << endl;
65             message_queue.push(udp_packets);
66             next_seqnum++;
67         }
68
69         // 检查超时并重发
70         if (clock() - start > MAX_TIME) {
71             setConsoleColor(12);
72             cout << "[TIMEOUT] 重发未确认数据包." << endl;
73             start = clock();
74
75             queue<my_udp> temp_queue = message_queue; // 创建副本用于重发

```

```

68         while (!temp_queue.empty()) {
69             send_packet_GBN(temp_queue.front(), SendSocket, RecvAddr);
70             setConsoleColor(7);
71             cout << "[RESEND] Packet SEQ=" <<
temp_queue.front().udp_header.SEQ << " resent." << endl;
72             temp_queue.pop();
73         }
74     }
75 }
76
77 // 等待接收线程结束
78 ack_thread.join();
79 setConsoleColor(10);
80 cout << "####文件传输完成!####" << endl << endl;
81 stream_seq_order++;
82 check_stream_seq();
83 delete[] binary_file_buf;
84 }

```

2. 接收发送双线程设计

本次实验采用了多线程的机制，将发送功能和接收ACK功能分离开来并建立双线程。

- **RDT3.0的停等机制中**，不论是发送方还是接收方，不论是握手，发送数据还是挥手，任何阶段出现了数据包丢失的情况进行超时重传都只需要一个while死循环下的非阻塞模式+recv_from即可。这是因为他们每个人在任何时刻都是基于停等机制，即发送完一个东西后，只需要接收到新消息才需要再次发送，而不需要在等待接收的时候一直发送。
- **但是引入了流水线协议后则不尽相同！**
 - **接收方**还是与RDT3.0中相同即只需要等待即可，因为他只需要等待接收自己此时窗口正在等待的数据包，他在等待时候不需要做任何事。换句话说，他所要做的任何事（包括发送ACK）都是在他接受到东西之后，而不是接收到东西的同时。因此对于接收方服务器端，其无需多线程，仍然用一个while即可。
 - **发送方（客户端）**则必须使用多线程的机制！这是因为和接收方不同，流水线协议的要求，强调了发送方不能发完一个数据包后就在那傻等，还需要与此同时接着发送数据包，流水线协议本身容许发送方在未得到对方确认的情况下一次发送多个分组。因此它的接收和发送要在同时进行，只使用一个while就不足够了。必须和Lab1的多人聊天室一样，基于多线程保证能够同时发送和接收消息。因为如果还是那样while死循环下的非阻塞模式+recv_from，然后在while循环了继续发送，可能导致发送过程中while循环没结束，这时候recv_from已经返回-1了，此时不会同时接收，就会出现ACK包丢失的情况！
 - **握手和挥手**：对于发送方和接收方，不论是谁，在握手和挥手阶段本质上都是基于的停等机制，他们在干完一件事后只需要等待对方的回复，回复后再干另一件，因此此处的超时重传之间用while死循环下的非阻塞模式+recv_from即可。
- **由于本次实验需要涉及多线程，而多线程如果需要进行彼此的通信就一定需要全局变量，而全局变量一旦在多线程中出现，反而又涉及竞争的问题！**即不能同时对其进行读写。因此本次实验中设计采用了锁机制来避免竞争。

```

1 //接收函数的线程
2 void receive_ACK_thread(SOCKET& SendSocket, sockaddr_in& RecvAddr, uint16_t&
base, int& ACK_index, queue<my_udp>& message_queue, int& packet_num,
clock_t& start) {
3     char RecvBuf[UDP_LEN];

```



```

4     my_udp Recv_udp;
5     int RecvAddrSize = sizeof(RecvAddr);
6
7     while (ACK_index < packet_num) { // 持续接收 ACK, 直到所有数据包被确认
8         if (recvfrom(SendSocket, RecvBuf, UDP_LEN, 0, (sockaddr*)&RecvAddr,
9 &RecvAddrSize) > 0) {
10             memcpy(&Recv_udp, RecvBuf, UDP_LEN);
11
12             // 检查是否是有效的 ACK 包
13             if (Recv_udp.udp_header.Flag == ACK &&
14 checksum((uint16_t*)&Recv_udp, UDP_LEN) == 0) {
15                 lock_guard<mutex> lock(mtx); // 保护 base 和队列的访问
16                 if ((base % DEFAULT_SEQNUM) == Recv_udp.udp_header.SEQ) {
17                     // 弹出队列中已确认的数据包
18                     if (!message_queue.empty()) {
19                         message_queue.pop();
20                     }
21                     start = clock(); // 重置计时器
22                     base++; // 滑动窗口前进
23                     ACK_index++;
24                     setConsoleColor(7);
25                     cout << "[ACK RECEIVED] base=" << base << ", ACK_index="
26 << ACK_index << endl;
27
28                 }
29                 else {
30                     setConsoleColor(12);
31                     cout << "[DUPLICATE ACK] SEQ=" <<
32 Recv_udp.udp_header.SEQ << endl;
33                 }
34             }
35         }
36     }
37 }
38 }
39 }
40 }
41 }
42 }
43 }
44 }
45 }

```

需要注意的是，**窗口的滑动**也是依靠接收函数来进行的，每当有一个数据包ACK被成功确认，base就会向前移动一格（移动一格的原因是因为接收端采用了每次接收到数据包后都会发送一个ACK包的机制）**计时器也会被重置**。如果接收到的ACK不是预期想要接受的ACK，则会打印输出[DUPLICATE ACK]，即收到了重复的ACK包，在等待计时器超时后就会触发**超时重传机制**。

3.GBN接收端设计

从整体来看，**接收端的GBN设计其实与Part1中的停等机制非常类似**，其中只是改变了发送的ACK包并不是对接收到的数据包ACK，而是当前维护的最大有序接收的数据包ACK，对于失序的数据包选择直接丢弃。**值得一提的是**，接收端也可以设计为**延时确认**，比如每接收若干个数据包后只返回最大的ACK即可，表示之前的数据包已经全部接收，这样可以减轻接收端的发送负担，同时，发送端的发送窗口亦可以一次**向前滑动多格**。

```

1 void recv_file_GBN(SOCKET& RecvSocket, sockaddr_in& SenderAddr, int&
2 SenderAddrSize) {
3     char* file_content = new char[MAX_FILESIZE];
4     string filename = "";

```



```

4   long size = 0;
5   int iResult = 0;
6   bool flag = true;
7
8   while (flag) {
9       char* RecvBuf = new char[UDP_LEN]();
10      my_udp temp;
11      iResult = recvfrom(RecvSocket, RecvBuf, UDP_LEN, 0,
(SOCKADDR*)&SenderAddr, &SenderAddrSize);
12      if (iResult == SOCKET_ERROR) {
13          setConsoleColor(12);
14          cout << "Recvfrom failed with error: " << WSAGetLastError() <<
endl;
15      }
16      else {
17          memcpy(&temp, RecvBuf, UDP_LEN);
18
19          //人为设置丢包率
20          int drop_probability = rand() % 100;
21          if (drop_probability < 3) {
22              continue;
23          }
24
25          if (temp.udp_header.Flag == START) {
26              // 验证未通过
27              if (checksum((uint16_t*)&temp, UDP_LEN) != 0 ||
temp.udp_header.SEQ != uint16_t(seq_order)) {
28                  setConsoleColor(12);
29                  cout << "#####传输出错! 等待重传中 #####" << endl;
30                  Send_ACK(RecvSocket, SenderAddr, SenderAddrSize);
31                  continue; // 不进行处理直接丢弃该数据包
32              }
33              else {
34                  filename = temp.buffer;
35                  setConsoleColor(1);
36                  cout << "#####文件名: " << filename << endl;
37
38                  print_Recv_information(temp);
39                  // 发送ACK0的响应即可
40                  Send_ACK(RecvSocket, SenderAddr, SenderAddrSize);
41                  // check_seq();
42              }
43          }
44          else if (temp.udp_header.Flag == OVER) {
45              if (checksum((uint16_t*)&temp, UDP_LEN) != 0 ||
temp.udp_header.SEQ != uint16_t(seq_order + 1)) {
46                  setConsoleColor(12);
47                  cout << "#####Something wrong!! wait ReSend!! #####" <<
endl;
48                  Send_ACK(RecvSocket, SenderAddr, SenderAddrSize);
49                  continue; // 不进行处理直接丢弃该数据包
50              }
51              else {
52                  memcpy(file_content + size, temp.buffer,
temp.udp_header.datasize);
53                  size += temp.udp_header.datasize;

```

```

54         print_Rcv_information(temp);
55
56         ofstream fout(filename, ofstream::binary);
57         fout.write(file_content, size);
58         fout.close();
59         flag = false;
60
61         // SEQ回环
62         if (temp.udp_header.SEQ = uint16_t(seq_order + 1)) {
63             seq_order++;
64         }
65         Send_ACK(RcvSocket, SenderAddr, SenderAddrSize);
66         setConsoleColor(1);
67         cout << "####文件大小: " << size << " bytes" << endl;
68         setConsoleColor(10);
69         cout << "####成功接收文件! ####" << endl << endl;
70     }
71 }
72 // START_OVER表征发送端断连
73 else if (temp.udp_header.Flag == START_OVER) {
74     flag = false;
75     ready2quit = 1;
76     setConsoleColor(7);
77     cout << "####Sender马上断开连接! ####" << endl;
78 }
79 else {
80
81     if (checksum((uint16_t*)&temp, UDP_LEN) != 0 ||
temp.udp_header.SEQ != uint16_t(seq_order + 1)) {
82         setConsoleColor(12);
83         cout << "####传输出错! 等待重传中 ####" << endl;
84         Send_ACK(RcvSocket, SenderAddr, SenderAddrSize);
85         continue; // 不进行处理直接丢弃该数据包
86     }
87     else {
88         memcpy(file_content + size, temp.buffer,
temp.udp_header.datasize);
89         size += temp.udp_header.datasize;
90
91         print_Rcv_information(temp);
92         // 保留已经确认的分组的一个的序列号
93         if (temp.udp_header.SEQ = uint16_t(seq_order + 1)) {
94             seq_order++;
95         }
96         Send_ACK(RcvSocket, SenderAddr, SenderAddrSize);
97     }
98 }
99 }
100
101 delete[] RcvBuf;
102 }
103
104 stream_seq_order++;
105 check_stream_seq();
106 // 每一次获取文件后, 将seq_order清零
107 seq_order = 0;

```

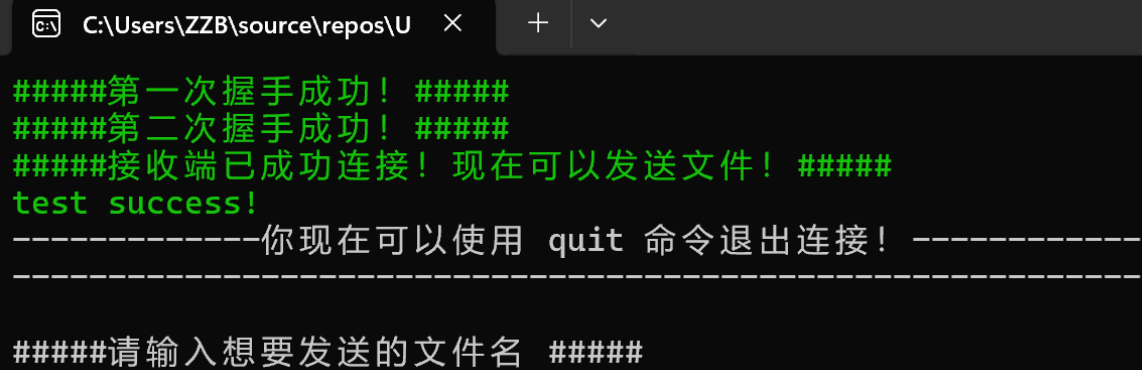
```
108     delete[] file_content;  
109 }
```

需要注意的是在发送文件名的第一个数据分组时，仍然采取的是停等协议内容，所以这里的 0 序列号确认后就不需要进行+1的操作了。

四、实验结果

1. 文件传输

首先是三次握手建立连接，这里与Part1一致



```
C:\Users\ZZB\source\repos\U  ×  +  ∨  
#####第一次握手成功! #####  
#####第二次握手成功! #####  
#####接收端已成功连接! 现在可以发送文件! #####  
test success!  
-----你现在可以使用 quit 命令退出连接! -----  
-----  
#####请输入想要发送的文件名 #####
```

这里我们选择发送helloworld.txt文件来进行测试，可以看到一些**日志信息**，包括文件大小，校验和。将要发送的数据包大小，滑动窗口大小等。

```
C:\Users\ZZB\source\repos\U  ×  +  ∨

#####第一次握手成功! #####
#####第二次握手成功! #####
#####接收端已成功连接! 现在可以发送文件! #####
test success!
-----你现在可以使用 quit 命令退出连接! -----
-----

#####请输入想要发送的文件名 #####
helloworld.txt

#####文件大小: 1655808 bytes
#####文件名校验和: 63258
#####送数据包的数量: 1618
#####滑动窗口大小: 32
已发送数据包0!   数据包大小为 14 bytes
校验和:63258      ACK:16
数据包已确认被接受!      当前发送进度:0/1618      ACK:2

[SEND] Packet SEQ=1 Checksum = 58339
[SEND] Packet SEQ=2 Checksum = 58338
[SEND] Packet SEQ=3 Checksum = 58337
[SEND] Packet SEQ=4 Checksum = 58336
[SEND] Packet SEQ=5 Checksum = 58335
[SEND] Packet SEQ=6 Checksum = 58334
[SEND] Packet SEQ=7 Checksum = 58333
[SEND] Packet SEQ=8 Checksum = 58332
[SEND] Packet SEQ=9 Checksum = 58331
[SEND] Packet SEQ=10 Checksum = 58330
[SEND] Packet SEQ=11 Checksum = 58329
[SEND] Packet SEQ=12 Checksum = 58328
```

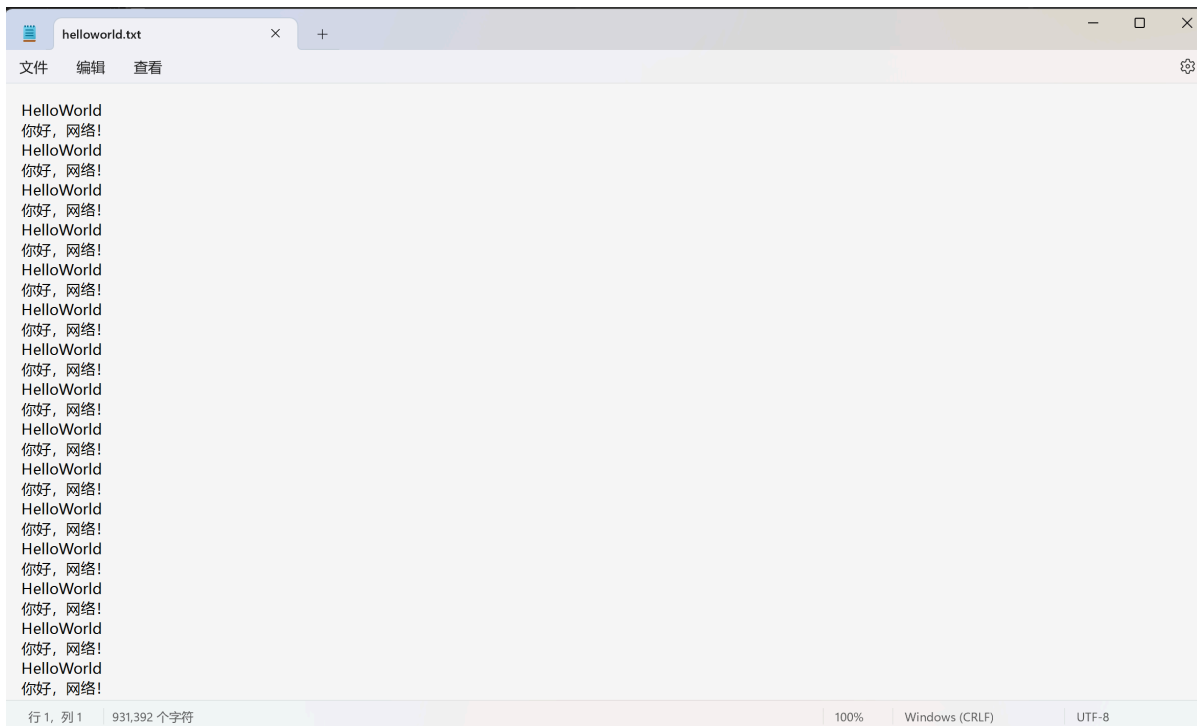
发送结束后检查文件，发现可以正常打开，传输成功！

```
已接收数据包1616      数据包大小为1024 bytes!
Check Sum:56724
已向发送端发送ACK包! 1616

已接收数据包1617      数据包大小为1024 bytes!
Check Sum:56723
已向发送端发送ACK包! 1617

已接收数据包1618      数据包大小为0 bytes!
Check Sum:57738
已向发送端发送ACK包! 1618

#####文件大小: 1655808 bytes
#####成功接收文件! #####
```



2. 丢包处理与超时重传

对于丢包，在发送端会打印如下日志输出，这里会重复打印这么多[DUPLICATE ACK]是因为在**多线程机制**下，接收ACK的线程每时每刻都在接收到重复的ACK，也就会一遍遍打印输出，直到主线程的while（）循环确认了超时重传触发，即[RESEND]输出，会将缓存区中所有的数据包重新发送。

```
[DUPLICATE ACK] SEQ=44
[DUPLICATE ACK] SEQ=44
[DUPLICATE ACK] SEQ=44
[DUPLICATE ACK] SEQ=44
[DUPLICATE ACK] SEQ=44
[DUPLICATE ACK] SEQ=44
[DUPLICATE ACK] SEQ=44
[DUPLICATE ACK] SEQ=44
[DUPLICATE ACK] SEQ=44
[DUPLICATE ACK] SEQ=44
[DUPLICATE ACK] SEQ=44
[DUPLICATE ACK] SEQ=44
[TIMEOUT] 重发未确认数据包。
[RESEND] Packet SEQ=45 resent.
[RESEND] Packet SEQ=46 resent.
[RESEND] Packet SEQ=47 resent.
[RESEND] Packet SEQ=48 resent.
[RESEND] Packet SEQ=49 resent.
[RESEND] Packet SEQ=50 resent.
[RESEND] Packet SEQ=51 resent.
```

五、总结与分析

1. 遇到的问题

本次实验遇到了诸多问题，主要包括**多线程**、**超时重传**以及**窗口大小**问题等

- 在处理**多线程**的问题时，遇到了引用传参失败以及全局变量冲突等问题（其实也是多线程中老生常谈的问题）通过不断调试并查找资料，添加了锁机制，问题得到解决。
- 超时重传**中出现的问题其实是与**窗口大小**问题紧密相关的，由于我一开始测试的是使用的窗口大小是5，问题还没有暴露，但是后来助教在群里通知，要求发送端窗口在20-32之间（所以不怪我bushi），然后问题就出现了，当我上调窗口大小至18及以上时，超时重传机制似乎就失效了，发送端会停滞在某一个数据包的发送中，接收端也会一直发送同一个ACK，即双方预期接收的ACK出现了错位。

至于原因，在我反复调试和查阅相关资料后，我怀疑问题可能出现在**缓存区大小**上，当发送窗口过大时，大量的数据包可能超出了队列缓存大小，导致了相关错误。所以我将原来每个数据包的大小从4096下调至1024，问题就得到了解决，甚至32大小的窗口也很稳定不出错。

2. 总结

在本次实验之中，深入地了解了**滑动窗口的运用、基于累计确认思想的GBN算法**。在实现过程之中，对于线程以及消息队列的相关知识进行了深入了解，将其与停等机制进行对比，确认类似流水线的形式能够带来很大的传输效率提升，进一步理解和体会了计 算机网络中可靠传输协议的设计与实现。