



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

---

## UDP 可靠传输 Part2

---

2014074 费泽锟

年级：2020 级

专业：信息安全

指导教师：徐敬东、张建忠

2024 年 11 月 27 日

## 摘要

在 UDP 实现可靠传输的 Part2 的设计中，主要是因为 Part1 之中实现的停等协议传输效率过低的原因，在停等协议之中，每次传输一个数据分组都得循环等待，直到接收对应的 ACK 消息为止。而本次设计之中，可选择的算法包括 GBN (go back N 算法) 和 SR (选择重传算法)，基于累计确认的思想，完成类似于流水线地发送数据分组形式。

在 Part2 的设计之中，实现了 GBN 算法和简略的 SR 算法，在以下部分会具体讲解实现的细节。在接收端之中，设置了人为的丢包率（也就是接收到了数据包，但是不对其进行处理），在传输过程之中的 log 信息验证了算法的正确性。

**关键字：UDP 可靠传输、GBN、SR、计时器设置**

## 目录

<b>一、 Part2 设计</b>	<b>1</b>
(一) GBN 发送端设计 . . . . .	1
(二) GBN 接收端设计 . . . . .	5
(三) SR 设计 . . . . .	9
<b>二、 可靠 UDP 传输流程展示</b>	<b>10</b>
<b>三、 总结</b>	<b>13</b>

## 一、 Part2 设计

在 Part2 的设计之中，我们可以选择实现 GBN 算法，也可以选择实现 SR 算法。这两种算法的最大差别就是，对于出现丢包之后的 SEQ 序列号出现失序的情形时，SR 算法会将滑动窗口内接受到的数据分组均标记为已经接收，并回复相应的 ACK 消息，而 GBN 算法则不会对失序到达的数据分组进行处理。

所以这就引出了计时器方面的巨大差距，GBN 算法只需要设置一个计时器即可，而 SR 算法需要设置多个计时器，也就是对每一个数据分组设置一个计时器，如果单纯地使用 clock 类型的数组来表示多个计时器，严格来讲它仍然是在运行的发送端程序之中进行每一个超时判断的处理，所以最理想的情形是使用 set timer 对每一个分组设置计时器，这样的话就得对每一个分组单开一个线程，且每一个线程都得 set timer 设置计时器，还涉及到子线程与主线程的消息接收问题，十分复杂。

因此，在 Part2 的设计之中，主要实现了完整的同步的 GBN 算法，并且还完成了基于一个计时器的简略 SR 算法。

### (一) GBN 发送端设计

对于 GBN 算法的实现，我们首先需要了解 GBN 算法之中的发送端和接收端的有限状态机的转换过程，以此为基础才能实现正确的 GBN 算法，下图展示的是 GBN 算法中的发送端有限状态机。

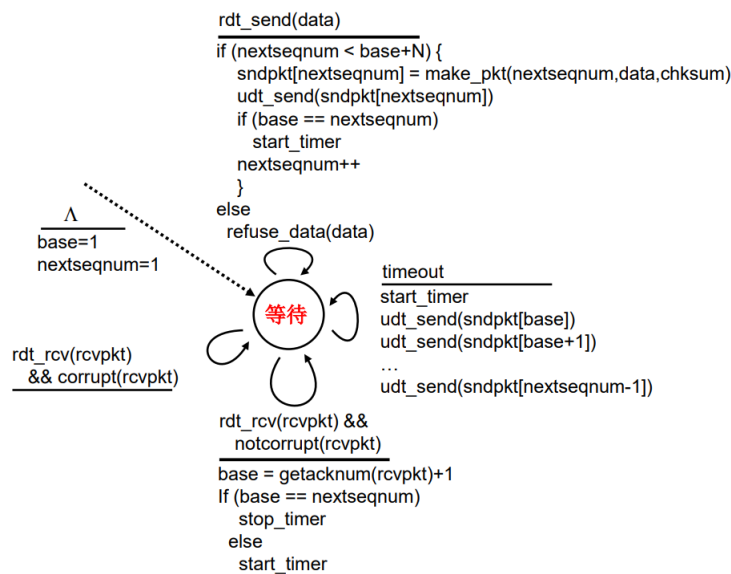


图 1: 发送端有限状态机

我们分析一下这个 FSM，首先就是我们需要一个滑动窗口，如果说在发送的过程之中这个滑动窗口还有空余的位置，且有需要继续发送的数据分组，那么就继续发送。我们需要一个 base 全局变量来标识窗口的首部，和一个 nextseqnum 来表示窗口的尾部（也就是将要发送的下一个数据分组的序列号）。

除此之外，我们在 GBN 的发送端之中只需要设置一个计时器即可，该计时器在发送第一个数据包之后开启，根据有限状态机之中的内容，当每次成功接收到序列号为 base 的 ACK 响应消息后，将会重新设置计时器，并且 base 向后移动一位。

如果在收到校验和没有检验成功的 ACK 响应消息时，只需要不对其进行任何处理即可。在计时器超出了我们设置的阈值时间后，就会将窗口内未成功接收到的数据包重新发送一遍。

具体的 sender 源代码如下：

#### GBN 算法（发送端）

```

1 // GBN部分的全局变量
2 const int N = 5; // 窗口大小
3 uint16_t base = 1; // 初始为1
4 uint16_t next_seqnum = 1;
5 int ACK_index = 0; // 判断结束
6 queue<my_udp> message_queue;
7
8 void GBN_init() {
9     base = 1;
10    next_seqnum = 1;
11    ACK_index = 0;
12 }
13
14 // GBN的发送文件函数
15 void send_file_GBN(string filename, SOCKET& SendSocket, sockaddr_in& RecvAddr
16 ) {
17     // 每次文件发送预先初始化
18     GBN_init();
19     int RecvAddrSize = sizeof(RcvAddr);
20
21     ifstream fin(filename.c_str(), ifstream::binary);
22     fin.seekg(0, std::ifstream::end);
23     long size = fin.tellg();
24     file_size = size;
25     fin.seekg(0);
26
27     char* binary_file_buf = new char[size];
28     cout << " ** 文件大小: " << size << " bytes" << endl;
29     fin.read(&binary_file_buf[0], size);
30     fin.close();
31
32     HEADER udp_header(filename.length(), 0, START, stream_seq_order, 0);
33     my_udp udp_packets(udp_header, filename.c_str());
34     uint16_t check = checksum((uint16_t*)&udp_packets, UDP_LEN); // 计算校验
35     和
36     udp_packets.udp_header.cksum = check;
37
38     int packet_num = size / DEFAULT_BUFLen + 1;
39     cout << " ** 文件名校验和: " << check << endl;
40     cout << " ** 发送数据包的数量: " << packet_num << endl;
41     cout << " ** Windows窗口大小: " << N << endl;
42
43     // 正常发送第一个文件名数据包

```

```

42     send_packet(udp_packets, SendSocket, RecvAddr);
43
44     clock_t start;
45     char* RecvBuf = new char[UDP_LEN];
46     my_udp Recv_udp;
47     start = clock();
48
49     // 处理SEQ回环, mod运算, 商和余数
50     // uint16_t quotient = 0;
51     uint16_t remainder = 0;
52     while (ACK_index < packet_num) {
53         if (next_seqnum < base + N && next_seqnum <= packet_num) {
54             // quotient = next_seqnum / DEFAULT_SEQNUM;
55             remainder = next_seqnum % DEFAULT_SEQNUM;
56             if (next_seqnum == packet_num) {
57                 udp_header.set_value(size - (next_seqnum - 1) *
58                                     DEFAULT_BUFLen, 0, OVER, stream_seq_order, remainder);
59                 udp_packets.set_value(udp_header, binary_file_buf + (
60                                     next_seqnum - 1) * DEFAULT_BUFLen, size - (next_seqnum -
61                                     1) * DEFAULT_BUFLen);
62                 check = checksum((uint16_t*)&udp_packets, UDP_LEN);
63                 udp_packets.udp_header.cksum = check;
64
65                 send_packet_GBN(udp_packets, SendSocket, RecvAddr);
66                 print_Send_information(udp_packets, "Send");
67                 message_queue.push(udp_packets);
68                 next_seqnum++;
69             }
70             else {
71                 udp_header.set_value(DEFAULT_BUFLen, 0, 0, stream_seq_order,
72                                     remainder);
73                 udp_packets.set_value(udp_header, binary_file_buf + (
74                                     next_seqnum - 1) * DEFAULT_BUFLen, DEFAULT_BUFLen);
75                 check = checksum((uint16_t*)&udp_packets, UDP_LEN);
76                 udp_packets.udp_header.cksum = check;
77
78                 send_packet_GBN(udp_packets, SendSocket, RecvAddr);
79                 print_Send_information(udp_packets, "Send");
80                 message_queue.push(udp_packets);
81                 next_seqnum++;
82             }
83         }
84     }
85
86     if (clock() - start > MAX_TIME) {
87         cout << "*** TIME OUT! ReSend Message *** " << endl;
88         start = clock();
89
90         for (int i = 0; i < message_queue.size(); i++) {

```

```

85         send_packet_GBN(message_queue.front(), SendSocket, RecvAddr);
86         print_Send_information(message_queue.front(), "ReSend");
87         message_queue.push(message_queue.front());
88         message_queue.pop();
89         // Sleep(10);
90     }
91 }
92
93 // 循环之中也接收ACK消息, 可封装
94 if (recvfrom(SendSocket, RecvBuf, UDP_LEN, 0, (sockaddr*)&RecvAddr, &
95 RecvAddrSize) > 0) {
96     memcpy(&Recv_udp, RecvBuf, UDP_LEN);
97     if (Recv_udp.udp_header.Flag == ACK && checksum((uint16_t*)&
98 Recv_udp, UDP_LEN) == 0) {
99         cout << "base: " << base << endl;
100        cout << "nextseqnum " << next_seqnum << endl;
101        // 丢弃重复响应的ACK, 取模防止回环问题
102        if ((base % DEFAULT_SEQNUM) == Recv_udp.udp_header.SEQ) {
103            base = base + 1; // 确认一个移动一个位置
104            cout << "Send has been confirmed! Flag:" << Recv_udp.
105                udp_header.Flag;
106            cout << " STREAM_SEQ:" << Recv_udp.udp_header.STREAM_SEQ
107                << " SEQ:" << Recv_udp.udp_header.SEQ << endl;
108            ACK_index++;
109            message_queue.pop();
110            start = clock();
111        }
112        else {
113            cout << "Repetitive ACK! Flag:" << Recv_udp.udp_header.
114                Flag;
115            cout << " STREAM_SEQ:" << Recv_udp.udp_header.STREAM_SEQ
116                << " SEQ:" << Recv_udp.udp_header.SEQ << endl;
117        }
118    }
119    else;
120 }
121
122 cout << "----- ***对方已成功接收文件! ***----- " << endl << endl;
123 stream_seq_order++;
124 check_stream_seq();
125 delete[] binary_file_buf;
126 }

```

因为对 Part1 部分进行了较好的封装, 我们这里只需要单独编写 send\_packet\_GBN 函数和 send\_file\_GBN 函数即可, 其余部分的框架不需要进行改变。

在 send\_packet\_GBN 函数之中, 不同于停等协议的部分是, 我们只需要将这个数据包发送出去, 判读 send 函数发送有无成功即可, 不需要在发送数据包之后一直等待对应的 ACK 响应

消息。

在 `send_file_GBN` 函数实现的过程之中，对于窗口的设置，因为 GBN 算法是严格遵循顺序接受的算法，所以我们不需要设置 `map` 类型的窗口或者 `char` 类型的二维数组窗口，来标记其是否被成功确认。GBN 算法之中，成功确认的一定是 `base` 序列号的数据包。所以我们使用 `queue` 数据类型就能够极大地简便对应的操作。

在设置了 `message_queue` 全局变量来表示发送窗口后，我们还设置了 `ACN_index` 全局变量，来记录一个文件已经被确认的数据分组的数量。这里我只设置了一层 `while` 循环就可以解决发送端的发送与接受问题，`while` 循环的判断条件为：已经确认的 ACK 数量与想要发送的文件的数据包数量不同，就会一直持续循环。

在该循环之中，只要窗口大小没有满，那么就会在每次循环之中发送下一个数据包，并将其加入 `message_queue` 队列之中。并且在循环之中会查看接收缓冲区有无接收到新的数据包，如果接收到了新的数据包就会进行 SEQ 和校验和检验，检验通过后将 `message_queue` 的队首进行出队列的操作，ACK 确认的值 +1。

如果超时则会遍历队列，将队列之中的每一个数据包重新发送。在发送端的设计之中，还利用了 `uint16_t` 数据类型如果 +1 溢出时会变为 0，这一特点，在判断条件处设置了强制类型转换，且设置了文件偏移来完成了对序列号回卷的处理。

到此，根据发送端有限状态机的发送函数编写完成。

下图展示了 GBN 算法可能出现的实际传输情况：

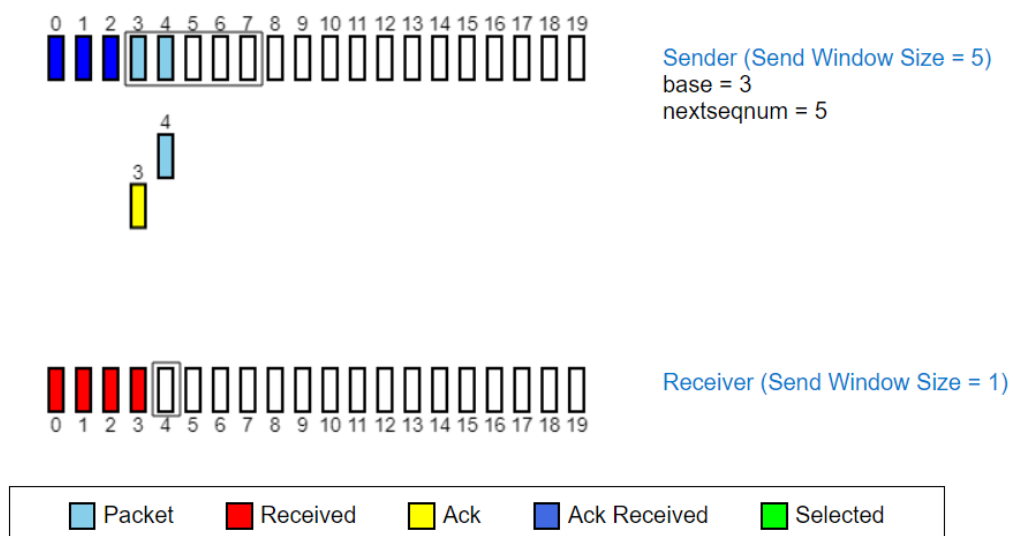


图 2: GBN 算法可能的实际传输情况

## (二) GBN 接收端设计

对于 GBN 算法的接收端设计，我们还是先对接收端的有限状态机进行分析，下图展示的是 GBN 算法中的接收端有限状态机。

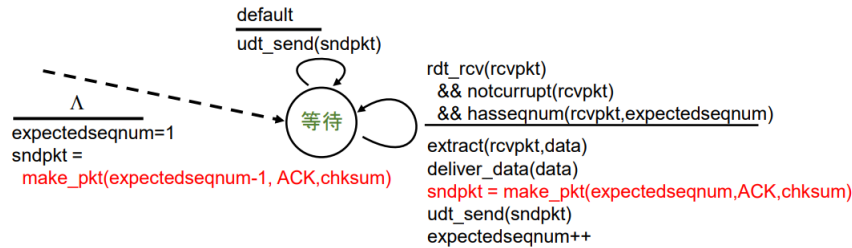


图 3: 接收端有限状态机

在接收端的有限状态机之中，可以发现其发送 ACK 响应消息的逻辑与停等协议之中的很为相似。在 GBN 算法的接收端中，只使用 ACK 确认按序正确接收的最高序号分组，这一点与我在 Part1 之中对停等协议的设计有所不同，在停等协议的设计之中，接收端发送的是期望获得的下一个序列号，所以这里需要修改。

接收端的 FSM：只使用 ACK 确认按序正确接收的最高序号分组，当出现数据包失序的时候，会产生重复的 ACK，需要保存希望接收的分组序号。对于失序分组的处理，采取的策略则是不缓存丢弃，并且重发 ACK，确认按序正确接收的最高序号分组。

具体发送端的源代码如下：

#### GBN 算法（接收端）

```

1 void recv_file_GBN(SOCKET& RecvSocket, sockaddr_in& SenderAddr, int&
  SenderAddrSize) {
2   char* file_content = new char[MAX_FILESIZE]; // 偷懒了，直接调大
3   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      cout << "Recvfrom failed with error: " << WSAGetLastError() <<
        endl;
14    }
15    else {
16      memcpy(&temp, RecvBuf, UDP_LEN);
17
18      // 仅作为测试
19      int drop_probability = rand() % 10;
20      cout << drop_probability << endl;
21      if (drop_probability < 1) {
22        continue;
23      }
24    }
  }
  
```



```

25     if (temp.udp_header.Flag == START) {
26         // 验证未通过, uint16_t是为了处理SEQ回环的
27         if (checksum((uint16_t*)&temp, UDP_LEN) != 0 || temp.
28             udp_header.SEQ != uint16_t(seq_order)) {
29             cout << "*** Something wrong!! Wait ReSend!! *** " <<
30                 endl;
31             Send_ACK(RecvSocket, SenderAddr, SenderAddrSize);
32             continue; // 不进行处理直接丢弃该数据包
33         }
34         else {
35             filename = temp.buffer;
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.
46             udp_header.SEQ != uint16_t(seq_order + 1)) {
47             cout << "*** Something wrong!! Wait ReSend!! *** " <<
48                 endl;
49             Send_ACK(RecvSocket, SenderAddr, SenderAddrSize);
50             continue; // 不进行处理直接丢弃该数据包
51         }
52         else {
53             memcpy(file_content + size, temp.buffer, temp.udp_header.
54                 datasize);
55             size += temp.udp_header.datasize;
56             print_Recv_information(temp);
57
58             ofstream fout(filename, ofstream::binary);
59             fout.write(file_content, size); // 这里还是size,如果使用
60                 string.data或c_str的话图片不显示, 经典深拷贝问题
61             fout.close();
62             flag = false;
63
64             // SEQ回环
65             if (temp.udp_header.SEQ == uint16_t(seq_order + 1)) {
66                 seq_order++;
67             }
68             Send_ACK(RecvSocket, SenderAddr, SenderAddrSize);
69
70             cout << "*** 文件大小: " << size << " bytes" << endl;
71             cout << "-----*** 成功接收文件 ***-----" << endl << endl;
72         }
73     }

```

```

67     }
68     // START_OVER表征发送端断连
69     else if (temp.udp_header.Flag == START_OVER) {
70         flag = false;
71         ready2quit = 1; // 偷懒 (全局变量标识准备进行四次挥手)
72         cout << "-----*** Sender马上断开连接! ***-----" << endl;
73     }
74     else {
75         // 这里可以封装一个Send_ACK
76         if (checksum((uint16_t*)&temp, UDP_LEN) != 0 || temp.
77             udp_header.SEQ != uint16_t(seq_order + 1)) {
78             cout << "*** Something wrong!! Wait ReSend!! *** " <<
79                 endl;
80             Send_ACK(RecvSocket, SenderAddr, SenderAddrSize);
81             continue; // 不进行处理直接丢弃该数据包
82         }
83         else {
84             memcpy(file_content + size, temp.buffer, temp.udp_header.
85                 datasize);
86             size += temp.udp_header.datasize;
87
88             print_Recv_information(temp);
89             // 保留已经确认的分组的最后一个的序列号
90             if (temp.udp_header.SEQ == uint16_t(seq_order + 1)) {
91                 seq_order++;
92             }
93             Send_ACK(RecvSocket, SenderAddr, SenderAddrSize);
94         }
95     }
96 }
97
98 delete[] RecvBuf;
99
100 stream_seq_order++;
101 check_stream_seq();
102 // 每一次获取文件后, 将seq_order清零
103 seq_order = 0;
104 delete[] file_content;
105 }

```

源代码之中的内容与停等协议基本一致, 只不过在发送 ACK 序列号的过程之中, 发送的是已经确认的最高序号分组, 并且在每次接收到比该最高序号分组大 1 的数据分组时, 对最高序号进行更新。

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。

### (三) SR 设计

讲完了 GBN 算法之中的内容后，这一部分讲解的是我对简略 SR 算法实现的部分，以及对 SR 算法实现的思考与探究。

我在实现 SR 算法的过程之中，因为如果对每个分组单开一个线程设置计时器（使用 set timer）会出现主线程与子线程的消息通讯的各种问题，这些显然都是数据分组多线程的问题，所以在实现时就只设置了一个计时器，采取的策略与 GBN 算法的计时器更新策略相同。

如果只使用一个计时器，那么就是当 base 数据分组未被确认超时后，将发送缓冲区之中的所有未被确认的数据分组都进行重发，虽然这对于每个数据分组而言一定会有延迟，但是也能大大提升发送效率（本实验之中，程序运行导致的延迟并不是很大）。

SR 算法中额外变量

```

1 #define BUFFER_SIZE sizeof(Packet) // 缓冲区大小
2 #define WINDOW_SIZE 16 // 滑动窗口大小
3
4 unsigned int packetNum; // 发送数据包的数量
5 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
6 unsigned int nextSeqNum; // 指向下一个可用但还未发送的分组序号
7
8 char** selectiveRepeatBuffer; // 选择重传缓冲区

```

在 SR 算法实现的过程之中，我们就不能简单地使用一个 queue 类型的消息队列来解决问题了，我们需要设置发送缓冲区，以及对应大小的缓冲区确认数组，因为缓冲区涉及到寻址过程，所以这里实现时采用了二维数组的形式作为输入缓冲区（也就是滑动窗口）。

当接收到对应的序列号的 ACK 消息后，需要对其进行寻址，将缓冲区的标记数组设置为 1，如果在 base 已经确认之后，一直移动 base 的位置直到第一个没有确认的数据分组。

下图展示了 SR 算法可能出现的实际传输情况：

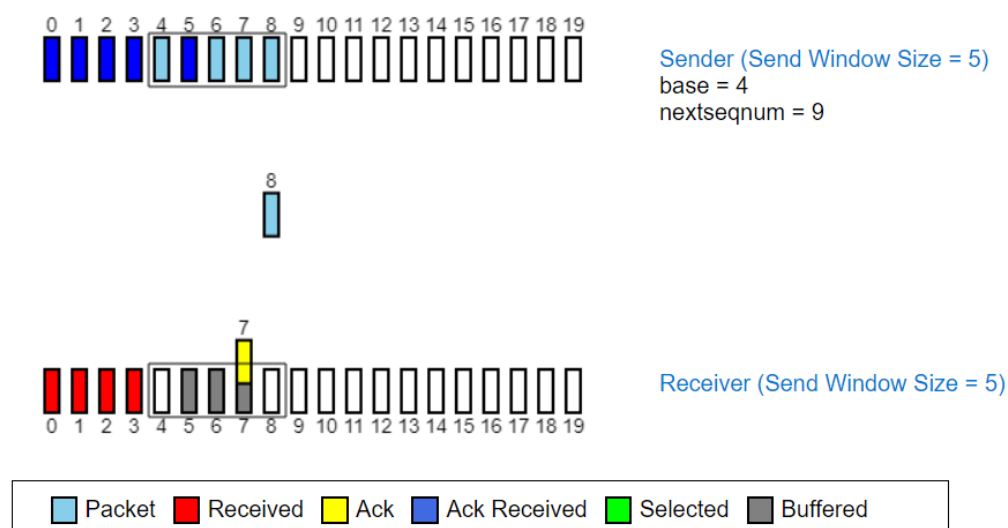


图 4: SR 算法可能的实际传输情况

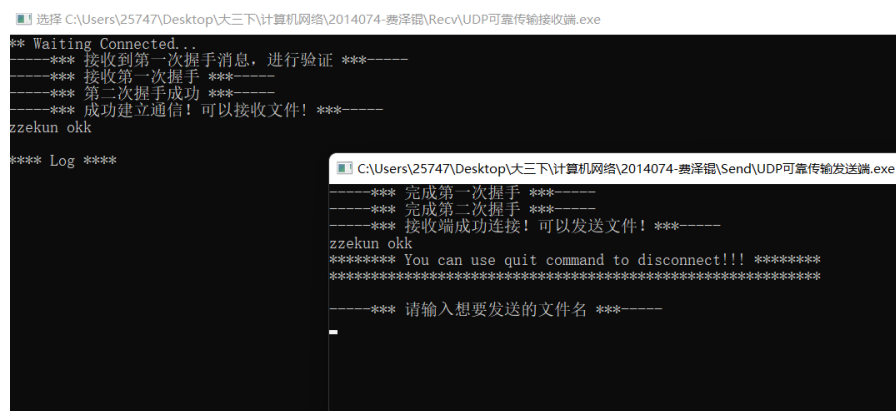
## 二、可靠 UDP 传输流程展示

又是讲了一大堆理论和代码的内容，显得枯燥乏味，这里将会直接展示，GBN 算法在出现丢包时会执行的操作，来验证编写的 GBN 算法的正确性。

三次握手和四次挥手的过程沿用了 Part1 部分的 connect 和 disconnect 部分，同样当发送端不想继续发送数据时，只需要输入 quit 命令就可以进行四次挥手的过程退出连接了。

对于丢包的设置也沿用了 Part1 的设计，生成一个 0-9 的随机数，小于 1 时对数据包不对数据包进行处理也就是模拟了 10% 丢包率时的丢包的操作。

首先就是三次握手建立连接：



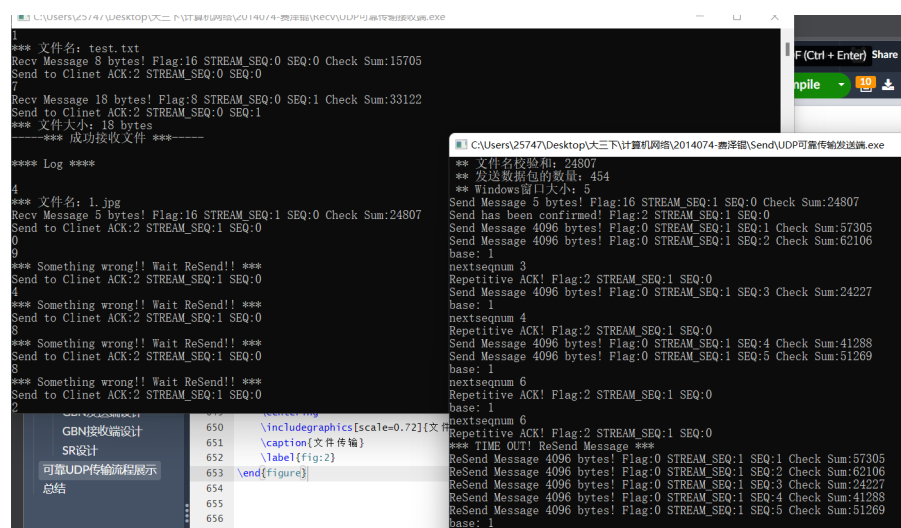
```
C:\Users\25747\Desktop\大三下\计算机网络\2014074-费泽锦\Recv\UDP可靠传输接收端.exe
*** Waiting Connected...
*** 接收到第一次握手消息，进行验证 ***-----
*** 接收第一次握手 ***-----
*** 第二次握手成功 ***-----
*** 成功建立通信！可以接收文件！ ***-----
zzekun okk

*** Log ***

C:\Users\25747\Desktop\大三下\计算机网络\2014074-费泽锦\Send\UDP可靠传输发送端.exe
*** 完成第一次握手 ***-----
*** 完成第二次握手 ***-----
*** 接收端成功连接！可以发送文件！ ***-----
zzekun okk
***** You can use quit command to disconnect!!! *****
*****
*** 请输入想要发送的文件名 ***-----
```

图 5: 三次握手

接下来我们选择一个测试文件来进行传输：



```
C:\Users\25747\Desktop\大三下\计算机网络\2014074-费泽锦\Recv\UDP可靠传输接收端.exe
*** 文件名: test.txt
Recv Message 8 bytes! Flag:16 STREAM_SEQ:0 SEQ:0 Check Sum:15705
Send to Client ACK:2 STREAM_SEQ:0 SEQ:0
7
Recv Message 18 bytes! Flag:8 STREAM_SEQ:0 SEQ:1 Check Sum:33122
Send to Client ACK:2 STREAM_SEQ:0 SEQ:1
*** 文件大小: 18 bytes
*** 成功接收文件 ***-----

*** Log ***

4
*** 文件名: 1.jpg
Recv Message 5 bytes! Flag:16 STREAM_SEQ:1 SEQ:0 Check Sum:24807
Send to Client ACK:2 STREAM_SEQ:1 SEQ:0
9
*** Something wrong!! Wait ReSend!! ***
Send to Client ACK:2 STREAM_SEQ:1 SEQ:0
4
*** Something wrong!! Wait ReSend!! ***
Send to Client ACK:2 STREAM_SEQ:1 SEQ:0
8
*** Something wrong!! Wait ReSend!! ***
Send to Client ACK:2 STREAM_SEQ:1 SEQ:0
8
*** Something wrong!! Wait ReSend!! ***
Send to Client ACK:2 STREAM_SEQ:1 SEQ:0
2

C:\Users\25747\Desktop\大三下\计算机网络\2014074-费泽锦\Send\UDP可靠传输发送端.exe
*** 文件名校验和: 24807
*** 发送数据包的数量: 454
*** Windows窗口大小: 5
Send Message 5 bytes! Flag:16 STREAM_SEQ:1 SEQ:0 Check Sum:24807
Send has been confirmed! Flag:2 STREAM_SEQ:1 SEQ:0
Send Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:1 Check Sum:57305
Send Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:2 Check Sum:62106
base: 1
nextseqnum 3
Repetitive ACK! Flag:2 STREAM_SEQ:1 SEQ:0
Send Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:3 Check Sum:24227
base: 1
nextseqnum 4
Repetitive ACK! Flag:2 STREAM_SEQ:1 SEQ:0
Send Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:4 Check Sum:41288
Send Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:5 Check Sum:51269
base: 1
nextseqnum 6
Repetitive ACK! Flag:2 STREAM_SEQ:1 SEQ:0
base: 1
nextseqnum 6
Repetitive ACK! Flag:2 STREAM_SEQ:1 SEQ:0
*** TIME OUT! ReSend Message ***
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:1 Check Sum:57305
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:2 Check Sum:62106
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:3 Check Sum:24227
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:4 Check Sum:41288
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:5 Check Sum:51269
base: 1
nextseqnum 6
```

图 6: 文件传输

我们接着来看看其中的具体出现丢包重传时的过程，当出现丢包时，发送端会显示随机数为 0，并且对于失序的数据包丢弃，持续发送当前确认的最高分组序号的 ACK 响应。如下图：

```

*** Log ***
4
*** 文件名: 1.jpg
Recv Message 5 bytes! Flag:16 STREAM_SEQ:1 SEQ:0 Check Sum:24807
Send to Clinet ACK:2 STREAM_SEQ:1 SEQ:0
0
9
*** Something wrong!! Wait ReSend!! ***
Send to Clinet ACK:2 STREAM_SEQ:1 SEQ:0
4
*** Something wrong!! Wait ReSend!! ***
Send to Clinet ACK:2 STREAM_SEQ:1 SEQ:0
8
*** Something wrong!! Wait ReSend!! ***
Send to Clinet ACK:2 STREAM_SEQ:1 SEQ:0
8
*** Something wrong!! Wait ReSend!! ***
Send to Clinet ACK:2 STREAM_SEQ:1 SEQ:0
2
Recv Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:1 Check Sum:57305
Send to Clinet ACK:2 STREAM_SEQ:1 SEQ:1
4
Recv Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:2 Check Sum:62106
Send to Clinet ACK:2 STREAM_SEQ:1 SEQ:2
5
Recv Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:3 Check Sum:24227
Send to Clinet ACK:2 STREAM_SEQ:1 SEQ:3

```

图 7: 接收端 Log

很幸运的是,传输过程之中第一个数据分组就被丢弃了(就不需要麻烦我们去找了...)。可以看到随机数为 0 产生了丢包的操作,接着 log 信息之中会显示“Something wrong!! Wait ReSend!!”字样,并且会显示发送的 ACK 响应数据包的具体内容。

可以看到其中的 SEQ 号一直为 0,也就是第一个数据包 base 处于的位置,当前设置的窗口大小为 5,那么在重发了四次 SEQ 为 0 的响应消息后,继续进行了接收数据包的操作。

接下来我们来看看发送端的 log 信息会给我们什么信息:

```

C:\Users\25747\Desktop\大三下\计算机网络\2014074-费泽福\Send\UDP可靠传输发送端.exe
** 文件名校验和: 24807
** 发送数据包的数量: 454
** Windows窗口大小: 5
Send Message 5 bytes! Flag:16 STREAM_SEQ:1 SEQ:0 Check Sum:24807
Send has been confirmed! Flag:2 STREAM_SEQ:1 SEQ:0
Send Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:1 Check Sum:57305
Send Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:2 Check Sum:62106
base: 1
nextseqnum 3
Repetitive ACK! Flag:2 STREAM_SEQ:1 SEQ:0
Send Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:3 Check Sum:24227
base: 1
nextseqnum 4
Repetitive ACK! Flag:2 STREAM_SEQ:1 SEQ:0
Send Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:4 Check Sum:41288
Send Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:5 Check Sum:51269
base: 1
nextseqnum 6
Repetitive ACK! Flag:2 STREAM_SEQ:1 SEQ:0
base: 1
nextseqnum 6
Repetitive ACK! Flag:2 STREAM_SEQ:1 SEQ:0
*** TIME OUT! ReSend Message ***
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:1 Check Sum:57305
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:2 Check Sum:62106
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:3 Check Sum:24227
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:4 Check Sum:41288
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:5 Check Sum:51269
base: 1
nextseqnum 6

```

图 8: 发送端 Log

我们可以看到 base 因为接收到了重复的 ACK 响应后会显示“Repetitive ACK!”字样,并且显示的接受的重复 ACK 消息的 SEQ 均为 0。但是这其中 nextseqnum 会一直递增,直到其变为 6,此时的 base 为 1,也就是此时的发送了 5 个数据分组,窗口可用大小已经为 0 了。

当超时了之后就会重新将 queue 队列之中的数据分组全部重发,如下图展示的一样:

```

nextseqnum 0
Repetitive ACK! Flag:2 STREAM_SEQ:1 SEQ:0
*** TIME OUT! ReSend Message ***
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:1 Check Sum:57305
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:2 Check Sum:62106
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:3 Check Sum:24227
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:4 Check Sum:41288
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:5 Check Sum:51269
base: 1
nextseqnum 6
Send has been confirmed! Flag:2 STREAM_SEQ:1 SEQ:1
Send Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:6 Check Sum:14829
base: 2
nextseqnum 7
Send has been confirmed! Flag:2 STREAM_SEQ:1 SEQ:2
Send Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:7 Check Sum:17524
base: 3
nextseqnum 8
Send has been confirmed! Flag:2 STREAM_SEQ:1 SEQ:3

```

图 9: 超时重传

我们可以看到发送端重发了 SEQ 号从 1 到 5 的所有数据分组，并且在接下来的过程之中接收到了 ACK SEQ 为 1 的响应消息。

最终会显示文件传输成功：

```

9
Recv Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:448 Check Sum:45828
Send to Clinet ACK:2 STREAM_SEQ:1 SEQ:448
6
Recv Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:449 Check Sum:50918
Send to Clinet ACK:2 STREAM_SEQ:1 SEQ:449
1
Recv Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:450 Check Sum:41022
Send to Clinet ACK:2 STREAM_SEQ:1 SEQ:450
7
Recv Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:451 Check Sum:24630
Send to Clinet ACK:2 STREAM_SEQ:1 SEQ:451
7
Recv Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:452 Check Sum:16761
Send to Clinet ACK:2 STREAM_SEQ:1 SEQ:452
1
Recv Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:453 Check Sum:26207
Send to Clinet ACK:2 STREAM_SEQ:1 SEQ:453
9
Recv Message 1865 bytes! Flag:8 STREAM_SEQ:1 SEQ:454 Check Sum:12767
Send to Clinet ACK:2 STREAM_SEQ:1 SEQ:454
*** 文件大小: 1857353 bytes
*** 成功接收文件 ***
**** Log ****

```

图 10: 文件传输成功

在发送端我们也能看到此次传输的传输时间和吞吐率：

```

Send has been confirmed! Flag:2 STREAM_SEQ:1 SEQ:450
base: 451
nextseqnum 455
Send has been confirmed! Flag:2 STREAM_SEQ:1 SEQ:451
base: 452
nextseqnum 455
Send has been confirmed! Flag:2 STREAM_SEQ:1 SEQ:452
base: 453
nextseqnum 455
Send has been confirmed! Flag:2 STREAM_SEQ:1 SEQ:453
base: 454
nextseqnum 455
Send has been confirmed! Flag:2 STREAM_SEQ:1 SEQ:454
----- **对方已成功接收文件! **-----
**传输文件时间为: 16s
**吞吐率为:116085 bytes/s
-----*** 请输入想要发送的文件名 ***-----

```

图 11: 传输时间和吞吐率

可以看到当使用了 GBN 算法后，传输 1.jpg 仅仅使用了 16s，吞吐率为 116085 bytes/s，与

停等协议接近于 45000 bytes/s 的吞吐率相比，传输速率有了显著提升。

最后就是我们可以使用 quit 命令来结束发送文件，进行四次挥手断开连接：

```
-----*** 请输入想要发送的文件名 ***-----
quit
-----*** quit命令发送成功 ***-----
-----*** 完成第一次挥手 ***-----
-----*** 接收到第二次挥手消息，进行验证 ***-----
-----*** 完成第二次挥手 ***-----
-----*** 接收到第三次挥手消息，进行验证 ***-----
-----*** 完成第三次挥手 ***-----
-----*** 完成第四次挥手 ***-----
zzekun okk

Exiting...
请按任意键继续. . .
```

图 12: 四次挥手

结束整个的流程。

### 三、 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 算法以及 SR 选择重传算法。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。

这里给出的是个人的 Github 仓库链接：

[Computer Network](#)

## 参考文献

02-计算机网络-第二章-2022（第二部分）.pdf

03-计算机网络-第三章-2022-marked.pdf