南開大學
Nankai University

计算机学院

计算机网络实验报告

3-2: 基于 UDP 服务设计可靠传输协议并编程实现

姓名：李欣

学号：2011165

专业：计算机科学与技术

2022 年 12 月 3 日

目录

1 实验要求

作业要求：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。

- 多个序列号；
- 发送缓冲区、接受缓冲区；
- 滑动窗口：Go Back N；
- 有必要日志输出（须显示传输过程中发送端、接收端的窗口具体情况）。

2 实验环境

VS2019
语言:C++
转发：router 路由器

3 实验设计

3.1 设计说明

本次实验在 3-1 的基础上，将停等机制改成了基于滑动窗口的流量控制机制，滑动窗口选用 BGN，连接过程实现了三次握手、四次挥手；传输过程在发送端创建了两个线程：发送数据包和接收确认 ack，接收端的实现参考 GBN，始终等待期待的数据包。并参考 rdt3.0，实现超时重传和差错检验。

3.2 数据报格式

数据报格式在实验 3-1 中具体展示了，这里仅展示关键数据结构。
需要注意的是，基于 GBN 滑动窗口，我们这里的 ack 和 seq 不再时 0/1 而是变成了 [0,2k-1]
下面是数据头部的数据结构，在头部后存储的是数据部分。

```
1  struct header {
2      // u_short 2 字节 (2B) 对应 16 位 (16 bit) 无符号比特
3      u_short ack;            //[0,2^k-1]
4      u_short seq;            //[0,2^k-1]
5      u_short flag;            //bigend 0 位 SYN, 1 位 ACK, 2 位 FIN
6      u_short source_port;    //源端口
7      u_short dest_port;       //目的端口
8      u_short length;           //消息长度
9      u_short checksum;        //校验和
10
11     header()
12     {
```

```
13          source_port = SourcePort;
14          dest_port = DestPort;
15          ack = seq = flag = length = checksum = 0;
16      }
17  };
```

3.3 GBN 滑动窗口

图 3.1: 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, 2k-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

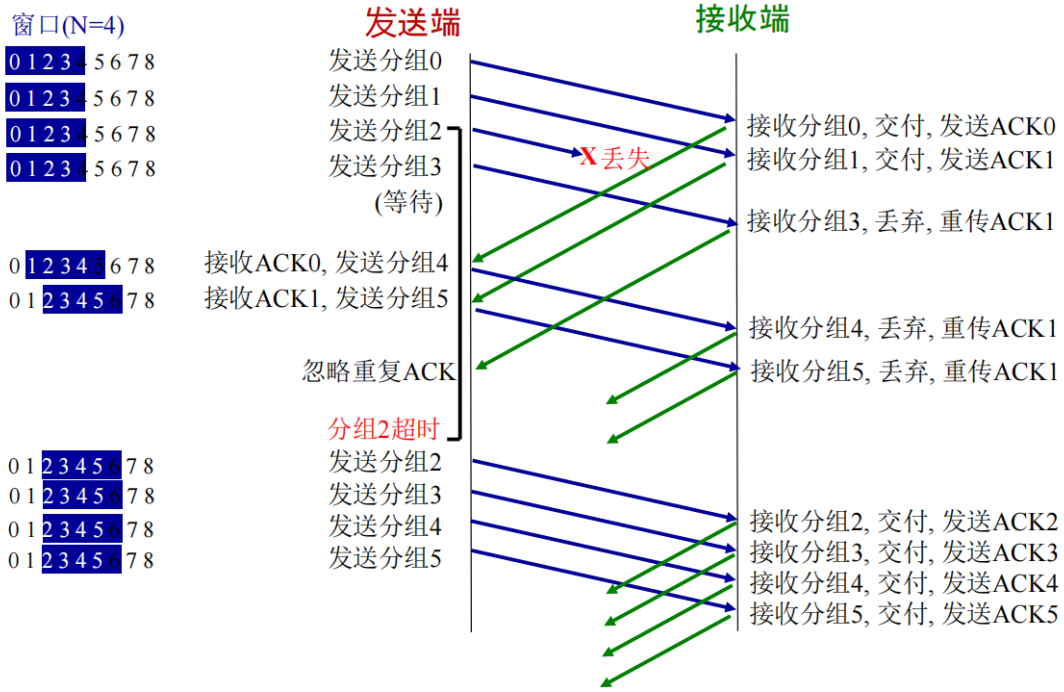

图 3.2: GBN

4 实验流程

4.1 三次握手

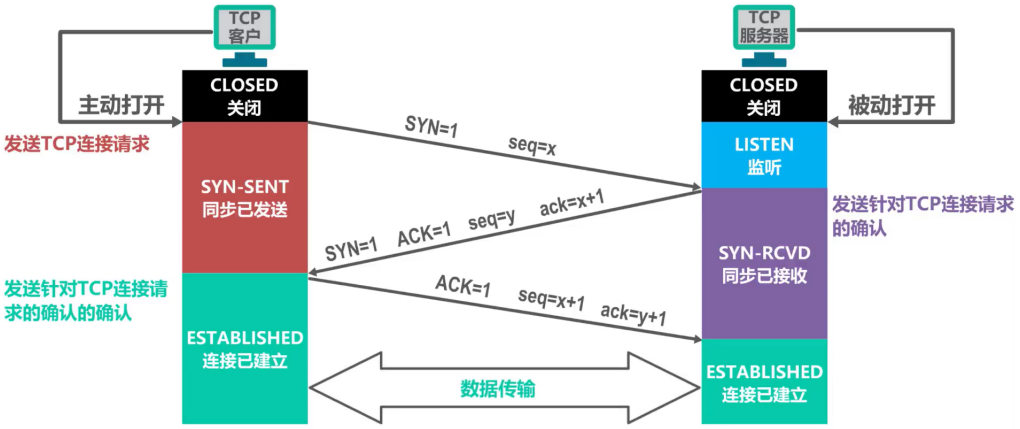

图 4.3: 发送端

(1) 首先，由发送端将头部的 flag 位设置位 SYN，即，请求接收端与之建立连接。、

若这期间，发送端超过最大等待时长没有收到接收端的 (SYN,ACK) 回复，则再次发送 SYN 连接请求。若连续发送 SYN 请求超过 2 秒没有收到 (SYN,ACK) 回复，则说明服务器并未打开或者出现了故障，此时不再发送 SYN 连接请求，而是主动关闭程序。

此时，接收端处于阻塞通信状态，若收到 SYN 请求，则回复 (SYN,ACK) 数据包，即第二次握手请求，若没有收到信息或校验和出现错误，则继续进入等待状态。

(2) 若发送端成功收到 (SYN,ACK) 回复，则说明第一次和第二次发送握手请求都成功了。

(3) 此时发送端发送第三次握手请求 ACK，此处有两种方法让发送端确认接收端成功接收到了第三次握手 ACK。

第一种方法是等待一定时长没有再次收到 (SYN,ACK) 数据包，即，接收端在等待时长内收到了 ACK，故不会认为 (SYN,ACK) 丢失，所以不用重发 (SYN,ACK) 数据包，因此，发送端不会收到任何数据包。

第二种方法是基于我们的实验假定，接收端到发送端发送的包是不会丢失的，所以，只要在接收端收到第三次握手 (ACK) 时，再向发送端发一个确认收到的包就可以了。

这两种方法看似都可行，但在实际运用中，并不能保证接收端到发送端的传输一定是可靠的，故选取了第一种方式确定第三次握手 ACK 发送成功

三次握手在 3-1 中已经给出详细的理论、程序设计与代码实现说明，这里不再做赘述。这里仅展示部分代码。

```
1
2       result = sendto(s, sendbuff, header_size, 0, (sockaddr*)&router_addr, routerlen);
3       if (result == -1)
4       {
5           cout << "[握手请求]:   [1] 发送数据包 失败" << endl;
6           exit(-1);//退出程序
7       }
```

```
8       else
9           cout << "[握手请求]:   [1] 发送数据包 成功" << endl;
```

4.2 发送端

在发送端，我们创建了两个线程，一个用于发送数据包，一个用于接收 ack。另外，我们需要一些全局变量来构建两个线程之间的联系。

GBN 发送端拓展 FSM 如下图所示。发送端主要需要两个线程，一个用于发送数据包，一个用于接收 ack。由于我们在这里采取的是 GBN，所以在超时重传的时机判断上也会略有不同，接收 ack 和发送 seq 的具体思路与代码实现会在后面每个线程中说明。

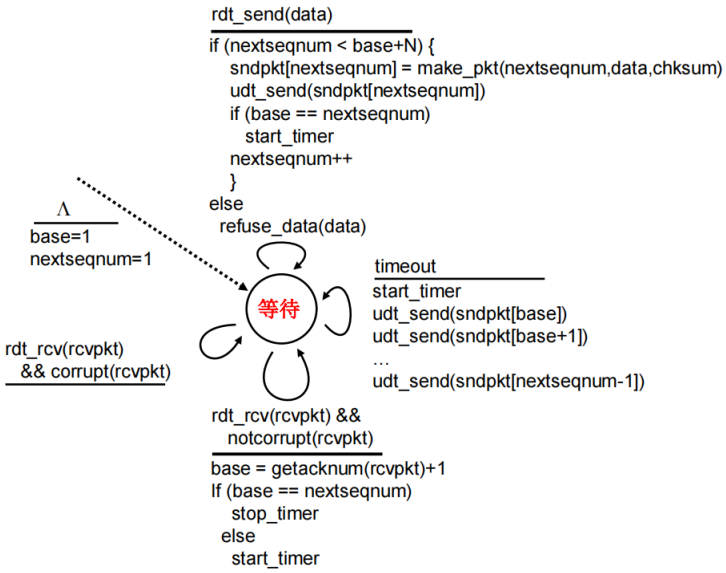

图 4.4: 发送端

4.2.1 全局变量

首先，我们设定了一些全局变量，有滑动窗口的大小，总的需要发送的数据包个数，也就是总的 seq 号，当前需要发送的 seq，发送缓冲区，接收缓冲区，用于记录每个包发送时间的计时数组。

全局变量

```
1   const int window_len = 4;//滑动窗口的大小 为4
2   int seq_total = 0;// 总的seq号
3   int current_seq = 0;//当前需要发送的seq
4   int current_hope_ack = 0;//当前期待收到的ack
5   char* sendbuff_data;//发送的包,包含了数据
6   char* recbuff_data;//收到的包,只需要接收数据头
7   double recordtime[window_len] = { 0,0,0,0 };//每个包发送时的时间
8   unsigned long long int packnum = 0;//总共要发送的数据包个数
```

4.2.2 线程一：发送数据

在发送数据线程中，我们要判断能不能在最大等待时间内接收到期待的 ack 回复，如果我们在最大等待时间内不能收到期待的 ack，那么就判定此 ack 对应的 seq 数据包分组丢失了，按照 GBN 的规则，重传所有未确认的分组。

判断超时

```
//判断超时: 当前时间 - 期待收到的ack对应的seq数据包发送的时间
if (clock() - recordtime[current_hope_ack % window_len] > max_waitingtime)
{
  current_seq = current_hope_ack;//丢失的包开始发
  cout << "[ACK超时]:       超时未收到期待的ack [" << current_hope_ack << "]
      重新传输" << endl;
}
```

我们要判断当前窗口是否有剩余的窗口来发送分组，如果剩余窗口大小 >=1，且要发送的 seq 分组号小于总的 seq 分组号，则说明可以发送下一个分组。在发送分组时，需要将发送分组的时间记录下来，用于超时判断。

发送数据包

```
// 如果当前窗口有空闲 且 发送的seq不超过总的数据包数量
if ((current_seq - (current_hope_ack - 1) <= window_len) & (current_seq <=
    packnum))
{
  result = sendto(s, sendbuff_data, header_size + thislen, 0,
      (sockaddr*)&router_addr, routerlen);//发送当前数据包
  recordtime[current_seq % window_len] = clock();//记录seq对应的发送时间
  current_seq++;//更新下一个要发送的seq号
}
```

4.2.3 线程二：接收 ack

另一个线程就是用于接收接收端发送来的确认收到分组的 ack，若收到的 ack 是我们期待收到的 ack，则确认分组，并将窗口向后滑动，以便发送下一个分组。若收到的不是期待的 ack，则继续等待 ack。

接收 ack

```
//接收ack
if (recvfrom(s, recbuff_data, header_size, 0, (sockaddr*)&router_addr,
    &routerlen) > 0)
{
  //收到回复，判断回复的对不对
  memcpy(&e_header, recbuff_data, header_size);
  //判断是否为期待接收的ack
  if (e_header.ack == current_hope_ack && checksum((u_short*)&e_header,
      header_size) == 0)
  {
```

```
        cout << "\n[期待ACK]        " << current_hope_ack << endl;
        cout << "[接收ACK]:       " << e_header.ack << endl;
        //期待接收下一个数据包的ack
        current_hope_ack++;
      }
      else
        continue;
    }
```

4.3 接收端：接收数据并返回 ack

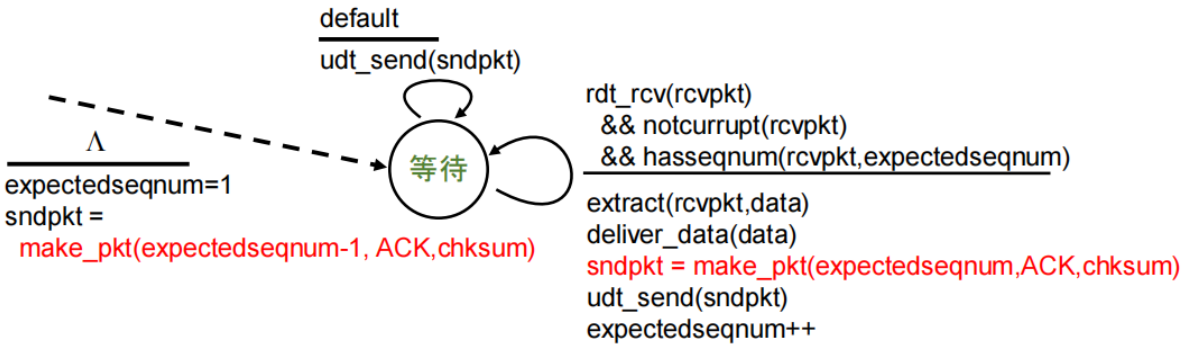

图 4.5: 接收端

接收端的实现方式较为简单，就是等待我们期望的分组。如果我们收到的是期望的 seq 分组，则回复 ack=seq 的数据包。如果我们收到的不是期望的 seq 分组，由于 GBN 只确认连续正确接收分组的最大序列号，则回复累计确认的最大序列号。然后继续等待期望的分组。

判断数据包并返回 ack

```
        //接收到了期待的seq
    if (e_header.seq == hope_seq && checksum((u_short*)recbuff, header_size +
        e_header.length) == 0)
    {
      //将接收到的数据区域存入总的存储数组message中
      memcpy(message + offset, recbuff + header_size, e_header.length);
      offset += e_header.length;
      sendto(s, sendbuff, header_size, 0, (sockaddr*)&router_addr, routerlen);
      //更新hope_seq和ack
      //debug!!!
      send_ack = hope_seq;//新收到了seq，更新累计确认ack
      hope_seq = hope_seq + 1;//下一次期待的seq
    }
    //不正确，返回的是累计确认的ack，并重新等待
    else
    {
      cout << "[接收数据]:   接收 ["<<e_header.seq<<"] 的数据包 但期待的是 [" <<
          hope_seq <<"]"<< endl;
```

```
17        sendto(s, sendbuff, header_size, 0, (sockaddr*)&router_addr, routerlen);
18        continue;
19    }
```

4.4 四次挥手

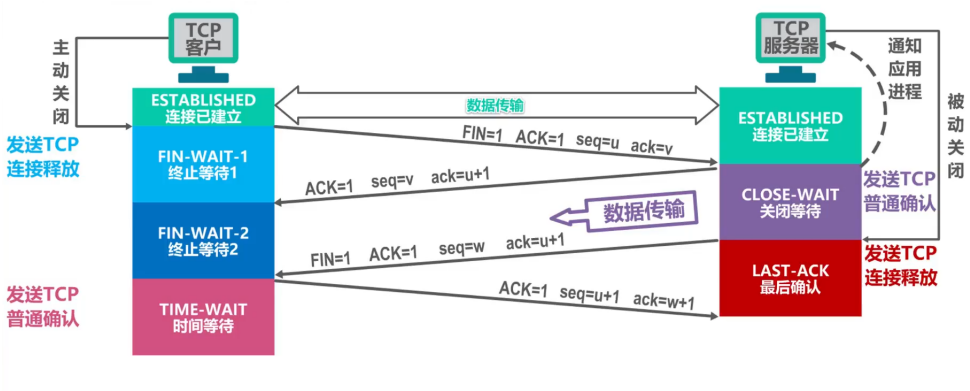

图 4.6: 发送端

在四次挥手中，本实验假定的是由发送端最开始请求关闭连接。

(1) 首先，由发送端发送 FIN 数据包请求关闭连接。若超过最长等待时间没有收到接收端送来的 (FIN,ACK), 则说明 FIN 数据包丢失了，那么再发送，再次等待，知道收到 (FIN,ACK) 数据包。

(2) 若收到接收端发来的 (FIN,ACK), 则继续等待下一个 (FIN,ACK) (这里为什么需要两个 (FIN,ACK), 我们在前面的实验中其实已经涉及过了，有时候虽然单向的连接关闭，但反向的数据传输可能还没有结束等原因，在此不做赘述。)

(3) 接收到第二个 (FIN,ACK)

(4) 继续向接收端发送一个确认 ACK，由于发送端到接收端可能存在丢包的情况，所以这里和握手请求的最后一个 ACK 类似，要么等待一定时间，要么由可靠的接收端向服务器发送最后一个 FINAL_CHECK，这里采取了后者.

(5) 发送端接收到了 FINAL_CHEAK，此时就可以安心的关闭了。

四次握手在 3-1 中已经给出详细的理论、程序设计与代码实现说明，这里仅展示部分代码。

```
1  const u_short FIN = 0x4;//OVER=0,FIN=1,ACK=0,SYN=0
2  const u_short FIN_ACK = 0x6;//OVER=0,FIN=1,ACK=1,SYN=0
3  const u_short FINAL_CHECK = 0x20;//FC=1,OVER=0,FIN=0,ACK=0,SYN=0
```

5 实验结果

5.1 三次握手

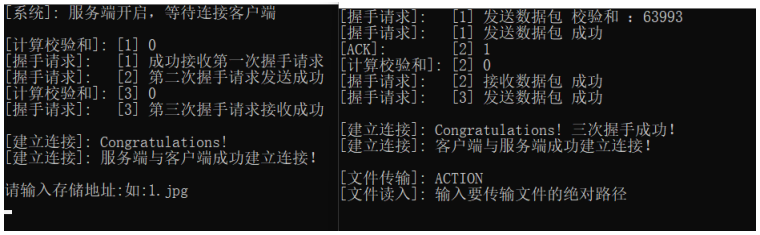

图 5.7: 三次握手

日志会打印输出三次握手的建联过程，如上图所示。

5.2 传输过程

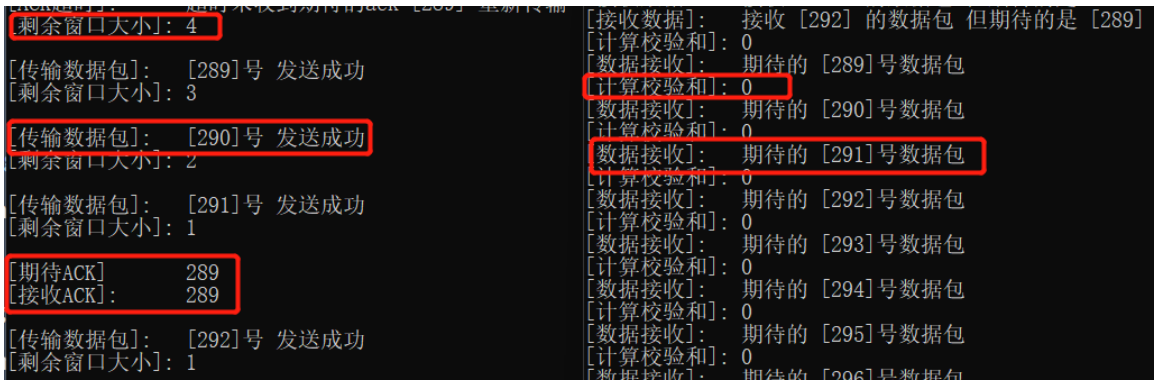

图 5.8: 传输过程

传输过程中，发送线程会打印传输数据包的序列号与剩余窗口的大小，也会打印接收 ack 的值和期待收到的 ack 的值，以便观察 ack 是否匹配。接收线程会打印期待接收到的分组序列号和计算的校验和。

5.3 超时重传

图 5.9: 超时重传

图 5.10: 判断超时后重新发送丢失包后的所有包

查看传输过程，以 1799 为例分析丢包与超时重传过程。

首先看发送端，在连续三次收到了重复的 ACK：1798 后，程序根据时间的计算，日志给出了超时的判断，于是判定 1799 包丢失了，所以重新传输 1799 以及之后的所有包。

再看接收端，由于没有收到期待的 1799 数据包，虽然收到了 1800，1801，1802 三个数据包，忽略了这三个包，返回累计确认的 ACK=1798 并且继续等待。由发送端判定超时后发送丢失的包，成狗接收到了期待的 1799 数据包。

5.4 总体信息打印

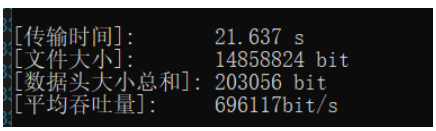

图 5.11: 总体信息打印

打印了传输时间，文件大小，数据头总大小与平均吞吐率等总体信息。

5.5 四次挥手

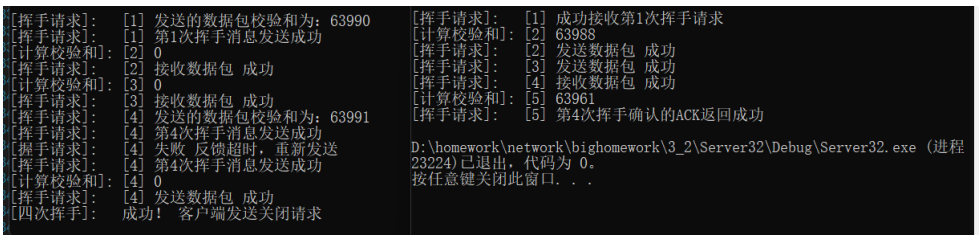

图 5.12: 四次挥手

日志会打印输出四次挥手的断联过程，如上图所示。

6 遇到的问题与解决

6.1 多线程未成功关闭的问题

问题:

在接收到的 ack 为 OVER_ACK 时，即收到最后一个数据包返回的 ack 后，使用 closeHandle 线程并没有关闭线程，仍然接收处理了四次挥手接收端传来的第二次挥手请求。即在结果运行中的“文件传输完毕!”后就已经“关闭”了线程，但是很显然，**[重复 ACK]: [0] 号重复 ACK**，这条打印日志的输出说明线程并没有关闭且收到了四次挥手接收端发来的第二个挥手请求。

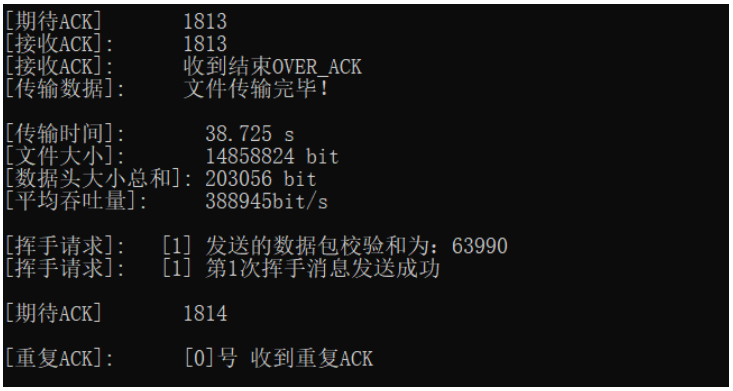

图 6.13: 多线程出错

解决方法:

在接收线程接收到 OVER_ACK 时，使用 break 跳出循环，线程返回 0（return 0;）可以看到显示了正确的结果。

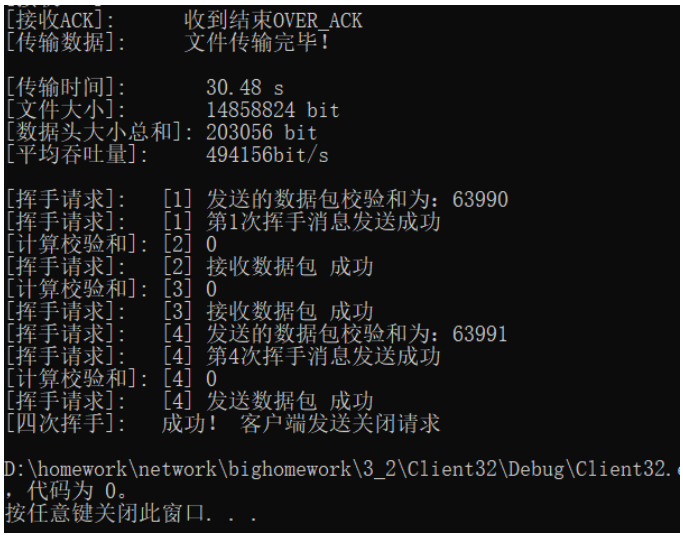

图 6.14: 修正多线程

原因:

closeHandel(ThreadHandle); 只是关闭了一个线程句柄对象，表示我不再使用该句柄，即不对这个句柄对应的线程做任何干预了。并**没有结束**线程。

6.2 四次挥手客户端第一次挥手超时未收到反馈

问题:

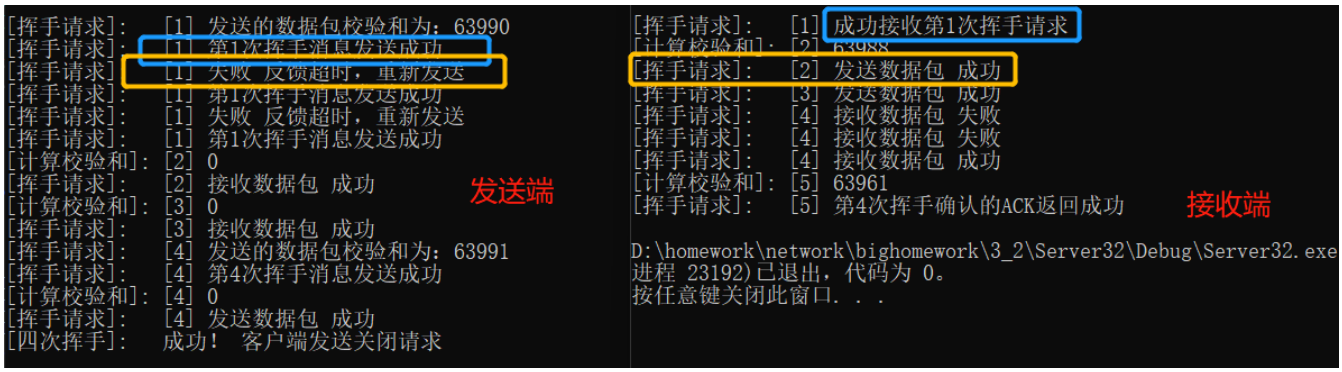

图 6.15: 可靠的接收端发送了第二次挥手请求，但发送端却没有收到

可以看到，发送端发送了第一次挥手请求，接收端也确实收到了第一次挥手请求，并且可靠的接收端发送了第二次挥手请求，但接收端却没有收到，打印日志显示超时未收到反馈。

解决方法:

让发送端在等待接收端发来的第二次挥手请求的时候，将最大等待时间设置的大一些。

原先判断超时的时间:0.2s

```
1         if ((end - start) > 2 * max_waitingtime)
2           over_time_flag = true; //超时设置flag为1
```

修正后判断超时的时间: 1s

```
1            if ((end - start) > 10 * max_waitingtime)
2              over_time_flag = true; //超时设置flag为1
```

修正结果展示

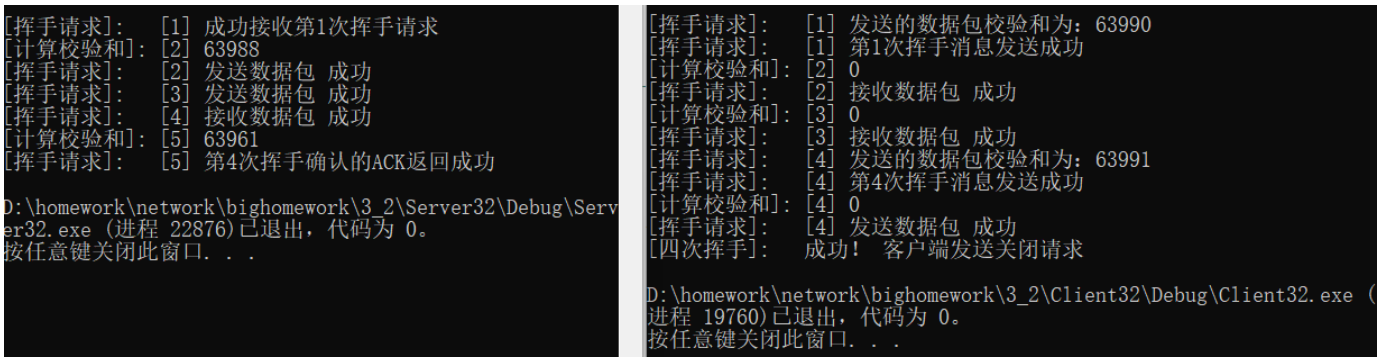

图 6.16: 成功解决超时等待判断错误问题

6.3 计时问题

问题:

原先我的设想是，用一个变量记录开始时间，滑动窗口每滑动一次应该重新计时，但是按照 GBN 的设置，判断超时应该从发送数据包开始的，这时就会出现一个问题，在开始时，剩余窗口为窗口大小 windows_len，如果我只记录第一个数据包发送的时间的话，后面几个发送的数据包没办法记录发送时的时间。

解决方法:

设置了一个和窗口大小一样大的 double 数组，用于记录每个数据包发送时的时间，并在需要判断超时时，用当前时间减去期待收到 ack 对应 seq 数据包的发送时间（而发送的时间正好记录在了 recordtime 数组中），另，为什么只用开窗口大小呢，因为根据 GBN 的规则设置，只有收到 seq 的 ack，才能发送 seq+window_len 的数据，所以不会存在取模冲突的情况。

7 总结

在实验 3-2 中，我在 3-1 rdt3.0+ 停等机制的基础上，对基于滑动窗口的流量控制机制，采用固定窗口大小，支持累积确认的传输方式进行了编程实现，对 GBN 滑动窗口，累计确认，超时重传，差错检验等有了更深的理解，并且能够正确实现期望的功能。