南開大學
Nankai University

计算机学院

计算机网络实验报告

3-3: 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 29 日

目录

1 实验要求 **2**

2 实验环境 **2**

3 实验设计 **2**
3.1 设计说明 . . . 2
3.2 数据报格式 . . . 3
3.3 RENO 算法 . . . 3
3.3.1 慢启动 . . . 4
3.3.2 拥塞避免 . . . 4
3.3.3 快速恢复 . . . 4

4 实验流程 **4**
4.1 三次握手 . . . 4
4.2 发送端 . . . 5
4.2.1 全局变量 . . . 5
4.2.2 线程一：计时函数 . . . 6
4.2.3 线程二：接收 ack . . . 6
4.2.4 线程三：发送数据 . . . 7
4.3 接收端: 接收数据并返回 ack . . . 8
4.4 四次挥手 . . . 9

5 实验结果 **10**
5.1 三次握手 . . . 10
5.2 传输过程 . . . 10
5.2.1 超时状态切换 . . . 10
5.2.2 NEW ACK 状态切换 . . . 10
5.2.3 重复 ACK 状态切换 . . . 12
5.3 总体信息打印 . . . 13
5.4 四次挥手 . . . 13
5.5 传输结果 . . . 13

6 遇到的问题与解决 **14**
6.1 变量大小比较问题 . . . 14
6.2 多线程打印混乱 . . . 14
6.3 加锁解决多线程变量冲突 . . . 14

1 实验要求

作业要求：在实验 3-2 的基础上，选择实现一种拥塞控制算法，也可以是改进的算法，完成给定测试文件的传输。

- RENO 算法；
- 也可以自行设计协议或实现其他拥塞控制算法；
- 给出实现的拥塞控制算法的原理说明；
- 有必要日志输出（须显示窗口大小改变情况）。

2 实验环境

VS2019
语言:C++
转发：router 路由器

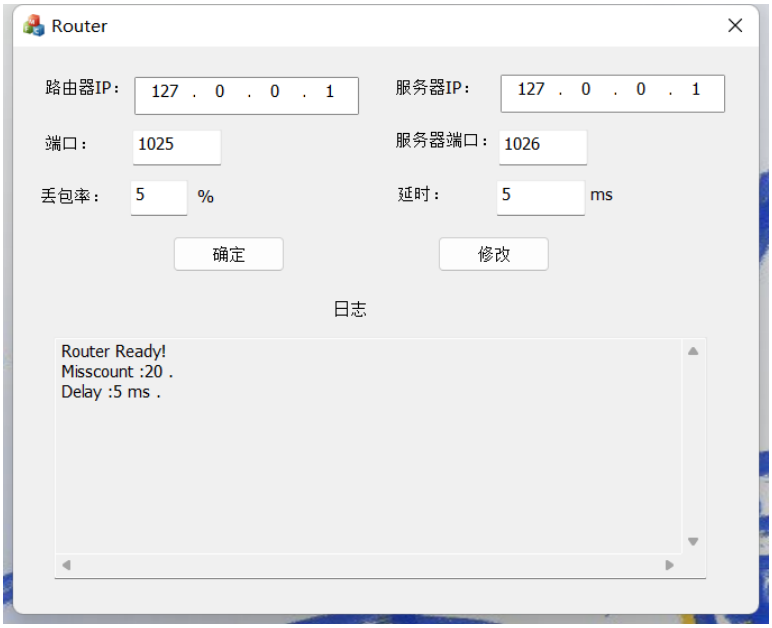

图 2.1: router

3 实验设计

3.1 设计说明

本次实验在 3-2，即基于滑动窗口的流量控制机制的基础上，实现了 RENO 拥塞控制算法，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多个线程：超时判断、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。
下面是数据头部的数据结构，在头部后存储的是数据部分。

```
struct header {
    u_short ack;              //[0,2^k-1]
    u_short seq;              //[0,2^k-1]
    u_short flag;              //bigend 0 位 SYN, 1 位 ACK, 2 位 FIN
    u_short source_port;     //源端口
    u_short dest_port;         //目的端口
    u_short length;             //消息长度
    u_short checksum;         //校验和
    int start;           //起始位置
    header()
    {
        start = 0;
        source_port = SourcePort;
        dest_port = DestPort;
        ack = seq = flag = length = checksum = 0;
    }
};
```

3.3 RENO 算法

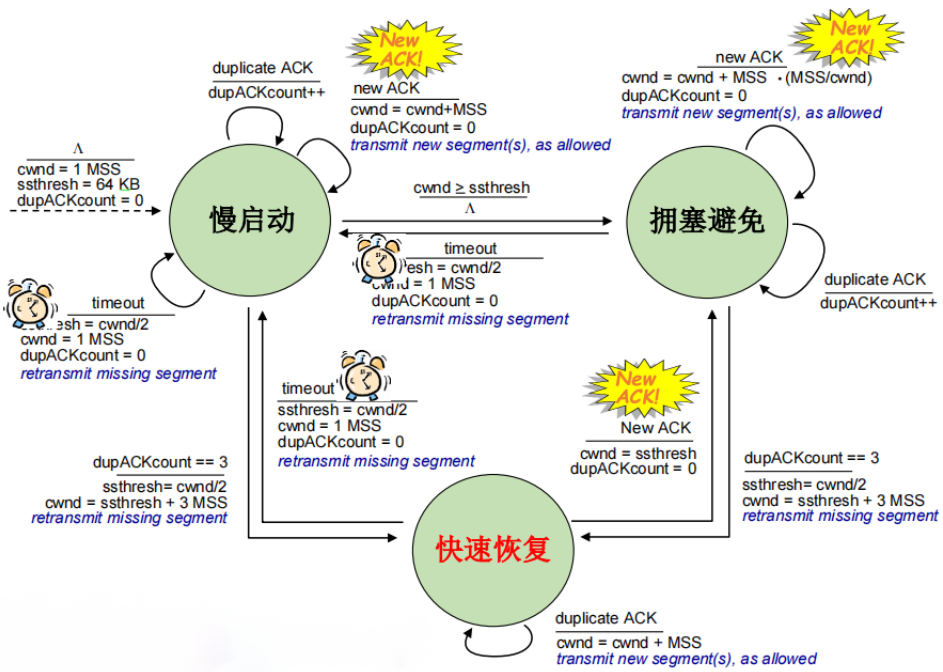

图 3.2: RENO 算法状态机

RENO 算法发送端状态机如上图所示。

3.3.1 慢启动

- 初始状态：设置初值 cwnd=1MSS, 阈值 ssthresh=64KB，重复 ACK 计数 dupACKcount=0;
- 超时 ssthresh=cwnd/2,cwnd=1MSS, dupACKcount=0, 状态**仍处于慢启动**;
- 重复 ACK：在接收到三次重复 ACK 后，ssthresh = cwnd/2, cwnd = ssthresh + 3 MSS, **状态转换为快速恢复**;
- NEW ACK:在接收到新的 ACK 后,cwnd = cwnd + MSS, dupACKcount = 0, 如果 cwnd>ssthresh, 那么**状态转换为拥塞避免**。

3.3.2 拥塞避免

- 超时 ssthresh=cwnd/2,cwnd=1MSS, dupACKcount=0, **状态转换为慢启动**;
- 重复 ACK：在接收到三次重复 ACK 后，ssthresh = cwnd/2, cwnd = ssthresh + 3 MSS, **状态转换为快速恢复**;
- NEW ACK：在接收到新的 ACK 后，cwnd 线性增长，cwnd = cwnd + MSS · (MSS/cwnd), dupACKcount = 0, 状态**仍处于拥塞避免**。

3.3.3 快速恢复

- 超时 ssthresh=cwnd/2,cwnd=1MSS, dupACKcount=0, **状态转换为慢启动**;
- 重复 ACK：cwnd = ssthresh + MSS, 状态**仍处于快速恢复**;
- NEW ACK：在接收到新的 ACK 后，cwnd = ssthresh, dupACKcount = 0, **状态转换为拥塞避免**。

4 实验流程

4.1 三次握手

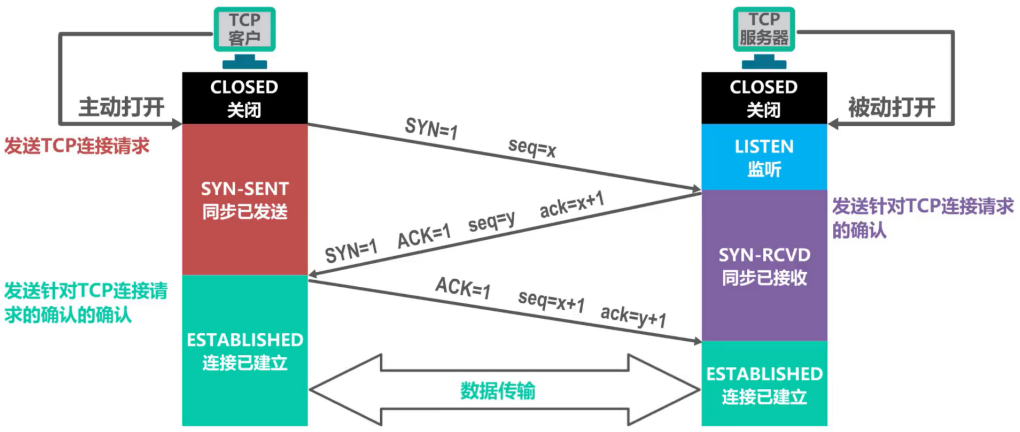

图 4.3: 发送端

(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。

```

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

4.2 发送端

4.2.1 全局变量

下面列出了本次实验中比较重要的全局变量，也是线程间构建联系的重要桥梁，cwnd 为窗口大小，cwnd 会随着超时、接收新 ACK，重复 ACK 等状况发生相应的变化，ssthresh 为与之大小，LastByteAcked 为接收端已经确认接收到的最后一个字节，LastByteSent 为发送端最近发送的最后一个字节，dupACKcount 为接收重复 ACK 的次数，state 为当前状态，状态又分为慢启动，拥塞避免和快速恢复三个状态，分别对应 slow_start,avoid_congestion,quick_recovery。

全局变量

```
long long int cwnd = MSS;//初始时化cwnd大小
long long int ssthresh = 6*MSS;//阈值大小
long long int LastByteAcked = -1;
```

```
4     long long int LastByteSent = -1;
5     int dupACKcount = 0;//接收重复ACK的次数
6     enum {slow_start,avoid_congestion,quick_recovery};//三个状态
7     int state = slow_start;//当前状态
```

4.2.2 线程一：计时函数

RENO 算法，发送端对于发送的包进行超时的判断，若超时，阈值减半，cwnd 设置为 MSS，重复 ACK 计数 dupACKcount 设置为 0，无论处于何种状态，状态均切换为慢启动，还需要将 LastByteSent 设置为 LastByteAcked，当前发送的数据包 seq 号设置为期待接收到的 ack。

超时状态切换

```
1       ssthresh = cwnd / 2;
2       cwnd = MSS;
3       dupACKcount = 0;
4       if (state == slow_start)
5       {
6         ::cout << "\n[超时]          仍处于慢启动" << endl;
7       }
8       // 拥塞避免状态超时
9       else if (state == avoid_congestion)
10      {
11        state = slow_start;
12        ::cout << "\n[超时状态切换] 拥塞避免->慢启动" << endl;
13      }
14      // 快速恢复状态超时
15      else if (state == quick_recovery)
16      {
17        state = slow_start;
18        ::cout << "\n[超时状态切换] 快速恢复->慢启动" << endl;
19      }
20      LastByteSent = LastByteAcked;//丢失的字节开始发
21      current_seq = current_hope_ack;
```

4.2.3 线程二：接收 ack

接收新的 ACK，我们在 3.3RENO 算法状态机中已经给出了具体的分析，在此不做赘述，需要注意的是编程实现中 cwnd = cwnd + MSS * (MSS*1.0 / cwnd) 的 1.0，否则会按照 int 整型计算，无法做到线性增长。

接收新的 ACK

```
1       if (state == slow_start)
2       {
3         cwnd = cwnd + MSS;
4         dupACKcount = 0;
5         if (cwnd >= ssthresh)
```

```
        {
          state = avoid_congestion;
        }

      }
      else if (state == avoid_congestion)
      {
        cwnd = cwnd + MSS * (MSS*1.0 / cwnd);
        dupACKcount = 0;
      }
      else if (state == quick_recovery)
      {
        cwnd = ssthresh;
        dupACKcount = 0;
        state = avoid_congestion;
      }
```

接收重复 ACK，我们在 3.3RENO 算法状态机中已经给出了具体的分析，在此不做赘述。

接收重复 ACK

```
      if (state == slow_start)
      {
        dupACKcount++;
        if (dupACKcount == 3)
        {
          ssthresh = cwnd / 2;
          cwnd = ssthresh + 3 * MSS;
          state = quick_recovery;
        }
      }
      else if (state == avoid_congestion)
      {
        dupACKcount++;
        if (dupACKcount == 3)
        {
          ssthresh = cwnd / 2;
          cwnd = ssthresh + 3 * MSS;
          state = quick_recovery;
        }
      }
      else if (state == quick_recovery)
      {
        cwnd = cwnd + MSS;
      }
```

4.2.4 线程三：发送数据

发送数据与前一次的改动不是特别大，在此只给出部分代码。

发送数据包

```
e_header.start = LastByteSent + 1;
::memcpy(sendbuff_data + header_size, message + LastByteSent+1, thislen);
e_header.checksum = 0;
::memcpy(sendbuff_data, &e_header, header_size);
//计算校验和
e_header.checksum = checksum((u_short*)sendbuff_data, header_size + thislen);
//重新填充头部
::memcpy(sendbuff_data, &e_header, header_size);
result = sendto(s, sendbuff_data, header_size + thislen, 0,
    (sockaddr*)&router_addr, routerlen);
```

4.3 接收端: 接收数据并返回 ack

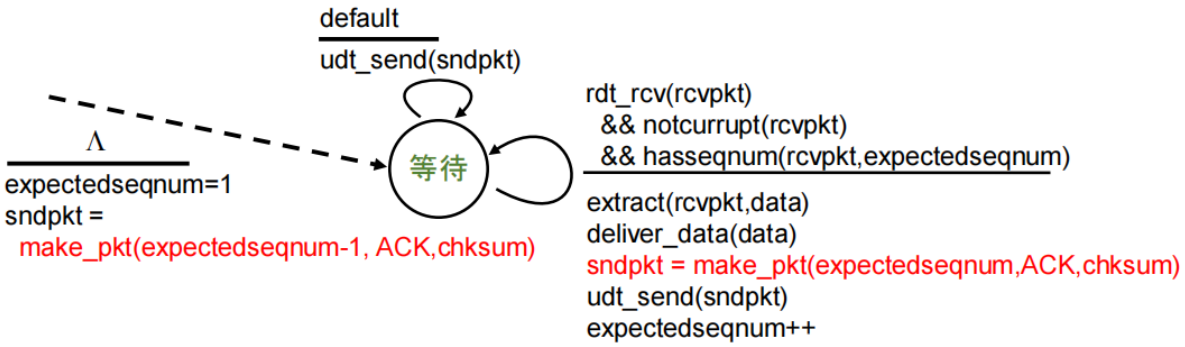

图 4.4: 接收端

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
```

```
14      else
15      {
16        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

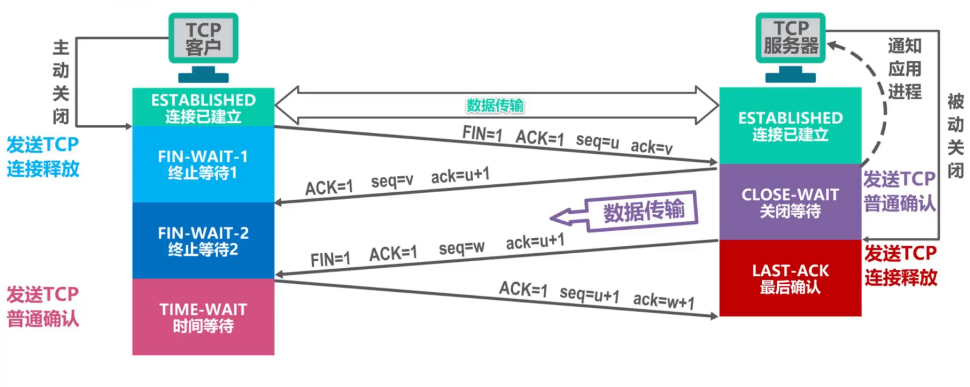

图 4.5: 四次挥手

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

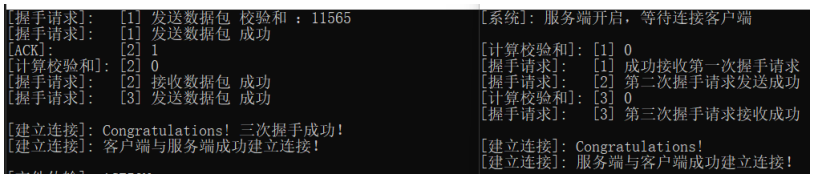

图 5.6: 三次握手

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

5.2.1 超时状态切换

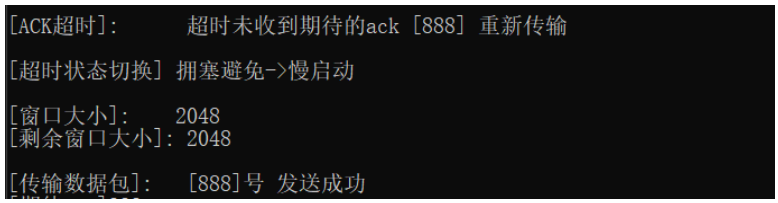

图 5.7: 拥塞避免-> 慢启动

在拥塞避免状态判定数据包超时，状态转换为慢启动，可以看到窗口大小变为了 MSS(2048)

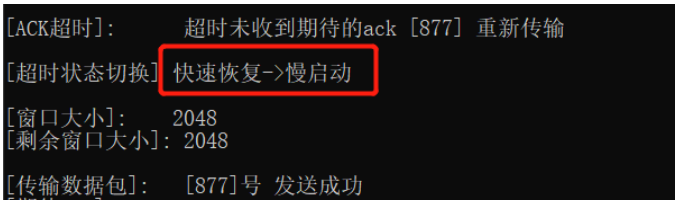

图 5.8: 快速恢复-> 慢启动

在快速恢复状态判定数据包超时，状态转换为慢启动，可以看到窗口大小变为了 MSS(2048)

5.2.2 NEW ACK 状态切换

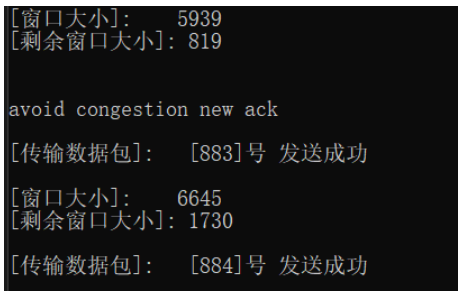

图 5.9: 拥塞避免-> 拥塞避免

在拥塞避免状态接收到了新的 ACK，窗口大小线性增长，cwnd = cwnd+MSS*(MSS/cwnd)，即新窗口的大小为 5939+2048*(2048*5939)=6645, 实验结果与计算结果相匹配。

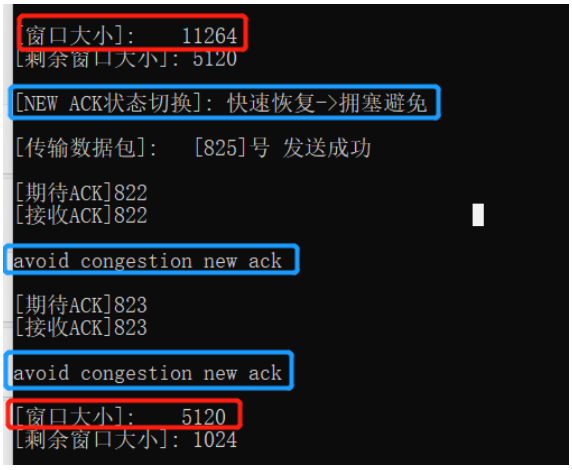

图 5.10: 快速恢复-> 拥塞避免

在快速恢复状态接收到了新的 ACK，cwnd = ssthresh，即新窗口的大小为阈值大小, 由于图片所示结果中还在拥塞避免阶段接收到了两个新的 ACK，故 cwnd 变为阈值后，又经过了两次线性增长，实验结果与计算结果相匹配。

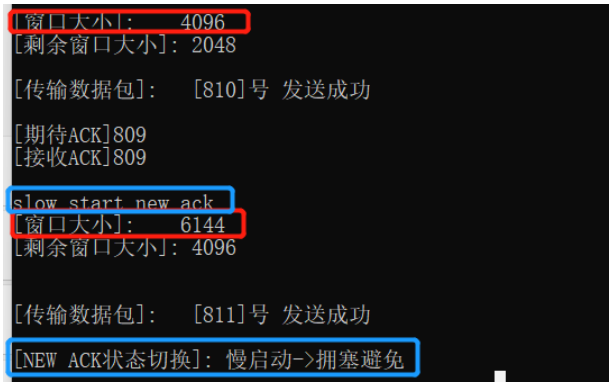

图 5.11: 慢启动-> 拥塞避免

慢启动接收到新的 ACK，cwnd=cwnd+MSS=4096+2048=6144，计算结果与实验结果相符合，且此时 cwnd 大于阈值，状态由慢启动切换到了拥塞避免。

5.2.3 重复 ACK 状态切换

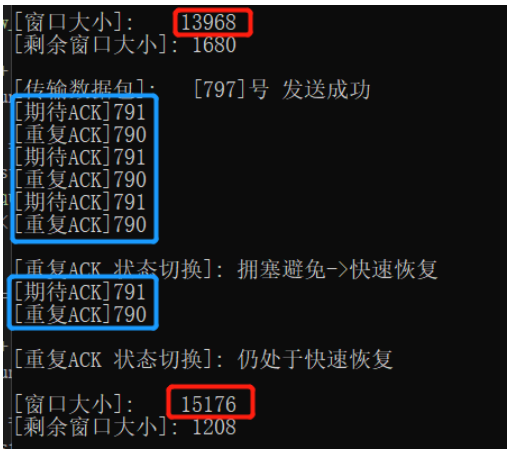

图 5.12: 拥塞避免-> 快速恢复

拥塞避免状态接收到三个重复的 ACK，阈值 ssthresh 减半，cwnd=cwnd+3*ssthresh，状态转换为快速恢复。

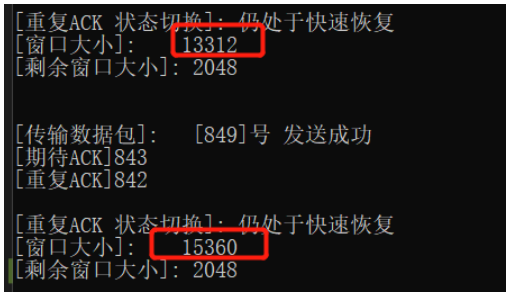

图 5.13: 快速恢复-> 快速恢复

快速恢复状态每接收到一个重复的 ACK，cwnd=cwnd+MSS=13312+2048=15360，状态保持不变。

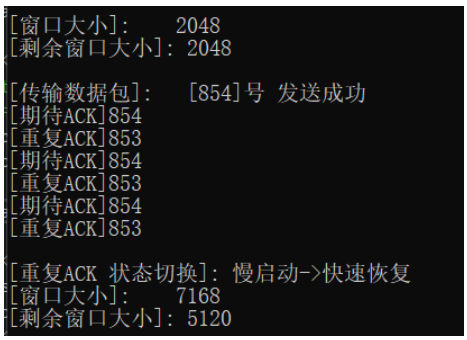

图 5.14: 慢启动-> 快速恢复

慢启动状态接收到三个重复的 ACK，阈值 ssthresh 减半，cwnd=cwnd+3*ssthresh，状态转换为快速恢复。

5.3 总体信息打印

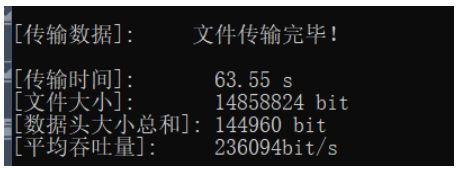

图 5.15: 总体信息打印

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。

5.4 四次挥手

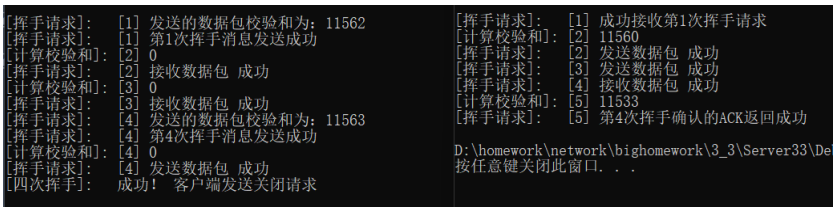

图 5.16: 四次挥手

日志会打印输出四次挥手的断联过程，如上图所示。

5.5 传输结果

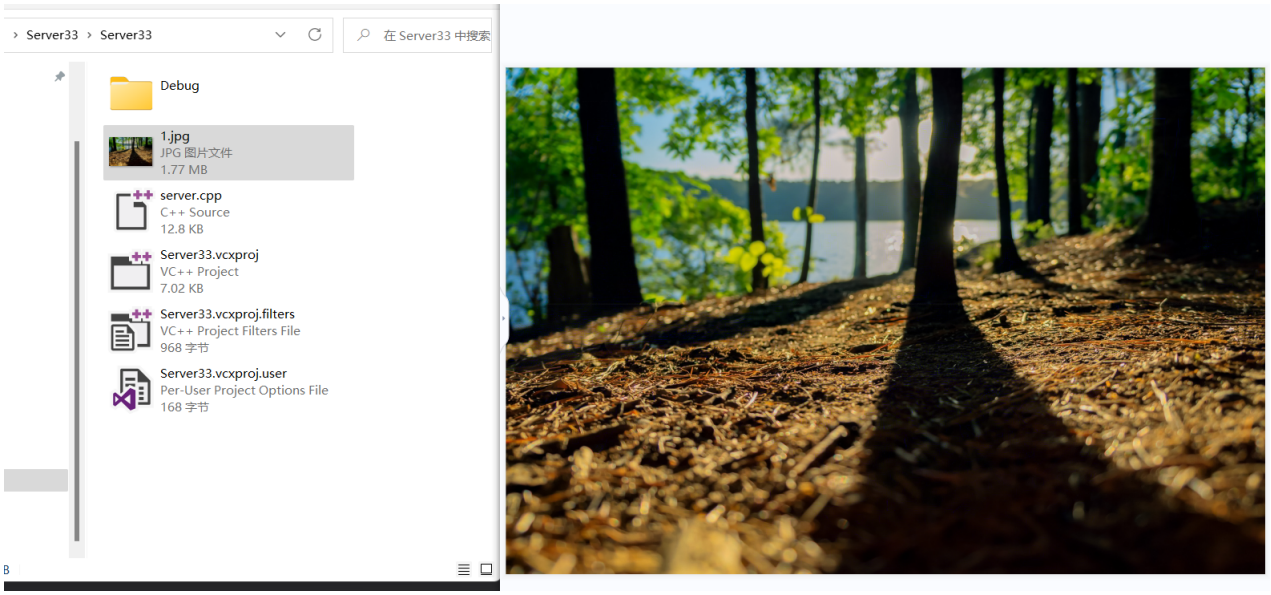

图 5.17: 四次挥手

可以看到，图片被完整的传输了。

6 遇到的问题与解决

6.1 变量大小比较问题

问题

在发送消息的时候，需要比较 LastByteSent 与整个文件的大小，由于 LastByteSent 的初值为-1，故 LastByteSent 不能设置为 unsigned long long 类型，而我将文件的总字节数设置为无符号长整型，经过答应发现，-1 与 unsigned long long 比较的结果为 false，导致发送线程无法进入循环发送数据。

解决方法

将 LastByteSent 和记录文件总字节数的变量 whole_len 都设置为 long long int 类型即可。debug 了很久，虽然这个问题就实验本身而言并不是什么理解性的错误，但我明白了要想编出可靠准确的程序，要夯实 C++ 语言的基础学习。

6.2 多线程打印混乱

问题:

由于开启了多线程，所以各个线程的打印没有依赖关系，如果出现 cout«"aaa"« 变量 «endl; 的情况，就会出现一个线程的句子打印到一半，另一个线程就插进来了。

解决方法:

字符串的打印是原子的操作，而出现问题的 cout 可以看出并不是原子操作，C++ 支持 to_string 将整型/浮点型转换为字符型。我们还可以通过加锁的方式来保证 IO 操作的原子性。

解决多线程打印混乱

```
1        in_re_ack = "\n[期待ACK]" + to_string(current_hope_ack);
2        ::cout << in_re_ack << endl;
```

6.3 加锁解决多线程变量冲突

问题:

当**计时线程**判定当前期待的数据包超时时，就会把 LastByteSent 更新为 LastByteAcked，但与此同时**接收线程**接收到了期待接收的 ACK，此时 LastByteAcked>LastByteSent。在**发送数据包线程**，是根据 LastByteSent 来发送数据的，如果出现以下情况：LastByteSent<LastByteAcked，窗口没有剩余，超时时窗口大小回到 MSS，线程将 LastByteSent 更新为 LastByteAcked，注意，因为 LastByteSent<LastByteAcked，所以剩余窗口没有剩余可以发送了。

解决方法:

我们可以通过对变量加锁来保证原子操作，以解决这个问题。这里仅展示部分加锁。

加锁解决多线程引起的冲突

```
1        #include<pthread.h>
2       pthread_mutex_lock(&counter_mutex);
3       result = sendto(s, sendbuff_data, header_size + thislen, 0,
            (sockaddr*)&router_addr, routerlen);
4       LastByteSent += thislen;
5       pthread_mutex_unlock(&counter_mutex);
```