



南开大学  
Nankai University

计算机学院  
计算机网络实验报告

实验 3: 基于 UDP 服务设计可靠传输协  
议并编程实现 (3-1)

姓名：谢畅

学号：2113665

专业：计算机科学与技术

2023 年 11 月 17 日

# 目录

<b>1</b>	<b>前期准备</b>	<b>1</b>
1.1	实验概述 . . . . .	1
1.2	前期准备技术 . . . . .	1
<b>2</b>	<b>协议设计: 原理分析</b>	<b>1</b>
2.1	核心原理 . . . . .	1
2.2	总体实现 . . . . .	4
2.2.1	数据包格式 . . . . .	5
2.2.2	交互方式 . . . . .	6
<b>3</b>	<b>实现方法: 代码逻辑</b>	<b>7</b>
3.1	数据包相关 . . . . .	7
3.2	差错检验: 校验和 . . . . .	11
3.3	建立连接与断开连接 . . . . .	12
3.3.1	建立连接 . . . . .	12
3.3.2	断开连接 . . . . .	14
3.4	传输逻辑: rdt3.0 . . . . .	17
3.4.1	发送端实现逻辑 . . . . .	17
3.4.2	接收端实现逻辑 . . . . .	23
3.4.3	停等机制逻辑分析 . . . . .	26
<b>4</b>	<b>实验思考以及问题分析</b>	<b>27</b>
4.1	延时与丢包 . . . . .	27
4.2	解决阻塞次数过多的情况 . . . . .	27
<b>5</b>	<b>实验结果</b>	<b>28</b>
5.1	文件传输测试 . . . . .	28
5.2	延时与丢包测试 . . . . .	31

# 1 前期准备

## 1.1 实验概述

本次实验利用数据报套接字在用户空间实现面向连接的可靠数据传输，功能包括：建立连接、差错检测、接收确认、超时重传等。流量控制采用停等机制，完成给定测试文件的传输。

其中差错检验采用校验和，建立连接采用 tcp 的握手、挥手方式，接受确认、确认重传采用 rdt3.0 的状态机，单向传输：客户端向服务器发送文件，流量控制的停等机制为：发送端发送一个分组，然后等待接收端响应。

## 1.2 前期准备技术

- 数据包格式。自定义消息类型，对消息进行封装，便于客户端与服务器读取信息，同时定义一系列消息的发送与解析的流程。
- 建立连接、断开连接。学习 tcp 的三次握手、四次挥手进行设计。
- rdt3.0 的运转方式
- 校验和的运转方式

# 2 协议设计: 原理分析

## 2.1 核心原理

### Socket

- Stream Sockets

流式套接字，也称为 TCP 套接字，是一种面向连接的套接字类型。它提供了可靠的、双向的、基于字节流的通信方式。流式套接字使用 TCP（传输控制协议）作为底层协议。确保数据按顺序到达，且没有丢失。

- Datagram Sockets

数据报套接字，也称为 UDP 套接字，是一种无连接的套接字类型。它使用 UDP（用户数据报协议）作为底层协议。不需要建立连接，可以直接发送数据包，没有顺序。

### 差错检测: 校验和

udp 数据报的差错检验中, 采取在数据报头部增加一个 checksum 字段的方式。发送方在发送数据时计算出 checksum 的值并填入其中。而接收方会判断将接受的数据进行差错检验, 具体如下:

### 1. 对于发送方

- 数据划分: 要发送的数据被分成固定大小的数据块, 每块称为一个字。如果数据长度不能被 16 位整除, 会添加填充以满足这个条件。
- 求和: 对所有的 16 位字进行求和。这里使用的是二进制求和, 也就是将所有的 16 位字看作无符号整数, 然后将它们相加。这个求和过程是逐字进行的, 每相邻两个字相加, 超出 16 位的部分被回卷 (即被截断并加到最低位)。
- 取反: 得到的和被取反 (按位取反), 即将所有的比特位 (0 变 1, 1 变 0)。最终得到的取反值就是 UDP 数据包的校验和字段。这个校验和随数据一起发送, 接收端在接收数据后也会执行相同的校验和算法。

### 2. 对于接收方

- 接收数据: 接收数据包并提取校验和字段。
- 计算校验和: 接收端使用与发送端相同的算法对接收到的数据进行校验和计算。
- 比较校验和: 接收端计算得到的校验和值与数据包中的校验和字段进行比较。如果这两个值不相等, 就意味着数据在传输过程中发生了改变。

## 可靠数据传输

RDT (可靠数据传输) 协议是一种在不可靠的通信信道上实现可靠数据传输的协议。RDT 3.0 (可靠数据传输 3.0) 是其中的一个版本, 它通过有限状态机实现了可靠的数据传输。

这里为了完成接受确认、超时重传, 采用 rdt3.0 的方式设计。rdt3.0 可以解决下层通道的分组丢失 (数据分组或者 ACK 分组) 的问题, 同时这里采取停等机制。

### 1. 对于发送端状态机

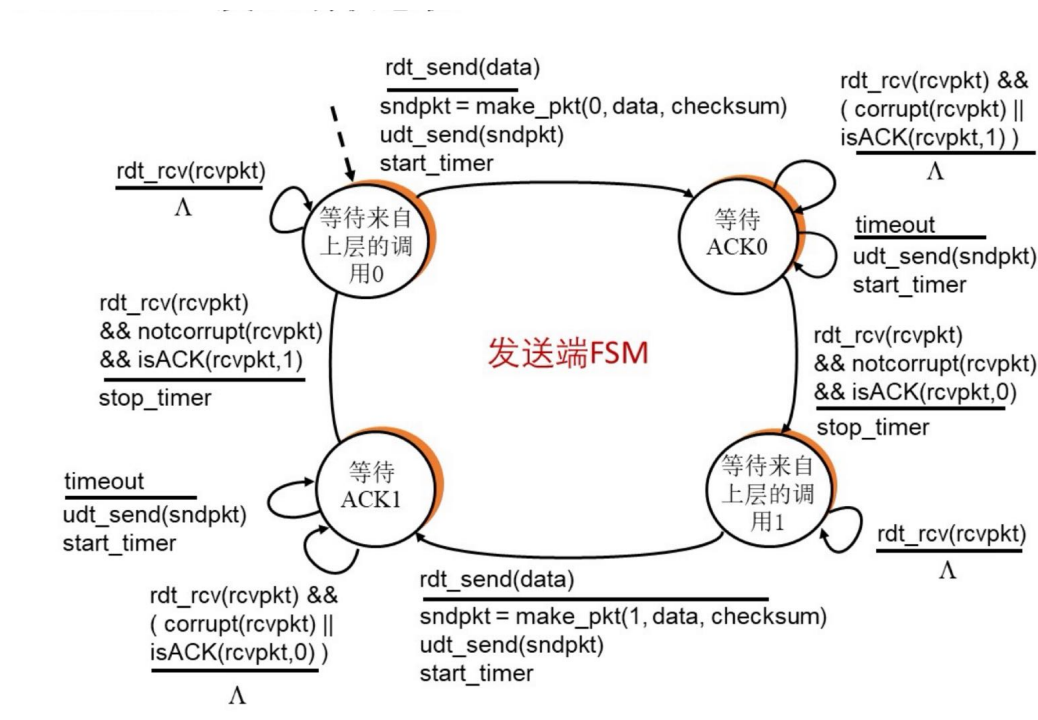


图 2.1: rdt3.0: 发送端状态机

- 首先这里的 ACK 确认，数据报有 2 种状态，即 0、1 状态，作用是为了区分新收到的与过去收到的数据段。
- 等待上层数据：发送端一开始处于等待上层数据的状态，等待来自应用层的数据。
- 打包并发送数据：一旦收到来自应用层的数据，发送端将数据打包成一个数据包，并设置序列号。随后将该数据包发送到接收端。
- 等待确认：发送端等待接收到来自接收端的确认。如果在超时时间内没有收到确认，发送端会重新发送当前数据包。
- 收到确认：一旦收到确认，发送端将根据确认号确定哪些数据已经被成功接收，然后更新发送窗口和等待发送的数据，进入下一个数据发送的状态。

## 2. 对于接收端状态机

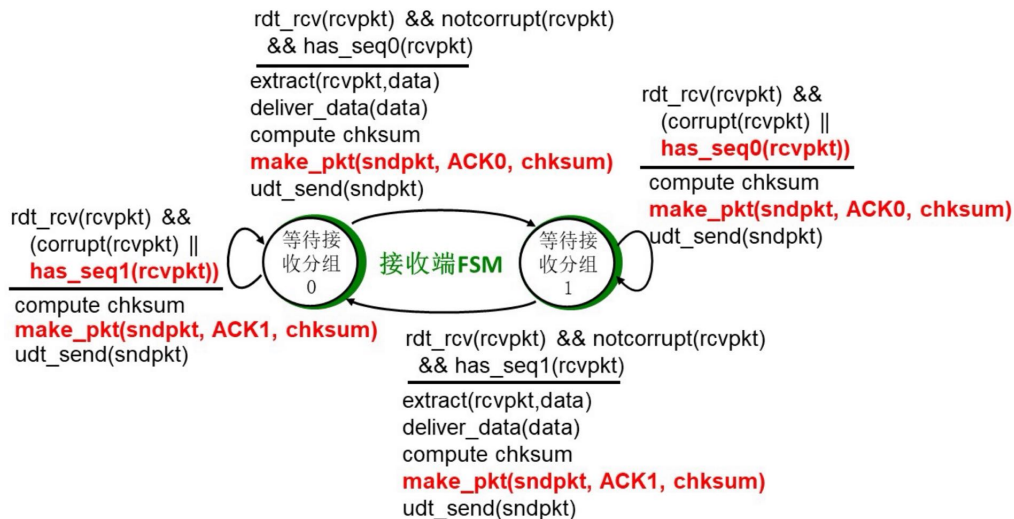


图 2.2: rdt3.0: 接收端状态机

- 等待数据：接收端一开始处于等待数据的状态，等待从发送端接收到数据包。
- 检查校验和：接收端收到数据后首先验证校验和字段，如果校验和不匹配，接收端会丢弃这个数据包。
- 检查序列号：如果校验和验证通过，接收端检查数据包的序列号，以确定数据包是否按序到达。如果接收到的数据包的序列号与期望的序列号相匹配，接收端会发送确认，然后将数据交付给应用层；如果不匹配，则丢弃该数据包并发送上一个正确接收的数据包的确认。
- 发送确认：接收端向发送端发送确认，确认号表示接收到的最后一个连续的数据包。

### 3. 超时和重传, 这里只有发送方会检测超时的情况

- 超时处理：发送端在发送数据后启动一个定时器，在规定时间内未收到确认，则触发超时事件。
- 超时重传：超时事件发生时，发送端会重新发送未收到确认的数据包，并重新启动定时器。

## 2.2 总体实现

基于上述原理，在这次实验我实现了采用基于 udp 的可靠数据传输协议，设计了数据包的格式，规定了文件传输的方式（交互方式），客户端向服务器单向发送文件，同时在发送文件前要建立连接，在发完文件后会断开连接。为了确保可靠数据传输，采用差错检验、rdt3.0 的模式。

下面为具体的实现:

- 采取数据报套接字 (Datagram sockets), 所以我们的底层协议是基于 udp 的。udp 不需要建立连接, 采用 *sendto* 与 *recvfrom* 进行收发消息即可。所以为了在不可靠信道上实现可靠数据传输的协议, 我选择 rdt3.0 进行实现。
- 数据包的格式借鉴了 UDP 与 TCP 的数据包格式, 设计了一种包含头部及数据的数据包格式。头部包含发送的源端口、目的端口、数据包的序列号、确认号、标志位、区分 ack/seq: 0/1 状态的 ack\_id 字段, 数据包长度、校验和。同时设计了产生数据包、解析数据包的相关函数, 以及一些标志位:ACK 等等。
- 对于建立连接, 这里采用 TCP: 三次握手与四次挥手的方式。  
三次握手: 客户端发送连接请求; 服务端响应并确认; 客户端确认。  
四次挥手: 客户端发送关闭请求; 服务端确认关闭请求; 服务端发送关闭请求; 客户端确认关闭。  
由于此实验提供的路由器程序中, 只有客户端发给服务器的消息可靠 (服务器一定有回应), 而服务器发给客户的信息不可靠, 即客户可能没有回应。说明服务器可能在最后一次收不到确认, 这里做了相应的处理。
- 对于差错检验, 实现的逻辑其实是 udp 校验和的逻辑, 具体原理已经解释过, 发送方与接收方都做同样的相加取反的操作, 但是发送方的相加取反是为了得到校验和字段, 而接收方是为了判断数据的正确性。
- 对于可靠数据传输, 采用 rdt3.0 模式, 与上面的原理图实现一致。值得注意的是这里 *recvfrom* 要设置成非阻塞模式, 即超时模式。当然只需要发送端设置即可, 然后与 rdt3.0 一样的是, 发送端在超时时间如果没有收到确认, 就会重发当前数据包。实现代码在后面会具体讲解。
- 在本次实验中实现的是单方可靠的数据传输, 即从客户端向服务器端发送文件。

### 2.2.1 数据包格式

在这里我设计了数据包的格式, 即 *Mymsg* 结构体, 如下图2.3所示。

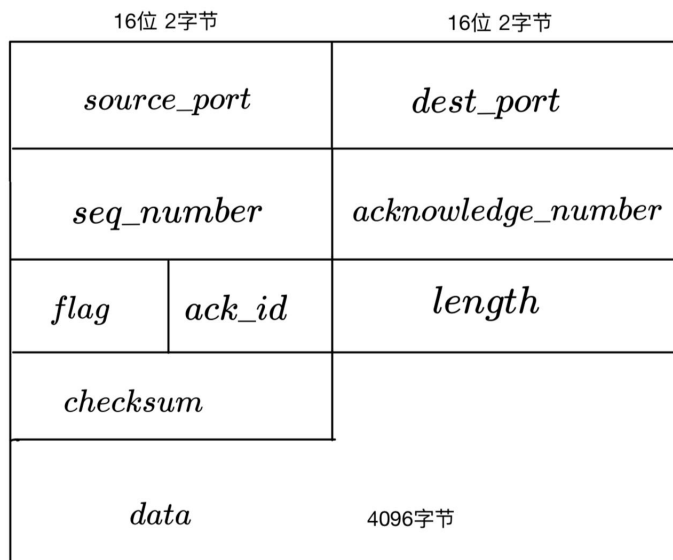


图 2.3: 自定义数据包格式 Mymsg 类型

其中头部大部分数据为 16 位, 即 2 字节的 *unsigned short* 类型, *source\_port* 表示源端口, *dest\_port* 表示发送的目的端口。*seq\_number* 表示当前数据包的序号, *acknowledge\_number* 表示分组的确认号, *length* 表示 *data* 的实际字节数 (最后一个分组可能不足 4096 字节), 而 *checksum* 表示校验和。

而 *flag*、*ack\_id* 为 8 位, 是 *char* 类型。其中 *flag* 表示标志位, 在这里我设计了 *ACK*, *FIN*, *SYN*, *LAST* 四个标志位, 在文件传输的过程中会用到。而 *ack\_id* 会在 *rdt3.0* 中用到, 表示当前传输的 *ack/seq* 的状态为 0 或 1, 作用是区分新分组与旧分组。

而这里的数据段, 我设置为 4096 字节, 在此次实验中传输文件较大时, 可以保持一个合理的分组数量。经过计算, 这里一个 *Mymsg* 占用 4110 个字节。

### 2.2.2 交互方式

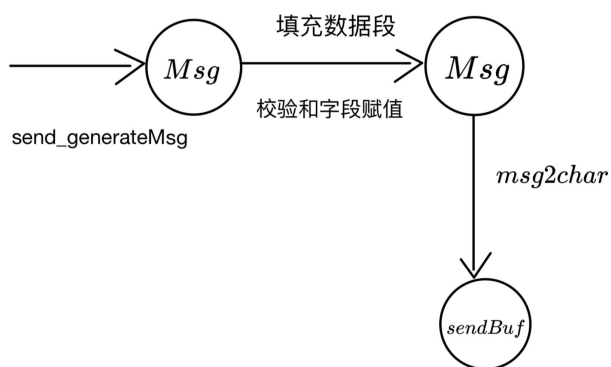


图 2.4: 发送流程

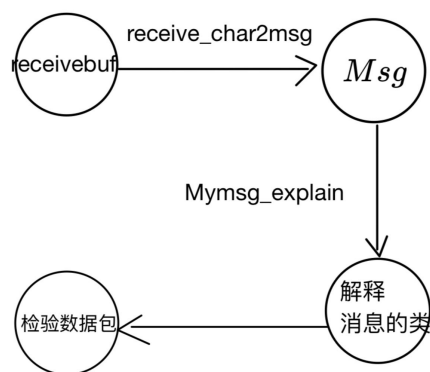


图 2.5: 接受流程



在图2.4中, 是发送数据包的流程。

- 首先调用 `send_generate` 函数, 完成头部相关的字段的赋值, 其中如果没有数据段 `data`, 那么在此函数中即可计算完校验和字段, 赋值即可。
- 对于需要发送文件的情况, 此时还需要调用 `memcpy` 完成 `data` 字段的赋值, 然后将校验和字段置 0, 最后调用 `check_sun` 计算校验和的值, 填入 `checksum` 即可。
- 将得到的数据包转为 `char*`, 调用 `sendto` 函数进行发送。

在图2.5中, 是接受数据包的流程。

- 首先调用 `recvfrom` 函数, 接受 `char*receivebuf`, 我们的数据包是固定大小 `Msg_size`, 将接收到的字节数储存即可。
- 调用 `receive_char2msg` 将接受的字节数还原为一个 `Msg` 数据包。
- 由于我们需要解析 `Msg` 数据包, 这里设计了一个类 `Mymsg_explain`, 里面存储一个 `Msg`, 成员函数用来解析 `Msg`。
- 最后做差错检验、判断标志位等等即可, 若出错那么根据 `rdt3.0` 的重传相关 `ACK` 即可。

此外, 客户端发消息, 服务端收消息的逻辑基于 `rdt3.0`, 原理在上一节已经提及。

## 3 实现方法: 代码逻辑

### 3.1 数据包相关

#### 1. 数据包与相关标志位

这里先定义了四个标志位, 对应 `Msg` 中的 `flag` 字段, 一个 `char` 有 8 位, 因此先暂时利用 4 位设置 4 个 `flag`。ACK 表示接受确认标志, SYN 表示请求连接的标志, FIN 表示请求断开连接的标志, LAST 标志着此数据包为最后一个分组。

`TIMEOUT` 表示着超时时间, 发送端接受 ACK 的 `recvfrom` 的超时时间会设置为 `TIMEOUT`, 同时如果超时了, 发送端会重传上一个分组。而 `MISSCOUNT` 用来表示发送方一个分组连续传送的最大次数, 超过此次数, 就会开始增加 `TIMEOUT` 时间, 因此此时网络延时明显大于我们设置的 `TIMEOUT`, 其中实验结果证明此方法是有效的, 可以避免过多的重传分组。

---

<sup>1</sup> //标志位

<sup>2</sup> `#define ACK 0B00000001`

```

3  #define SYN  0B00000010
4  #define FIN  0B00000100
5  #define LAST 0B00001000
6  //一个数据包重传 12 次, 增大 TIMEOUT 时间
7  #define MISSCOUNT 12
8  //超时的时间
9  #define TIMEOUT 200
10 //数据段最大的大小
11 #define MSS 4096
12 struct Mymsg {
13     unsigned short source_port; //源端口
14     unsigned short dest_port; //目的端口
15     unsigned short seq_number; //发送数据包的序号 16 位
16     unsigned short acknowledge_number; //确认号字段 16 位
17     char flag; //标志位
18     char ack_id; //rdt3.0 ack 有 ack0 与 ack1
19     unsigned short length; //数据包长度 16 位
20     unsigned short checksum; //校验和 16 位
21     char data[MSS]; //数据大小, 最长 MSS
22 };
23 //4110 字节
24 #define Msg_size sizeof(Mymsg)

```

## 2.send\_generateMsg: 赋值数据包头部

这个函数的作用是先形成一个 Msg 数据包, 为头部赋值。在函数体内还会调用 *check\_sum* 计算校验和, 需要强调的是, 必须是数据段为空才能计算; 如果有数据 data, 需要将 Msg 的 data 先赋值, 然后置其校验和字段为 0, 再计算校验和字段, 赋值。

```

1 Mymsg send_generateMsg(unsigned short s_p, unsigned short d_p, unsigned short seq,
2     unsigned short ack, char flag, char ack_id, unsigned short length = MSS,
3     unsigned short ck_sum = 0) {
4     Mymsg temp;
5     temp.source_port = s_p;
6     temp.dest_port = d_p;
7     temp.seq_number = seq;
8     temp.acknowledge_number = ack;
9     temp.flag = flag;
10    temp.ack_id = ack_id;
11    temp.length = MSS;

```

```

12     temp.checksum = ck_sum;
13     char* p = msg2char(temp);
14     //产生 checksum 的值
15     temp.checksum = check_sum(p, Msg_size);
16     return temp;
17 }

```

---

### 3. 数据包 Msg 转为 char\*

```

1 //需要把封装的 message 转化为 char 去发送
2 char* msg2char(Mymsg message) {
3     char temp[Msg_size];
4     memcpy(temp, &message, sizeof(Mymsg)); //直接将字节数拷贝
5     return temp;
6 }

```

---

### 4. 将接受的 char\* 转为 Msg 数据包

```

1 //接收的 char* 转为 message
2 Mymsg receive_char2msg(char* receive_buf) {
3     Mymsg temp;
4     memcpy(&temp, receive_buf, sizeof(Mymsg));
5     return temp;
6 }

```

---

### 5. 定义 Mymsg\_explain 解析数据包

这个类中存有一个成员变量 Msg，成员函数解析其标志位等等。

```

1 class Mymsg_explain {
2 public:
3     Mymsg msg;
4     Mymsg_explain(Mymsg s) :msg(s) {}
5     void Mymsg_copy(Mymsg s) { memcpy(&msg, &s, Msg_size); }
6     bool isACK0() {
7         if (msg.ack_id == '0' && msg.flag & ACK) {
8             return 1;
9         }
10        return 0;
11    }

```

```
12     bool isACK1() {
13         if (msg.ack_id == '1' && msg.flag & ACK) {
14             return 1;
15         }
16         return 0;
17     }
18     bool isACK() {
19         if (msg.flag & ACK)
20             return 1;
21         return 0;
22     }
23     bool isSYN() {
24         if (msg.flag & SYN)
25             return 1;
26         return 0;
27     }
28     bool isFIN() {
29         if (msg.flag & FIN)
30             return 1;
31         return 0;
32     }
33     bool isSeq0() {
34         if (msg.ack_id == '0')
35             return 1;
36         return 0;
37     }
38     bool isSeq1() {
39         if (msg.ack_id == '1')
40             return 1;
41         return 0;
42     }
43     bool isLAST() {
44         if (msg.flag & LAST)
45             return 1;
46         return 0;
47     }
48 };
```

---

## 3.2 差错检验: 校验和

---

```
1 //这里的 buf 是 sento/recvfrom 的参数 buf
2 //差错检验
3 unsigned short check_sum(char* buf, int length) {
4     // unsigned long 占 4 个字节
5     unsigned long sum = 0; //32 位
6     for (int i = 0; i < length; i += 2) {
7         //获取 2 个 char 转为 16 位
8         unsigned short temp;
9         if (i == length - 1) {
10             memset(&temp, 0, 2);
11             //注意是填充 temp 的高 8 位, 低 8 位是 0
12             memcpy((char*)&temp + 1, buf + i, 1);
13         }
14         else
15             memcpy(&temp, buf + i, 2);
16         sum += temp;
17         if (sum & 0xFFFF0000) {
18             sum &= 0xFFFF;
19             sum++;
20         }
21     }
22     return ~(sum & 0xFFFF);
23 }
```

---

- 变量定义: sum 是一个无符号长整型 (unsigned long) 32 位, 用于存储计算过程中的累加和。因为我们是 16 位相加, 可能超过 16 位, 即 17 位。所以选取 32 位。
- 循环遍历数据: 使用 for 循环遍历 buf 中的数据。在每次迭代中, 取出两个连续的字符 (即两个字节), 将它们合并成一个 16 位的无符号短整型 temp。如果最后只有 1 字节, 没有 2 字节, 将这个字节填充至高 8 位, 低 8 位为 0。
- 校验和计算: 将取出的 16 位 temp 加到 sum 中。如果 sum 的高 16 位不为零 (即大于等于 0xFFFF0000), 说明发生了进位, sum 加 1 即可。
- 结果返回: 最终返回的是 sum 的 16 位取反值 (~sum & 0xFFFF), 作为校验和值。

综上, 这个函数按两个字节 (16 位) 为单位对数据进行累加求和, 以计算校验和。在每次累加后, 它检查是否有发生进位, 如果有, 则将进位的部分加到低 16 位上。最后返

回的是校验和的取反值, 用于在传输结束后与接收到的校验和进行比较, 从而判断数据是否发生了变化。

### 3.3 建立连接与断开连接

#### 3.3.1 建立连接

这里采用三次握手, 目的是确保双方都能够正确地发送和接收数据。在实验给出的路由器程序可以设置丢包, 因此这里三次握手可以检测出传输存在问题。由于路由器是客户端发送文件给服务器的单向传输, 因此服务器给客户发送的消息, 很可能没有确认回信(丢失)。

因此这里, 服务器在前面发送的 SYN+ACK, 很可能没有回信, 也就是服务器可能收不到第三次握手确认消息。所以最后, 这里限制了服务器接受的 *recvfrom* 超时时间, 超过时间服务器会退出, 开始文件传输的接受。因为实际上客户端发送给服务器消息并不会出现问题, 所以可以继续单向传输。

#### 1. 客户端

---

```

1 void hand3Shakes() {
2     char buffer[Msg_size];
3     //避免收不到阻塞
4     int timeout = TIMEOUT * 4;
5     setsockopt(clientSocket, SOL_SOCKET, SO_RCVTIMEO, (const char*)&timeout, sizeof(timeout));
6     while (1) {
7         //SYN 请求
8         Mymsg temp = send_generateMsg(source_port, dest_port, 0, 0, SYN, '0');
9         sendto(clientSocket, msg2char(temp), Msg_size, 0,
10             (struct sockaddr*)&serverAddr, server_addrlen);
11         cout << "[客户端]: 发送第一次握手 SYN 请求" << endl;
12         int bytesRead = recvfrom(clientSocket, buffer, Msg_size, 0,
13             (struct sockaddr*)&serverAddr, &server_addrlen);
14         Mymsg_explain cur = Mymsg_explain(receive_char2msg(buffer));
15         if (bytesRead > 0 && check_sum(buffer, Msg_size) == 0 && cur.isSYN()
16             && cur.isACK()) {
17             cout << "[客户端]: 第二次握手接受 SYN+ACK" << endl;
18             //第三次握手, 可能丢失
19             Mymsg temp1 = send_generateMsg(source_port, dest_port,
20                 cur.msg.acknowledge_number, cur.msg.seq_number + 1, ACK, '0');
21             sendto(clientSocket, msg2char(temp1), Msg_size, 0,

```

```

22         (struct sockaddr*)&serverAddr, server_addrlen);
23         cout << "[客户端]: 发送第三次握手 ACK" << endl;
24         break;
25     }
26     else
27         continue;//重发第一次
28
29 }
30 cout << "[客户端]: 三次握手连接结束" << endl;
31 cout << "[客户端]: 目的端口为:" << dest_port << endl;
32 }

```

## 2. 服务器端

```

1  //3 次握手
2  void hand3Shakes() {
3      int tv = TIMEOUT * 4;
4      //最后一次挥手可能丢包, 这里避免阻塞
5      setsockopt(serverSocket, SOL_SOCKET, SO_RCVTIMEO, (const char*)&tv, sizeof(tv));
6      char buffer[Msg_size];
7      while (1) {
8          int bytesRead = recvfrom(serverSocket, (char*)buffer, Msg_size, 0,
9              (struct sockaddr*)&clientAddr, &client_addrlen);
10         if (bytesRead <= 0 && check_sum(buffer, Msg_size)) {
11             cout << "[服务器]: 第一次握手重连中" << endl;
12             continue;
13         }
14         else {
15             Mymsg_explain cur = Mymsg_explain(receive_char2msg(buffer));
16             if (check_sum(buffer, Msg_size) == 0 && cur.isSYN()) {
17                 cout << "[服务端]: 第一次握手, 接收到客户端 SYN 请求!" << endl;
18                 //记录客户端端口号
19                 dest_port = ntohs(clientAddr.sin_port);
20                 //准备发送 SYN+ACK
21                 Mymsg temp = send_generateMsg(source_port, dest_port,
22                     cur.msg.acknowledge_number, cur.msg.seq_number + 1, SYN | ACK, '0');
23                 sendto(serverSocket, msg2char(temp), Msg_size,
24                     0,
25                     (struct sockaddr*)&clientAddr, client_addrlen);

```

```

26         cout << "[服务端]: 第二次握手, 发送 SYN+ACK!" << endl;
27         //清 Obuffer 重新接受
28         memset(buffer, 0, Msg_size);
29         int ret = recvfrom(serverSocket, (char*)buffer, Msg_size,
30                             0, (struct sockaddr*)&clientAddr, &client_addrlen);
31         Mymsg_explain cur1 = Mymsg_explain(receive_char2msg(buffer));
32         if (ret > 0 && (check_sum(buffer, Msg_size) == 0 && cur1.isACK())) {
33             cout << "[服务端]: 第三次握手, 收到 ACK 字段, 成功连接!" << endl;
34             break;
35         }
36         else {
37             cout << "[服务端]: 第三次握手失败, 只能确保客户端发送服务器可靠\n";
38             break;
39         }
40     }
41 }
42 else
43     continue;
44 }
45 break;
46 }
47 cout << "[服务端]: 三次握手流程结束" << endl;
48 cout << "[服务器]: 目的端口为:" << dest_port << endl;
49 }

```

### 3.3.2 断开连接

采用四次挥手的方式, 确保双方都完成了数据的传输。其中由于此实验所给路由器是单向传输可靠, 即客户端发送文件给服务器可靠, 那么说明服务器发送的消息, 可能客户端不会发送确认。

这意味着在最后一次挥手时, 由于是客户端对服务器之前挥手的确认, 所以可能不会有这个确认。说明服务器可能收不到第四次挥手, 因此在设计时需要考虑这点, 设置 *recvfrom* 的最大延时, 超过延时自动退出即可。

所以其实服务器可以在最后一次挥手时, 发现自己传输的数据包可能无法被确认, 这样验证了我们是单向传输可靠, 客户发送的数据包一定会被确认。

- 通过四次挥手, 双方确认彼此已经完成数据的传输, 并正常释放连接和网络资源, 避免了未完成数据传输和资源占用的情况。



- 保证了在关闭连接之后, 双方的状态都能够正确地维护和处理, 避免了潜在的问题和错误连接的出现。

## 1. 客户端

```

1
2 void hand4Shakes() {
3     while (1) {
4         //seq 代表分组编号 ack=seq+1 发送 FIN+ACK, 断开
5         Mymsg s_temp1 = send_generateMsg(source_port, dest_port, 1, 1, FIN | ACK, '0');
6         sendto(clientSocket, msg2char(s_temp1), Msg_size,
7         0, (struct sockaddr*)&serverAddr, server_addrlen);
8         printf("[客户端]: 发送第一次挥手 FIN+ACK, 校验和:%d,seq:%d,ack:%d\n",
9         s_temp1.checksum, s_temp1.seq_number, s_temp1.acknowledge_number);
10        char curr_buf[Msg_size];
11        //之前设置过超时时间, 不会阻塞
12        int ret = recvfrom(clientSocket, curr_buf, Msg_size,
13        0, (struct sockaddr*)&serverAddr, &server_addrlen);
14        Mymsg_explain r_temp1 = Mymsg_explain(receive_char2msg(curr_buf));
15        if (ret > 0 && check_sum(curr_buf, Msg_size) == 0 && r_temp1.isACK()) {
16            printf("[客户端]: 收到第二次挥手 ACK,seq:%d,ack:%d\n",
17            r_temp1.msg.seq_number, r_temp1.msg.acknowledge_number);
18            memset(curr_buf, 0, Msg_size);
19            int ret2 = recvfrom(clientSocket, curr_buf, Msg_size,
20            0, (struct sockaddr*)&serverAddr, &server_addrlen);
21            Mymsg_explain r_temp2 = Mymsg_explain(receive_char2msg(curr_buf));
22            if (ret2 > 0 && check_sum(curr_buf, Msg_size) == 0
23            && r_temp2.isFIN() && r_temp2.isACK()) {
24                printf("[客户端]: 收到第三次挥手 FIN+ACK,seq:%d,ack:%d\n"
25                , r_temp2.msg.seq_number, r_temp2.msg.acknowledge_number);
26                //seq 代表分组编号 ack=seq+1
27                Mymsg s_temp2 = send_generateMsg(source_port, dest_port,
28                r_temp2.msg.acknowledge_number, r_temp2.msg.seq_number + 1, ACK, '0');
29                sendto(clientSocket, msg2char(s_temp2), Msg_size, 0,
30                (struct sockaddr*)&serverAddr, server_addrlen);
31                printf("[客户端]: 发送第四次挥手 ACK, 校验和:%d,seq:%d,ack:%d\n",
32                s_temp2.checksum, s_temp2.seq_number, s_temp2.acknowledge_number);
33                cout << "-----退出-----" << endl;
34                Sleep(2 * TIMEOUT);

```

```

35         return;
36     }
37 }
38 else
39     continue;//从头开始
40 }
41
42 }

```

## 2. 服务器端

```

1  //4 次挥手
2  void hand4Shakes() {
3      int tv = TIMEOUT;
4      //最后一次挥手可能丢包，这里避免阻塞
5      setsockopt(serverSocket, SOL_SOCKET, SO_RCVTIMEO, (const char*)&tv, sizeof(tv));
6      while (1) {
7          char buf1[Msg_size];
8          int bytesRead = recvfrom(serverSocket, (char*)buf1, Msg_size, 0,
9              (struct sockaddr*)&clientAddr, &client_addrlen);
10         Mymsg_explain r_temp1 = Mymsg_explain(receive_char2msg(buf1));
11         if (bytesRead > 0 && r_temp1.isFIN() && r_temp1.isACK()
12             && check_sum(buf1, Msg_size) == 0) {
13             printf("[服务端]: 收到第一次挥手的 FIN+ACK,seq:%d,ack:%d\n",
14                 r_temp1.msg.seq_number, r_temp1.msg.acknowledge_number);
15             Mymsg temp1 = send_generateMsg(source_port, dest_port,
16                 1, r_temp1.msg.seq_number + 1, ACK, '0');//seq 代表分组编号
17             sendto(serverSocket, msg2char(temp1), Msg_size, 0,
18                 (struct sockaddr*)&clientAddr, client_addrlen);
19             printf("[服务端]: 发送第二次挥手的 ACK, 校验和:%d,seq:%d,ack:%d\n",
20                 temp1.checksum, temp1.seq_number, temp1.acknowledge_number);
21             //这里服务器发送的不会丢包，简便起见不重传了
22             Mymsg temp2 = send_generateMsg(source_port, dest_port, 1,
23                 r_temp1.msg.seq_number + 1, FIN | ACK, '0');//seq 代表分组编号
24             sendto(serverSocket, msg2char(temp2), Msg_size, 0,
25                 (struct sockaddr*)&clientAddr, client_addrlen);
26             printf("[服务端]: 发送第三次挥手的 FIN+ACK, 校验和:%d,seq:%d,ack:%d\n",
27                 temp2.checksum, temp2.seq_number, temp2.acknowledge_number);
28             //第四次

```

```

29     unsigned long long start_stamp = GetTickCount64();//开始计时
30     while (1) {
31         memset(buf1, 0, Msg_size);
32         int bytesRead1 = recvfrom(serverSocket, (char*)buf1, Msg_size
33             , 0, (struct sockaddr*)&clientAddr, &client_addrlen);
34         Mymsg_explain r_temp2 = Mymsg_explain(receive_char2msg(buf1));
35         if (bytesRead > 0 && r_temp2.isACK() && check_sum(buf1, Msg_size) == 0) {
36             printf("[服务器]: 收到第四次挥手的 ACK,seq:%d,ack:%d\n"
37                 , r_temp2.msg.seq_number, r_temp2.msg.acknowledge_number);
38             cout<<"-----退出-----"<<endl;
39             return;
40         }
41         else {
42             if (GetTickCount64() - start_stamp > TIMEOUT) {
43                 //客户发送的不可靠,可能不见了
44                 cout << "[服务器]: 第四次挥手因故障未收到, 自动退出" << endl;
45                 return;
46             }
47             else
48                 continue;
49         }
50     }
51 }
52 else
53     continue;
54 }
55 }

```

### 3.4 传输逻辑:rdt3.0

原理图在第二节已经分析完毕, 这里分析代码的实现逻辑。

#### 3.4.1 发送端实现逻辑

由于代码过长, 这里分成两部分: 第一部分解释对于数据段超过 MSS, 需要分组的情况; 第二部分解释对于不需要分组的部分。

首先这里发送端是一个有 4 状态的状态机。在这个实验中, 需要完成调用 0, 等待 ACK0, 调用 1, 等待 ACK1 四种状态的转换。在代码注释里可以清楚的区分四个状态的代码。

- **调用 0:** 即完成对分组数据包的发送, 且发送 seq 的状态为 0(状态只有 0/1 之分)。

具体而言获取到当前要发送的分组序号 seq, 产生该分组, 进行发送, 随即转为等待 ACK0 的状态。

- **等待 ACK0 状态:** 等待接收方传来 ACK0, 即对之前发送的 seq0 进行确认。

具体来说, 首先我已经设置了 `recvfrom` 的最大延时为 `timeout ms`。如果接受的数据包崩坏, 或者不是 ACK0, 那么将继续此循环直至超时, 当超时的时候, 会首先重发分组, 并且记录当前分组重发个数, 为 `resend_flag`, 如果重发个数超过 `MISSCOUNT`, 将 `timeout` 变为原来的 1.5 倍, 继续循环。

如果发现接受的数据包正确, 通过校验和、ACK0 的确认, 那么可以退出此状态, 进入调用 1 状态。

- **调用 1:** 即完成对分组数据包的发送, 且发送 seq 的状态为 1。

获取当前应该传送的数据段, 产生相应的分组, 进行发送, 然后转为等待 ACK1 的状态。

- **等待 ACK1 状态:** 等待接收方传来的 ACK1, 即对之前发送的 seq1 进行确认。

与等待 ACK1 的操作其实一致, 只是当前要获取的 ACK 为状态 1 的 ACK, 与之前旧的 ACK0 作区分。

- 最后, 在传输最后一个分组时, 需要注意此时数据长度不为固定的 MSS。此时我们要更新 `length` 的值, 同时设置分组的 `LAST` 标志位。

---

```

1 //buffer 是存储要传输的字节数组
2 //len 为总共的字节数
3 void sendMsg(char* buffer, int len) {
4     //分组
5     //使用停等机制, 发送一个分组等待接收端响应
6     int timeout = TIMEOUT;
7     setsockopt(clientSocket, SOL_SOCKET, SO_RCVTIMEO,
8         (const char*)&timeout, sizeof(timeout));
9     if (len > MSS) {
10         int groupN = MSS * len / MSS < len ? len / MSS + 1 : len / MSS;
11         //开始传送
12         int seq = 0;
13         //由于单向传输, 服务器端只赋值 seq 字段, 客户端只赋值 ack 字段, 对应分组编号
14         while (1) {
15             //进入调用 0

```

```
16     if (seq >= groupN)
17         break;
18     //seq 代表分组编号
19     Mymsg temp0 = send_generateMsg(source_port, dest_port, seq, 0, 0, '0');
20     //内容赋值
21     if (seq == groupN - 1) {
22         temp0.length = len - (groupN - 1) * MSS;
23         temp0.flag |= LAST;
24         memcpy(temp0.data, buffer + seq * MSS, len - (groupN - 1) * MSS);
25     }
26     else
27         memcpy(temp0.data, buffer + seq * MSS, MSS);
28     temp0.checksum = 0;
29     char* p = msg2char(temp0);
30     //产生 checksum 的值
31     temp0.checksum = check_sum(p, Msg_size);
32     sendto(clientSocket, msg2char(temp0), Msg_size,
33     0, (struct sockaddr*)&serverAddr, server_addrlen);
34     printf(" 发送分组 seq:%d, 分组状态 0, 校验和:%d\n", seq, temp0.checksum);
35     unsigned long long start_stamp = GetTickCount64(); //开始计时
36     //进入等待 ACK0 状态
37     while (1) {
38         char curr_buf[Msg_size];
39         int ret = recvfrom(clientSocket, curr_buf, Msg_size,
40         0, (struct sockaddr*)&serverAddr, &server_addrlen);
41         Mymsg_explain r_temp0 = Mymsg_explain(recv_char2msg(curr_buf));
42         //正确, 准备进入下一个状态
43         if (ret > 0 && r_temp0.isACK0() && check_sum(curr_buf, Msg_size) == 0)
44         { printf(" 接受到确认 ACK0,ack:%d\n", r_temp0.msg.acknowledge_number);
45         break; }
46         else {} //什么也不做等待超时
47         //超时
48         if (GetTickCount64() - start_stamp > timeout) {
49             //重传分组
50             if (resend_flag > MISSCOUNT) {
51                 timeout *= 1.5;
52                 setsockopt(clientSocket, SOL_SOCKET, SO_RCVTIMEO,
53                 (const char*)&timeout, sizeof(timeout));
54                 resend_flag = 0;
```

```

55         cout << "timeout 修改为" << timeout << "ms\n";
56     }
57     sendto(clientSocket, msg2char(temp0), Msg_size,
58     0, (struct sockaddr*)&serverAddr, server_addrlen);
59     //重新计时
60     start_stamp = GetTickCount64();
61     cout << " 超时重传分组 seq:" << temp0.seq_number << endl;
62     resend_flag++;
63 }
64
65 }
66 resend_flag = 0;
67 //进入调用 1
68 seq++; //下一个分组
69 if (seq >= groupN)
70     break;
71 Mymsg temp1 = send_generateMsg(source_port, dest_port, seq, 0, 0, '1');
72 //内容赋值
73 if (seq == groupN - 1) {
74     temp1.length = len - (groupN - 1) * MSS;
75     temp1.flag |= LAST;
76     memcpy(temp1.data, buffer + seq * MSS, len - (groupN - 1) * MSS);
77 }
78 else
79     memcpy(temp1.data, buffer + seq * MSS, MSS);
80 temp1.checksum = 0;
81 p = msg2char(temp1);
82 //产生 checksum 的值
83 temp1.checksum = check_sum(p, Msg_size);
84 sendto(clientSocket, msg2char(temp1), Msg_size,
85 0, (struct sockaddr*)&serverAddr, server_addrlen);
86 printf(" 发送分组 seq:%d, 校验和:%d\n", seq, temp1.checksum);
87 //开始计时
88 start_stamp = GetTickCount64();
89 //进入等待 ACK1 状态
90 while (1) {
91     char curr_buf1[Msg_size];
92     int ret = recvfrom(clientSocket, curr_buf1, Msg_size,
93     0, (struct sockaddr*)&serverAddr, &server_addrlen);

```

```

94         Mymsg_explain r_temp1 = Mymsg_explain(receive_char2msg(curr_buf1));
95
96         //正确, 准备进入下一个状态
97         if (ret > 0 && r_temp1.isACK1() && check_sum(curr_buf1, Msg_size) == 0)
98         { printf(" 接受到确认 ACK1,ack:%d\n", r_temp1.msg.acknowledge_number);
99           break; }
100        else {}//什么也不做等待超时
101        if (GetTickCount64() - start_stamp > timeout) {
102            if (resend_flag > MISSCOUNT) {
103                timeout *= 1.5;
104                setsockopt(clientSocket, SOL_SOCKET, SO_RCVTIMEO,
105                    (const char*)&timeout, sizeof(timeout));
106                resend_flag = 0;
107                cout << "timeout 修改为" << timeout << "ms\n";
108            }
109            //重传分组
110            sendto(clientSocket, msg2char(temp1), Msg_size,
111                0, (struct sockaddr*)&serverAddr, server_addrlen);
112            //重新计时
113            start_stamp = GetTickCount64();
114            cout << " 超时重传分组 seq:" << temp1.seq_number << endl;
115            resend_flag++;
116        }
117
118    }
119    seq++;//下一个
120    resend_flag = 0;
121    //第四个状态完毕, 从头开始
122    continue;
123 }
124 cout << " 成功传输完数据" << endl;
125 }
126 //传一个即可
127 else {...}
128 }

```

当只用传送一个分组时, 其实比较简单。首先发送一个分组, 注意此分组即为最后一个, 设置标志位为 *LAST*, 然后等待 ACK0, 如果超时, 那么重传分组 0。接收到 ACK0 的时候, 说明已经传输完毕, 直接返回即可。在本次实验中, 需要先传世文件名, 显然要

用到此部分。

---

```

1 void sendMsg(char* buffer, int len) {
2     ...
3     //传一个即可
4     else {
5
6         Mymsg temp = send_generateMsg(source_port, dest_port, 0, 0, LAST, '0');
7         memcpy(temp.data, buffer, len);
8         temp.length = len;
9         temp.checksum = 0;
10        char* p = msg2char(temp);
11        //产生 checksum 的值
12        temp.checksum = check_sum(p, Msg_size);
13        sendto(clientSocket, msg2char(temp), Msg_size,
14        0, (struct sockaddr*)&serverAddr, server_addrlen);
15        printf(" 发送分组 seq:%d, 校验和:%d\n", 0, temp.checksum);
16        unsigned long long start_stamp = GetTickCount64(); //开始计时
17        while (1) {
18            char curr_buf[Msg_size];
19            int ret = recvfrom(clientSocket, curr_buf, Msg_size,
20            0, (struct sockaddr*)&serverAddr, &server_addrlen);
21            Mymsg_explain r_temp = Mymsg_explain(receive_char2msg(curr_buf));
22            //什么也不做等待超时
23            if (ret <= 0 || ret > 0 && r_temp.isACK1()) {}
24            //正确, 准备进入下一个状态
25            if (ret > 0 && r_temp.isACK0() && check_sum(curr_buf, Msg_size) == 0)
26            { break; }
27            if (GetTickCount64() - start_stamp > timeout) {
28                if (resend_flag > MISSCOUNT) {
29                    timeout *= 1.5;
30                    setsockopt(clientSocket, SOL_SOCKET, SO_RCVTIMEO,
31                    (const char*)&timeout, sizeof(timeout));
32                    resend_flag = 0;
33                    cout << "timeout 修改为" << timeout << "ms\n";
34                }
35                //重传分组
36                sendto(clientSocket, msg2char(temp), Msg_size,
37                0, (struct sockaddr*)&serverAddr, server_addrlen);

```



```

38         //重新计时
39         start_stamp = GetTickCount64();
40         cout << " 超时重传分组 seq:" << temp.msg.seq_number << endl;
41         resend_flag++;
42     }
43 }
44 resend_flag = 0;
45 cout << " 成功传输完数据" << endl;
46 }
47 }

```

---

### 3.4.2 接收端实现逻辑

接收端是一个有 2 状态的状态机。在 rdt3.0 中, 接收端有等待接受分组 0、等待分组 1 两种状态。

- **等待接受分组 0:** 等待数据包 seq0 的传输。如果想跳入其他状态, 必须满足下面的条件: 分组状态为 0, 经过差错检测无误。如果不满足, 那么将发送 ACK1, 代表确认上一个分组, 等待分组 0; 如果满足上面的条件, 可以提取分组的数据, 存入 buffer 数组, 然后发送 ACK0, 代表已经确认 seq0。如果此分组含有标志位 LAST, 说明是最后一个分组, 此时结束状态循环, 退出即可; 如果不是最后一个分组, 转为等待接受分组 1, 继续循环。
- **等待接受分组 1:** 等待数据包 seq1 的传输。同样的, 只有满足接受分组状态为 1, 且经过差错检验, 才能提取数据。如果不满足, 那么发送 ACK0, 代表确认上一个分组, 等待分组 1, 继续此状态; 如果满足, 提取数据, 然后发送 ACK1, 代表已经确认 seq1。值得注意的是, 如果接受的分组有 LAST 标志, 说明传输完毕, 直接退出即可; 如果不含, 那么转为等待分组 0 即可, 继续循环。
- 由于这里需要考虑重传与丢失, 可以发现发送端会多次重传。如果三次握手客户端重传了第一次握手, 那么可能在这里会接收到。由于发送端发送的数据包不可能含有 SYN、ACK 标志位, 如果含有只可能是连接的时候的数据包, 所以需要丢弃。而且如果文件名传输完毕, 在获取文件内容时又收到了文件名的延时包, 也需要丢弃。

---

```

1 //buffer 储存收到的数据
2 void recvMsg(char* buffer, int& len) {
3     //设置超时时间
4     len = 0;
5     int seq = 0;

```

```

6   while (1) {
7       while (1) {
8           char curr_buf[Msg_size];
9           int ret = recvfrom(serverSocket, curr_buf, Msg_size,
10              0, (struct sockaddr*)&clientAddr, &client_addrlen);
11           Mymsg_explain r_temp0 = Mymsg_explain(recv_char2msg(curr_buf));
12           //cout << r_temp0.msg.data;
13           //正确, 准备进入下一个状态
14           if (ret > 0 && r_temp0.isSeq() && check_sum(curr_buf, Msg_size) == 0) {
15               //提取数据
16               //错误信息 说明文件名重传了, 握手重传了, 需要丢弃
17               if (name_flag == 1 && seq == 0 && r_temp0.isLAST() ||
18                  r_temp0.isACK() || r_temp0.isSYN())
19                   continue;
20               seq = r_temp0.msg.seq_number;
21               printf(" 接收到分组 seq:%d, 分组状态:0,check_sum:0\n", seq);
22               if (r_temp0.msg.length < MSS) {
23                   memcpy(buffer + seq * MSS, r_temp0.msg.data, r_temp0.msg.length);
24                   len += r_temp0.msg.length;
25               }
26               else {
27                   memcpy(buffer + seq * MSS, r_temp0.msg.data, MSS);
28                   len += MSS;
29               }
30               Mymsg s_temp0 = send_generateMsg(source_port, dest_port,
31                  0, seq + 1, ACK, '0'); //seq 代表分组编号 ack=seq+1
32               sendto(serverSocket, msg2char(s_temp0), Msg_size,
33                  0, (struct sockaddr*)&clientAddr, client_addrlen);
34               if (r_temp0.isLAST()) {
35                   cout << " 传输完毕" << endl;
36                   return;
37               }
38               break;
39           }
40           else {
41               //发送 ack1
42               Mymsg temp1 = send_generateMsg(source_port, dest_port,
43                  0, seq + 1, ACK, '1'); //seq 代表分组编号 ack=seq+1
44               sendto(serverSocket, msg2char(temp1), Msg_size,

```

```

45         0, (struct sockaddr*)&clientAddr, client_addrlen);
46         cout << " 发送 ack1, 期待分组:" << seq + 1 << endl;
47     }
48 }
49 //等待接受分组 1
50 while (1) {
51     char curr_buf[Msg_size];
52     int ret = recvfrom(serverSocket, curr_buf, Msg_size,
53         0, (struct sockaddr*)&clientAddr, &client_addrlen);
54     Mymsg_explain r_temp1 = Mymsg_explain(recv_char2msg(curr_buf));
55     //正确, 准备进入下一个状态
56     if (ret > 0 && r_temp1.isSeq1() && check_sum(curr_buf, Msg_size) == 0) {
57         //提取数据
58         seq = r_temp1.msg.seq_number;
59         printf(" 接收到分组 seq:%d, 分组状态:1,check_sum:0\n", seq);
60         if (r_temp1.msg.length < MSS) {
61             memcpy(buffer + seq * MSS, r_temp1.msg.data, r_temp1.msg.length);
62             len += r_temp1.msg.length;
63         }
64         else {
65             memcpy(buffer + seq * MSS, r_temp1.msg.data, MSS);
66             len += MSS;
67         }
68         Mymsg s_temp1 = send_generateMsg(source_port, dest_port,
69             0, seq + 1, ACK, '1'); //seq 代表分组编号 ack=seq+1
70         sendto(serverSocket, msg2char(s_temp1), Msg_size,
71             0, (struct sockaddr*)&clientAddr, client_addrlen);
72         if (r_temp1.isLAST()) {
73             cout << " 传输完毕" << endl;
74             return;
75         }
76     }
77     break;
78 }
79 else {
80     //发送 ack0
81     Mymsg temp0 = send_generateMsg(source_port, dest_port,
82         0, seq + 1, ACK, '0'); //seq 代表分组编号 ack=seq+1
83     sendto(serverSocket, msg2char(temp0), Msg_size,

```

```
84         0, (struct sockaddr*)&clientAddr, client_addrlen);
85         cout << " 发送 ack0, 期待分组:" << seq + 1 << endl;
86     }
87 }
88 }
89 }
```

---

### 3.4.3 停等机制逻辑分析

在此次实验中，我采用的 *RDT3.0* 机制中，使用了停等 (Stop-and-Wait) 机制来实现流量控制。停等是一种简单而有效的流量控制方法，它确保了数据的可靠传输，同时避免了过多的数据拥塞。

停等机制的原理涉及到发送方和接收方之间的交互，以及发送方如何根据接收到的确认信息来控制流量。

#### 1. 发送方：

- 发送方将数据分割成固定大小的数据包，并依次发送到接收方。
- 每发送一个数据包后，发送方启动一个定时器，等待接收到对应的确认信息或者超时。
- 如果定时器超时，发送方会认为数据包丢失，会重新发送该数据包。

#### 2. 接收方：

- 接收方接收到数据包后，会发送确认信息 (ACK) 给发送方，通知发送方已成功接收到数据。
- 如果接收方发现数据包出现错误 (比如校验和失败)，它会丢弃该数据包，并不发送确认信息。这会被发送方视为数据包丢失，从而触发重传机制。

#### 3. 流量控制：

- 发送方每次只发送一个数据包，并等待接收到对应的确认信息后才发送下一个数据包。
- 接收方的确认信息充当了流量控制的信号。只有当发送方收到确认信息后，它才会发送下一个数据包。
- 这确保了发送方和接收方之间的同步，避免了数据的过度发送，防止了网络拥塞。

## 4 实验思考以及问题分析

### 4.1 延时与丢包

在我把延时设置为 1000ms 时，发现程序出现问题。经过探索之后问题出在文件名的传输上。可以发现我把文件名的长度设置为 30，结果接收端显示长度为 4096 个字节。

```

    }
    else {
        cout << r_temp0.isSYN() << endl;
        cout << r_temp0.isACK() << endl;
        memcpy(buffer + seq * MSS, r_temp0.msg.data, MSS);
        len += MSS;
    }
    Mmsg s_temp0 = send_generateMsg(source port, dest port, 0, seq

```

图 4.6: 错误位置

经过调试后，我发现接收端错误是因为接收到的 `r_temp0` 实际上是连接的时候发送端重发的数据包。可以看到 `r_temp0` 的 `SYN` 标志位为 1，验证了猜想。

```

D:\大二上\计算机网络作业及考试\projects\lab5\server\debug\server.exe
[服务端]:第一次握手，接收到客户端SYN请求!
[服务端]:第二次握手，发送SYN+ACK!
[服务端]:第三次握手失败,只能确保客户端发送服务器可靠
[服务端]:三次握手流程结束
[服务器]:目的端口为:4001
-----开始接受文件名-----
接收到分组seq:0, 分组状态:0, check_sum:0
1
0

```

图 4.7: 错误原因

因此将代码修改为下面的形式，增加判断，解决了问题。因为在传输文件是单向的，因此发送方的数据包不可能有 `ACK` 与 `SYN` 标志位。

```

1 //错误信息 说明文件名重传了，握手重传了，需要丢弃
2 if (name_flag == 1 && seq == -1 && r_temp0.isLAST()
3 || r_temp0.isACK() || r_temp0.isSYN())
4     continue;

```

### 4.2 解决阻塞次数过多的情况

程序设定的超时时间如果不恰当，小于路由器的延迟，会出现持续阻塞的情况。

因为这意味着发送方在发送数据后等待确认的时间，比数据在网络中传输所需的时间要短。如果网络延迟大于超时时间，发送方会在超时之前无法收到对应的确认信息，从而导致它错误地认为数据包丢失，并进行重传。

这样的情况会导致持续的阻塞，因为发送方不断重传数据，但由于网络延迟高于超时时间，接收方收到的数据和确认信息也无法及时返回给发送方。这种情况下，发送方会持续重传，增加了网络的负担，并且浪费了带宽，但实际上数据可能已经到达接收方。

解决问题的措施在前面的代码实现中已经提及，是检测一个分组重传次数，如果超过 *MISSCOUNT*，会上调程序的 *TIMEOUT* 时间。

下面是一个例子，验证了合理性



```
Microsoft Visual Studio 调试控制台
超时重传分组seq:1
超时重传分组seq:1
超时重传分组seq:1
超时重传分组seq:1
超时重传分组seq:1
超时重传分组seq:1
超时重传分组seq:1
超时重传分组seq:1
超时重传分组seq:1
超时重传分组seq:1
timeout修改为450ms
超时重传分组seq:1
超时重传分组seq:1
超时重传分组seq:1
微软拼音 半 :寸
```

图 4.8: 阻塞多次更改 *TIMEOUT*



```
超时重传分组seq:2
接受到确认ACK0, ack:3
发送分组seq:3, 校验和:16665
超时重传分组seq:3
接受到确认ACK1, ack:4
发送分组seq:4, 分组状态0, 校验和:26902
接受到确认ACK0, ack:5
发送分组seq:5, 校验和:55741
接受到确认ACK1, ack:6
发送分组seq:6, 分组状态0, 校验和:58692
微软拼音 半 :寸
```

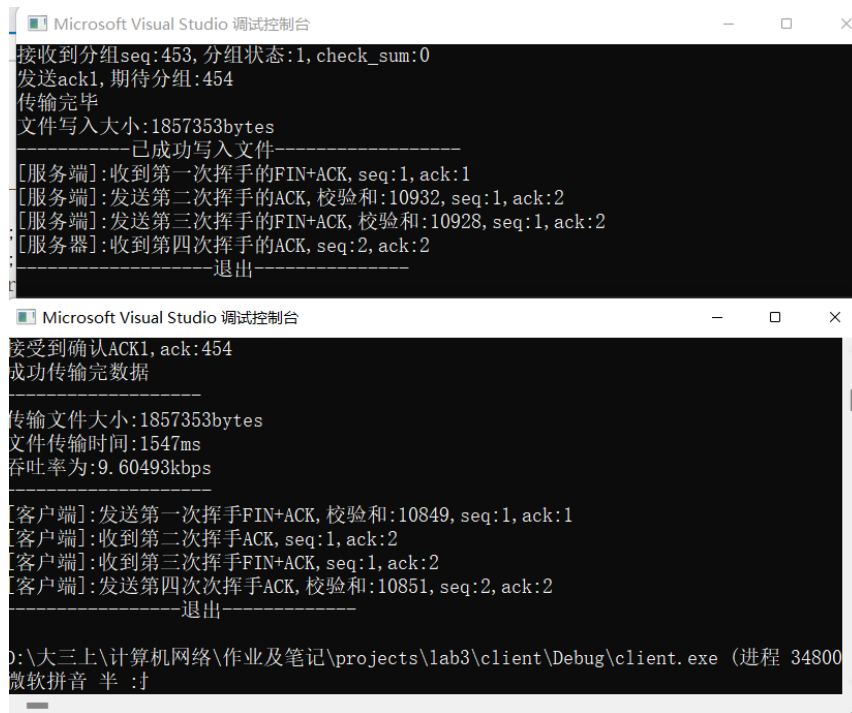
图 4.9: 解决了阻塞

## 5 实验结果

### 5.1 文件传输测试

这些测试有一定延时以及丢包，这里是证明文件传输正确。

1. 图片 1.jpg 测试



```

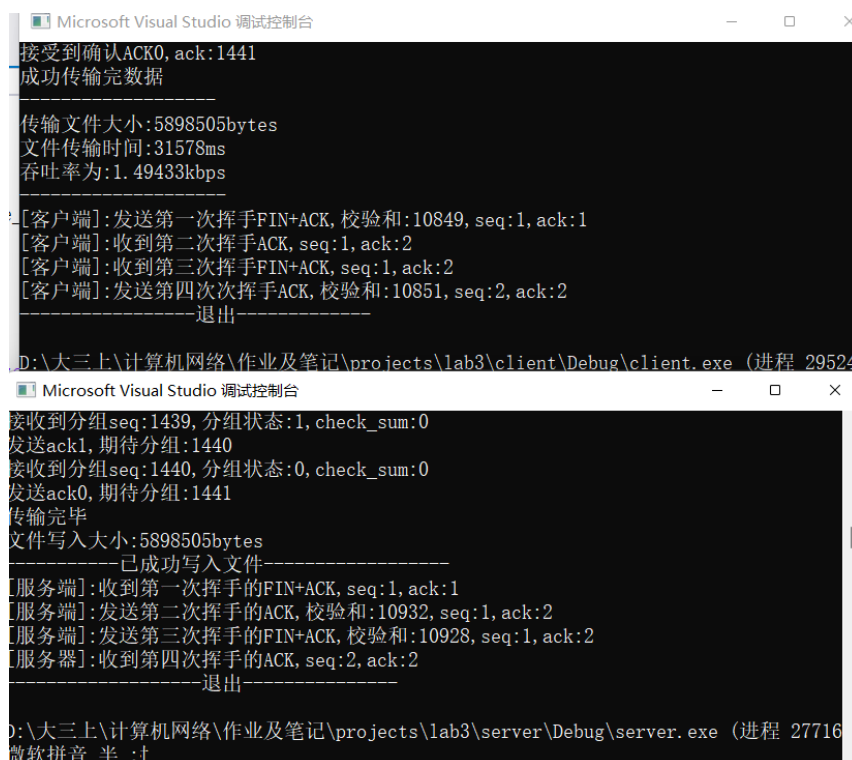
Microsoft Visual Studio 调试控制台
接收到分组seq:453, 分组状态:1, check_sum:0
发送ack1, 期待分组:454
传输完毕
文件写入大小:1857353bytes
-----已成功写入文件-----
[服务端]:收到第一次挥手的FIN+ACK, seq:1, ack:1
[服务端]:发送第二次挥手的ACK, 校验和:10932, seq:1, ack:2
[服务端]:发送第三次挥手的FIN+ACK, 校验和:10928, seq:1, ack:2
[服务器]:收到第四次挥手的ACK, seq:2, ack:2
-----退出-----

Microsoft Visual Studio 调试控制台
接受到确认ACK1, ack:454
成功传输完数据
-----
传输文件大小:1857353bytes
文件传输时间:1547ms
吞吐率为:9.60493kbps
-----
[客户端]:发送第一次挥手FIN+ACK, 校验和:10849, seq:1, ack:1
[客户端]:收到第二次挥手ACK, seq:1, ack:2
[客户端]:收到第三次挥手FIN+ACK, seq:1, ack:2
[客户端]:发送第四次挥手ACK, 校验和:10851, seq:2, ack:2
-----退出-----
D:\大三上\计算机网络\作业及笔记\projects\lab3\client\Debug\client.exe (进程 34800)
微软拼音 半:寸

```

图 5.10: 1.jpg 传输测试

## 2. 图片 2.jpg 测试



```

Microsoft Visual Studio 调试控制台
接受到确认ACK0, ack:1441
成功传输完数据
-----
传输文件大小:5898505bytes
文件传输时间:31578ms
吞吐率为:1.49433kbps
-----
[客户端]:发送第一次挥手FIN+ACK, 校验和:10849, seq:1, ack:1
[客户端]:收到第二次挥手ACK, seq:1, ack:2
[客户端]:收到第三次挥手FIN+ACK, seq:1, ack:2
[客户端]:发送第四次挥手ACK, 校验和:10851, seq:2, ack:2
-----退出-----
D:\大三上\计算机网络\作业及笔记\projects\lab3\client\Debug\client.exe (进程 29524)

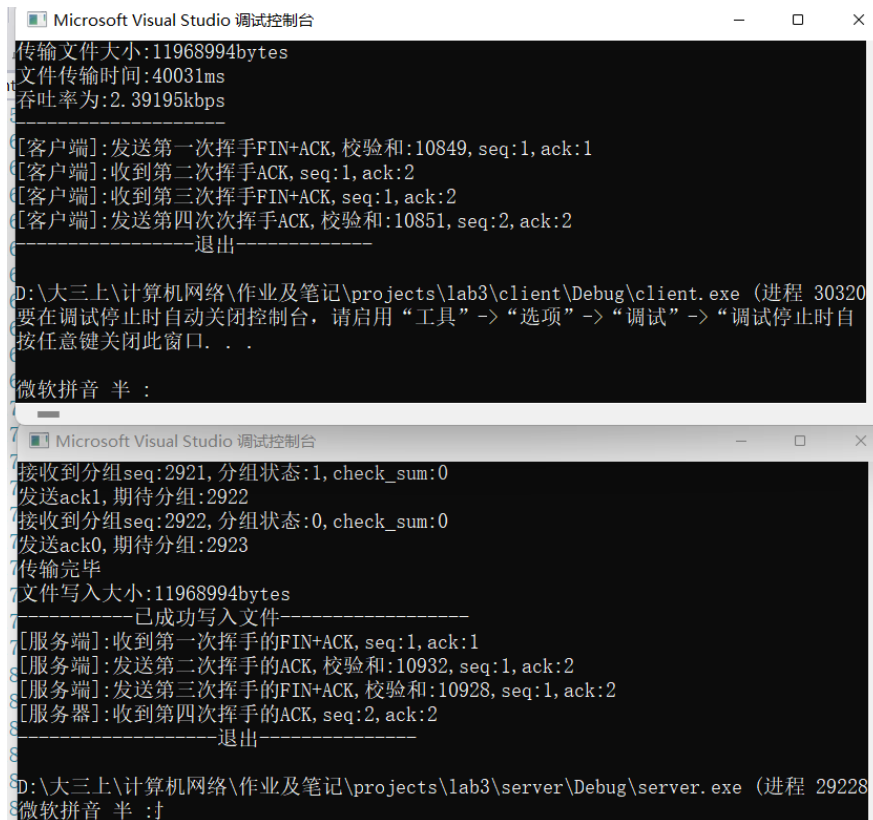
Microsoft Visual Studio 调试控制台
接收到分组seq:1439, 分组状态:1, check_sum:0
发送ack1, 期待分组:1440
接收到分组seq:1440, 分组状态:0, check_sum:0
发送ack0, 期待分组:1441
传输完毕
文件写入大小:5898505bytes
-----已成功写入文件-----
[服务端]:收到第一次挥手的FIN+ACK, seq:1, ack:1
[服务端]:发送第二次挥手的ACK, 校验和:10932, seq:1, ack:2
[服务端]:发送第三次挥手的FIN+ACK, 校验和:10928, seq:1, ack:2
[服务器]:收到第四次挥手的ACK, seq:2, ack:2
-----退出-----
D:\大三上\计算机网络\作业及笔记\projects\lab3\server\Debug\server.exe (进程 27716)
微软拼音 半:寸

```

图 5.11: 2.jpg 传输测试

## 3. 图片 3.jpg 测试





```

Microsoft Visual Studio 调试控制台
传输文件大小:11968994bytes
文件传输时间:40031ms
吞吐率为:2.39195kbps
-----
[客户端]:发送第一次挥手FIN+ACK, 校验和:10849, seq:1, ack:1
[客户端]:收到第二次挥手ACK, seq:1, ack:2
[客户端]:收到第三次挥手FIN+ACK, seq:1, ack:2
[客户端]:发送第四次挥手ACK, 校验和:10851, seq:2, ack:2
-----退出-----
D:\大三上\计算机网络\作业及笔记\projects\lab3\client\Debug\client.exe (进程 30320)
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自
按任意键关闭此窗口。...
微软拼音 半:

Microsoft Visual Studio 调试控制台
接收到分组seq:2921, 分组状态:1, check_sum:0
发送ack1, 期待分组:2922
接收到分组seq:2922, 分组状态:0, check_sum:0
发送ack0, 期待分组:2923
传输完毕
文件写入大小:11968994bytes
-----已成功写入文件-----
[服务端]:收到第一次挥手的FIN+ACK, seq:1, ack:1
[服务端]:发送第二次挥手的ACK, 校验和:10932, seq:1, ack:2
[服务端]:发送第三次挥手的FIN+ACK, 校验和:10928, seq:1, ack:2
[服务端]:收到第四次挥手的ACK, seq:2, ack:2
-----退出-----
D:\大三上\计算机网络\作业及笔记\projects\lab3\server\Debug\server.exe (进程 29228)
微软拼音 半:

```

图 5.12: 3.jpg 传输测试

## 4. helloworld.txt 测试



```

传输文件大小:1655808bytes
文件传输时间:5641ms
吞吐率为:2.34825kbps
-----
[客户端]:发送第一次挥手FIN+ACK, 校验和:10849, seq:1, ack:1
[客户端]:收到第二次挥手ACK, seq:1, ack:2
[客户端]:收到第三次挥手FIN+ACK, seq:1, ack:2
[客户端]:发送第四次挥手ACK, 校验和:10851, seq:2, ack:2
-----退出-----
Microsoft Visual Studio 调试控制台
发送ack0, 期待分组:405
传输完毕
文件写入大小:1655808bytes
-----已成功写入文件-----
[服务端]:收到第一次挥手的FIN+ACK, seq:1, ack:1
[服务端]:发送第二次挥手的ACK, 校验和:10932, seq:1, ack:2
[服务端]:发送第三次挥手的FIN+ACK, 校验和:10928, seq:1, ack:2
[服务端]:收到第四次挥手的ACK, seq:2, ack:2
-----退出-----

```

图 5.13: helloworld.txt 传输测试

## 5. 建立连接



```

Microsoft Visual Studio 调试控制台
请输入要传送的文件名:
helloworld.txt
成功初始化Socket DLL, 网络环境加载成功
数据报套接字创建成功
客户绑定端口和Ip成功
[客户端]:发送第一次握手SYN请求
[客户端]:第二次握手接受SYN+ACK
[客户端]:发送第三次握手ACK
[客户端]:三次握手连接结束
[客户端]:目的端口为:4001
-----开始传输文件名-----
发送分组seq:0, 校验和:12667

Microsoft Visual Studio 调试控制台
成功初始化Socket DLL, 网络环境加载成功
数据报套接字创建成功
服务器绑定端口和Ip成功
[服务端]:第一次握手, 接收到客户端SYN请求!
[服务端]:第二次握手, 发送SYN+ACK!
[服务端]:第三次握手, 收到ACK字段, 成功连接!
[服务端]:三次握手流程结束
[服务端]:目的端口为:4001
-----开始接受文件名-----
接收到分组seq:0, 分组状态:0, check_sum:0
发送ack0, 期待分组:1
传输完毕
helloworld.txt
微软拼音 半 :f

```

图 5.14: 建立连接测试

## 6. 断开连接

```

[客户端]:发送第一次挥手FIN+ACK, 校验和:10849, seq:1, ack:1
[客户端]:收到第二次挥手ACK, seq:1, ack:2
[客户端]:收到第三次挥手FIN+ACK, seq:1, ack:2
[客户端]:发送第四次挥手ACK, 校验和:10851, seq:2, ack:2
-----退出-----

Microsoft Visual Studio 调试控制台
-----已成功写入文件-----
[服务端]:收到第一次挥手的FIN+ACK, seq:1, ack:1
[服务端]:发送第二次挥手的ACK, 校验和:10932, seq:1, ack:2
[服务端]:发送第三次挥手的FIN+ACK, 校验和:10928, seq:1, ack:2
[服务端]:收到第四次挥手的ACK, seq:2, ack:2
-----退出-----

```

图 5.15: 断开连接测试

## 5.2 延时与丢包测试

### 1. 总体测试结果

丢包 50%, 延时 200ms 测试, 可以发现传输成功了。

由于我在程序中设定的 *TIMEOUT* 是 200ms, 而路由器延时为 200ms, 会造成不断重发一个分组, 至此会触发我的 *MISSCOUNT* 设置, 增大程序内部 *TIMEOUT*, 最终使得顺畅的传输, 而不是一直阻塞。



图 5.16: 丢包 50%, 延时 200ms 测试

由于丢包率 50%，最终四次挥手的第四次服务器不会收到确认，从图5.16得到了验证。

## 2. 细致探究

将延时设置为 300ms，由于我们的程序 *TIMEOUT* 为 200ms，会导致阻塞过多的情况。可以看到下面图5.17与图5.18，成功更改 *TIMEOUT*，解决了问题，最终只用重传 1 次即可，而更改之前重传次数超过 10 次。

图 5.17: 阻塞多次更改 *TIMEOUT*

```
超时重传分组seq:2
接受到确认ACK0, ack:3
发送分组seq:3, 校验和:16665
超时重传分组seq:3
接受到确认ACK1, ack:4
发送分组seq:4, 分组状态0, 校验和:26902
接受到确认ACK0, ack:5
发送分组seq:5, 校验和:55741
接受到确认ACK1, ack:6
发送分组seq:6, 分组状态0, 校验和:58692
微软拼音 半 :f
```

图 5.18: 解决了阻塞

相关原理:

程序设定的超时时间为 200ms，但是路由器的延迟为 300ms。

说明发送方在发送数据后等待确认的时间，比数据在网络中传输所需的时间要短。因此，如果网络延迟大于超时时间，发送方会在超时之前无法收到对应的确认信息，从而导致它错误地认为数据包丢失，并进行重传。

这样的情况会导致持续的阻塞，因为发送方不断重传数据，但由于网络延迟高于超时时间，接收方收到的数据和确认信息也无法及时返回给发送方。这种情况下，发送方会持续重传，增加了网络的负担，并且浪费了带宽，但实际上数据可能早已到达接收方。

解决这个问题的方法是根据实际的网络延迟来调整超时时间。我的解决方案是，调整 *TIMEOUT* 为原来的 1.5 倍，继续观测。