Lab3-3——基于UDP服务设计可靠传输协议并编程实现

学号：2212452
姓名：孟启轩
专业：计算机科学与技术

一、实验要求

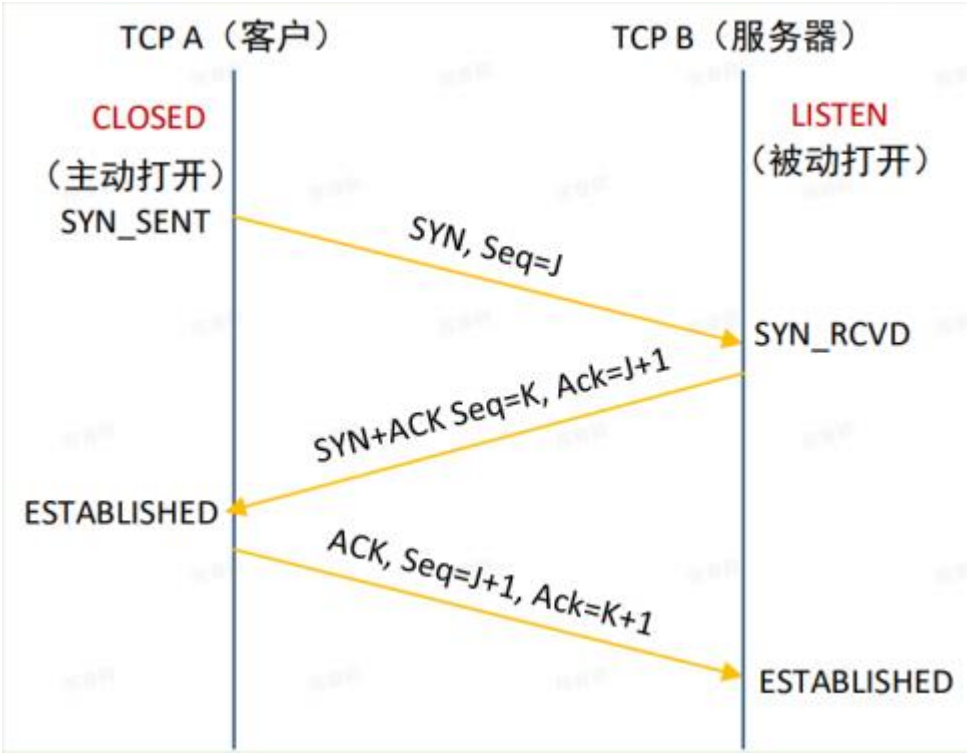
实验3-3：在实验3-2的基础上，选择实现一种拥塞控制算法，也可以是改进的算法，完成给定测试文件的传输。

二、实验设计

(一)原理探究

> 三次握手

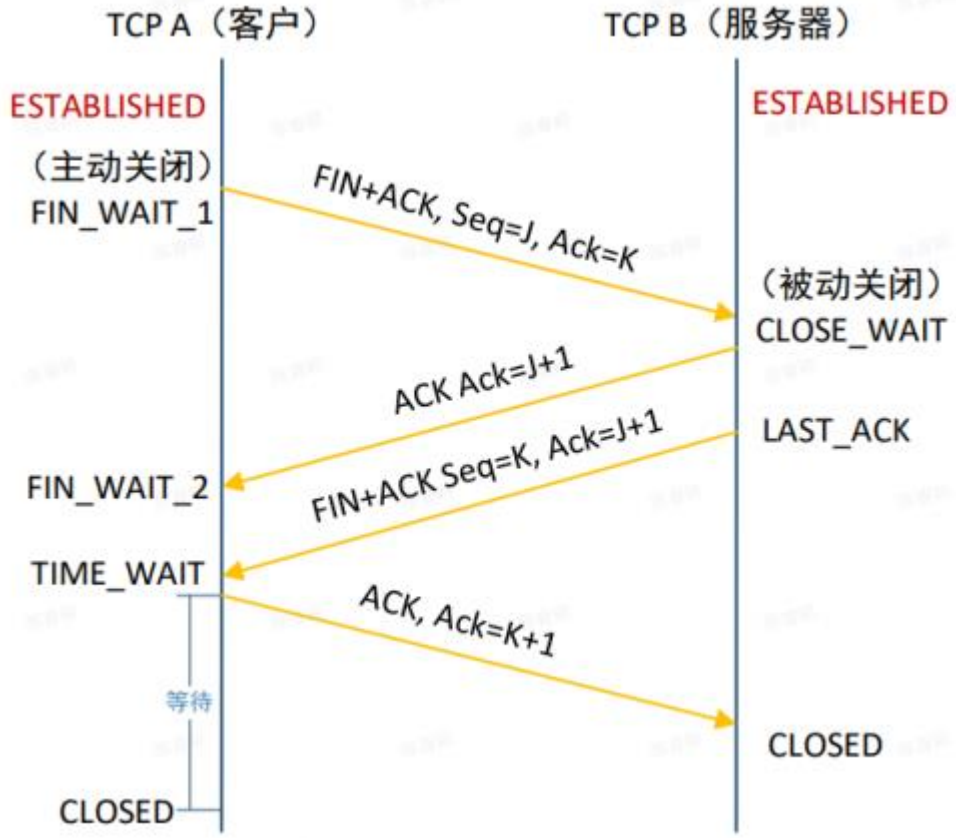
1. 第一次握手：客户端发送SYN包(seq=x)到服务器，并进入SYN_SEND状态，等待服务器确认
2. 第二次握手：服务器收到SYN包，必须确认客户的SYN(ack=x+1)，同时自己也发送一个SYN包(seq=y)，即SYN+ACK包，此时服务器进入SYN_RECV状态
3. 第三次握手：客户端收到服务器的SYN+ACK包，向服务器发送确认包ACK(ack=y+1)，此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手握手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数据。理想状态下，TCP连接一旦建立，在通信双方中的任何一方主动关闭连接之前，TCP连接都将被一直保持下去。



> 四次挥手

1. 第一次挥手：Client发送一个FIN，用来关闭Client到Server的数据传送，Client进入FIN_WAIT_1状态。
2. 第二次挥手：Server收到FIN后，发送一个ACK给Client，确认序号为收到序号+1（与SYN相同，一个FIN占用一个序号），Server进入CLOSE_WAIT状态。

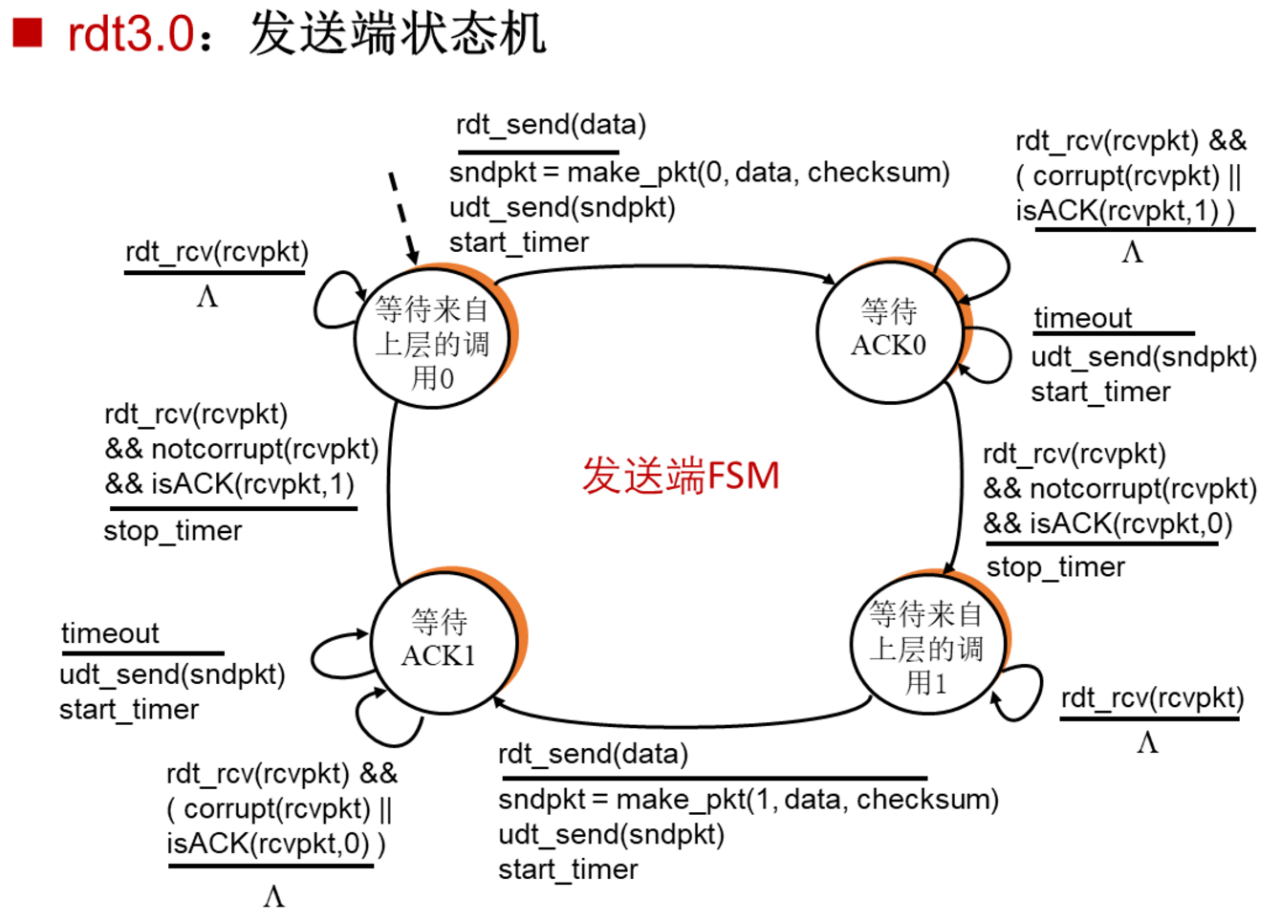
3. 第三次挥手：Server发送一个FIN，用来关闭Server到Client的数据传送，Server进入LAST ACK状态。
4. 第四次挥手：Client收到FIN后，Client进入TIME_WAIT状态，接着发送一个ACK给Server，确认序号为收到序号+1，Server进入CLOSED状态，完成四次挥手。



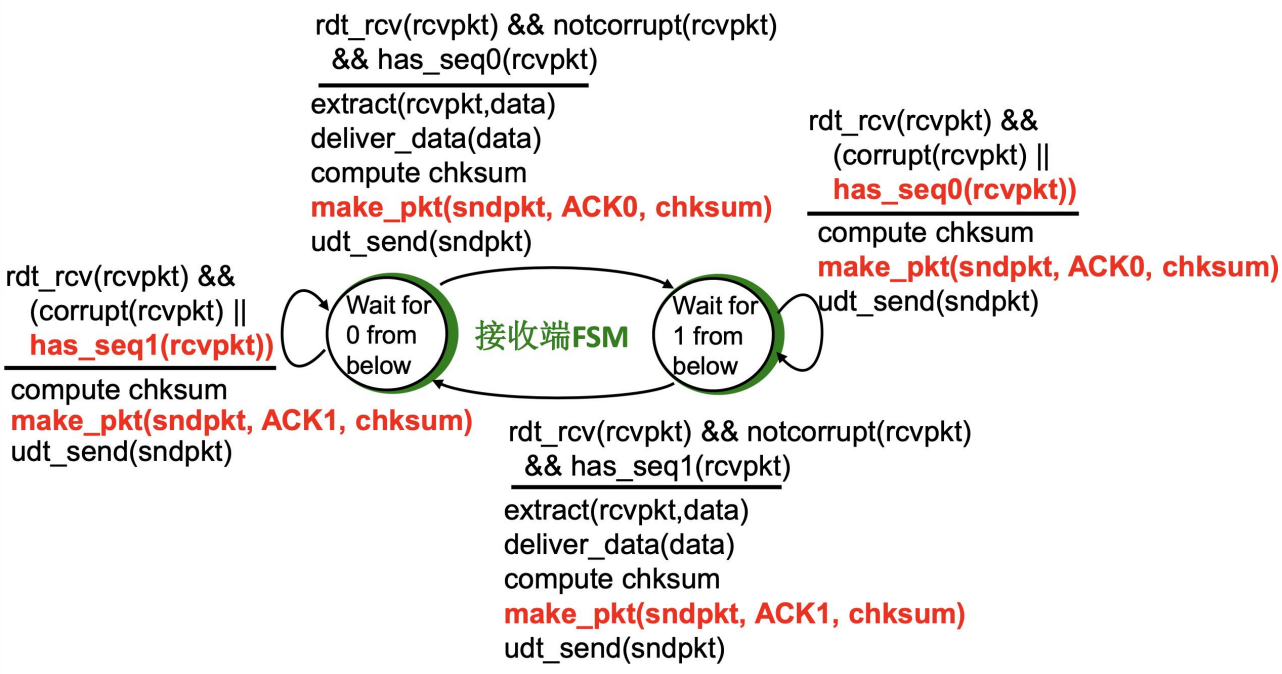
数据传输

发送端和接收端的接收机都采用rdt3.0的设计原则。

- 发送端的有限状态机：



- 接收端的有限状态机：



序列号与确认应答号

数据报的传输顺序通过序列号来保障，以确保数据传输的可靠性。每个响应包中也包含一个序列号，用于表明接收方已准备好处理对应的包。在传输数据报时，这个数据报会被放入重发队列，同时启动一个计时器。如果收到该数据包的确认信息，发送方会将此数据包从队列中删除。如果在计时器超时后仍未收到确认信息，则需

要重新发送该数据包。此外，通过数据分段中的序列号，可以确保所有传输的数据按正确的顺序进行重组，从而保证数据传输的完整性。

> 差错检测

差错检测（error detection），是指在发送的码序列（码字）中加入适当的冗余度以使得接收端能够发现传输中是否发生差错的技术。除了用于通信外，差错检测技术也广泛用于信息存储中。

> 超时重传

超时重传指的是在发送数据报文段后开始计时，设置一个等待确认应答到来的那个时间间隔。如果超过这个时间间隔，仍未收到对方发来的确认应答，发送端将进行数据重传。

> UDP协议

UDP是User Datagram Protocol的简称,中文名是用户数据报协议,是OSI参考模型中的传输层协议,它是一种无连接的传输层协议,提供面向事务的简单不可靠信息传送服务。

(二)协议设计

本程序中的相关设计如下：

> 三次握手建立连接

1. 客户端首先向服务端发送第一次握手请求(0x1)。
2. 服务端收到第一次握手请求后，向客户端发送第二次握手请求(0x2)。
3. 客户端收到第二次握手请求后，向服务端发送第三次握手请求(0x3)。
4. 服务端收到第三次握手请求后，握手成功，从而建立起通信连接。

> 四次挥手断开连接

1. 客户端首先向服务端发送第一次挥手请求(0x4)。
2. 服务端收到第一次挥手请求后，向客户端发送第二次挥手请求(0x5)。
3. 客户端收到第二次挥手请求后，挥手成功，通信正式断开

> 数据传输

- 客户端将文件数据划分为多个数据包，并通过UDP协议将这些数据包发送给服务器。每个数据包包含数据头部（包括数据大小、校验和、标志等信息）以及实际的文件数据。服务器在收到数据包后，会进行校验和确认，并根据数据包的标志进行相应的处理。
- 如果接收到的是正常的数据包，服务器会向客户端发送确认数据包。如果接收到的是最后一个数据包，服务器将发送一个带有结束标志（0x7）的确认数据包给客户端。

> 超时重传

- 在初始化函数中，设置了超时时间（MAX_WAIT_TIME）。
- 在Send_Message函数和初始化函数中，使用计时器来检测是否超时，并在超时后进行相应的重传操作。

> 差错检验

- 在Header类中，使用sum字段来存储校验和。
- 在check_sum函数中，实现了报文的校验和计算和设置。

- 而在Send_Message函数中，在构建发送的报文时计算校验和，并在接收端对收到的报文进行校验和的验证。

> 滑动窗口与GO BACK N协议

发送窗口大小大于1，而接收窗口大小为1。在这种协议下，发送方不需要在收到上一帧的ACK后才能发送下一帧。如果接收方检测到失序的帧，它会要求发送方重发最后一个正确帧之后的所有未确认帧；或者当发送方发送了N个帧后，如果发现前一个帧在计时器超时后仍未返回确认信息，则该帧被判定为出错或丢失，发送方将不得不重传该出错帧及之后的N个帧。

为了减少开销，Go Back N协议还规定，接收方不一定每收到一个正确的数据帧就必须立即发回一个确认，而是可以在连续收到多个正确的数据帧后，对最后一个数据帧发送确认信息，或者在有数据要发送时，对以前正确收到的帧进行捎带确认。

> 累积确认

累积确认是一种简单高效的ACK机制，主要特点是每个ACK都隐式确认了之前所有连续接收到的数据包。累积确认可以减少ACK数量，简化接收端设计，与GBN结合使用，帮助发送端管理重传。

> 拥塞控制（Reno算法）

RENO算法指的是TCP Reno，它是传输控制协议（TCP）的一种实现，专注于拥塞控制。TCP Reno在TCP Tahoe的基础上进行了改进，引入了更高效的拥塞控制机制，使得数据传输更加稳定和高效。

在后面·三、(二)TCP Reno 算法详解 中详细解释

三、设计实现

（一）关键变量与数据结构

- **拥塞窗口与阈值**：

```
int cwnd = 1;        // 拥塞窗口，单位为MSS
int ssthresh = 16;   // 慢启动阈值，单位为MSS
```

- **状态管理**：

```
#define Slow_Start 0
#define Congestion_Avoid 1
#define Quick_Recover 2
int state = Slow_Start;
```

- **发送与接收缓冲区**：

```
vector<Package *> GBN_BUFFER; // 存储待发送或已发送但未确认的包
```

- **计时器**:

```
struct Timer
{
    clock_t start;
    mutex mtx;
    void start_() { /* 启动计时 */ }
    bool is_time_out() { /* 判断是否超时 */ }
} timer;
```

（二）TCP Reno 算法详解

TCP Reno 算法是传输控制协议（TCP）的一种实现，专注于拥塞控制。TCP Reno在TCP Tahoe的基础上进行了改进，引入了更高效的拥塞控制机制，使得数据传输更加稳定和高效。

1. **慢启动（Slow Start）**
2. **拥塞避免（Congestion Avoidance）**
3. **快速恢复（Fast Recovery）**

其中，慢启动和拥塞避免是 TCP 的核心部分，负责调整发送速率，而快速恢复用于在特定情况下优化传输性能。

> MSS（Maximum Segment Size）：TCP 报文段的最大大小，通常由网络路径的 MTU（Maximum Transmission Unit）决定。

---

1. 慢启动

慢启动 旨在迅速探测网络的可用带宽。通过指数增长的方式快速增加发送速率，直到检测到网络拥塞或达到阈值 `ssthresh`。具体实现如下：

- **初始设置**:
  - 拥塞窗口（`cwnd`）初始化为 1 MSS（Maximum Segment Size）。

```
int cwnd = 1;         // 拥塞窗口，初始值为1 MSS
int ssthresh = 16;    // 慢启动阈值，初始值为16 MSS
int state = Slow_Start; // 当前状态，初始为慢启动
```

- **增长机制**:
  - 在 `Receive_Message` 函数中，当接收到一个有效的ACK时，如果当前状态为 `Slow_Start`，`cwnd` 增加1。

```
if (state == Slow_Start)
{
```

```
    cwnd++;
    if (cwnd >= ssthresh)
        state = Congestion_Avoid; // 达到阈值，进入拥塞避免阶段
}
```

  - 例如：
    - 初始 `cwnd = 1 MSS`，发送一个报文段，成功确认后，`cwnd = 2 MSS`。
    - 发送两个报文段，成功确认后，`cwnd = 4 MSS`。
    - 继续发送四个报文段，成功确认后，`cwnd = 8 MSS`。

- **增长特性**：
  - `cwnd` 以指数级别增长，每经过一个往返时间（RTT），发送速率翻倍。
- **终止条件**：
  - **超时（Timeout）**：
    - 若在慢启动过程中发生数据传输超时，设置慢启动阈值（`ssthresh`）为 `cwnd / 2`，将 `cwnd` 重置为 1 MSS，重新开始慢启动。
  - **达到阈值（ssthresh）**：
    - 当 `cwnd` 增加到等于或超过 `ssthresh` 时，停止指数增长，转入拥塞避免阶段。
  - **触发快速恢复**：
    - 若在慢启动阶段接收到三次重复 ACK（触发快速重传条件），进入快速恢复阶段，同时设置 `ssthresh = cwnd / 2`，并将 `cwnd` 设置为 `ssthresh + 3 MSS`。

---

2. 拥塞避免

拥塞避免 阶段，拥塞窗口 `cwnd` 以线性增长的方式增加，以避免过快地增加发送速率，从而减少网络拥塞的风险。具体实现如下：

- **增长机制**：
  - 在 `Receive_Message` 函数中，当状态为 `Congestion_Avoid` 时，`cwnd` 的增加是线性的：

```
else if (state == Congestion_Avoid)
{
    step++;
    if (step >= cwnd)
    {
        step = 0;
        cwnd++; // 每经过一个RTT，cwnd增加1
    }
}
```

  - 例如：
    - 如果 `cwnd = 10 MSS`，则每收到一个 ACK，`step`增加1，等`step>=cwnd`，`cwnd`增加1。（也就是每收到一个 ACK，`cwnd` 增加 `1/10 MSS`）
    - 在一个 RTT 内，收到 10 个 ACK 后，`cwnd` 增加 1 MSS。

- **增长特性**：
  - 线性增长避免了网络拥塞的风险，使 `cwnd` 稳步增加。
- **终止条件**：
  - **超时（Timeout）**：
    - 若在拥塞避免阶段发生超时，设置 `ssthresh = cwnd / 2`，将 `cwnd` 重置为 1 MSS，重新进入慢启动阶段。
  - **触发快速恢复**：
    - 若接收到三次重复 ACK，认为发生了拥塞，进入快速恢复阶段，同时设置 `ssthresh = cwnd / 2`，并将 `cwnd` 设置为 `ssthresh + 3 MSS`。

---

3. 快速恢复

快速恢复 阶段用于在检测到丢包后，迅速调整 `cwnd`，避免进入慢启动阶段，从而保持较高的传输速率。具体实现如下：

- **进入快速恢复**：
  - 当接收到的ACK的序列号小于当前 `first_pos` 时，认为是重复ACK，收到三次重复 ACK时，进入快速恢复阶段。

```
if (int(header.SEQ) < first_pos)
{
    dup++;
    if (dup == 3)
    {
        // 触发快速重传
    }
}
```

```
if (dup == 3)
{
    if (state == Slow_Start || state == Congestion_Avoid)
    {
        state = Quick_Recover;
        ssthresh = cwnd / 2;
        cwnd = ssthresh + 3;
    }
    Package *p = GBN_BUFFER[0];
    sendto(*socketClient, (char *)p, p->header.datasize + sizeof(HEADER), 0,
(sockaddr *)server_addr, server_addr_len);
    // 日志输出
}
```

  - 状态被设置为 `Quick_Recover`。
  - 设置 `ssthresh = cwnd / 2`。

    - 设置 `cwnd = ssthresh + 3 MSS`。
    - 重传第一个未确认的数据包 `GBN_BUFFER[0]`。
- **恢复机制**：
    - 在快速恢复期间，每收到一个新的 ACK，`cwnd` 增加 1 MSS，以便逐步恢复发送速率。

```
if (state == Quick_Recover)
    cwnd++;
```

- **退出快速恢复**：
    - **超时（Timeout）**：
        - 若在快速恢复过程中发生超时，设置 `ssthresh = cwnd / 2`，将 `cwnd` 重置为 1 MSS，重新进入慢启动阶段。
    - **收到新的 ACK**：
        - 一旦进入快速恢复阶段，并处理完所有重复ACK后，退出快速恢复，进入拥塞避免阶段，并将 `cwnd` 设置为 `ssthresh`：

```
else
{
    // 快速恢复结束
    cwnd = ssthresh;
    state = Congestion_Avoid;
    step = 0;
}
```

---

4. 超时处理

当发送的数据包在预定时间内未收到ACK时，认为包丢失，进行超时重传，并将 `ssthresh` 设置为当前 `cwnd` 的一半，将 `cwnd` 重置为1，重新进入慢启动阶段。

代码实现

- **检测超时**： 在 `Receive_Message` 函数中，如果计时器超时：

```
if (timer.is_time_out())
{
    // 超时处理
    dup = 0;
    ssthresh = cwnd / 2;
    cwnd = 1;
    state = Slow_Start;
    // 重传所有未确认的包
    for (auto package : GBN_BUFFER)
    {
        sendto(*socketClient, (char *)package, package->header.datasize +
sizeof(HEADER), 0,
```

```
                    (sockaddr *)server_addr, server_addr_len);
            // 日志输出
        }
        timer.start_();
    }
```

`ssthresh` 被设置为 `cwnd` 的一半，`cwnd` 被重置为1，状态回到 `Slow_Start`，并重传所有在 `GBN_BUFFER` 中未确认的包。

TCP Reno 通过结合**慢启动**、**拥塞避免**、和**快速恢复** 三大机制，使得Reno能够在不同网络条件下，动态调整传输速率，实现了高效且稳定的拥塞控制：

1. **慢启动**：快速探测网络带宽，`cwnd` 指数增长，直到检测到拥塞或达到阈值。
2. **拥塞避免**：以线性方式增长 `cwnd`，防止过快增加导致拥塞。
3. **快速恢复**：在丢包后迅速调整 `cwnd`，避免重新进入慢启动，保持较高的传输速率。

（三）发送与接收流程

发送流程 (`Send_Message` 函数)

1. **分段发送**：将文件内容分割为多个包，每个包的大小不超过 `BUFFER_SIZE`。
2. **拥塞窗口控制**：在发送每个包前，检查当前 `cwnd` 和 `MAX_WINDOW` 是否允许发送。如果窗口已满，发送线程会阻塞，等待ACK。
3. **发送数据包**：将数据包放入 `GBN_BUFFER`，发送到服务器，并在日志中记录发送情况。
4. **启动接收线程**：启动 `Receive_Message` 线程来接收ACK并调整 `cwnd`。
5. **发送结束标志**：发送所有数据包后，发送一个 `END` 标志包，并等待对方确认。

接收ACK流程 (`Receive_Message` 函数)

1. **接收ACK**：在非阻塞模式下接收ACK包。
2. **处理ACK**：
   - **有效ACK**：更新 `first_pos`，从 `GBN_BUFFER` 中移除已确认的包，调整 `cwnd`。
   - **重复ACK**：如果收到3个重复ACK，触发快速重传和快速恢复。
   - **超时检测**：如果超时，触发超时重传，重置 `cwnd` 和 `ssthresh`，重新进入慢启动。
3. **计时器重启**：每次收到有效ACK后，重启计时器以监控下一个包的ACK情况。

（四）多线程

上次实验采用单线程导致传输速度过慢，本次实验加以改进，采用多线程，加快传输速率。

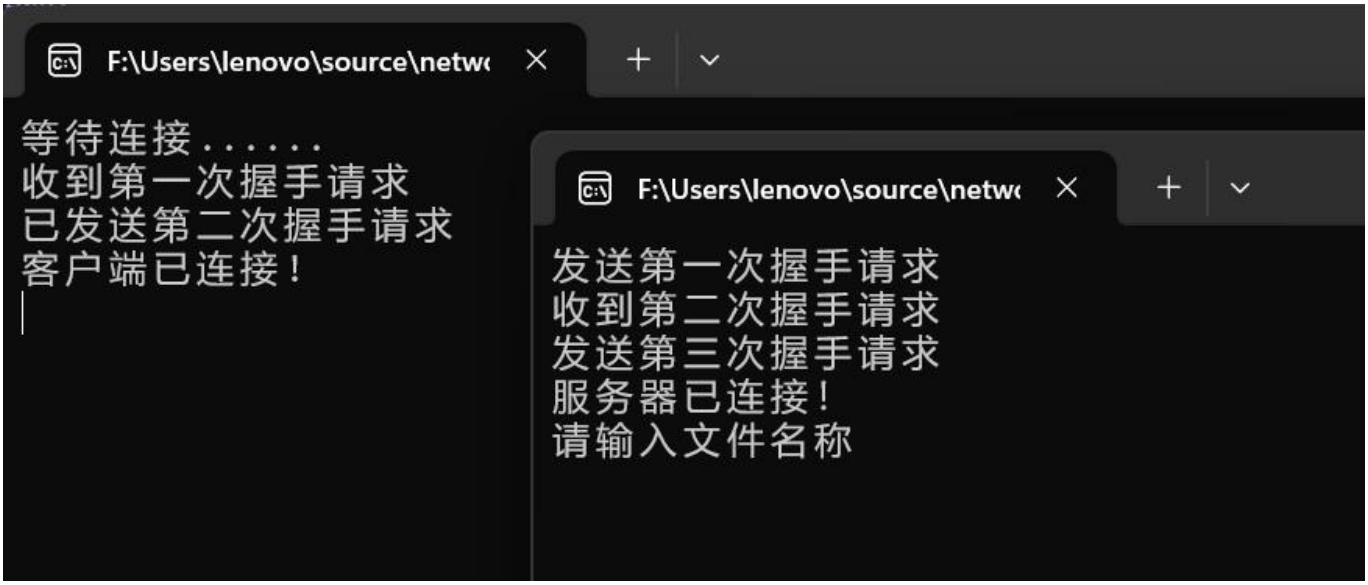
- 在 `Send_Message` 函数中，使用`thread Receive_Thread(Receive_Message, socketClient, server_addr);`启动 `Receive_Message` 函数作为一个独立的线程。
- 主线程负责发送数据包，而接收线程负责异步接收ACK并调整拥塞控制参数。
- 使用`mutex`互斥锁保护共享资源，确保线程之间的安全访问和操作。
- 总共两个线程：

- 主线程：负责发送数据和程序的主流程。(Send_Message)
- 接收线程：负责异步接收ACK并处理拥塞控制逻辑。(Receive_Message)

四、实验结果

(一)建立连接

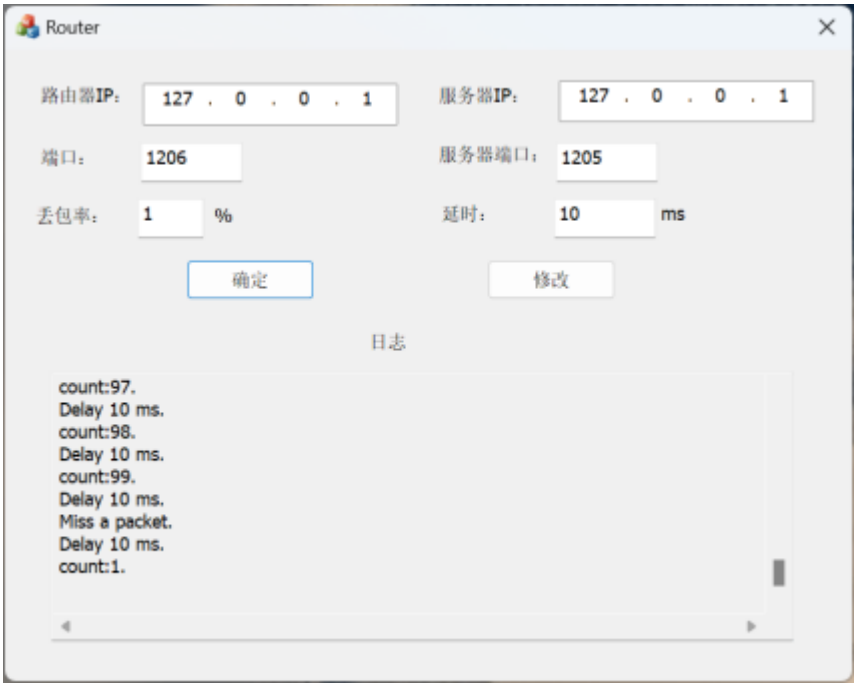
首先打开客户端和服务器端，握手建立连接如下图所示：



(二)数据传输

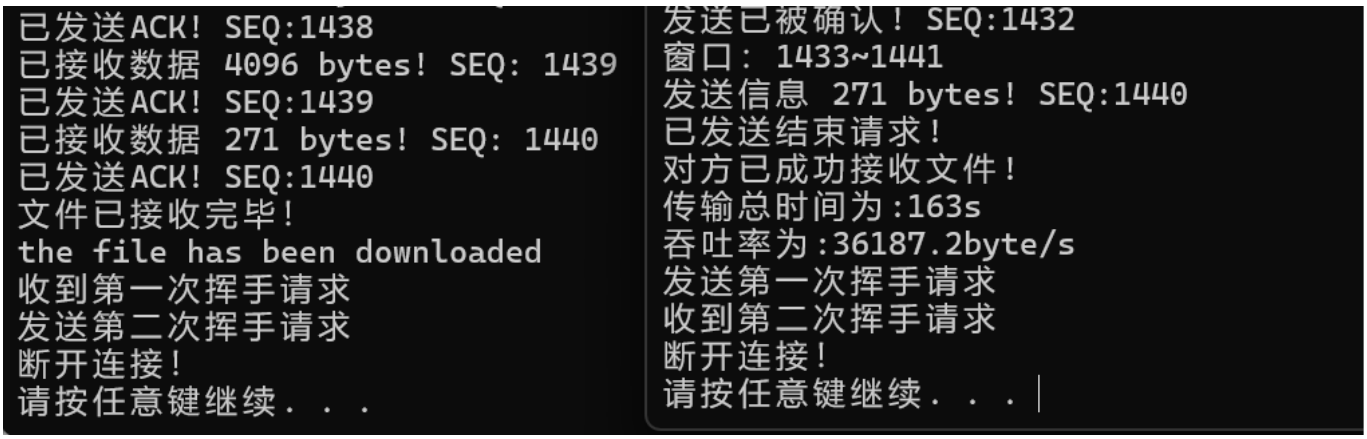
本次共需要传送四个文件，每个文件的传输及其传输后的总时间以及吞吐率如以下图片所示：

> 设置丢包

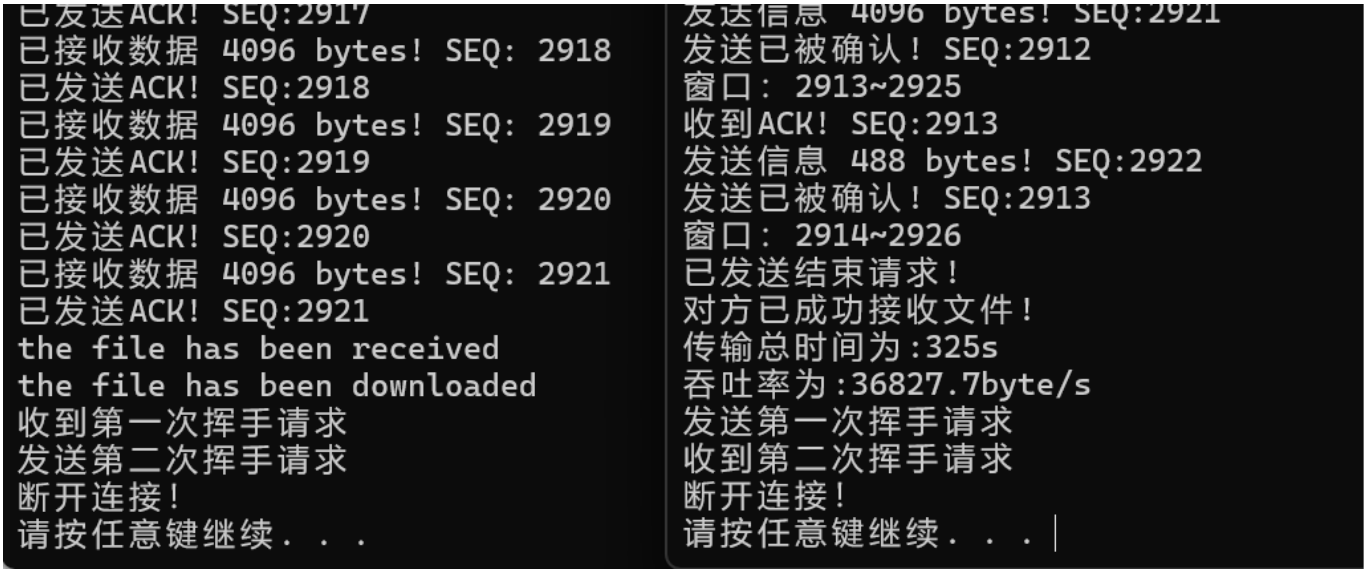


> 1.jpg

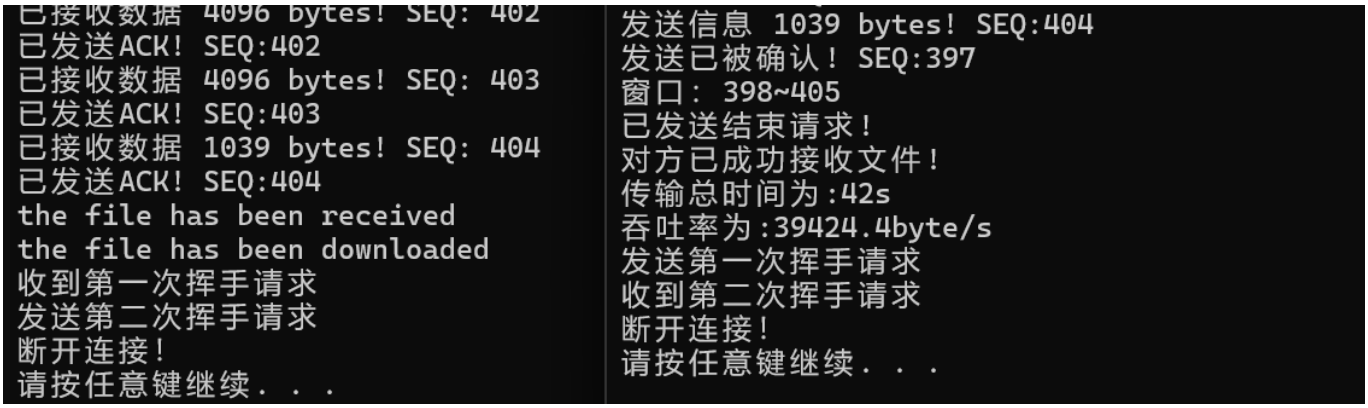
2.jpg



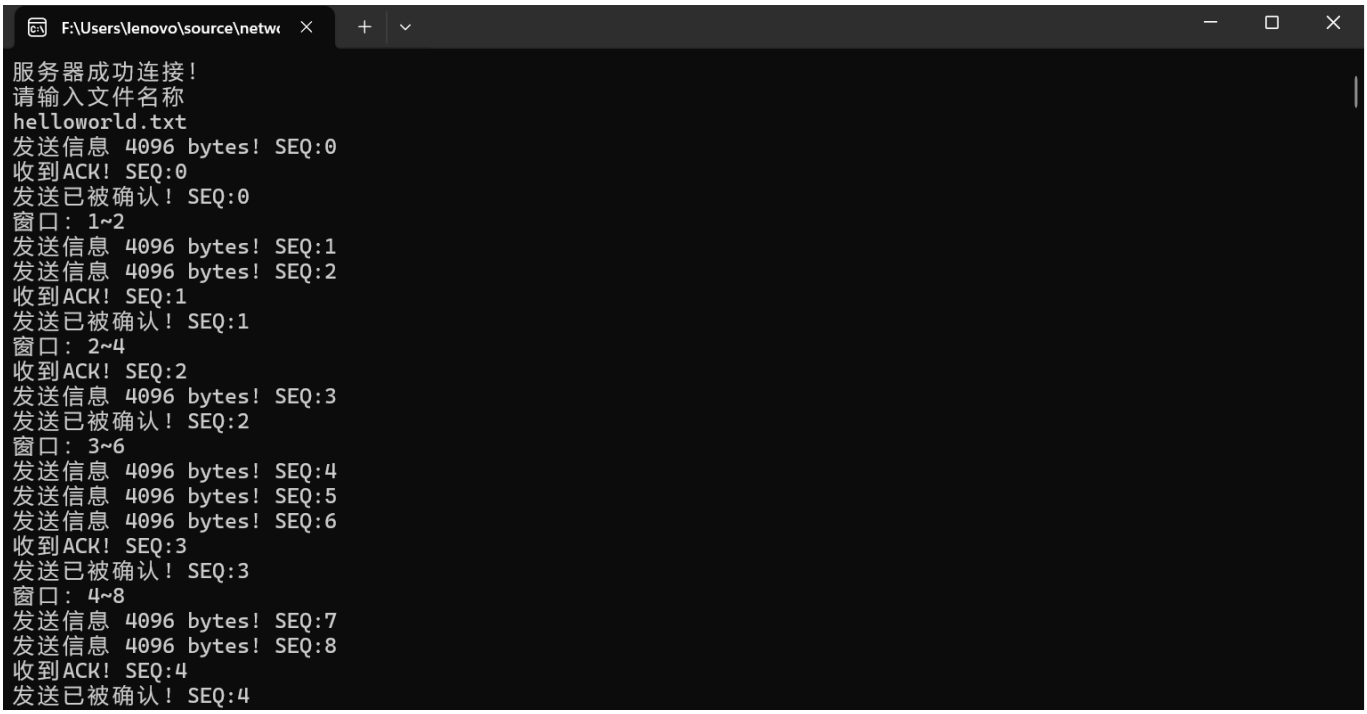
3.jpg

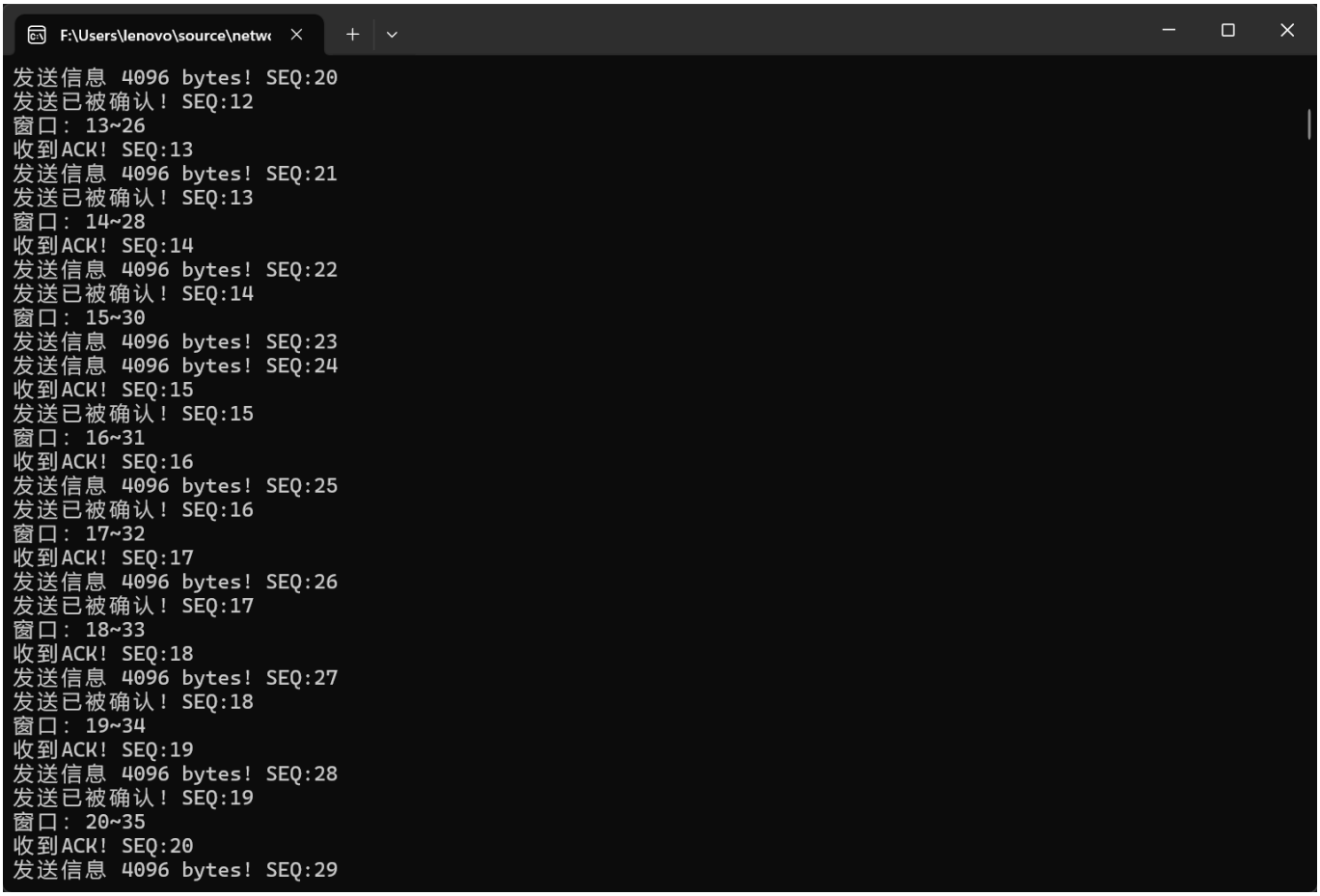


helloworld.txt

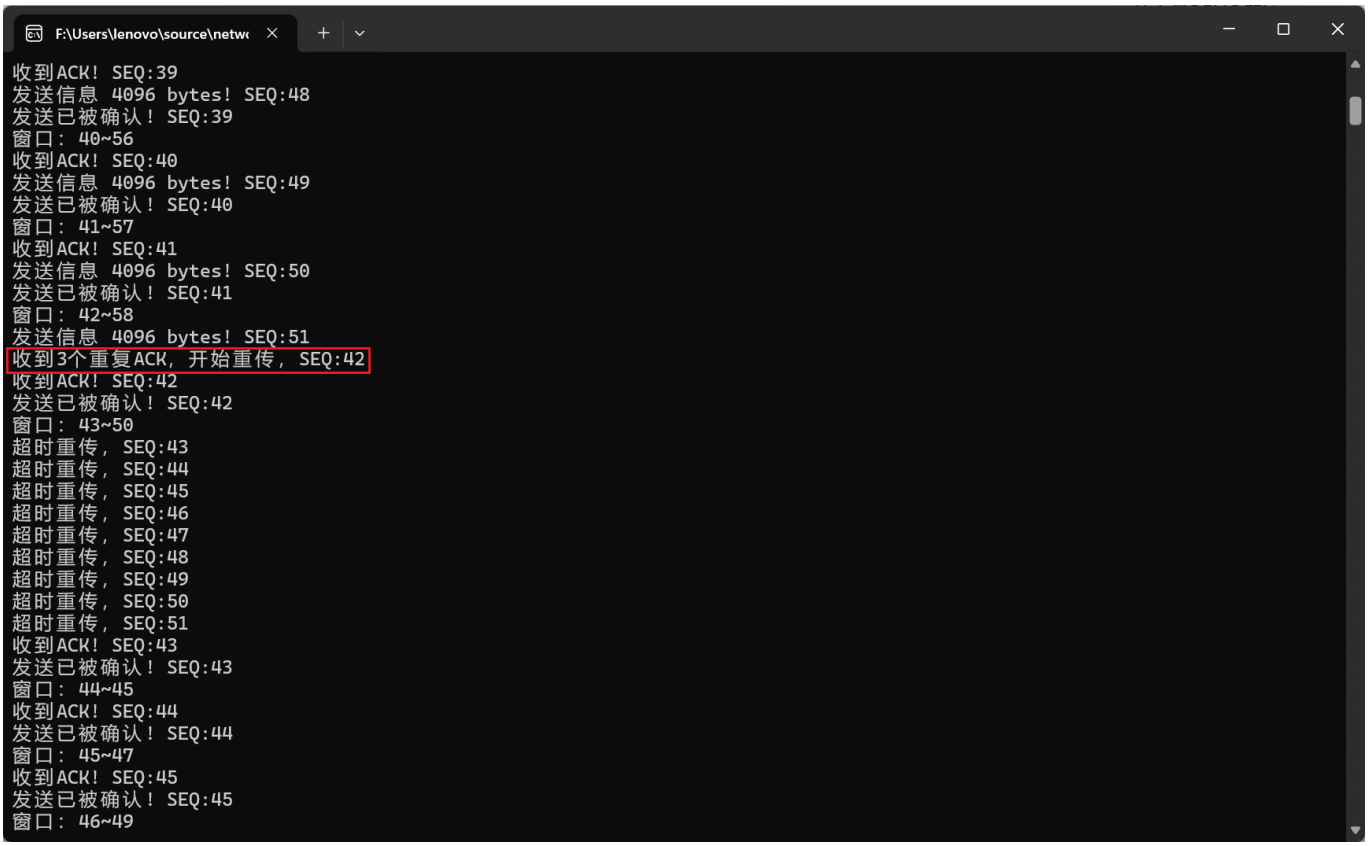

已接收数据 4096 bytes! SEQ: 402
已发送ACK! SEQ:402
已接收数据 4096 bytes! SEQ: 403
已发送ACK! SEQ:403
已接收数据 1039 bytes! SEQ: 404
已发送ACK! SEQ:404
the file has been received
the file has been downloaded
收到第一次挥手请求
发送第二次挥手请求
断开连接!
请按任意键继续. . .
发送信息 1039 bytes! SEQ:404
发送已被确认! SEQ:397
窗口: 398~405
已发送结束请求!
对方已成功接收文件!
传输总时间为:42s
吞吐率为:39424.4byte/s
发送第一次挥手请求
收到第二次挥手请求
断开连接!
请按任意键继续. . .

(三)窗口变化


F:\Users\lenovo\source\netw
服务器成功连接!
请输入文件名称
helloworld.txt
发送信息 4096 bytes! SEQ:0
收到ACK! SEQ:0
发送已被确认! SEQ:0
窗口: 1~2
发送信息 4096 bytes! SEQ:1
发送信息 4096 bytes! SEQ:2
收到ACK! SEQ:1
发送已被确认! SEQ:1
窗口: 2~4
收到ACK! SEQ:2
发送信息 4096 bytes! SEQ:3
发送已被确认! SEQ:2
窗口: 3~6
发送信息 4096 bytes! SEQ:4
发送信息 4096 bytes! SEQ:5
发送信息 4096 bytes! SEQ:6
收到ACK! SEQ:3
发送已被确认! SEQ:3
窗口: 4~8
发送信息 4096 bytes! SEQ:7
发送信息 4096 bytes! SEQ:8
收到ACK! SEQ:4
发送已被确认! SEQ:4

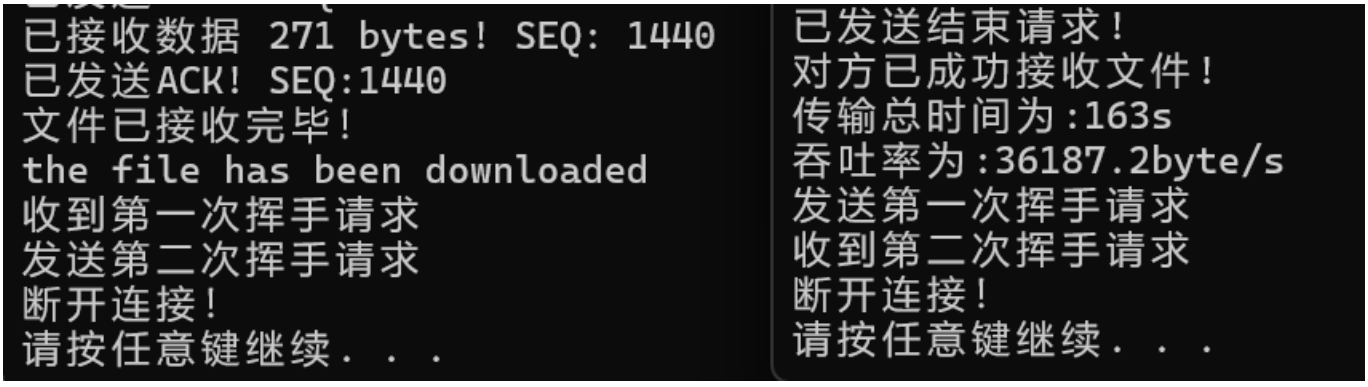

F:\Users\lenovo\source\netw
发送信息 4096 bytes! SEQ:20
发送已被确认! SEQ:12
窗口: 13~26
收到ACK! SEQ:13
发送信息 4096 bytes! SEQ:21
发送已被确认! SEQ:13
窗口: 14~28
收到ACK! SEQ:14
发送信息 4096 bytes! SEQ:22
发送已被确认! SEQ:14
窗口: 15~30
发送信息 4096 bytes! SEQ:23
发送信息 4096 bytes! SEQ:24
收到ACK! SEQ:15
发送已被确认! SEQ:15
窗口: 16~31
收到ACK! SEQ:16
发送信息 4096 bytes! SEQ:25
发送已被确认! SEQ:16
窗口: 17~32
收到ACK! SEQ:17
发送信息 4096 bytes! SEQ:26
发送已被确认! SEQ:17
窗口: 18~33
收到ACK! SEQ:18
发送信息 4096 bytes! SEQ:27
发送已被确认! SEQ:18
窗口: 19~34
收到ACK! SEQ:19
发送信息 4096 bytes! SEQ:28
发送已被确认! SEQ:19
窗口: 20~35
收到ACK! SEQ:20
发送信息 4096 bytes! SEQ:29

(四)三次重复ACK



(五)断开连接

- 挥手断开连接及文件的存储如下图所示:



(六)结果展示

接收到的文件:

可以看到四个文件都与原文件信息完全相同！

五、总结与感悟

在本次实验中，我基于实验3-2成功实现了TCP Reno拥塞控制算法，并将其集成到自定义的UDP传输协议中，以完成指定测试文件的可靠传输。通过实现慢启动、拥塞避免和快速恢复等关键机制，我能够有效地管理网络拥塞，优化数据传输的吞吐量。多线程设计使得数据发送与ACK接收能够并行进行，显著提升了传输效率。同时，在实现过程中，我深入理解了TCP拥塞控制的原理及其在实际应用中的挑战，如线程同步和精确计时的重要性。此次实验不仅巩固了我对网络协议的理论知识，还增强了我在实际编程中处理并发与错误管理的能力，展示了拥塞控制算法在提升网络通信性能中的关键作用。