南开大学
Nankai University

南　　开　　大　　学

计算机网络实验报告

实验 3-3

姓名：李雅帆

学号：2213041

年级：2022 级

专业：信息安全

2024 年 12 月 14 日

目录

一、 实验要求

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。

- 协议设计：数据包格式，发送端和接收端交互，详细完整
- 发送缓冲区、接收缓冲区
- RENO 算法或者自行设计其他拥塞控制算法
- 日志输出：收到/发送数据包的序号、ACK、校验和等，发送端和接收端的窗口大小等情况，传输时间与吞吐率
- 测试文件：必须使用助教发的测试文件（1.jpg、2.jpg、3.jpg、helloworld.txt）

二、 协议设计

(一) 拥塞控制

我在代码中采取 RENO 算法实现了基于 TCP 的拥塞控制，主要包括以下三种阶段：慢启动（Slow Start）、拥塞避免（Congestion Avoidance）和快速恢复（Fast Recovery）。

关键变量:

- cwnd：拥塞窗口大小，单位是 MSS（最大分节大小）。表示发送端当前可以发送的最大未确认数据包数。
- threshold：阈值，用于控制从慢启动阶段切换到拥塞避免阶段。
- status：拥塞控制状态，取值如下：
  - SS_STATUS：慢启动阶段（Slow Start）
  - CA_STATUS：拥塞避免阶段（Congestion Avoidance）
  - QR_STATUS：快速恢复阶段（Quick Recovery）
- dup_ack_times: 记录重复 ACK 的次数，用于检测丢包。
- TIMEOUT: 超时时间（单位：ms），用于判断是否发生丢包。

1. 慢启动阶段（Slow Start）

   - **进入条件**
     - 初始状态（cwnd = 1）。
     - 或者在超时后重置拥塞窗口进入慢启动阶段:

```
threshold = cwnd / 2;
cwnd = 1; // 拥塞窗口大小重置为 1
status = SS_STATUS; // 切换到慢启动状态
```

   - **增长机制**

     在慢启动阶段，窗口大小 cwnd 指数增长:

```
if (status == SS_STATUS) {
    cwnd++; // 每次收到 ACK, 窗口大小翻倍增长
    if (cwnd > threshold) {
        status = CA_STATUS; // 超过阈值后进入拥塞避免阶段
    }
}
```

     - 每收到一个 ACK，将 cwnd 增加 1 MSS:
     - 指数增长：每个成功的 ACK 会使 cwnd 增加 1 MSS，因此窗口会快速扩大。
     - 切换条件：当 cwnd > threshold 时，进入拥塞避免阶段。
     - 特点：指数增长，适用于网络拥塞未知的情况，快速探测网络容量。

2. 拥塞避免阶段（Congestion Avoidance）

- **进入条件** 当 cwnd 增长到超过 threshold 时，慢启动阶段切换到拥塞避免阶段：

```
if (cwnd > threshold) {
    status = CA_STATUS;
}
```

- **增长机制**

  在拥塞避免阶段，cwnd 增长速度变慢，采用线性增长：

```
if (status == CA_STATUS) {
    cwnd += 1 / ((cwnd > 1) ? floor(cwnd) : 1); // 增长速率减缓
}
```

  - 线性增长：每个 ACK 都会使 cwnd 增加一小部分：

$$\text{增加值} = \frac{1}{\lfloor \text{cwnd} \rfloor}$$

    如果当前 cwnd 是较大的值，则每次增加的部分会较小。

    每个 RTT（ACK 全部返回）后，cwnd 增加约 1 MSS，增长速率较慢。
  - 稳定传输：线性增长适合稳定的网络状态，避免网络拥塞。

3. 快速恢复阶段（QR_STATUS）

- **进入条件** 当检测到三次重复 ACK（未发生超时）时，进入快速恢复阶段：

```
if (status != QR_STATUS && dup_ack_times == 3) {
    status = QR_STATUS; // 切换到快速恢复阶段
    threshold = cwnd / 2; // 阈值减半
    cwnd = threshold + 3; // 拥塞窗口设置为阈值加 3 MSS
    resend = true; // 重传丢失的数据包
}
```

- **调整逻辑**
  - 将 threshold 设置为当前窗口大小的一半。
  - 设置 cwnd = threshold + 3 MSS，以适度增加窗口，快速恢复传输能力。
  - 重传丢失的数据包。
- **增长机制**

  在快速恢复阶段，每次收到重复 ACK，窗口继续增加：

```
if (status == QR_STATUS) {
    cwnd++; // 每次收到重复 ACK, 窗口增加 1 MSS
}
```

- **退出条件**

  当收到新的 ACK（非重复 ACK）时，快速恢复阶段结束，重新进入拥塞避免阶段：

```
if (status == QR_STATUS) {
    status = CA_STATUS; // 切换回拥塞避免阶段
    cwnd = threshold; // 恢复窗口大小为阈值
}
```

(二) 超时重传机制

超时重传是确保可靠数据传输的重要手段。当发送方在一定时间内未收到接收方的确认（ACK）时，假定对应的数据包可能丢失，触发重传机制。重传机制结合定时器用于判断数据是否超时，从而实现对数据丢失的自动修复。

1. 触发条件

- 超时重传机制的触发条件是在指定时间内未收到 ACK。超时时间由变量 TIMEOUT 定义：

```
#define TIMEOUT 100 // 超时时间，单位：毫秒
```

- 在定时器线程中，程序通过对 start 和 last 的时间差进行判断，如果超出 TIMEOUT 则触发超时处理：

```
if (last - start >= TIMEOUT) {
    // 触发超时处理
    threshold = cwnd / 2; // 阈值减半
    cwnd = 1; // 拥塞窗口重置为 1
    status = SS_STATUS; // 进入慢启动阶段
    dup_ack_times = 0; // 重置重复 ACK 计数
    resend = true; // 标记需要重传
    cout << "[超时]" << endl;
}
```

2. 拥塞控制调整

- 调整 threshold（阈值），将当前拥塞窗口大小 cwnd 减半。

```
threshold = cwnd / 2;
```

- 重置 cwnd（拥塞窗口大小），将 cwnd 重置为 1 MSS。

```
cwnd = 1;
```

- 重置状态，将状态切换为慢启动阶段。

```
status = SS_STATUS;
```

- 重置重复 ACK 计数，清零重复 ACK 的计数器，避免错误判断。

```
dup_ack_times = 0;
```

3. 重传逻辑

- 当超时触发后，程序会标记 resend 为 true，进入重传逻辑：

```
resend = true;
```

- 在发送文件的主循环中，检测到 resend 标记时，重新发送窗口中所有未确认的数据包：

```
if (resend) {
    // 根据超时重新计算发送起点
    seq = base; // 将序号从未确认的起点重新开始
    resend = false; // 清除重传标记
    continue; // 跳过当前循环，重新发送
}
```

(三) 滑动窗口实现

1. 发送端滑动窗口实现

- **关键变量**
  - base: 滑动窗口的起始位置，表示第一个未确认的分组序号。
  - cwnd: 拥塞窗口大小，控制可以发送的分组数量（单位为 MSS）。
  - seq: 当前要发送的分组序号。
  - ACK: 用于滑动窗口确认接收端的序号并调整 base。
- **滑动窗口范围**

  窗口内允许发送的分组序号范围是:

```
[base, base + cwnd - 1]
```

  只有当 seq < base + cwnd 时，才能发送新的分组。
- **窗口滑动机制**

  当发送端接收到接收端的 ACK 时，更新 base:

```
if (recvBuf[FLAG_BIT_POSITION] == 0b100 && ack_opp >= base) {
    base = ack_opp + 1; // 滑动窗口的起点向前移动
}
```

- **滑动窗口大小变化**
  - 发送端窗口大小 cwnd 的定义：cwnd 的单位是 MSS（最大分节大小，即 DATASIZE）。表示发送端当前允许未确认的最大数据包数。
  - 窗口的实际大小为:

```
窗口范围 = [base, base + cwnd - 1]
```

    其中 base 是窗口的起始位置，表示第一个未确认的分组序号，cwnd 是窗口的大小，动态变化。
  - 慢启动阶段，cwnd 从 1 MSS 开始，每收到一个正确的 ACK，cwnd 增加 1 MSS，指数增长，直到达到 threshold 或发生丢包。

```
if (status == SS_STATUS) {
    cwnd++; // 每次收到 ACK，窗口大小翻倍增长
    if (cwnd > threshold) {
        status = CA_STATUS; // 超过阈值后进入拥塞避免阶段
    }
}
```

- 拥塞避免阶段 cwnd 增加缓慢，每轮收到的 ACK 增加 1 / cwnd MSS。

```
if (status == CA_STATUS) {
    cwnd += 1 / ((cwnd > 1) ? floor(cwnd) : 1); // 线性增长
}
```

- 快速恢复阶段将 cwnd 减半，设为 threshold + 3 MSS, 重传丢失的数据包后，每次收到重复 ACK，cwnd 增加 1 MSS, 恢复后进入拥塞避免阶段。

```
if (status != QR_STATUS && dup_ack_times == 3) {
    status = QR_STATUS; // 进入快速恢复
    threshold = cwnd / 2;
    cwnd = threshold + 3;
    resend = true; // 重传丢失的分组
}
```

- 超时重传阶段, 将 threshold 设为 cwnd / 2, 将 cwnd 重置为 1 MSS, 重新进入慢启动阶段。

```
if (last - start >= TIMEOUT) {
    threshold = cwnd / 2;
    cwnd = 1;
    status = SS_STATUS; // 回到慢启动阶段
    resend = true;
}
```

- **发送逻辑**

  发送数据时检查当前分组是否在窗口范围内，如果窗口已满（seq >= base + cwnd），发送端会等待窗口滑动，即等待接收到新的 ACK。

```
if (seq < base + cwnd) {
    // 生成数据分组
    header[SEQ_BITS_START] = (u_char)(seq & 0xFF);
    header[SEQ_BITS_START + 1] = (u_char)(seq >> 8);
    // 将数据部分填充到分组中
    memcpy(sendBuf + HEADERSIZE, dataSegment, sendSize - HEADERSIZE);

    // 发送分组
    sendto(sendSocket, sendBuf, sendSize, 0, (SOCKADDR*)&recvAddr, sizeof(
        SOCKADDR));
    seq++; // 更新发送序号
}
```

2. 接收端滑动窗口实现

- **关键变量**
  - expectedSeq: 接收端期望接收到的分组序号。
  - ACK: 接收端回复给发送端的确认序号。

- **滑动窗口范围**

  接收端的滑动窗口大小为 1，即接收端一次只能接收一个按序到达的数据包。接收端的窗口范围是单一序号，即期望接收到的 expectedSeq。如果收到的数据序号与 expectedSeq 不一致，接收端直接丢弃数据包，并重发上一次的 ACK。

```
if (seq_opp == expectedSeq) {
    // 正确的数据包
    memcpy(dataSegment, recvBuf + HEADERSIZE, dataLength);
    out.write(dataSegment, dataLength); // 写入文件

    // 更新期望序号
    expectedSeq++;

    // 发送 ACK
    header[ACK_BITS_START] = (u_char)(expectedSeq & 0xFF);
    header[ACK_BITS_START + 1] = (u_char)(expectedSeq >> 8);
    sendto(recvSocket, header, HEADERSIZE, 0, (SOCKADDR*)&sendAddr, sizeof(
        SOCKADDR));
}
```

- **丢弃逻辑**

  如果接收到的分组序号与 expectedSeq 不匹配，接收端不会接收乱序到达的分组，从而确保数据的有序性。

```
if (seq_opp != expectedSeq) {
    // 重发上一次的 ACK
    sendto(recvSocket, header, HEADERSIZE, 0, (SOCKADDR*)&sendAddr, sizeof(
        SOCKADDR));
}
```

3. 滑动窗口流程总结

- **发送端**
  - 在窗口范围内发送数据。
  - 等待接收端的 ACK，确认已发送的数据。
  - 根据 ACK 滑动窗口的起点 base。
  - 处理超时或重复 ACK 时触发的重传。
- **接收端**
  - 接收分组并检查序号。
  - 如果序号正确，处理数据并发送对应的 ACK。
  - 如果序号不正确，丢弃分组并重发上一次的 ACK。

（四） 差错检验

本实验的差错检验采用经典的 16 位校验和算法（Checksum）。在数据传输过程中，使用校验和对数据包进行完整性检查，接收端通过验证校验和判断数据是否被篡改或损坏。

1. 校验和计算的原理。

   - 将数据划分为若干个 16 位块（2 字节）。
   - 将所有 16 位块逐块相加，产生一个 16 位的累加和。
   - 如果累加和溢出（超过 16 位），则将高位的溢出部分加回到低位。
   - 对累加和取反，得到校验和。
   - 在数据传输时，将计算得到的校验和附加到报文中。
   - 接收方重新计算收到数据的校验和。
   - 将接收到的校验和与计算结果相加，验证总和是否为 0xFFFF。
   - 若结果为 0xFFFF，说明数据完整；否则认为数据损坏。

   cksum 函数来实现 16 位校验和的计算：

```
checksum = checkSum(sendBuf, sendSize);
header[CHECKSUM_BITS_START] = sendBuf[CHECKSUM_BITS_START] = checksum & 0xFF;
header[CHECKSUM_BITS_START + 1] = sendBuf[CHECKSUM_BITS_START + 1] = checksum
    >> 8;
```

   发送端设置校验和: 在发送数据包前，计算校验和并存储在头部。

```
sendto(sendSocket, sendBuf, sendSize, 0, (SOCKADDR*)&recvAddr, sizeof(
    SOCKADDR));
```

   接收端验证校验和: 接收端收到数据包后，重新计算校验和进行校验。

```
if (checksum == 0) {
    // 数据包正确
    memcpy(dataSegment, recvBuf + HEADERSIZE, dataLength);
} else {
    // 校验和错误
    cout << "校验和错误" << endl;
    continue;
}
```

2. 差错检验的使用场景

   - 三次握手和四次挥手中的校验: 在三次握手和四次挥手过程中，每个报文都附带校验和，用于检测控制报文的完整性。
   - 数据传输中的校验: 在文件数据包的传输过程中，每个数据包都计算校验和并附加到报文头部, 接收方对收到的数据包进行校验，若校验失败则丢弃该包并等待重传。

3. 差错处理

   如果校验和不匹配：丢弃数据包, 不发送 ACK, 等待发送端的重传。

   - 未收到 ACK 时的超时处理: 如果接收端未发送 ACK（因为数据包校验失败被丢弃），发送端在超时时间后会触发重传机制。

```
if (last - start >= TIMEOUT) {
    threshold = cwnd / 2; // 阈值减半
    cwnd = 1; // 拥塞窗口重置为 1
    status = SS_STATUS; // 切换到慢启动阶段
    resend = true; // 触发重传
}
```

- 收到重复 ACK: 如果接收端收到的数据乱序（可能是校验失败导致丢失的包未被确认），会重复发送上一个正确数据包的 ACK，发送端检测到重复的 ACK 后可能触发快速恢复。

```
if (dup_ack_times == 3) {
    status = QR_STATUS; // 快速恢复阶段
    threshold = cwnd / 2;
    cwnd = threshold + 3;
    resend = true; // 触发丢包的重传
}
```

（五） 三次握手建立连接

1. 客户端发送 SYN：客户端向服务器发起连接请求。
2. 服务器返回 SYN+ACK：服务器接收到请求后进行响应，表示可以建立连接。
3. 客户端发送 ACK：客户端接收服务器响应后发送确认报文，完成连接的建立。

（六） 四次挥手断开连接

1. 第一次挥手：客户端发送 FIN 请求断开连接。
2. 第二次挥手：服务器接收 FIN，并返回 ACK 确认。
3. 第三次挥手：服务器发送 FIN+ACK 请求断开连接。
4. 第四次挥手：客户端接收 FIN+ACK，并返回 ACK 确认。

三、 实现流程

(一) 多线程的实现

1. 发送端

   - 接收响应线程：recvRespondThread，用于接收 ACK 包。
   - 定时器线程：timerThread，用于超时检测。

```
thread recvRespond(recvRespondThread);
recvRespond.detach();

thread timer(timerThread);
timer.detach();
```

2. 客户端

   - 文件接收线程负责循环接收数据包。

```
thread recvfile_thread(recvfile);
recvfile_thread.join();
```

(二) 初始化与套接字建立

1. 发送端（客户端）流程

- **初始化网络环境**:
  - 调用 `WSAStartup` 函数，初始化网络环境。
  - 如果初始化失败，输出错误信息并退出程序。
- **创建套接字**:
  - 使用 `socket` 函数创建一个 UDP 套接字。
  - 如果套接字创建失败，输出错误信息并退出程序。
- **配置目标地址**:
  - 配置目标服务器的 IP 地址和端口号。

2. 接收端（服务器端）流程

- **初始化网络环境**:
  - 调用 `WSAStartup` 函数，初始化网络环境。
  - 如果初始化失败，输出错误信息并退出程序。
- **创建套接字**:
  - 使用 `socket` 函数创建一个 UDP 套接字。

  - 如果套接字创建失败，输出错误信息并退出程序。

- **绑定套接字**:
  - 使用 `bind` 函数绑定本地的 IP 地址和端口号。
  - 如果绑定失败，输出错误信息并退出程序。

(三) 握手连接（三次握手）

1. 发送端（客户端）流程

- **第一次握手**:
  - 客户端向服务器发送一个 SYN 包，请求建立连接。
  - 包中包含一个初始序号 `seq`，并设置标志位为 SYN。
  - 客户端计算并填充校验和，确保包的完整性。
  - 发送包到服务器，等待服务器响应。
- **第二次握手**:
  - 客户端接收服务器返回的 SYN+ACK 包，提取服务器的初始序号 `seq_opp` 并验证 ACK 是否为 `seq+1`。
  - 如果收到的包标志位为 SYN+ACK 且 ACK 正确，握手继续；否则，重新发送第一步的 SYN 包。
- **第三次握手**:
  - 客户端发送一个带有 ACK 标志的包，确认服务器的初始序号。
  - 此时，客户端的连接状态变为” 已建立”。

2. 接收端（服务器端）流程

1. **第一次握手**:
   - 服务器接收来自客户端的 SYN 包，提取客户端的初始序号 `seq`。
   - 校验包的校验和，确保数据完整性。
   - 如果校验和正确且标志位为 SYN，握手继续；否则，丢弃包。
2. **第二次握手**:
   - 服务器发送一个 SYN+ACK 包，包含自身的初始序号 `seq` 和对客户端序号的确认 ACK（`seq+1`）。
   - 计算校验和并将其写入包头。
   - 将包发送回客户端，等待客户端响应。
3. **第三次握手**:
   - 服务器接收客户端返回的 ACK 包，验证 ACK 是否正确（即确认服务器的序号）。
   - 如果正确，握手完成，连接建立，进入文件传输阶段。

（四） 文件传输

1. 发送端（客户端）流程

- **发送文件名**:
  - 客户端构造一个带有文件名的包，并设置标志位表明这是文件名信息。
  - 发送该包到服务器，等待文件名确认。
- **发送文件大小**:
  - 客户端构造一个带有文件大小的包，标志位表明这是文件大小信息。
  - 发送该包到服务器，等待文件大小确认。
- **初始化滑动窗口**:
  - 初始化 `base`（窗口起始位置）、`cwnd`（拥塞窗口大小）和 `threshold`（拥塞控制阈值）。
  - 设置初始发送序号 `seq`，并开启定时器用于超时检测。
- **发送数据包**:
  - 遵循滑动窗口范围 $[base, base + cwnd - 1]$，依次发送数据包。
  - 每个数据包包含:
    - 序号 `seq`
    - 数据部分
    - 校验和
  - 如果窗口已满（`seq >= base + cwnd`），发送端会等待窗口滑动。
- **处理 ACK**:
  - 接收服务器返回的 ACK 包，检查 ACK 是否在窗口范围内。
  - 如果 ACK 对应的序号大于或等于 `base`:
    - 滑动窗口，更新 `base`。
    - 根据拥塞控制状态调整 `cwnd`:
      - 在慢启动阶段，`cwnd` 指数增长。
      - 在拥塞避免阶段，`cwnd` 线性增长。
      - 在快速恢复阶段，`cwnd` 恢复到 `threshold + 3`。
- **超时重传**:
  - 如果超过 `TIMEOUT` 时间未收到 ACK:
    - 将 `cwnd` 重置为 1（慢启动阶段）。
    - 阈值 `threshold` 减半。
    - 重传窗口中的所有未确认数据包。
- **完成文件发送**:
  - 当所有数据包都已被确认，文件传输完成。

2. 接收端（服务器端）流程

- **接收文件名**：
  - 接收一个带有文件名标志的包，提取并保存文件名。
  - 验证校验和，确保文件名包未损坏。
- **接收文件大小**：
  - 接收一个带有文件大小标志的包，提取并记录文件大小。
  - 验证校验和，确保文件大小包未损坏。
- **接收数据包**：
  - 初始化期望序号 `expectedSeq`。
  - 循环接收数据包，按以下逻辑处理：
    - 验证校验和是否正确。
    - 如果序号与 `expectedSeq` 一致：
      - 提取数据部分并写入文件。
      - 更新 `expectedSeq`，并发送对应的 ACK。
    - 如果校验失败或序号不匹配：
      - 丢弃数据包，重发上一次的 ACK。
- **完成文件接收**：
  - 当接收到的总数据量等于文件大小时，文件接收完成。

（五） 挥手断开连接（四次挥手）

1. 发送端（客户端）流程

- **第一次挥手**：
  - 客户端发送一个带 FIN 标志的包，表示请求断开连接。
  - 等待服务器返回 ACK。
- **第二次挥手**：
  - 客户端接收服务器返回的 ACK 包，确认服务器已收到 FIN。
- **第三次挥手**：
  - 客户端接收服务器发送的 FIN 包，表示服务器同意断开连接。
- **第四次挥手**：
  - 客户端发送一个 ACK 包，确认接收到服务器的 FIN。
  - 连接断开，关闭套接字。

2. 接收端（服务器端）流程

- **第一次挥手**:
  - 服务器接收客户端发送的 FIN 包，记录序号，表示客户端请求断开连接。
- **第二次挥手**:
  - 服务器发送一个 ACK 包，确认收到客户端的 FIN。
- **第三次挥手**:
  - 服务器发送一个带 FIN 标志的包，表示服务器同意断开连接。
- **第四次挥手**:
  - 服务器接收客户端返回的 ACK 包，确认客户端已收到 FIN。
  - 连接断开，关闭套接字。

四、 运行结果

(一) 启动服务器端和客户端

- 如图1所示，服务器端成功启动后会出现“启动成功”的字样。
- 客户端启动后进行三次握手连接。

图 1: 启动

(二) 发送文件

- 如图2所示，发送 1.jpg 图片文件。

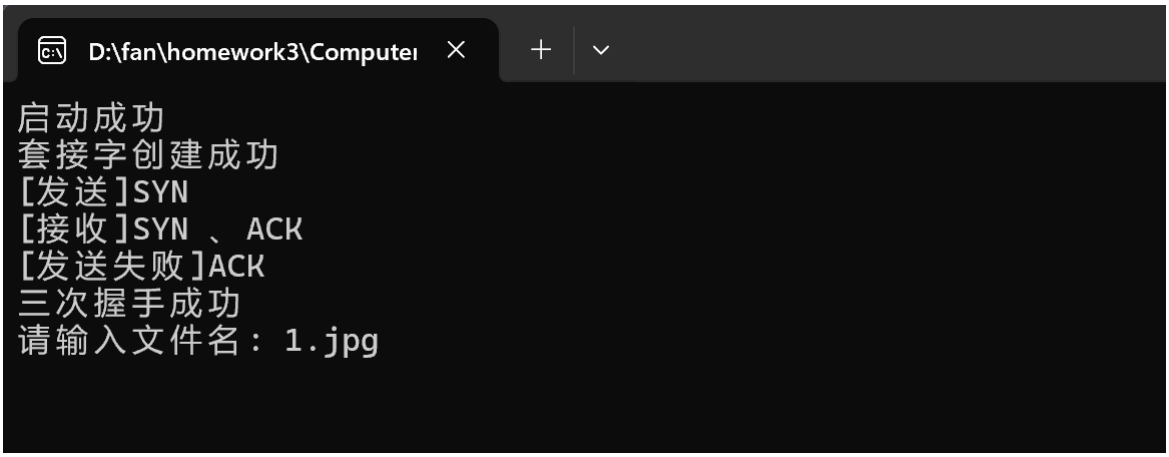

图 2: 发送文件

- 客户端发送成功。

图 3: 发送成功

- 服务器端接收成功。

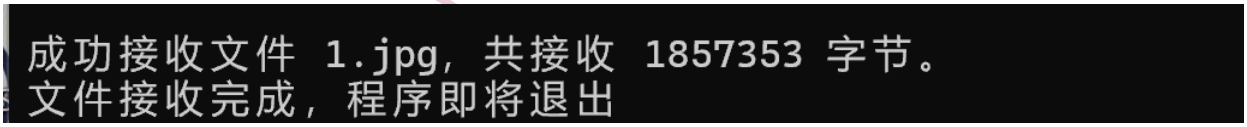

图 4: 接收成功

- 成功下载到本地。

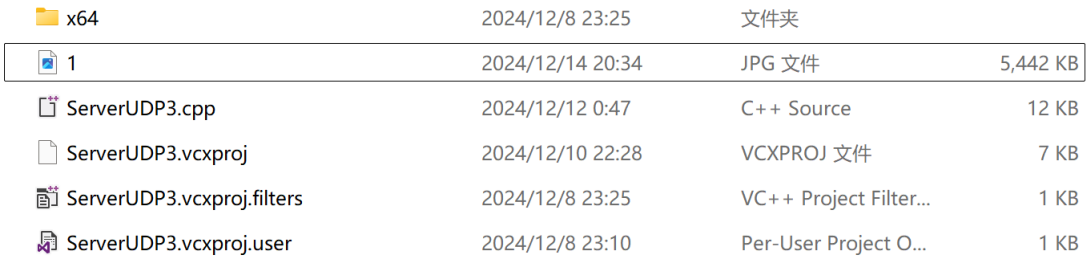

图 5: 成功下载到本地

(三) 超时重传进入慢启动阶段

```
[超时]
当前threshold = 8.0358 MSS
cwnd = 1 MSS
重复ACK次数 = 1
收到错误的ack = 862
当前threshold = 8.0358 MSS
cwnd = 1 MSS

收到正确的ack = 863
当前threshold = 8.0358 MSS
cwnd = 2 MSS

收到正确的ack = 864
当前threshold = 8.0358 MSS
cwnd = 3 MSS

收到正确的ack = 865
当前threshold = 8.0358 MSS
cwnd = 4 MSS

收到正确的ack = 866
当前threshold = 8.0358 MSS
cwnd = 5 MSS

收到正确的ack = 867
当前threshold = 8.0358 MSS
cwnd = 6 MSS

收到正确的ack = 868
当前threshold = 8.0358 MSS
cwnd = 7 MSS

收到正确的ack = 869
当前threshold = 8.0358 MSS
cwnd = 8 MSS

收到正确的ack = 870
当前threshold = 8.0358 MSS
cwnd = 9 MSS
```

图 6: 慢启动阶段

(四) 拥塞避免阶段

当 cwnd > threshold 进入拥塞避免阶段

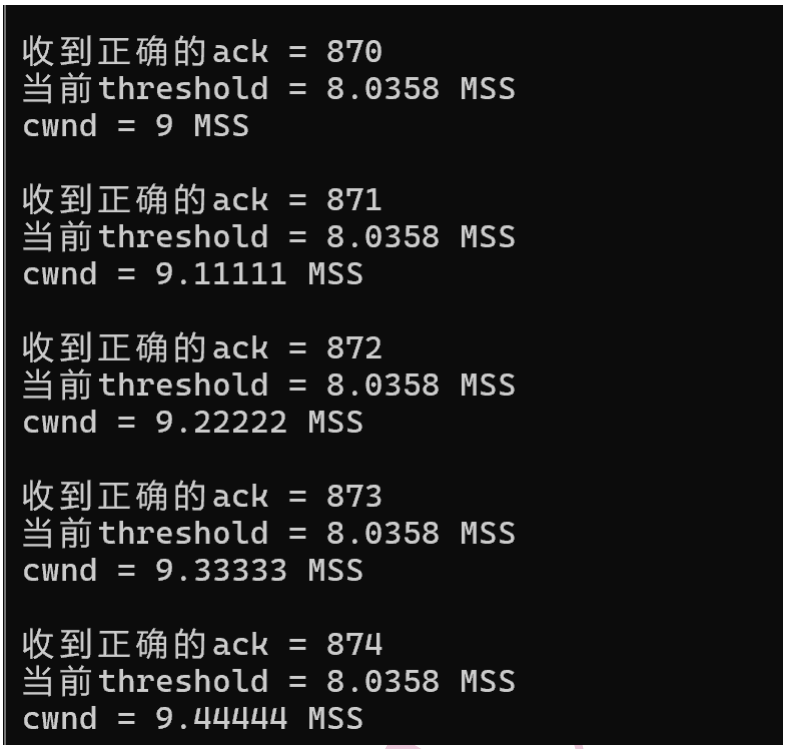

图 7: 拥塞避免阶段

（五） 四次挥手断开连接

- 客户端:

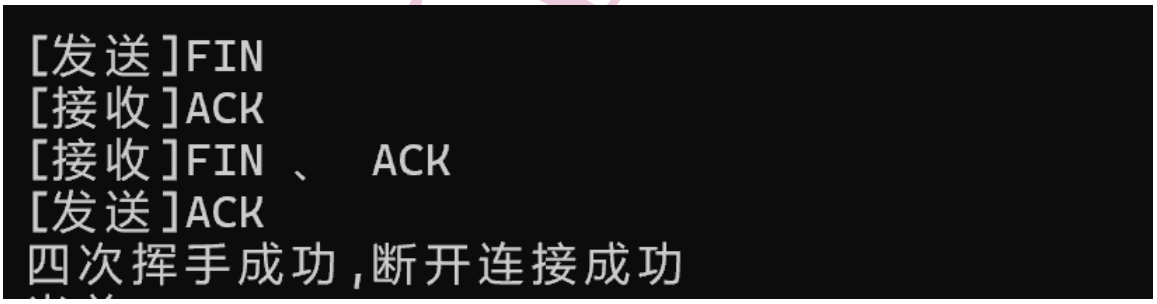

图 8: 客户端四次挥手

- 服务器端:

```
成功接收文件 1.jpg, 共接收 1857353 字节。
文件接收完成, 程序即将退出
[接收]FIN
[发送]ACK
[发送]FIN 、 ACK
[接收]ACK
四次挥手成功,已断开连接
```

图 9: 服务器端四次挥手

(六) 传输时间和吞吐率

```
总发送时间: 84.346 秒
发送的总字节数: 1857353 字节
吞吐率: 176.155 kbps
```

图 10

五、 传输结果分析

本实验在 UDP 底层基础上模拟了 TCP 式的可靠传输过程。测试结果表明:

1. **数据传输效率**:
   - 通过滑动窗口机制和拥塞控制策略，传输效率较高，吞吐量与网络状况密切相关。
   - 吞吐量公式:
   $$\text{吞吐量 (kbps)} = \frac{\text{总数据量 (字节)} \times 8}{\text{总时间 (秒)} \times 1000}$$
2. **可靠性**:
   - 使用校验和检测数据完整性，保证未损坏的数据被正确接收。
   - 超时重传和重复 ACK 机制有效处理了丢包问题，实现了可靠传输。
3. **拥塞控制效果**:
   - 实现了慢启动、拥塞避免和快速恢复三种状态的动态调整。
   - 在网络拥塞发生时，阈值和窗口大小调整避免了进一步丢包，并快速恢复传输状态。
4. **丢包与延迟分析**:
   - 丢包率通过接收端统计计算，重传机制有效弥补了丢包影响。
   - 延迟影响通过窗口控制和超时设置进行了适当处理。