南開大學
Nankai University

计算机学院

计算机网络

基于 UDP 服务设计可靠传输协议并编程实现 3-2

专业：计算机科学与技术

学号：2212032

姓名：徐亚民

日期：2024 年 12 月 6 日

目录

一、 实验要求

1. 实现单向传输。
2. 对于每个任务要求给出详细的协议设计。
3. 给出实现的拥塞控制算法的原理说明。
4. 完成给定测试文件的传输，显示传输时间和平均吞吐率。
5. 性能测试指标：吞吐率、文件传输时延，给出图形结果并进行分析。
6. 完成详细的实验报告。
7. 编写的程序应该结构清晰，具有较好的可读性。
8. 现场演示。
9. 提交程序源码、可执行文件和实验报告。

二、 协议设计

2.1 协议概述

本文设计了一种基于滑动窗口的可靠传输协议，采用了类似于 TCP 的机制，能够在 UDP 协议上实现可靠的数据传输。协议通过引入序列号、确认号以及超时重传机制，确保数据包的可靠到达。

2.2 数据包格式

协议中的数据包由以下字段组成，具体如下：

字段名称	大小	描述
seq_num	4 字节	发送序号，标识该数据包的唯一性。
ack_num	4 字节	确认序号，用于确认收到的数据包。
name	32 字节	文件名，发送的数据包对应的文件名称。
data	1024 字节	传输的实际数据内容。
data_len	4 字节	数据长度，表示当前数据包中实际数据的大小。
check_sum	2 字节	校验和，用于检测数据包传输的完整性。

字段名称	大小	描述
ACK	1 字节	确认标志位，标识该数据包是否是一个确认包（ACK）。
SYN	1 字节	同步标志位，标识该数据包是否为建立连接的同步包（SYN）。
FIN	1 字节	结束标志位，标识该数据包是否为结束连接的包（FIN）。

2.3 协议交互过程

本协议采用了类似 TCP 的三次握手和四次挥手来实现连接的建立与关闭，并使用滑动窗口机制进行数据传输。

2.3.1 连接建立（三次握手）

连接建立过程中，客户端和服务器通过三次消息交互来完成握手过程：

1. **客户端发送 SYN 包**：客户端发送一个 SYN 包，请求建立连接。SYN 包包含客户端的发送序号（seq_num）以及 SYN 标志位。
2. **服务器接收 SYN 包并回复 SYN-ACK 包**：服务器接收到 SYN 包后，回复一个 SYN-ACK 包，SYN 标志位为 1，ACK 标志位为 1，ack_num 为客户端发送的序号 +1。
3. **客户端回复 ACK 包**：客户端接收到 SYN-ACK 包后，发送一个 ACK 包，确认建立连接。ACK 包的 ack_num 为服务器发送的序号 +1。

2.3.2 数据传输

在连接建立后，客户端开始通过滑动窗口协议发送数据包。具体流程如下：

1. 客户端将文件分成多个数据包，每个数据包包含一定长度的数据（最大为 1024 字节）。每个数据包都包含发送序号（seq_num）和校验和（check_sum），确保数据的完整性。
2. 客户端按滑动窗口机制发送多个数据包，窗口大小为 10。每次发送后，客户端等待服务器的确认（ACK 包）。
3. 服务器收到数据包后，检查校验和，若无误则发送确认包（ACK）。确认包包含当前接收到的序号（ack_num）。

4. 客户端接收到确认包后，移除已确认的数据包并发送新的数据包。若确认包超时，则进行重传。

2.3.3 连接关闭（四次挥手）

连接关闭的过程包括四次挥手：

1. **客户端发送 FIN 包**：当客户端发送完所有数据后，发送一个 FIN 包请求关闭连接。FIN 包的 seq_num 为当前的发送序号，FIN 标志位为 1。
2. **服务器接收 FIN 包并回复 ACK 包**：服务器收到 FIN 包后，回复一个 ACK 包，ack_num 为客户端发送的序号 +1。
3. **服务器发送 FIN 包**：服务器准备关闭连接时，发送一个 FIN 包，请求关闭连接。
4. **客户端接收 FIN 包并回复 ACK 包**：客户端接收到 FIN 包后，回复一个 ACK 包，ack_num 为服务器发送的序号 +1，连接正式关闭。

2.3.4 重传机制

为了确保数据包的可靠传输，本协议实现了重传机制。具体如下：

- 客户端在发送数据包后会启动一个定时器，等待服务器的确认包。如果在规定时间内未收到确认包（即超时），则重传该数据包。
- 服务器收到数据包后，校验数据包的校验和，若正确则发送确认包。如果校验和错误，则丢弃数据包并不发送确认包。
- 客户端在收到确认包后，移除已确认的数据包，继续发送新的数据包。若超时则重新发送未确认的数据包。

2.4 协议设计总结

本协议设计通过滑动窗口控制发送的数据量，保证了在 UDP 上实现可靠数据传输。通过三次握手建立连接，四次挥手关闭连接，并通过校验和、超时重传等机制保障数据传输的正确性与完整性。

三、 代码分析

3.1 共享：数据包结构体

3.1.1 核心代码

```
struct Packet {
    int seq_num = 0;                    // 发送序号
    int ack_num = 0;                    // 确认序号
    char name[MAX_FILENAME_SIZE];       // 文件名称
    char data[MAX_DATA_SIZE];           // 发送数据
    int data_len = 0;                   // 数据长度
    int check_sum = 0;                  // 校验和
    bool ACK = false;                   // 标志位
    bool SYN = false;                   // 标志位
    bool FIN = false;                   // 标志位
};
```

3.1.2 代码分析

- **结构体定义：** 该代码定义了一个名为 **Packet** 的结构体，用于表示网络数据包。结构体包含多个字段，每个字段有特定的功能，主要用于封装和管理数据包的信息。
- **seq_num:** 该字段表示数据包的发送序号，用于标识数据包在发送序列中的位置。在数据传输中，序号用于确保数据包的顺序。
- **ack_num:** 该字段表示确认序号，用于确认接收到的数据包的顺序。接收方通过发送确认序号来告诉发送方哪些数据包已成功接收。
- **name[MAX_FILENAME_SIZE]:** 该字段用于存储文件名称。数据包可能用于传输文件，因此需要存储文件的名称以便于接收方识别。
- **data[MAX_DATA_SIZE]:** 该字段用于存储实际的数据内容。数据字段可能包含发送的数据，如文件的部分内容或其他信息。
- **data_len:** 该字段表示数据部分的长度，用于告知接收方数据的实际大小。
- **check_sum:** 该字段存储校验和，用于检测数据在传输过程中是否发生了错误。通过校验和，接收方可以验证数据包的完整性。
- **ACK, SYN, FIN:** 这些字段是标志位，用于表示数据包的状态或控制信息：
  - **ACK** 表示该数据包是否为确认包。
  - **SYN** 用于建立连接时的同步标志，常用于三次握手过程中的第一个数据包。

– **FIN** 用于表示连接的终止，发送此标志时表示数据传输已结束。

3.2 共享：计算校验和

3.2.1 核心代码

```
unsigned short calculate_checksum(char* data) {
    unsigned short len = strlen(data);
    unsigned short checksum = 0;
    unsigned short* ptr = (unsigned short*)data;  // 将数据视为16位单位的数组

    // 对数据进行16位加法求和
    for (int i = 0; i < len / 2; i++) {
        checksum += ptr[i];

        // 如果有进位，进行回卷
        if (checksum > 0xFFFF) {
            checksum = (checksum & 0xFFFF) + 1;
        }
    }

    // 如果数据长度是奇数，处理剩余的最后一个字节
    if (len % 2 != 0) {
        checksum += (unsigned short)(data[len - 1] << 8);

        // 如果有进位，进行回卷
        if (checksum > 0xFFFF) {
            checksum = (checksum & 0xFFFF) + 1;
        }
    }

    // 返回反码
    return ~checksum;
}
```

3.2.2 代码分析

- **函数定义：** 该函数 **calculate_checksum** 计算并返回给定数据的校验和。校验和用于检测数据在传输过程中是否出现错误，是数据传输协议中常见的技术。
- **参数：**
  - **char* data:** 输入参数，指向要计算校验和的字符数据。函数会基于该数据计算校验和。
- **变量定义：**

- **unsigned short len:** 存储输入数据的长度，即数据的字节数。通过 **strlen(data)** 获取。
- **unsigned short checksum:** 用于存储计算中的校验和值，初始化为 0。
- **unsigned short* ptr:** 将输入的字符数据 **data** 强制类型转换为指向 **unsigned short** 类型的指针，使得每次加法运算是基于 16 位数据进行的。

- **加法运算：**
  - 通过循环遍历数据，将数据按 16 位分组进行加法运算。每次加法运算后，若校验和大于 0xFFFF（即 16 位最大值），则发生进位（通过回卷操作 **checksum = (checksum & 0xFFFF) + 1;**）以保证校验和保持在 16 位范围内。
- **处理奇数长度数据：**
  - 如果数据长度为奇数，循环完成后剩下一个字节未被处理，函数将最后一个字节按 16 位的高位填充到校验和中。对于剩余字节，使用位移 **data[len - 1] « 8** 来模拟 16 位数据，并继续进行进位处理。
- **返回反码：**
  - 最终返回计算得到的校验和的反码。通过 **checksum** 进行反转，常见于许多网络协议中的校验和计算方法（例如 TCP/IP 协议）。

3.3 共享：发送数据包

3.3.1 核心代码

```
int send_packet(SOCKET& sock, struct sockaddr_in& receiver_addr, Packet& pkt) {
    // 计算校验和
    pkt.check_sum = calculate_checksum(pkt.data);

    // 发送数据包
    int sent_len = sendto(sock, (char*)&pkt, sizeof(pkt), 0, (struct sockaddr*)&
        receiver_addr, sizeof(receiver_addr));

    if (sent_len == SOCKET_ERROR) {
        cerr << "发送数据包失败" << endl;
        return -1;
    }
    else {
        cout << "发送数据包，发送序号: " << pkt.seq_num << "，确认序号: " << pkt.
            ack_num
            << "，数据大小: " << pkt.data_len << "，校验和: " << pkt.check_sum
            << "，ACK: " << pkt.ACK << "，SYN: " << pkt.SYN << "，FIN: " << pkt.FIN <<
                endl;
```

```
        return 0;
    }
}
```

3.3.2 代码分析

- **函数定义：**该函数 **send_packet** 用于向指定的接收地址发送一个数据包。函数接受三个参数：
  - **sock:** 一个引用类型的 SOCKET，表示用于发送数据的套接字。
  - **receiver_addr:** 一个引用类型的 sockaddr_in 结构体，表示接收方的地址信息。
  - **pkt:** 一个引用类型的 Packet 结构体，表示需要发送的数据包。
- **计算校验和：**在发送数据包之前，函数通过调用 **calculate_checksum** 函数计算数据包的校验和，并将其赋值给 **pkt.check_sum** 字段。校验和用于检查数据的完整性，确保数据在传输过程中没有发生错误。
- **发送数据包：**使用 **sendto** 函数发送数据包。具体过程是将 **pkt** 结构体的地址强制转换为 char* 类型，并将其发送到指定的接收地址 **receiver_addr**。发送的字节数是数据包的大小，使用 sizeof(pkt) 来确定。
- **错误处理：**如果 **sendto** 返回 SOCKET_ERROR，表示发送数据包失败，函数会输出错误信息并返回 -1。否则，表示数据包发送成功，函数会打印发送的数据包的一些信息，并返回 0。
  - 打印的内容包括：发送序号 **pkt.seq_num**、确认序号 **pkt.ack_num**、数据长度 **pkt.data_len**、校验和 **pkt.check_sum**、以及标志位 **pkt.ACK**、**pkt.SYN** 和 **pkt.FIN**。

3.4 接收数据包（客户端）

3.4.1 核心代码

```
//接收数据包（发送端）
int receive_packet(SOCKET& sock, struct sockaddr_in& sender_addr, Packet& pkt_received)
    {
    int len = sizeof(sender_addr);
    char buffer[MAX_PACKET_SIZE];
    int recv_len = recvfrom(sock, buffer, sizeof(buffer), 0, (struct sockaddr*)&
        sender_addr, &len);
```

```
6
7       if (recv_len == SOCKET_ERROR) {
8           cerr << "接收数据包失败" << endl;
9           return -1;
10      }
11
12      // 提取数据包内容
13      memcpy(&pkt_received, buffer, sizeof(pkt_received));
14
15      // 计算校验和
16      unsigned short calculated_checksum = calculate_checksum(pkt_received.data);
17
18      // 检查校验和
19      if (pkt_received.check_sum != calculated_checksum) {
20          cerr << "接收校验和: " << pkt_received.check_sum << ", 计算校验和: " <<
                calculated_checksum << ", 校验和错误, 丢弃数据包" << endl;
21          return -1;
22      }
23
24      // 检查确认序号是否正确 (客户端发送数据包, 服务器返回确认序号)
25      if (pkt_received.ACK && pkt_received.ack_num != ack_num_expected && !pkt_received.
            FIN && !pkt_received.SYN) {
26          cout << "预期确认序号: " << ack_num_expected << ", 实际确认序号: " <<
                pkt_received.ack_num << ", 确认序号不正确, 丢弃数据包" << endl;
27          return -1;
28      }
29
30      // 更新预期确认序号
31      ack_num_expected++;
32
33      // 发送数据包的内容
34      cout << "接收数据包, 发送序号: " << pkt_received.seq_num << ", 确认序号: " <<
            pkt_received.ack_num
35          << ", 数据大小: " << pkt_received.data_len << ", 校验和: " << pkt_received.
                check_sum
36          << ", ACK: " << pkt_received.ACK << ", SYN: " << pkt_received.SYN << ", FIN: "
                << pkt_received.FIN << endl;
37
38      return 0;
39  }
```

3.4.2 代码分析

- **函数定义：** 该函数 **receive_packet** 用于接收从发送端发送过来的数据包。函数接受三个参数：
  - **sock:** 一个引用类型的 SOCKET，表示用于接收数据的套接字。
  - **sender_addr:** 一个引用类型的 sockaddr_in 结构体，表示发送方的地址信息。
  - **pkt_received:** 一个引用类型的 Packet 结构体，用于存储接收到的数据包。

- **接收数据：**函数使用 **recvfrom** 函数接收数据包。接收到的数据存储在一个缓冲区 **buffer[MAX_PACKET_SIZE]** 中，并通过 **memcpy** 将缓冲区的内容复制到 **pkt_received** 数据包中。接收到的数据包长度保存在 **recv_len** 中，如果接收失败，函数会输出错误信息并返回 -1。
- **校验和检查：**接收到数据包后，首先调用 **calculate_checksum** 函数计算接收到的数据包的校验和，并与接收到的数据包的 **check_sum** 进行对比。如果两者不匹配，表示数据包在传输过程中可能发生了损坏，函数会输出错误信息并丢弃该数据包，返回 -1。
- **确认序号检查：**如果数据包的 **ACK** 标志位为真且 **ack_num** 不等于预期的确认序号（存储在 **ack_num_expected** 变量中），同时数据包不包含 **FIN** 和 **SYN** 标志，则认为确认序号不正确，函数会输出相关信息并丢弃数据包，返回 -1。
- **更新确认序号：**如果数据包的确认序号正确，函数会增加 **ack_num_expected**，表示下一个期望的确认序号。
- **输出接收数据包信息：**函数打印接收到的数据包的信息，包括发送序号 **pkt_received.seq_num**、确认序号 **pkt_received.ack_num**、数据大小 **pkt_received.data_len**、校验和 **pkt_received.check_sum** 以及标志位 **pkt_received.ACK**、**pkt_received.SYN** 和 **pkt_received.FIN**。
- **返回值：**如果接收和校验过程都成功，函数返回 0，表示数据包接收成功；否则返回 -1，表示数据包处理失败。

3.5 接收数据包（服务器）

3.5.1 核心代码

```
//接收数据包（接收端）
int receive_packet(SOCKET& sock, struct sockaddr_in& sender_addr, Packet& pkt_received)
    {
    int len = sizeof(sender_addr);
    char buffer[MAX_PACKET_SIZE];
    int recv_len = recvfrom(sock, buffer, sizeof(buffer), 0, (struct sockaddr*)&
        sender_addr, &len);

    if (recv_len == SOCKET_ERROR) {
        cerr << "接收数据包失败" << endl;
        return -1;
    }

    // 提取数据包内容
    memcpy(&pkt_received, buffer, sizeof(pkt_received));

    // 计算校验和
    unsigned short calculated_checksum = calculate_checksum(pkt_received.data);
```

```
17
18      // 检查校验和
19      if (pkt_received.check_sum != calculated_checksum) {
20          cerr << "接收校验和: " << pkt_received.check_sum << ", 计算校验和: " <<
                calculated_checksum << ", 校验和错误, 丢弃数据包" << endl;
21          return -1;
22      }
23
24      // 接收到错误的数据包
25      if (pkt_received.seq_num > seq_num_expected) {
26          cerr << "期望序号: " << seq_num_expected << ", 接收序号: " << pkt_received.
                seq_num << ", 序号不正确, 丢弃数据包" << endl;
27          return -1;
28      }
29
30      // 发送数据包的内容
31      cout << "接收数据包, 发送序号: " << pkt_received.seq_num << ", 确认序号: " <<
            pkt_received.ack_num
32          << ", 数据大小: " << pkt_received.data_len << ", 校验和: " << pkt_received.
                check_sum
33          << ", ACK: " << pkt_received.ACK << ", SYN: " << pkt_received.SYN << ", FIN: "
                << pkt_received.FIN << endl;
34
35      // 接收到正确的数据包
36      if (pkt_received.seq_num == seq_num_expected) {
37          // 更新预期发送序号
38          seq_num_expected++;
39          return 0;
40      }
41
42      // 接收到重复的数据包
43      if (pkt_received.seq_num < seq_num_expected) {
44          return 1;
45      }
46
47      return 0;
48  }
```

3.5.2 代码分析

该函数 **receive_packet** 用于接收从发送端发送过来的数据包，并对其进行校验、序号检查等操作。以下是代码的逐步分析：

- **接收数据包：** 函数首先使用 **recvfrom** 函数从套接字 `sock` 中接收数据，并将接收到的数据存储在缓冲区 `buffer[MAX_PACKET_SIZE]` 中。接收到的数据长度由 `recv_len` 变量记录。如果接收数据失败，返回 -1，并输出错误信息”接收数据包失败”。
- **提取数据包内容：** 使用 **memcpy** 将接收到的缓冲区数据复制到 `pkt_received` 结构体中。该结构体用于存储数据包的各项内容，如序号、校验和等信息。

- **计算和检查校验和：**调用 **calculate_checksum** 函数计算接收到数据包的校验和，并与接收到的校验和进行比对。如果两者不一致，表示数据包可能发生了错误，函数输出相关的错误信息并丢弃该数据包，返回 -1。
- **序号检查：**接收到的数据包的序号与预期序号（存储在 `seq_num_expected` 变量中）进行比较。如果接收到的数据包的序号大于预期序号，表示该数据包是错误的（可能是数据包丢失后的重新发送），函数会输出错误信息并丢弃该数据包，返回 -1。
- **打印接收到数据包的信息：**如果校验和正确且序号符合预期，函数会打印接收到的数据包的详细信息，包括：
  - 发送序号 `pkt_received.seq_num`
  - 确认序号 `pkt_received.ack_num`
  - 数据包大小 `pkt_received.data_len`
  - 校验和 `pkt_received.check_sum`
  - 各标志位（`ACK`、`SYN`、`FIN`）
- **处理正确的数据包：**如果接收到的数据包的序号与预期序号相同，表示该数据包是正确的。此时，函数更新预期的序号（`seq_num_expected++`），并返回 0，表示成功接收到数据包。
- **处理重复的数据包：**如果接收到的数据包的序号小于预期序号，表示该数据包是一个重复的包（可能是因为网络中丢包或重传）。函数会直接返回 1，表示接收到了重复的数据包，但不做进一步处理。
- **返回值：**
  - 如果校验和错误或序号不正确，函数返回 -1，表示数据包处理失败。
  - 如果接收到的数据包正确，函数返回 0，表示数据包处理成功。
  - 如果接收到的是重复的数据包，函数返回 1，表示接收到了重复的数据包，但不进行进一步处理。

3.6 发送文件（客户端）

3.6.1 核心代码

```
// 发送文件
int send_file(SOCKET& sock, struct sockaddr_in& receiver_addr, string filename) {
    // 创建发送队列，实现滑动窗口流量控制
    queue<Packet> send_queue;

    // 超时重传次数
    int timeout = 0;

    // 记录文件的开始时间
    auto start_time = chrono::high_resolution_clock::now();

    // 打开文件
    ifstream file("send/" + filename, ios::binary);
    if (!file.is_open()) {
        cerr << "打开文件失败" << endl;
        return -1;
    }

    // 获取文件大小用以计算吞吐率
    file.seekg(0, ios::end); // 移动到文件末尾
    long file_size = file.tellg(); // 获取文件大小
    file.seekg(0, ios::beg); // 将文件指针重置到文件开头

    // 设置接收超时时间，单位为毫秒
    int timeout_ms = 100;  // 毫秒超时
    setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, (const char*)&timeout_ms, sizeof(
        timeout_ms));

    while (!file.eof()) { // 读取文件，滑动窗口发送数据包，接收确认数据包

        // 发送窗口填充数据包
        while (send_queue.size() < WINDOW_SIZE && !file.eof()) {
            Packet pkt;
            strncpy_s(pkt.name, sizeof(pkt.name), filename.c_str(), sizeof(pkt.name) -
                1);  // 设置文件名称
            pkt.name[sizeof(pkt.name) - 1] = '\0';  // 文件名称
            file.read(pkt.data, sizeof(pkt.data));  // 读取数据
            int read_len = file.gcount();           // 读取长度
            pkt.data_len = read_len;                // 数据长度
            pkt.seq_num = seq_num_share++;          // 发送序号
            send_queue.push(pkt);                   // 添加队列
            cout << "发送窗口填入新数据包，发送窗口大小：" << WINDOW_SIZE
                << "，已用分组数量：" << send_queue.size() << "，可用分组数量：" <<
                    WINDOW_SIZE - send_queue.size() << endl;
        }

        // 更新预期确认序号
        ack_num_expected = send_queue.front().seq_num + 1;

        // 发送窗口发送数据包
        queue<Packet> tempQueue = send_queue;  // 复制队列，避免改变原队列内容
        while (!tempQueue.empty()) {
            send_packet(sock, receiver_addr, tempQueue.front());  // 发送队头数据包
            tempQueue.pop();  // 删除队头数据包（不会影响原队列）
        }

```

```
54          // 发送窗口接收数据包
55          int receive_num = send_queue.size();
56          // 如果接收服务器的数据包超时（1ms），就重新把发送窗口里面剩下的数据包发送一次
57          for (int i = 0; i < receive_num; i++) {
58              Packet pkt_received;
59              int result = receive_packet(sock, receiver_addr, pkt_received);
60              if (result == 0) {
61                  send_queue.pop();
62                  cout << "接收窗口确认新数据包，发送窗口大小: " << WINDOW_SIZE
63                      << "，已用分组数量: " << send_queue.size() << "，可用分组数量: " <<
                            WINDOW_SIZE - send_queue.size() << endl;
64              }
65              else {
66                  // 超时处理：如果接收超时，则进行重传
67                  timeout++;
68                  cout << "ACK超时，重新发送窗口内数据包，当前累积重传次数: " << timeout
                        << endl;
69                  break; // 下一次 while 循环，重新发送窗口内数据包
70              }
71          }
72      }
73
74      // 记录文件传输结束时间
75      auto end_time = chrono::high_resolution_clock::now();
76      chrono::duration<double> duration = end_time - start_time;
77
78      // 输出超时重传次数
79      cout << "超时重传次数为: " << timeout << " 次" << endl;
80
81      // 计算吞吐率（文件大小 / 传输时间）
82      double throughput = (double)file_size / duration.count(); // 吞吐率，单位字节/秒
83      cout << "文件传输时间: " << duration.count() << " 秒" << endl;
84      cout << "吞吐率: " << throughput / 1024 << " KB/s" << endl; // 吞吐率单位: KB/s
85
86      file.close();
87
88      return 0;
89  }
```

3.6.2 代码分析

- **函数定义：** 该代码定义了一个名为 **send_file** 的函数，用于通过网络发送文件。函数接受三个参数：
  - **sock:** 网络套接字，用于数据传输。
  - **receiver_addr:** 接收方的地址结构体，用于指定目标接收地址。
  - **filename:** 需要发送的文件名。
- **发送队列：** **send_queue** 是一个 **Packet** 类型的队列，用于实现滑动窗口流量控制。

它保存正在等待确认的包，并控制数据包的发送顺序。

- **超时重传：** **timeout** 变量用于记录超时重传的次数。在数据传输过程中，如果接收方的确认包超时，将触发重传机制。
- **文件打开：** 使用 `ifstream` 打开指定文件。如果文件无法打开，程序输出错误信息并返回 -1。
- **文件大小：** 通过 `seekg` 函数移动文件指针到文件末尾，获取文件大小，用于后续计算吞吐率。
- **超时设置：** 使用 `setsockopt` 函数设置接收方的超时时间为 100 毫秒。如果在此时间内没有接收到确认包，程序会进行重传。
- **发送数据包：** 使用滑动窗口机制发送数据包。通过 `file.read` 逐步读取文件内容，并将数据分包存入 **send_queue** 队列中。每当队列中的数据包数量小于窗口大小时，继续读取数据并填充数据包。
- **发送确认：** 在发送完数据包后，程序通过 `send_packet` 函数发送队列中的数据包，并使用临时队列 **tempQueue** 来避免修改原队列的内容。
- **接收确认：** 程序通过 `receive_packet` 函数接收接收方的确认包。如果接收到确认包（`result == 0`），则从发送队列中弹出已确认的数据包。如果超时，则进行重传。
- **超时处理：** 如果接收确认包超时（1ms），则重新发送队列中的数据包，并增加超时计数 **timeout**。
- **传输时间和吞吐率：** 在文件传输结束后，记录传输的时间，并根据文件大小和传输时间计算吞吐率。吞吐率的单位为 **KB/s**。
- **文件关闭：** 文件操作完成后，调用 `file.close()` 关闭文件。
- **返回值：** 函数返回 0 表示文件发送成功。

3.7 主函数（客户端）

3.7.1 核心代码

```
// 发送端主函数
#pragma comment(lib,"ws2_32.lib")
int main() {
    // 初始化WinSock
    WSADATA wsaData;
```

```
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
        cerr << "WinSock 初始化失败" << endl;
        return -1;
    }

    // 创建UDP套接字
    SOCKET sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (sock == INVALID_SOCKET) {
        cerr << "创建套接字失败" << endl;
        return -1;
    }

    // 创建接收方地址
    sockaddr_in receiver_addr;
    receiver_addr.sin_family = AF_INET;  // 设置地址族为IPv4
    receiver_addr.sin_port = htons(SERVER_PORT);  // 设置服务器端口号为20000
    receiver_addr.sin_addr.s_addr = inet_addr(IPADDR);  // 设置IP地址为127.0.0.1

    // 创建本地客户端地址
    sockaddr_in client_addr;
    client_addr.sin_family = AF_INET;  // 设置地址族为IPv4
    client_addr.sin_port = htons(CLIENT_PORT);  // 设置客户端端口号为10000
    client_addr.sin_addr.s_addr = inet_addr(IPADDR);  // 设置客户端IP地址为127.0.0.1

    // 绑定客户端套接字到本地端口
    if (bind(sock, (struct sockaddr*)&client_addr, sizeof(client_addr)) == SOCKET_ERROR
        ) {
        cerr << "绑定套接字失败" << endl;
        return -1;
    }

    // 三次握手第一步：发送SYN包
    Packet pkt0;
    pkt0.SYN = true;
    pkt0.seq_num = seq_num_share++;
    // cout << "发送SYN包，请求建立连接" << endl;
    send_packet(sock, receiver_addr, pkt0);

    // 三次握手第二步：接收SYN-ACK包
    Packet pkt_received0;
    receive_packet(sock, receiver_addr, pkt_received0);

    // 三次握手第三步：发送ACK包
    if (pkt_received0.SYN && pkt_received0.ACK && pkt_received0.ack_num ==
        seq_num_share) {
        Packet pkt1;
        pkt1.ACK = true;
        pkt1.seq_num = seq_num_share++;
        pkt1.ack_num = pkt_received0.seq_num + 1;
        send_packet(sock, receiver_addr, pkt1);
    }

    // 创建文件名称
    string filename;

    // 循环发送文件
```

```
60      while (true) {
61          // 持续读取用户输入的文件名
62          cout << "请输入要发送的文件名（输入exit退出）: ";
63          getline(cin, filename);
64
65          if (filename == "exit")
66              break;  // 用户输入exit时退出
67          else
68              send_file(sock, receiver_addr, filename);  // 调用send_file发送文件
69      }
70
71      // 发送完文件后，开始四次挥手过程
72
73      // 四次挥手第一步：发送FIN包
74      Packet pkt2;
75      pkt2.FIN = true;
76      pkt2.seq_num = seq_num_share++;
77      send_packet(sock, receiver_addr, pkt2);
78
79      // 四次挥手第二步：接收ACK包
80      Packet pkt_received1;
81      receive_packet(sock, receiver_addr, pkt_received1);
82
83      // 四次挥手第三步：接收FIN-ACK包
84      Packet pkt_received2;
85      receive_packet(sock, receiver_addr, pkt_received2);
86
87      // 四次挥手第四步：发送ACK包
88      Packet pkt3;
89      pkt3.ACK = true;
90      pkt3.seq_num = seq_num_share++;
91      pkt3.ack_num = pkt_received2.seq_num + 1;
92      send_packet(sock, receiver_addr, pkt3);
93
94      closesocket(sock);
95      WSACleanup();
96
97      return 0;
98  }
```

3.7.2 代码分析

- **初始化 WinSock:** 代码首先使用 **WSAStartup** 初始化 WinSock 库，确保网络编程功能的可用性。若初始化失败，程序会输出错误信息并返回-1。
- **创建 UDP 套接字:** 使用 **socket** 函数创建一个 UDP 套接字。套接字是网络通信的基础，用于发送和接收数据。如果创建失败，程序会输出错误信息并返回-1。
- **设置接收方地址:** 通过填充 **receiver_addr** 结构体来指定接收方的网络地址，包括地址族（IPv4）、端口号和 IP 地址。这里使用了 127.0.0.1 作为接收方 IP 地址，表示本地通信。

- **设置本地客户端地址:** 类似地，设置本地客户端地址 **client_addr**，包括本地端口号（10000）和本地 IP 地址（127.0.0.1）。
- **绑定客户端套接字:** 使用 **bind** 函数将客户端套接字绑定到本地端口。如果绑定失败，程序会输出错误信息并返回-1。
- **三次握手过程:**
  - **第一步（发送 SYN 包）:** 创建并发送一个 SYN 包，表示客户端请求建立连接。设置 **SYN** 标志为 true，并增加序列号 **seq_num**。
  - **第二步（接收 SYN-ACK 包）:** 客户端等待接收服务器响应的 SYN-ACK 包。接收到的数据包存储在 **pkt_received0** 中。
  - **第三步（发送 ACK 包）:** 如果接收到的包符合预期（SYN 和 ACK 标志均为真），则客户端发送一个 ACK 包作为响应，确认连接建立。
- **文件发送过程:** 客户端会不断循环提示用户输入文件名，并调用 **send_file** 函数发送文件。如果用户输入”exit”，程序退出循环。
- **四次挥手过程:**
  - **第一步（发送 FIN 包）:** 客户端发送一个 FIN 包，表示数据传输结束，开始关闭连接。
  - **第二步（接收 ACK 包）:** 客户端等待并接收确认包（ACK 包），确认接收方已收到 FIN 包。
  - **第三步（接收 FIN-ACK 包）:** 客户端接收来自服务器的 FIN-ACK 包，表示服务器也准备关闭连接。
  - **第四步（发送 ACK 包）:** 客户端发送 ACK 包，确认已收到 FIN-ACK 包，完成四次挥手过程。
- **关闭套接字和清理资源:** 使用 **closesocket** 函数关闭套接字，并使用 **WSACleanup** 清理 WinSock 库资源，结束网络通信。
- **主函数返回值:** 程序最终返回 0，表示客户端成功执行完毕。

3.8 处理数据包（服务器）

3.8.1 核心代码

```
// 处理数据包
int handle_packet(SOCKET& sock, struct sockaddr_in& sender_addr, queue<Packet>&
    receive_queue) {

    // 接收数据包
    Packet pkt_received;
    int ret = receive_packet(sock, sender_addr, pkt_received);

    // 特殊情况判断
    if (ret == -1) { // 接收到错误的数据包
        return 0;
    }
    if (ret == 1) { // 接收到重复的数据包
        // 发送ACK包
        Packet pkt;
        pkt.ACK = true;
        pkt.seq_num = seq_num_share++;
        pkt.ack_num = pkt_received.seq_num + 1;
        send_packet(sock, sender_addr, pkt);
        return 0;
    }

    // 处理三次握手
    if (pkt_received.SYN) {
        three_handshakes = true;
        // 收到SYN包, 发送SYN-ACK包
        Packet pkt;
        pkt.SYN = true;
        pkt.ACK = true;
        pkt.seq_num = seq_num_share++;
        pkt.ack_num = pkt_received.seq_num + 1;
        send_packet(sock, sender_addr, pkt);
        return 0;
    }

    // 处理三次握手和处理四次挥手
    if (pkt_received.ACK && pkt_received.ack_num == seq_num_share) {
        if (three_handshakes) {
            three_handshakes = false;
            // 更新预期发送序号
            seq_num_expected = pkt_received.seq_num + 1;
            return 0;       // 如果是三次握手的第三步ACK那么就return 0
        }
        if (waving_four_times) {
            waving_four_times = false;
            return -1;      // 如果是四次挥手的第四步ACK那么就return -1
        }
        return 0;
    }

    // 处理四次挥手
    if (pkt_received.FIN) {
        waving_four_times = true;
        // 收到FIN包, 发送ACK包
        Packet pkt1;
        pkt1.ACK = true;
```

```
56          pkt1.seq_num = seq_num_share++;
57          pkt1.ack_num = pkt_received.seq_num + 1;
58          send_packet(sock, sender_addr, pkt1);
59          // 收到FIN包，发送FIN-ACK包
60          Packet pkt2;
61          pkt2.FIN = true;
62          pkt2.ACK = true;
63          pkt2.seq_num = seq_num_share++;
64          pkt2.ack_num = pkt_received.seq_num + 1;
65          send_packet(sock, sender_addr, pkt2);
66          return 0;
67      }
68
69      // 不是三次握手和四次挥手，那就将数据包存入接收窗口
70      if (receive_queue.size() < WINDOW_SIZE) {
71          receive_queue.push(pkt_received);
72      }
73
74      // 接收窗口满了或者文件传输完毕，把数据写入到文件中
75      if (receive_queue.size() == WINDOW_SIZE || pkt_received.data_len < MAX_DATA_SIZE) {
76          while (!receive_queue.empty()) {
77              // 写入数据到文件中
78              Packet tempPkt = receive_queue.front();
79              receive_queue.pop();
80              ofstream file("receive/" + string(tempPkt.name), ios::binary | ios::app);
81              if (file.is_open()) {
82                  file.write(tempPkt.data, tempPkt.data_len);
83                  file.close();
84                  cout << "数据已写入文件。路径: receive/" + string(tempPkt.name) << endl
                        ;
85              }
86              else {
87                  cerr << "写入文件失败" << endl;
88              }
89              // 发送ACK包
90              Packet pkt;
91              pkt.ACK = true;
92              pkt.seq_num = seq_num_share++;
93              pkt.ack_num = tempPkt.seq_num + 1;
94              send_packet(sock, sender_addr, pkt);
95          }
96      }
97      return 0;
98  }
```

3.8.2 代码分析

- **函数概述：**该函数 **handle_packet** 负责处理接收到的数据包。其核心任务是根据不同的条件（如数据包类型、握手过程等）进行相应的操作，包括发送数据包、存储数据包到接收队列中，或者执行连接的管理操作（如三次握手、四次挥手等）。

- **接收数据包：**首先，通过 **receive_packet** 函数接收数据包。如果接收到的数据包出

错（返回值为 -1），则返回 0，表示没有进一步操作。如果接收到的是重复的数据包（返回值为 1），则发送一个带有确认标志（ACK）的数据包，并返回 0。

- **三次握手处理：**如果接收到的数据包包含 **SYN** 标志（即是一个同步请求），则表示客户端正在发起连接。此时，设置 **three_handshakes** 为真，表示正在进行三次握手过程。接着，发送一个带有 **SYN** 和 **ACK** 标志的数据包作为响应，继续三次握手过程。

- **三次握手和四次挥手 ACK 处理：**如果接收到的数据包包含 **ACK** 标志且其确认号（ack_num）等于共享的发送序号 **seq_num_share**，则根据当前的状态进行相应的处理：

  - 如果是三次握手过程的最后一步（即收到 ACK 包），则更新预期的发送序号，并返回 0。

  - 如果是四次挥手过程的第四步 ACK 包（表示连接关闭），则返回 -1，表示结束。

- **四次挥手处理：**如果接收到的数据包包含 **FIN** 标志，表示连接关闭请求。此时，设置 **waving_four_times** 为真，表示正在处理四次挥手过程。首先，发送一个带有 **ACK** 标志的数据包响应对方。然后，发送一个带有 **FIN** 和 **ACK** 标志的数据包，表示结束连接。

- **接收数据包存储：**如果数据包不是三次握手或四次挥手相关的数据包，则将数据包存入接收队列 **receive_queue**，前提是接收队列的大小小于设定的窗口大小（**WINDOW_SIZE**）。

- **数据包写入文件：**如果接收队列已满或者接收到的数据包的数据长度小于最大数据包长度（**MAX_DATA_SIZE**），则开始将接收到的数据包写入文件。遍历接收队列，将每个数据包的内容写入到文件中，并发送一个带有 **ACK** 标志的确认数据包。

- **文件写入失败处理：**在写入文件时，如果文件打开失败，则输出错误信息，表示写入文件失败。

- **ACK 包发送：**每次处理完数据包后，都会发送一个带有 **ACK** 标志的确认包，以告知发送方数据已成功接收。

3.9 主函数（服务器）

3.9.1 核心代码

```cpp
// 接收端主函数
#pragma comment(lib,"ws2_32.lib")
int main() {
    // 初始化WinSock
    WSADATA wsaData;
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
        cerr << "WinSock 初始化失败" << endl;
        return -1;
    }

    // 创建UDP套接字
    SOCKET sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (sock == INVALID_SOCKET) {
        cerr << "创建套接字失败" << endl;
        return -1;
    }

    // 创建服务器地址
    sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;  // 设置地址族为IPv4
    server_addr.sin_port = htons(SERVER_PORT);  // 设置端口号为30000
    server_addr.sin_addr.s_addr = inet_addr(IPADDR);  // 设置IP地址为127.0.0.1

    // 创建客户端地址
    sockaddr_in client_addr;
    client_addr.sin_family = AF_INET;  // 设置地址族为IPv4
    client_addr.sin_port = htons(CLIENT_PORT);  // 设置客户端端口号为20000
    client_addr.sin_addr.s_addr = inet_addr(IPADDR);  // 设置客户端IP地址为127.0.0.1

    // 绑定套接字
    if (bind(sock, (struct sockaddr*)&server_addr, sizeof(server_addr)) == SOCKET_ERROR
        ) {
        cerr << "绑定套接字失败" << endl;
        return -1;
    }

    // 创建接收队列，实现滑动窗口流量控制
    queue<Packet> receive_queue;

    // 循环处理
    while (true) {
        int end = handle_packet(sock, client_addr, receive_queue);  // 接收数据包并处理
        if (end == -1)
            break;
    }

    closesocket(sock);
    WSACleanup();

    return 0;
}
```

3.9.2 代码分析

- **WSAStartup:** 程序开始时调用 **WSAStartup** 函数初始化 WinSock 库，以便使用 Windows 套接字 API。如果初始化失败，程序会输出错误信息并返回 -1 退出。
- **socket:** 使用 **socket** 函数创建一个 UDP 套接字，指定使用 IPv4 协议族 (**AF_INET**) 和 UDP 协议 (**SOCK_DGRAM**)。如果套接字创建失败，程序输出错误信息并返回 -1 退出。
- **server_addr:** 创建一个 **server_addr** 变量，并设置其为服务器的 IP 地址和端口。使用 **htons** 将端口号转换为网络字节顺序，使用 **inet_addr** 将 IP 地址字符串转换为网络字节序的二进制格式。
- **client_addr:** 创建一个 **client_addr** 变量，并设置其为客户端的 IP 地址和端口，类似于服务器地址设置。此处使用 **htons** 和 **inet_addr** 进行端口和地址转换。
- **bind:** 调用 **bind** 函数将套接字与服务器地址绑定。若绑定失败，程序输出错误信息并返回 -1 退出。
- **receive_queue:** 创建一个 **receive_queue** 队列，用于存储接收到的数据包。这个队列可以用于实现流量控制和滑动窗口机制，以确保数据包按顺序和不丢失地接收。
- **while 循环:** 程序通过一个无限循环来持续接收并处理数据包。每次循环中，调用 **handle_packet** 函数接收数据包并进行处理。如果返回值为 -1，则表示接收结束，程序跳出循环。
- **closesocket:** 关闭套接字，释放系统资源。
- **WSACleanup:** 在程序结束时，调用 **WSACleanup** 函数清理 WinSock 库，释放初始化时占用的资源。
- **handle_packet:** 它负责接收数据包并将其处理（如解包、存储或确认接收）。该函数返回一个整数，如果返回 -1，则表示数据包接收已结束，程序可以终止。

四、 编译指令

4.1 指令代码

```
g++ -o receiver.exe receiver.cpp -lws2_32
g++ -o sender.exe sender.cpp -lws2_32
```

4.2 指令含义

1. ‘g++ -o receiver.exe receiver.cpp -lws2_32‘

- ‘g++‘：GNU 编译器，用于编译 C++ 源代码。
- ‘-o receiver.exe‘：指定输出文件的名称为 ‘receiver.exe‘，这是编译后的可执行文件。
- ‘receiver.cpp‘：源代码文件，包含接收端程序的实现。
- ‘-lws2_32‘：链接 WinSock 库 ‘ws2_32.lib‘，这是 Windows 下用于网络编程的库，用于提供 socket 编程支持。

2. ‘g++ -o sender.exe sender.cpp -lws2_32‘

- ‘g++‘：同上，GNU 编译器。
- ‘-o sender.exe‘：指定输出文件的名称为 ‘sender.exe‘，这是编译后的发送端可执行文件。
- ‘sender.cpp‘：源代码文件，包含发送端程序的实现。
- ‘-lws2_32‘：同上，链接 ‘ws2_32.lib‘ 库，用于网络编程。

这两条指令分别用于编译接收端（‘receiver.exe‘）和发送端（‘sender.exe‘）的程序，并确保在 Windows 平台上能够正确进行网络通信。

五、 运行截图

在实验过程中，我们对协议的各个环节进行了测试和调试，确保每个模块正常工作。以下为实验运行截图，展示了文件传输过程中的关键时刻，例如超时重传和数据包的正常传输。

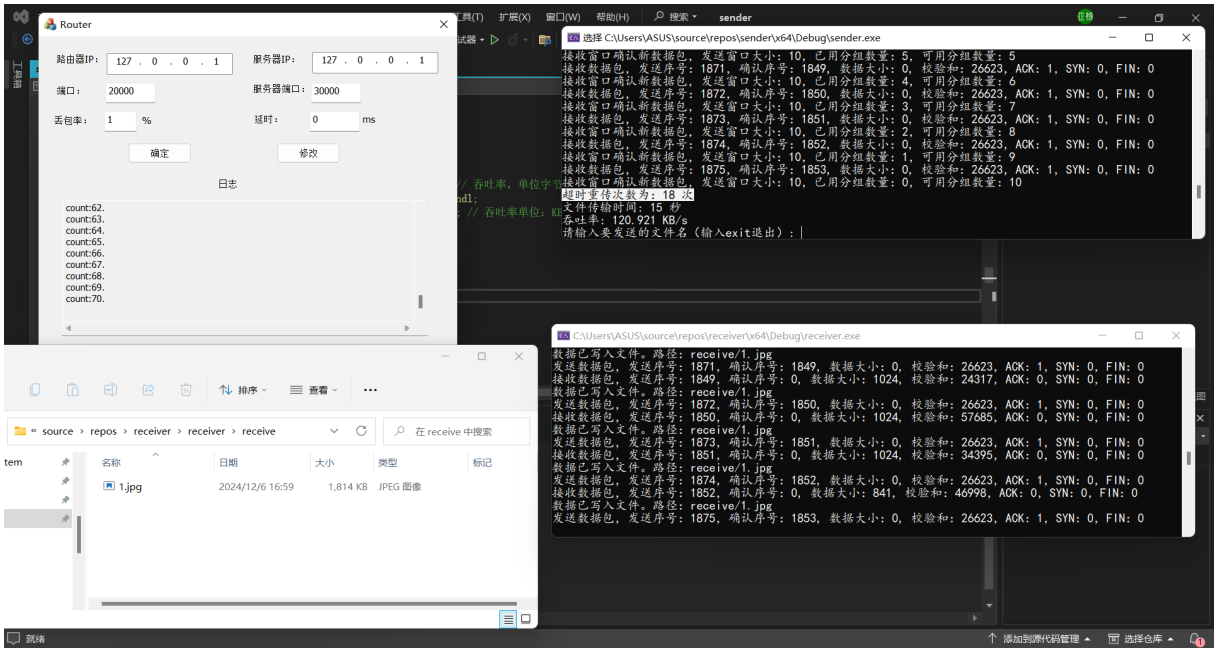

图 1: 文件传输超时重传

在此截图中，我们可以看到客户端在传输过程中遇到的超时重传情况。当数据包未能在规定时间内收到确认时，客户端会重新发送该数据包。这一机制确保了即使在不可靠的网络环境中，数据也能够被可靠传输。在图中，我们可以看到某些数据包经过多次重传，直到成功接收到确认包。

六、 结果分析

实验结果展示了协议在不同网络条件下的表现。我们主要考察了以下几个指标：吞吐率、文件传输时延以及网络拥塞情况下的表现。通过性能图表和数据，我们对协议的效果进行了详细分析。

在此图中，我们可以看到文件在传输过程中的大小变化。实验中，文件被分成若干个数据包进行传输，并且在每个数据包到达目标端后，文件的完整性得到了保证，文件大小始终与原始文件一致。这说明我们的协议能够有效地传输完整文件，并且未发生数据丢失或错误。

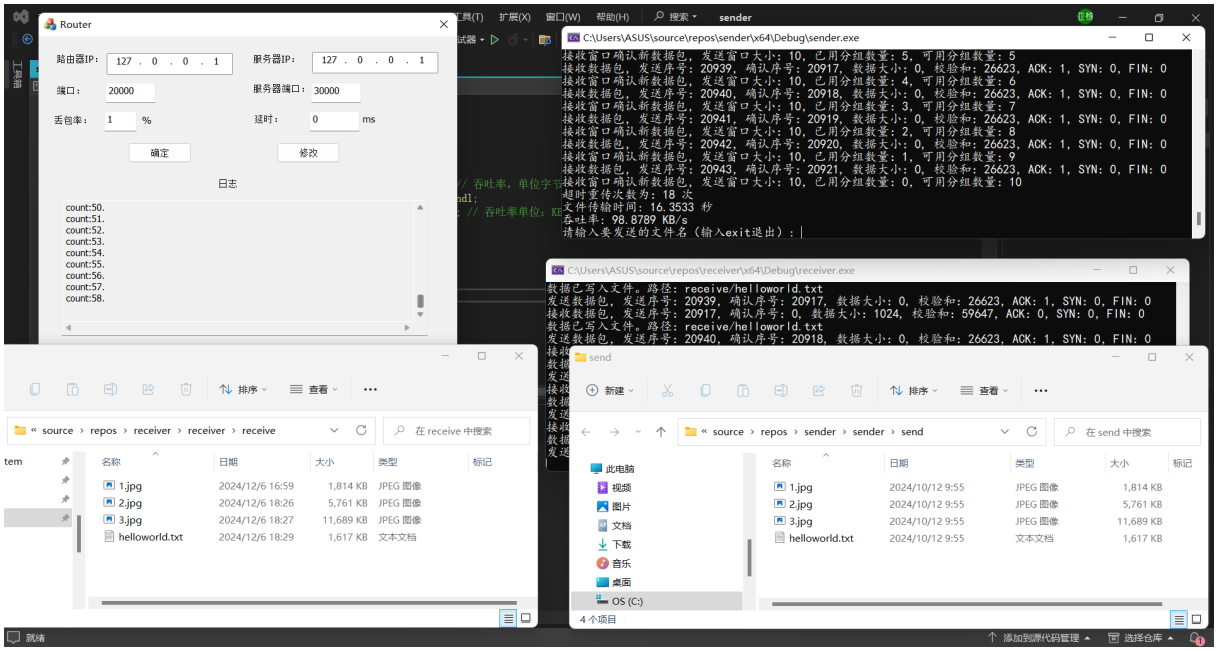

图 2: 文件大小保持一致

吞吐率方面，我们测试了协议在不同带宽和丢包率条件下的表现。实验结果表明，当带宽较高且丢包率较低时，吞吐率保持在较高水平。反之，当网络条件变差（例如丢包率增高或带宽下降）时，吞吐率会显著降低。这是由于重传机制和滑动窗口的工作方式，尤其是在拥塞较大的网络环境中，协议需要进行更多的重传来保证数据的可靠到达，导致传输效率降低。

文件传输时延是另一个重要的性能指标。我们通过测量文件从客户端发送到服务器所需的时间，发现随着网络延迟的增加，文件传输时延也有所增加。当网络质量较差时（例如高丢包率或较大的网络延迟），传输时延会显著增加，这主要是由于每个数据包的重传和确认过程需要更多的时间。通过优化重传机制和动态调整滑动窗口大小，可以进一步降低时延。

综上所述，实验结果表明，设计的协议在较好的网络条件下表现良好，但在网络不稳定或丢包严重的情况下，吞吐率和时延会受到较大影响。通过进一步优化协议的参数和重传机制，能够进一步提高协议的性能，适应更复杂的网络环境。

七、 实验总结

本次实验旨在设计并实现一个基于 UDP 的可靠数据传输协议，模拟了 TCP 协议的一些核心机制，如滑动窗口、三次握手、四次挥手以及重传机制，通过这些机制确保了数据的可靠性和完整性。通过本实验，我们不仅深入理解了 TCP 协议的工作原理，还掌握了如何在 UDP 上实现类似功能。

在协议设计部分，我们首先设计了数据包格式，确定了必要的控制字段，如序列号、确认号、校验和、同步标志（SYN）、确认标志（ACK）以及结束标志（FIN）。通过这些控制字段，能够有效地管理数据的传输过程，确保每一个数据包都能够正确无误地传递到目的地。

为了实现可靠传输，协议采用了滑动窗口机制进行流量控制和拥塞控制。通过控制滑动窗口的大小，协议能够有效避免网络中的过载，并且保证了数据包按序到达。对于数据包的丢失，我们设计了超时重传机制，确保在传输过程中丢失的数据能够及时重发。

在连接的建立和关闭过程中，我们模拟了 TCP 的三次握手和四次挥手过程。通过这种方式，协议能够在通信双方建立连接时进行协商，并在通信结束时正确地关闭连接。三次握手和四次挥手确保了连接的可靠性和终结的完整性，避免了资源的浪费。

在实验过程中，我们还进行了性能测试，包括吞吐率和文件传输时延的测量。实验结果表明，协议在一定的传输条件下能够提供稳定的吞吐率和较低的传输时延。然而，当网络拥塞或者丢包率较高时，传输效率会受到一定的影响。在这些情况下，重传机制和滑动窗口的调整起到了重要作用，帮助提高了协议的鲁棒性。

总的来说，本次实验通过实现基于 UDP 的可靠传输协议，帮助我们深入理解了传输层协议的工作原理，并为日后在网络编程中实现类似协议提供了实践经验。通过调试和优化协议，我们进一步提高了程序的性能和可靠性，为网络通信领域的研究和应用积累了宝贵的经验。