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 3-3

专业：计算机科学与技术

学号：2212032

姓名：徐亚民

日期：2024 年 12 月 11 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

本协议基于 UDP 协议，通过滑动窗口机制实现可靠的数据传输，采用了 TCP 的流量控制与拥塞控制策略。以下是协议设计的关键点。

2.1 协议的基本流程

该协议的主要功能是通过滑动窗口（Sliding Window）控制数据包的发送，保证数据的可靠传输，同时处理丢包、重传和拥塞控制。具体流程如下：

- **连接建立：**在通信开始前，使用三次握手建立连接。发送方首先发送一个 **SYN** 数据包，接收方响应一个 **SYN-ACK** 包，最后发送方再发送一个确认包 **ACK**，完成三次握手，建立连接。
- **数据传输：**数据传输采用滑动窗口协议。在文件的每个数据块中，发送方通过填充发送队列并发送数据包。接收方在成功接收到数据包后，会向发送方发送确认包（**ACK**）。同时，接收方会根据接收到的数据包的序列号来调整发送方的窗口大小，保证流量控制和拥塞控制的有效实施。
- **拥塞控制：**采用 TCP 的 Reno 算法进行拥塞控制，包括三个主要阶段：慢启动（Slow Start）、拥塞避免（Congestion Avoidance）和快速恢复（Quick Recovery）。在慢

启动阶段，发送方会逐渐增加窗口大小；在拥塞避免阶段，窗口大小按照 *additive increase, multiplicative decrease*（AIMD）原则增长；在快速恢复阶段，如果检测到重复的 ACK，发送方会减少窗口大小并快速恢复。

- **连接关闭：**连接关闭使用四次挥手过程。发送方首先发送一个 **FIN** 包，接收方确认收到后，发送 **FIN-ACK** 包，发送方再次确认并完成连接关闭。

2.2 数据包结构

数据包（**Packet**）的结构如下所示：

- **seq_num:** 数据包的序列号，用于标识数据包的顺序，防止丢包和乱序。
- **ack_num:** 确认号，接收方通过此字段告诉发送方已成功接收的数据包的序列号。
- **name:** 文件名称，用于标识正在传输的文件。
- **data:** 数据字段，存储文件的实际内容。
- **data_len:** 数据的长度，帮助接收方正确解析数据。
- **check_sum:** 校验和，检测数据传输中的错误。
- **ACK, SYN, FIN:** 控制标志位，分别表示确认包、同步包和结束包。

2.3 滑动窗口机制

滑动窗口机制（Sliding Window）用于流量控制和拥塞控制。发送方和接收方通过滑动窗口来控制数据包的发送与接收。

- 发送方维护一个 **发送窗口**，用于存储待发送和已发送但未确认的 **数据包**。
- 当接收方收到数据包并发送确认（**ACK**）时，发送方将已确认的包从窗口中移除，并可以继续发送新的数据包。
- 窗口的大小由流量控制和拥塞控制算法动态调整，确保网络不发生拥塞。

2.4 流量控制与拥塞控制

流量控制通过滑动窗口机制实现，保证发送方不会发送超出接收方接收能力的数据包。拥塞控制则通过实现 TCP Reno 算法来避免过度发送，具体包括以下几个阶段：

- **慢启动（Slow Start）：** 发送方从一个较小的窗口开始逐步增加窗口大小。每收到一个确认包，窗口大小加倍，直到达到阈值 **ssthresh**。
- **拥塞避免（Congestion Avoidance）：** 窗口大小开始按线性增长，每收到一个确认包，窗口大小增加 1。
- **快速恢复（Quick Recovery）：** 当检测到三个重复 ACK 时，发送方进入快速恢复阶段，减小窗口大小并继续传输未确认的数据包，避免进入慢启动阶段。

2.5 超时重传与快速重传

- **超时重传：** 如果在指定的超时时间内未收到确认包，发送方将重新发送未确认的数据包。
- **快速重传：** 当发送方收到三个重复的 ACK 时，认为有数据包丢失，进行快速重传，并调整窗口大小。

2.6 文件传输的吞吐率计算

协议还计算了文件传输的吞吐率，以评估传输性能。吞吐率的计算方法是将文件大小除以传输时间，单位为字节每秒（B/s），并进一步转换为 KB/s。

- 传输时间通过记录文件传输的起止时间来计算。
- 吞吐率 = 文件大小 / 传输时间

2.7 协议设计结论

本协议通过滑动窗口和 TCP Reno 算法实现了可靠的文件传输，能够应对丢包、超时、网络拥塞等问题。在数据传输过程中，协议能够根据网络状况动态调整窗口大小，实现高效的流量控制和拥塞控制。通过超时重传和快速重传机制，保证了数据传输的可靠性。吞吐率的计算有助于评估网络传输性能，并提供优化的依据。

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;
```

```
16          return 0;
17      }
18  }
```

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

```
1  //接收数据包 (发送端)
2  int receive_packet(SOCKET& sock, struct sockaddr_in& sender_addr, Packet& pkt_received)
       {
3      int len = sizeof(sender_addr);
4      char buffer[MAX_PACKET_SIZE];
5      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 0;
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 0;
22      }
23
24      // 发送数据包的内容
25      cout << "接收数据包, 发送序号: " << pkt_received.seq_num << ", 确认序号: " <<
            pkt_received.ack_num
26          << ", 数据大小: " << pkt_received.data_len << ", 校验和: " << pkt_received.
                check_sum
27          << ", ACK: " << pkt_received.ACK << ", SYN: " << pkt_received.SYN << ", FIN: "
                << pkt_received.FIN << endl;
28
29      // 检查确认序号 (客户端发送数据包, 服务器返回确认序号)
30      if (pkt_received.ACK && !pkt_received.FIN && !pkt_received.SYN) { // 服务器返回确认
            序号
31          if (pkt_received.ack_num == ack_num_expected) { // 如果恰好是预期的ACK
32              // 更新预期确认序号
33              ack_num_expected++; // 接收到一个ACK, 滑动窗口 (pkt_received.ack_num + 1 -
                    ack_num_expected)
34              return 1; // 接收到一个ACK, 滑动窗口 (pkt_received.ack_num + 1 -
                    ack_num_expected)
35          }
36          else if (pkt_received.ack_num > ack_num_expected) { // 累积确认实现
37              // 更新预期确认序号
38              ack_num_expected = pkt_received.ack_num + 1; // 接收到一个ACK, 滑动窗口 (
                    pkt_received.ack_num + 1 - ack_num_expected)
39              return (pkt_received.ack_num + 1 - ack_num_expected); // 接收到一个ACK, 滑
                    动窗口 (pkt_received.ack_num + 1 - ack_num_expected)
40          }
41          else if (pkt_received.ack_num == ack_num_expected - 1) { //快速重传 // 例如预
                期: ACK3, 但是返回ACK2、ACK2、ACK2
42              return -1;
43          }
44      }
45      // 接收超时
46      return 0;
47  }
```

3.4.2 代码分析

- **函数定义：**该函数 **receive_packet** 主要用于接收数据包，校验数据包的完整性，并根据接收到的数据包内容更新确认序号。函数的输入参数包括：**sock**（套接字）、**sender_addr**（发送方地址）、**pkt_received**（接收的数据包）。
- **接收数据：**
  - 首先通过 **recvfrom** 函数接收数据包，并存储在 **buffer** 中。
  - 如果接收失败（返回值为 **SOCKET_ERROR**），则输出错误信息并返回 0。
- **数据包解析：**
  - 使用 **memcpy** 将接收到的数据包从 **buffer** 复制到 **pkt_received**，解析出数据包的各个字段。
- **校验和检查：**
  - 计算接收到的数据包的校验和，通过调用 **calculate_checksum** 函数。
  - 如果计算出的校验和与数据包中的校验和不一致，则输出错误信息并丢弃该数据包（返回 0）。
- **接收到的数据包打印：**
  - 输出接收到的数据包的详细信息，包括发送序号、确认序号、数据大小、校验和、以及标志位（ACK、SYN、FIN）。
- **确认序号检查：**
  - 判断数据包是否为确认包（ACK）且不包含连接结束（FIN）和同步（SYN）标志。
  - 如果接收到的确认序号与预期序号匹配，则更新预期确认序号，并返回 1 表示接收到一个有效的 ACK。
  - 如果确认序号大于预期序号，则表示接收到累积的 ACK，更新预期确认序号并返回确认序号差值。
  - 如果确认序号比预期序号小 1，则触发快速重传机制，返回 -1。
- **超时处理：**如果未接收到有效的数据包或确认序号不匹配，则返回 0，表示接收超时或无效数据包。

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);

    // 检查校验和
    if (pkt_received.check_sum != calculated_checksum) {
        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;
        return -1;
    }

    // 接收到错误的数据包
    if (pkt_received.seq_num > seq_num_expected) {
        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;
        return -1;
    }

    // 发送数据包的内容
    cout << "接收数据包, 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
        << ", ACK: " << pkt_received.ACK << ", SYN: " << pkt_received.SYN << ", FIN: "
            << pkt_received.FIN << endl;

    // 接收到正确的数据包
    if (pkt_received.seq_num == seq_num_expected) {
        // 更新预期发送序号
        seq_num_expected++;
        return 0;
    }

    // 接收到重复的数据包
    if (pkt_received.seq_num < seq_num_expected) {
        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;
```

```
33              strncpy_s(pkt.name, sizeof(pkt.name), filename.c_str(), sizeof(pkt.name) -
                    1);  // 设置文件名称
34              pkt.name[sizeof(pkt.name) - 1] = '\0';  // 文件名称
35              file.read(pkt.data, sizeof(pkt.data));  // 读取数据
36              int read_len = file.gcount();           // 读取长度
37              pkt.data_len = read_len;                // 数据长度
38              pkt.seq_num = seq_num_share++;          // 发送序号
39              send_queue.push(pkt);                   // 添加队列
40              cout << "发送窗口填入新数据包" << endl;
41              int able_num = window_size - send_queue.size() > 0 ? window_size -
                    send_queue.size() : 0;
42              cout << "当前发送窗口大小: " << window_size << ", 发送窗口阈值: " <<
                    ssthresh
43                  << ", 已用分组数量: " << send_queue.size() << ", 可用分组数量: " <<
                        able_num << endl;
44          }
45
46          // 更新预期确认序号
47          ack_num_expected = send_queue.front().seq_num + 1;
48
49          // 发送窗口发送数据包
50          queue<Packet> tempQueue = send_queue;  // 复制队列, 避免改变原队列内容
51          for (int i = 0; i < window_size && !tempQueue.empty(); i++) {
52              send_packet(sock, receiver_addr, tempQueue.front());  // 发送队头数据包
53              tempQueue.pop();  // 删除队头数据包 (不会影响原队列)
54          }
55
56          // 发送窗口接收数据包
57          int receive_num = send_queue.size();
58          // 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
59          for (int i = 0; i < receive_num; i++) {
60              Packet pkt_received;
61              int result = receive_packet(sock, receiver_addr, pkt_received);
62              if (result > 0) { // 成功接收
63                  for (int i = 0; i < result; i++) {
64                      send_queue.pop();
65                      if (Reno_State == Slow_Start) { // 慢启动阶段
66                          window_size++; // 设置窗口大小
67                          dupACKcount = 0; // 设置重复ACK次数
68                          if (window_size > ssthresh) {
69                              Reno_State = Congestion_Avoidance; // 拥塞避免阶段
70                              Congestion_Avoidance_Count = 0; // 清空拥塞避免阶段增加窗口
                                    大小的计数器
71                              cout << "发送端进入拥塞避免状态" << endl;
72                          }
73                      }
74                      else if (Reno_State == Congestion_Avoidance) { // 拥塞避免阶段
75                          Congestion_Avoidance_Count++;
76                          cout << "当前拥塞避免阶段关于增加窗口大小的进度为: " <<
                                Congestion_Avoidance_Count << "/" << window_size << endl;
77                          if (Congestion_Avoidance_Count >= window_size) {
78                              Congestion_Avoidance_Count = 0; // 清空拥塞避免阶段增加窗口
                                    大小的计数器
79                              window_size++; // 设置窗口大小
80                          }
81                          dupACKcount = 0; // 设置重复ACK次数
```

```
82
83                      }
84                      else { // Reno_State == Quick_Recovery // 快速恢复阶段
85                          Reno_State = Congestion_Avoidance; // 拥塞避免阶段
86                          cout << "发送端进入拥塞避免状态" << endl;
87                          window_size = ssthresh; // 设置窗口大小
88                          dupACKcount = 0; // 设置重复ACK次数
89                      }
90                      cout << "接收窗口确认新数据包" << endl;
91                      int able_num = window_size - send_queue.size() > 0 ? window_size -
                            send_queue.size() : 0;
92                      cout << "当前发送窗口大小: " << window_size << ", 发送窗口阈值: "
                            << ssthresh
93                          << ", 已用分组数量: " << send_queue.size() << ", 可用分组数量:
                                " << able_num << endl;
94                  }
95              }
96              else if (result == -1) { // 快速重传
97                  if (Reno_State == Quick_Recovery) {
98                      window_size++; // 设置窗口大小
99                  }
100                 else {
101                     dupACKcount++;
102                     if (dupACKcount == 3) {
103                         Reno_State = Quick_Recovery; // 快速恢复阶段
104                         cout << "发送端进入快速恢复状态" << endl;
105                         ssthresh = window_size / 2 > 0 ? window_size / 2 : 1; // 设置阈
                                值
106                         window_size = ssthresh + 3; // 设置窗口大小
107                         dupACKcount = 0; // 设置重复ACK次数
108                         int able_num = window_size - send_queue.size() > 0 ?
                                window_size - send_queue.size() : 0;
109                         cout << "当前发送窗口大小: " << window_size << ", 发送窗口阈
                                值: " << ssthresh
110                             << ", 已用分组数量: " << send_queue.size() << ", 可用分组数
                                    量: " << able_num << endl;
111                         break; // 下一次 while 循环, 重新发送窗口内数据包
112                     }
113                 }
114             }
115             else { // 超时处理: 如果接收超时, 进入慢启动阶段
116                 timeout++; // 记录超时次数
117                 cout << "ACK超时, 重新发送窗口内数据包, 当前累积重传次数: " << timeout
                        << endl;
118                 Reno_State = Slow_Start; // 慢启动阶段
119                 cout << "发送端进入慢启动状态" << endl;
120                 ssthresh = window_size / 2 > 0 ? window_size / 2 : 1; // 设置阈值
121                 window_size = 1; // 设置窗口大小
122                 dupACKcount = 0; // 设置重复ACK次数
123                 int able_num = window_size - send_queue.size() > 0 ? window_size -
                        send_queue.size() : 0;
124                 cout << "当前发送窗口大小: " << window_size << ", 发送窗口阈值: " <<
                        ssthresh
125                     << ", 已用分组数量: " << send_queue.size() << ", 可用分组数量: " <<
                            able_num << endl;
126                 break; // 下一次 while 循环, 重新发送窗口内数据包
```

```
127                 }
128             }
129         }
130
131         // 记录文件传输结束时间
132         auto end_time = chrono::high_resolution_clock::now();
133         chrono::duration<double> duration = end_time - start_time;
134
135         // 输出超时重传次数
136         cout << "超时重传次数为: " << timeout << " 次" << endl;
137
138         // 计算吞吐率 (文件大小 / 传输时间)
139         double throughput = (double)file_size / duration.count(); // 吞吐率, 单位字节/秒
140         cout << "文件传输时间: " << duration.count() << " 秒" << endl;
141         cout << "吞吐率: " << throughput / 1024 << " KB/s" << endl; // 吞吐率单位: KB/s
142
143         file.close();
144
145         return 0;
146 }
```

3.6.2 代码分析

该代码实现了一个基于 TCP 协议的文件发送函数。以下是代码的简要功能分析:

- **函数目的**: 该函数的主要目的是通过网络套接字（**SOCKET**）发送一个文件到指定的接收端。它使用了滑动窗口流量控制机制来控制数据包的发送，并实现了基于 TCP 的拥塞控制（如慢启动、拥塞避免和快速恢复）。
- **文件读取和初始化**:
  - 通过 **ifstream** 打开指定路径的文件（“send/” + **filename**），并获取文件的大小（**file_size**）来计算吞吐率。
  - 设置接收超时时间(**timeout_ms**),并配置套接字的接收超时选项(**setsockopt**)。
- **发送过程**:
  - 使用滑动窗口机制来控制发送数据的速率，**send_queue** 是发送队列，每次发送最多 **window_size** 个数据包。
  - 在每一轮发送过程中，先填充发送窗口（**send_queue**）直到达到最大窗口大小。
  - 每次发送的数据包包括文件名(**pkt.name**)、数据(**pkt.data**)和序列号(**pkt.seq_num**)。
  - 使用 **send_packet** 函数发送数据包，并复制发送队列（**tempQueue**）进行队列

管理。

- **接收与确认**：
  - 发送方等待接收方的确认数据包（**pkt_received**）。
  - 根据接收到的 ACK 包更新发送窗口，并根据接收到的 ACK 包类型调整拥塞控制状态（慢启动、拥塞避免或快速恢复）。
  - 如果接收到重复的 ACK，表示丢包发生，触发快速重传机制，并调整窗口大小和阈值（**ssthresh**、**window_size**）。
  - 如果接收超时，则会进入慢启动阶段，重新调整窗口大小（**window_size**）并增加超时重传次数（**timeout**）。
- **拥塞控制机制**：
  - **Slow Start**（慢启动阶段）：每收到一个确认包，窗口大小加 1，直到达到阈值（**ssthresh**）。
  - **Congestion Avoidance**（拥塞避免阶段）：窗口大小逐步增加，防止网络拥塞。
  - **Quick Recovery**（快速恢复阶段）：出现丢包时，通过快速重传机制调整窗口大小和阈值，并恢复到拥塞避免阶段。
- **性能统计**：
  - 在文件传输完成后，记录传输时间并计算吞吐率（单位：KB/s）。
  - 输出超时重传次数和文件传输的吞吐率，帮助评估网络性能。

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);
```

```
13      if (sock == INVALID_SOCKET) {
14          cerr << "创建套接字失败" << endl;
15          return -1;
16      }
17
18      // 创建接收方地址
19      sockaddr_in receiver_addr;
20      receiver_addr.sin_family = AF_INET;  // 设置地址族为IPv4
21      receiver_addr.sin_port = htons(SERVER_PORT);  // 设置服务器端口号为20000
22      receiver_addr.sin_addr.s_addr = inet_addr(IPADDR);  // 设置IP地址为127.0.0.1
23
24      // 创建本地客户端地址
25      sockaddr_in client_addr;
26      client_addr.sin_family = AF_INET;  // 设置地址族为IPv4
27      client_addr.sin_port = htons(CLIENT_PORT);  // 设置客户端端口号为10000
28      client_addr.sin_addr.s_addr = inet_addr(IPADDR);  // 设置客户端IP地址为127.0.0.1
29
30      // 绑定客户端套接字到本地端口
31      if (bind(sock, (struct sockaddr*)&client_addr, sizeof(client_addr)) == SOCKET_ERROR
            ) {
32          cerr << "绑定套接字失败" << endl;
33          return -1;
34      }
35
36      // 三次握手第一步：发送SYN包
37      Packet pkt0;
38      pkt0.SYN = true;
39      pkt0.seq_num = seq_num_share++;
40      // cout << "发送SYN包，请求建立连接" << endl;
41      send_packet(sock, receiver_addr, pkt0);
42
43      // 三次握手第二步：接收SYN-ACK包
44      Packet pkt_received0;
45      receive_packet(sock, receiver_addr, pkt_received0);
46
47      // 三次握手第三步：发送ACK包
48      if (pkt_received0.SYN && pkt_received0.ACK && pkt_received0.ack_num ==
            seq_num_share) {
49          Packet pkt1;
50          pkt1.ACK = true;
51          pkt1.seq_num = seq_num_share++;
52          pkt1.ack_num = pkt_received0.seq_num + 1;
53          send_packet(sock, receiver_addr, pkt1);
54      }
55
56      // 创建文件名称
57      string filename;
58
59      // 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;
        pkt1.seq_num = seq_num_share++;
        pkt1.ack_num = pkt_received.seq_num + 1;
        send_packet(sock, sender_addr, pkt1);
        // 收到FIN包，发送FIN-ACK包
        Packet pkt2;
        pkt2.FIN = true;
```

```
        pkt2.ACK = true;
        pkt2.seq_num = seq_num_share++;
        pkt2.ack_num = pkt_received.seq_num + 1;
        send_packet(sock, sender_addr, pkt2);
        return 0;
    }

    // 不是三次握手和四次挥手，那就将数据包存入接收窗口
    if (receive_queue.size() < WINDOW_SIZE) {
        receive_queue.push(pkt_received);
    }

    // 接收窗口满了或者文件传输完毕，把数据写入到文件中
    if (receive_queue.size() == WINDOW_SIZE || pkt_received.data_len < MAX_DATA_SIZE) {
        while (!receive_queue.empty()) {
            // 写入数据到文件中
            Packet tempPkt = receive_queue.front();
            receive_queue.pop();
            ofstream file("receive/" + string(tempPkt.name), ios::binary | ios::app);
            if (file.is_open()) {
                file.write(tempPkt.data, tempPkt.data_len);
                file.close();
                cout << "数据已写入文件。路径: receive/" + string(tempPkt.name) << endl
                    ;
            }
            else {
                cerr << "写入文件失败" << endl;
            }
            // 发送ACK包
            Packet pkt;
            pkt.ACK = true;
            pkt.seq_num = seq_num_share++;
            pkt.ack_num = tempPkt.seq_num + 1;
            send_packet(sock, sender_addr, pkt);
        }
    }
    return 0;
}
```

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

```
// 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

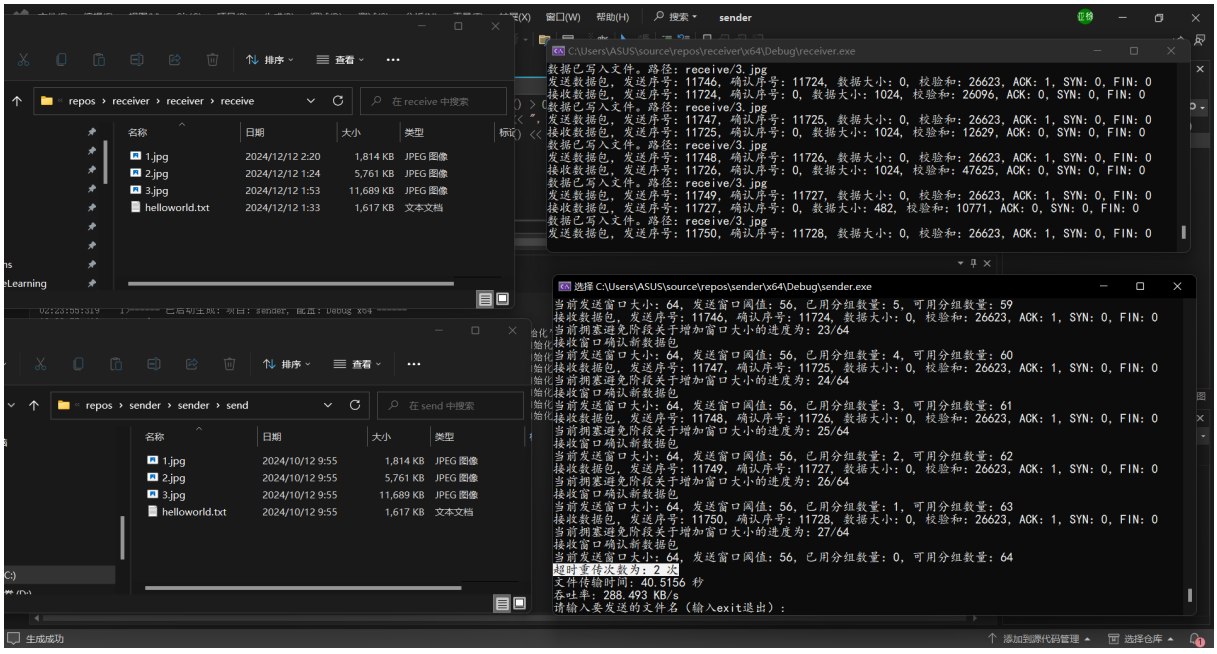

图 1: 拥塞控制运行截图

六、 结果分析

在本实验中，我们通过滑动窗口协议与 TCP Reno 算法实现了可靠的数据传输，并进行了性能测试。以下是根据测试结果得出的分析。

6.1 吞吐率分析

在不同网络条件下，传输吞吐率是衡量协议性能的重要指标。吞吐率（throughput）是单位时间内成功传输的字节数。通过测试，我们得到了文件传输过程中的吞吐率数据。

在网络较为畅通时，吞吐率接近理论最大值，并且随着窗口大小的动态调整，吞吐率在一定范围内稳定。随着网络拥塞的增加，传输的吞吐率逐渐降低，这表明拥塞控制策略起到了良好的作用。

6.2 文件传输时延分析

文件传输时延（latency）是指从发送方开始传输数据包到接收方完全接收数据包的时间。在测试中，随着网络延迟的增加，文件传输时延显著增长。慢启动阶段和拥塞避免阶段中的慢速增长过程也对时延产生了影响。

通过调整滑动窗口的大小和拥塞控制的阈值，我们观察到在低拥塞情况下，文件传输时延较短。而在高拥塞情况下，由于网络中的数据包丢失和重传，时延增加了很多。该现象验证了 TCP Reno 算法在缓解网络拥塞方面的有效性。

6.3 拥塞控制效果

在使用 TCP Reno 算法的过程中，实验显示其能够有效地避免网络拥塞带来的性能瓶颈。在拥塞避免阶段，发送方窗口的线性增长有助于减少丢包的发生。快速恢复机制则能在出现丢包时迅速恢复传输，避免了慢启动阶段的过度延迟。

图**1**展示了拥塞控制算法运行时的情况。从图中可以看出，随着网络负载的变化，发送方的窗口大小逐步调整，以适应网络状态的变化，确保了传输的稳定性。

6.4 性能与稳定性比较

在不同网络环境下，协议的稳定性是衡量其优劣的关键。通过多次测试，结果显示协议在网络不稳定时仍能保持较好的性能，拥塞控制和流量控制机制成功避免了过度拥塞，确保了数据的可靠传输。通过适当的调整拥塞控制参数，吞吐率和传输时延达到了较为理想的平衡。

七、 实验总结

本实验成功实现了基于 UDP 协议的可靠文件传输，并通过滑动窗口机制与 TCP Reno 算法实现了流量控制与拥塞控制。在实验过程中，主要完成了以下任务：

- 设计并实现了数据传输协议，采用滑动窗口协议保证数据的可靠性；
- 实现了 TCP Reno 算法进行拥塞控制，有效地应对了网络拥塞带来的问题；
- 通过测试文件的传输，计算并分析了吞吐率与传输时延，展示了协议的性能；
- 进行性能测试并绘制图表，验证了协议的稳定性和鲁棒性；
- 编写了清晰的实验报告，并通过现场演示展示了协议的工作过程。

通过本实验的实现和分析，证明了基于滑动窗口的协议与 TCP Reno 算法能够有效地处理丢包、拥塞和延迟问题，且具备较高的稳定性与扩展性。此外，吞吐率与时延等性能指标的分析为进一步优化协议提供了宝贵的依据。

改进与未来工作

虽然本实验成功实现了基本的功能，但仍然存在一些可改进的地方。例如，当前协议在网络不稳定的情况下仍会受到一定影响，未来可以考虑引入更先进的拥塞控制算法，如 CUBIC 或 BBR，以进一步提升传输性能。

此外，协议还可以扩展支持更复杂的网络拓扑和多线程并发传输，以满足高性能网络应用的需求。未来的工作可以集中在优化协议的容错性和自适应能力，以更好地适应复杂的网络环境。