



南開大學
Nankai University

计算机学院
计算机网络实验报告

实验 2：设计可靠传输协议并编程实现

姓名：王旭
学号：2312166
专业：计算机科学与技术

2025 年 12 月 26 日

目录

1 实验要求	3
2 协议设计	3
2.1 协议概述	3
2.2 数据包头部设计	3
2.3 连接管理	4
2.3.1 连接建立（三次握手）	4
2.3.2 连接关闭（四次挥手）	5
2.4 差错检测	5
2.5 确认重传机制	5
2.6 流量控制	5
2.7 拥塞控制	6
2.8 数据传输流程	6
2.9 异常处理	6
2.10 自实现的丢包率和延时的设计	7
2.10.1 丢包率模拟设计	7
2.10.2 网络延时模拟设计	7
2.10.3 参数配置和使用示例	8
2.11 整体交互流程	8
2.11.1 连接建立阶段	8
2.11.2 数据传输阶段（流水线方式）	8
2.11.3 连接关闭阶段	9
3 代码实现与分析	9
3.1 Receiver 接收端:receiver.cpp	9
3.1.1 全局变量和缓冲区	9
3.1.2 发送 ACK 函数	10
3.1.3 主函数和参数处理	10
3.1.4 握手和挥手逻辑	11
3.1.5 数据处理和流量控制	12
3.1.6 Receiver 端的整体工作流程	13
3.2 Sender 发送端: sender.cpp	14
3.2.1 全局变量和状态管理	14
3.2.2 发送数据包函数	15
3.2.3 握手函数	16
3.2.4 文件加载函数	17
3.2.5 挥手函数	18
3.2.6 主函数: 参数处理和初始化	19
3.2.7 主函数: 数据传输循环	20
3.2.8 主函数: 统计输出信息	23
3.2.9 Sender 端的整体工作流程	23

4 结果演示	24
4.1 0 丢包率与延时下的传输测试	24
4.2 3% 丢包率下的传输测试	25
4.3 3% 丢包率与 5ms 延时下的传输测试	26
4.4 窗口大小设为 5 的传输测试	27
5 总结	28
6 github 仓库源码	28

1 实验要求

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：

1. 连接管理：包括建立连接、关闭连接和异常处理。
2. 差错检测：使用校验和进行差错检测。
3. 确认重传：支持流水线方式，采用选择确认。
4. 流量控制：发送窗口和接收窗口使用相同的固定大小窗口。
5. 拥塞控制：实现 RENO 算法。

2 协议设计

2.1 协议概述

本次我们设计的该协议命名为 **RDT (Reliable Data Transfer)**，基于 UDP 数据报套接字在用户空间实现面向连接的可靠数据传输。协议设计在建立连接部分参考了 TCP 的核心机制并进行了适当简化实现，以适应实验要求。协议支持单向数据传输（发送方到接收方），控制信息双向交互。整体架构包括连接管理、差错检测、确认重传、流量控制和拥塞控制五个核心功能模块。

协议采用客户端-服务器模型：

- **发送方 (Sender)**：负责文件分片、打包、发送、拥塞控制和超时重传。
- **接收方 (Receiver)**：负责接收、校验、缓存、确认和流量控制。

协议使用自定义的数据包结构，通过序列号和确认号实现可靠性，通过窗口机制和 RENO 算法实现流量和拥塞控制。

2.2 数据包头部设计

数据包头部 (PacketHeader) 是协议的核心，定义在 `rdt.h` 文件中。头部采用 1 字节对齐 (`#pragma pack(push, 1)`) 以确保网络传输的一致性，避免编译器填充字节导致的字节序问题。

数据包头部定义

```

1 struct PacketHeader {
2     uint32_t seq;           // 序列号：标识数据包顺序，从 1 开始递增
3     uint32_t ack;          // 确认号：用于 SR 选择确认，确认收到的包序列号
4     uint16_t flags;         // 标志位：SYN (0x01)、ACK (0x02)、FIN (0x04)
5     uint16_t checksum;      // 校验和：16 位反码求和，用于差错检测
6     uint16_t length;        // 数据长度：数据部分字节数，最大 1024 字节 (MSS)
7     uint16_t window;        // 窗口大小：保留字段，用于扩展（当前未使用）
8     uint32_t padding;       // 填充：确保头部总长 20 字节
9 };
10
11 // 标志位
12 const uint16_t FLAG_SYN = 0x01;      // 同步标志，用于连接建立

```

```

13 const uint16_t FLAG_ACK = 0x02;      // 确认标志, 用于确认收到数据
14 const uint16_t FLAG_FIN = 0x04;      // 结束标志, 用于断开连接

```

- `uint32_t seq` : 32 位无符号整数, 数据包从 1 开始编号, 握手包为 0。用于标识包的顺序和检测丢失。
- `uint32_t ack` : 32 位无符号整数, 在 SR 模式下, ack 字段设置为**收到的包序列号**, 比如收到数据包 3, 那么回复的 ACK 包的 ack 即为 3, 实现选择确认。
- `uint16_t flags` : 16 位, 使用位掩码表示包类型:
 - `SYN (0x01)` : 连接建立同步标志。
 - `ACK (0x02)` : 确认标志。
 - `FIN (0x04)` : 连接关闭标志。
- `uint16_t checksum` : 16 位, 使用标准互联网校验和算法 (反码求和), 覆盖头部和数据部分。
- `uint16_t length` : 16 位, 指示数据部分的字节数。
- `uint16_t window` : 16 位, 保留用于流量控制扩展。
- `uint32_t padding` : 32 位, 确保头部固定 20 字节, 便于解析。

数据包结构 (Packet) 由头部和数据部分组成:

数据包结构

```

1 struct Packet {
2     PacketHeader header;
3     char data[MSS]; // 数据缓冲区, 最大 1024 字节
4 }

```

2.3 连接管理

协议实现面向连接的传输, 使用三次握手建立连接, 四次挥手关闭连接。所有控制包 (握手、挥手) 不携带数据 (`length=0`), 仅用于状态同步。

2.3.1 连接建立 (三次握手)

1. 客户端发送 SYN: `seq=0, flags=SYN`。
2. 服务器响应 SYN+ACK: `seq=0, ack=1, flags=SYN|ACK`。
3. 客户端发送 ACK: `seq=1, ack=1, flags=ACK`。

握手成功后, 连接状态置为 `connected=true`, 数据传输开始。握手使用 `select` 函数设置 2 秒超时, 避免死锁。

2.3.2 连接关闭（四次挥手）

1. 客户端发送 FIN: seq= 最后数据包的 seq 号 +1, flags=FIN。
2. 服务器响应 ACK: ack=FIN 包的 seq+1, flags=ACK。
3. 服务器关闭连接, 客户端等待 2 秒后清理资源。

异常处理: 握手失败或超时直接退出程序。

2.4 差错检测

使用 16 位校验和进行差错检测。算法实现 (`calculate_checksum`) 遵循 RFC 1071:

- 将头部和数据视为 16 位字序列, 反码求和。
- 处理奇数长度数据。
- 最终取反得到校验和。

接收方收到包后, 首先计算校验和并与包内校验和比较。若不匹配, 丢弃包并输出 "[Checksum Error]"。

2.5 确认重传机制

采用 选择性重传 (Selective Repeat, SR) 机制, 支持流水线传输:

- **发送窗口**: 大小固定 (默认 20), 由 `base` 和 `nextSeqNum` 维护。
- **接收窗口**: 大小固定 (默认 20), 由 `expectedSeq` 和缓冲区维护。
- **确认策略**: 接收方对每个收到的包 (无论顺序) 立即发送 ACK, `ack` 字段设置为包序列号。
- **重传触发**:
 - **超时重传**: 发送方为每个包记录 `sendTime`, 若 `base` 包超过 1000ms 未确认, 重传该包。
 - **快速重传**: 收到 3 个重复 ACK 时, 重传 `base` 包 (不等待超时)。

发送缓冲区 (`vector<SenderPacket>`) 记录每个包的状态 (`sent`, `acked`, `sendTime`)。接收缓冲区 (`map<uint32_t, Packet>`) 缓存乱序包。

2.6 流量控制

发送窗口和接收窗口使用相同的固定大小 (通过命令行参数配置, 默认 20)。流量控制通过窗口边界强制执行:

- **发送方**: 窗口大小 = $\min(\text{cwnd}, \text{maxWindowSize})$, 限制未确认包数量。
- **接收方**: 显式检查 `if (seq >= expectedSeq + rcvWindowSize)`, 超出窗口的包直接丢弃且不发送 ACK, 迫使发送方停止发送。

2.7 拥塞控制

实现 **TCP RENO** 算法，使用状态机管理拥塞窗口 (cwnd) 和慢启动阈值 (ssthresh)：

- **状态**: SLOW_START, CONGESTION_AVOIDANCE, FAST_RECOVERY。
- **慢启动**: cwnd 从 1 开始，每收到 ACK 则 $+1$ (**指数增长**)，直到 $\geq ssthresh$ 。
- **拥塞避免**: 每收到 ACK, $cwnd += 1/cwnd$ (**线性增长**)。
- **快速恢复**:
 - 如果在慢启动/拥塞避免阶段收到 3 重复 ACK: $ssthresh = cwnd/2$, $cwnd = ssthresh + 3$, 进入 FAST_RECOVERY。
 - 收到重复 ACK: $cwnd += 1$ 。
 - 收到新 ACK: $cwnd = ssthresh$, 回到 CONGESTION_AVOIDANCE。
- **超时处理**: $ssthresh = cwnd/2$, $cwnd = 1$, 回到 SLOW_START。

2.8 数据传输流程

1. **初始化**: 握手建立连接，加载文件并分包。
2. **发送循环**:
 - 发送窗口内未发包。
 - 等待 ACK (10ms 超时)。
 - 处理 ACK: 更新窗口、状态机。
 - 检查超时: 重传并重置状态。
3. **接收循环**:
 - 接收包，校验和检查。
 - 流量控制检查。
 - 发送 ACK。
 - 顺序包写入文件，乱序包缓存。
4. **结束**: 挥手关闭连接，输出传输时间 `totalTimeSec` 和吞吐率 `Throughput`。

2.9 异常处理

- **校验和错误**: 丢弃包。
- **握手超时**: 退出程序。
- **窗口溢出**: 丢弃包 (流量控制)。
- **超时**: 重传并拥塞控制。
- **重复包**: 重发 ACK (SR 特性)。

2.10 自实现的丢包率和延时的设计

为了验证协议在不同网络环境下的性能，我们在 Sender 发送方实现了 丢包率和网络延时 的模拟功能，允许用户通过命令行参数配置模拟参数，便于观察不同丢包率对拥塞控制的影响，以及延时对传输时间的影响。

2.10.1 丢包率模拟设计

- **目的：** 模拟真实网络中的包丢失现象，测试协议的重传机制和拥塞控制算法的有效性。
- **实现方式：** 在 `send_packet` 函数中，使用随机数生成器模拟丢包。代码片段如下：

丢包率模拟代码

```

1 if (packetLossRate > 0.0 && ((rand() % 1000) / 1000.0 < packetLossRate)) {
2     // 模拟丢包：不调用 sendto，但逻辑上标记为已尝试发送
3     if (!sp.sent) {
4         sp.sent = true;
5         sp.sendTime = chrono::steady_clock::now();
6     }
7     return;
8 }
```

- `packetLossRate` 为丢包率（0.00 到 1.00），通过命令行参数 `[loss_rate]` 配置。
- 使用 `(rand() % 1000) / 1000.0` 生成 0 到 1 之间的随机数，若小于 `packetLossRate`，则模拟丢包。
- 丢包仅针对数据包，不影响握手和挥手包，以确保连接管理不受干扰。
- **影响分析：**高丢包率会导致更多超时和快速重传，`cwnd` 频繁重置，吞吐率下降。日志输出 `[Timeout]` 或 `[Fast Retransmit]` 可观察到拥塞控制的响应。

2.10.2 网络延时模拟设计

- **目的：** 模拟网络传播延时，测试协议对高延迟环境的适应性，观察对传输时间和吞吐率的影响。
- **实现方式：** 在 `send_packet` 函数中，使用线程睡眠模拟延时。代码片段如下：

网络延时模拟代码

```

1 if (delayMs > 0) {
2     this_thread::sleep_for(chrono::milliseconds(delayMs));
3 }
```

- `delayMs` 为模拟延时（毫秒），通过命令行参数 `[delay_ms]` 配置。
- 使用 `std::this_thread::sleep_for` 在发送前阻塞线程，模拟单向网络延时。
- **影响分析：** 延时增加会导致 ACK 往返时间 (RTT) 延长，超时阈值 (1000ms) 可能触发更多重传，降低吞吐率。结合丢包，可模拟复杂网络条件。

2.10.3 参数配置和使用示例

- **命令行参数:** Sender 支持可选参数 [loss_rate] [window_size] [delay_ms]。

- **示例:**

1. 无模拟: `sender.exe 127.0.0.1 8080 received_final.txt`
2. 模拟 5% 丢包: `sender.exe 127.0.0.1 8080 received_final.txt 0.05`
3. 模拟 5ms 延时: `sender.exe 127.0.0.1 8080 received_final.txt 0.0 20 5`
4. 综合模拟: `sender.exe 127.0.0.1 8080 received_final.txt 0.03 20 5`

- **性能观察:** 程序输出传输时间和吞吐率，用户可通过多次运行对比不同参数下的性能变化。

通过自实现丢包率与延时机制，增强了协议的测试能力，确保在模拟环境下验证可靠性机制的有效性。

2.11 整体交互流程

协议的整体交互流程从连接建立到关闭，涵盖所有模块的协同工作。

2.11.1 连接建立阶段

1. **Sender → Receiver:** SYN (seq=0, flags=SYN)
2. **Receiver → Sender:** SYN+ACK (seq=0, ack=1, flags=SYN|ACK)
3. **Sender → Receiver:** ACK (seq=1, ack=1, flags=ACK)
4. 连接建立，双方进入数据传输状态。

2.11.2 数据传输阶段（流水线方式）

1. Sender 发送数据包: seq=1,2,3,... (窗口大小限制发送数量)。
2. Receiver 接收包后立即发送选择确认的 ACK 包: ack=1,2,3,...。
3. Sender 根据 ACK 滑动窗口，继续发送新包。
4. 若发生丢包:
 - Receiver 收到乱序包，缓存并 ACK。
 - Sender 收到重复 ACK 或超时，重传丢失包。
5. 拥塞控制: Sender 根据 ACK 调整 cwnd，进入不同状态 (慢启动 → 拥塞避免 → 快速恢复)。

2.11.3 连接关闭阶段

1. Sender → Receiver: FIN (seq= 最后包 +1, flags=FIN)
2. Receiver → Sender: ACK (ack=FIN seq+1, flags=ACK)
3. Receiver 关闭套接字。
4. Sender 等待 2 秒后关闭。

整个流程通过 UDP 实现双向控制信息交互，确保单向数据传输的可靠性。协议状态机确保在任何时候时刻都能正确处理异常，如超时或校验错误。

此协议设计确保了可靠性、高效性和符合实验要求，代码实现清晰可读。

3 代码实现与分析

3.1 Receiver 接收端:receiver.cpp

`receiver.cpp` 文件实现接收端的逻辑，负责接收数据包、校验、缓存、确认和流量控制，体现实验要求的差错检测、确认重传和流量控制功能。

3.1.1 全局变量和缓冲区

接收端使用全局变量管理状态和缓冲区：

全局变量和缓冲区

```

1 SOCKET sock;                                // Socket, 用于 UDP 通信
2 sockaddr_in clientAddr;                      // 客户端地址 (IP + 端口), 用于发送 ACK
3 int addrLen = sizeof(clientAddr);            // 地址长度, 用于 recvfrom 和 sendto 函数
4 bool connected = false;                       // 连接状态, 确保握手完成后才处理数据包
5 int rcvWindowSize = 20;                        // 接收窗口大小 (默认 20), 可通过命令行参数修改
6
7 // 接收缓冲区 (用于乱序重排)
8 map<uint32_t, Packet> recvBuffer;           // 序列号 到 数据包 Packet 的映射
9 uint32_t expectedSeq = 1;                     // 期望收到的下一个序列号, 数据包从 1
10      开始, 握手包序列号为 0
11
12 ofstream outFile;

```

- `SOCKET sock`：用于 UDP 通信的套接字。
- `sockaddr_in clientAddr`：客户端地址 (IP + 端口)，用于发送 ACK。
- `int addrLen = sizeof(clientAddr)`：地址长度，用于 `recvfrom` 和 `sendto` 函数。
- `bool connected = false`：连接状态，确保握手完成后才处理数据包。
- `int rcvWindowSize = 20`：接收窗口大小 (默认 20)，可通过命令行参数修改。
- `map<uint32_t, Packet> recvBuffer`：使用 `std::map` 存储乱序到达的包，键为序列号，实现 SR 选择性重传的缓存机制。

- `uint32_t expectedSeq = 1`：跟踪期望的下一个包序列号，用于滑动窗口。
- `ofstream outFile`：输出文件流，用于将接收的数据写入文件。

3.1.2 发送 ACK 函数

`send_ack` 函数实现选择确认机制：

发送 ACK 函数

```

1 void send_ack(uint32_t ackNum, sockaddr_in& targetAddr) {
2     Packet ackPkt;
3     memset(&ackPkt, 0, sizeof(ackPkt));
4     ackPkt.header.flags = FLAG_ACK;
5     ackPkt.header.ack = ackNum; // ACK 字段设置为收到的包的序列号，配合发送端的 SR
6     // 逻辑
7     ackPkt.header.length = 0;
8     ackPkt.header.checksum = calculate_checksum(&ackPkt);
9
10    sendto(sock, (char)&ackPkt, sizeof(PacketHeader), 0, (sockaddr)&targetAddr,
11           sizeof(targetAddr));
12    // cout << "[ACK] Sent ACK for " << ackNum << endl;
13 }

```

- 函数构造纯 ACK 包，`ack` 字段设置为收到的包序列号，实现“收到什么确认什么”的 SR 策略。
- ACK 包不携带数据，仅用于确认，发送后可在日志中看到 ACK 信息（当前注释掉以减少输出）。

3.1.3 主函数和参数处理

主函数处理命令行参数和初始化：

主函数和参数处理

```

1 int main(int argc, char* argv[]) {
2     // 参数检查，必须至少是3个：(程序名 + port + output_file)
3     if (argc < 3) {
4         cout << "Usage: " << argv[0] << " <port> <output_file> [window_size]" << endl;
5         return 1;
6     }
7
8     int port = atoi(argv[1]); // 将端口号字符串转换为整数
9     string outFileName = argv[2]; // 输出文件名
10
11    if (argc >= 4) {
12        rcvWindowSize = atoi(argv[3]);
13        cout << "Receive Window Size set to: " << rcvWindowSize << endl;
14    }
15
16    WSADATA wsaData;
17    WSAStartup(MAKEWORD(2, 2), &wsaData); // 初始化 Winsock，版本 2.2

```

```

18     sock = socket(AF_INET, SOCK_DGRAM, 0); // 创建 UDP 套接字, IPv4 和 UDP 数据报模式
19
20     sockaddr_in serverAddr;           // 服务器地址
21     serverAddr.sin_family = AF_INET;   // 地址族: IPv4
22     serverAddr.sin_port = htons(port); // 主机字节序转换为网络字节序(大端)
23     serverAddr.sin_addr.s_addr = INADDR_ANY; // 
24
25     cout << "INADDR_ANY 表示绑定到所有本地接口的 IP 地址, 这样服务器可以接受发送到任何本地 IP 地址的数据包."
26
27     if (bind(sock, (sockaddr*)&serverAddr, sizeof(serverAddr)) == SOCKET_ERROR) {
28         cout << "Bind failed." << endl;
29         return 1;
30     }
31
32     cout << "Server listening on port " << port << endl;
33     outFile.open(outFileName, ios::binary);           //
34     // 以二进制模式打开输出文件, 确保数据按原始字节写入
35     if (!outFile.is_open()) {
36         cout << "Failed to open output file: " << outFileName << endl;
37         return 1;
38     }
39
40     // ... 主循环 ...
41 }
```

- 参数处理支持可选的 `window_size`, 允许配置接收窗口大小。
- 初始化 Winsock 和 UDP 套接字, 绑定到指定端口, 打开输出文件。

3.1.4 握手和挥手逻辑

接收端处理连接管理:

握手和挥手逻辑

```

1 // 握手逻辑
2 // SYN 处理: 收到 SYN, 发送 SYN+ACK(TCP 三次握手的第二步)
3 if (recvPkt.header.flags & FLAG_SYN) {
4     cout << "[Handshake] SYN received." << endl;
5
6     Packet synAckPkt;
7     memset(&synAckPkt, 0, sizeof(synAckPkt));
8     synAckPkt.header.flags = FLAG_SYN | FLAG_ACK;
9     synAckPkt.header.seq = 0;
10    synAckPkt.header.ack = recvPkt.header.seq + 1;
11    synAckPkt.header.checksum = calculate_checksum(&synAckPkt);
12
13    sendto(sock, (char)&synAckPkt, sizeof(PacketHeader), 0, (sockaddr)&fromAddr,
14            fromLen);
15    cout << "[Handshake] SYN+ACK sent." << endl;
16    continue;
```

```

16 }
17
18 // 握手最后一步 ACK
19 // ACK 处理: 收到纯 ACK(非 SYN), 建立连接(第三步)
20 if ((recvPkt.header.flags & FLAG_ACK) && !connected && recvPkt.header.length == 0 &&
21   !(recvPkt.header.flags & FLAG_SYN)) {
22   cout << "[Handshake] Connection Established." << endl;
23   connected = true;
24   clientAddr = fromAddr;      // 保存客户端地址, 用于发送 ACK
25   continue;
26 }
27
28 // 挥手逻辑
29 // FIN 处理: 收到 FIN, 发送 ACK 确认, 并关闭连接
30 if (recvPkt.header.flags & FLAG_FIN) {
31   cout << "[Teardown] FIN received." << endl;
32   Packet ackPkt;
33   memset(&ackPkt, 0, sizeof(ackPkt));
34   ackPkt.header.flags = FLAG_ACK;           // 发送 ACK 确认
35   ackPkt.header.ack = recvPkt.header.seq + 1;
36   ackPkt.header.checksum = calculate_checksum(&ackPkt);
37   sendto(sock, (char)&ackPkt, sizeof(PacketHeader), 0, (sockaddr)&fromAddr,
38         fromLen);
39   cout << "[Teardown] ACK sent. Closing." << endl;
40   break;
41 }

```

- 实现三次握手和四次挥手，设置 `connected` 标志控制数据处理。
- 握手:** 收到 SYN 后发送 SYN+ACK，收到纯 ACK 后建立连接。
- 挥手:** 收到 FIN 后发送 ACK，并关闭连接。

3.1.5 数据处理和流量控制

核心数据处理逻辑实现 SR 和流量控制：

数据处理和流量控制

```

1 // 数据处理
2 if (connected && recvPkt.header.length > 0) {
3   uint32_t seq = recvPkt.header.seq;
4
5   // 流量控制: 如果序列号超出接收窗口, 直接丢弃, 不发送 ACK
6   if (seq >= expectedSeq + rcvWindowSize) {
7     // cout << "[Flow Control] Drop packet " << seq << " outside window." << endl;
8     continue;
9   }
10
11 // 发送 ACK (SR 模式: 收到什么确认什么)

```

```

12     send_ack(seq, fromAddr);

13

14     if (seq == expectedSeq) {
15         // 收到期望的包，写入文件
16         outFile.write(recvPkt.data, recvPkt.header.length);
17         expectedSeq++;
18
19         // 检查缓冲区是否有后续包
20         // .count() 用于检查 map 中是否存在指定键，返回 1 表示存在，0 表示不存在
21         while (recvBuffer.count(expectedSeq)) {
22             Packet& bufferedPkt = recvBuffer[expectedSeq];
23             outFile.write(bufferedPkt.data, bufferedPkt.header.length);
24             recvBuffer.erase(expectedSeq);
25             expectedSeq++;
26         }
27     } else if (seq > expectedSeq) {
28         // 乱序包，缓存
29         if (recvBuffer.find(seq) == recvBuffer.end()) {
30             recvBuffer[seq] = recvPkt;
31             // cout << "[Buffer] Buffered packet " << seq << endl;
32         }
33     }
34     // 如果 seq < expectedSeq，说明是重复包，已经 ACK 过了，上面已经重发了 ACK
35 }

```

- 流量控制通过检查 `seq >= expectedSeq + rcvWindowSize` 实现，超出窗口的包直接丢弃且不 ACK，迫使发送方停止发送。
- SR 逻辑：收到期望包时写入文件并滑动窗口，乱序包缓存，重复包重发 ACK。

3.1.6 Receiver 端的整体工作流程

接收端的工作流程如下：

1. **初始化**：解析参数，初始化套接字，绑定端口，打开输出文件。
2. **监听和接收**：进入主循环，阻塞等待数据包。
3. **校验和检查**：计算校验和，若不匹配丢弃包。
4. **连接管理**：处理 SYN、ACK、FIN 包，实现握手和挥手。
5. **数据处理**（连接建立后）：
 - 检查流量控制：超出窗口的包丢弃。
 - 发送 ACK：立即确认收到的包，并采用的是 SR 选择确认的方式。
 - 处理顺序包：写入文件，滑动 `expectedSeq`，检查缓冲区。
 - 处理乱序包：缓存到 `recvBuffer`。
 - 处理重复包：重发 ACK。

6. 结束：收到 FIN 后发送 ACK，关闭文件和套接字 Socket 资源。

通过这样的流程，确保 Receiver 接收端高效处理乱序和丢失，实现可靠的数据接收和流量控制。

3.2 Sender 发送端：sender.cpp

sender.cpp 文件实现发送端的逻辑，负责文件读取、打包、发送、拥塞控制和超时重传，体现实验要求的连接管理、确认重传和拥塞控制功能。

3.2.1 全局变量和状态管理

发送端使用全局变量管理连接、窗口和拥塞控制状态：

全局变量和状态管理

```

1 // 全局变量
2 SOCKET sock;
3 sockaddr_in serverAddr;
4 int addrLen = sizeof(serverAddr);
5 double packetLossRate = 0.0; // 丢包率 (0.0 - 1.0)
6 int maxWindowSize = 20;      // 最大发送窗口大小 (默认 20)
7 int delayMs = 0;            // 模拟延时 (毫秒)

8

9 // RENO 状态
10 enum RenoState {
11     SLOW_START,           // 慢启动
12     CONGESTION_AVOIDANCE, // 拥塞避免
13     FAST_RECOVERY         // 快速恢复
14 };
15

16 // 发送缓冲区中的包结构
17 struct SenderPacket {
18     Packet pkt;          // 数据包
19     bool acked;          // 是否被确认
20     bool sent;            // 是否已发送
21     chrono::steady_clock::time_point sendTime; // 发送时间, 用于超时检测
22 };
23

24 vector<SenderPacket> packets; // 发送缓冲区, 索引对应包的序列号减1(从 0 开始, 而 seq
25   从 1 开始)
26 double cwnd = 1.0;           // 拥塞窗口, 代表"发送方觉得网络能承受多少包"
27 int ssthresh = 16;           // 慢启动阈值
28 int dupAckCount = 0;          // 重复 ACK 计数器, 用于快速重传
29 RenoState state = SLOW_START;

30 // 发送窗口变量
31 int base = 0;                // 已确认的包的下一个索引(滑动窗口左边界)
32 int nextSeqNum = 0;           // 下一个要发送的包的索引(滑动窗口右边界)

```

- `SOCKET sock`：用于 UDP 通信的套接字。

- `sockaddr_in serverAddr`：服务器地址。
- `double packetLossRate = 0.0`：丢包率 (0.0-1.0)，用于模拟网络丢包。
- `int maxWindowSize = 20`：最大发送窗口大小 (默认 20)，用于流量控制。
- `int delayMs = 0`：模拟延时 (毫秒)，用于模拟网络延迟。
- `enum RenoState`：定义 RENO 拥塞控制的三种状态。
- `struct SenderPacket`：发送缓冲区中的包结构，包含数据包、确认状态、发送状态和发送时间。
- `vector<SenderPacket> packets`：发送缓冲区，索引对应包的序列号减 1。
- `double cwnd = 1.0`：拥塞窗口，初始值为 1。
- `int ssthresh = 16`：慢启动阈值，初始值为 16。
- `int dupAckCount = 0`：重复 ACK 计数器，用于快速重传。
- `RenoState state = SLOW_START`：RENO 状态，初始为慢启动。
- `int base = 0`：已确认的包的下一个索引 (滑动窗口左边界)。
- `int nextSeqNum = 0`：下一个要发送的包的索引 (滑动窗口右边界)。

3.2.2 发送数据包函数

`send_packet` 函数处理单个包的发送，包括模拟丢包和延时：

发送数据包函数

```

1 void send_packet(int seq_index) {
2     if (seq_index >= packets.size()) return;
3
4     SenderPacket& sp = packets[seq_index];
5
6     // 模拟丢包 (仅针对数据包, 不丢握手包)
7     // 如果概率 < packetLossRate 则模拟丢包
8     if (packetLossRate > 0.0 && ((rand() % 1000) / 1000.0 < packetLossRate)) {
9         // cout << "[Simulated Loss] Packet " << sp.pkt.header.seq << " dropped." <<
10        // endl;
11        // 即使 "丢包", 逻辑上也认为尝试发送了, 只是没调用 sendto
12        if (!sp.sent) {
13            sp.sent = true;
14            sp.sendTime = chrono::steady_clock::now();
15        }
16        return;
17    }
18
19    sp.pkt.header.checksum = 0;
20    sp.pkt.header.checksum = calculate_checksum(&sp.pkt);

```

```

20
21     // 模拟网络延时
22     if (delayMs > 0) {
23         this_thread::sleep_for(chrono::milliseconds(delayMs));
24     }
25
26     sendto(sock, (char)&sp.pkt, sizeof(PacketHeader) + sp.pkt.header.length, 0,
27             (sockaddr)&serverAddr, addrLen);
28
29     sp.sent = true;
30     sp.sendTime = chrono::steady_clock::now();
31     // print_packet_info("SEND", sp.pkt); // 调试输出
32 }
```

- 函数首先模拟丢包（概率 `packetLossRate`），若丢包则标记已发送但不实际发送。
- 计算校验和后，模拟延时 (`delayMs`)，然后调用 `sendto` 发送包，并记录发送时间用于超时检测。

3.2.3 握手函数

通过 `handshake` 函数实现三次握手流程：

握手函数

```

1  bool handshake() {
2      Packet synPkt;
3      memset(&synPkt, 0, sizeof(synPkt));
4      synPkt.header.flags = FLAG_SYN;
5      synPkt.header.seq = 0;           // 初始序列号设置为0
6      synPkt.header.length = 0;
7      synPkt.header.checksum = calculate_checksum(&synPkt);
8
9      // 发送 SYN
10     sendto(sock, (char)&synPkt, sizeof(PacketHeader), 0, (sockaddr)&serverAddr,
11             addrLen);
12     cout << "[Handshake] SYN sent." << endl;
13
14     // 等待 SYN + ACK
15     Packet recvPkt;                // 文件描述符集合, 用于 select
16     fd_set readfds;               // 函数, 监控套接字的可读状态
17     timeval tv;                  // 超时设置
18     tv.tv_sec = 2;                // 2秒超时
19     tv.tv_usec = 0;               // 微秒部分设置为0
20
21     FD_ZERO(&readfds);          // 清空集合
22     FD_SET(sock, &readfds);       // 将套接字加入集合, 以监控其可读状态
23
24     // select 等待, 它的作用是"带超时的等待". 如果 2 秒内没收到服务器的回复, select
25     // 就会返回 0, 握手失败, 避免程序死锁。
26 }
```

```

24     int ret = select(0, &readfds, NULL, NULL, &tv);
25
26     if (ret > 0) {
27         int len = recvfrom(sock, (char)&recvPkt, sizeof(recvPkt), 0,
28                             (sockaddr)&serverAddr, &addrLen);
29         if (len > 0 && (recvPkt.header.flags & (FLAG_SYN | FLAG_ACK))) {
30             if (calculate_checksum(&recvPkt) == recvPkt.header.checksum) {
31                 cout << "[Handshake] SYN+ACK received." << endl;
32
33                 // 发送 ACK
34                 Packet ackPkt;
35                 memset(&ackPkt, 0, sizeof(ackPkt));
36                 ackPkt.header.flags = FLAG_ACK;
37                 ackPkt.header.seq = 1;                                // 客户端初始序列号为1
38                 ackPkt.header.ack = recvPkt.header.seq + 1;          // 确认号为服务器初始序列号+1
39                 ackPkt.header.length = 0;
40                 ackPkt.header.checksum = calculate_checksum(&ackPkt);
41
42                 sendto(sock, (char)&ackPkt, sizeof(PacketHeader), 0,
43                         (sockaddr)&serverAddr, addrLen);
44                 cout << "[Handshake] ACK sent. Connection Established." << endl;
45                 return true;
46             }
47         }
48     cout << "[Handshake] Failed." << endl;
49     return false;
50 }
```

- 发送 SYN，等待 SYN+ACK，使用 `select` 设置 2 秒超时避免阻塞。
- 若成功，发送 ACK 并返回 true，实现连接建立。

3.2.4 文件加载函数

`load_file` 函数读取文件并打包成数据包：

文件加载函数

```

1 void load_file(const string& filename) {
2     ifstream file(filename, ios::binary);           // 以二进制模式打开文件
3     if (!file.is_open()) {
4         cerr << "Failed to open file: " << filename << endl;
5         exit(1);
6     }
7
8     file.seekg(0, ios::end);                      // file.seekg
9     用于设置输入流的读取位置，这里将读取位置移动到文件末尾
```

```

9     int fileSize = file.tellg();           // file.tellg
10    用于获取当前读取位置的偏移量,这里获取的是文件大小
11    file.seekg(0, ios::beg);               // 将读取位置移动回文件开头
12
13    int seq = 1; // 数据包序列号从1开始
14    while (file.tellg() < fileSize) {
15        SenderPacket sp;
16        memset(&sp, 0, sizeof(sp));
17        sp.pkt.header.seq = seq++;          // 序列号从1开始递增
18        sp.pkt.header.flags = 0; // 普通数据包
19        sp.acked = false;
20        sp.sent = false;
21
22        int remaining = fileSize - (int)file.tellg(); // 剩余未读字节数
23        int len = min(MSS, remaining);             // 本次读取的字节数不超过 MSS
24        和剩余字节数
25        file.read(sp.pkt.data, len);            // 读取数据到包的 data
26        部分,长度为 len
27        sp.pkt.header.length = len;            // 设置包的长度字段为 len
28
29        packets.push_back(sp);                //
30        将打包好的数据包加入发送缓冲区
31    }
32    file.close();
33    cout << "File loaded. Total packets: " << packets.size() << endl;
34 }
```

- 以二进制模式读取文件,按我们定义好的MSS(1024字节)分片,每个包分配递增序列号(seq序号从1开始),存储到packets这个vector向量中。

3.2.5 挥手函数

`teardown` 函数实现连接关闭:

挥手函数

```

1 void teardown() {
2     Packet finPkt;
3     memset(&finPkt, 0, sizeof(finPkt));
4     finPkt.header.flags = FLAG_FIN;
5     finPkt.header.seq = packets.size() + 1;      // FIN
6     包的序列号在发送的最后一个数据包之后
7     finPkt.header.length = 0;
8     finPkt.header.checksum = calculate_checksum(&finPkt);
9
10    sendto(sock, (char)&finPkt, sizeof(PacketHeader), 0, (sockaddr)&serverAddr,
11          addrLen);
12    cout << "[Teardown] FIN sent." << endl;
13 }
```

```

12 // 简单等待 ACK
13 Packet recvPkt;
14 timeval tv = {2, 0};
15 fd_set readfds;
16 FD_ZERO(&readfds);
17 FD_SET(sock, &readfds);
18
19 if (select(0, &readfds, NULL, NULL, &tv) > 0) {
20     recvfrom(sock, (char)&recvPkt, sizeof(recvPkt), 0, (sockaddr)&serverAddr,
21             &addrLen);
22     if (recvPkt.header.flags & FLAG_ACK) {
23         cout << "[Teardown] ACK received. Connection Closed." << endl;
24     }
25 }
}

```

- 在 Sender 发送端向 Receiver 接收端发送 FIN，等待 ACK，关闭连接。

3.2.6 主函数：参数处理和初始化

main 函数首先先处理命令行参数并进行初始化工作：

主函数：参数处理和初始化

```

1 int main(int argc, char* argv[]) {
2     srand(time(0)); // 初始化随机种子
3
4     if (argc < 4) {
5         cout << "Usage: " << argv[0] << " <server_ip> <server_port> <file_path>
6             [loss_rate] [window_size] [delay_ms]" << endl;
7         cout << "Example: " << argv[0] << " 127.0.0.1 8080 data.txt 0.1 20 100" <<
8             endl;
9         return 1;
10    }
11
12    string serverIp = argv[1]; // 服务器 IP 地址
13    int serverPort = atoi(argv[2]); // 服务器端口
14    string filePath = argv[3]; // 发送文件路径
15    if (argc >= 5) { // 可选参数：丢包率
16        packetLossRate = atof(argv[4]);
17        cout << "Packet Loss Rate set to: " << packetLossRate << endl;
18    }
19    if (argc >= 6) { // 可选参数：窗口大小
20        maxWindowSize = atoi(argv[5]);
21        cout << "Max Window Size set to: " << maxWindowSize << endl;
22    }
23    if (argc >= 7) { // 可选参数：延时
24        delayMs = atoi(argv[6]);
25        cout << "Delay set to: " << delayMs << " ms" << endl;
}

```

```

24 }
25
26 // 初始化 Winsock
27 WSADATA wsaData;
28 WSAStartup(MAKEWORD(2, 2), &wsaData);
29
30 sock = socket(AF_INET, SOCK_DGRAM, 0);
31
32 serverAddr.sin_family = AF_INET; // 地址族: IPv4
33 serverAddr.sin_port = htons(serverPort); // 主机字节序转换为网络字节序(大端)
34 // 使用 inet_addr 替代 inet_nton 以兼容 MinGW
35 serverAddr.sin_addr.s_addr = inet_addr(serverIp.c_str()); // 服务器 IP 地址
36
37 if (!handshake()) {
38     closesocket(sock);
39     WSACleanup();
40     return 1;
41 }
42
43 // 读取并打包文件
44 load_file(filePath);
45 }

```

- 解析我们运行时的命令行参数，并初始化 Winsock 和套接字，执行握手 `handshake()` 和文件加载 `load_file()` 工作。

3.2.7 主函数：数据传输循环

在处理完初始化与资源连接工作之后，主循环实现窗口机制下流水线发送和 RENO 拥塞控制，以及数据包发送后的超时工作：

主函数：数据传输循环

```

1 // 主循环: 发送和接收
2 while (base < packets.size()) {
3     // 1. 发送窗口内的包
4     int windowSize = min((int)cwnd, maxWindowSize); // 计算当前窗口大小，取拥塞窗口和最大窗口的较小值
5     // 发送窗口内未发送的包，直达到窗口上限
6     while (nextSeqNum < packets.size() && nextSeqNum < base + windowSize) {
7         if (!packets[nextSeqNum].sent) {
8             send_packet(nextSeqNum);
9         }
10        nextSeqNum++;
11    }
12
13    // 2. 接收 ACK
14    fd_set readfds;

```

```

15     FD_ZERO(&readfds);
16     FD_SET(sock, &readfds);
17     timeval tv = {0, 10000}; // 10ms 超时, 这里的 10ms 代表等待 ACK 的时间间隔
18
19     int ret = select(0, &readfds, NULL, NULL, &tv);
20     if (ret > 0) {
21         Packet recvPkt;
22         sockaddr_in fromAddr;
23         int fromLen = sizeof(fromAddr);
24         int len = recvfrom(sock, (char*)&recvPkt, sizeof(recvPkt), 0,
25             (sockaddr*)&fromAddr, &fromLen);
26
27         if (len > 0 && (recvPkt.header.flags & FLAG_ACK)) {
28             if (calculate_checksum(&recvPkt) == recvPkt.header.checksum) {
29                 int ackSeq = recvPkt.header.ack; // 接收端返回的是它收到的包的序列号 (SR 选择确认)
30                 int ackIndex = ackSeq - 1; // 索引号为序列号减 1
31
32                 // 如果该包在窗口范围内, 那么标记为已确认
33                 if (ackIndex >= base && ackIndex < packets.size()) {
34                     if (!packets[ackIndex].acked) {
35                         packets[ackIndex].acked = true;
36
37                         if (ackIndex == base) { // 收到 Base 的 ACK, 滑动窗口
38                             while (base < packets.size() && packets[base].acked) {
39                                 base++;
40                             }
41
42                         // RENO: 收到新数据的 ACK
43                         if (state == FAST_RECOVERY) {
44                             // 标准 RENO: 收到新数据的 ACK, 退出快速恢复 (Deflation)
45                             // 将 cwnd 恢复为 ssthresh
46                             cwnd = (double)ssthresh;
47                             state = CONGESTION_AVOIDANCE;
48                             dupAckCount = 0;
49                         } else {
49                             // 正常状态下的 ACK 处理
50                             dupAckCount = 0;
51
52                             if (state == SLOW_START) {
53                                 cwnd += 1.0; // 每收到一个 ACK, 窗口加 1
54                                 if (cwnd >= ssthresh) {
55                                     state = CONGESTION_AVOIDANCE;
56                                 }
57                             } else if (state == CONGESTION_AVOIDANCE) {
58                                 cwnd += 1.0 / cwnd; // 每收到一个 ACK, 窗口加 1/cwnd
59                             }
60                         }
61                     }
62                 }
63             }
64         }
65     }
66 }

```

```

59
60
61 } else {
62     // 收到乱序 ACK (SACK) -> 视为重复 ACK
63     if (state == FAST_RECOVERY) {
64         // 标准 RENO: 快速恢复期间, 每收到一个重复 ACK, cwnd
65         // 加 1 (允许发送新数据)
66         cwnd += 1.0;
67     } else {
68         dupAckCount++;
69         if (dupAckCount == 3) {
70             // 快速重传
71             cout << "[Fast Retransmit] Packet " <<
72                 packets[base].pkt.header.seq << endl;
73             send_packet(base); // 重传 Base
74
75             // 进入快速恢复
76             ssthresh = max(2, (int)cwnd / 2); // 阈值减半
77             cwnd = ssthresh + 3; // 快速恢复阶段, 拥塞窗口设置为阈值加3
78             state = FAST_RECOVERY; // 显式进入快速恢复状态
79         }
80     }
81 }
82 }
83 }
84 }

// 3. 处理超时
85 if (base < packets.size()) {
86     auto now = chrono::steady_clock::now();
87     auto duration = chrono::duration_cast<chrono::milliseconds>(now -
88         packets[base].sendTime).count();
89     if (packets[base].sent && duration > TIMEOUT_MS) {
90         cout << "[Timeout] Packet " << packets[base].pkt.header.seq << endl;
91         send_packet(base); // 重传 Base
92
93         // RENO 超时处理
94         ssthresh = max(2, (int)cwnd / 2);
95         cwnd = 1.0; // 重置为 1
96         state = SLOW_START; // 重新进入慢启动阶段
97         dupAckCount = 0;
98     }
99 }
100 }
101 }

```

- 循环发送窗口内包 Packet，接收 Receiver 端回复的确认 ACK 包并更新 RENO 状态机，处理超时重传。体现流水线传输和 RENO 拥塞控制。

3.2.8 主函数：统计输出信息

在所有的数据包传输完成后，计算并输出统计信息：

主函数：统计输出

```

1 auto endTime = chrono::steady_clock::now();
2 auto totalTimeUs = chrono::duration_cast<chrono::microseconds>(endTime -
3   startTime).count();                                // 除以一千万转换为秒 (microseconds 转为 seconds)
4
5 // 计算吞吐率 (Bytes / Second) -> MB/s
6 // 估算总传输数据量 = 包数量 * 数据长度 (忽略重传的开销，计算有效吞吐率 Goodput)
7 long long totalBytes = 0;
8 for(const auto& p : packets) totalBytes += p.pkt.header.length;
9
10 double throughput = (double)totalBytes / 1024.0 / 1024.0 / totalTimeSec;           // MB/s
11
12 cout << endl << "Transfer Complete!" << endl;
13 cout << "Time: " << totalTimeSec << " s" << endl;
14 cout << "Throughput: " << throughput << " MB/s" << endl;
15
16 teardown();
17
18 closesocket(sock);
19 WSACleanup();
20 return 0;

```

- 记录传输时间 `totalTimeSec` 和有效吞吐率 `Throughput`，执行挥手并清理资源。

3.2.9 Sender 端的整体工作流程

发送端的工作流程如下：

1. **初始化**：解析参数，初始化套接字，执行握手，建立连接。
2. **文件准备**：加载文件，分包存储到缓冲区。
3. **数据传输循环**：

- 计算当前窗口大小 `(min(cwnd, maxWindowSize))`。
- 发送窗口内未发包。
- 等待 ACK (10ms 超时)。
- 处理 ACK：标记确认，滑动窗口，更新 RENO 状态机（慢启动、拥塞避免、快速恢复）。

- 检查超时：重传并重置状态。
4. 结束：传输完成，输出统计，执行挥手，关闭连接。

通过这样的流程，确保了高效、可靠的数据发送，并且集成了 RENO 拥塞控制和窗口机制下的流量控制。

4 结果演示

我们通过下面的指令进行编译，从而得到 exe 可执行文件。

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

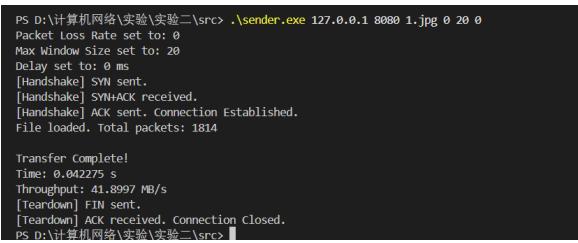
然后进行运行指令：

```
1 // 先运行 Receiver 接收端，接收文件命名为 received_final.jpg
2 .\receiver.exe 8080 received_final.jpg ([window_size])
3
4 // 再另开一个终端运行 Sender 发送端，发送文件 1.jpg
5 .\sender.exe 127.0.0.1 8080 1.jpg ([loss_rate] [window_size] [delay_ms])
```

4.1 0 丢包率与延时下的传输测试

我们首先在 0 丢包率与延时下进行我们的传输测试：

```
1 // 先运行 Receiver 接收端，接收文件命名为 received_final.jpg
2 .\receiver.exe 8080 received_final.jpg
3
4 // 再另开一个终端运行 Sender 发送端，发送文件 1.jpg
5 .\sender.exe 127.0.0.1 8080 1.jpg 0 20 0
```

(a) Receiver 端

(b) Sender 端

图 4.1: 终端运行结果

可以看到程序正常输出了三次握手的 [HandShake] 过程，以及在 Sender 发送端成功输出了 load_file 的文件打包结果，并且在传输过程中没有出现任何丢包或延时导致的 [Timeout] 超时或者 [Fast Retransmit] 快速重传信号，这说明我们在丢包率与延时情况下，程序成功并且正确且高速地完成了文件传输过程。并且在最后传输完毕后成功输出了总时间 Time 与吞吐率 Throughput，最终发送 FIN 信号挥手断开连接。

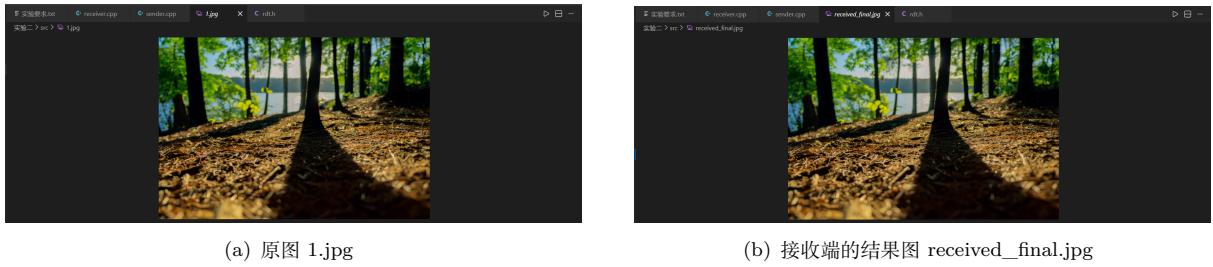


图 4.2: 发送图与接收图一致

4.2 3% 丢包率下的传输测试

我们下面在 3% 丢包率下进行我们的传输测试:

```

1 // 先运行 Receiver 接收端, 接收文件命名为 received_final.jpg
2 .\receiver.exe 8080 received_final.jpg
3
4 // 再另开一个终端运行 Sender 发送端, 发送文件 1.jpg, 丢包率设置为 3%
5 .\sender.exe 127.0.0.1 8080 1.jpg 0.03 20 0

```

```

PS D:\计算机网络\实验\实验二\src> .\receiver.exe 8080 received_final.jpg
Server listening on port 8080
[Handshake] SYN received.
[Handshake] SYNACK sent.
[Handshake] Connection Established.
[Teardown] FIN received.
[Teardown] ACK sent. Closing.
PS D:\计算机网络\实验\实验二\src>

```

图 4.3: Receiver 接收端

```

PS D:\计算机网络\实验\实验二\src> .\sender.exe 127.0.0.1 8080 1.jpg 0.03 20 0
Packet Loss Rate set to: 0.03
Max Window Size set to: 20
Delay set to: 0 ms
[Handshake] SYN sent.
[Handshake] SYNACK received.
[Handshake] ACK sent. Connection Established.
File loaded. Total packets: 1814
[Fast Retransmit] Packet 7
[Timeout] Packet 15
[Timeout] Packet 22
[Fast Retransmit] Packet 79
[Fast Retransmit] Packet 115
[Fast Retransmit] Packet 133
[Fast Retransmit] Packet 148
[Fast Retransmit] Packet 172
[Fast Retransmit] Packet 193
[Fast Retransmit] Packet 210
[Fast Retransmit] Packet 265
[Fast Retransmit] Packet 338
[Fast Retransmit] Packet 353
[Timeout] Packet 362
[Fast Retransmit] Packet 1567
[Fast Retransmit] Packet 1576
[Fast Retransmit] Packet 1594
[Timeout] Packet 1597
[Fast Retransmit] Packet 1634
[Timeout] Packet 1636
[Fast Retransmit] Packet 1654
[Timeout] Packet 1661
[Fast Retransmit] Packet 1714
[Fast Retransmit] Packet 1789
[Timeout] Packet 1789
[Timeout] Packet 1794
[Timeout] Packet 1803
Transfer Complete!
Time: 12.264 s
Throughput: 0.144431 MB/s
[Teardown] FIN sent.
[Teardown] ACK received. connection Closed.
PS D:\计算机网络\实验\实验二\src>

```

图 4.4: Sender 发送端

可以看到程序在除了正常输出了三次握手的 [HandShake] 过程, 以及在 Sender 发送端成功输出了 load_file 的文件打包结果之外, 由于我们设置了 3% 的丢包率, 因此在传输过程中出现了由于丢包而导致的 [Timeout] 超时以及由于拥塞而导致的 [Fast Retransmit] 快速重传信号, 这也说明我们正确处理了超时重传, 以及 RENO 算法下的拥塞控制的机制, 同时也验证了我们选择确认实现的正确性。最后程序成功并且正确地完成了文件传输过程。并且在最后传输完毕后成功输出了总时间 Time 与吞吐率 Throughput, 最终发送 FIN 信号挥手断开连接。

可以看到由于丢包的影响，程序的运行时间与 0 丢包率相比，由 0.04s 变为了 12s，同时程序的吞吐率 Throughput 大大降低，这也是我们由于拥塞控制而导致的流量控制。

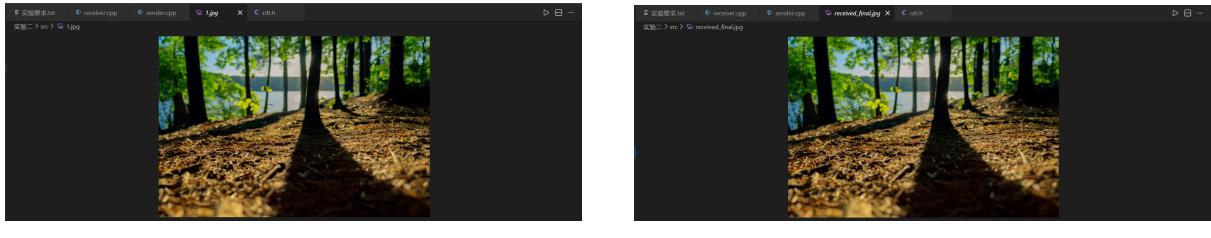


图 4.5: 发送图与接收图一致

4.3 3% 丢包率与 5ms 延时下的传输测试

我们下面在 3% 丢包率与 5ms 延时下进行我们的传输测试：

```

1 // 先运行 Receiver 接收端，接收文件命名为 received_final.jpg
2 .\receiver.exe 8080 received_final.jpg
3
4 // 再另开一个终端运行 Sender 发送端，发送文件 1.jpg，丢包率设置为 3%，延时设为 5ms
5 .\sender.exe 127.0.0.1 8080 1.jpg 0.03 20 5

```



图 4.6: Receiver 接收端

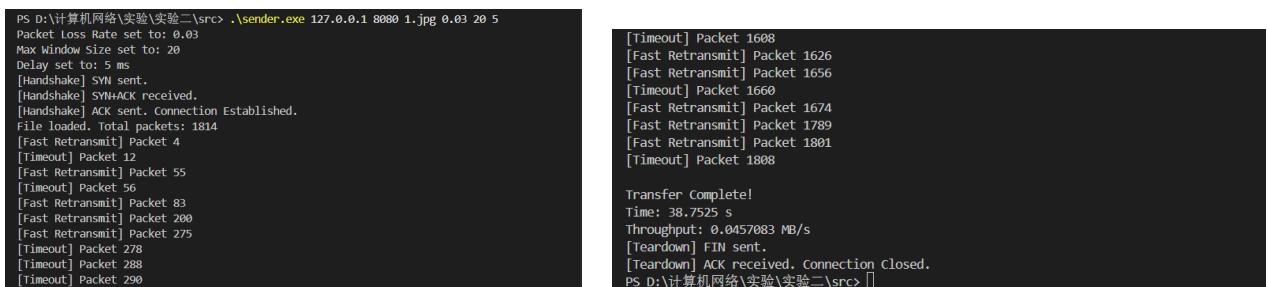


图 4.7: Sender 发送端

可以看到在丢包的影响之外，再加入 5ms 延时的影响之后，程序的运行时间与 0 丢包率和 3% 丢包率相比，由 0.04s 和 12s 变为了 38s，同时程序的吞吐率 Throughput 也比前两种情况都大大降低，这也是我们由于拥塞控制和延时机制下的流量控制而导致的流量减少。

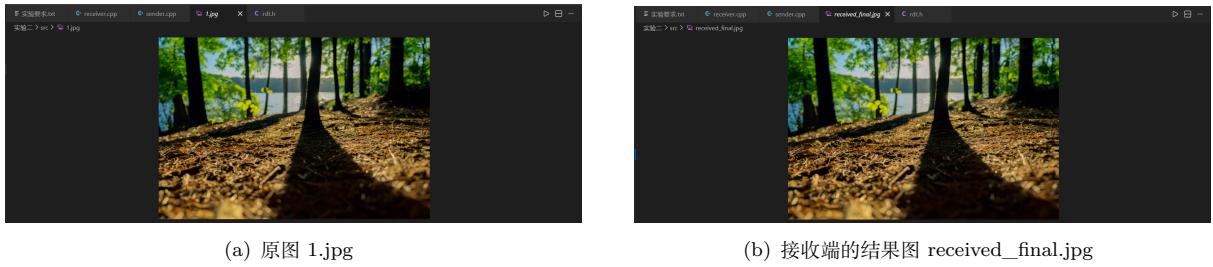


图 4.8: 发送图与接收图一致

4.4 窗口大小设为 5 的传输测试

我们在默认情况下，Sender 发送端的最大窗口大小 maxWindowSize 与 Receiver 接收端的窗口大小 window_size 都设置为 20，现在我们在命令行将窗口大小设置为 5，来观察测试结果：

```

1 // 先运行 Receiver 接收端，接收文件命名为 received_final.jpg，窗口大小设为 5
2 .\receiver.exe 8080 received_final.jpg
3
4 // 再另开一个终端运行 Sender 发送端，发送文件 1.jpg
5 .\sender.exe 127.0.0.1 8080 1.jpg 0 5 0

```

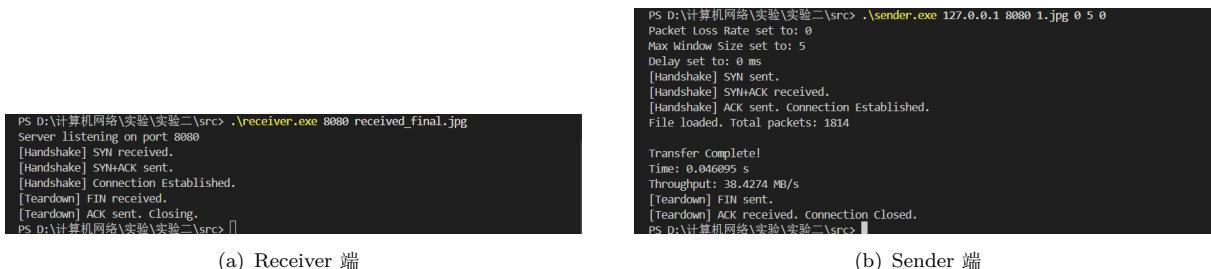


图 4.9: 终端运行结果

可以看到将窗口大小设置为 5 之后，程序的运行时间与原来的 20 窗口大小相比，由 0.042275s 变为了 0.046095s，同时程序的吞吐率 Throughput 由 41.8MB/s 变为了 38.4MB/s，这也是我们由于窗口机制下的流量控制。

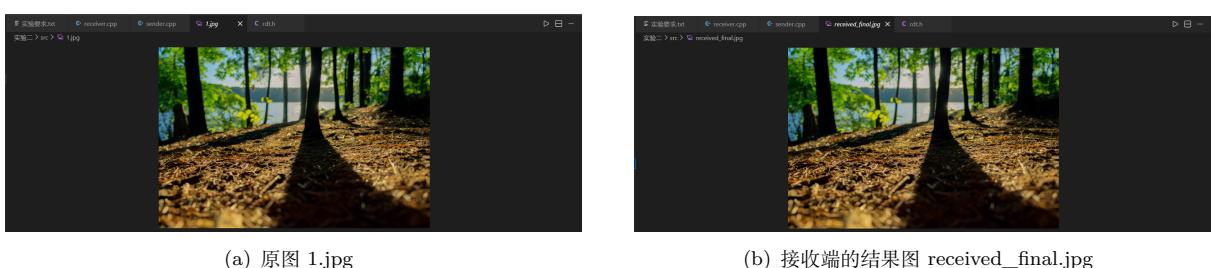


图 4.10: 发送图与接收图一致

5 总结

总的来说，通过本次实验，我们成功设计并实现了一个基于 UDP 数据报套接字的用户空间可靠数据传输协议 (RDT)。我们首先详细设计了协议头部结构、状态机和交互流程，然后利用 C++ 语言和 Winsock API，完成了**发送端 (Sender)** 和**接收端 (Receiver)** 程序。协议实现了面向连接的传输，包括连接管理（三次握手建立、四次挥手关闭）、差错检测（16 位校验和）、确认重传（选择性确认重传 SR 机制）、流量控制（固定窗口大小）和拥塞控制（**TCP RENO 算法**）。此外，我们还在发送端手动添加实现了**丢包率**和**网络延时**的模拟功能，便于测试不同网络条件下的性能表现。

在实现过程中，我们根据实验要求：单向数据传输（文件传输），双向控制信息交互；通过命令行参数配置窗口大小和丢包率，观察其对传输时间和吞吐率的影响。并通过日志输出（如 **[Fast Retransmit]** 和 **[Timeout]**）验证了 RENO 算法的正确执行。最后我们的实验演示也验证了协议的可靠性，能够在模拟丢包环境下正确重传和恢复传输，确保文件完整性。

通过动手实现这个可靠传输协议，加深了我对 UDP 编程、TCP 协议栈机制（如滑动窗口、状态机）、网络差错处理和 RENO 拥塞控制算法的理解。同时，也提升了使用 Winsock API 进行底层网络编程的能力，以及在用户空间模拟复杂网络行为的技巧，收获很大。

6 github 仓库源码

代码详见：<https://github.com/Wangxuuuuu/Computer-Network>