南开大学
Nankai University

南　　开　　大　　学

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

黄尚扬

年级：2023 级

专业：计算机科学与技术

指导教师：徐敬东 张建忠

2025 年 12 月 25 日

摘要

关键字：计算机网络 TCP 协议 socket 编程 UDP RDT 差错检测 选择确认 滑动窗口 RENO

本报告是计算机网络课程的第二次实验。

在这一部分，笔者将根据所学内容进行可靠传输协议的实验及分析。并且，笔者将完成具体实验要求以获取更加深刻的实验理解。

目录

一、 实验目的

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。

二、 项目结构

实验的 git 链接为：Hsy23333/NKU_ComputerNetworks

本次实验存放在 l2 文件夹中，分别包含 server.cpp、clientt.cpp 以及其分别的可执行文件。正常运行中，需要先启动 server.exe，而后启动给定的实验环境路由器 router.exe，设置好相关信息后再打开 client.exe 即可。

三、 原理说明/协议设计

(一) 原理介绍

在这一部分，笔者将会进行一些必要概念的解释，在保证报告完整性的同时展现笔者的理解。

1. UDP

UDP（User Datagram Protocol，用户数据报协议）是一种位于传输层的**无连接通信协议**，用于在网络中以较为简洁的方式传输数据。与 TCP（Transmission Control Protocol，传输控制协议）相比，UDP 不提供可靠性保障机制，不具备流量控制、拥塞控制以及数据重传等功能，因此通常被称为一种“不可靠”的传输协议。正是由于其协议机制较为简单，UDP 具有**传输开销小、时延低**的特点，适合应用于对实时性要求较高、但能够容忍一定数据丢失的场景，例如视频流传输、语音通信以及 DNS 查询等。

UDP 的工作机制主要体现在以下几个方面：

- **无连接特性**：UDP 在进行数据传输前无需建立连接，发送端与接收端之间不存在连接建立与释放的过程。每个 UDP 数据报都是相互独立的，通信双方不维护传输状态。
- **简洁的数据报结构**：UDP 数据包结构相对简单，头部仅包含源端口号、目的端口号、长度以及校验和等必要字段。与 TCP 相比，其协议头部开销更小，从而提高了数据传输的效率。
- **缺乏确认与重传机制**：UDP 在发送数据时不要求接收方返回确认信息，也不提供重传机制。当数据包在传输过程中丢失或出错时，发送端不会再次发送该数据，接收端收到数据后也无需反馈确认。

由于 UDP 不保证数据传输的可靠性，因此更适用于对延迟敏感、但允许一定数据丢失的实时应用，如在线游戏、视频会议和语音通信等。

在本实验中，通过对 UDP 通信过程的模拟与分析，重点观察其无连接特性及数据传输行为，并研究丢包和延迟等因素对通信效果的影响。实验结果有助于加深对 UDP 协议基本特性的理解，并为实际网络应用中协议的选择提供理论参考。

2. 校验和机制

校验和是一种用于**检测数据在传输过程中是否发生错误**的基本手段。在 UDP 等传输协议中，校验和通过对数据报内容进行计算生成，用于验证数据的完整性。发送端在发送数据前计算校验和并填入报文中，接收端在收到数据后重新计算校验和，通过比较计算结果判断数据在传输过程中是否被篡改或损坏。

UDP 校验和的计算范围通常包括**伪首部、UDP 首部以及数据载荷**，其中伪首部并不参与实际传输，而是用于增强校验对源地址、目的地址以及协议类型等关键信息的保护能力，从而提高差错检测的可靠性。

发送端的处理流程如下：

- 构造伪首部，并将 UDP 首部中的校验和字段清零；
- 若数据报长度不是 16 位的整数倍，则在末尾使用 0 进行填充；
- 将伪首部、UDP 首部和数据载荷视为一个由 16 位整数组成的序列；
- 对该序列执行 **16 位二进制反码求和运算**（即遇到进位时将进位加回到最低位）；
- 将最终求和结果按位取反，并将结果写入 UDP 首部的校验和字段中。

接收端的处理流程如下：

- 根据接收到的数据报重新构造伪首部；
- 若数据长度不是 16 位的整数倍，同样使用 0 进行填充；
- 将伪首部和完整数据报按 16 位整数序列进行二进制反码求和运算；
- 若计算结果为全 1（即 0xFFFF），则认为数据在传输过程中未检测到差错；否则，说明数据报可能在传输过程中发生了错误。

需要注意的是，校验和机制只能**检测错误而不能纠正错误**，同时也无法保证一定能够发现所有差错。但由于其计算过程简单、开销较小，校验和在实际网络通信中被广泛采用，尤其适用于对性能和实时性要求较高的场景。在 UDP 协议中，校验和是唯一的差错检测手段，对于保障数据传输的基本正确性具有重要意义。

3. 超时重传机制

超时重传是一种常见的**差错控制与可靠性保障机制**，主要用于应对数据包在传输过程中发生丢失的情况。其基本思想是：发送方在发送数据后启动定时器，若在预设的时间内未收到来自接收方的确认（ACK），则认为该数据包在传输过程中可能丢失或发生错误，从而触发对该数据包的重新发送。

基本原理 在数据发送完成后，发送端为该数据包设置一个超时计时器。若在计时器到期前成功收到对应的确认消息，则认为该数据包已被正确接收，计时器随即取消；反之，若计时器到期仍未收到确认，则发送端判定该数据包未能成功到达接收端，并执行重传操作，以保证数据最终能够被正确接收。

超时时间的计算 超时时间的设定通常基于网络的 ** 往返时间（RTT，Round-Trip Time）** 估计。发送端可以通过测量数据包从发送到接收确认返回所经历的时间来估算 RTT，并在此基础上设置合理的超时阈值。为了适应网络延迟的波动，超时时间往往采用动态调整策略，避免因超时过短导致不必要的重传，或因超时过长而降低传输效率。

应用层中的实现 在基于 UDP 的通信中，由于 UDP 协议本身不提供可靠传输机制，也不支持确认与重传功能，因此需要由**应用层自行实现超时重传机制**。通过在应用层引入确认、计时与重传逻辑，可以在一定程度上弥补 UDP 在可靠性方面的不足，使其适用于对数据完整性有一定要求的应用场景。

4. 滑动窗口机制

滑动窗口机制是一种用于**提高数据传输效率并实现流量控制**的重要技术，广泛应用于可靠数据传输协议中。该机制允许发送方在未收到确认的情况下连续发送多个数据包，从而避免每发送一个数据包就必须等待确认所带来的效率下降。

基本原理 在滑动窗口机制中，发送方维护一个发送窗口，用于表示当前允许发送但尚未被确认的数据包序号范围。窗口内的数据包可以连续发送，而不需要逐个等待接收方的确认。接收方在成功接收数据包后，会向发送方返回确认信息，告知其已经正确接收的数据序号。

当发送方收到确认（ACK）后，窗口的起始位置向前移动，即“滑动”，从而释放出新的窗口空间，允许发送方继续发送后续的数据包。通过这种方式，发送与确认过程可以并行进行，显著提高链路利用率。

发送窗口与接收窗口 滑动窗口机制通常涉及**发送窗口**和**接收窗口**两个概念：

- **发送窗口**：由发送方维护，表示当前允许发送的数据包序号区间。窗口大小决定了发送方在未收到确认前最多可以发送的数据包数量。
- **接收窗口**：由接收方维护，用于限制能够接收的数据包序号范围，防止数据包乱序或溢出接收缓冲区。

通过接收窗口的限制，接收方可以对发送方进行一定程度的流量控制，从而避免接收端处理能力不足导致的数据丢失。

5. 选择确认机制

选择确认（Selective Acknowledgment，简称 SACK）是一种用于**提高可靠数据传输效率**的确认机制，常与滑动窗口和超时重传机制配合使用。与传统的累计确认方式不同，选择确认允许接收方对**已经正确接收的数据包进行逐个或分段确认**，从而帮助发送方更精确地识别丢失的数据包，减少不必要的重传。

基本原理 在选择确认机制中，接收方在返回确认信息时，不仅会指明已经连续正确接收的最大序号，还会明确告知发送方**哪些序号的数据包已经成功接收、哪些尚未收到**。这样，即使数据包发生乱序到达，接收方仍可以对已接收的数据包进行确认。

发送方在收到选择确认信息后，只需对**未被确认的数据包进行重传**，而无需重新发送窗口中已经正确到达的数据包，从而显著提高了传输效率。

与累计确认的对比 与累计确认机制相比，选择确认具有以下特点：

- **重传更精确**：仅重传实际丢失的数据包，避免重复发送已成功接收的数据；
- **更适合高丢包或高延迟环境**：在网络状况较差时，能够有效减少冗余传输；
- **实现复杂度更高**：需要接收方维护更详细的接收状态，发送方也需要解析并处理更复杂的确认信息。

相比之下，累计确认机制实现简单，但在发生丢包时往往会导致大量不必要的重传，效率较低。

与滑动窗口和重传机制的配合 选择确认通常与滑动窗口机制共同使用。发送方根据选择确认信息动态调整发送窗口，并结合超时重传机制，对未确认的数据包进行有针对性的重传。这种机制能够在保证可靠性的同时，提高链路利用率和整体传输性能。

Reno 拥塞控制机制

Reno 是一种经典的拥塞控制算法，在 Tahoe 的基础上引入了快速重传（Fast Retransmit）和快速恢复（Fast Recovery）机制，能够在检测到网络拥塞时有效调整发送速率，在保证网络稳定性的同时提高传输效率。

基本思想 Reno 的核心目标是在充分利用网络带宽的同时避免网络拥塞。它通过动态调整拥塞窗口（Congestion Window，cwnd）大小来控制发送方在网络中的数据注入速率，并以此实现端到端的拥塞控制。

Reno 将网络状态划分为不同阶段，根据丢包信号对发送速率进行相应调整，从而在吞吐量和拥塞之间取得平衡。

慢启动（Slow Start） 在连接建立初期或发生严重拥塞后，Reno 采用慢启动机制。此时拥塞窗口从一个较小的初始值开始，并在每经过一个往返时间（RTT）后呈指数增长。慢启动阶段的目的是快速探测网络的可用带宽。

当拥塞窗口达到慢启动阈值（ssthresh）时，Reno 结束慢启动阶段，进入拥塞避免阶段。

拥塞避免（Congestion Avoidance） 在拥塞避免阶段，Reno 以较为保守的方式增加发送速率。拥塞窗口的增长由指数增长变为线性增长，通常表现为每经过一个 RTT，cwnd 仅增加一个最大报文段（MSS），以避免对网络造成过大冲击。

快速重传（Fast Retransmit） 当发送方连续收到三个相同的重复确认（Duplicate ACK）时，TCP Reno 认为某个数据包可能已经丢失，即使尚未发生超时，也会立即重传该数据包。这一机制可以显著减少等待超时所带来的延迟。

快速恢复（Fast Recovery） 在快速重传触发后，Reno 并不会将拥塞窗口重置为初始值，而是进入快速恢复阶段。发送方将慢启动阈值设置为当前拥塞窗口的一半，并将拥塞窗口调整为新的阈值附近，然后继续进行线性增长。这样可以避免在轻微拥塞情况下退回到慢启动阶段，提高整体传输效率。

特点与局限性 TCP Reno 在单个数据包丢失的情况下表现良好，但在一个窗口内发生**多个数据包丢失**时，其性能会明显下降。此外，Reno 对选择确认（SACK）的支持有限，在高丢包或高带宽延迟积网络中容易出现恢复效率不足的问题。

(二) 报文设计/模块设计

这一部分笔者将会展示数据报文、校验方法等耦合模块的设计。

1. 自定义数据报文格式

实验中定义了一种应用层可靠传输报文 `RDT_Packet`，其结构同时包含控制信息与数据载荷，用于支持连接管理、可靠传输和差错检测。报文主要由以下字段组成：

- **序号（seq_num）**

  用于唯一标识当前数据包在发送序列中的位置，是实现滑动窗口、乱序接收以及重传机制的基础。

- **确认号（ack_num）**

  表示接收方期望接收的下一个数据包序号，用于实现确认机制，支持累计确认和选择确认的扩展。

- **控制位（flags）**

  采用位标志方式表示不同控制含义，包括 SYN、ACK 和 FIN，用于连接建立、数据确认和连接释放等过程。

- **数据长度（length）**

  指示当前数据包中有效数据的字节数，便于接收方正确解析数据内容。

- **校验和（checksum）**

  用于检测数据在传输过程中是否发生差错，校验范围覆盖伪首部与完整数据报内容。

- **数据载荷（truedata）**

  用于存放实际传输的数据，最大长度为 `MAX_DATA_SIZE`，以支持文件或数据流的分段发送。

通过上述字段设计，该报文格式能够在 UDP 之上同时承担控制报文和数据报文的功能。

2. 伪首部设计与校验和支持

为增强校验和对网络层关键信息的保护能力，实验中引入了伪首部（Pseudo Header）的概念，其结构包含：

- 源 IP 地址
- 目的 IP 地址
- 保留字段（固定为 0）
- 协议号
- 数据报总长度（首部 + 数据）

伪首部并不参与实际网络传输，仅在发送端和接收端计算校验和时使用。校验和采用 **16 位二进制反码求和**的方式计算，若接收端计算结果为全 1，则认为数据未检测到差错。

3. 发送端滑动窗口结构设计

为支持流水线发送与超时重传机制，发送端设计了基于数组的滑动窗口结构。

发送窗口槽位（SendSlot） 每个窗口槽位用于记录一个已发送或待确认的数据包，主要包含：

- 对应的数据包内容；
- 最近一次发送时间，用于判断是否发生超时；
- 确认状态标志，用于标识该数据包是否已被成功确认。

发送窗口（SendWindow） 发送窗口由固定大小的槽位数组组成，并维护以下关键状态：

- **base**：当前窗口起始序号，对应最早未被确认的数据包；
- **next_seq**：下一个允许发送的数据包序号；
- **count**：当前窗口内有效数据包数量。

发送端仅允许 `next_seq` 落在 `[base, base + WINDOW_SIZE)` 范围内的数据包被发送，从而实现对发送速率的控制。

4. 接收端滑动窗口结构设计

接收端同样维护一个滑动窗口，用于支持乱序接收与选择确认。

接收窗口槽位（RecvSlot） 每个槽位保存一个已接收的数据包及其接收状态，用于判断数据是否重复到达或乱序到达。

接收窗口（RecvWindow） 接收窗口结构主要维护：

- **base_seq**：当前期望接收的最小序号；
- **窗口槽位数组**：用于缓存窗口范围内的已接收数据包；
- **count**：窗口内已接收但尚未交付上层的数据包数量。

当从 `base_seq` 开始的数据连续到达后，窗口向前滑动，并将数据按序交付上层。

5. 设计总结

通过上述报文与数据结构设计，实验在 UDP 协议之上实现了包括：

- 序号与确认机制
- 校验和差错检测
- 滑动窗口流水线传输
- 超时重传与选择确认支持

在保证结构清晰和实现可控的前提下，较好地模拟了 TCP 可靠传输的核心机制，为后续实验功能的实现提供了基础支撑。

(三) 协议设计

这一部分，笔者将会说明数据传输过程中各重要方法、协议的设计思路。

1. 连接管理

为在 UDP 协议之上实现面向连接的可靠数据传输，本实验在应用层自行设计并实现了一套**连接管理机制**，整体流程参考 TCP 的连接建立与释放过程，但在实现上进行了适度简化，以适配实验需求。

连接管理主要包括 **连接建立（三次握手）**和 **连接释放（四次挥手）**两个阶段，并结合校验和与超时重传机制，保证在存在丢包或差错的情况下仍能正确完成连接状态转换。

连接建立机制（三次握手） 连接建立阶段采用 **三次握手（Three-Way Handshake）**的方式完成客户端与服务器之间的连接同步。

1. 初始序号生成

- 客户端和服务器在程序启动时分别随机生成初始序号（`client_seq` 与 `server_seq`），用于后续数据传输中的序号空间管理。
- 该设计有助于避免旧连接残留报文对新连接造成干扰。

2. 第一次握手：客户端发送 SYN

客户端向服务器发送一个仅包含控制信息的报文：

- 设置 `SYN` 标志位；
- `seq_num` 为客户端当前初始序号；
- 不携带数据载荷；
- 计算并填写校验和。

若在规定时间内未收到服务器回应，客户端会**超时重传 SYN 报文**，直到成功收到合法回应为止。

3. 第二次握手：服务器回复 SYN + ACK

服务器在正确接收到并校验客户端 SYN 报文后，返回一个同时包含 `SYN` 和 `ACK` 标志的报文：

- `seq_num` 为服务器自身初始序号；
- `ack_num` 为客户端序号加一，表示确认客户端的连接请求；
- 不携带数据；
- 使用伪首部参与校验和计算。

若客户端的 ACK 未能及时返回，服务器会对该 `SYN+ACK` 报文进行重传。

4. 第三次握手：客户端发送 ACK

客户端收到并验证服务器的 `SYN+ACK` 报文后，发送最终确认报文：

- 设置 `ACK` 标志位；

- `seq_num` 为客户端序号加一；
- `ack_num` 为服务器序号加一。

此后，客户端会短暂监听是否仍有 `SYN+ACK` 重复到达，若检测到，说明前一次 ACK 丢失，将重新发送 ACK 报文，以确保服务器成功进入已连接状态。

当该过程结束后，客户端与服务器均认为连接建立成功，进入数据传输阶段。

连接释放机制（四次挥手） 连接释放阶段采用类似 TCP 的 **四次挥手（Four-Way Handshake）**，以保证双方能够可靠地释放连接并完成资源回收。

1. 第一次挥手：客户端发送 FIN

当客户端完成所有数据发送后，主动发起连接关闭：

- 发送设置 `FIN` 标志位的报文；
- `seq_num` 为客户端当前序号；
- `ack_num` 为服务器当前序号；
- 不携带数据。

若未收到服务器确认，客户端会周期性重发 FIN 报文。

2. 第二次挥手：服务器回复 ACK

服务器收到并校验客户端的 FIN 报文后：

- 回复一个仅包含 `ACK` 标志位的报文；
- 确认号为客户端 FIN 的序号加一；
- 表示已确认客户端的关闭请求。

3. 第三次挥手：服务器发送 FIN

在确认客户端关闭请求后，服务器向客户端发送自己的 FIN 报文，表示服务器端也准备关闭连接。

若客户端未能及时响应，服务器会对该 FIN 报文进行重传。

4. 第四次挥手：客户端发送 ACK

客户端收到服务器 FIN 后：

- 发送最终的 ACK 报文；
- 确认号为服务器 FIN 的序号加一。

随后客户端进入短暂等待状态，若未再收到服务器报文，则认为连接已成功关闭，并释放相关资源。

可靠性保障措施 在连接管理的各个阶段，系统引入了以下可靠性保障机制：

1. **校验和校验**

   所有控制报文均参与伪首部校验和计算，若校验失败则直接丢弃。

2. **超时重传机制**

   对 SYN、ACK、FIN 等关键控制报文均设置超时检测，若在规定时间内未收到有效响应，则进行重传。

3. **非阻塞 I/O 轮询**

   通过非阻塞 socket 与周期性轮询方式，实现对丢包与延迟的容错处理。

连接管理小结 通过上述连接管理机制，本实验在 UDP 之上成功实现了：

- 面向连接的通信抽象；
- 类 TCP 的三次握手与四次挥手流程；
- 具备差错检测与超时重传能力的连接控制。

该设计为后续滑动窗口数据传输、拥塞控制（Reno）等机制提供了稳定可靠的连接基础。

2. 差错检测

在基于 UDP 的可靠数据传输实验中，由于底层协议本身不提供端到端的数据完整性保障机制，本实验在应用层自行实现了一套**差错检测模块**，用于检测数据在传输过程中是否发生比特级错误，从而避免错误数据被错误接收或确认。

该模块主要基于 **16 位二进制反码校验和（Internet Checksum）**，并引入伪首部参与计算，以增强校验对关键信息的保护能力。

设计目标 差错检测模块的设计目标包括：

1. 检测数据在传输过程中是否发生任意比特错误；
2. 防止错误数据包被误认为合法并进入接收窗口；
3. 为超时重传与选择确认机制提供可靠的错误判定依据；
4. 在保证检测效果的同时尽量降低计算复杂度。

校验范围与报文组成 在本实验中，校验和的计算范围并非仅限于数据载荷，而是覆盖以下内容：

- **伪首部（Pseudo Header）**

  包含源 IP、目的 IP、协议号及数据报长度等信息；

- **完整应用层数据报**

  包括自定义报文首部字段（序号、确认号、控制位、长度）以及实际数据载荷。

通过将网络层的关键信息纳入校验范围，可以有效避免数据被错误投递到错误端点而未被检测的问题。

发送端校验和计算流程 发送端在构造每一个数据包或控制包时，均按照如下流程计算并填写校验和字段：

1. **构造伪首部**

   填写源 IP、目的 IP、协议号和数据报总长度等字段；

2. **校验和字段清零**

   在计算校验和前，将数据报中的校验和字段置为 0，避免旧值干扰计算结果；

3. **数据对齐处理**

   若伪首部或数据报长度为奇数，则在末尾补 0，使其长度为 16 位整数倍；

4. **16 位反码求和**

   将伪首部和数据报视为一组连续的 16 位整数序列，逐个进行二进制反码加法；

5. **折叠进位**

   若求和结果超过 16 位，则将高位进位折叠回低 16 位，直至结果不再产生进位；

6. **结果取反**

   对最终结果取反，得到校验和值，并写入数据报的校验和字段。

接收端校验和验证流程 接收端在收到任意数据包或控制包后，首先执行校验和验证，具体流程如下：

1. **重新构造伪首部**

   使用接收端已知的源 IP、目的 IP、协议号和报文长度信息；

2. **按原始内容参与计算**

   不对校验和字段做任何修改，将整个报文与伪首部一起参与校验和计算；

3. **16 位反码求和运算**

   与发送端采用相同的计算方式进行求和；

4. **结果判定**

   - 若计算结果为全 1（0xFFFF），则认为未检测到差错；
   - 否则，判定该数据报在传输过程中发生差错。

差错处理策略 当接收端检测到校验和错误时，系统采取以下处理策略：

- **直接丢弃错误数据包**

  不将其放入接收窗口，也不向上层交付；

- **不发送确认报文**

  避免发送错误确认导致发送端误判；

- **依赖发送端超时重传**

  通过超时机制促使发送端重新发送正确的数据包。

该策略能够有效避免错误数据被传播，同时保持协议实现的简洁性。

与可靠传输机制的协同作用 差错检测模块与本实验中的其他可靠性机制紧密配合：

- 为滑动窗口提供合法数据包的判定依据；
- 为选择确认机制区分“未到达”和“错误到达”的数据包；
- 为超时重传机制提供触发条件；
- 保证连接管理阶段（SYN / FIN 报文）的正确性。

校验设计总结 通过在应用层实现基于伪首部的 16 位反码校验和机制，本实验在 UDP 之上实现了有效的差错检测功能。该模块结构清晰、计算开销低，并能够与滑动窗口、重传和拥塞控制机制协同工作，为整个可靠数据传输协议提供了坚实的正确性保障。

3. 确认重传

为保证在不可靠的 UDP 传输之上实现可靠数据传输，本实验在应用层设计并实现了**确认与重传机制**。该机制与滑动窗口、差错检测模块协同工作，用于应对数据包丢失、乱序到达以及确认报文丢失等情况，从而保证数据能够被完整、正确地传输。

确认机制设计 这里我们主要关注报文格式、方式和时机的问题。

1. 确认报文格式

确认信息通过自定义数据报中的 **确认号（ack_num）**字段携带，并配合 **ACK 控制位**使用。确认报文本身同样参与校验和计算，确保确认信息的可靠性。

2. 确认方式

实验中采用**以累计确认为主、选择确认为辅**的确认策略：

- **累计确认**

  接收方在成功接收到从 `base_seq` 开始的连续数据包后，返回一个确认号，表示当前已经正确接收的最大连续序号加一。

- **选择确认（Selective ACK）**

  当窗口内数据包乱序到达时，接收方会记录已接收的数据包，并通过确认信息告知发送方哪些序号的数据已经正确接收，从而避免重复重传。

3. 确认发送时机

接收方在以下情况下发送确认报文：

- 成功接收到校验正确的数据包；
- 接收窗口发生滑动；
- 收到重复数据包但仍需提示发送方当前接收状态。

若接收到的数据包校验失败，则直接丢弃，不发送确认，由发送端通过超时机制进行处理。

重传机制设计 这里我们主要关注其与窗口的耦合性，以及丢包的判断处理。

1. 超时重传机制

发送端在发送每一个数据包后，都会记录其最近一次发送时间，并启动对应的超时检测。

- 若在预定超时时间内未收到该数据包的确认；
- 或确认号未能推进窗口起始位置；

则认为该数据包可能丢失，发送端会对其进行重传。

超时时间的设定参考网络往返时间（RTT），并在实验中采用固定阈值与动态检测相结合的方式。

2. 基于窗口的重传策略

发送端仅对**窗口内且未被确认的数据包**执行重传操作，已经确认的数据包不会被重复发送。

当确认号推进 `base` 后，窗口整体向前滑动，窗口外的数据包即被视为完成传输。

3. 确认丢失的处理

若数据包已被接收，但对应的确认报文在返回过程中丢失：

- 发送端在超时后会重新发送该数据包；
- 接收端检测到重复数据包后，不会重复交付数据，而是重新发送确认信息。

该机制能够保证即使 ACK 报文丢失，也不会影响数据的最终正确交付。

快速重传思想的体现 在实验实现中，当发送端连续接收到多个针对同一序号的重复确认时，可以推断后续数据包已到达但某一数据包发生丢失。此时，发送端可在超时发生之前对该数据包进行重传，从而减少等待超时所带来的延迟。

该设计在思想上与 TCP Reno 中的**快速重传机制**保持一致，但在实现上进行了简化，更适合实验环境。

与滑动窗口和差错检测的协同 确认与重传机制与其他模块的协同关系如下：

- **差错检测模块**

  负责判断数据包是否合法，为确认与否提供依据；

- **滑动窗口机制**

  限制可同时未确认的数据包数量，提高链路利用率；

- **选择确认机制**

  精确定位丢失数据包，减少不必要的重传；

- **超时检测**

  保证在确认丢失或数据包丢失的情况下仍能恢复传输。

重传设计总结 通过在应用层实现确认与重传机制，本实验在 UDP 之上成功实现了可靠数据传输的核心功能。该机制能够有效应对数据包丢失、确认丢失以及乱序到达等常见网络问题，并与滑动窗口和差错检测模块协同工作，为整个可靠传输协议提供了关键保障。

4. 流量控制

设计目标 流量控制的主要目的是协调发送端与接收端的数据传输速率，防止发送端发送过快而导致接收端缓存溢出。在基于 UDP 的可靠传输实验中，由于底层协议不提供任何流量控制机制，本实验在应用层自行实现了简化的流量控制策略。

固定窗口大小的流量控制策略 本实验采用**发送窗口与接收窗口大小相同的固定窗口机制**来实现流量控制。具体而言：

- 发送端和接收端均使用大小为 `WINDOW_SIZE` 的滑动窗口；
- 窗口大小在连接建立阶段即确定，在整个数据传输过程中保持不变；
- 发送端最多允许同时发送 `WINDOW_SIZE` 个尚未被确认的数据包；
- 接收端最多缓存 `WINDOW_SIZE` 个未按序交付的数据包。

该设计通过限制发送端在任意时刻可发送的数据包数量，实现了对发送速率的有效约束。

发送端流量控制行为 发送端通过维护发送窗口状态来实现流量控制，其主要约束体现在：

- 仅当 `next_seq` 落在发送窗口允许的范围内时，才允许发送新的数据包；
- 当发送窗口已满（即窗口内存在 `WINDOW_SIZE` 个未确认数据包）时，发送端将暂停发送新数据，等待确认报文到达；
- 随着确认报文的到来，窗口起始位置向前滑动，释放新的发送空间。

通过上述方式，发送端始终保证发送速率不超过接收端的处理能力。

接收端流量控制行为 接收端同样通过固定大小的接收窗口来限制自身缓存能力：

- 仅接收并缓存序号落在接收窗口范围内的数据包；
- 对于超出窗口范围的数据包，直接丢弃，不予缓存；
- 当接收窗口中从窗口起始序号开始的数据连续到达后，窗口向前滑动，并释放缓存空间。

这种设计能够防止接收端因乱序或过量数据到达而发生缓存溢出。

发送窗口与接收窗口等大的意义 发送窗口与接收窗口采用相同的固定大小具有以下优点：

1. **实现简单**

   无需额外设计窗口协商或动态调整机制，降低实现复杂度；

2. **避免窗口不匹配问题**

   防止发送端窗口大于接收端窗口而导致的缓存溢出；

3. **便于实验分析**

   固定窗口大小有利于对不同网络条件下协议行为进行对比和观察。

5. 拥塞控制

设计背景与目标 拥塞控制的目标是防止发送端向网络中注入过多数据而导致链路拥塞、丢包率上升甚至网络崩溃。与流量控制关注接收端处理能力不同，拥塞控制主要关注**网络整体承载能力**。

由于 UDP 协议本身不提供拥塞控制机制，本实验在应用层借鉴 TCP Reno 的思想，实现了一种简化的拥塞控制策略，用于在发生丢包或重传事件时动态调整发送速率。

核心参数设计 在 Reno 拥塞控制机制中，发送端主要维护以下两个关键参数：

- **拥塞窗口（Congestion Window，cwnd）**

  表示在不引发网络拥塞的前提下，发送端允许同时在网络中存在的最大未确认数据包数量。

- **慢启动阈值（ssthresh）**

  用于区分慢启动阶段和拥塞避免阶段，是发送速率调整的重要边界。

在本实验中，实际可发送的数据包数量由 `min(cwnd, WINDOW_SIZE)` 共同限制，确保与固定流量控制窗口兼容。

慢启动阶段（Slow Start） 在连接建立完成后，或在发生严重拥塞（如超时重传）之后，发送端进入慢启动阶段：

- 拥塞窗口 `cwnd` 从较小初始值开始；
- 每收到一个新的有效确认，`cwnd` 增加 1；
- 从整体效果上看，`cwnd` 在每个 RTT 内呈指数增长。

慢启动阶段的目的是快速探测当前网络环境下可用的带宽资源。

当 `cwnd` 增长至慢启动阈值 `ssthresh` 后，发送端进入拥塞避免阶段。

拥塞避免阶段（Congestion Avoidance） 在拥塞避免阶段，发送端采取更加保守的速率增长策略：

- 拥塞窗口按线性方式增长；
- 通常表现为每经过一个 RTT，`cwnd` 仅增加 1；
- 通过缓慢增加发送速率，降低触发网络拥塞的风险。

该阶段体现了 Reno 算法“加性增大（Additive Increase）”的思想。

快速重传与快速恢复 这部分是 RENO 的核心。

1. 快速重传（Fast Retransmit）

当发送端连续收到多个针对同一序号的重复确认时，认为某个数据包可能已丢失：

- 无需等待超时事件发生；
- 立即对该数据包进行重传；
- 有效降低因等待超时带来的额外延迟。

2. 快速恢复（Fast Recovery）
在快速重传触发后，Reno 不会像慢启动那样将发送速率降至最低，而是：

- 将慢启动阈值 `ssthresh` 设置为当前 `cwnd` 的一半；
- 将 `cwnd` 调整为新的 `ssthresh` 附近；
- 随后直接进入拥塞避免阶段。

该策略体现了 Reno 算法“乘性减小（Multiplicative Decrease）”的思想，避免在轻度拥塞情况下过度降低发送速率。

超时事件的处理 当发送端发生**超时重传**时，认为网络中存在较为严重的拥塞：

- 将 `ssthresh` 设为当前 `cwnd` 的一半；
- 将 `cwnd` 重置为初始值；
- 重新进入慢启动阶段。

该处理方式保证在网络状况恶化时能够迅速降低发送负载。

与流量控制的协同关系 在本实验中：

- **流量控制**通过固定大小的发送/接收窗口限制接收端压力；
- 通过动态调整 `cwnd` 控制网络注入速率；

实际可发送的数据包数量由两者共同决定：

$$\text{发送上限} = \min(\text{发送窗口大小}, \text{拥塞窗口} cwnd)$$

这种设计保证了协议在实验环境中同时具备流量控制与拥塞控制能力。

设计总结 通过在应用层引入基于 TCP Reno 的拥塞控制机制，本实验在 UDP 之上实现了对网络拥塞的基本感知与响应能力。该机制能够根据丢包和重传事件动态调整发送速率，在保证协议稳定性的同时，提高了整体传输效率。

尽管相较于完整 TCP Reno 实现进行了适当简化，但其核心思想与行为模式保持一致，达到了实验设计的预期目标。

四、 源码说明

笔者的通信框架分为服务端和客户端两个部分，接下来笔者会以功能模块为单位，根据运行逻辑为顺序进行介绍。

1. 建立连接：三次握手

和 lab1 类似地，我们在 server 端初始化了 sock 并指定 port:115，而 client 端则建立了向 router 地址 192.168.56.1:114 的传输通道。

事前准备

```
//server.cpp
WSADATA wsaData;
WSAStartup(MAKEWORD(2,2), &wsaData);

SetConsoleOutputCP(CP_UTF8);

SOCKET sock = socket(AF_INET, SOCK_DGRAM, 0);

struct sockaddr_in servaddr, cliaddr;
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(port);
servaddr.sin_addr.s_addr = INADDR_ANY;

bind(sock, (struct sockaddr*)&servaddr, sizeof(servaddr));
printf("Server running on port %d\n", port);

int cliLen = sizeof(cliaddr);
RDT_Packet pkt, send_pkt;

uint32_t server_seq = rand() % 10000;


u_long mode=1;
ioctlsocket(sock, FIONBIO, &mode);//设置非阻塞模式

printf("等待连接\n");



//client.cpp
WSADATA wsaData;
WSAStartup(MAKEWORD(2,2), &wsaData);

SetConsoleOutputCP(CP_UTF8);
SetConsoleCP(CP_UTF8);

SOCKET sock = socket(AF_INET, SOCK_DGRAM, 0);

struct sockaddr_in servaddr;
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(114);
servaddr.sin_addr.s_addr = inet_addr("192.168.56.1");//router, not server
socklen_t servLen = sizeof(servaddr);
```

```
44
45      srand((unsigned)time(NULL));
46      uint32_t client_seq = rand() % 10000;
47      uint32_t server_seq = 0;
48
49
50      u_long mode=1;
51      ioctlsocket(sock, FIONBIO, &mode);//设置非阻塞模式
```

这里使用了非阻塞的模式，是因为我们需要由不可靠的 UDP 协议搭建可靠的类 TCP 协议，如果阻塞则不好处理丢包问题，同时也会大大降低效率。

在建立完基本的连接后，采用三**次握手**机制。根据状态机实际分为以下几个阶段：

1.server 以 50ms 的周期轮询，试图接收 SYN 并紧接着回传 SYN+ACK；此时主动发包方是 client，以 1200ms 的周期轮询 server 可能回传的 SYN+ACK，若收到则发送 ACK 进入下一阶段，没收到则重发 SYN；

2.server 以 1200ms 的周期轮询，试图接收 ACK，若收到则不回应、顺利建立连接，收不到则重发 SYN+ACK，迫使 client 重发 ACK 确认；client 以 1200ms 的周期轮询，若收到 SYN+ACK 则重发 ACK，若没收到则说明连接成功。

三次握手

```
1   //server.cpp
2     while (1) {
3
4         Sleep(50);
5         int recl=recvfrom(sock, (char*)&pkt, sizeof(pkt), 0,
6                 (struct sockaddr*)&cliaddr, &cliLen);
7         if(recl<=0 || recl==SOCKET_ERROR){
8             //printf("未收到包，等待\n");
9             continue;
10        }
11        printf("收到请求\n");
12
13        // ---------- 校验 client -> server ----------
14        PseudoHeader ph_recv;
15        ph_recv.src_ip = cliaddr.sin_addr.s_addr;
16        ph_recv.dst_ip = inet_addr("192.168.56.1");
17        ph_recv.zero = 0;
18        ph_recv.protocol = 17;
19        ph_recv.length = htons(sizeof(RDT_Packet) - MAX_DATA_SIZE + pkt.
              length);
20
21        uint16_t ck = pkt.checksum;
22        pkt.checksum = 0;
23        if (ck != checksum_with_pseudo(&ph_recv, &pkt, pkt.length)){
24            printf("校验和错误(%d)，应为(%d)，跳过该包\n",
                  checksum_with_pseudo(&ph_recv, &pkt, pkt.length),ck);
25            continue;
26        }
```

```
27
28
29          // ---------- SYN ----------
30          if (pkt.flags & FLAG_SYN) {
31              send_pkt.seq_num = server_seq;
32              send_pkt.ack_num = pkt.seq_num + 1;
33              send_pkt.flags = FLAG_SYN | FLAG_ACK;
34              send_pkt.length = 0;
35              send_pkt.checksum = 0;
36
37              // ---------- server -> client ----------
38              PseudoHeader ph_send;
39              ph_send.src_ip = SERVER_IP;
40              ph_send.dst_ip = cliaddr.sin_addr.s_addr;
41              ph_send.zero = 0;
42              ph_send.protocol = 17;
43              ph_send.length = htons(sizeof(RDT_Packet) - MAX_DATA_SIZE);
44
45              send_pkt.checksum = checksum_with_pseudo(&ph_send, &send_pkt,
                    send_pkt.length);
46              sendto(sock, (char*)&send_pkt, sizeof(send_pkt), 0,
47                     (struct sockaddr*)&cliaddr, cliLen);
48
49              // ---------- 等 ACK ----------
50              while (1) {
51                  Sleep(1200);
52                  int recl=recvfrom(sock, (char*)&pkt, sizeof(pkt), 0,(struct
                        sockaddr*)&cliaddr, &cliLen);
53                  if(recl<=0 || recl==SOCKET_ERROR){
54                      printf("未收到包, 重发SYN+ACK\n");
55                      PseudoHeader ph_send;
56                      ph_send.src_ip = SERVER_IP;
57                      ph_send.dst_ip = cliaddr.sin_addr.s_addr;
58                      ph_send.zero = 0;
59                      ph_send.protocol = 17;
60                      ph_send.length = htons(sizeof(RDT_Packet) - MAX_DATA_SIZE
                            );
61
62                      send_pkt.checksum = checksum_with_pseudo(&ph_send, &
                            send_pkt, send_pkt.length);
63                      sendto(sock, (char*)&send_pkt, sizeof(send_pkt), 0,
64                          (struct sockaddr*)&cliaddr, cliLen);
65                      continue;
66                  }
67
68                  ck = pkt.checksum;
69                  pkt.checksum = 0;
70                  if (ck != checksum_with_pseudo(&ph_recv, &pkt, pkt.length)){
```

```
71                      printf("校验和错误(%d), 应为(%d), 重发SYN+ACK\n",
                            checksum_with_pseudo(&ph_recv, &pkt, pkt.length),ck);
72                      PseudoHeader ph_send;
73                      ph_send.src_ip = SERVER_IP;
74                      ph_send.dst_ip = cliaddr.sin_addr.s_addr;
75                      ph_send.zero = 0;
76                      ph_send.protocol = 17;
77                      ph_send.length = htons(sizeof(RDT_Packet) - MAX_DATA_SIZE
                            );
78
79                      send_pkt.checksum = checksum_with_pseudo(&ph_send, &
                            send_pkt, send_pkt.length);
80                      sendto(sock, (char*)&send_pkt, sizeof(send_pkt), 0,
81                          (struct sockaddr*)&cliaddr, cliLen);
82                      continue;
83                  }
84
85
86                  if ((pkt.flags & FLAG_ACK) && pkt.ack_num == server_seq + 1)
                        {
87                      printf("成功连接\n");
88                      connection_loop(sock, &cliaddr, &cliLen, server_seq + 1);
89                      return;
90                  }
91              }
92          }
93      }
94
95
96  //client.cpp
97  RDT_Packet pkt, recv_pkt;
98      PseudoHeader ph;
99
100     // ---------- SYN ----------
101     ph.src_ip = CLIENT_IP;
102     ph.dst_ip = inet_addr("192.168.56.1");
103     ph.zero = 0;
104     ph.protocol = 17;
105     ph.length = htons(sizeof(RDT_Packet) - MAX_DATA_SIZE);
106
107     pkt.seq_num = client_seq;
108     pkt.ack_num = 0;
109     pkt.flags = FLAG_SYN;
110     pkt.length = 0;
111     pkt.checksum = checksum_with_pseudo(&ph, &pkt, pkt.length);
112     sendto(sock, (char*)&pkt, sizeof(pkt), 0,
113             (struct sockaddr*)&servaddr, servLen);
114
```

```
115     // ———————— SYN+ACK ————————
116     while (1) {
117         printf("等待SYN+ACK\n");
118         Sleep(1200);
119         int recl=recvfrom(sock, (char*)&recv_pkt, sizeof(recv_pkt), 0,
120                 (struct sockaddr*)&servaddr, &servLen);
121         if(recl<=0 || recl==SOCKET_ERROR){
122             printf("未收到SYN+ACK包, 重发SYN\n");
123             sendto(sock, (char*)&pkt, sizeof(pkt), 0,(struct sockaddr*)&
                    servaddr, servLen);
124             continue;
125         }
126         printf("Received␣packet,␣checksum:␣%d\n", recv_pkt.checksum);
127 
128         PseudoHeader ph_recv;
129         ph_recv.src_ip = SERVER_IP; // server IP
130         ph_recv.dst_ip = CLIENT_IP;                    // client IP
131         ph_recv.zero = 0;
132         ph_recv.protocol = 17;
133         ph_recv.length = htons(sizeof(RDT_Packet) - MAX_DATA_SIZE);
134 
135         uint16_t ck = recv_pkt.checksum;
136         recv_pkt.checksum = 0;
137         if (ck != checksum_with_pseudo(&ph_recv, &recv_pkt, recv_pkt.length))
                {
138             printf("校验和错误(%d)应为(%d), 重发SYN\n", checksum_with_pseudo
                    (&ph_recv, &recv_pkt, recv_pkt.length), ck);
139             sendto(sock, (char*)&pkt, sizeof(pkt), 0,(struct sockaddr*)&
                    servaddr, servLen);
140             continue;
141         }
142 
143 
144         if ((recv_pkt.flags & FLAG_SYN) && (recv_pkt.flags & FLAG_ACK)) {
145             printf("Received␣SYN+ACK\n");
146             server_seq = recv_pkt.seq_num;
147             break;
148         }
149     }
150 
151     // ———————— ACK ————————
152     pkt.seq_num = client_seq + 1;
153     pkt.ack_num = server_seq + 1;
154     pkt.flags = FLAG_ACK;
155     pkt.length = 0;
156     pkt.checksum = 0;
157     pkt.checksum = checksum_with_pseudo(&ph, &pkt, pkt.length);
158     sendto(sock, (char*)&pkt, sizeof(pkt), 0,
```

```
159             (struct sockaddr*)&servaddr, servLen);
160
161     while(1){//根据是否接到到SYN+ACK包来判断ACK是否丢失
162         Sleep(1200);
163         memset(&recv_pkt, 0, sizeof(recv_pkt));
164         int recl=recvfrom(sock, (char*)&recv_pkt, sizeof(recv_pkt), 0,
165                 (struct sockaddr*)&servaddr, &servLen);
166         if(recl<=0 || recl==SOCKET_ERROR){//未收到, 连接成功
167             break;
168         }
169         else{
170             printf("收到SYN+ACK包, 需要重发ACK\n");
171             pkt.seq_num = client_seq + 1;
172             pkt.ack_num = server_seq + 1;
173             pkt.flags = FLAG_ACK;
174             pkt.length = 0;
175             pkt.checksum = 0;
176             pkt.checksum = checksum_with_pseudo(&ph, &pkt, pkt.length);
177             sendto(sock, (char*)&pkt, sizeof(pkt), 0,
178                 (struct sockaddr*)&servaddr, servLen);
179         }
180     }
181
182
183
184
185
186     printf("连接成功\n");
```

到这里，连接已经成功建立，进入到文件传输阶段。

2. 文件交互准备工作

在一开始，server 进入以 50ms 为周期的轮询状态，试图捕获 client 发送的文件信息包。

而 client 则打开了控制台输入，供用户输入当前目录下需要传输的文件名。它需要做两件事：一是把文件的信息发送给 server，让其能在服务端新建相应的文件空壳；二是把文件本身传输给 server。下面是它做的第一件事及 server 相应的片段：

文件传输预备

```
1   //server.cpp, 这部分封装了一个函数
2
3   void connection_loop(SOCKET sock, struct sockaddr_in* cliaddr, int* cliLen,
        uint32_t server_seq) {
4       RDT_Packet pkt, send_pkt;
5       FILE* fp = NULL;
6       uint32_t expected_seq = 0;
7       int fileSize = 0;
8       int total_received = 0;
9       char filename[512] = {0};
```

```
    int stage = 0; // 0: 等文件名, 1: 等文件大小, 2: 数据传输
    uint32_t client_data_base = 0;

    _mkdir("./serverrecv");
    DWORD start_time;

    while (1) {
        //printf("**循环接收包\n");
        memset(&pkt, 0, sizeof(pkt));
        if(stage==0)    Sleep(50);
        else Sleep(1200);

        int recv_len = recvfrom(sock, (char*)&pkt, sizeof(pkt), 0,
                                (struct sockaddr*)cliaddr, cliLen);
        //printf("**收到包\n");
        //printf("[SERVER] recv_len = %d\n", recv_len);
        //printf("[SERVER] checksum offset = %zu\n",
        //    offsetof(RDT_Packet, checksum));
        if (recv_len <= 0){
            continue;
        }

        // 校验伪头 + checksum
        PseudoHeader ph_recv;
        ph_recv.src_ip = cliaddr->sin_addr.s_addr;
        ph_recv.dst_ip = inet_addr("192.168.56.1");
        ph_recv.zero = 0;
        ph_recv.protocol = 17;
        ph_recv.length = htons(sizeof(RDT_Packet) - MAX_DATA_SIZE + pkt.
            length);

        //printf("pkt.length=%d\n",pkt.length);
        uint16_t ck = pkt.checksum;
        pkt.checksum = 0;
        if (ck != checksum_with_pseudo(&ph_recv, &pkt, pkt.length)){
            //printf("length=%d",pkt.length);
            printf("校验和错误(%d), 应为(%d), 跳过该包\n",
                checksum_with_pseudo(&ph_recv, &pkt, pkt.length),ck);
            continue;
        }

        // ---------- FIN ----------
        if (pkt.flags & FLAG_FIN) {
            printf("收到 FIN, 准备回传挥手ACK\n");
            Sleep(50);
            PseudoHeader ph_send;
            ph_send.src_ip = SERVER_IP;
            ph_send.dst_ip = cliaddr->sin_addr.s_addr;
```

```
56              ph_send.zero = 0;
57              ph_send.protocol = 17;
58              ph_send.length = htons(sizeof(RDT_Packet) - MAX_DATA_SIZE);
59
60              send_pkt.seq_num = server_seq;
61              send_pkt.ack_num = pkt.seq_num + 1;
62              send_pkt.flags = FLAG_ACK;
63              send_pkt.length = 0;
64              send_pkt.checksum = checksum_with_pseudo(&ph_send, &send_pkt,
                    send_pkt.length);
65              sendto(sock, (char*)&send_pkt, sizeof(send_pkt), 0,
66                      (struct sockaddr*)cliaddr, *cliLen);
67
68              printf("回传后, 等待发送第三次挥手\n");
69              Sleep(50);
70
71              send_pkt.flags = FLAG_FIN;
72              send_pkt.checksum = checksum_with_pseudo(&ph_send, &send_pkt,
                    send_pkt.length);
73              sendto(sock, (char*)&send_pkt, sizeof(send_pkt), 0,
74                      (struct sockaddr*)cliaddr, *cliLen);
75
76              if (fp) fclose(fp);
77              printf("发送了第三次, 等待客户端回应第四次ACK\n");
78              Sleep(50);
79              while(1){
80                  memset(&pkt, 0, sizeof(pkt));
81                  Sleep(1200);
82                  recv_len=recvfrom(sock, (char*)&pkt, sizeof(pkt), 0,
83                                    (struct sockaddr*)cliaddr, cliLen);
84                  if (recv_len <= 0){
85                      printf("未收到包, 重发FIN\n");
86                      sendto(sock, (char*)&send_pkt, sizeof(send_pkt), 0,
87                          (struct sockaddr*)cliaddr, *cliLen);
88                      continue;
89                  }
90
91
92                  PseudoHeader ph_recv;
93                  ph_recv.src_ip = CLIENT_IP;  // client 真正 IP
94                  ph_recv.dst_ip = SERVER_IP;                  // server 自己 IP
95                  ph_recv.zero = 0;
96                  ph_recv.protocol = 17;
97                  // 注意: checksum 长度 = sizeof(RDT_Packet) - MAX_DATA_SIZE +
                        pkt.length
98                  ph_recv.length = htons(sizeof(RDT_Packet) - MAX_DATA_SIZE +
                        pkt.length);
99
```

```
100                 // 不要 memset pkt.truedata, 保持接收到的原样
101                 uint16_t ck = pkt.checksum;
102                 pkt.checksum = 0;
103                 //printf("len=%d???/n", (sizeof(RDT_Packet) - MAX_DATA_SIZE +
                        pkt.length));
104                 if (ck != checksum_with_pseudo(&ph_recv, &pkt, pkt.length)){
105                     printf("校验和错误(%d), 应为(%d), 跳过该包\n",
106                         checksum_with_pseudo(&ph_recv, &pkt, pkt.length), ck)
                            ;
107                     //printf("包信息: seq_num=%u, ack_num=%u, flags=%u,
                            length=%u, checksum=%u\n", pkt.seq_num, pkt.ack_num,
                            pkt.flags, pkt.length, pkt.checksum);
108                     continue;
109                 }
110
111                 printf("收到ACK, 准备关闭\n");
112                 break;
113             }
114             printf("连接成功关闭,5000ms后回退\n");
115             Sleep(5000);
116             return;
117         }
118
119         // ---------- 文件名 ----------
120         if (stage == 0) {
121             memcpy(filename, pkt.truedata, pkt.length);
122             filename[pkt.length] = 0; // 确保字符串结束
123
124             start_time = GetTickCount();//开始计时
125
126             printf("尝试接收文件:␣%s\n", filename);
127             stage = 1;
128             expected_seq = pkt.seq_num + pkt.length;
129
130             // 发送 ACK
131             PseudoHeader ph_send;
132             ph_send.src_ip = SERVER_IP;
133             ph_send.dst_ip = cliaddr->sin_addr.s_addr;
134             ph_send.zero = 0;
135             ph_send.protocol = 17;
136             ph_send.length = htons(sizeof(RDT_Packet) - MAX_DATA_SIZE);
137
138             send_pkt.seq_num = server_seq;
139             send_pkt.ack_num = expected_seq;
140             send_pkt.flags = FLAG_ACK;
141             send_pkt.length = 0;
142             send_pkt.checksum = checksum_with_pseudo(&ph_send, &send_pkt,
                    send_pkt.length);
```

```
143             sendto(sock, (char*)&send_pkt, sizeof(send_pkt), 0,
144                    (struct sockaddr*)cliaddr, *cliLen);
145         }
146         // ---------- 文件大小 ----------
147         else if (stage == 1) {//printf("debug\n");
148             char sizeStr[32] = {0};
149             memcpy(sizeStr, pkt.truedata, pkt.length);
150             sizeStr[pkt.length] = 0;
151             fileSize = atoi(sizeStr);
152             printf("文件大小: %d bytes\n", fileSize);
153 
154             char path[512];
155             snprintf(path, sizeof(path), "./serverrecv/%s", filename);
156             fp = fopen_utf8(path, "wb");
157             if (!fp) {
158                 printf("Failed to open file %s\n", path);
159                 return;
160             }
161             stage = 2;
162             client_data_base = pkt.seq_num + pkt.length;
163             expected_seq = client_data_base;
164             total_received = 0;
165 
166             // 发送 ACK
167             PseudoHeader ph_send;
168             ph_send.src_ip = SERVER_IP;
169             ph_send.dst_ip = cliaddr->sin_addr.s_addr;
170             ph_send.zero = 0;
171             ph_send.protocol = 17;
172             ph_send.length = htons(sizeof(RDT_Packet) - MAX_DATA_SIZE);
173 
174             send_pkt.seq_num = server_seq;
175             send_pkt.ack_num = expected_seq;
176             send_pkt.flags = FLAG_ACK;
177             send_pkt.length = 0;
178             send_pkt.checksum = checksum_with_pseudo(&ph_send, &send_pkt,
                    send_pkt.length);
179             sendto(sock, (char*)&send_pkt, sizeof(send_pkt), 0,
180                    (struct sockaddr*)cliaddr, *cliLen);
181             printf("准备进入文件传输\n");
182 
183             //在这里就开始滑动窗口接收
184             sliding_window_recv(fp, sock, cliaddr, cliLen, ph_recv, filename,
                    &stage, fileSize, client_data_base);
185             stage=0;
186 
187 
188 
```

```
189             DWORD end_time = GetTickCount();
190             double elapsed_sec = (end_time - start_time) / 1000.0;
191             double throughput = fileSize / elapsed_sec / 1024.0; // KB/s
192             printf("文件 %s 传输成功\n", filename);
193             printf("服务端传输总时间: %.2f 秒\n", elapsed_sec);
194             printf("平均吞吐率: %.2f KB/s\n", throughput);
195
196         }
197         // ---------- 文件数据 ----------
198         else if (stage == 2 && pkt.length > 0) {
199         printf("ERROR:不应该出现在这里\n");
200
201
202         }
203     }
204 }
205
206
207
208
209 //client.cpp
210 // ---------- 文件交互 ----------
211     while (1) {
212         char filename[256];
213         printf("输入文件名（end 结束）: ");
214         fgets(filename, sizeof(filename), stdin);
215         filename[strcspn(filename, "\r\n")] = 0;
216         if (strcmp(filename, "end") == 0) break;
217         printf("尝试传输文件...\n");
218         send_file(sock, &servaddr, servLen, &client_seq, server_seq, filename
                );
219     }
220
221 //同样是client.cpp
222
223 void send_file(SOCKET sock, struct sockaddr_in* servaddr, socklen_t servLen,
224                uint32_t* seq, uint32_t server_seq,
225                const char* filename) {
226
227     FILE* fp = fopen_utf8(filename, "rb");
228     if (!fp) {
229         printf("无法打开文件 %s\n", filename);
230         return;
231     }
232
233     DWORD start_time = GetTickCount();//开始计时
234
235     RDT_Packet pkt, recv_pkt;
```

```
236     PseudoHeader ph;
237
238     ph.src_ip = CLIENT_IP;
239     ph.dst_ip = servaddr->sin_addr.s_addr;
240     ph.zero = 0;
241     ph.protocol = 17;
242
243     PseudoHeader ph_recv;
244     ph_recv.src_ip = servaddr->sin_addr.s_addr;
245     ph_recv.dst_ip = CLIENT_IP;
246     ph_recv.zero = 0;
247     ph_recv.protocol = 17;
248
249     printf("尝试开始发送文件: %s\n", filename);
250     // ---------- 发送文件名 ----------
251     pkt.seq_num = *seq;
252     pkt.ack_num = server_seq;
253     pkt.flags = 0;
254     pkt.length = (uint16_t)strlen(filename);
255     memcpy(pkt.truedata, filename, pkt.length);
256
257     ph.length = htons(sizeof(RDT_Packet) - MAX_DATA_SIZE + pkt.length);
258
259     pkt.checksum = 0;
260     pkt.checksum = checksum_with_pseudo(&ph, &pkt, pkt.length);
261     sendto(sock, (char*)&pkt, sizeof(pkt), 0,
262            (struct sockaddr*)servaddr, servLen);
263
264     // 等 ACK
265     while (1) {
266         printf("等待文件名ACK\n");
267         Sleep(1200);
268         int recl=recvfrom(sock, (char*)&recv_pkt, sizeof(recv_pkt), 0,
269                 (struct sockaddr*)servaddr, &servLen);
270         if(recl<=0 || recl==SOCKET_ERROR){
271             printf("未收到文件名ACK包, 重发文件名\n");
272             sendto(sock, (char*)&pkt, sizeof(pkt), 0,
273                    (struct sockaddr*)servaddr, servLen);
274             continue;
275         }
276
277
278         uint16_t ck = recv_pkt.checksum;
279         recv_pkt.checksum = 0;
280
281         ph_recv.src_ip = SERVER_IP;
282         ph_recv.dst_ip = CLIENT_IP;
283         ph_recv.zero = 0;
```

```
284         ph_recv.protocol = 17;
285         ph_recv.length = htons(sizeof(RDT_Packet) - MAX_DATA_SIZE + recv_pkt.
                length);
286         if (ck != checksum_with_pseudo(&ph_recv, &recv_pkt, recv_pkt.length))
                {
287             printf("校验和错误(%d), 应为(%d), 重发文件名\n",
                    checksum_with_pseudo(&ph_recv, &recv_pkt, recv_pkt.length),
                    ck);
288             sendto(sock, (char*)&pkt, sizeof(pkt), 0,
289                    (struct sockaddr*)servaddr, servLen);
290             continue;
291         }
292
293
294         if ((recv_pkt.flags & FLAG_ACK) &&
295             recv_pkt.ack_num == *seq + pkt.length)
296             break;
297     }
298
299     *seq += pkt.length;
300
301     // ---------- 发送文件大小 ----------
302     fseek(fp, 0, SEEK_END);
303     long fileSize = ftell(fp);
304     fseek(fp, 0, SEEK_SET);
305
306     char sizeStr[32];
307     snprintf(sizeStr, sizeof(sizeStr), "%ld", fileSize);
308
309     pkt.seq_num = *seq;
310     pkt.ack_num = server_seq;
311     pkt.flags = 0;
312     pkt.length = (uint16_t)strlen(sizeStr);
313     memcpy(pkt.truedata, sizeStr, pkt.length);
314
315     ph_recv.src_ip = SERVER_IP;
316     ph_recv.dst_ip = CLIENT_IP;
317     ph_recv.zero = 0;
318     ph_recv.protocol = 17;
319     ph.length = htons(sizeof(RDT_Packet) - MAX_DATA_SIZE + pkt.length);
320
321     pkt.checksum = 0;
322     pkt.checksum = checksum_with_pseudo(&ph, &pkt, pkt.length);
323     printf("发送文件大小: %s␣字节\n", sizeStr);
324     sendto(sock, (char*)&pkt, sizeof(pkt), 0,
325            (struct sockaddr*)servaddr, servLen);
326
327     // 等 ACK
```

```
328     while (1) {
329         Sleep(1200);
330         int recl=recvfrom(sock, (char*)&recv_pkt, sizeof(recv_pkt), 0,
331                 (struct sockaddr*)servaddr, &servLen);
332         if(recl<=0 || recl==SOCKET_ERROR){
333             printf("未收到文件大小ACK包, 重发文件大小\n");
334             sendto(sock, (char*)&pkt, sizeof(pkt), 0,
335                     (struct sockaddr*)servaddr, servLen);
336             continue;
337         }
338 
339         uint16_t ck = recv_pkt.checksum;
340         recv_pkt.checksum = 0;
341         ph_recv.length = htons(sizeof(RDT_Packet) - MAX_DATA_SIZE + recv_pkt.
                length);
342         if (ck != checksum_with_pseudo(&ph_recv, &recv_pkt, recv_pkt.length))
                {
343             printf("校验和错误(%d), 应为(%d), 重发文件大小\n",
                    checksum_with_pseudo(&ph_recv, &recv_pkt, recv_pkt.length),
                    ck);
344             sendto(sock, (char*)&pkt, sizeof(pkt), 0,
345                     (struct sockaddr*)servaddr, servLen);
346             continue;
347         }
348 
349 
350         if ((recv_pkt.flags & FLAG_ACK) &&
351             recv_pkt.ack_num == *seq + pkt.length)
352             break;
353     }
354 
355     *seq += pkt.length;
356 
357     // ---------- 发送文件数据 ----------
358     sliding_window_send(WINDOW_SIZE, seq, fp, ph, sock, servaddr, servLen);
359 
360     fclose(fp);
361 
362 
363 
364     DWORD end_time = GetTickCount();
365     double elapsed_sec = (end_time - start_time) / 1000.0;
366     double throughput = fileSize / elapsed_sec / 1024.0; // KB/s
367     printf("文件 %s 传输成功\n", filename);
368     printf("客户端传输总时间: %.2f 秒\n", elapsed_sec);
369     printf("平均吞吐率: %.2f KB/s\n", throughput);
370 }
```

可以注意到在这里的交互中仍然沿用了和建立连接时差不多的机制，基本上是以 1200ms 为周期进行 ACK 的轮询和发送；同时，为了防止数据损坏，校验码的作用也在此清晰可见——当校验码不同时，我们会丢弃这个包，并视作没有收到。而校验码的函数定义如下：

校验和函数

```
  //common.h

// ---------------------------------
// 校验和函数
// ---------------------------------
static inline uint16_t checksum_with_pseudo(PseudoHeader* ph, void* truedata,
     size_t len) {
    uint32_t retsum = 0;
    uint16_t* ph_ptr = (uint16_t*)ph;
    for (size_t i = 0; i < sizeof(PseudoHeader)>>1; ++i) retsum += ph_ptr[i];
    if (sizeof(PseudoHeader) % 2) retsum += ((uint8_t*)ph)[sizeof(
        PseudoHeader)-1];

    uint16_t* dataptr = (uint16_t*)truedata;
    for (size_t i = 0; i < len>>1; ++i) retsum += dataptr[i];
    if (len % 2) retsum += ((uint8_t*)truedata)[len-1];

    // 折叠进位
    while (retsum >> 16) retsum = (retsum & 0xFFFF) + (retsum >> 16);

    return ~retsum;
}
```

正如同我们在协议介绍处所说的，这个校验和函数实现的是一种**带伪首部的 16 位反码校验机制**，整体流程与 UDP/TCP 的校验和计算方式一致。函数首先将用于累加的变量 `retsum` 初始化为 0，作为整个校验过程中的中间和。随后，函数把伪首部 `PseudoHeader` 视为一段连续的内存，并按 **16 位无符号整数**的方式逐项累加其中的内容。如果伪首部的总字节数为奇数，则最后剩余的 1 个字节也会被单独加入到累加和中，以保证所有字节都参与校验。

在完成伪首部的累加后，函数继续对真实报文数据进行同样的处理。它将 `truedata` 指向的报文内容按 16 位为单位累加到 `retsum` 中；如果数据长度 `len` 为奇数，则最后一个字节同样会被补入计算。这一步确保了报文首部和数据字段中的每一个字节都会影响最终的校验结果，从而提高差错检测能力。

由于累加过程中使用的是反码加法，可能会产生超过 16 位的进位，因此函数接下来通过循环的方式对高 16 位的进位进行折叠处理：将高位进位加回到低 16 位中，直到结果完全落在 16 位范围内。这一步模拟了网络协议中规定的反码加法规则，保证校验和计算结果的正确性。

最后，函数对折叠后的 16 位结果按位取反，并将其作为校验和返回。发送端在构造报文时把该值写入 `checksum` 字段；接收端在校验时对包含该字段在内的报文重新计算校验和，如果结果为全 1，则说明在传输过程中未检测到差错，否则认为报文发生了损坏并予以丢弃。

3. 文件交互：滑动窗口 +RENO

笔者对于文件传输的实现，从简单到复杂一共经历了三个阶段：

1. 普通的等待确认协议
2. 固定窗口大小的滑动窗口协议
3. **结合 RENO 的滑动窗口协议**

其中，阶段 3 是笔者最终实现的版本；而我们的介绍会基于版本 2，围绕版本 3 进行开展。

在 client 端,我们维护一个 SendWindow 对象,内含窗口格子等信息,我们用 WINDOW_SIZE 全局变量来控制。起始时会先初始化它，并读取窗口格子数量个文件数据包。在**朴素的滑动窗口**中，我们会在大循环内部用一个 while 循环控制“窗口固定时”的逻辑——轮询窗口中包的状态以确认是否被确认，再在每次循环中重发未确认或未发过的包；而后检查窗口头是否被确认，若确认则左移窗口，跳出内部循环。

而在朴素的滑动窗口机制之上，这里引入了基于 TCP Reno 思想的拥塞控制。我们额外维护了 `cwnd` 表示拥塞窗口大小（以“包”为单位），`ssthresh` 表示慢启动阈值，同时记录最近一次收到的新 ACK 值 `last_ack` 以及重复 ACK 的计数器 `dup_ack_cnt`。实际允许发送的数据量由发送窗口大小和拥塞窗口大小共同限制，即每次最多只能在窗口中放入 `min(cwnd, WINDOW_SIZE)` 个未确认分组。

对于尚未被确认的分组，如果是第一次发送，或者已经超过了设定的超时时间，就会触发发送或重传。若检测到超时，这是 Reno 中最严重的拥塞信号，程序会将 `ssthresh` 置为当前 `cwnd` 的一半（至少为 1），并将 `cwnd` 重置为 1，从而重新进入慢启动阶段，同时清空重复 ACK 计数。

在每一轮发送之后，client 会以非阻塞方式尝试接收 server 返回的 ACK 分组。收到 ACK 后先根据伪首部重新计算校验和，若校验失败则直接丢弃，等待下一次轮询。若 ACK 有效，则进入 Reno 的确认处理逻辑：当 ACK 号等于上一次确认号时，认为这是重复 ACK，重复次数加一；当连续收到三个重复 ACK 时，触发快速重传机制，将 `ssthresh` 更新为当前 `cwnd` 的一半，并把 `cwnd` 调整为 `ssthresh + 3`，同时立即重传窗口头部的分组。若 ACK 号大于 `last_ack`，说明收到了新的确认，此时重置重复 ACK 计数，并根据当前所处阶段调整 `cwnd`：在慢启动阶段（`cwnd < ssthresh`）下按指数增长，每收到一个新 ACK 就将 `cwnd` 加一；在拥塞避免阶段则采用近似的线性增长策略，使窗口增长速度明显放缓。

通过这种方式，client 在保持固定接收窗口大小的前提下，将 Reno 拥塞控制机制自然地嵌入到滑动窗口发送逻辑中，实现了慢启动、拥塞避免、超时重传以及快速重传等关键行为，使得发送速率能够根据网络状况动态调整，而不是一味地以最大窗口发送数据。

发送窗口

```cpp
//client.cpp


void sliding_window_send(int window_size,uint32_t *seq,FILE *fp,PseudoHeader ph,SOCKET sock,sockaddr_in *servaddr,socklen_t servLen){//滑动窗口发送端

    SendWindow win;
    memset(&win, 0, sizeof(win));
    win.base = *seq;  // 初始 seq
    win.next_seq = *seq;
    win.count = 0;
    //初始化窗口

    // ========== [Reno] 新增变量 ==========
    int cwnd = 1;                    // 拥塞窗口（以包为单位）
    int ssthresh = WINDOW_SIZE;      // 慢启动阈值
```

```
    uint32_t last_ack = win.base;
    int dup_ack_cnt = 0;
    // ==========================================================

    char buf[512];
    size_t n;
    while((n = fread(buf, 1, sizeof(buf), fp))>0){//只要没发完就一直这么做

        // ================== [Reno] 限制发送窗口 ==================
        int send_limit = cwnd < WINDOW_SIZE ? cwnd : WINDOW_SIZE;
        if (win.count < send_limit) {
        // ==========================================================
            SendSlot *slot = &win.slots[win.count];
            slot->pkt.seq_num = win.next_seq;
            slot->pkt.length = (uint16_t)n;
            memcpy(slot->pkt.truedata, buf, n);
            ph.length = htons(sizeof(RDT_Packet) - MAX_DATA_SIZE + slot->pkt.
                length);
            slot->pkt.checksum = 0;
            slot->pkt.checksum = checksum_with_pseudo(&ph, &slot->pkt, slot->
                pkt.length);
            //printf("length=%d",slot->pkt.length);
            slot->send_time = GetTickCount();
            slot->acked = -1;

            win.next_seq += n;
            win.count++;
        }

        //这一部分是滑动窗口固定时的逻辑
        RDT_Packet pkt;//暂存的server返回包
        while(1){
            if(!win.count)  break;//没有数据包

            //先传包
            for (int i = 0; i < win.count; i++) {
                SendSlot *slot = &win.slots[i];
                if (!slot->acked || slot->acked == -1) {//没被确认
                    if (GetTickCount() - slot->send_time > 500 || slot->acked
                        == -1) {//超时重传, 或第一次传输
                        if(slot->acked==0){
                            //printf("重传包, seq = %d\n", slot->pkt.seq_num)
                                ;
                        }
                        else{
                            //printf("第一次传输包, seq = %d\n", slot->pkt.
                                seq_num);
                        }
```

```
59
60                  // ========== [Reno] 超时处理 ==========
61                  if (GetTickCount() - slot->send_time > 500) {
62                      ssthresh = cwnd / 2;
63                      if (ssthresh < 1) ssthresh = 1;
64                      cwnd = 1;
65                      dup_ack_cnt = 0;
66                  }
67                  //
                        ==========
68
69                  slot->acked=0;//标记为传输过但未被确认
70
71                  sendto(sock, (char*)&slot->pkt, sizeof(slot->pkt), 0,
72                      (struct sockaddr*)servaddr, servLen);
73                  slot->send_time = GetTickCount();
74              }
75          }
76      }
77
78      //然后看看有没有server返回包
79      memset(&pkt, 0, sizeof(pkt));
80      int recvl=recvfrom(sock,(char*)&pkt,sizeof(pkt),0,(struct
            sockaddr*)servaddr,&servLen);
81      //非阻塞地接收server返回包
82      PseudoHeader ph_recv;
83      ph_recv.src_ip = SERVER_IP;
84      ph_recv.dst_ip = CLIENT_IP;
85      ph_recv.zero = 0;
86      ph_recv.protocol = 17;
87      ph_recv.length = htons(sizeof(RDT_Packet) - MAX_DATA_SIZE + pkt.
            length);
88      int ck=checksum_with_pseudo(&ph_recv,&pkt,pkt.length);
89      if(recvl==SOCKET_ERROR || ck!=pkt.checksum){//没收到包或包损坏,
            休息一小会继续检测即可
90          if (recvl != SOCKET_ERROR) {
91              printf("校验和错误(计算得%d, 应为%d), 跳过该包\n", ck,
                    pkt.checksum);
92          }
93          continue;
94      }
95      else{//接收到返回包, 更新slot确认情况
96
97          // ========== [Reno] ACK 处理 ==========
98          if (pkt.ack_num == last_ack) {
99              dup_ack_cnt++;
```

```
100                     if (dup_ack_cnt == 3) {
101                         // 快速重传
102                         ssthresh = cwnd / 2;
103                         if (ssthresh < 1) ssthresh = 1;
104                         cwnd = ssthresh + 3;
105
106                         SendSlot *slot = &win.slots[0];
107                         sendto(sock, (char*)&slot->pkt, sizeof(slot->pkt), 0,
108                                (struct sockaddr*)servaddr, servLen);
109                         slot->send_time = GetTickCount();
110                     }
111                 }
112                 else if (pkt.ack_num > last_ack) {
113                     last_ack = pkt.ack_num;
114                     dup_ack_cnt = 0;
115
116                     if (cwnd < ssthresh) {
117                         cwnd++;             // 慢启动
118                     } else {
119                         cwnd += 1 / cwnd; // 拥塞避免（近似）
120                     }
121                 }
122                 // ==========================================================
123
124                 for(int i=0;i<win.count;++i){
125                     SendSlot *slot=&win.slots[i];
126                     if(slot->pkt.seq_num==pkt.ack_num){
127                         slot->acked=1;
128                         break;
129                     }
130                 }
131             }
132
133             if(win.slots[0].acked==1){
134                 //如果窗口头被确认，滑动窗口
135                 win.base+=win.slots[0].pkt.length;
136                 for(int i=0;i<win.count-1;i++){
137                     win.slots[i]=win.slots[i+1];
138                 }
139                 --win.count;
140                 break;//出去继续读取下一包准备发送
141             }
142         }
143     }
144 }
```

而后则是 server 端的接收逻辑。接收端同样维护一个接收窗口 `RecvWindow`，其中保存当前期望接收的序号 `base_seq` 以及窗口内各个槽位的接收状态。函数开始时初始化窗口，将 `base_seq`

设为初始期望序号，并清空所有槽位，随后进入一个循环，不断从 socket 中接收来自发送端的数据包。

每当收到一个数据包后，首先根据发送方和接收方的 IP 等信息构造伪首部，重新计算校验和并与包中携带的校验和进行比较，若校验失败则认为数据损坏，直接丢弃该包而不做进一步处理。若校验正确，则根据该包的序号与当前 `base_seq` 计算它在接收窗口中的相对位置，如果该位置超出窗口范围，说明这是一个过旧或过新的分组，同样直接丢弃。

对于位于窗口范围内的有效分组，接收端会将其存入对应的窗口槽位，并标记为“已收到”。如果该分组之前已经接收过，则将其视为重复包，仅用于确认而不再次缓存。无论是否重复，只要校验正确，接收端都会立即向发送端返回一个选择确认 ACK，ACK 号对应当前收到的数据包序号，用于通知发送端该分组已经成功到达。

在缓存分组之后，接收端会检查窗口头部的槽位是否已经收到数据。如果窗口最前端的分组已经就绪，说明从当前 `base_seq` 开始的数据是连续完整的，此时就将该分组的数据按顺序写入文件，并更新 `base_seq`。随后窗口整体向前滑动，继续检查新的窗口头，只要连续分组都已经收到，就会不断写入并滑动窗口。

整个过程会持续进行，直到累计接收的数据长度等于文件总大小为止，此时说明文件已经完整接收，接收端关闭文件并结束接收逻辑。整体来看，该接收流程实现了基于固定大小接收窗口的选择重传机制，能够正确处理乱序到达、重复到达以及数据损坏等情况，同时通过逐包 ACK 配合发送端的滑动窗口与拥塞控制，保证数据可靠、有序地写入文件。

如此，我们的主体任务就完成了。

4. 断开连接：四次挥手

断开连接的逻辑采用四次挥手：在控制台输入 end 时，先由 client 端发送 FIN 信号，而后 server 连续发送一个 ACK 和一个 FIN；最后 client 发送一个 ACK，收到后双方自行关闭连接退出。

其逻辑和三次握手类似，控制一个 1200ms 的轮询周期以确保交互通畅；若 client 未能收到 ACK 或 FIN，就重发自己的 FIN；若 client 在最后一轮后收到了对方的多余回信，说明末尾 ACK 未发出，重发即可。

断开连接

```
//server.cpp
//在接收包的while中
// ---------- FIN ----------
      if (pkt.flags & FLAG_FIN) {
          printf("收到 FIN，准备回传挥手ACK\n");
          Sleep(50);
          PseudoHeader ph_send;
          ph_send.src_ip = SERVER_IP;
          ph_send.dst_ip = cliaddr->sin_addr.s_addr;
          ph_send.zero = 0;
          ph_send.protocol = 17;
          ph_send.length = htons(sizeof(RDT_Packet) - MAX_DATA_SIZE);

          send_pkt.seq_num = server_seq;
          send_pkt.ack_num = pkt.seq_num + 1;
          send_pkt.flags = FLAG_ACK;
          send_pkt.length = 0;
```

```
18          send_pkt.checksum = checksum_with_pseudo(&ph_send, &send_pkt,
                send_pkt.length);
19          sendto(sock, (char*)&send_pkt, sizeof(send_pkt), 0,
20                 (struct sockaddr*)cliaddr, *cliLen);
21
22          printf("回传后, 等待发送第三次挥手\n");
23          Sleep(50);
24
25          send_pkt.flags = FLAG_FIN;
26          send_pkt.checksum = checksum_with_pseudo(&ph_send, &send_pkt,
                send_pkt.length);
27          sendto(sock, (char*)&send_pkt, sizeof(send_pkt), 0,
28                 (struct sockaddr*)cliaddr, *cliLen);
29
30          if (fp) fclose(fp);
31          printf("发送了第三次, 等待客户端回应第四次ACK\n");
32          Sleep(50);
33          while(1){
34              memset(&pkt, 0, sizeof(pkt));
35              Sleep(1200);
36              recv_len=recvfrom(sock, (char*)&pkt, sizeof(pkt), 0,
37                                (struct sockaddr*)cliaddr, cliLen);
38              if (recv_len <= 0){
39                  printf("未收到包, 重发FIN\n");
40                  sendto(sock, (char*)&send_pkt, sizeof(send_pkt), 0,
41                      (struct sockaddr*)cliaddr, *cliLen);
42                  continue;
43              }
44
45
46              PseudoHeader ph_recv;
47              ph_recv.src_ip = CLIENT_IP;  // client 真正 IP
48              ph_recv.dst_ip = SERVER_IP;                  // server 自己 IP
49              ph_recv.zero = 0;
50              ph_recv.protocol = 17;
51              // 注意: checksum 长度 = sizeof(RDT_Packet) - MAX_DATA_SIZE +
                    pkt.length
52              ph_recv.length = htons(sizeof(RDT_Packet) - MAX_DATA_SIZE +
                    pkt.length);
53
54              // 不要 memset pkt.truedata, 保持接收到的原样
55              uint16_t ck = pkt.checksum;
56              pkt.checksum = 0;
57              //printf("len=%d???/n", (sizeof(RDT_Packet) - MAX_DATA_SIZE +
                    pkt.length));
58              if (ck != checksum_with_pseudo(&ph_recv, &pkt, pkt.length)){
59                  printf("校验和错误(%d), 应为(%d), 跳过该包\n",
60                      checksum_with_pseudo(&ph_recv, &pkt, pkt.length), ck)
```

```
                        ;
61                  //printf("包信息: seq_num=%u, ack_num=%u, flags=%u,
                        length=%u, checksum=%u\n", pkt.seq_num, pkt.ack_num,
                        pkt.flags, pkt.length, pkt.checksum);
62                  continue;
63              }
64
65              printf("收到ACK, 准备关闭\n");
66              break;
67          }
68          printf("连接成功关闭,5000ms后回退\n");
69          Sleep(5000);
70          return;
71      }
72
73
74
75  //client.cpp
76
77  // ========== 关闭连接 ==========
78  void close_connection(SOCKET sock, struct sockaddr_in* servaddr, socklen_t
        servLen,
79                        uint32_t* seq, uint32_t server_seq) {
80      RDT_Packet pkt, recv_pkt;
81      printf("正在关闭连接\n");
82
83      // ---- FIN ----
84      pkt.seq_num = *seq;
85      pkt.ack_num = server_seq;
86      pkt.flags   = FLAG_FIN;
87      pkt.length  = 0;
88
89      //计算校验和使用的长度（照文件发送逻辑）
90      int pkt_len = sizeof(RDT_Packet) - MAX_DATA_SIZE + pkt.length;
91
92      //构造伪头部（与文件发送保持一致）
93      PseudoHeader ph;
94      ph.src_ip   = CLIENT_IP;
95      ph.dst_ip   = servaddr->sin_addr.s_addr;
96      ph.zero     = 0;
97      ph.protocol = 17; // UDP
98      ph.length   = htons(pkt_len);
99
100     //清空数据区（防止未初始化导致 checksum 不一致）
101     memset(pkt.truedata, 0, MAX_DATA_SIZE);
102
103     // 计算 checksum（与文件发送一致）
104     pkt.checksum = 0;
```

```
105     pkt.checksum = checksum_with_pseudo(&ph, &pkt, pkt.length);
106     //printf("pkt.length = %d\n", pkt_len);
107
108     printf("发送FIN, 校验和(%d)\n", pkt.checksum);
109
110     sendto(sock, (char*)&pkt, pkt_len, 0,
111         (struct sockaddr*)servaddr, servLen);
112
113
114
115     // ——— 等 ACK ———
116     while (1) {
117         printf("等待ACK\n");
118         Sleep(1200);
119         int recl=recvfrom(sock, (char*)&recv_pkt, sizeof(recv_pkt), 0,
120                 (struct sockaddr*)servaddr, &servLen);
121         if(recl<=0 || recl==SOCKET_ERROR){
122             printf("未收到ACK包, 重发FIN\n");
123             sendto(sock, (char*)&pkt, pkt_len, 0,
124                 (struct sockaddr*)servaddr, servLen);
125             continue;
126         }
127
128         printf("收到包\n");
129
130
131         PseudoHeader ph_recv;
132         ph_recv.src_ip = SERVER_IP;          // server 真正 IP
133         ph_recv.dst_ip = CLIENT_IP;          // client IP
134         ph_recv.zero   = 0;
135         ph_recv.protocol = 17;
136         ph_recv.length = htons(sizeof(RDT_Packet) - MAX_DATA_SIZE + recv_pkt.
                length);
137
138         uint16_t ck = recv_pkt.checksum;
139         recv_pkt.checksum = 0;
140         if (ck != checksum_with_pseudo(&ph_recv, &recv_pkt, recv_pkt.length))
                {
141             printf("校验和错误(%d), 应为(%d), 重发FIN\n",
                    checksum_with_pseudo(&ph_recv, &recv_pkt, recv_pkt.length),
                    ck);
142             sendto(sock, (char*)&pkt, pkt_len, 0,
143                 (struct sockaddr*)servaddr, servLen);
144             continue;
145         }
146
147
148         if ((recv_pkt.flags & FLAG_ACK) &&
```

```
149             recv_pkt.ack_num == *seq + 1)
150             break;
151     }
152
153     // ——— 等 server FIN ———
154     while (1) {//事实上在现在的框架里，由于server->client不会丢包，所以这一部
            分不会重传
155         printf("等待server FIN\n");
156         //Sleep(1200);
157         int recl=recvfrom(sock, (char*)&recv_pkt, sizeof(recv_pkt), 0,
158                 (struct sockaddr*)servaddr, &servLen);
159         if(recl<=0 || recl==SOCKET_ERROR){
160             printf("未收到server FIN包，重发FIN\n");
161             // ——— FIN ———
162             pkt.seq_num = *seq;
163             pkt.ack_num = server_seq;
164             pkt.flags   = FLAG_FIN;
165             pkt.length  = 0;
166             int pkt_len = sizeof(RDT_Packet) - MAX_DATA_SIZE + pkt.length;
167             PseudoHeader ph;
168             ph.src_ip   = CLIENT_IP;
169             ph.dst_ip   = servaddr->sin_addr.s_addr;
170             ph.zero     = 0;
171             ph.protocol = 17; // UDP
172             ph.length   = htons(pkt_len);
173             memset(pkt.truedata, 0, MAX_DATA_SIZE);
174             pkt.checksum = 0;
175             pkt.checksum = checksum_with_pseudo(&ph, &pkt, pkt.length);
176
177             sendto(sock, (char*)&pkt, pkt_len, 0,
178                 (struct sockaddr*)servaddr, servLen);
179             continue;
180         }
181
182
183
184         PseudoHeader ph_recv;
185         ph_recv.src_ip = SERVER_IP;          // server 真正 IP
186         ph_recv.dst_ip = CLIENT_IP;          // client IP
187         ph_recv.zero   = 0;
188         ph_recv.protocol = 17;
189         ph_recv.length = htons(sizeof(RDT_Packet) - MAX_DATA_SIZE + recv_pkt.
                length);
190
191         uint16_t ck = recv_pkt.checksum;
192         recv_pkt.checksum = 0;
193         if (ck != checksum_with_pseudo(&ph_recv, &recv_pkt, recv_pkt.length))
                {
```

```
194                 // ——— FIN ———
195             pkt.seq_num = *seq;
196             pkt.ack_num = server_seq;
197             pkt.flags   = FLAG_FIN;
198             pkt.length  = 0;
199             int pkt_len = sizeof(RDT_Packet) - MAX_DATA_SIZE + pkt.length;
200             PseudoHeader ph;
201             ph.src_ip   = CLIENT_IP;
202             ph.dst_ip   = servaddr->sin_addr.s_addr;
203             ph.zero     = 0;
204             ph.protocol = 17; // UDP
205             ph.length   = htons(pkt_len);
206             memset(pkt.truedata, 0, MAX_DATA_SIZE);
207             pkt.checksum = 0;
208             pkt.checksum = checksum_with_pseudo(&ph, &pkt, pkt.length);
209
210             sendto(sock, (char*)&pkt, pkt_len, 0,
211                 (struct sockaddr*)servaddr, servLen);
212             printf("校验和错误(%d), 应为(%d), 重发FIN\n",
                    checksum_with_pseudo(&ph_recv, &recv_pkt, recv_pkt.length),
                    ck);
213             continue;
214         }
215
216         printf("收到包, 继续发送最后一次挥手\n");
217
218         if (recv_pkt.flags & FLAG_FIN) {//printf("???");
219             while(1){
220                 pkt.seq_num = *seq + 1;
221                 pkt.ack_num = recv_pkt.seq_num + 1;
222                 pkt.flags = FLAG_ACK;
223                 pkt.length = 0;
224
225                 // 构造新的伪首部
226                 PseudoHeader ph;
227                 ph.src_ip   = CLIENT_IP;
228                 ph.dst_ip   = SERVER_IP;
229                 ph.zero     = 0;
230                 ph.protocol = 17; // UDP
231                 ph.length   = htons(pkt_len);
232
233                 pkt.checksum = 0;
234                 pkt.checksum = checksum_with_pseudo(&ph, &pkt, pkt.length);
235
236                 sendto(sock, (char*)&pkt, sizeof(pkt), 0,
237                     (struct sockaddr*)servaddr, servLen);
238                 Sleep(1200);
239
```

```
240                 memset(&recv_pkt, 0, sizeof(recv_pkt));
241                 int recl=recvfrom(sock, (char*)&recv_pkt, sizeof(recv_pkt),
                        0,
242                         (struct sockaddr*)servaddr, &servLen);
243                 if(recl<=0 || recl==SOCKET_ERROR){//没收到, 可以断开
244                     break;
245                 }//否则重发ACK
246             }
247
248             //printf("发送包信息: seq_num=%u, ack_num=%u, flags=%u, length=%u
                    , checksum=%u\n", pkt.seq_num, pkt.ack_num, pkt.flags, pkt.
                    length, pkt.checksum);
249             break;
250         }
251     }
252     printf("连接已关闭,5000ms后回退\n");
253     Sleep(5000);
254 }
```

如此，在额外增添部分细节后，我们的实验代码就完全完成了。

五、 性能分析

(一) 实验数据

笔者利用给定的测试用文件，进行了一定的性能分析。下面是控制 WINDOW_SIZE 变量的实验，为了尽可能减少传输以外的性能开销对实验的影响因素，传输速度和吞吐率结果为 server 和 client 显示数值的算术平均数。

(环境：1.jpg(1857353bytes)，0ms 延迟，丢包率 0)

WINDOW_SIZE	传输时间 (s)	传输速度（KB/s）
3	6.69	271.21
5	6.20	292.36
7	6.62	273.78
9	6.42	282.44

下面是控制变量丢包率的传输速率。由于丢包率高时传输过慢，所以实验使用了自己写的大小适中的 1.txt 作为传输文件。

(环境：1.txt(28842bytes)，0ms 延迟，WINDOW_SIZE=5)

丢包率	传输时间 (s)	传输速度（KB/s）
0	2.50	11.27
5%	4.30	6.55
10%	5.94	4.74
15%*	9.70	2.90

* 备注：由于助教给的实验环境是按照“每隔几个包丢一个”来取整的，实际上这里的丢包率是 16.67

下面则是控制延时的实验数据：

（环境：1.txt(28842bytes)，丢包率 0，WINDOW_SIZE=5）

延时 (ms)	传输时间 (s)	传输速度（KB/s）
0	2.45	11.48
10	4.22	6.68
20	4.25	6.63
30	5.12	5.50

（二） 实验结果分析

结合窗口大小变化的实验数据，可以从滑动窗口机制与协议实现开销的角度对结果进行分析。在 0 ms 延迟、无丢包的理想网络环境下，传输性能主要受窗口规模对链路利用率和控制开销的影响。

当 WINDOW_SIZE = 3 时，由于窗口较小，发送端需要频繁等待 ACK 返回才能继续发送新数据，窗口利用率受限，导致吞吐率较低。窗口增大到 WINDOW_SIZE = 5 后，传输时间明显缩短，传输速率达到最高值，说明该窗口规模能够在减少 ACK 等待的同时，又不会引入过多窗口管理和重传开销，发送端与接收端的协同效率较高。继续增大窗口到 7 和 9 后，传输性能未进一步提升，反而出现一定波动，表明在无延迟、无丢包环境下，窗口过大带来的额外维护开销逐渐抵消了其潜在收益。因此，本实验条件下存在一个较优的窗口规模区间，且 WINDOW_SIZE = 5 表现最佳。

进一步的实验表明，丢包率和网络延时的增加都会显著降低传输性能，其变化趋势与可靠数据传输和拥塞控制理论一致。随着丢包率上升，发送端需要频繁进行超时重传或快速重传，同时 Reno 拥塞控制会不断缩小拥塞窗口 \verb|cwnd|，使发送速率显著下降，整体吞吐率急剧降低。相比之下，延时的增加主要通过增大 RTT 限制窗口推进速度，对性能的影响相对平缓，但同样会导致传输时间上升、传输速率下降。

综合来看，丢包率对传输性能的影响明显强于单纯的延时增加。这不仅体现了重传带来的直接开销，也验证了 Reno 算法中将丢包视为拥塞信号的设计思想，以及滑动窗口和拥塞控制机制对网络条件变化的高度敏感性。

六、 总结

本实验基于 UDP 实现了一套面向连接的可靠数据传输机制，并在此基础上引入了滑动窗口、选择确认、超时重传以及 Reno 拥塞控制算法，对其在不同网络条件下的性能表现进行了系统分析。实验结果表明，该实现能够在不可靠的 UDP 服务之上正确完成文件传输，并具备基本的差错检测、丢包恢复和流量调节能力。

本次实验加深了笔者对网络传输协议及其算法的理解，也使我对计算机网络课程有了更深刻的认识。