南　　开　　大　　学

计　算　机　学　院

计算机网络第二次实验报告

设计可靠传输协议并编程实现

查科言

年级：2023 级

专业：计算机科学与技术

指导教师：徐敬东

2025 年 12 月 27 日

目录

一、 实验目的

(一) 功能要求

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：

- 连接管理：包括建立连接，关闭连接，异常处理
- 差错检测：使用校验和进行差错检测
- 确认重传：支持流水线方式，采用选择确认
- 流量控制：发送窗口和接受窗口使用固定的相同的大小窗口
- 拥塞控制：实现 RENO 算法

(二) 实验要求

- 实现单向数据传输，控制信息需要实现双向交互
- 给出详细的协议设计说明
- 给出详细的实现方法说明
- 利用 C 或 C++ 语言，使用最基本的 Socket 函数进行程序编写，不允许使用 CSocket 等封装后的类
- 在规定的测试环境中，完成测试文件的传输，显示传输时间和平均吞吐率，并观察不同发送窗口和接受窗口对传输性能的影响，以及不同丢包率对传输性能的影响
- 编写的程序应结构清晰，具有较好可读性，提交源码，可执行文件，和实验报告

二、 Github 仓库

本次实验的完整代码、可执行文件及本实验报告已上传至个人 GitHub 仓库，可通过以下链接访问，便于查阅与复用：

https://github.com/Ke-yyan/Computer-Internet-/tree/main/Lab2

三、 总体设计

本实验在 UDP 数据报套接字之上自行设计并实现了一个面向连接的可靠传输协议，后文称为 **MiniTCP**。

(一) 设计目标

- 在 UDP 之上实现一个 TCP-like 的可靠传输协议（后文简称 **MiniTCP**）。
- 满足实验要求的五项功能：连接管理、差错控制、确认重传、流量控制、Reno 拥塞控制。

(二) 协议架构概述

- 整体采用客户端/服务器模式：send 端作为主动打开方，recv 端作为被动打开方。
- 数据平面：基于流水线滑动窗口发送 DATA 分组。
- 控制平面：利用 ACK 报文携带累计确认与 SACK 区间，同时承担流量控制与拥塞控制信息传递。

(三) 整体架构

1. 进程角色与调用关系

系统采用典型的 Client/Server 结构：

- **发送端进程（Client）**：主动发起连接，读取本地输入文件，将数据切分成若干 MiniTCP 报文，按照滑动窗口和 Reno 拥塞控制进行发送，同时处理接收端返回的 ACK+SACK 报文，完成重传与统计。对应函数 `runSender`。
- **接收端进程（Server）**：在指定 UDP 端口上监听，执行三次握手建立连接后，按序写入收到的数据到输出文件，对乱序分组进行缓存，并构造 ACK+SACK 反馈给发送端；当收到 FIN 后按照四次挥手流程关闭连接。对应函数 `runReceiver`。

主函数 `main` 仅负责解析命令行参数并选择运行模式：

- `rudp.exe recv <port> <output_file>[window_size]`：以接收端模式启动,调用 `runReceiver`;
- `rudp.exe send <server_ip> <port> <input_file>[delay\_ms] [loss\_percent]`：以发送端模式启动，调用 `runSender`。

在程序入口处调用 `WSAStartup` 与 `WSACleanup` 完成 Winsock 初始化与清理。

2. 模块划分

为了提高代码的可读性与可维护性，MiniTCP 的实现按功能划分为如下几个源文件：

- `rudp.h`：**公共头文件**，包含协议参数常量（如最大报文负载、接收窗口大小、超时重传时间等）、分组头部 `PacketHeader` 与 SACK 区间 `SackBlock` 的结构定义，以及公共工具函数和 `runSender` / `runReceiver` 的声明。
- `rudp_common.cpp`：**与 TCP 具体算法无关的通用工具模块**，包括：
  - （1）`setRecvTimeout` 封装 `setsockopt(SO_RCVTIMEO)`;
  - （2）16 位互联网校验和函数 `checksum16`;
  - （3）统一的 `sendPacket` / `recvPacket` 接口，用于打包头部、计算校验和、调用 UDP 的 `sendto` / `recvfrom`，并在接收端对校验和进行验证。
- `rudp_sender.cpp`：**发送端协议逻辑**。实现客户端侧三次握手、滑动窗口发送、选择确认解析、超时重传、快重传、快恢复、Reno 拥塞控制以及 RTT/吞吐量/丢包率等统计功能。
- `rudp_receiver.cpp`：**接收端协议逻辑**。实现服务端侧三次握手、乱序缓存与 SACK 生成、有序数据写入文件、以及被动四次挥手机制。

我们这种模块化设计使得协议头部与工具函数可以被双端重用，而发送端与接收端只需关注自身逻辑，符合总体设计原则。

（四） 报文格式与关键字段

1. 固定头部

MiniTCP 自定义了一个 16 字节的固定头部 PacketHeader，各字段含义见表，和 TCP 协议定义的一致。1。

字段名	类型	说明
seq	uint32_t	序号，对 DATA 报文有效。发送端在切分文件时从 1 开始为每个数据报文分配递增序号，用于在接收端恢复顺序与检测丢失。
ack	uint32_t	累计确认号，对 ACK 报文有效，含义与 TCP 相同：表示“已经连续正确收到的最大序号”。
len	uint16_t	本报文负载长度（字节数），由 sendPacket 在发送前自动填充。
wnd	uint16_t	接收窗口通告，用于简单流量控制，表示接收端当前仍可接受的最大缓存分组数。
checksum	uint16_t	16 位互联网校验和，覆盖头部与数据，用于差错检测。
flags	uint8_t	标志位：FLAG_SYN、FLAG_ACK、FLAG_FIN、FLAG_DATA，支持三次握手、四次挥手以及数据报文的区分。
reserved	uint8_t	保留字段，对齐用，固定置 0。

表 1: MiniTCP 报文头部格式

2. SACK 载荷格式

为了实现选择确认，MiniTCP 在 ACK 报文中携带了若干乱序接收区间。

接收端维护buffer[seq] 形式的乱序缓存，并将其中序号大于 cumulativeAck 的元素合并成最多 MAX_SACK_BLOCKS 个连续区间 [start, end]，打包成如下载荷格式：

- 首部：uint16_t blkCount，表示后续 SACK 区间的个数；
- 随后依次为 blkCount 个 SackBlock 结构体，每个包含 start 和 end 两个 32 位无符号整数，表示包含端点的序号区间。

发送端在处理 ACK 时首先根据累计确认号更新已确认分组，然后解析 SACK 区间，对每个 [start, end] 内的序号提前标记为已确认，从而减少不必要的重传。

（五） 连接管理流程

1. 三次握手

连接建立阶段完全在应用层模拟 TCP 三次握手，总体流程为：

1. **客户端：**调用 senderHandshake 构造并发送 SYN 报文（flags=FLAG_SYN），其中 seq=0 作为客户端初始序号。**进入等待“SYN+ACK”状态**

2. **服务器：**在 `receiverHandshake` 中阻塞等待，收到 SYN 后回复一个 SYN+ACK 报文，`seq=100`（服务器初始序号）、`ack=syn.seq+1`，并通告自身接收窗口 `wnd`。**进入“等待 ACK”状态**
3. **客户端：**收到 SYN+ACK 且校验通过后,发送最终的 ACK 报文,`seq=syn.seq+1`,`ack=resp.seq+1`，至此双方进入已连接状态。

容错机制：所有握手报文均使用 `HANDSHAKE_TIMEOUT_MS` 作为超时重传周期，超时后最多重试 `MAX_TRY` 次，以保证在丢包情况下仍能完成连接建立或及时失败返回。

2. 四次挥手

连接关闭采用 TCP 风格的四次挥手：

- **客户端：**在所有数据确认后调用 `senderFourWayClose` 主动发送 FIN 报文，并等待对端 ACK；随后阻塞等待服务器 FIN；收到后再次发送 ACK 完成关闭。
- **服务器：**在数据接收循环中收到 FIN 后，先发送一个 ACK 作为对客户端 FIN 的确认，然后再发送自己的 FIN，并等待客户端的最后一个 ACK；若超时则重发 FIN。

握手与挥手阶段均使用统一的超时与重试策略，实现简化但完整的连接管理流程。

(六) 差错控制

我们设计的 MiniTCP 采用**“检错 + 重传”**的差错控制思路：

- **检错：**用 16 位互联网校验和检测比特错误
- **重传：**超时重传 + 选择重传，恢复丢失、损坏的报文

1. 16 位互联网校验和

- **chechsum16：**在公共模块 `rudp_common.cpp` 中实现了标准的 16 位互联网校验和算法 `checksum16`。该函数以 16bit 为单位对报文（头部 + 数据）进行一补码加法叠加，出现进位时回卷到低 16 位，最后对和取反得到校验和。
- **发送端：**`sendPacket` 先将 `PacketHeader` 与可选负载拷贝到一块连续缓冲区，暂时将头部中的 `checksum` 置 0，然后调用 `checksum16` 计算整个缓冲区的校验和，并回填到头部相应字段，最后调用 UDP 的 `sendto` 发送。
- **接受端：**`recvPacket` 先从 UDP 套接字中读入一个最大长度缓冲区，检查总长度至少包含一个完整头部，然后将报文中的原始 `checksum` 缓存下来，临时把缓冲区头部的 `checksum` 位置清零，再次调用 `checksum16` 对收到的数据重新计算校验和并比较。若两者不相等，则认为该报文在传输过程中出现了位错误，打印 `[recvPacket] checksum error` 日志并直接丢弃，不向上层协议交付该包。

这种处理方式与 TCP 一致：差错检测在接收端完成，错误分组被视为“丢失”，由发送端的 ARQ 机制负责后续恢复。

（七） 确认与重传机制

1. 序号与累计确认

数据阶段采用与 TCP 类似的序号与累计确认机制。

- **发送端：**发送端在读取输入文件后，将其切分为多个不超过 `MAX_PAYLOAD` 字节的数据块，每个数据块对应一个发送槽 `SendSlot`，并从 1 开始为其分配递增的分组序号 `hdr.seq`，握手报文的序号空间与数据分组分离。
- **接收端：**接收端维护一个期望的下一个有序序号 `expectedSeq`。收到 DATA 报文后：
  1. 若 `seq >= expectedSeq`，则将其插入乱序缓存 `buffer[seq]`；
  2. 随后不断检查`buffer` 中是否存在 `expectedSeq` 对应的数据块，若存在则写文件，删除缓存，`expectedSeq++`
  3. 构造 ACK 报文，`cumulativeAck = expectedSeq - 1`（表示“已连续收到的最大序号”）。

这样可以保证应用层看到的数据始终是严格有序的。

2. 选择确认（SACK）

为了降低乱序情况下的重传开销，MiniTCP 在 ACK 报文中携带选择确认信息。

- **接收端**：在 `sendAckWithSack` 中遍历乱序缓存，将大于 `cumulativeAck` 的序号聚合成最多 `MAX_SACK_BLOCKS` 个区间，打包进 ACK 载荷；
- **发送端**：处理 ACK 时，先通过 `cumulativeAck` 标记已确认分组，再解析 SACK 区间，提前标记对应序号为“已接收”，避免不必要的重传。

3. 超时与重传策略

- **超时检测**:发送端遍历滑动窗口内的 `SendSlot`,若分组“已发送未确认”且 `now - lastSendTime > TIMEOUT_MS`（实验设为 500ms），则重传该分组；
- **拥塞控制**：超时重传时，按 Reno 规则调整：

$$\text{ssthresh} = \max\{2,\ \lfloor \text{cwnd}/2 \rfloor\}, \quad \text{cwnd} = 1$$

（注：本实验 RTO 为固定值，RTT 仅用于统计，未自适应调整 RTO）。

这样可以在丢包发生时迅速降低发送速率，配合后续 ACK 驱动的窗口增长，自适应网络状况。

（八） 数据传输与窗口控制总体思路

1. 发送端滑动窗口

发送端在读入输入文件后，按照 `MAX_PAYLOAD` 字节对数据进行分片，并为每个片段构造一个 `SendSlot` 结构，内部包含该分组的头部、数据缓冲区以及首次发送时间、最近发送时间、发送/确认状态等信息。

数据序号从 1 开始递增，与握手报文分开。

发送窗口大小为：

$$W_{\text{send}} = \min(\text{cwnd}, \text{peerWnd}, \text{待发送分组数})$$

其中 `cwnd` 为拥塞窗口，`peerWnd` 为接收端通告窗口；发送端在窗口范围内依次发送新报文，并为每个已发送分组维护 `lastSendTime`（用于超时重传）以及首发时间（用于 RTT 统计）。当收到 ACK/SACK 后，发送端对 `SendSlot` 进行确认标记，并通过移动 `base` 实现滑动窗口前移。

2. 接收端乱序缓存与顺序交付

接收端针对每个到达的 DATA 分组执行以下逻辑：

1. 缓存乱序分组：`seq >= expectedSeq` 且未缓存的报文，插入 `buffer`；
2. 顺序交付：循环检查 `buffer` 中是否存在 `expectedSeq`，若存在则写文件、删除缓存、`expectedSeq++`；
3. 反馈 ACK+SACK：构造包含累计确认号 `cumulativeAck=expectedSeq-1` 以及 SACK 区间（乱序缓存的连续段）的 ACK 报文，发送给发送端。

通过这种设计，接收端既能保证最终文件内容的有序性，又能充分利用 SACK 告知发送端已收到的乱序分组，减少无谓重传开销。

3. 流量控制与拥塞控制协同

- **流量控制**：接收端在 ACK 中通过 `wnd` 字段通告当前可用窗口，实现方式为“接收窗口上限 − 当前乱序缓存占用”（例如 `wnd = recvWindow - buffer.size()`，并保证最小为 1）。发送端在计算有效发送窗口时取

  $$W_{\text{send}} = \min(\text{cwnd},\ \text{peerWnd},\ \text{待发送分组数}),$$

  防止发送速率超过接收端缓存能力。

- **拥塞控制（Reno 基本增长）**：当本次 ACK/SACK 带来新的确认（`anyNewAck` 为真）时，发送端根据 `cwnd` 与 `ssthresh` 关系决定处于慢启动或拥塞避免阶段：`cwnd < ssthresh` 时采用较快增长（每轮确认 `cwnd += 1`），否则进入拥塞避免（每轮确认 `cwnd += 1/cwnd`），从而在吞吐量与稳定性之间折中。

- **快重传（Fast Retransmit）**：发送端维护 `lastAckSeq` 与 `dupAckCount`，当累计确认号 `ackHdr.ack` 连续重复到 3 次及以上（且窗口内仍存在未确认分组）时，认为窗口左端的分组（`base` 对应的最早未确认分组）丢失，**不等待超时**，立即重传该分组，从而更快填补“缺口”。

- **快恢复（Fast Recovery）**：触发快重传后进入快恢复状态（`inFastRecovery`），将 `ssthresh` 设置为当前 `cwnd` 的一半（下限为 2），并将 `cwnd` 暂时提升为 `ssthresh + 3` 以反映已收到的 3 个重复 ACK，保持管道中仍有一定在途数据。在快恢复期间，每收到一个额外重复 ACK，`cwnd` 线性加 1，并允许继续发送新的分组；当累计 ACK 前进并超过进入快恢复时记录的恢复边界（`recoverSeq`，即当时已发送的最高序号）后，退出快恢复，并将 `cwnd` 回落为 `ssthresh`。

- **超时重传与回退**：若某分组在超时阈值内仍未被确认，则触发超时重传；发送端将其视为更强的拥塞信号，更新 `ssthresh = max(2, cwnd/2)`，并将 `cwnd` 调整为 `ssthresh`（窗口减半），从而降低发送速率，缓解潜在拥塞。

上述机制共同保证：在低丢包/低时延环境下，窗口能够快速增长并充分利用带宽；在出现丢包时，SACK 用于减少无谓重传，快重传/快恢复用于更快定位并修复窗口缺口，而超时回退则在严重丢包或拥塞时主动收敛发送速率，避免拥塞进一步恶化。整体而言，我们设计的 MiniTCP 在用户态复现了 TCP 的主要控制逻辑，并通过模块化的数据结构与报文格式，使协议行为易于理解与扩展，为后续性能测试与实验分析提供了良好的基础。

四、 程序实现与模块划分

(一) 整体结构

- `rudp.h`：协议头部、常量、接口函数声明。
- `rudp_common.cpp`：socket 工具函数、校验和、统一收发接口。
- `rudp_sender.cpp`：发送端逻辑（三次握手、滑动窗口、Reno、快重传、快恢复、统计）。
- `rudp_receiver.cpp`：接收端逻辑（三次握手、缓存 + SACK、四次挥手）。
- `main.cpp`：命令行解析，选择 `send`/`recv` 模式。

(二) 主要数据结构

- 分组头部 `PacketHeader`。
- SACK 区间 `SackBlock`。
- 发送端槽位 `SendSlot`（包含状态、时间戳等）。
- 接收端乱序缓冲区 `std::map<uint32_t, std::vector<char>>`。

(三) 关键函数流程

- `senderHandshake()` / `receiverHandshake()`：三次握手。
- `runSender()`：主循环中“发送新分组—处理 ACK+SACK—超时重传-快重传-快恢复”
- `runReceiver()`：接收 DATA、写入文件、构造 ACK+SACK
- 四次挥手机制相关函数：发送端/接收端关闭流程。

五、 rudp.h

这个文件中，我们基于 UDP 实现了可靠传输协议，主要作用是定义协议规则、公共数据结构、工具函数和核心接口，为后续的发送端（Sender）和接收端（Receiver）实现提供统一规范。

(一) 协议核心参数

```
1
2 inline constexpr int MAX_PAYLOAD          = 1000;
3 inline constexpr int RECV_WINDOW          = 32;
4 inline constexpr int TIMEOUT_MS           = 500;
```

```
inline constexpr int HANDSHAKE_TIMEOUT_MS  = 1000;
inline constexpr int MAX_SACK_BLOCKS       = 4;
```

- **MAX_PAYLOAD:** 每个数据分组的最大负载（1000 字节），一次只能发 1000 字节的实际数据，超过了需要拆分成多个分组
- **RECV_WINDOW:** 接收窗口的大小，接收端最多能缓存 32 个未处理的分组，发送端不能超过这个值
- **TIMEOUT_MS:** 数据分组超过时间, 发送端发送完一个分组后, 500ms 内没有收到 ACK 就重发
- **HANDSHAKE_TIMEOUT_MS:** 握手/挥手阶段超时（1000 毫秒）, 建立连接（握手）或断开连接（挥手）时，超时时间设置为 1s
- **MAX_SACK_BLOCKS:** 一次 ACK 最多携带 4 个 SACK 区间, SACK 是“选择性确认”，用于告诉发送端“哪些分组收到了”，最多同时确认 4 个不连续的区间

我们通过常量统一协议行为，后续要调整性能（比如增大负载、延长超时），直接改这里就行，无需改代码。

(二) 协议控制结构

该模块定义了网络传输的核心数据结构，确保发送端和接收端对数据格式达成共识，实现“可互认”的通信。

1. 标志位枚举

我们用 0x01、0x02、0x04、0x08 这种“2 的幂”值，是为了通过位运算判断标志（比如 `if (flags & FLAG_DATA)`就能判断是不是数据分组）。

```
enum PacketFlags : uint8_t
{
    FLAG_SYN  = 0x01,
    FLAG_ACK  = 0x02,
    FLAG_FIN  = 0x04,
    FLAG_DATA = 0x08
};
```

- **FLAG_ACK:** 确认标志位，当 ACK=0 时，ack 有效，当 ACK=1 时，ack 无效。只有握手 1 的 ACK=0，其余所有报文段 ACK=1
- **FLAG_SYN:** 同步位，只有握手 1，握手 2 的 SYN 为 1，其余 TCP 报文段都是 SYN=0
- **FLAG_FIN:** 终止位，只有挥手 1，挥手 3 的 FIN 为 1，其余 TCP 报文段的 FIN 都是 0
- **FLAG_DATA:** 数据标志，用于传输实际文件数据

2. 分组头部

头部总大小：4+4+2+2+2+1+1 = 16 字节（固定长度）；

核心作用是当接收端拿到分组后，先读头部，就知道“这是什么类型的分组”“数据有多长”“要不要确认”“数据有没有坏”。

我们参考这个 TCP 协议首部来实现

```
// 分组头部（16 字节）
#pragma pack(push, 1)
struct PacketHeader
{
    uint32_t seq;       // 序号（对 DATA 有效）
    uint32_t ack;       // 确认号（对 ACK 有效：累计确认）
    uint16_t len;       // 负载长度
    uint16_t wnd;       // 接收窗口通告
    uint16_t checksum;  // 16 位校验和（头 + 数据）
    uint8_t  flags;     // 标志位
    uint8_t  reserved;  // 对齐用，置 0
};
#pragma pack(pop)
```

- **seq:** 序号（只对 DATA 分组有效）：标识当前数据块的位置（比如第 1 个数据分组 seq=1，第 2 个 seq=2）
- **ack:** 确认号（只对 ACK 分组有效）：累计确认，比如 ack=5 表示“我已经收到 seq<=5 的所有分组”
- **len:** 负载长度:当前分组的 payload 实际有多少字节(最大不超过 MAX_PAYLOAD=1000)
- **wnd:** 接收窗口通告：告诉对方“我现在还能接收多少个分组”（流量控制核心）
- **checksum:**16 位校验和：对“头部 + 负载”计算的校验值，用于判断数据传输中是否损坏
- **flags:** 标志位：上面定义的 PacketFlags（比如是 DATA 还是 ACK）
- **reserved:** 保留字段（占位用，必须置 0）：为了让头部总长度是 16 字节（对齐，传输更高效）

3. SACK 区间

这是选择性确认的辅助结构。

```
struct SackBlock
{
    uint32_t start;  // 区间起始序号
    uint32_t end;    // 区间结束序号（包含端点）
};
```

用途是解决“累计确认的缺陷”。

比如接收端收到了 seq=1、3、4、5 的分组，累计确认只能说 ack=2（表示收到 seq<=2），但发送端不知道 3-5 已经收到。

用 SACK 可以告诉发送端 `SackBlock{3,5}`，表示“3-5 的分组我已经收到了，你不用重发”，提高传输效率。

(三) 公共工具函数

我们定义了一些工具函数,，封装了重复的逻辑，比如“设置接收超时”“计算校验和”“发送分组”“接收分组”，避免发送端和接收端各自写一遍。

```
void setRecvTimeout(SOCKET s, int ms);
void printLastError(const char* where);

// 16 位互联网校验和
uint16_t checksum16(const char* data, size_t len);

// 发送一个分组（负责填充 hdr.len / hdr.checksum）
bool sendPacket(
    SOCKET s,
    const sockaddr_in& addr,
    PacketHeader hdr,
    const char* payload,
    uint16_t payloadLen);

// 接收一个分组并校验，成功返回 true，失败/超时返回 false
bool recvPacket(
    SOCKET s,
    PacketHeader& hdr,
    std::vector<char>& payload,
    sockaddr_in& from);
```

这些函数将在 rudp_common.cpp 中实现，此处只简要介绍功能

- **setRecvTimeout:** 设置 socket 的接收超时时间。
- **printLastError:** 打印 Windows 网络错误信息。
- **checksum16:** 计算 16 位互联网校验和。
- **sendPacket:** 封装分组发送的完整逻辑。
- **recvPacket:** 封装分组接收和校验的完整逻辑。

(四) 核心业务接口

这两个是“对外暴露的接口”，后续会有.cpp 文件实现这两个函数 (rudp_sender.cpp,rudp_receiver.cpp)

```
void runSender(const std::string& ip, uint16_t port, const std::string&
    inputFile);
void runReceiver(uint16_t port, const std::string& outputFile);
```

六、 rudp_common.cpp

该文件实现了 rudp.h 中声明的公共工具函数，是 RUDP 协议的核心通用逻辑实现，负责网络配置、错误处理、数据校验、分组发送 / 接收等基础功能。

(一) 设置 Socket 接收超时

```
void setRecvTimeout(SOCKET s, int ms)
{
    int opt = ms;
    setsockopt(s, SOL_SOCKET, SO_RCVTIMEO,
               reinterpret_cast<char*>(&opt), sizeof(opt));
}
```

这个函数的功能是：设置 Socket 的接收超时时间，避免 recvfrom 函数无限阻塞（比如没收到数据时一直等）。

实现逻辑如下：

步骤 1： 定义超时时间变量 `opt`，赋值为传入的毫秒数 `ms`；

步骤 2： 调用 Windows Socket 核心函数 `setsockopt` 配置 Socket 选项：

- `s`：待配置的 Socket 句柄；
- `SOL_SOCKET`：选项层级（通用 Socket 选项，非特定协议选项）；
- `SO_RCVTIMEO`：选项名称（指定为“接收超时”）；
- `reinterpret_cast<char*>(&opt)`：将超时时间变量转换为字符指针（`setsockopt` 要求的参数类型）；
- `sizeof(opt)`：超时时间变量的字节长度。

超时时间生效后，`recvfrom` 函数若超过 `ms` 毫秒未收到数据，会返回 `SOCKET_ERROR`，错误码为 `WSAETIMEDOUT`（超时）；

(二) 打印网络错误信息

```
void printLastError(const char* where)
{
    std::cerr << "[" << where << "] WSA error: "
              << WSAGetLastError() << std::endl;
}
```

这个函数的功能是：捕获并打印 Windows 网络操作的错误详情，为调试提供依据（如 `sendto` 失败时，明确错误原因）。

实现逻辑如下：

步骤 1： 调用 `WSAGetLastError()` 函数，获取上一次网络操作的错误码（Windows 网络编程标准错误获取接口）；

步骤 2： 以格式 `[错误位置] WSA error: 错误码`输出错误信息：

- `where`：传入的错误发生位置标识（如“sendto”“recvfrom”）；
- `std::cerr`：标准错误输出流（区别于普通日志输出）。

所有网络操作（sendto、recvfrom、bind 等）失败时调用，快速定位问题。

若 `sendto` 函数因网络不可达失败，输出为：

```
[sendto] WSA error: 10065
```

其中错误码 10065 对应 Windows 网络错误“无路由到主机”。

(三) 计算 16 位互联网校验和

这个函数的功能是计算数据的 16 位互联网校验和（IP/TCP/UDP 协议均使用此算法），用于验证数据传输是否损坏。

算法思路为： 数据以 16bit 为一组，进行二进制加法，最高产生的进位需要回卷，将最终的加法结果逐位取反，得到 16bit 校验和。

核心原理是冗余校验：发送端计算校验和并随数据发送，接收端重新计算数据和校验和，若加法结果不是全为 1，则说明数据传输中出错。

```
uint16_t checksum16(const char* data, size_t len)
{
    uint32_t sum = 0;
    size_t i = 0;
    // 步骤1：处理偶数字节（每次取2字节，组成16位整数）
    while (i + 1 < len)
    {
        uint16_t word =
            (static_cast<uint8_t>(data[i]) << 8) |
             static_cast<uint8_t>(data[i+1]);
        sum += word;
        if (sum & 0x10000)                  // 产生进位则回卷
            sum = (sum & 0xFFFF) + 1;
        i += 2;
    }
    // 步骤2：处理奇数字节（若总长度为奇数，补0凑16位）
    if (i < len)
    {
        uint16_t word = static_cast<uint8_t>(data[i]) << 8;
        sum += word;
        if (sum & 0x10000)
            sum = (sum & 0xFFFF) + 1;
    }
     // 步骤3：取反得到校验和（16位）
    return static_cast<uint16_t>(~sum);
}
```

我们这个函数的实现步骤如下，分为奇数字节和偶数字节处理：

步骤 1： 初始化 32 位累加和 `sum=0`（用 32 位存储是为了避免中间溢出）；

步骤 2： 处理偶数字节（每次取 2 字节，组成 16 位整数）：

- 循环条件：`i + 1 < len`（确保每次能取到 2 字节）；
- 字节拼接：将第 `i` 字节作为高 8 位，第 `i+1` 字节作为低 8 位，组成 16 位整数：

$$\text{word} = (\text{data}[i] << 8) \mid \text{data}[i+1]$$

- 累加与进位回卷：将 `word` 加入 `sum`，若 `sum` 超过 16 位（`sum & 0x10000` 为真），则保留低 16 位并加 1（避免溢出）：

$$\text{sum} = (\text{sum} \& 0xFFFF) + 1$$

– 索引递增：`i += 2`（处理下一组 2 字节）。

步骤 3: 处理奇数字节（若总长度为奇数，最后 1 字节补 0 凑 16 位）：

– 字节拼接：将最后 1 字节作为高 8 位，低 8 位补 0：

$$\text{word} = \text{data}[i] << 8$$

– 重复累加与进位回卷逻辑。

步骤 4: 取反得到校验和：将最终 16 位累加和按位取反（ `sum`），转换为 `uint16_t` 类型返回。

关键说明：

- **在校验和计算时需保证“字节序一致性”：**实现默认数据为大端序（网络字节序），符合 TCP/IP 协议规范；
- **取反操作的目的：**接收端校验时，可将“数据 + 校验和”重新累加，若结果为全 1（`0xFFFF`），则数据完整，简化校验逻辑。

（四） 封装并发送 RUDP 分组

这个函数的功能是封装 RUDP 分组的发送：自动填充头部字段、拼接头部和负载、计算校验和，最终通过 sendto 发送到目标地址。

我们可以实现调用者无需关心分组的底层封装细节（比如校验和计算、缓冲区拼接），只需传入“头部参数 + 负载数据”，即可完成发送。

```
bool sendPacket(
    SOCKET s,
    const sockaddr_in& addr,
    PacketHeader hdr,
    const char* payload,
    uint16_t payloadLen)
{
    hdr.len      = payloadLen;
    hdr.checksum = 0;

    const size_t totalLen = sizeof(PacketHeader) + payloadLen;
    std::vector<char> buffer(totalLen);

    std::memcpy(buffer.data(), &hdr, sizeof(PacketHeader));
    if (payloadLen > 0 && payload != nullptr)
        std::memcpy(buffer.data() + sizeof(PacketHeader),
                    payload, payloadLen);

    // 计算校验和
    hdr.checksum = checksum16(buffer.data(), buffer.size());
    std::memcpy(buffer.data(), &hdr, sizeof(PacketHeader));

    int ret = sendto(
        s,
```

```
25          buffer.data(),
26          static_cast<int>(buffer.size()),
27          0,
28          reinterpret_cast<const sockaddr*>(&addr),
29          sizeof(addr));
30
31      if (ret == SOCKET_ERROR)
32      {
33          printLastError("sendto");
34          return false;
35      }
36      return true;
37  }
```

我们实现这个函数的步骤如下：

步骤 1: **填充头部核心字段：**

- `hdr.len = payloadLen`：设置负载长度（实际发送的数据字节数）；
- `hdr.checksum = 0`：校验和先置 0（计算时需排除自身，避免干扰）。

步骤 2: **创建发送缓冲区：**

- 缓冲区总长度 = 头部长度 + 负载长度（`sizeof(PacketHeader) + payloadLen`）；
- 使用 `std::vector<char>` 作为缓冲区（动态扩容，避免栈溢出，自动管理内存）。

步骤 3: **拼接头部与负载到缓冲区：**

- 调用 `std::memcpy` 将头部拷贝到缓冲区起始位置；
- 若负载长度大于 0 且负载指针非空，将负载拷贝到头部之后的位置。

步骤 4: **计算并更新校验和：**

- 调用 `checksum16` 计算缓冲区（头部 + 负载）的校验和；
- 将计算结果赋值给头部的 `checksum` 字段，重新拷贝头部到缓冲区（覆盖原校验和 0）。

步骤 5: **调用 `sendto` 发送分组：**

- 核心参数：Socket 句柄 `s`、目标地址 `addr`（转换为通用 `sockaddr` 类型）、缓冲区数据、缓冲区长度；
- 若 `sendto` 返回 `SOCKET_ERROR`（发送失败），调用 `printLastError` 打印错误，返回 `false`；
- 发送成功则返回 `true`。

（五） 接收并解析 RUDP 分组

这个函数的功能是接收网络数据，解析 RUDP 分组的头部和负载，验证数据完整性，最终将结果返回给调用者

接收端循环调用该函数监听网络，获取合法的 RUDP 分组后，再根据头部的 flags 字段（如 FLAG_DATA、FLAG_ACK）处理后续逻辑（如重组文件、回复 ACK）。

```cpp
bool recvPacket(
    SOCKET s,
    PacketHeader& hdr,
    std::vector<char>& payload,
    sockaddr_in& from)
{
    char buffer[sizeof(PacketHeader) + MAX_PAYLOAD];
    int  fromLen = sizeof(from);

    int ret = recvfrom(
        s,
        buffer,
        sizeof(buffer),
        0,
        reinterpret_cast<sockaddr*>(&from),
        &fromLen);

    if (ret == SOCKET_ERROR)
    {
        int err = WSAGetLastError();
        if (err == WSAETIMEDOUT || err == WSAEWOULDBLOCK)
        {
            // 超时：由调用者决定是否重试
            return false;
        }
        printLastError("recvfrom");
        return false;
    }

    if (ret < static_cast<int>(sizeof(PacketHeader)))
    {
        std::cerr << "[recvPacket] packet too short\n";
        return false;
    }

    // 拷贝头部
    PacketHeader wireHdr{};
    std::memcpy(&wireHdr, buffer, sizeof(PacketHeader));
    uint16_t recvChecksum = wireHdr.checksum;

    // 重新计算校验和
    reinterpret_cast<PacketHeader*>(buffer)->checksum = 0;
    uint16_t calcChecksum = checksum16(buffer, ret);

    if (recvChecksum != calcChecksum)
    {
        std::cerr << "[recvPacket] checksum error\n";
```

```
48          return false;
49      }
50
51      hdr = wireHdr;
52      int payloadLen = ret - static_cast<int>(sizeof(PacketHeader));
53      payload.assign(buffer + sizeof(PacketHeader),
54                     buffer + sizeof(PacketHeader) + payloadLen);
55      return true;
56  }
```

我们实现这个函数的步骤如下：

步骤 1: **创建接收缓冲区：**

- 缓冲区大小 = 头部长度 + 最大负载长度（`sizeof(PacketHeader) + MAX_PAYLOAD`）；
- 采用固定大小数组（栈上分配），确保能容纳最大长度分组，避免溢出。

步骤 2: **调用 `recvfrom` 接收数据：**

- 核心参数：Socket 句柄 `s`、接收缓冲区、缓冲区大小、发送方地址 `from`（输出参数）；
- `fromLen`：传入地址结构体长度，传出实际接收的地址长度。

步骤 3: **处理接收结果：**

- 若返回 `SOCKET_ERROR`：
  - 错误码为 `WSAETIMEDOUT`（超时）或 `WSAEWOULDBLOCK`（非阻塞无数据）：返回 `false`（由调用者决定是否重试）；
  - 其他错误：调用 `printLastError` 打印详情，返回 `false`；
- 若接收长度 < 头部长度（`sizeof(PacketHeader)`）：分组不完整，输出错误并返回 `false`。

步骤 4: **校验和验证：**

- 提取网络传输的头部（`wireHdr`）和其携带的校验和（`recvChecksum`）；
- 将缓冲区中的校验和字段置 0（还原发送时的状态），重新计算校验和（`calcChecksum`）；
- 若 `recvChecksum != calcChecksum`：数据损坏，输出错误并返回 `false`。

步骤 5: **解析并输出结果：**

- 将解析后的头部 `wireHdr` 赋值给输出参数 `hdr`；
- 计算负载长度 = 总接收长度 - 头部长度；
- 提取缓冲区中头部后的负载数据，赋值给输出参数 `payload`（`vector::assign` 自动扩容）；
- 返回 `true` 表示接收成功。

（六） 整体总结

我们设计的 rudp_common.cpp 的核心作用是封装通用网络操作，让发送端和接收端聚焦于自身的核心逻辑（如发送端的序号管理、重传控制；接收端的窗口管理、文件重组），无需关注底层的校验和计算、分组拼接、错误处理等重复工作。

七、 rudp_sender.cpp

(一) 整体概述

rudp_sender.cpp 实现了 MiniTCP 协议的发送端逻辑，是整个系统中“主动打开连接、读取文件并可靠发送”的一侧。其主要功能包括：

1. 通过 UDP 套接字与接收端完成三次握手，建立“逻辑连接”；
2. 将输入文件按固定大小切分为多个数据分组，维护发送缓冲区；
3. 按滑动窗口 + Reno 拥塞控制规则发送数据，并根据 ACK + SACK 进行确认与窗口更新；
4. 对超时未确认分组进行重传，并在重传时触发拥塞控制；
5. 在所有数据传输完成后，通过四次挥手关闭连接；
6. 在整个过程中统计 RTT、重传次数、丢包率、吞吐量等指标，并在发送结束后输出。

(二) 核心数据结构：发送槽 SendSlot

我们首先定义一个发送端内部使用的数据机构 SendSlot。

```
struct SendSlot
{
    PacketHeader hdr{};
    std::vector<char> data;

    bool sent      = false;   // 是否发送过（任意一次）
    bool acked     = false;   // 是否已被确认
    bool firstSent = false;   // 是否已经记录 firstSendTime

    Clock::time_point firstSendTime;
    Clock::time_point lastSendTime;
};
```

我们的设计目的如下：

- 每一个 SendSlot 对应一个要发送的数据报文（DATA 分组）。
- **hdr** 保存该分组的协议头部（包含 seq、flags 等）。
- **data** 保存该分段的数据内容（从输入文件读出的一个片段）
- **sent** 表明是否曾发送该分组（包括首发和重传）。
- **acked** 标记是否已经被接收端确认（通过累计 ACK 或 SACK）。
- **firstSent、firstSendTime** 用于记录第一次发送时间，方便计算 RTT。
- **lastSendTime** 用于和当前时间比较，判断是否超时从而触发重传。

我们将所有要发送的分组被组织为一个 `std::vector<SendSlot> slots`，配合滑动窗口索引 base、next 使用。

(三) 连接管理

1. 连接建立：三次握手 senderHandshake

这个函数的功能是在客户端一侧模拟 TCP 的三次握手过程，同步序号和窗口参数，建立可靠连接。

```
static bool senderHandshake(SOCKET s, const sockaddr_in& serverAddr)
{
    setRecvTimeout(s, HANDSHAKE_TIMEOUT_MS);//设置接收超时时间

    PacketHeader syn{};//构造SYN报文
    syn.seq   = 0;
    syn.ack   = 0;
    syn.flags = FLAG_SYN;
    syn.wnd   = RECV_WINDOW;
    syn.reserved = 0;

    const int MAX_TRY = 5;//最多MAX_TRY次重试循环

    for (int i = 0; i < MAX_TRY; ++i)
    {
        std::cout << "[sender] send SYN\n";
        sendPacket(s, serverAddr, syn, nullptr, 0);

        PacketHeader resp{};
        std::vector<char> payload;
        sockaddr_in from{};

        if (recvPacket(s, resp, payload, from))
        {   //收到报文后进行判断标志位
            bool isSynAck =
                ((resp.flags & (FLAG_SYN | FLAG_ACK)) ==
                 (FLAG_SYN | FLAG_ACK));
            //判断确认号
            if (isSynAck && resp.ack == syn.seq + 1)
            {
                std::cout << "[sender] recv SYN-ACK\n";

                PacketHeader ack{};//构造最终ACK报文
                ack.seq   = syn.seq + 1;
                ack.ack   = resp.seq + 1;
                ack.flags = FLAG_ACK;
                ack.wnd   = RECV_WINDOW;
                ack.reserved = 0;
                //发送ack，并输出handshake success
                sendPacket(s, serverAddr, ack, nullptr, 0);
                std::cout << "[sender] handshake success\n";
                return true;
            }
```

```
        }

        std::cout << "[sender] handshake retry " << (i + 1) << "\n";
    }

    std::cerr << "[sender] handshake failed\n";
    return false;
}
```

该函数的流程如下：

1. 设置接收超时为 HANDSHAKE_TIMEOUT_MS，避免长时间阻塞；
2. 构造 SYN 报文：seq = 0，ack = 0，flags = FLAG_SYN，并设置本端接收窗口 wnd = RECV_WINDOW；
3. 在最多 MAX_TRY 次重试循环内：
   (a) 发送 SYN，并打印日志 [sender] send SYN；
   (b) 阻塞等待接收端响应，若超时则打印重试信息并重发 SYN；
   (c) 收到报文后，检查其是否同时包含 FLAG_SYN|FLAG_ACK，且 resp.ack == syn.seq + 1；
   (d) 若条件满足，则构造最终 ACK 报文：seq = syn.seq + 1，ack = resp.seq + 1，flags = FLAG_ACK，发送后握手成功，返回 true。

若在指定重试次数内始终未收到合法的 SYN+ACK，则认为握手失败，返回 false，上层函数会关闭套接字并退出。

我们的这一过程对应 TCP 中的 SYN → SYN+ACK → ACK，只是序号取值较为简化（固定从 0 开始）。

2. 连接终止：四次挥手 senderFourWayClose

这个函数的功能是在发送端主动发起连接关闭，模拟 TCP 的四次挥手过程。

```
static bool senderFourWayClose(SOCKET s, const sockaddr_in& serverAddr)
{
    setRecvTimeout(s, HANDSHAKE_TIMEOUT_MS);//设置接收超时时间

    PacketHeader fin1{};//构造第一次FIN报文
    fin1.seq   = 1;
    fin1.ack   = 0;
    fin1.flags = FLAG_FIN;
    fin1.wnd   = 0;
    fin1.reserved = 0;

    const int MAX_TRY = 5;

    for (int i = 0; i < MAX_TRY; ++i)
    {
```

```
16          std::cout << "[sender] send FIN\n";//发送日志
17           // sendPacket: 调用公共工具函数, 发送fin1包 (已提前设置flags=
                 FLAG_FIN、seq=1)
18          sendPacket(s, serverAddr, fin1, nullptr, 0);
19
20          // 等待 ACK (第二次挥手)
21          PacketHeader resp{};// 存储接收端的响应头部
22          std::vector<char> payload;// 存储响应的负载 (ACK包无负载, 仅占位)
23          sockaddr_in from{}; // 存储响应发送方的地址 (应等于接收端地址)
24          //等待接收端ACK超时, 重发
25          if (!recvPacket(s, resp, payload, from))
26          {
27              std::cout << "[sender] FIN wait ACK timeout, retry\n";
28              continue;
29          }
30          //收到接收端ACK
31          if ((resp.flags & FLAG_ACK) && resp.ack == fin1.seq + 1)
32          {
33              std::cout << "[sender] recv ACK of FIN\n";
34
35              // 等待对端 FIN (第三次挥手)
36              PacketHeader peerFin{};
37              std::vector<char> dummy;
38              if (!recvPacket(s, peerFin, dummy, from))
39              {
40                  std::cout << "[sender] wait peer FIN timeout, retry\n";
41                  continue; //重发自己的 FIN
42              }
43              // 校验: 接收端的响应是否是FIN包 (四次挥手第3次的合法响应)
44              if (peerFin.flags & FLAG_FIN)
45              {
46                  std::cout << "[sender] recv peer FIN\n";
47
48                  // 发送最后一个 ACK (第四次挥手)
49                  PacketHeader ack2{};
50                  ack2.seq   = 0;
51                  ack2.ack   = peerFin.seq + 1;
52                  ack2.flags = FLAG_ACK;
53                  ack2.wnd   = RECV_WINDOW;
54                  ack2.reserved = 0;
55
56                  sendPacket(s, serverAddr, ack2, nullptr, 0);
57                  std::cout << "[sender] four-way close done\n";
58                  return true;
59              }
60          }
61      }
62
```

```
63      std::cerr << "[sender] four-way close failed\n";
64      return false;
65  }
```

该函数的实现流程如下：

1. 设置接收超时为 HANDSHAKE_TIMEOUT_MS;
2. 构造第一次 FIN 报文（flags = FLAG_FIN），发送给接收端;
3. 在最多 MAX_TRY 次循环内:
   (a) 等待接收端返回对该 FIN 的 ACK（第二次挥手），检查 resp.ack == fin1.seq + 1;
   (b) 收到合法 ACK 后，继续等待接收端发送 FIN（第三次挥手）;
   (c) 收到对端 FIN 后，构造并发送最后一个 ACK（第四次挥手），日志打印 four-way close done，返回 true;
   (d) 在等待 ACK 或等待 FIN 任意一阶段超时时，重发自身 FIN 并重试。

若多次重试仍未完成四次挥手，则打印错误信息four-way close failed并返回 false。

（四） 主函数 runSender: 发送端整体流程

1. 函数定义

```
1  void runSender(const std::string& ip, uint16_t port, const std::string&
       inputFile)
```

该函数是发送端的核心，负责从套接字创建、握手，到数据传输、统计和挥手结束全过程。

2. 初始化阶段：Socket 创建与连接建立

这一阶段我们的目的是创建网络通信载体（UDP Socket）、解析接收端地址、通过三次握手建立可靠连接。

套接字创建 我们采用socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP) 生成 UDP 套接字,INVALID_SOCKET 表示创建失败（如系统资源不足、权限不够）;

```
1  CKET s = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
2      if (s == INVALID_SOCKET)
3      {
4          printLastError("socket");
5          return;
6      }
```

解析接收端地址 接下来是地址转换:

```
1  sockaddr_in server{};
2      server.sin_family      = AF_INET;
3      server.sin_port        = htons(port);
4      server.sin_addr.s_addr = inet_addr(ip.c_str());
```

- `htons(port)`：将主机字节序（小端序）的端口转换为网络字节序（大端序），确保跨设备兼容；
- `inet_addr(ip.c_str())`：将字符串 IP（如"192.168.1.100"）转换为网络字节序的 32 位整数；

三次握手建立连接 在三次握手阶段，我们调用函数、`senderHandshake` 函数完成连接建立（SYN→SYN-ACK→ACK），握手失败则直接退出，避免无效资源占用。

```
if (!senderHandshake(s, server))
    {
        closesocket(s);
        return;
    }
```

打开待发送文件 对于将要发送的文件，我们用、`std::ios::binary` 确保二进制文件（图片、视频等）不被篡改，打开失败则打印错误并关闭 Socket。

```
std::ifstream fin(inputFile, std::ios::binary);
    if (!fin)
    {
        std::cerr << "[sender] open input file failed\n";
        closesocket(s);
        return;
    }
```

3. 文件分片阶段：拆分文件为数据分组

这一阶段我们的目标是将大文件按 MAX_PAYLOAD=1000 字节拆分，每个分片封装为 SendSlot（含序号、数据、传输状态），便于后续分块传输和管理。

存储所有文件分片 首先进行初始化：

```
std::vector<SendSlot> slots;
char buffer[MAX_PAYLOAD];
uint32_t seq = 1;
```

设置临时缓冲区，每次读取 1000 字节，并设定数据分组起始序号，与握手序号独立，避免冲突。

循环读取文件

```
while (true)
{
    fin.read(buffer, MAX_PAYLOAD); // 每次读取最多 1000 字节
    std::streamsize n = fin.gcount(); // 实际读取的字节数
    if (n <= 0)
        break; // 读取完毕，退出循环

    // 初始化一个分片
    SendSlot slot;
```

```
        slot.hdr.seq   = seq++;           // 分配唯一序号
        slot.hdr.ack   = 0;               // 数据分组的 ack 字段无效，置 0
        slot.hdr.flags = FLAG_DATA;       // 标记为数据分组
        slot.hdr.wnd   = 0;               // 数据分组不通告窗口，置 0
        slot.hdr.reserved = 0;            // 保留字段置 0
        slot.data.assign(buffer, buffer + n); // 拷贝实际读取的数据到分片
        slots.push_back(std::move(slot)); // 加入分片列表（move 避免拷贝开销）
    }
    fin.close(); // 关闭文件，释放资源
```

我们固定按 1000 字节拆分（MAX_PAYLOAD），最后一块按实际剩余字节数封装，避免数据冗余；

用数据结构 SendSlot：是每个分片的“完整信息载体”，包含分组头部（序号、标志位）、数据内容、传输状态（是否发送 / 确认），后续所有操作围绕 slots 列表展开。根据发送的数据报设置 Slots。

空文件处理 若文件为空，直接发起四次挥手关闭连接，避免无效传输。

```
// 处理空文件场景
if (slots.empty())
{
    std::cout << "[sender] input file empty, nothing to send\n";
    senderFourWayClose(s, server); // 发起四次挥手关闭连接
    closesocket(s);
    return;
}
```

4. 传输参数初始化：窗口与统计配置

这个阶段我们的目标是初始化滑动窗口、拥塞控制、传输统计相关参数，为后续数据传输做准备。

```
const uint32_t firstDataSeq = 1; // 数据序号从 1 开始

// 拥塞控制: Reno
double cwnd     = 1.0;   // 拥塞窗口（单位: 分组）
double ssthresh = 16.0;  // 慢启动阈值

uint16_t peerWnd = static_cast<uint16_t>(g_recvWindow); // 对端通告窗口（初值）

// Reno 额外状态: 快重传 / 快恢复
uint32_t lastAckSeq     = firstDataSeq - 1; // 最近一次累计 ACK 序号
int      dupAckCount    = 0;                // 连续重复 ACK 次数
bool     inFastRecovery = false;            // 是否处于快速恢复
uint32_t recoverSeq     = 0;                // 进入快恢复时已发送的最高序号

size_t base = 0;   // 最早未确认分组下标
size_t next = 0;   // 下一个待发送分组下标
```

```
17
18 setRecvTimeout(s, 10); // 数据阶段：短超时轮询
```

这一部分有几个关键点需要说明：
首先是拥塞控制参数。

- **cwnd=1.0**：初始只能发送 1 个分组（慢启动起点）；
- **ssthresh=16.0**：慢启动阶段的上限，超过后进入线性增长的拥塞避免阶段；
- **peerWnd = RECV_WINDOW**：保存对端通告窗口。

接下来是我们定义的滑动窗口指针。

- base：标记“最早未被接收端确认”的分组下标，窗口左移的依据；
- next：标记“下一个待发送”的分组下标，窗口右移的依据；
- 窗口内分组：[base, next) 是“已发送未确认”，[next, base+windowLimit) 是“可发送未发送”；

最后是传输统计参数。

- **startTime, endTime：**记录整体传输时间；
- **bytesDelivered：**按 ACK 成功确认的数据字节总数；
- **totalPacketsSent：**DATA 分组发送总次数（包括重传）；
- **retransmissions：**DATA 分组重传次数；
- **rttSumUs, rttSamples：**以微秒为单位统计 RTT 总和与样本数。

我们将接收超时设置为 50 ms，用于数据阶段轮询，便于检查 ACK 和超时。

5. 主循环：接收并处理 ACK/SACK

主循环条件为 while (base < slots.size())，即只要存在未确认分组就持续执行。

```
1 while (base < slots.size())
```

批量发送分组 我们分为三个部分：

计算窗口上限 首先计算当前发送窗口上限：**计算窗口上限**：

$$\text{windowLimit} = \min\{\text{cwnd},\ \text{peerWnd},\ \text{剩余分组数}\};$$

```
1 // 计算当前可发送的窗口大小 = min(拥塞窗口, 接收端窗口, 剩余未发送分组数)
2 int windowLimit = static_cast<int>(
3     std::min<double>(
4         cwnd,
5         std::min<double>(peerWnd, static_cast<double>(slots.size() - base))))
          ;
```

发送新的 DATA 分组 接着，在窗口允许范围内发送新的 DATA 分组：

```cpp
// 发送窗口内的新分组（next 不超过窗口限制）
while (next < slots.size() && static_cast<int>(next - base) < windowLimit)
{
    SendSlot& slot = slots[next];
    auto now = Clock::now();

    // 记录首次发送时间（用于 RTT 统计）
    if (!slot.firstSent)
    {
        slot.firstSent    = true;
        slot.firstSendTime = now;

        if (!started)
        {
            started  = true;
            startTime = now; // 记录整个传输的开始时间
        }
    }

    // 更新最近发送时间，标记为已发送
    slot.lastSendTime = now;
    slot.sent         = true;

    // 调用工具函数发送分组（自动封装头部+计算校验和）
    if (!sendPacket(s, server,
                    slot.hdr,
                    slot.data.data(),
                    static_cast<uint16_t>(slot.data.size())))
    {
        closesocket(s); // 发送失败（如网络中断），直接退出
        return;
    }

    ++totalPacketsSent; // 统计总发送次数
    ++next;             // 窗口右边界右移，指向 next 待发送分组
}
```

对于每个待发送的 SendSlot:

- 首次发送时设置 `firstSent = true`，记录 `firstSendTime`;
- 每次发送前更新 `lastSendTime`;
- 调用 `sendPacket` 发送，`totalPacketsSent++`，`next++`;

尝试接收 ACK + SACK 这一部分我们的目标是接收接收端的确认包，标记已确认的分组（含累计确认和 SACK 选择性确认），更新窗口和拥塞控制参数。

这里我们分为四个部分来解释：

接受 ACK 包 这一部分做了接受 ACK 包后的初始处理:

```
// 接收 ACK 包（含 SACK 信息）
PacketHeader ackHdr{};
std::vector<char> ackPayload;
sockaddr_in from{};

if (recvPacket(s, ackHdr, ackPayload, from)) // 成功接收 ACK
{
    if (ackHdr.flags & FLAG_ACK) // 仅处理 ACK 类型的包
    {
        // 更新接收端通告窗口（0 时设为 1，避免窗口为 0 导致阻塞）
        peerWnd = (ackHdr.wnd == 0 ? 1 : ackHdr.wnd);

        bool anyNewAck = false; // 标记是否有新分组被确认
        auto now = Clock::now();

        // lambda 函数: 标记分组为已确认，并更新统计
        auto markIndexAcked = [&](size_t idx)
        {
            if (idx >= slots.size()) return;
            SendSlot& slot = slots[idx];
            if (!slot.acked) // 仅处理未确认的分组
            {
                slot.acked = true;
                anyNewAck  = true;

                bytesDelivered += slot.data.size(); // 累计交付字节数

                // 计算 RTT（当前时间 - 首次发送时间）
                if (slot.firstSent)
                {
                    auto rttUs = std::chrono::duration_cast<std::chrono::
                        microseconds>(
                        now - slot.firstSendTime).count();
                    if (rttUs > 0)
                    {
                        rttSumUs += static_cast<uint64_t>(rttUs);
                        ++rttSamples;
                    }
                }
            }
        };
```

首先进行缓冲区初始化，存储接受端 ACK 包的头部，负载，以及发送 ACK 包的端地址。

接着，进行接受和类型检验，调用工具函数接收 UDP 数据，成功返回 true（50ms 短超时，避免阻塞发送流程），我们仅处理带 ACK 标志的包，过滤数据、FIN 等无关包，避免无效处理。

其次，我们需要更新接收端的窗口实现流量控制。

- 接收端通过 ACK 头部的 wnd 字段告知“当前剩余缓存能接收的分组数”，发送端后续发送需遵守该限制（避免接收端缓冲区溢出）；
- 特殊处理：wnd=0 时强制设为 1，防止接收端暂时无缓存导致发送端“窗口为 0 阻塞”，确保传输流程不中断。

最后，进行确认逻辑的封装，目的是将“标记分组确认 + 统计更新”的逻辑复用，避免后续累计确认和 SACK 处理重复编码；

核心逻辑如下：

- **下标越界检查：** `idx >= slots.size()` 时直接返回，避免数组访问错误；
- **仅处理未确认分组：** `!slot.acked` 防止重复标记和统计错误；
- **统计更新：**累计交付字节数(`bytesDelivered`)、RTT 总和(`rttSumUs`)和样本数(`rttSamples`)，为最终传输报告提供数据；
- **anyNewAck 标记：**记录本次处理是否有新分组被确认，用于后续窗口和拥塞控制的触发条件。

处理累计确认 这一部分我们实现 TCP 累计确认，根据收到的确认号，标记发送窗口中**已经成功接收**的分组为**已确认**，释放资源。

```
1  // 1. 处理累计确认: 确认序号 ackHdr.ack - 1 的所有分组
2  if (ackHdr.ack >= firstDataSeq)
3  {
4      // 最大确认序号 = min(ackHdr.ack - 1, 最后一个分组序号)
5      uint32_t maxAckSeq = std::min<uint32_t>(
6          ackHdr.ack - 1,
7          firstDataSeq + static_cast<uint32_t>(slots.size()) - 1);
8
9      // 标记该范围内的分组为已确认
10     for (uint32_t seqNum = firstDataSeq; seqNum <= maxAckSeq; ++
       seqNum)
11     {
12         size_t idx = static_cast<size_t>(seqNum - firstDataSeq);
13         markIndexAcked(idx);
14     }
15 }
```

实现逻辑如下：

- **确认号过滤：** 首先我们需要过滤无效的确认号`if (ackHdr.ack >= firstDataSeq)`，只有当确认号大于发送窗口的起始序号时，才可能是确认的分组。
- **计算最大可确认序号：** 防止确认号超出当前发送窗口范围
- **标记范围内分组为已确认：** 遍历从「窗口起始序号 firstDataSeq」到「最大可确认序号 maxAckSeq」的所有分组。

处理 SACK 选择性确认 这一部分我们实现了 TCP 选择性确认（SACK）处理，解决**累计确认只能确认连续分组**的缺陷，专门标记**不连续的已接收分组**为**已确认**，提升传输效率。

```
        // 2. 处理 SACK 选择性确认: 确认不连续的分组 (解决累计确认缺陷)
        if (ackPayload.size() >= sizeof(uint16_t))
        {
            uint16_t blkCount = 0;
            // 从 ACK 负载中读取 SACK 区间数量
            std::memcpy(&blkCount, ackPayload.data(), sizeof(uint16_t));
            blkCount = std::min<uint16_t>(blkCount, MAX_SACK_BLOCKS); // 最多 4 个区间

            size_t offset = sizeof(uint16_t); // 负载偏移量 (跳过区间数量字段)
            for (uint16_t i = 0; i < blkCount; ++i)
            {
                // 检查负载是否足够存储一个 SACK 区间
                if (offset + sizeof(SackBlock) > ackPayload.size())
                    break;

                SackBlock blk{};
                // 读取一个 SACK 区间 (start=起始序号, end=结束序号)
                std::memcpy(&blk, ackPayload.data() + offset, sizeof(SackBlock));
                offset += sizeof(SackBlock);

                // 修正 SACK 区间 (避免超出当前发送的分组范围)
                uint32_t start = std::max<uint32_t>(blk.start, firstDataSeq);
                uint32_t end   = std::min<uint32_t>(blk.end, firstDataSeq + static_cast<uint32_t>(slots.size()) - 1);

                if (end < start) continue; // 无效区间, 跳过

                // 标记 SACK 区间内的分组为已确认
                for (uint32_t seqNum = start; seqNum <= end; ++seqNum)
                {
                    size_t idx = static_cast<size_t>(seqNum - firstDataSeq);
                    markIndexAcked(idx);
                }
            }
        }
```

实现的逻辑如下：

- **检查 SACK 数据是否有效：**判断 ACK 负载是否足够存储 SACK 区间数量（至少占 2 字节），若不够，说明没有 SACK 信息，直接跳过
- **读取 SACK 区间数量并限制上限：**从负载开头读取 SACK 区间数量，限制最大区间数

- **遍历每个 SACK 区间：**记录当前读取到的负载的位置，检查一个负载是否足够存储一个 SACK 区间，不够则退出循环，读取一个 SACK 区间的起始序号结束序号，修正区间，若正区间后无效，则跳过该区间，遍历区间内所有序号，计算数组索引并标记为已确认。

如果中间某个分组丢失，后续分组先收到时，通过 SACK 精准标记已收分组，无需等丢失分组重传后再累计确认，提升 TCP 传输效率。

更新窗口与拥塞控制（Reno + 快重传/快恢复） 该部分位于 ACK 处理分支内。代码中通过 `lastAckSeq / dupAckCount / inFastRecovery / recoverSeq` 实现 Reno 的快速重传与快速恢复，并与慢启动/拥塞避免共同构成完整的拥塞控制流程。

```
// 处理累计 ACK（Reno 部分）
if (ackHdr.ack >= firstDataSeq)
{
    if (ackHdr.ack > lastAckSeq)          // ACK 前进
    {
        lastAckSeq  = ackHdr.ack;
        dupAckCount = 0;

        // 若此前在快速恢复，则当 ACK 超过 recover 点时退出快恢复
        if (inFastRecovery && ackHdr.ack > recoverSeq)
        {
            inFastRecovery = false;
            cwnd           = ssthresh;
            if (cwnd > 64.0) cwnd = 64.0;
        }
    }
    else if (ackHdr.ack == lastAckSeq)    // 重复 ACK（dupACK）
    {
        ++dupAckCount;

        // 快速重传：dupACK >= 3 且仍有未确认分组
        if (!inFastRecovery && dupAckCount >= 3 && base < slots.size())
        {
            size_t lossIdx = base;
            if (slots[lossIdx].sent && !slots[lossIdx].acked)
            {
                // 进入快恢复：ssthresh 减半，cwnd = ssthresh + 3
                ssthresh = (cwnd / 2.0 < 2.0) ? 2.0 : (cwnd / 2.0);
                cwnd     = ssthresh + 3.0;
                if (cwnd > 64.0) cwnd = 64.0;

                inFastRecovery = true;
                recoverSeq = (next > 0) ? slots[next - 1].hdr.seq : slots[
                    lossIdx].hdr.seq;

                // 立即重传推测丢失的 base 分组
                slots[lossIdx].lastSendTime = now;
                sendPacket(... slots[lossIdx] ...);
```

```
                ++totalPacketsSent;
                ++retransmissions;
            }
        }
        else if (inFastRecovery)
        {
            // 快速恢复阶段: 每个额外 dupACK 线性增大 cwnd
            cwnd += 1.0;
            if (cwnd > 64.0) cwnd = 64.0;
        }
    }
    else
    {
        dupAckCount = 0;
    }
}

// 有新分组确认时: 窗口前移 + Reno 正常增长 (慢启动/拥塞避免)
if (anyNewAck)
{
    while (base < slots.size() && slots[base].acked) ++base;

    if (cwnd < ssthresh) cwnd += 1.0;          // 慢启动
    else                 cwnd += 1.0 / cwnd;   // 拥塞避免

    if (cwnd > 64.0) cwnd = 64.0;
}
```

- **快重传(Fast Retransmit)**:当累计确认号 `ackHdr.ack` 连续重复到第 3 次(`dupAckCount>=3`)时, 发送端推测窗口最左侧未确认分组 (`base`) 丢失, **不等待超时**, 立即重传 `slots[base]`。
- **快恢复 (Fast Recovery)**: 触发快重传后, 设置 `ssthresh` = $\max(2, \texttt{cwnd}/2)$, 并令 `cwnd` = `ssthresh` + 3, 进入 `inFastRecovery` 状态; 在快恢复阶段, 每收到一个额外重复 ACK, 执行 `cwnd+=1` 保持管道中仍有足够在途数据。当累计 ACK 前进并超过 `recoverSeq` (进入快恢复时记录的“已发送最高序号”) 后, 退出快恢复, 并令 `cwnd` = `ssthresh`。
- **慢启动/拥塞避免**: 当存在新确认 (`anyNewAck=true`) 时, 窗口前移并按 Reno 规则增长: `cwnd<ssthresh` 时慢启动 (`cwnd+=1`), 否则拥塞避免 (`cwnd+=1/cwnd`), 并将 `cwnd` 上限限制为 64。

超时重传处理 这一阶段我们的目标是检测“已发送未确认”的分组是否超时 (超过 TIMEOUT_MS=500ms), 超时则重传, 并调整拥塞控制参数 (应对网络拥塞)。

```
auto now = Clock::now();
// 遍历所有“已发送未确认”的分组 (base    i < next)
for (size_t i = base; i < next; ++i)
{
    SendSlot& slot = slots[i];
```

```
    if (!slot.sent || slot.acked)
        continue; // 跳过未发送或已确认的分组

    // 计算距离最近一次发送的时间
    auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(
        now - slot.lastSendTime).count();

    if (elapsed > TIMEOUT_MS) // 超时（超过 500ms 未确认）
    {
        std::cout << "[sender] timeout seq=" << slot.hdr.seq << "\n";

        // 重传该分组
        slot.lastSendTime = now; // 更新最近发送时间
        if (!sendPacket(s, server,
                        slot.hdr,
                        slot.data.data(),
                        static_cast<uint16_t>(slot.data.size())))
        {
            closesocket(s); // 重传失败，退出
            return;
        }

        // 更新统计
        ++totalPacketsSent;
        ++retransmissions;

        // 拥塞控制：超时视为网络拥塞，退回慢启动
        ssthresh = (cwnd / 2.0 < 2.0) ? 2.0 : (cwnd / 2.0); // 阈值设为当前
            cwnd 的一半（最小 2）
        cwnd     = 1.0; // 拥塞窗口重置为 1，重新慢启动
    }
}
```

其中有两个关键的统计变量：

1. **totalPacketsSent:** 统计已发送的总分组数（含首次发送和重传）
2. **retransmissions:** 统计重传的分组总数，用于后续计算重传率

我们的实现逻辑如下：

- 首先，记录当前时间，作为超时判断的基准
- 遍历“已发送未确认的分组”（base i < next 是发送窗口的核心范围）
  - 对每个符合条件的分组，计算从“最后一次发送”到当前的时间差；
  - 若时间差超过超时阈值（TIMEOUT_MS, 500ms），重传该分组，并更新统计状态，触发拥塞控制退避，超时我们视为严重拥塞，退回慢启动阶段。
  - 若重传失败，关闭连接并退出。

对于拥塞控制退避，超时被视为“严重网络拥塞”，说明网络已无法承载当前发送速率，因此触发 Reno 算法的“最严厉退避”：

我们需要更新慢启动阈值 ssthresh，设为当前 cwnd 的一半，相当于拥塞窗口减半，降低后续发送速率上限

其次，需要重置拥塞窗口 cwnd 为 1，重新进入“慢启动阶段”，从最小窗口开始试探网络容量，避免继续给拥塞的网络施压。

上一段代码是“收到新确认（ACK）时的拥塞窗口增长”（正向增长），本轮代码是“超时未收到 ACK 时的拥塞窗口退避”（反向退避），两者共同构成 Reno 拥塞控制的完整逻辑：

- 正常情况（收到 ACK）：慢启动 → 拥塞避免，窗口逐步增长；
- 异常情况（超时）：窗口减半 + 重置为 1，退回慢启动，应对拥塞。

6. 传输结束：关闭连接与统计输出

这段代码是我们的 RUDP 发送端的“传输收尾与统计报告”模块，核心作用是：当所有数据分组都被接收端确认后，优雅关闭连接、计算关键传输指标（吞吐量、丢包率、平均 RTT 等），并输出最终统计报告，用于评估传输性能。

```
// 所有分组已确认，记录传输结束时间
endTime = Clock::now();

// 发起四次挥手，优雅关闭连接
senderFourWayClose(s, server);
closesocket(s); // 关闭 Socket，释放网络资源

// 计算统计指标
double durationSec = std::chrono::duration<double>(endTime - startTime).count
    ();
if (durationSec <= 0.0) durationSec = 1e-6; // 避免除零错误

// 吞吐量：MB/s 和 Mbps（1 MB/s = 8 Mbps）
double throughputMBps = static_cast<double>(bytesDelivered) / durationSec /
    (1024.0 * 1024.0);
double throughputMbps = throughputMBps * 8.0;

// 丢包率：重传次数 / 总发送次数（近似值）
double lossRate = (totalPacketsSent > 0)
    ? static_cast<double>(retransmissions) / static_cast<double>(
        totalPacketsSent)
    : 0.0;

// 平均 RTT
double avgRttUs = (rttSamples > 0)
    ? static_cast<double>(rttSumUs) / static_cast<double>(rttSamples)
    : 0.0;

// 输出统计报告
std::cout << "=====␣RUDP␣Statistics␣(Sender)␣=====\n";
```

```
28 std::cout << "Bytes␣delivered:␣␣␣␣␣␣␣" << bytesDelivered << "␣bytes\n";
29 std::cout << "Data␣packets␣sent:␣␣␣␣␣" << totalPacketsSent
30           << "␣(retransmissions=" << retransmissions << ")\n";
31 std::cout << "Approx.␣loss␣rate:␣␣␣␣␣" << lossRate * 100.0 << "␣%\n";
32 std::cout << "Average␣RTT:␣␣␣␣␣␣␣␣␣␣" << avgRttUs << "␣us\n";
33 std::cout << "Throughput:␣␣␣␣␣␣␣␣␣␣␣" << throughputMBps
34           << "␣MB/s␣(" << throughputMbps << "␣Mbps)\n";
```

我们的实现逻辑如下：

- **记录传输结束时间：** startTime（传输开始时记录）配合计算总耗时。
- **发起四次挥手关闭连接：**我们利用`senderFourWayClose`模拟 TCP 的四次挥手流程，病关闭套接字，释放资源
- **计算性能指标：**我们计算出来传输总耗时，吞吐量，丢包率，平均 RTT，并输出统计报告。

吞吐量的定义是，在单位时间内成功交付的有效数据量，是评估传输效率的核心指标。**吞吐量 (MB/s) = 总有效字节数 ÷ 总耗时 ÷ 1MB（1024×1024 字节）**

丢包率的定义是，重传的分组数占总发送分组数的比例，反映网络的可靠性（丢包率越低，网络越稳定）。

平均 RTT 的定义是，分组从发送端发出，到接收端确认收到的“往返时间”，反映网络延迟。用所有有效分组的 RTT 总和除以有效 RTT 的采样个数可以得到。

我们的指标全面，让传输性能直观可见，可以直观地调试和优化。

(五) 总结

总体来看，我们设计的 `rudp_sender.cpp` 将“tcp-like 可靠传输”的所有发送端逻辑集中在一个模块中：前半部分负责连接管理（三次握手与四次挥手），中间的大循环实现基于滑动窗口 + SACK + Reno 的可靠数据传输，尾部负责统计与结果输出，结构清晰，在实验报告中我们已经进行逐块展开分析。

八、 rudp_receiver.cpp

(一) 整体概述

该代码是接收端的完整实现，核心功能包括：三次握手建立连接、数据接收（含乱序缓存 +SACK 选择性确认）、四次挥手关闭连接，完全匹配 TCP 协议“可靠传输”的核心需求。

我们的代码分为 3 个核心模块还有主函数的入口。

- **三次握手模块（receiverHandshake）**：建立可靠连接，确认双方通信能力；
- **SACK 确认模块（sendAckWithSack）**：构造带选择性确认的 ACK 报文，告知发送端“哪些分组已收到、哪些丢失”；
- 主逻辑模块（runReceiver）：接收数据、乱序缓存、有序写入文件，触发四次挥手；
- **四次挥手模块（内嵌在 runReceiver 末尾）**：优雅关闭连接，确保数据无残留。

接下来我们将逐部分进行讲解。

(二) 连接管理：三次握手

这部分实现了服务器端的连接管理三次握手，与发送端的三次握手协同工作。整体流程是：

- 阻塞等待客户端发来 SYN；
- 收到 SYN 后回复 SYN-ACK；
- 等待客户端最后一个 ACK 确认，如果迟迟等不到就重发 SYN-ACK，最多尝试若干次；
- 成功收到最后的 ACK 则握手成功，否则握手失败。

```
static bool receiverHandshake(SOCKET s, sockaddr_in& clientAddr)
{    // 1. 第一步：阻塞等待客户端的 SYN 报文（不设超时，直到收到为止）
    setRecvTimeout(s, 0);

    PacketHeader syn{};
    std::vector<char> dummy;

    std::cout << "[receiver] wait for SYN...\n";
    while (true)
    {    // 循环接收报文，直到收到“带 SYN 标志”的报文
        if (!recvPacket(s, syn, dummy, clientAddr))
            continue;
        // 检查报文是否带 SYN 标志（0x01，假设 FLAG_SYN 是预定义常量）
        if (syn.flags & FLAG_SYN)
            break;// 收到 SYN，退出循环
    }

    std::cout << "[receiver] recv SYN\n";
    // 2. 第二步：发送 SYN-ACK 报文（回应客户端的 SYN）
    PacketHeader synAck{};
    synAck.seq    = 100;              // 服务端自己的初始序号（随便选）
    synAck.ack    = syn.seq + 1;
    synAck.flags = FLAG_SYN | FLAG_ACK;
    synAck.wnd    = RECV_WINDOW;
    synAck.reserved = 0;

    std::cout << "[receiver] send SYN-ACK\n";
    sendPacket(s, clientAddr, synAck, nullptr, 0);// 发送 SYN-ACK，无数据（
        nullptr）

    // 等最后一个 ACK
    setRecvTimeout(s, HANDSHAKE_TIMEOUT_MS);// 设超时（如 1000ms），避免无限
        等
    const int MAX_TRY = 5;// 最多重传 5 次 SYN-ACK，还没收到就握手失败

    for (int i = 0; i < MAX_TRY; ++i)
    {
        PacketHeader last{};
```

```
37          std::vector<char> dummy2;
38          sockaddr_in from{};
39          if (!recvPacket(s, last, dummy2, from))// 等待客户端 ACK
40          {
41              // 超时则重发 SYN-ACK
42              std::cout << "[receiver] wait ACK timeout, resend SYN-ACK\n";
43              sendPacket(s, clientAddr, synAck, nullptr, 0);
44              continue;
45          }
46
47          if ((last.flags & FLAG_ACK) && last.ack == synAck.seq + 1)
48          {
49              std::cout << "[receiver] handshake success\n";
50              setRecvTimeout(s, 0); //握手成功, 数据传输阶段改回阻塞模式
51              return true;
52          }
53      }
54      // 5 次重传后仍没收到 ACK, 握手失败
55      std::cerr << "[receiver] handshake failed\n";
56      return false;
57  }
```

我们的实现逻辑如下：

首先将 socket 设置为“无限阻塞等待”，即 recvPacket 一直等到收到数据才返回。

第二步，构造并发送 SYN-ACK，服务器设置一个自己的初始序号，我们设置为 100，并告诉客户端收到了你序号为 syn.seq 的 SYN，你的下一个期望序号是 syn.seq + 1”。设置 flags，携带 SYN+ACK 标志，并告诉对端接收窗口的大小，客户端需要根据这个值来设置发送窗口的大小。此处负载为 0，只发头部。

第三步等待客户端的最后一个 ACK，并且支持重发 SYN-ACK，对应 TCP 三次握手中的“第三次握手”。我们最多尝试 5 次，如果连续 5 次尝试都收不到合适的 ACK，则认为握手失败。

最后一步，如果始终没有收到符合条件的 ACK，则返回 false，关闭连接。

(三) SACK 确认: 解决乱序 / 丢包问题

UDP 传输可能出现“分组乱序”（后发的分组先到）或“分组丢失”，而传统 TCP 用“累积确认”（只确认最后一个有序分组），会导致发送端重传大量已收到的分组。

我们对这一部分进行优化，实现选择性确认，接收端告诉发送端“已收到的不连续分组范围”，发送端只需重传真正丢失的分组，提升传输效率。

核心逻辑为：

- 从乱序缓存中提取“已收到但未连续”的分组序号，整理成“序号区间”（如 [3,5] 表示序号 3、4、5 的分组已收到）；
- **构造 ACK 报文：**包含“累积确认号”（最后一个有序收到的分组序号）+ SACK 区间（乱序收到的分组范围）；
- 发送 ACK 报文，更新接收窗口（告知发送端当前还能缓存多少分组）。

```cpp
// 构造 ACK + SACK payload 并发送
static void sendAckWithSack(
    SOCKET s,
    const sockaddr_in& clientAddr,
    uint32_t cumulativeAck,
    const std::map<uint32_t, std::vector<char>>& buffer)
{
    //1. 第一步: 从乱序缓存中整理 SACK 区间 (把分散的序号合并成连续区间)
    std::vector<SackBlock> blocks;
    uint32_t lastStart = 0, lastEnd = 0;
    bool hasRange = false;
    // 遍历乱序缓存 (map 是有序的, 所以序号从小到大遍历)
    for (const auto& kv : buffer)
    {
        uint32_t seq = kv.first; // 当前分组的序号
        if (seq <= cumulativeAck)// 已被累积确认的分组, 不用加入 SACK
            continue;
        //还没开始构建区间, 初始化第一个区间
        if (!hasRange)
        {
            lastStart = lastEnd = seq;
            hasRange  = true;
        } // 当前序号和上一个区间连续 (如 lastEnd=3, seq=4), 合并区间
        else if (seq == lastEnd + 1)
        {
            lastEnd = seq;
        }
        else// 当前序号和上一个区间不连续 (如 lastEnd=3, seq=5), 保存上一个
            区间, 开始新区间
        {
            blocks.push_back(SackBlock{lastStart, lastEnd});
            if (blocks.size() >= MAX_SACK_BLOCKS)
                break;
            lastStart = lastEnd = seq;
        }
    }
     // 把最后一个未保存的区间加入 SACK
    if (hasRange && blocks.size() < MAX_SACK_BLOCKS)
    {
        blocks.push_back(SackBlock{lastStart, lastEnd});
    }

    uint16_t blkCount = static_cast<uint16_t>(blocks.size());

    //2. 第二步: 构造 SACK 报文的 payload (数据部分)
    std::vector<char> payload;
    // payload 格式: [区间数 (2字节)][区间1 (SackBlock)][区间2 (SackBlock)
```

```
        ]...
    payload.resize(sizeof(uint16_t) + blkCount * sizeof(SackBlock));
    std::memcpy(payload.data(), &blkCount, sizeof(uint16_t));
    size_t offset = sizeof(uint16_t);
    for (size_t i = 0; i < blocks.size(); ++i)
    {
        std::memcpy(payload.data() + offset,
                    &blocks[i], sizeof(SackBlock));
        offset += sizeof(SackBlock);
    }

    PacketHeader ackHdr{};
    ackHdr.seq    = 0;
    ackHdr.ack    = cumulativeAck;
    ackHdr.flags = FLAG_ACK;
    // 简单流量控制: 窗口 = RECV_WINDOW - 当前缓存的分组数
    uint16_t avail = static_cast<uint16_t>(
        std::max<int>(1, RECV_WINDOW -
                         static_cast<int>(buffer.size())));
    ackHdr.wnd    = avail;
    ackHdr.reserved = 0;

    sendPacket(s, clientAddr,
               ackHdr,
               payload.data(),
               static_cast<uint16_t>(payload.size()));
}
```

我们的实现逻辑如下:

- **从 buffer 中构造 SACK 区间列表:** 根据接收端的乱序缓冲区 buffer 中已经到达但未按顺序交付的分组，构造若干个 SACK 区间 [start,end]，用于选择性确认。
- **将 SACK 区间打包到 payload:** SACK 信息编码到 ACK 报文的负载部分 payload 中，首部 2 字节表示有多少个 SACK 区间，后面一次是 blkCount 个 SackBlock 结构体。当发送端收到后，就根据格式解析出来所有 SACK 区间，逐个标记这些序号范围为已确认。
- **构造 ACK 头部 + 简单流量控制，并发送:** 构造 ACK 的头部（不含 SACK，只是 ACK 主体），并携带刚刚构造好的 payload 一起发出去。

  发送端收到后:

  - 先根据 ackHdr.ack 做累计确认，标记 [firstSeq, ack-1] 为已确认;
  - 再解析 payload 中的 SACK 区间，补上乱序收到的块，提升利用率、减少不必要重传。

(四) 主逻辑：接收数据 + 有序写入

这段代码是我们的接收端的核心工作流: 绑定端口 → 三次握手 → 循环接收数据 → 乱序缓存 → 有序写入文件 → 触发四次挥手。

```cpp
void runReceiver(uint16_t port, const std::string& outputFile)
{
    ...// 1. 初始化 UDP 套接字（SOCK_DGRAM 表示 UDP 协议）
    ...// 2. 绑定端口（接收端需要固定端口，让发送端能找到）
    ...//3. 三次握手建立连接（失败则关闭套接字退出）
    ...//4. 打开输出文件（二进制模式，避免文本模式修改数据）
    uint32_t expectedSeq = 1; // 期望的下一个有序分组号
    std::map<uint32_t, std::vector<char>> buffer; // 乱序缓存
    bool finReceived = false;是否收到客户端的 FIN 报文（关闭连接请求）
    uint32_t finSeq  = 0;//FIN 报文的序号（用于后续 ACK 确认）
    while (!finReceived)
    {
        PacketHeader hdr{};
        std::vector<char> data;
        sockaddr_in from{};

        if (!recvPacket(s, hdr, data, from))
            continue;

        if (hdr.flags & FLAG_DATA)
        {
            // 收到数据分组
            if (hdr.seq >= expectedSeq)
            {
                // 只缓存之前没收到的 seq
                if (buffer.find(hdr.seq) == buffer.end())
                {
                    buffer[hdr.seq] = data;
                }

                // 把连续有序的分组写入文件
                while (true)
                {
                    auto it = buffer.find(expectedSeq);
                    if (it == buffer.end())
                        break;

                    fout.write(it->second.data(),
                               static_cast<std::streamsize>(
                                   it->second.size()));
                    buffer.erase(it);
                    ++expectedSeq;
                }
            }

            uint32_t cumulativeAck = expectedSeq - 1;
            sendAckWithSack(s, clientAddr, cumulativeAck, buffer);
```

```
        }
        ...//6.2 收到 FIN 报文（客户端请求关闭连接）
    }
```

我们的实现的基本逻辑如下：

- **创建 Socket 并绑定端口：**我们创建一个 UDP 套接字，填充本地地址结构
- **三次握手建立连接：**我们调用之前实现的 receiverHandshake，握手成功后，clientAddr 里保存了对端（发送端）的地址，后续发送 ACK/ACK+SACK 都用这个地址；
- **打开文件：**以二进制方式打开输出文件 outputFile；
- **初始化有序接收状态与乱序缓存：**设置期望的下一个分组号，初始化缓存，用 map 的排序特性，遍历时序号天然有序，方便构造 SACK 区间和检查连续段。
- **主接收循环接收数据 / FIN，重组并确认：**先处理数据分组, 如果是数据包，只缓存之前没收到的 seq”，写入连续有序数据到文件，发送 ACK + SACK。如果收到 FIN 报文，跳出循环。
- **结束关闭文件**

（五） 关闭连接：四次挥手

这一部分实现四次挥手，数据传输完成后，通过四次挥手确保双方“都已完成数据接收”，避免直接关闭导致的数据残留（UDP 无连接，直接关闭可能丢数据）。

```
// ======= 四次挥手（服务端被动关闭） =======
// 已经完成：(1) 客户端 ---> FIN
// 现在要完成：(2) 服务端 ---> ACK；(3) 服务端 ---> FIN；(4) 客户端 ---> ACK
    ...//(2) 第一步：ACK 客户端的 FIN 报文
    std::cout << "[receiver] send ACK of FIN\n";
    // (3)第二步：发送服务端自己的 FIN 报文（告知客户端“我也准备关闭”）
    PacketHeader fin2{};
    fin2.seq   = 2;
    fin2.ack   = 0;
    fin2.flags = FLAG_FIN;
    fin2.wnd   = 0;
    fin2.reserved = 0;

    setRecvTimeout(s, HANDSHAKE_TIMEOUT_MS);
    const int MAX_TRY = 5;

    for (int i = 0; i < MAX_TRY; ++i)
    {
        std::cout << "[receiver] send FIN\n";
        sendPacket(s, clientAddr, fin2, nullptr, 0);

        ...// 等待客户端最后的 ACK
        if ((resp.flags & FLAG_ACK) && resp.ack == fin2.seq + 1)
        {
```

```
25              std::cout << "[receiver] four-way close done\n";
26              break;
27          }
28      }
29      closesocket(s);
30  }
```

服务器端是被动关闭方，客户端先发起 FIN，核心流程就是“ACK→FIN→ 等 ACK”；

发送 FIN 时，wnd=0 告知客户端“我不再接收任何数据了”，避免客户端继续发数据；并且实现超时重传机制服务端发送 FIN 后，客户端可能没收到，所以最多重传 5 次，确保客户端能收到。

(六) 总结

我们的这部分代码实现了接收端的可靠传输，用三次握手建立可靠通信的前提，采用 SACK 机制解决了 UDP 乱序丢包问题，并通过乱序缓存和有序写入确保接收端最终写入文件的数据是有序的完整的，最后四次挥手，优雅关闭连接。同时，通过接收窗口动态调整，避免缓存溢出。

整个流程完全模拟了 TCP 的核心可靠机制，但基于 UDP 实现。

九、 main.cpp

该文件作为程序入口，完成以下工作，代码不再展示：

- **初始化网络库**：调用 `WSAStartup` 初始化 Winsock，在程序结束前统一调用 `WSACleanup` 释放资源。
- **解析命令行参数**：根据第一个参数判断工作模式，并支持可选参数：
  - `recv <port> <output_file> [window_size]`:作为接收端运行,其中可选的 `window_size` 用于指定接收端允许的滑动窗口大小；
  - `send <server_ip> <port> <input_file> [delay_ms] [loss_percent]`:作为发送端运行，可选的 `delay_ms` 和 `loss_percent` 用于模拟链路延迟和丢包率。
- **参数合法性检查**：对不同模式分别检查参数数量是否为允许的几种形式：
  - `recv` 模式仅接受 3 个或 4 个附加参数（带或不带 `window_size`）；
  - `send` 模式仅接受 3 个或 5 个附加参数（带或不带 `delay_ms` 和 `loss_percent`）。

  若参数数量不符合预期或模式非法，则调用 `printUsage()` 打印用法提示并退出。
- **分发到具体逻辑**：
  - 在 `recv` 模式下,解析端口和输出文件名;若给出了 `window_size`,则通过 `clampWindowSize` 进行上下界限制后赋值给全局变量 `g_recvWindow`，否则使用默认窗口大小，然后调用 `runReceiver(port, outFile)` 完成握手、接收与写文件；
  - 在 `send` 模式下,解析服务器 IP、端口和输入文件名;若给出了 `delay_ms` 和 `loss_percent`，则分别转换为整数延迟和 $[0, 1]$ 范围内的丢包率，并通过 `setLinkOptions(delayMs, lossRate)` 进行全局设置，最后调用 `runSender(ip, port, file)` 完成握手与可靠发送。

十、 实验方案设计

(一) 测试场景与步骤

我们进行本机回环测试。

1. **本机回环测试**

   - 接收端启动：在命令行执行 `rudp.exe recv 9000 out.txt`，初始化接收端套接字并进入阻塞等待状态，等待发送端的 SYN 报文。
   - 发送端启动：在另一个命令行执行 `rudp.exe send 127.0.0.1 9000 test.txt`，完成三次握手后开始传输 `test.txt` 内容。
   - 传输完成后，接收端自动将数据写入 `out.txt` 并通过四次挥手关闭连接。

(二) 测试指标

1. **RTT 计算**：发送端通过记录分组首次发送时间 (`firstSendTime`) 与收到 ACK 的时间差，累计有效样本数 `rttSamples`，并计算平均 RTT：

$$\overline{\mathrm{RTT}} = \frac{\texttt{rttSumUs}}{\texttt{rttSamples}}.$$

2. **丢包率**：基于发送端统计的重传次数 `retransmissions` 和总发送次数 `totalPacketsSent`，定义

$$\text{丢包率} = \frac{\texttt{retransmissions}}{\texttt{totalPacketsSent}} \times 100\%.$$

3. **吞吐量**：

   - 字节吞吐量：$\mathrm{Throughput}_{\mathrm{B/s}} = \dfrac{\texttt{bytesDelivered}}{\texttt{endTime} - \texttt{startTime}}$，再换算为 MB/s（1MB = 1024×1024 字节）。
   - 比特吞吐量：比特吞吐量 = 字节吞吐量 ×8，单位 Mbps。

(三) 参数设置

1. **基础参数**：

   - 报文负载大小：`MAX_PAYLOAD` = 1000 字节（见 `rudp.h`，用于限制单个 DATA 分组的最大负载）。
   - 接收窗口（流量控制窗口通告）：本实现使用运行时变量 `g_recvWindow` 表示接收端窗口大小，默认 `DEFAULT_RECV_WINDOW` = 64（见 `rudp.h`），接收端可在命令行中通过 `rudp.exe recv <port> <out> [window_size]` 指定窗口大小；实验中取值 8、16、32、64（默认 64）。
   - 超时时间：握手阶段采用动态超时 `HANDSHAKE_TIMEOUT_MS` + 2 × `g_linkDelayMs`（见 `senderHandshake()/receiverHandshake()`），其中 `HANDSHAKE_TIMEOUT_MS` = 1000ms;数据阶段的重传超时使用运行时变量 `g_dataTimeoutMs`,由 `setLinkOptions()` 统一设置为 300ms （见 `rudp_common.cpp`）。

2. **文件测试集**：

- 文本文件：helloword.txt
- 二进制文件：测试集中的三张图片.png）和压缩包（.zip）作为大文件。

3. **丢包与延迟模拟**：

- 丢包/延迟的注入位置在 sendPacket 内部完成（见 rudp_common.cpp）。
  具体做法是：
  - 对**非纯 ACK** 报文（DATA/SYN/FIN 或带 SYN/FIN 的控制包）进行随机丢包与延迟；
  - 对**纯 ACK**（仅带 FLAG_ACK 且不含 DATA/SYN/FIN）不做丢包、不做延迟，以避免 ACK 被随机丢弃导致发送端/接收端在流控与确认上陷入长时间停滞。
- 随机丢包实现：sendPacket 使用线程本地随机数引擎 mt19937 生成 $[0,1)$ 的均匀分布随机数，若 dist(rng) < g_lossRate 则直接“什么都不发”（跳过 sendto，返回 true），从而模拟链路丢包：

```
// sendPacket(...) 内部：对非纯 ACK 包按 g_lossRate 概率丢弃
static thread_local std::mt19937 rng{std::random_device{}()};
std::uniform_real_distribution<double> dist(0.0, 1.0);
if (dist(rng) < g_lossRate) {
    return true; // 丢包：不调用 sendto
}
```

- 延迟模拟实现：在真正调用 sendto 之前，若 g_linkDelayMs > 0，则先睡眠对应毫秒数，模拟“单向链路时延”：

```
// sendPacket(...) 内部：发送前人为延迟（单向）
if (g_linkDelayMs > 0) {
    std::this_thread::sleep_for(
        std::chrono::milliseconds(g_linkDelayMs));
}
```

延迟设置方式（命令行参数） 本实现通过发送端命令行参数设置延迟与丢包率（见 main.cpp）：

```
{rudp.exe send <server_ip> <port> <input_file> [delay_ms] [loss_percent]}
```

当提供 delay_ms 与 loss_percent 时,main 会调用 setLinkOptions(delayMs, lossPercent/100.0)，从而设置全局变量 g_linkDelayMs 与 g_lossRate，并将数据阶段重传超时 g_dataTimeoutMs 统一调整为 300ms。

例如：

```
rudp.exe send 127.0.0.1 9000 in.jpg 8 5
```

表示对发送端发出的**非纯 ACK** 报文引入 8ms 单向延迟与 5% 随机丢包。

接收端窗口大小通过

```
rudp.exe recv <port> <out> [window_size]
```

设置，默认大小是 64，例如：

rudp.exe recv 9000 out.zip 32。

十一、 实验结果与分析

(一) 基础功能验证

1. 正常传输验证

我们在接收端输入命令 `rudp.exe recv 9000 out.txt`，发送端输入命令 `rudp.exe send 129.0.0.1 9000 test.txt`。

```
F:\Reports\Computer_Internet\Code\Lab2>main.exe recv 9000 out.txt
[receiver] wait for SYN...
[receiver] wait for SYN...
[receiver] recv SYN
[receiver] send SYN-ACK
[receiver] handshake success
[receiver] recv FIN
[receiver] send ACK of FIN
[receiver] send FIN
[receiver] four-way close done
```

图 1: 接收端

```
F:\Reports\Computer_Internet\Code\Lab2>main.exe send 127.0.0.1 9000 helloworld.txt
[sender] send SYN
[sender] recv SYN-ACK
[sender] handshake success
[sender] send FIN
[sender] recv ACK of FIN
[sender] recv peer FIN
[sender] four-way close done
===== RUDP Statistics (Sender) =====
Bytes delivered:       1655808 bytes
Data packets sent:     1656 (retransmissions=0)
Approx. loss rate:     0 %
Average RTT:           1166.03 us
Throughput:            23.2864 MB/s (186.291 Mbps)
```

图 2: 发送端

从终端输出可以看到，

- 三次握手：发送端日志依次输出 `[sender] send SYN`，接收端输出 `[receiver] recv SYN`，随后发送端输出 `[sender] recv SYN-ACK`，表明握手完成。
- 数据传输：接收端 `out.txt` 完整包含 `test.txt` 内容，传输字节数为 1655808，传输数据包数为 1656，并且丢包率为 0，平均 RTT 为 1166.03us，传输速率为 23.2864MB/s。
- 四次挥手：发送端输出 `[sender] send FIN`，接收端输出 `[receiver] send ACK of FIN` 与 `[receiver] send FIN`，最终发送端输出 `[sender] four-way close done`，连接正常关闭。

2. 异常处理验证

我们不开接收端，发送端在超时重传 5 次后输出 `[sender] handshake failed` 并退出。

```
F:\Reports\Computer_Internet\Code\Lab2>main.exe send 127.0.0.1 9000 helloworld.txt
[sender] send SYN
[recvfrom] WSA error: 10054
[sender] handshake retry 1
[sender] send SYN
[recvfrom] WSA error: 10054
[sender] handshake retry 2
[sender] send SYN
[recvfrom] WSA error: 10054
[sender] handshake retry 3
[sender] send SYN
[recvfrom] WSA error: 10054
[sender] handshake retry 4
[sender] send SYN
[recvfrom] WSA error: 10054
[sender] handshake retry 5
[sender] handshake failed
```

图 3: 发送端异常处理

并且，在网络短暂中断后恢复的场景中，发送端依靠基于 `lastSendTime` 的超时重传机制，仍可继续完成文件传输。

(二) 窗口大小对吞吐量的影响

1. 实验数据（本机回环，无丢包）

接收窗口大小（分组）	平均 RTT（us）	吞吐量（MB/s）	总发送次数
8	165.149	43.6787	1656
16	473.329	30.8166	1656
32	1163.06	20.526	1656
64	2057.25	17.5951	1656

表 2: 不同接收窗口大小下的 RTT 与吞吐量（本机回环，无丢包）

2. 分析

结果波动原因分析 在相同接收窗口大小下多次重复实验时，发现测得的吞吐量和 RTT 存在较为明显的波动。经过查阅资料和分析，我得出结论，这并不是我们的协议本身不稳定，而主要由以下几个因素共同造成：

- **本机回环 + 极小 RTT**。在本机回环（`127.0.0.1`）上，真实 RTT 只有几十到几百微秒，`std::chrono` 读时钟的开销、一次调度或中断本身就可能是 $\mathcal{O}(10 \sim 100)$ 微秒级，因此计时抖动相对 RTT 的占比很大，导致同一窗口下多次实验的平均 RTT 差异明显。
- **操作系统调度的随机性**。发送端和接收端都是普通用户进程，CPU 被其他进程抢占、内核中断、上下文切换等，都会短暂推迟分组发送或 ACK 处理，从而拉长部分 RTT 和总耗时。
- **实验时间短，固定开销占比高**。在回环环境下，文件传输往往只需几十毫秒，握手、文件 I/O 和少量调度延迟在总时间中占比很大，不同实验中这些固定开销略有变化，就会放大到吞吐量统计结果上。

- **缓存效应**。首次发送需要从磁盘读入文件，之后多次实验则主要命中页缓存和 CPU 缓存，不同实验之间缓存状态不同，也会引入不可控的时间偏差。

综上，即便在相同接收窗口设置下，由于本机回环环境 RTT 极小、操作系统调度和缓存状态等多种不确定因素的叠加。

参数设置导致结果波动分析 可以观察到两个明显特征：

- **平均 RTT 随接收窗口大小单调增大**：从 8 分组到 64 分组，平均 RTT 由约 165 μs 增加到约 2057 μs，接近一个数量级的增长。
- **吞吐量反而随窗口增大而下降**：8 分组时约为 43.68 MB/s, 64 分组时下降到约 17.60 MB/s。同时可以看到，总发送次数在四组实验中均为 1656，说明在无丢包的本机回环环境下，基本不存在重传，差异主要来自时延和调度开销，而不是分组丢失。

这种现象与“理论上窗口越大越容易填满链路、吞吐量越高”的直觉看起来相反，原因主要与实验环境和实现细节有关：

从理论上看，增大接收窗口可以容纳更多在途数据，在存在明显带宽–时延积的真实网络中有利于提高吞吐量。**而本机回环环境的带宽–时延积非常小**。

但在本机回环环境下，RTT 极小、带宽–时延积也非常小，当窗口达到 8 个分组左右时，就已经足以“填满”回环链路；

此后继续增大窗口，发送端会在拥塞窗口增长的驱动下产生更大的发送突发，导致用户态与内核中出现更多短时排队和上下文切换，从而使平均 RTT 增大、整体传输时间变长，吞吐量反而被拉低。

因此，这组实验结果可以理解为：**在本机回环这种极低时延环境中，小窗口已经足够大，过大的接收窗口不会进一步提升吞吐量，反而因为突发性和调度开销导致 RTT 上升、吞吐量下降**。

同时，**窗口过大反而引入额外排队与调度抖动**。当允许的接收窗口从 8 增大到 32、64 时，发送端在拥塞窗口增长的驱动下会一次性发送更大的报文突发，造成用户态与内核缓冲区内更多“在途分组”排队。在真实广域网中这有利于充分利用带宽，但在本机回环中，**链路本身几乎没有瓶颈**，多出来的突发只会让内核和应用线程在处理这些数据和 ACK 时产生更多上下文切换和短暂阻塞，于是表现为平均 RTT 增大、整体传输时间变长。这一结论更多反映了“本机回环 + 用户态协议实现”的特性，而不是一般意义上“窗口越大吞吐量越高”的普适规律。

(三) 丢包对性能的影响

1. 实验数据（窗口大小 64，延迟时间 8ms，本机回环）

我们用测试集中的第一个图片进行验证。

丢包率	平均吞吐量（MB/s）	重传次数	总发送次数	实际丢包率
0%	0.0527	0	1858	0
5%	0.0580	93	1951	4.76679%
10%	0.0452	194	2052	9.45419%
20%	0.0235	446	2304	19.3575%

表 3: 不同丢包率下 Reno 拥塞控制的性能表现

2. 结果分析

从表 3 可以观察到：随着配置丢包率从 0% 提升到 20%，吞吐量显著下降，而重传次数与总发送次数明显上升；同时，“实际丢包率”与配置值基本一致，说明随机丢包注入生效且统计稳定。

(1) 实际丢包率的含义与计算方式 本实验中的“实际丢包率”并非直接测得链路层丢弃，而是基于发送端统计近似估计：

$$p_{\text{loss,real}} \approx \frac{N_{\text{retrans}}}{N_{\text{sent}}}.$$

其中 N_{retrans} 为数据包重传次数，N_{sent} 为数据包总发送次数（包含重传）。代入表中数据可验证该估计与配置丢包率高度吻合：

$$\frac{93}{1951} \approx 4.77\%, \quad \frac{194}{2052} \approx 9.45\%, \quad \frac{446}{2304} \approx 19.36\%.$$

这表明每次随机丢包通常会导致一次额外的数据包重传，因此该比例可以作为丢包强度的有效近似指标。

(2) 吞吐量随丢包率升高而显著下降的原因 当丢包率升高时，吞吐量由 0.0580 MB/s（5%）下降到 0.0452 MB/s（10%）以及 0.0235 MB/s（20%），主要原因包括：

- **重传开销增加：**丢包会引发重传，导致发送端需要发送更多的报文才能完成同一文件传输，其中一部分发送用于“补齐缺失数据”，无法带来新的有效进展，从而拉低有效吞吐量。
- **拥塞控制导致窗口收缩：**丢包被视为拥塞信号，会触发 Reno 的窗口回退（包括快重传/快恢复或超时回退），使 `cwnd` 在较高丢包率下长期维持在更小水平，限制了在途并行分组数量，降低链路利用率。

(3) 总发送次数与重传次数的变化趋势 随着丢包率上升，N_{retrans} 与 N_{sent} 均明显增大（例如 5%：1951 次发送/93 次重传；20%：2304 次发送/446 次重传）。这符合直觉：丢包越多，协议为保证可靠交付需要进行更多补发，从而提升总发送次数并进一步压低吞吐。

(4) 关于 0% 基线数据的可比性说明 在表 3 中，0% 行的吞吐量（0.0527 MB/s）与总发送次数（1858）与 5%/10%/20% 三组处于同一数量级，说明我们的各组实验在**相同测试文件与相同链路参数**下进行，仅改变了丢包率配置，因此横向对比是成立的。

0% 与 5% 的吞吐量出现了小幅波动（0.0527 → 0.0580 MB/s），这属于随机丢包/计时统计的正常扰动（例如运行时抖动、ACK/SACK 触发节奏差异、系统调度与计时分辨率等）。

结论 丢包率升高会显著增加重传开销并触发更频繁的拥塞窗口回退，使得有效吞吐量快速下降；同时，实验统计显示“实际丢包率”与配置值一致，说明随机丢包模拟与发送端统计方法可靠。

十二、 程序功能总结与心得体会

(一) 程序功能总结

本次实验在 Windows 下使用 C++ 和 WinSock2，在 UDP 之上实现了一个简化的可靠传输协议（RUDP）。主要功能如下：

- 程序分为发送端和接收端两种模式，通过命令行参数选择 `send` 或 `recv`。
- 在 UDP 之上自定义了分组头部，包含序号、确认号、窗口大小、校验和和标志位等。
- 实现了类似 TCP 的三次握手建立“连接”，以及四次挥手关闭“连接”。
- 使用滑动窗口一次发送多个未确认分组，提高链路利用率。
- 在发送端实现了简化的 Reno 拥塞控制，包括慢启动、拥塞避免和三次重复 ACK 触发的快速重传。
- 支持 SACK（选择确认），可以精确告知发送端哪些分组已经到达，减少不必要的重传。
- 对报文计算 16 位校验和，在接收端重新校验，保证数据正确。
- 通过参数设置链路的延迟和丢包率，在本机环境中模拟复杂网络条件。
- 发送端统计总发送分组数、重传次数、平均 RTT 和吞吐率等指标，用于简单性能分析。

(二) 知识点与收获

通过本次实验，我对传输层协议和网络编程有了更加直观的理解，主要收获如下：

- 进一步理解了 UDP 和 TCP 的区别，体会到 TCP 为了实现可靠传输需要做的大量工作。
- 熟悉了 WinSock2 的基本使用流程，包括初始化、创建套接字、`sendto`/`recvfrom` 调用以及超时设置等。
- 对三次握手和四次挥手的过程有了更清晰的认识，不再只是记住步骤，而是理解其作用。
- 理解了滑动窗口的具体实现方法，知道了如何通过维护 `base` 和 `next` 来控制未确认分组的数量。
- 掌握了基本的超时重传思想，以及如何通过记录发送时间和 ACK 时间来估计 RTT。
- 通过自己写 Reno 拥塞控制代码，加深了对慢启动、拥塞避免、快速重传和快速恢复的理解。
- 初步体会了 SACK 的优势：在乱序和丢包环境下，可以明显减少多余重传，提高效率。
- 学会了用程序模拟不同网络环境（延迟、丢包），并通过统计结果简单分析协议性能。

总体来说，本次实验让我从“使用者”的角度转变为“实现者”的角度，更深入地理解了可靠传输协议的设计思想和实现细节。