南開大學
Nankai University

计算机学院

计算机网络实验报告

实验 2：设计可靠传输协议并编程实现

课程：计算机网络

姓名：付家权

学号：2312236

专业：计算机科学与技术

25th December 2025

Contents

1 实验要求

1.1 实验任务概述

本实验要求在 UDP（数据报套接字）之上，在用户态实现面向连接的可靠传输，完成单向文件传输：发送端读取本地文件并分段发送，接收端按序重组并写入输出文件。虽然数据只从发送端流向接收端，但为了保证“可靠”，控制信息（ACK、SACK、握手、关闭等）必须双向交互。

1.2 实现功能

协议需要覆盖以下能力，并在报告中说明设计理由与实现方法：

连接管理 建立连接、关闭连接，以及异常场景下的处理（重试、放弃、复位）。

差错检测 使用校验和检测报文在传输过程中是否被破坏；校验失败的报文不能当作有效数据处理。

确认重传 支持流水线方式发送，采用选择确认（SACK），让发送端能更快定位丢失段并重传。

流量控制 发送窗口和接收窗口使用相同的固定大小窗口，避免接收端缓存被冲垮。

拥塞控制 实现 Reno，让发送端在“网络拥塞/丢包”时能够收敛窗口而不是无限重传。

2 实验原理 & 背景知识

2.1 在 UDP 上实现可靠传输

UDP 的特点是“尽力而为”：它把应用写入的数据当作一个个独立的数据报发送出去，网络层与链路层可能会导致**丢包、乱序、重复、损坏**。在局域网里这些问题不一定频繁出现，但一旦引入丢包/延时模拟，它们就会变得很明显。文件传输如果直接用 UDP，每丢一个包就会出现“文件缺一块”，而且接收端也不知道缺在哪里、更不知道该补什么。

所谓“可靠传输”，本质上是把 UDP 当作一条不可靠的管道，在应用层自己补齐三件事：**如何发现缺失（确认机制）、如何补回缺失（重传机制）、如何保持双方状态一致（连接管理）**。

2.2 按序到达：序号、累计确认与 SACK

为了能把分段后的数据重新拼回原始文件，每个数据段必须携带**序号**。接收端维护一个“我下一段最想要的序号”（记为 `expected`）。如果收到了 `expected`，就可以把它交付给上层并把 `expected` 往前推；如果收到了比 `expected` 更大的序号，就说明出现了乱序或丢包，可以先将这个包缓存起来。

最朴素的确认机制是**累计 ACK**：接收端在 ACK 中告诉发送端“我下一段希望收到的是多少”。累计 ACK 的优点是简单；缺点是当网络乱序时，接收端可能已经拿到了很多后续段，但仍然只能反复回“我还在等某一段”，发送端不知道“后面的到底到了没有”，可能会将“后面已经到达并在接收端缓冲之后的数据”重新发送，造成网络资源的浪费。

本实验要求支持**选择确认（SACK）**：除了累计 ACK 之外，接收端还要额外告诉发送端“在我已经收到的那些乱序段里，哪些段已经到了”。这里我们用一个 32 位的位图来表达：以累计 ACK 为基

准，后面 32 个序号每个用 1 bit 表示是否到达。这样发送端就能把“已经到达但还没能按序交付”的段标记出来，只针对真正缺失的段做重传，减少无谓的重复发送。

2.3 流水线与滑动窗口：实现传输连续

如果发送端每发送一个段都要等一个 ACK 才继续发送下一个段，效率会很低。而滑动窗口的思路是：允许发送端在未收到确认之前先发送多个段，只要“未确认段的数量”不超过窗口上限。窗口向前移动的时刻发生在：接收端确认了更靠前的数据，发送端就可以释放旧段并发送新段。

在**流量控制**部分：接收端的缓存能力是有限的，如果发送端窗口太大，接收端可能来不及缓存乱序段甚至来不及写磁盘，因此需要双方使用相同的固定窗口大小，并在 ACK 中显式通告窗口大小，保证发送端不越界发送，同时也能控制接收端接收的数据不会超出缓冲区。

2.4 拥塞控制

流量控制解决的是“接收端会不会被撑爆”，拥塞控制解决的是“网络会不会被我压垮”。即使接收端窗口很大，如果网络中间已经丢包严重，继续加大发送只会让队列更满、丢包更多，最后吞吐反而下降。

Reno 的核心思想是：“把丢包当作网络拥塞信号”，具体的做法是：维护一个**拥塞窗口 `cwnd`**，把它当作“我在网络里同时允许悬而未决的数据量”。当网络表现良好（不断收到新 ACK）时逐步增大 `cwnd`；当出现丢包迹象（超时或大量重复 ACK）时就迅速减小 `cwnd`。最终发送端的实际可用窗口取多个约束的最小值：本地固定窗口、对端通告窗口、以及 `cwnd`。

2.4.1 拥塞避免:TCP Reno 算法

Reno 是经典的 AIMD 拥塞控制：在网络不拥塞时**加性增大**发送速率，在出现拥塞信号时**乘性减小**发送速率，从而形成“锯齿状”稳定振荡并趋于公平。Reno 通常维护三个核心状态变量：拥塞窗口 `cwnd`、慢启动阈值 `ssthresh`、重复 ACK 计数 `dupAckCount`。发送端的实际可发送上限由

$$\texttt{send\_cap} = \min(\texttt{rwnd}, \texttt{cwnd})$$

决定，其中 `rwnd` 是接收端通告窗口（流量控制），`cwnd` 反映网络可承载能力（拥塞控制）。在实现上 `cwnd` 可以以 MSS 为单位（字节）计量；在这里我们以“段”为单位近似 MSS，因此用“段数”来理解 `cwnd` 。

(1) 慢启动：指数探测可用带宽 连接刚建立时不知道网络能承载多少在途数据，Reno 从很小的 `cwnd` 开始（典型初值为 1 MSS 或若干 MSS），每收到一个“新 ACK”（确认号前进）就做一次增长：

$$\texttt{cwnd} \leftarrow \texttt{cwnd} + 1$$

这使得一个 RTT 内大约会收到 `cwnd` 个 ACK，从而 `cwnd` 在每个 RTT 近似翻倍（指数增长）。当 `cwnd` 增长到阈值 `ssthresh` 时，认为继续指数增长会过于激进，进入拥塞避免阶段。

（2）拥塞避免：加性增大（Reno 的“线性增长”） 拥塞避免的目标是：在不引发明显排队溢出/丢包的前提下缓慢提高发送速率。Reno 在 $\texttt{cwnd} \geq \texttt{ssthresh}$ 时采用加性增大：每收到一个新 ACK，只

做一个很小的增量，使得一个 RTT 内 cwnd 总共只增加约 1 MSS。常见等价写法为：

$$\texttt{cwnd} \leftarrow \texttt{cwnd} + \frac{1}{\texttt{cwnd}}$$

从“段”为单位理解时，这表示：窗口越大，每个 ACK 带来的增量越小；累计一个 RTT 后总增量约为 1 段，从而形成线性增长。相比慢启动，拥塞避免更加保守，降低了把网络队列瞬间打满的概率。

（3）快速重传：用三次重复 ACK 作为“早期丢包信号” 当接收端持续回同一个累计 ACK，通常说明出现了一个缺口段未到达，但后续段可能已经到达并被缓存。Reno 将 **3 次重复 ACK** 视为丢包的早期信号：不等待超时，立即重传缺口段（Fast Retransmit）。同时执行乘性减小：把阈值设为当前窗口的一半

$$\texttt{ssthresh} \leftarrow \max(2,\ \texttt{cwnd}/2)$$

并进入快速恢复阶段。

（4）快速恢复：在“半窗”附近维持管道不空 Reno 的关键改进之一是：三次重复 ACK 通常意味着网络仍在持续交付数据（ACK 在到来），因此不必像超时那样把窗口退回到 1。Reno 在进入快速恢复时通常设置：

$$\texttt{cwnd} \leftarrow \texttt{ssthresh} + 3$$

其中“+3”对应已观测到的 3 个重复 ACK，表示网络中大概率还有 3 个段在路上。此后每额外收到一个重复 ACK，可令 $\texttt{cwnd} \leftarrow \texttt{cwnd} + 1$，并在允许时继续发送新段，以维持“在途数据”不至于骤降。直到收到一个**新 ACK**（确认号真正前进，说明缺口段已经被补齐并交付），Reno 退出快速恢复并把窗口收敛回：

$$\texttt{cwnd} \leftarrow \texttt{ssthresh}$$

然后回到拥塞避免阶段继续线性增长。

（5）超时：更强的拥塞信号与回到慢启动 若发生重传超时，Reno 认为拥塞更严重（可能队列长期溢出或路径中断），因此采取更激进的退让：同样把 ssthresh 设为 cwnd/2，但把 cwnd 直接重置为 1（回到慢启动），并重新用指数增长探测可用带宽。直观上，“超时”比“三次重复 ACK”意味着更差的网络条件，因此恢复策略也更保守。

（6）总结 Reno 的“线性增大 + 乘性减小”会让 cwnd 在某个区间内上下摆动：当网络可用资源变多时，窗口逐步爬升；一旦出现拥塞信号就立刻减半并重新爬升。这种机制能在共享链路上形成一定的公平性，也能让吞吐在丢包/延迟变化时自动收敛到合适的水平。本实验在 UDP 之上实现 Reno，本质是在应用层复现这套基于 ACK 反馈调节发送速率的控制回路。

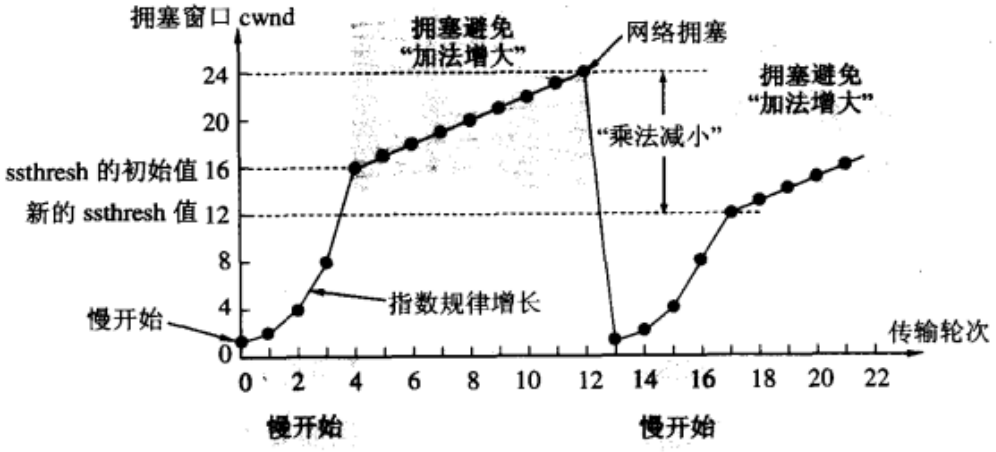


Figure 1: Reno 拥塞控制

2.4.2 RTO 估计：Jacobson/Karels 算法与 Karn 策略

超时重传的关键在于：**RTO（Retransmission Timeout）必须 "既不过小也不过大"**。RTO 太小会导致大量不必要的超时重传，引发拥塞控制误判并恶化队列；RTO 太大又会让真正丢包时恢复过慢。TCP 经典做法是用 RTT 采样估计 "平滑 RTT" 与 "RTT 波动"，再由它们计算一个动态 RTO。

(1) RTT 采样与平滑：SRTT 发送端在发出某个数据段后记录时间戳，收到能确认该段的 "新 ACK" 时可得到一个 RTT 样本 RTT（毫秒）。为了避免 RTT 受瞬时抖动影响过大，Jacobson/Karels 采用指数加权移动平均（EWMA）得到平滑 RTT：

$$\texttt{SRTT} \leftarrow (1-\alpha)\cdot \texttt{SRTT} + \alpha \cdot \texttt{RTT}, \quad \alpha = \frac{1}{8}$$

首次采样时通常直接初始化 SRTT ← RTT。

（2）RTT 波动估计：RTTVAR 仅有 SRTT 仍不足以决定 "要预留多少安全边际"。网络抖动越大，RTO 应该越保守。Jacobson/Karels 同时维护 RTT 波动（方差的近似）：

$$\texttt{RTTVAR} \leftarrow (1-\beta)\cdot \texttt{RTTVAR} + \beta \cdot |\texttt{RTT} - \texttt{SRTT}|, \quad \beta = \frac{1}{4}$$

首次初始化常用 RTTVAR ← SRTT/2，相当于先给出较大的不确定性，再随着更多样本收敛。

（3）RTO 的计算公式与裁剪 有了 SRTT 与 RTTVAR 后，RTO 的经典形式为：

$$\texttt{RTO} \leftarrow \texttt{SRTT} + 4 \cdot \texttt{RTTVAR}$$

其中系数 4 提供了对抖动的容忍度。工程实现通常还会对 RTO 做上下限裁剪，避免极端值：设定 $\texttt{RTO} \in [\texttt{MIN\_RTO}, \texttt{MAX\_RTO}]$，并给一个合理初值（本项目初始 RTO=1000ms，裁剪为 $\texttt{MIN\_RTO} = 20ms$、$\texttt{MAX\_RTO} = 60s$）。

（4）Karn 策略：重传段不参与 RTT 测量 如果一个段发生了重传，那么当 ACK 到达时无法确定它对应 "原始发送" 还是 "重传发送"，此时用该 RTT 样本更新 SRTT 会产生偏差。Karn 策略要求：

对发生过重传的段，不用其 ACK 来更新 RTT 估计。实现上可以为每个段维护“是否重传过”的标志，只对未重传且已发送的段采样 RTT。

（5）超时后的指数退避：避免持续加压 发生超时通常意味着网络已经拥塞或路径质量突然变差，因此除了触发重传，RTO 还应进行指数退避：

$$\texttt{RTO} \leftarrow \min(2 \cdot \texttt{RTO},\ \texttt{MAX\_RTO})$$

这能避免在拥塞时“越重传越拥塞”的恶性循环，并与 Reno 的窗口收缩共同作用，使发送端更快收敛到网络可承载的发送速率。

2.4.3 窗口探测机制

当接收端缓存/磁盘写入跟不上时，可能会在 ACK 中通告 `rwnd` $= 0$（零窗口），要求发送端暂停发送数据。若发送端完全依赖“对端主动发窗口更新”来恢复发送，会存在一个经典死锁风险：接收端窗口恢复为非零后发送了窗口更新 ACK，但该更新在网络中丢失；此后接收端在等待数据到来而不再主动发送 ACK，发送端又因为认为 `rwnd` $= 0$ 而一直不发数据，双方就会僵持。

Persist Timer 的目标就是打破这种僵持：当发送端观察到 `rwnd` $= 0$ 时进入 persist 状态，启动持续计时器；计时器到期后发送一个**窗口探测段**（window probe），其特点是载荷很小（TCP 常用 1 字节，或携带序号但不增加有效负载），目的不是传数据而是“强迫接收端回一个 ACK”，从而让发送端获知最新的窗口通告值。只要接收端窗口已恢复为非零，探测触发的 ACK 就会带回新的 `rwnd` > 0，发送端即可退出 persist 状态继续发送。

（1）探测触发条件与交互

- 进入条件：收到 ACK 后发现 `rwnd` $= 0$，发送端停止发送新数据段。
- 探测报文：发送端周期性发送窗口探测包（本项目发送一个空 payload 的 `ACK` 探测包）。
- 响应行为：接收端对探测包回 ACK，并在 ACK 的 `wnd` 字段携带当前可用接收窗口。
- 退出条件：发送端在后续 ACK 中看到 `rwnd` > 0，立即恢复正常发送并停止持续计时器。

（2）退避策略：避免探测过于频繁 在零窗口持续较久时，探测如果过于频繁会造成额外负担，因此 Persist Timer 常采用指数退避：探测间隔从一个初值开始逐步翻倍，并设定最大间隔上限。工程上常用类似

$$\texttt{interval} = \min(\texttt{base} \cdot 2^{k},\ \texttt{maxInterval})$$

的策略。本项目的窗口探测以 5s 起步，随后指数退避并封顶到 60s，从而在窗口长期为 0 的情况下也能持续“轻量唤醒”，同时避免持续发送探测包。

（3）与 RTO 的区别 RTO 面向的是“已经发送的数据段何时判定丢失并重传”，而 Persist Timer 面向的是“零窗口时仍要周期性探测对端窗口是否恢复”。因此 Persist Timer 不依赖于在途数据是否存在，也不应因为缺少 ACK 而永久停止；它的作用是保证流量控制在极端情况下也能自恢复。

3 协议设计 & 说明

3.1 报文格式设计

本项目定义了一个紧凑对齐的报文首部 PacketHeader（见 include/rtp.h），并统一采用网络字节序进行传输。首部的设计遵循一个朴素原则：**接收端能靠首部信息判断“这段是什么、应该放哪里、我该回什么 ACK”**，发送端也能靠首部信息判断“哪些已经确认、哪些需要重传、窗口是否该调整”。因此首部既要能描述数据（序号、长度），也要能描述控制（标志位、窗口、选择确认），同时还必须能检错（校验和）。

Table 1: PacketHeader 字段定义（网络字节序、紧凑对齐）

字段	位宽	说明
seq	32	发送序号（以“段”为单位）。数据段为 isn+seg_id，seg_id 从 1 开始。
ack	32	累计确认号，表示接收端期望的下一个有序段序号（绝对序号）。
flags	16	标志位：SYN/ACK/FIN/DATA/RST，用于区分控制与数据报文类型。
wnd	16	接收端通告窗口大小（段）。本项目窗口上限与 SACK 位宽一致，最大为 32。
checksum	16	16 位反码校验和校验和，覆盖首部与 payload，计算时该字段置 0。
len	16	payload 长度（字节），用于解析与一致性校验。
sack_mask	32	32 位 SACK 位图：位 i 表示 ack $+1+i$ 是否已到达接收端缓存。

Table 2: 标志位定义

标志	取值	说明
FLAG_SYN	0x01	建立连接请求/响应。
FLAG_ACK	0x02	确认报文（携带累计 ACK 与 SACK 位图）。
FLAG_FIN	0x04	关闭连接请求/响应。
FLAG_DATA	0x08	数据段。
FLAG_RST	0x10	异常复位，强制终止连接。

字段含义用通俗的话解释如下：

seq 数据段序号（以“段”为单位递增）。发送端把文件切成很多段，每个段就是一个连续的序号。序号的存在，让接收端可以把乱序到达的数据放回正确位置。

ack 累计确认号，表示“我下一段最想收到的序号”。当接收端已经连续拿到了 1..k 段，就会把 ack 设为 k+1。

flags 标志位，用来区分“握手、确认、数据、关闭、复位”等不同类型的报文：SYN/ACK/DATA/FIN/RST。

wnd 接收端通告窗口大小（以段为单位）。实验要求发送/接收采用同样的固定窗口，这里我们把窗口上限限制为 32，以适配 32 位 SACK 位图的表达范围。

len 有效载荷长度（字节）。它用于校验“首部 +payload”的整体长度是否一致，避免解析出错。

sack_mask 32 位选择确认位图。它以 ack 为基准，告诉发送端：在 ack 之后的 32 个序号中，哪些段已经到达接收端缓存。

checksum 16 位校验和，覆盖首部和 payload。接收端通过它判断报文是否被破坏；如果校验失败，报文直接丢弃。

3.2 消息传输机制

本节从“消息如何在网络中流动”的角度，把协议拆成一条清晰的链路：建立连接 → 差错检测 → 滑动窗口与选择确认 → 超时重传 → 断开连接。与其把它当成很多零散规则，不如把它当作两个状态机（发送端与接收端）在不断交换少量控制信息，从而把 UDP 的不可靠性“围起来”，最终让文件在应用层表现得像一条可靠的字节流。

3.2.1 建立连接（三次握手）

UDP 不会替应用维护连接状态，因此在开始传输前双方必须先“对齐状态”。我们采用 TCP 的三次握手机制，三次握手可以理解成一次双向确认：发送端先发 SYN 表示“我要开始了”，其中携带发送端的起始序号（ISN）；接收端回 SYN|ACK 表示“我收到了你的开始请求，我也准备好了”，同时携带接收端的 ISN；最后发送端回一个 ACK 表示“我知道你已经准备好了”，到这里双方才进入同一个“连接”状态点。

在握手阶段可能会面临丢包的问题,因此需要超时重试:实现中握手超时为 HANDSHAKE_TIMEOUT_MS=8000ms，每次超时会重发对应控制报文，重试次数上限为 5。这样即使 SYN 或 SYN|ACK 偶尔丢失，也能在合理时间内恢复；如果多次重试仍失败，则认为对端不可达或环境异常，连接建立失败并退出。

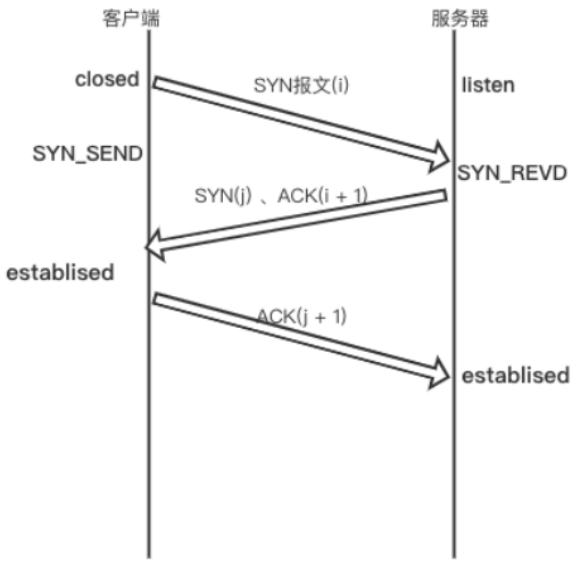


Figure 2: 三次握手示意图

3.2.2 差错检测（反码校验和）

可靠传输首先要保证“收到的数据本身没坏”。本项目使用 16 位反码校验和覆盖首部与 payload：发送端在发送前计算校验和并填入首部；接收端收到后重新计算整包校验，如果结果不为 0，则说明

报文在传输中被破坏。

当校验失败的时候，我们将采取相应的处理策略：将这个包直接丢弃，我们将其作为超时来处理。

3.2.3 滑动窗口与选择确认

接收端：累计 ACK + SACK 位图 在接收端维护一个最核心的变量 `expected`（即 `ack` 的含义）：它表示“我下一段最希望收到的序号”。当接收端收到了 `expected` 段，就能把它交付给文件并把 `expected` 往前推进到下一个 `expected` 缺失的位置；如果收到了比 `expected` 更大的段，说明出现了乱序或丢包，接收端会把它暂存到窗口缓存中等待缺口被补齐。

为了让发送端准确知道“缺了哪段、哪些段其实已经到了”，接收端的每个 ACK 都携带两部分信息：累计 ACK（当前 `expected`）与 32 位 `sack_mask`。位图以累计 ACK 为基准，向后覆盖 32 个段：哪一位为 1，就表示对应的乱序段已经在接收端缓存中。接收端同时在 ACK 中通告固定窗口大小 `wnd`，让发送端不越窗发送。

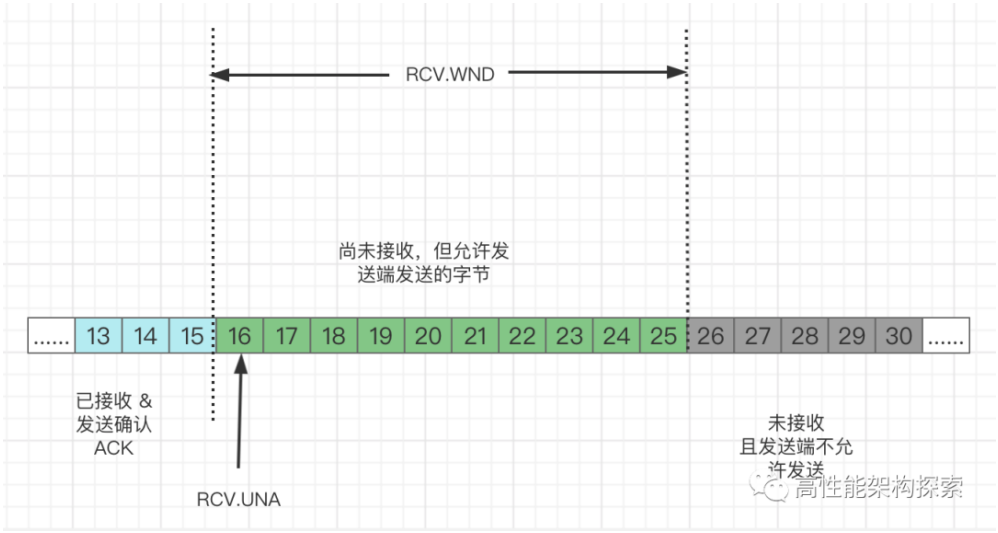


Figure 3: 接收端滑动窗口

发送端：滑动发送窗口 + 结合 SACK 的选择性补发 发送端维护一个发送窗口，窗口左边界是“最早未确认段”（`base`），右边界由可用窗口上限决定。只要 `next` 还没超过窗口允许的范围，发送端就持续把新段发出去，这就是我们要实现的流水线发送。收到 ACK 后，发送端会把累计 ACK 之前的段标记为已确认并滑动左边界；同时解析 `sack_mask`，把已被选择确认的段也标记为“无需重传”。

这样一来，发送端面对乱序时不需要盲目重发一大串段：它会优先补发“真正缺失”的那几段。为了避免丢包环境下过度重传，本项目还对“按 SACK 补发缺口”设置了节流：每个 ACK 最多补发若干段，并且同一缺口段两次补发之间需要有最小时间间隔。

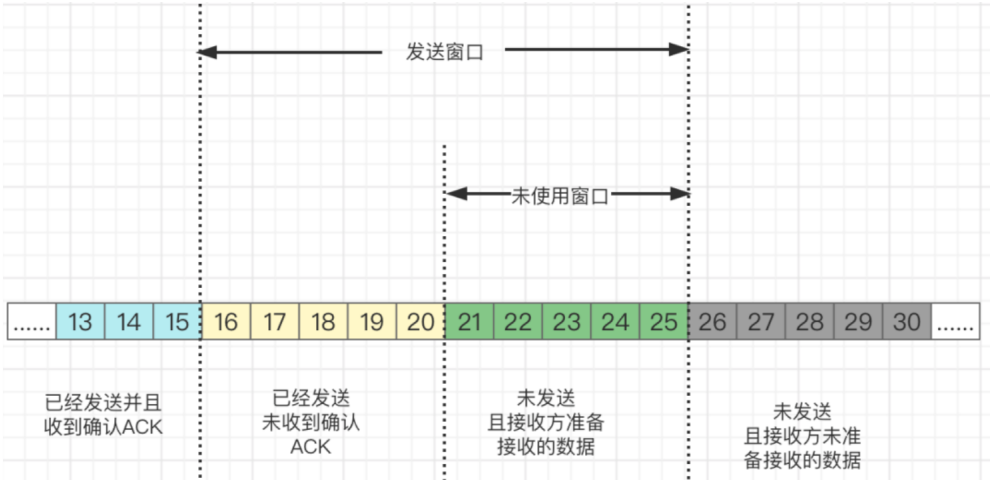


Figure 4: 发送端滑动窗口

3.2.4 发送上限

本实验要求发送窗口与接收窗口使用相同的固定大小窗口，这是流量控制的核心：它保证接收端缓存与写文件能力不会被发送端压垮。但网络本身也会“变慢”，因此还需要拥塞控制窗口 `cwnd` 来反映网络承载能力。

综合起来，发送端在任意时刻允许悬而未决（已发未确认）的段数不超过：

$$\min(\text{本地固定窗口}, \text{对端通告窗口}, \lfloor \texttt{cwnd} \rfloor, 32).$$

这条规则能解释我们在实验里看到的现象：窗口设置较大时流水线更深、吞吐往往更高；丢包变多时 `cwnd` 会回退，即使固定窗口很大，发送也会变得更“保守”。

3.2.5 超时重传（以及快速重传的触发信号）

由于 UDP 不保证交付，发送端必须能在“某段迟迟没有被确认”时主动补发。最直接的做法是超时重传：我们在发送端为每个已发送未确认段记录最后发送时间，一旦超过数据超时阈值仍未被确认，就重发该段，并把它计入重传统计。

除了等待超时，我们设计的协议采用 RENO 算法，把 ACK 的重复现象作为“丢包早期信号”：当接收端一直回同一个累计 ACK，说明缺口段迟迟没到，但后续段可能已经到达并被缓存。这时发送端如果观察到足够多的 3 个重复 ACK，就可以不等超时直接重发缺口段，也就是进入快速重传，并配合拥塞控制进行窗口调整。与此同时，如果长时间收不到任何 ACK，发送端还会触发全局超时（30 秒）并放弃连接；若某个段的重传次数超过上限（15 次），也会认为连接不可恢复而终止。

3.2.6 断开连接（四次挥手）

关闭连接看起来只是“发一个结束标志”，但在不可靠网络里必须把“结束”也做成可靠消息。四次挥手的核心思想是把两个方向的关闭分开确认：一端说“我不再发送数据”（`FIN`），另一端先确认“我知道了”（`ACK`），然后再在自己准备好时说“我也不再发送”（`FIN`），最后对方再确认一次（`ACK`）。这样做的本质是让双方对“结束”达成一致，而不是靠猜。

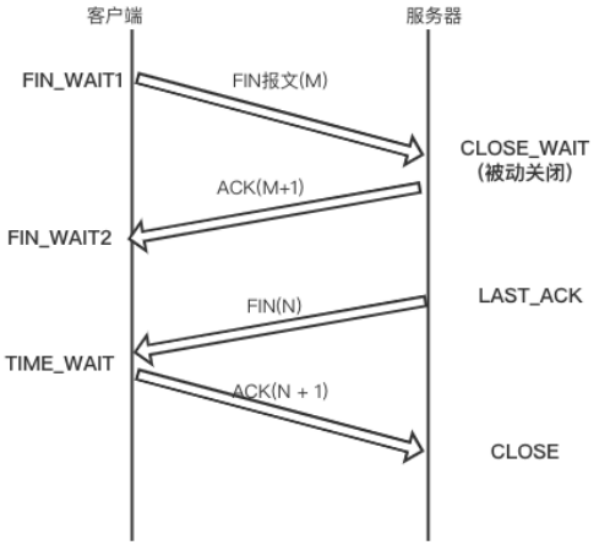


Figure 5: 四次握手示意图

本实验的数据传输是单向的，接收端没有反向应用数据要发送，因此实现上做了一个工程化合并：接收端在收到发送端的 FIN 后，通常用一个 FIN|ACK 报文同时表达“确认你结束”与“我也准备结束”，将四次挥手的两步合并成一步，但**概念上仍然遵循四次挥手的含义**。挥手阶段同样需要超时重试与重试上限，避免 FIN 或确认丢失导致一端永远等待。

3.2.7 发送端状态与接收端状态

把协议实现成状态机，是为了让每个收到的消息“都有明确解释”。发送端从 CLOSED 出发，发送 SYN 后进入“等待握手确认”的状态；握手成功后进入 ESTABLISHED 并开始发送数据；当所有数据都被确认后进入“关闭中”状态并发送 FIN，直到收到对端的关闭确认才回到可退出的终态。接收端则从 LISTEN 开始等待 SYN，回 SYN|ACK 后进入“等待最终确认”的状态；握手完成后进入 ESTABLISHED 接收数据并持续回 ACK/SACK；当收到 FIN 后进入“关闭握手”阶段，重试发送 FIN|ACK 并等待最终 ACK，最终退出。

Table 3: 发送端状态机摘要（关键事件）

状态	触发/事件	动作与转移
CLOSED	启动程序	创建 socket、绑定端口、生成 ISN，发送 SYN → SYN_SENT。
SYN_SENT	收到 SYN|ACK / 超时	收到则回 ACK → ESTABLISHED；超时重试，超过上限则失败退出/复位。
ESTABLISHED	收到 ACK/SACK	处理累计 ACK 与 SACK，必要时快速重传/缺口重传，更新 cwnd 与窗口边界。
ESTABLISHED	段超时	重传超时段，cwnd 退回慢启动。
FIN_SENT	收到 FIN|ACK / 超时	收到则回最终 ACK 并退出；超时重发 FIN，超过上限则放弃。

Table 4: 接收端状态机摘要（关键事件）

状态	触发/事件	动作与转移
LISTEN	收到 SYN	记录对端端点与对端 ISN，回 SYN|ACK → SYN_RCVD。
SYN_RCVD	收到 ACK 或首个 DATA	握手完成，初始化 expected=peer_isn+1 → ESTABLISHED。
ESTABLISHED	收到 DATA	窗口内缓存、提取连续段写文件、回 ACK+SACK。
ESTABLISHED	收到 FIN	回 FIN|ACK（突发重试）并短等待最终 ACK → 退出。

4 功能实现 & 代码分析

4.1 公共协议层

协议公共层定义在 include/rtp.h 与 src/rtp.cpp：负责首部结构、序列化/反序列化、校验和、时间戳工具、以及端点比对等。发送端与接收端都通过它来“读懂”对方的报文。

对应的头文件为

```
    constexpr size_t MAX_PAYLOAD = 1460;// 最大有效载荷大小（与典型 MTU 匹配）
    constexpr uint16_t FLAG_SYN = 0x01;// SYN
    constexpr uint16_t FLAG_ACK = 0x02;// ACK
    constexpr uint16_t FLAG_FIN = 0x04;// FIN
    constexpr uint16_t FLAG_DATA = 0x08;// 数据段
    constexpr uint16_t FLAG_RST = 0x10;// RST，复位段，用于异常终止连接

    constexpr int HANDSHAKE_TIMEOUT_MS = 8000;^^I// 握手超时时间（毫秒）
    constexpr int DATA_TIMEOUT_MS = 5000;^^I^^I// 数据传输超时时间（毫秒）

#pragma pack(push, 1)
    struct PacketHeader {
        uint32_t seq;         // 32 位发送序号
        uint32_t ack;         // 32 位确认号
        uint16_t flags;       // 16 位标志位
        uint16_t wnd;         // 16 位接收窗口通告
        uint16_t checksum;    // 16 位校验和
        uint16_t len;         // 16 位有效载荷长度
        uint32_t sack_mask;   // 32 位 SACK 掩码
    };
#pragma pack(pop)

    // 数据包结构体
    struct Packet {
        PacketHeader header{};
        vector<uint8_t> payload;
    };
    uint16_t compute_checksum(const uint8_t* data, size_t len);
    vector<uint8_t> serialize_packet(const PacketHeader& header, const vector<uint8_t>& payload);
    bool parse_packet(const uint8_t* data, size_t len, Packet& out);
    uint32_t generate_isn(const sockaddr_in& local, const sockaddr_in& remote);
    uint64_t now_ms();
```

```
33      string addr_to_string(const sockaddr_in& addr);
34      bool same_endpoint(const sockaddr_in& a, const sockaddr_in& b);
```

公共协议层：报文格式、字节序与合法性校验 协议把首部定义为紧凑对齐的 PacketHeader（#pragma pack(push, 1)），避免编译器填充导致的跨平台不一致。发送端/接收端在发送前都会走 serialize_packet()：先把 seq/ack/flags/wnd/len/sack_mask 转成网络字节序（大端），将 checksum 置 0 后对“首部+payload”计算 16 位反码校验和，再把校验和写回首部。接收侧统一通过 parse_packet() 做入口校验：长度必须不小于首部、整包校验和必须为 0、首部 len 必须与实际 payload 长度一致；随后再把字段转回主机字节序。这样可以把“坏包/截断包/伪造长度包”挡在状态机之外，减少上层逻辑分支。

```
1   // 序列化
2   vector<uint8_t> serialize_packet(const PacketHeader& header, const vector<uint8_t>& payload) {
3       PacketHeader net = header;
4       // 转换为大端序
5       net.seq = htonl(header.seq);
6       net.ack = htonl(header.ack);
7       net.wnd = htons(header.wnd);
8       net.len = htons(header.len);
9       net.flags = htons(header.flags);
10      net.sack_mask = htonl(header.sack_mask);
11      // 校验和字段置 0 以便计算校验和
12      net.checksum = 0;
13      vector<uint8_t> buffer(sizeof(PacketHeader) + payload.size());// 分配缓冲区
14      memcpy(buffer.data(), &net, sizeof(PacketHeader));        // 复制头部
15
16      // 如果有有效载荷，复制有效载荷
17      if (!payload.empty()) {
18          memcpy(buffer.data() + sizeof(PacketHeader), payload.data(), payload.size());
19      }
20      // 计算校验和并填入头部
21      uint16_t cs = compute_checksum(buffer.data(), buffer.size());
22      auto* hdr = reinterpret_cast<PacketHeader*>(buffer.data());
23      hdr->checksum = htons(cs);
24      return buffer;
25  }
26
27  // 解析
28  bool parse_packet(const uint8_t* data, size_t len, Packet& out) {
29      if (len < sizeof(PacketHeader)) {
30          // 数据包长度小于头部长度，非法包
31          return false;
32      }
33      PacketHeader net{};
34      memcpy(&net, data, sizeof(PacketHeader));
35      if (compute_checksum(data, len) != 0) {
36          // 计算的校验和不为 0，校验失败
37          return false;
38      }
39
```

```
    // 转为小端序
    out.header.seq = ntohl(net.seq);
    out.header.ack = ntohl(net.ack);
    out.header.wnd = ntohs(net.wnd);
    out.header.len = ntohs(net.len);
    out.header.flags = ntohs(net.flags);
    out.header.sack_mask = ntohl(net.sack_mask);
    out.header.checksum = ntohs(net.checksum);

    if (out.header.len + sizeof(PacketHeader) != len) {
        // 长度字段与实际长度不符，非法包
        return false;
    }

    // 提取有效载荷
    out.payload.clear();
    if (out.header.len > 0) {
        // 复制有效载荷数据
        // 从 data 偏移 sizeof(PacketHeader) 开始，到偏移 len 结束
        out.payload.insert(out.payload.end(), data + sizeof(PacketHeader), data + len);
    }
    return true;
}

// 计算校验和
uint16_t compute_checksum(const uint8_t* data, size_t len) {
    uint32_t sum = 0;  // 32 位累加器
        size_t i = 0;
    while (i + 1 < len) {
        // 16 位字加法
        uint16_t word = static_cast<uint16_t>((data[i] << 8) | data[i + 1]);
        sum += word;
        // 处理溢出
        sum = (sum & 0xFFFF) + (sum >> 16);
        i += 2;
    }
    if (i < len) {
        // 剩余一个字节，补 0 处理
        uint16_t word = static_cast<uint16_t>(data[i] << 8);
        sum += word;
        sum = (sum & 0xFFFF) + (sum >> 16);
    }
    // 取反得到校验和
    return static_cast<uint16_t>(~sum);
}
```

公共协议层：ISN 生成与端点绑定 本项目用 `generate_isn(local, remote)` 基于四元组做哈希（FNV-1a），并混入一次性生成的 secret salt，再叠加 `now_ms()` 作为时间计数，得到每条“连接”的初始序号。握手成功后双方都用 `same_endpoint()` 来验证对端地址（IP+ 端口），之后所有数据/控制

包都先做端点过滤，避免被其他进程或旁路流量干扰。

```
uint32_t fnv1a(const uint8_t* data, size_t len) {
    constexpr uint32_t FNV_PRIME = 16777619u;  // FNV 素数
    uint32_t hash = 2166136261u;        // FNV 偏移基数
    for (size_t i = 0; i < len; ++i) {
        hash ^= static_cast<uint32_t>(data[i]);
        hash *= FNV_PRIME;
    }
    return hash;
}

uint64_t generate_salt_once() {
    std::random_device rd;
    return (uint64_t(rd()) << 32) ^ rd();
}
uint64_t secret_salt() {
    static const uint64_t salt = generate_salt_once();
    return salt;
}
// 生成 isn
uint32_t generate_isn(const sockaddr_in& local, const sockaddr_in& remote) {
    // 模仿 RFC 6528 基于地址和端口的哈希生成 ISN
    uint8_t tuple_buf[12] = {0};
    memcpy(tuple_buf, &local.sin_addr.s_addr,
           sizeof(local.sin_addr.s_addr));^^I// 4 本地的 IP 地址
    memcpy(tuple_buf + 4, &local.sin_port, sizeof(local.sin_port));// 2 本地的端口号
    memcpy(tuple_buf + 6, &remote.sin_addr.s_addr, sizeof(remote.sin_addr.s_addr));
    memcpy(tuple_buf + 10, &remote.sin_port, sizeof(remote.sin_port));

    uint64_t salt = secret_salt();// 获取全局盐值
    uint8_t salt_buf[8] = {0};// 8 字节盐值缓冲区
    memcpy(salt_buf, &salt, sizeof(salt_buf));
    uint32_t hash = fnv1a(tuple_buf, sizeof(tuple_buf));
    hash ^= fnv1a(salt_buf, sizeof(salt_buf));

    uint32_t counter = static_cast<uint32_t>(now_ms());
    return hash + counter;
}

uint64_t now_ms() {
    using namespace std::chrono;
    return duration_cast<milliseconds>(steady_clock::now().time_since_epoch()).count();
}

string addr_to_string(const sockaddr_in& addr) {
    // 转换 IP 地址和端口为字符串形式 inet_ntoa: 将网络字节序的 IP 地址转换为点分十进制字符串
    char* ip_str = inet_ntoa(addr.sin_addr);
    return string(ip_str) + ":" + std::to_string(ntohs(addr.sin_port));
}
```

```
49
50 bool same_endpoint(const sockaddr_in& a, const sockaddr_in& b) {
51     return a.sin_addr.s_addr == b.sin_addr.s_addr && a.sin_port == b.sin_port;
52 }
```

统计与输出：把性能指标与协议行为对齐 统计模块 TransferStats 统一记录重传次数、超时次数、快速重传次数以及数据阶段起止时间，并在 sender/receiver 结束时输出吞吐率与丢包率等指标；发送端额外用 report_progress() 根据“已确认字节数”实时打印进度，便于观察在丢包/延迟条件下窗口推进是否顺畅。

4.2 拥塞控制窗口

我们先看拥塞控制窗口的声明，这部分我们主要是为了实现拥塞控制 Reno 算法，我们知道 Reno 的主要分为

```
1  #pragma once
2
3  #include <algorithm>
4  #include <cstdint>
5
6  namespace rtp {
7      class CongestionControl {
8         public:
9          explicit CongestionControl(double initial_ssthresh = 64.0);
10
11         // 处理收到新 ACK
12         void on_new_ack();
13
14         // 处理收到 dupACK
15         void on_duplicate_ack();
16
17         // 检测到 3 个重复 ACK，触发快速重传
18         bool should_fast_retransmit() const;
19
20         //  进入快速恢复
21         void on_fast_retransmit();
22
23         // 超时事件处理
24         void on_timeout();
25
26         // 获取当前拥塞窗口大小
27         double get_cwnd() const { return cwnd_; }
28
29         // 获取慢启动阈值
30         double get_ssthresh() const { return ssthresh_; }
31
32         // 是否处于快速恢复状态
33         bool in_fast_recovery() const { return in_fast_recovery_; }
34
```

```
35          // 获取重复 ACK 计数
36          uint32_t get_dup_ack_count() const { return dup_ack_count_; }
37
38          // 重置重复 ACK 计数
39          void reset_dup_ack_count() { dup_ack_count_ = 0; }
40
41        private:
42          double cwnd_;^^I^^I^^I  // 拥塞窗口
43          double ssthresh_;^^I^^I  // 慢启动阈值
44          uint32_t dup_ack_count_;  // 重复 ACK 计数
45          bool in_fast_recovery_;^^I  // 是否处于快速恢复状态
46      };
47
48  }  // namespace rtp
49
```

当发送端接收到新的 ACK 的时候，根据当前状态执行慢启动或拥塞避免算法。在快速恢复中收到“新 ACK”表示丢失段已被修复，退出快速恢复，退出时将 cwnd 设为 ssthresh，等待后续 ACK 再进入拥塞避免的线性增长。如果不处于快速恢复阶段，就进入慢启动或者拥塞避免：在慢启动阶段，指数增长，每个 ACK 使 cwnd 加 1；在拥塞避免阶段，线性增长，每 RTT 增加 1 个 MSS，即每个 ACK 增加 1/cwnd，cwnd 个 ACK 后总共增加 1。

```
1  void CongestionControl::on_new_ack() {
2      dup_ack_count_ = 0;^^I // 重置重复 ACK 计数
3
4      if (in_fast_recovery_) {
5          cwnd_ = ssthresh_;
6          in_fast_recovery_ = false;
7          return;
8      }
9
10     // 非快速恢复：慢启动或拥塞避免
11     if (cwnd_ < ssthresh_) {
12         cwnd_ += 1.0;
13     } else {
14         cwnd_ += 1.0 / cwnd_;
15     }
16 }
```

当在接收端收到重复包的时候，先增加重复包计数。如果此时已经在快速恢复中，继续膨胀窗口，每个重复 ACK 代表一个离开网络的数据包，我们就可以继续发送新包，所以增加 cwnd 以允许发送新数据

```
1  void CongestionControl::on_duplicate_ack() {
2      dup_ack_count_++;
3
4      if (in_fast_recovery_) {
5          cwnd_ += 1.0;
6      }
7  }
```

当检测到 3 个重复 ACK，触发快速重传：

```
bool CongestionControl::should_fast_retransmit() const
{
    return dup_ack_count_ == 3 && !in_fast_recovery_;
}
```

然后我们看对执行快速重传后拥塞控制部分的处理，我们将降低阈值为 cwnd 的一半，但是不小于 4，ssthresh = cwnd/2, cwnd = ssthresh + 3, 增加 3 个 MSS 以应对网络中离开的包，进入快速恢复。

```
void CongestionControl::on_fast_retransmit() {

    ssthresh_ = std::max(4.0, cwnd_ / 2.0);
    cwnd_ = ssthresh_ + 3.0;
    in_fast_recovery_ = true;
}
```

我们来看对超时事件的处理，当有包超时的时候我们就进入慢启动阶段：将 ssthresh 置为 cwnd 的一半，将 cwnd 置为一，同时重置拥塞控制的状态。

```
void CongestionControl::on_timeout() {
    ssthresh_ = std::max(4.0, cwnd_ / 2.0); // 最小为 4
    cwnd_ = 1.0;

    dup_ack_count_ = 0;
    in_fast_recovery_ = false;
}
```

4.3 接收缓冲区

接下来就是接收缓冲区的实现，接收缓冲区主要负责乱序段缓存、连续段提取以及 SACK 掩码生成

```
class ReceiveBuffer {
    public:
    explicit ReceiveBuffer(uint16_t window_size);

    // 添加接收到的段到缓冲区
    bool add_segment(uint32_t seq, const vector<uint8_t>& data);

    //  提取所有连续的段
    vector<vector<uint8_t>> extract_continuous_segments();

    // 构建 SACK 掩码
    // 位 i=1 表示序号 expected_seq+1+i 的段已到达
    uint32_t build_sack_mask() const;

    // 获取期望序号
    uint32_t get_expected_seq() const { return expected_seq_; }

```

```
    // 设置期望序号
    void set_expected_seq(uint32_t seq) { expected_seq_ = seq; }

    // 检查序号是否在窗口内
    bool is_in_window(uint32_t seq) const;

    // 获取窗口大小
    uint16_t get_window_size() const { return window_size_; }

    private:
    uint32_t expected_seq_;
    uint16_t window_size_;
    map<uint32_t, vector<uint8_t>> buffer_;
};
```

在将新加入的段加入缓冲区之前会先检测这个段是否已经存在，如果存在直接跳过，如果不存在则加入。

```
bool ReceiveBuffer::add_segment(uint32_t seq, const vector<uint8_t>& data) {
    if (buffer_.find(seq) != buffer_.end()) {
        return false;
    }

    buffer_.emplace(seq, data);
    return true;
}
```

接下来就是从 expected_seq 开始提取所有连续的段，当提取之后就从缓冲区移除这个段，并推进 expected_seq；当缓冲区中不存在期望的序号的时候，就是出现的缺口，这时候停止提取。

```
vector<vector<uint8_t>> ReceiveBuffer::extract_continuous_segments() {
    vector<vector<uint8_t>> result;
    while (true) {
        auto it = buffer_.find(expected_seq_);
        if (it == buffer_.end()) {
            break;
        }
        result.push_back(std::move(it->second));
        buffer_.erase(it);
        expected_seq_++;
    }
    return result;
}
```

同时缓冲区还负责构建 SACK 掩码，构建 SACK 掩码（32 位），用来标记 expected_seq+1 起的 32 个段，当位 i=1 表示序号 expected_seq+1+i 的段已到达。就是将 expected_seq 之后已经到达的段所在的位置进行置位。

```
uint32_t ReceiveBuffer::build_sack_mask() const {
    uint32_t mask = 0;
    for (uint32_t i = 0; i < 32; ++i) {
```

```
        uint32_t seq = expected_seq_ + 1 + i;
        if (buffer_.find(seq) != buffer_.end()) {
            mask |= (1u << i);// 该段已到达，置位
        }
    }
    return mask;
}
```

4.4 发送窗口

我们来看发送窗口的实现，首先我们构造 SegmentInfo 结构题来将数据段以及相关的信息进行存储。

```
class SendWindow {
  public:
    // 数据段信息
    struct SegmentInfo {
        vector<uint8_t> data;          // 段数据
        bool sent{false};              // 是否已发送
        bool acked{false};             // 是否已确认
        uint64_t last_send{0};         // 最后发送时间（用于超时检测）
        uint64_t last_sack_retx{0};    // 最后 SACK 重传时间（避免频繁重传）
        int retrans_count{0};          // 重传次数（用于检测连接断开）
        uint64_t send_timestamp{0};    // 首次发送时间戳
        bool is_retransmitted{false};  // 是否被重传过（Karn 算法）
    };
    SendWindow();

    // 初始化分段
    void initialize(const vector<uint8_t>& file_data);

    // 标记段为已确认
    void mark_acked(uint32_t seq);

    // 获取段信息
    SegmentInfo& get_segment(uint32_t seq);

    // 检查所有段是否都已确认
    bool all_acked() const;

    // 获取已发送未确认段数量，发送数量 - 确认数量
    size_t inflight_count() const;

    // 获取总段数
    uint32_t total_segments() const { return total_segments_; }

    // 获取/设置基序号
    uint32_t get_base_seq() const { return base_seq_; }
    void set_base_seq(uint32_t seq) { base_seq_ = seq; }

```

```
    // 获取/设置下一个待发送序号
    uint32_t get_next_seq() const { return next_seq_; }
    void set_next_seq(uint32_t seq) { next_seq_ = seq; }
    // 推进下一个待发送序号
    void advance_next_seq() { next_seq_++; }

    // 推进 base_seq 到第一个未确认的段
    void advance_base_seq();

    // 计算实际窗口大小 取本地窗口、对端窗口、拥塞窗口、SACK 位宽的最小值
    size_t calculate_window_size(uint16_t local_window, uint16_t peer_window, double cwnd, size_t
    ↪  sack_bits) const;

  private:
    vector<SegmentInfo> segments_;// 所有数据段
    uint32_t total_segments_;// 总段数
    uint32_t base_seq_;// 窗口左边界（最小未确认序号）
    uint32_t next_seq_;// 下一个待发送序号
};
```

首先就是初始化分段，先计算总段数，为每段分配空间，接着将文件数据分段存储，最后初始化基序号和下一个待发送序号

```
void SendWindow::initialize(const vector<uint8_t>& file_data) {
    total_segments_ = static_cast<uint32_t>((file_data.size() + MAX_PAYLOAD - 1) / MAX_PAYLOAD);
    segments_.assign(total_segments_, {});

    for (uint32_t i = 0; i < total_segments_; ++i) {
        size_t start = i * MAX_PAYLOAD;
        size_t len = std::min<size_t>(MAX_PAYLOAD, file_data.size() - start);
        segments_[i].data.insert(segments_[i].data.end(), file_data.begin() + start,
                                 file_data.begin() + start + len);
    }

    base_seq_ = 1;
    next_seq_ = 1;
}
```

当我们收到一个 ACK 的时候，我们就要将段标记为已确认，当序号无效的时候直接忽略，接着我们先获取对应的段信息，并将其标记为已确认。

```
void SendWindow::mark_acked(uint32_t seq) {
    if (seq == 0 || seq > total_segments_) {
        return;
    }
    auto& seg = segments_[seq - 1];^^I // 序号从 1 开始
    if (!seg.acked) {
        seg.acked = true;
        seg.last_sack_retx = 0;
    }
}
```

最后就是推进窗口，将窗口左边界推进到第一个没有被确认的位置

```
void SendWindow::advance_base_seq() {
    while (base_seq_ <= total_segments_ && segments_[base_seq_ - 1].acked) {
        base_seq_++;
    }
}
```

4.5 接收端

接下来我将对代码进行详细介绍我们将发送端的整体流程按时间顺序理解会更清楚：首先我们先整理一下我们的整体的流程：

```
int ReliableSender::run() {
    WSADATA wsa;
    if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0) {
        return -1;
    }
    sock_ = socket(AF_INET, SOCK_DGRAM, 0);
    if (!socket_valid(sock_)) {
        return 1;
    }
    sockaddr_in local_addr{};
    local_addr.sin_family = AF_INET;
    local_addr.sin_addr.s_addr = INADDR_ANY;
    local_addr.sin_port = htons(local_port_);
    if (bind(sock_, reinterpret_cast<const sockaddr*>(&local_addr), sizeof(local_addr)) < 0) {
        return 1;
    }
    //^^I 设置要发送的目标地址
    remote_.sin_addr.s_addr = inet_addr(dest_ip_.c_str());
    if (remote_.sin_addr.s_addr == INADDR_NONE) {
        return 1;
    }

    // 生成初始序列号
    isn_ = generate_isn(local_addr, remote_);

    // 执行三次握手
    if (!handshake()) {
        return 1;
    }

    // 读取输入文件
    ifstream in(file_path_, std::ios::binary);
    if (!in) {
        return 1;
    }
    // 按字节读取整个文件到内存
    file_data_ = vector<uint8_t>((std::istreambuf_iterator<char>(in)),
    ↪  std::istreambuf_iterator<char>());
```

```
38
39      // 确保初始的时候对端窗口非零
40      peer_wnd_ = peer_wnd_ ? peer_wnd_ : window_size_;
41
42      // 初始化拥塞控制
43      congestion_ = CongestionControl(std::max<double>(2.0, peer_wnd_));
44
45      // 初始化全局计时器
46      last_ack_time_ = now_ms();
47
48      while (!fin_complete_) {
49
50          // 全局超时检测：长时间没有接 ACK 响应
51          if (now_ms() - last_ack_time_ > GLOBAL_TIMEOUT_MS) {
52              // 30s 无响应，认为连接断开，发送 RST
53              send_rst();
54              return 1;
55          }
56
57          // 发送数据、处理网络事件、处理超时和窗口探测
58          try_send_data();
59          process_network();
60          handle_timeouts();
61          handle_window_probe();
62
63          // 记录数据传输时间点
64          if (!data_timing_recorded_ && window_.all_acked()) {
65              // 如果全部都确认，记录结束时间
66              stats_.set_end_time(now_ms());
67              data_timing_recorded_ = true;
68          }
69
70          // 尝试发送 FIN 包
71          try_send_fin();
72
73          // 检查 FIN 重传次数，超过限制则放弃
74          if (fin_sent_ && !fin_complete_ && fin_retry_count_ >= MAX_FIN_RETRIES) {
75              break;
76          }
77      }
78
79      // 记录结束时间
80      if (!data_timing_recorded_) {
81          if (stats_.get_start_time() == 0) {
82              stats_.set_start_time(now_ms());
83          }
84          stats_.set_end_time(now_ms());
85      }
86      return !fin_complete_;
87  }
```

（1）初始化、序号空间与握手 发送端启动后先创建 UDP socket 并绑定本地端口（bind），再通过 generate_isn(local, remote) 为该四元组生成初始序号 isn_。我们这里的数据段 seq 使用**绝对序号**：第 k 个数据段的序号为 isn_ + k（见 transmit_segment() 中 hdr.seq = isn_ + seq）。因此接收端在握手后把 expected 设为 peer_isn + 1，双方在同一序号空间上“对齐”，后续 ACK/SACK 才有共同语义。

发送方握手采用主动三次握手：发送端构造 SYN（syn.seq=isn_）发送后等待 SYN|ACK，超时按 HANDSHAKE_TIMEOUT_MS=8000ms 重试，最多 MAX_HANDSHAKE_RETRIES=5 次（src/sender.cpp）。收到响应后会先做端点过滤（same_endpoint(from, remote_)），避免非预期地址的报文影响状态；若收到 RST 则立即失败退出；若收到合法 SYN|ACK 且 ack=syn.seq+1，则记录对端 ISN（peer_isn_）与对端通告窗口（peer_wnd_），并回最后一个 ACK 完成建链。

```
bool ReliableSender::handshake() {
    // 构建 syn 包
    PacketHeader syn{};
    syn.seq = isn_;
    syn.ack = 0;
    syn.wnd = window_size_;
    syn.len = 0;
    syn.flags = FLAG_SYN;
    syn.sack_mask = 0;

    // 主动发送 SYN 包，等待 SYN+ACK 响应，最多 5 次重试
    for (int attempt = 0; attempt < MAX_HANDSHAKE_RETRIES; ++attempt) {
        send_raw(syn, {});^^I// 发送 SYN 包
        Packet pkt{};
        sockaddr_in from{};

        // 等待 SYN+ACK 响应，超时重试
        if (!wait_for_packet(pkt, from, HANDSHAKE_TIMEOUT_MS)) {
            continue;
        }

        if (!same_endpoint(from, remote_)) {
            continue;
        }

        // 收到 RST 段，连接被对方重置
        if (pkt.header.flags & FLAG_RST) {
            return false;
        }

        // 收到 SYN+ACK 段，发送 ACK 完成握手
        if ((pkt.header.flags & FLAG_SYN) && (pkt.header.flags & FLAG_ACK) && pkt.header.ack ==
        ↪  syn.seq + 1) {
            peer_isn_ = pkt.header.seq;
            peer_wnd_ = pkt.header.wnd;

            PacketHeader ack{};
            ack.seq = isn_ + 1;
```

```
            ack.ack = peer_isn_ + 1;
            ack.flags = FLAG_ACK;
            ack.wnd = window_size_;
            ack.len = 0;
            ack.sack_mask = 0;
            send_raw(ack, {}); // 发送 ACK 包完成握手
            return true;
        }
    }
    // 握手失败
    send_rst();
    return false;
}
```

（2）文件分段与发送窗口初始化 发送端把输入文件按 MAX_PAYLOAD=1460 字节切片，每片对应一个段序号，从 1 开始递增。分段后的段信息存入 SendWindow::segments_，每段维护“是否发送、是否确认、最后发送时间、重传次数”等字段。这样做的意义是：只要段的状态还没变成“已确认”，发送端就永远知道它在哪里、是否要重传、重传过几次。

（3）发送窗口推进 窗口内部使用相对段序号（从 1 到 total_segments）：base_seq_ 表示当前最小未确认段，next_seq_ 表示下一个待发送新段（见 include/send_window.h）。发送端在 try_send_data() 中先计算“实际可用窗口上限”

$$\texttt{window\_cap} = \min(\texttt{window\_size\_}, \texttt{peer\_wnd\_}, \lfloor \texttt{cwnd} \rfloor, \texttt{SACK\_BITS})$$

然后持续发送 next_seq_ 对应的新段，直到窗口满或没有新段可发。这样做把流量控制（对端 wnd）与拥塞控制（cwnd）统一为同一条“发送上限”，并保证只发“未发送过”的新段，重传由其他路径触发。

```
void ReliableSender::try_send_data() {
    // 零窗口
    if (peer_wnd_ == 0) {
        return;^^I
    }

    size_t window_cap = window_.calculate_window_size(window_size_, peer_wnd_,
    ↪  congestion_.get_cwnd(), SACK_BITS);

    while (window_.get_next_seq() <= window_.total_segments() &&
           window_.get_next_seq() < window_.get_base_seq() + window_cap) {
        auto& seg = window_.get_segment(window_.get_next_seq());
        if (!seg.sent) {
            transmit_segment(window_.get_next_seq());
            window_.advance_next_seq();
        } else {
            break;
        }
    }
}
```

(4) ACK/SACK 驱动的可靠性收敛 发送端每次收到 ACK 都会进入 `handle_ack()`：先更新对端窗口通告并检测零窗口（进入/退出 `zero_window_`），再把 ACK 的绝对序号 `ack_abs` 转换为相对段序号 $\texttt{ack} = \texttt{ack\_abs} - \texttt{isn\_}$。如果 `ack` 推进了 `base_seq_`，则视为“新 ACK”并批量标记 $[\texttt{base}, \texttt{ack})$ 的段已确认，同时调用 `congestion_.on_new_ack()` 更新 Reno 状态；如果 `ack` 等于 `base_seq_`，则视为“重复 ACK”，累计到 3 次后触发快速重传：调用 `congestion_.on_fast_retransmit()` 并重传 `base_seq_` 对应段（`fast_retransmit()`）。

```
void ReliableSender::process_network() {
    Packet pkt{};
    sockaddr_in from{};
    if (wait_for_packet(pkt, from, 50)) {
        if (!same_endpoint(from, remote_)) {
            return;
        }
        if ((pkt.header.flags & FLAG_FIN) && (pkt.header.flags & FLAG_ACK)) {
            // 处理 FIN+ACK 包，完成连接关闭
            handle_fin_ack();
            return;
        }
        if (pkt.header.flags & FLAG_ACK) {
            // 如果收到 ACK 包，处理 ACK
            handle_ack(pkt);
        }
    }
}
void ReliableSender::handle_ack(const Packet& pkt) {
    ...
    peer_wnd_ = new_peer_wnd;

    // 处理 ACK
    uint32_t ack_abs = pkt.header.ack;
    if (ack_abs <= isn_) {
        return;
    }
    // 转换为相对序号
    uint32_t ack = ack_abs - isn_;
    if (ack > window_.get_base_seq()) {
        // 新 ACK，推进窗口
        handle_new_ack(ack);
    } else if (ack == window_.get_base_seq() && window_.get_base_seq() <=
    ↪  window_.total_segments()) {
        // 重复 ACK，处理快速重传
        handle_duplicate_ack(ack);
    }

    // 处理 SACK 掩码
    handle_sack(ack, pkt.header.sack_mask);
    window_.advance_base_seq();
}
```

```
42
43 void ReliableSender::handle_new_ack(uint32_t ack) {
44     // 测量 RTT (Karn 算法: 只测量未重传的段来更新 RTT)
45     ....
46     // 标记所有被确认的段
47     for (uint32_t s = window_.get_base_seq(); s < ack && s <= window_.total_segments(); ++s) {
48         window_.mark_acked(s);
49     }
50
51     window_.set_base_seq(ack);
52
53     congestion_.on_new_ack();
54 }
55
56 void ReliableSender::handle_duplicate_ack(uint32_t ack) {
57     congestion_.on_duplicate_ack();
58     if (congestion_.should_fast_retransmit()) {
59         congestion_.on_fast_retransmit();
60         fast_retransmit();
61     }
62 }
```

对于 SACK，接收端在 ACK 中携带 32 位位图，发送端在 handle_sack(ack, mask) 中将位图中为 1 的段直接标记为已确认；随后扫描位图中为 0 的位置，若该段“已发送未确认”则视为缺口段，并按限速策略选择性重传：单次 ACK 最多重传 MAX_SACK_RETX_PER_ACK=4 个缺口段，且同一段的 SACK 缺口重传间隔至少为 MIN_GAP_RETX_INTERVAL_MS，避免在高丢包/高乱序下被位图反复触发导致过度重传。

```
1  void ReliableSender::handle_sack(uint32_t ack, uint32_t mask) {
2      // 标记 SACK 确认的段
3      for (size_t i = 0; i < SACK_BITS; ++i) {
4          if (mask & (1u << i)) {
5              uint32_t seq = ack + 1 + static_cast<uint32_t>(i);
6              window_.mark_acked(seq);
7          }
8      }
9
10     // 处理 SACK 缺口重传 (限速: 单次 ACK 最多重传 4 个)
11     int gap_retx_count = 0;
12     uint64_t now = now_ms();
13     for (size_t i = 0; i < SACK_BITS; ++i) {
14         uint32_t seq = ack + 1 + static_cast<uint32_t>(i);
15         if (seq > window_.total_segments()) {
16             break;
17         }
18         auto& seg = window_.get_segment(seq);
19         if (seg.sent && !seg.acked && !(mask & (1u << i))) {
20             uint64_t last_gap = seg.last_sack_retx ? seg.last_sack_retx : seg.last_send;
21             if (gap_retx_count < MAX_SACK_RETX_PER_ACK && now >= last_gap +
               ↪  MIN_GAP_RETX_INTERVAL_MS) {
```

```
                seg.last_sack_retx = now;
                ++gap_retx_count;
                transmit_segment(seq);
            }
        }
    }
}
```

(5) 超时重传、RTO 自适应与全局超时 发送端会周期性检查窗口内“已发送未确认”的段：若 now − last_send > rto_ 则触发超时重传(handle_timeouts()),并把该事件计入统计(stats_.record_timeout())。RTO 采用 Jacobson/Karels 算法自适应更新：在 handle_new_ack() 中用一个未被重传过的段测量 RTT 样本（Karn：重传段不参与 RTT 估计），再用 EWMA 更新 srtt_/rttvar_ 计算新的 rto_；一旦发生超时，还会把 rto_ 进行倍增退避（上限 60s），并调用 congestion_.on_timeout() 让 Reno 回到慢启动。为了避免极端情况下“永远重传”，实现还设置了两类硬上限：单段最大重传次数 MAX_RETRANSMITS=15（超过则发送 RST 认为连接断开），以及“30 秒收不到任何 ACK”的全局超时 GLOBAL_TIMEOUT_MS=30000（直接 RST 并退出）。

```
void ReliableSender::handle_new_ack(uint32_t ack) {
    // 测量 RTT (Karn 算法: 只测量未重传的段来更新 RTT)
    for (uint32_t i = window_.get_base_seq(); i < ack && i <= window_.total_segments(); ++i) {
        auto& seg = window_.get_segment(i);
        if (!seg.acked && seg.sent && !seg.is_retransmitted && seg.send_timestamp > 0) {
            uint64_t rtt_sample = now_ms() - seg.send_timestamp;
            update_rto(rtt_sample);^^I // 更新 RTO
            break;^^I
        }
    }
    // 标记所有被确认的段
    ...
}
```

(6) 零窗口探测与连接关闭 当对端通告窗口为 0 时，发送端进入 zero_window_ 并启动持续计时器(persist timer):按 5s 起步做指数退避,周期性发送空 ACK 探测包,直到对端窗口恢复(handle_window_probe())。当所有数据段都确认后,发送端发送 FIN(seq = isn_+total_segments+1),若在 HANDSHAKE_TIMEOUT_MS 内未收到 FIN|ACK 则重试，最多 MAX_FIN_RETRIES=5 次；收到 FIN|ACK 后发送最终 ACK 并结束。

```
void ReliableSender::handle_ack(const Packet& pkt) {
    last_ack_time_ = now_ms();
    uint16_t new_peer_wnd = std::min<uint16_t>(pkt.header.wnd, static_cast<uint16_t>(SACK_BITS));
    if (new_peer_wnd == 0 && !zero_window_) {
        // 进入零窗口状态
        zero_window_ = true;
        persist_backoff_ = 0;
        persist_timer_ = now_ms() + 5000;
        cout << "[windows]: zero window, start persist timer" << endl;
    } else if (new_peer_wnd > 0 && zero_window_) {
        // 离开零窗口状态
        zero_window_ = false;
```

```
        persist_backoff_ = 0;
        cout << "[windows]: reopen " << new_peer_wnd << endl;
    }
    ...
}
```

（7）四次握手断开连接 发送端只有在 window_.all_acked() 为真（所有数据段都被累计确认/被 SACK 标记确认并推进窗口）后，才会进入关闭流程并发送 FIN，避免“还有数据未可靠交付就提前断开”。

由于本实验是单向文件传输，接收端在数据阶段没有反向应用数据要发送，因此在收到发送端 FIN 后，接收端直接回一个 FIN|ACK：既确认“你的 FIN 我收到了”，也表示“我这边也可以结束了”。概念上仍对应四次挥手，但在报文交互上合并为三步：FIN → FIN|ACK → ACK。

```
^^Ivoid ReliableSender::try_send_fin() {
^^I^^Iuint64_t now = now_ms();  // 获取当前时间
^^I^^Iif (fin_complete_) {
^^I^^I^^Ireturn;
^^I^^I}
^^I^^Iif (!fin_sent_) {
^^I^^I^^Iif (!window_.all_acked()) {
^^I^^I^^I^^Ireturn;
^^I^^I^^I}
^^I^^I^^I// 发送 FIN 包
^^I^^I^^IPacketHeader fin{};
^^I^^I^^Ifin.seq = isn_ + window_.total_segments() + 1;
^^I^^I^^Ifin.ack = 0;
^^I^^I^^Ifin.flags = FLAG_FIN;
^^I^^I^^Ifin.wnd = window_size_;
^^I^^I^^Ifin.len = 0;
^^I^^I^^Ifin.sack_mask = 0;
^^I^^I^^Isend_raw(fin, {});
^^I^^I^^I// 记录 FIN 发送状态
^^I^^I^^Ifin_sent_ = true;
^^I^^I^^Ifin_last_send_ = now;
^^I^^I^^Ifin_retry_count_ = 0;
^^I^^I^^Ireturn;
^^I^^I}
^^I^^I// 处理 FIN 重传
^^I^^Iif (now - fin_last_send_ > HANDSHAKE_TIMEOUT_MS && fin_retry_count_ < MAX_FIN_RETRIES) {
^^I^^I^^I// 重传 FIN 包
^^I^^I^^IPacketHeader fin{};
^^I^^I^^Ifin.seq = isn_ + window_.total_segments() + 1;
^^I^^I^^Ifin.ack = 0;
^^I^^I^^Ifin.flags = FLAG_FIN;
^^I^^I^^Ifin.wnd = window_size_;
^^I^^I^^Ifin.len = 0;
^^I^^I^^Ifin.sack_mask = 0;
^^I^^I^^Isend_raw(fin, {});
^^I^^I^^I// 更新重传状态
```

```
^^I^^I^^Ifin_last_send_ = now;
^^I^^I^^Ifin_retry_count_++;
^^I^^I}
^^I}
```

关闭阶段同样可能丢包，因此发送端对 FIN 做超时重试；接收端在 handle_fin() 中对 FIN|ACK 做突发发送并短等待最终 ACK，在尽量提高送达概率的同时也避免在对端已退出时无限期阻塞。

4.6 接收端

我们先看接收端的整体的流程：

```
int ReliableReceiver::run() {
    // 创建 wsa
    ...
    // 创建 UDP Socket
    ...
    // 绑定监听端口
    ...

    // 执行三次握手
    if (!do_handshake()) {
        return 1;
    }

    // 打开输出文件
    ofstream out(output_path_, std::ios::binary);
    if (!out) {
        return 1;
    }

    // 记录开始时间
    stats_.set_start_time(now_ms());

    while (true) {
        Packet p{};
        sockaddr_in from{};
        // 等待数据包，5000ms 没有收到新的数据包则认为是超时，收到的是坏包也会继续等待
        if (!wait_for_packet(p, from, DATA_TIMEOUT_MS)) {
            consecutive_timeouts_++;
            if (consecutive_timeouts_ >= MAX_CONSECUTIVE_TIMEOUTS) {
                break;
            }
            continue;
        }
        consecutive_timeouts_ = 0;

        if (!same_endpoint(from, client_)) {
            continue;
        }
```

```
39
40          if (p.header.flags & FLAG_RST) {
41              break;
42          }
43
44          // 处理 FIN 包
45          if (p.header.flags & FLAG_FIN) {
46              handle_fin(p.header.seq);
47              break;
48          }
49
50          // 处理数据包
51          if (p.header.flags & FLAG_DATA) {
52              process_data_packet(p, out);
53          }
54      }
55
56      if (stats_.get_end_time() == 0) {
57          stats_.set_end_time(now_ms());
58      }
59
60      return 0;
61  }
```

接收端启动后先绑定监听端口，等待 SYN 建链。握手完成后，接收端打开输出文件并进入循环接收数据：

（1）只处理来自已连接对端的包 握手期间会锁定对端地址，后续收到的包如果不是来自该端点就直接忽略，避免“旁路流量”污染当前传输。

```
1   bool ReliableReceiver::do_handshake() {
2       Packet pkt{};
3       sockaddr_in from{};
4       while (true) {
5           // 无限等待 SYN 包
6           if (!wait_for_packet(pkt, from, -1)) {
7               continue;
8           }
9           // 检查并记录
10          if (!(pkt.header.flags & FLAG_SYN)) {
11              continue;
12          }
13          client_ = from;
14          peer_isn_ = pkt.header.seq;
15          cout << "[DEBUG] Received SYN from " << addr_to_string(from) << endl;
16
17          // 生成本端 ISN
18          ...
19          isn_ = generate_isn(local_info, client_);
20
```

```
        // 发送 SYN+ACK 包
        PacketHeader syn_ack{};
        ...

        bool acked = false;

        for (int attempt = 0; attempt < MAX_HANDSHAKE_RETRIES && !acked; ++attempt) {
            send_raw(syn_ack, {});
            // 等待 ACK 或数据包（隐式完成握手）
            Packet confirm{};
            sockaddr_in confirm_from{};
            if (wait_for_packet(confirm, confirm_from, HANDSHAKE_TIMEOUT_MS) &&
                same_endpoint(confirm_from, client_)) {

                if (confirm.header.flags & FLAG_RST) {
                    return false;
                }

                // 检查是否为 ACK 包
                if ((confirm.header.flags & FLAG_ACK) && confirm.header.ack == syn_ack.seq + 1) {
                    acked = true;
                } else if (confirm.header.flags & FLAG_DATA) {
                    acked = true;
                }
            }
        }

        if (acked) {
            buffer_.set_expected_seq(peer_isn_ + 1);
            return true;
        }

        send_rst();^^I // 握手失败，发送 RST 终止连接
    }
    return false;
}
```

(2)窗口内缓存 + 连续交付 接收端通过 ReceiveBuffer 维护乱序缓存(map<uint32_t, payload>):若 seq < expected 直接判为重复包,计数并回 ACK;若 seq 超出 [expected, expected+window_size) 则判为越窗包,丢弃并回 ACK;否则把 payload 缓存起来。随后调用 extract_continuous_segments() 从 expected 开始连续提取并写入文件，直到遇到缺口为止，保证应用层交付始终有序。

```
void ReliableReceiver::process_data_packet(const Packet& pkt, ofstream& out) {
    total_packets_received_++;
    uint32_t seq = pkt.header.seq;

    // 检查是否为重复包或窗口外的包
    if (seq < buffer_.get_expected_seq()) {
        duplicate_packets_++;
```

```
        send_ack();
        return;
    }
    if (!buffer_.is_in_window(seq)) {
        send_ack();
        return;
    }

    // 将接受的数据添加到缓冲区
    if (buffer_.add_segment(seq, pkt.payload)) {
        if (seq > buffer_.get_expected_seq()) {
            out_of_order_packets_++;
        }
    } else {
        duplicate_packets_++;
    }

    // 滑动窗口：提取数据并，推进窗口左边界
    auto segments = buffer_.extract_continuous_segments();
    for (const auto& data : segments) {
        out.write(reinterpret_cast<const char*>(data.data()),
        ↪  static_cast<std::streamsize>(data.size()));
        bytes_written_ += data.size();
    }

    send_ack();^^I
}

```

（3）ACK+SACK 的生成与时机 接收端对每个数据包都会立即回 ACK（send_ack()）：ack.ack 取 expected（累计 ACK，绝对序号），ack.wnd 通告接收窗口大小；同时用 build_sack_mask() 生成 32 位 SACK：位 i 表示 expected $+1+i$ 是否已在缓冲区中。发送端据此能把“已到达但乱序”的段直接标记为已确认，并仅对真正缺失的缺口段做选择性重传。

```
void ReliableReceiver::send_ack(bool fin, uint32_t fin_ack) {
    PacketHeader ack{};
    ack.seq = isn_ + 1;// 本端下一个序号
    // ack 是否为 FIN+ACK，否则就是下一个序号
    ack.ack = fin ? fin_ack : buffer_.get_expected_seq();     // 期望接收的下一个序列号
    // 设置标志位、窗口大小、SACK 掩码
    ack.flags = FLAG_ACK | (fin ? FLAG_FIN : 0);
    ack.wnd = window_size_;^^I                                  // 通告接收窗口大小
    ack.len = 0;                                               // 普通 ACK 携带 SACK 掩码，FIN+ACK 不
    ↪  携带
    ack.sack_mask = fin ? 0 : buffer_.build_sack_mask();     // SACK 掩码
    // 发送带 SACK 掩码的 ACK 包
    send_raw(ack, {});
}
```

(4)接收端超时与关闭 接收主循环以 DATA_TIMEOUT_MS=5000ms 为等待上限：连续超时累计到 MAX_CONSECUTIVE_TI 次则认为 sender 已断开并退出，避免 receiver 无限期阻塞。收到 FIN 后进入关闭处理：以“突发重传”方式发送 FIN|ACK（FIN_ACK_BURST=3），每次发送后最多等待 FIN_ACK_WAIT_MS=200ms 尝试接收最终 ACK；若最终 ACK 未出现也会结束退出，防止 sender 已退出导致 receiver 长时间卡住。

```
void ReliableReceiver::handle_fin(uint32_t fin_seq) {
    if (stats_.get_end_time() == 0) {
        stats_.set_end_time(now_ms());
    }
    uint32_t fin_ack_seq = fin_seq + 1;

    bool final_ack_seen = false;
    for (int i = 0; i < FIN_ACK_BURST; ++i) {
        send_ack(true, fin_ack_seq);
        Packet final_ack{};
        sockaddr_in from{};
        if (wait_for_packet(final_ack, from, FIN_ACK_WAIT_MS) && same_endpoint(from, client_)) {
            if (final_ack.header.flags & FLAG_ACK) {
                final_ack_seen = true;
                break;
            }
            if (final_ack.header.flags & FLAG_FIN) {
                send_ack(true, fin_ack_seq);
            }
        }
    }
}
```

5 改进

5.1 性能瓶颈定位

在进行多组文件传输测试后（测试文件和测试方式见“结果展示”），我们观察到：文件规模从百兆增大到数百兆乃至更大时，总耗时会出现明显的“非线性变慢”。该现象并非网络条件突然变差，而是实现中存在随“总分段数”放大的 CPU/内存开销，在没有设置丢包率的情况下传输 1G 文件后期会频繁触发 timeout。

Table 5: 各类型文件在发送端与接收端的传输性能对比

文件名称	大小 (Bytes)	发送时延 (s)	发送速度 (MiB/s)	接收时延 (s)	接收速度 (MiB/s)
1.jpg	1 857 353	0.017	104.195	0.098	18.075
3.jpg	11 968 994	0.190	60.076	0.630	18.118
helloworld.txt	1 655 808	0.016	98.694	0.088	17.944
medium175M.pdf	182 071 279	22.335	7.774	29.754	5.836
large1G.wav	1 250 467 918	1575.360	0.757	1625.720	0.734

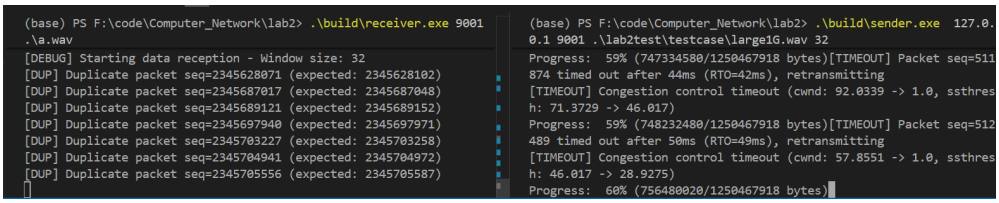


Figure 6: 1G 传输

为此，我们对发送端的热点路径进行了梳理，主要发现两类问题：

1. **超时检测扫描范围过大（handle_timeouts()）：**发送端每轮循环都会进行一次超时检测。如果每次都按“总分段数”线性扫描，即使窗口大小只有几十段，也会在大文件场景下产生巨大的无效遍历开销。
2. **文件切割与缓存策略过于激进：**初始化阶段将整个文件读入内存并一次性切割为所有数据段，会带来高额的内存占用与额外拷贝；文件越大，初始化时间越长，且可能触发频繁内存分配/拷贝，从而拖慢整体传输。

5.2 改进 1：超时检测仅扫描在途段（从 O(N) 到 O(window)）

我们将 `handle_timeouts()` 的扫描范围限制为“在途段”，即只遍历当前窗口内**已发送但未确认**的序号区间（$[\texttt{base\_seq}, \texttt{next\_seq})$）。由于本实验的窗口与 SACK 位图宽度均上限为 32，因此每次超时检测的遍历上限被限制在“窗口大小”量级，从而将每轮循环的检测开销从与文件规模相关的 $O(N)$ 降为与窗口相关的 $O(window)$。

```
void ReliableSender::handle_timeouts() {
    uint64_t now = now_ms();
    uint32_t base = window_.get_base_seq();
    uint32_t total = window_.total_segments();
    if (base == 0 || base > total) {
        return;
    }
    uint32_t next_seq = window_.get_next_seq();
    uint32_t scan_end_exclusive = std::min(next_seq, total + 1);

    // 仅扫描在途段：[base, next_seq)
    for (uint32_t i = base; i < scan_end_exclusive; ++i) {
        auto& seg = window_.get_segment(i);
        if (!seg.acked && seg.sent && now - seg.last_send > static_cast<uint64_t>(rto_)) {
            stats_.record_timeout();
            congestion_.on_timeout();
            rto_ = std::min(rto_ * 2, 60000);
            transmit_segment(i);
        }
    }
}
```

5.3 改进 2：按需读取分段并仅缓存在途数据（降低切割开销与内存占用）

为降低“切割/缓存”带来的启动与内存压力，我们将发送端的 payload 准备方式改为**按需读取**：

1. **不再一次性读入整个文件并全量切割：**发送端启动后只打开文件并获取文件大小，用文件大小计算总分段数，而不在初始化阶段生成所有段的 payload。
2. **发送/重传时再读取段数据：**当某个序号的数据段需要发送或重传时，才根据 `seq` 计算文件偏移并从文件中读取 $\leq$ `MAX_PAYLOAD` 字节作为 payload。
3. **ACK 后立即释放 payload：**当某段被累计确认或被 SACK 标记确认后，会尽快释放其 payload 缓存，避免在大文件场景下长期占用内存。

在 send_window.h 中：

```
SendWindow::SendWindow() : total_segments_(0), base_seq_(1), next_seq_(1) {}

void SendWindow::initialize(uint64_t file_size_bytes) {
    total_segments_ = static_cast<uint32_t>((file_size_bytes + MAX_PAYLOAD - 1) / MAX_PAYLOAD);
    segments_.clear();
    base_seq_ = 1;
    next_seq_ = 1;
}

void SendWindow::mark_acked(uint32_t seq) {
    if (seq == 0 || seq > total_segments_) {
        return;
    }
    auto it = segments_.find(seq);
    if (it == segments_.end()) {
        return;
    }
    auto& seg = it->second;
    if (seg.acked) {
        return;
    }
    seg.acked = true;
    seg.last_sack_retx = 0;
    seg.data.clear();
    seg.data_loaded = false;
}

SendWindow::SegmentInfo& SendWindow::get_segment(uint32_t seq) {
    auto it = segments_.find(seq);
    if (it != segments_.end()) {
        return it->second;
    }
    auto [ins_it, _] = segments_.emplace(seq, SegmentInfo{});
    return ins_it->second;
}

void SendWindow::set_base_seq(uint32_t seq) {
    base_seq_ = seq;
    for (auto it = segments_.begin(); it != segments_.end();) {
        if (it->first < base_seq_) {
```

```
            it = segments_.erase(it);
        } else {
            ++it;
        }
    }
}

bool SendWindow::all_acked() const {
    return base_seq_ > total_segments_;
}

size_t SendWindow::inflight_count() const {
    if (next_seq_ >= base_seq_) {
        return next_seq_ - base_seq_;
    }
    return 0;
}

void SendWindow::advance_base_seq() {
    while (base_seq_ <= total_segments_) {
        auto it = segments_.find(base_seq_);
        if (it == segments_.end()) {
            return;
        }
        if (!it->second.acked) {
            return;
        }
        segments_.erase(it);
        ++base_seq_;
    }
}
```

在 sender.h 中主要修改：

```
size_t ReliableSender::payload_len_for_seq(uint32_t seq) const {
    if (seq == 0 || file_size_ == 0) {
        return 0;
    }
    uint64_t offset = static_cast<uint64_t>(seq - 1) * static_cast<uint64_t>(MAX_PAYLOAD);
    if (offset >= file_size_) {
        return 0;
    }
    return static_cast<size_t>(std::min<uint64_t>(MAX_PAYLOAD, file_size_ - offset));
}

bool ReliableSender::load_segment_payload(uint32_t seq, vector<uint8_t>& out) {
    size_t len = payload_len_for_seq(seq);
    out.assign(len, 0);
    if (len == 0) {
        return true;
    }
```

```
    uint64_t offset = static_cast<uint64_t>(seq - 1) * static_cast<uint64_t>(MAX_PAYLOAD);

    input_.clear();
    input_.seekg(static_cast<std::streamoff>(offset), std::ios::beg);
    if (!input_) {
        return false;
    }
    input_.read(reinterpret_cast<char*>(out.data()), static_cast<std::streamsize>(len));
    return input_.gcount() == static_cast<std::streamsize>(len);
}
```

这样，发送端的内存占用从“与文件大小/总分段数线性相关”转为“与窗口大小线性相关”，上界约为 $O(window \times \texttt{MAX\_PAYLOAD})$，更适合传输超大文件；同时初始化阶段不再执行全量切割与大量拷贝，能够显著减少“切割开销”。

6 结果展示

6.1 实验环境与准备

本实验在 Windows 环境下开发与测试，使用 Winsock 进行 UDP 通信。为了复现实验要求的“不可靠网络”，我们使用 `lab2/clumsy` 中提供的 clumsy 程序注入丢包与延迟。测试文件位于 `lab2test/testcase`，包含三张图片与一个文本文件。

Figure 7: Caption

6.2 编译步骤

推荐使用 CMake 构建。首先在项目根目录配置构建目录，然后编译 Release：

```
cmake -S . -B build -G "MinGW Makefiles" -DCMAKE_C_COMPILER=gcc -DCMAKE_CXX_COMPILER=g++
-DCMAKE_SH="CMAKE_SH-NOTFOUND
cmake --build build --config Release
```

编译完成后会生成 `.\build\sender.exe` 与 `.\build\receiver.exe`。如果使用单配置生成器，二进制在 `build\` 目录下。

6.3 基础传输演示（无丢包/无延迟）

为了确认协议“最基本的正确性”，我们先在无丢包条件下进行传输，确保接收端输出文件大小一致、内容一致。运行时先启动接收端，再启动发送端：

```
receiver.exe <listen_port> <output_file> <window_size>
sender.exe <receiver_ip> <receiver_port> <input_file> <window_size>
```

当传输结束后，发送端会打印耗时、平均吞吐率、重传次数与丢包率；接收端会打印接收耗时、乱序/重复统计等信息。其中“丢包率”的统计口径为**数据段重传次数 ÷ 数据段总数**，不包含握手/挥手控制报文的丢失；“传输耗时”按实验要求只统计数据传输阶段（首个数据段发送到全部数据段被确认），因此更适合用来比较不同窗口与不同丢包率下的变化趋势。

我们先看 32 个窗口大小的改进后的结果：然后我们来做文件完整性验证。由于“能打开图片/能读文本”并不能排除**少量字节错误**（例如某个位置被改写、丢失或重复），更可靠的办法是对比发送端原文件与接收端输出文件的哈希值。这里我们使用 WSL 环境提供的 `sha256sum` 计算 SHA-256 摘要：

Table 6: 改进后的文件传输性能实验数据

文件名称	大小 (Bytes)	发送时延 (s)	发送速度 (MiB/s)	接收时延 (s)	接收速度 (MiB/s)
1.jpg	1 857 353	0.020	88.566	0.023	77.014
2.jpg	5 898 505	0.060	93.754	0.061	92.217
3.jpg	11 968 994	0.160	71.341	0.163	70.028
helloworld.txt	1 655 808	0.019	83.111	0.028	56.397
medium175M.pdf	182 071 279	2.275	76.324	2.187	79.395
large1G.wav	1 250 467 918	15.038	79.302	15.029	79.349

```
sha256sum <input_file>
sha256sum <output_file>
```

SHA-256 会把任意长度的文件映射为一个 256-bit 的固定长度摘要。只要两个文件的摘要值一致，就意味着它们在字节层面**几乎必然完全相同**：要想让两个不同文件碰撞到同一个 SHA-256 摘要，在工程上是不可行的（碰撞概率极低）。因此，在基础传输（无丢包/无延迟）场景下，`sha256sum` 的一致性可以作为“传输正确性”的强证据。

Figure 8: 基础传输（无丢包/无延迟）下的 SHA-256 完整性校验

从上述结果可以看到，各文件的“原文件摘要”和“接收文件摘要”完全一致，说明接收端输出文件与发送端输入文件在字节层面一致，从而完成基础传输正确性的完整性验证。

6.4 在不可靠网络下的传输（clumsy 注入丢包与延迟）

为了复现实验要求的“不可靠网络”，我们使用 clumsy 在本机对 UDP 流量注入**丢包**与**延迟**，从而观察协议在非理想网络下的行为与性能变化。与“基础传输演示”相比，该部分重点不再是极限吞吐，而是验证：在出现丢包/乱序/延迟时，协议仍能正确完成传输，并且能通过 ACK/SACK、重传与拥塞控制逐步恢复吞吐。

实验设置 本次实验选取丢包率 5%、固定延迟 5ms 的组合进行验证。窗口大小固定为 32（与 SACK 位图宽度一致），以便把变化主要归因于网络条件而非窗口配置。

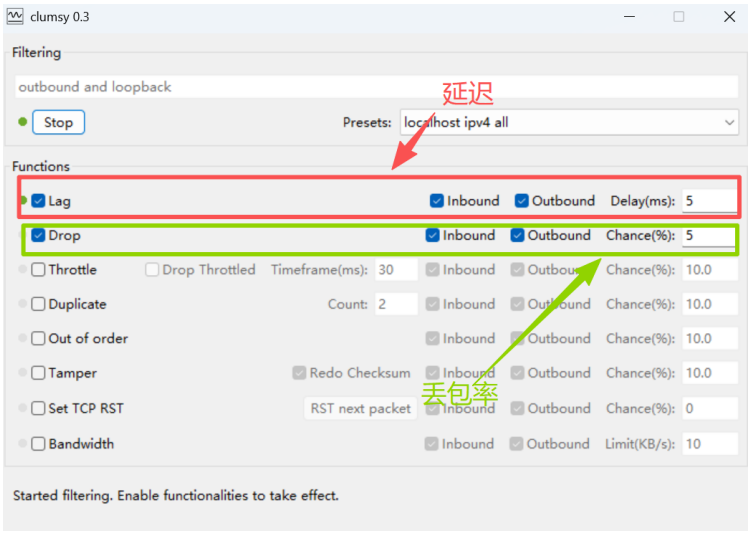


Figure 9: clumsy 注入丢包与延迟的设置

为突出“可靠性与机制正确性”，这里优先使用小文件（如 `helloworld.txt`、`1.jpg`、`2.jpg`、`3.jpg`）进行多次重复实验。

Table 7: 弱网环境下文件传输性能测试结果 (设置丢包率: 5%, 延迟: 5ms)

文件名称	**总分段数** (Segments)	**实测丢包率** (%)	**发送时延 (s)**	**接收时延 (s)**	**平均吞吐量** (MiB/s)
1.jpg	1273	7.6198	23.609	23.610	0.0750
2.jpg	4041	7.7951	88.185	88.228	0.0638
3.jpg	8198	7.8189	187.886	187.912	0.0608
helloworld.txt	1135	9.3392	28.730	28.759	0.0549

现象与原因分析 以传输 `2.jpg` 的日志片段为例（图10），可以更直观看到弱网下协议的典型行为链路：接收端侧的乱序/重复现象 → 发送端侧的重复 ACK/SACK 反馈 → 快速重传或超时重传 → Reno 调整拥塞窗口并继续推进。

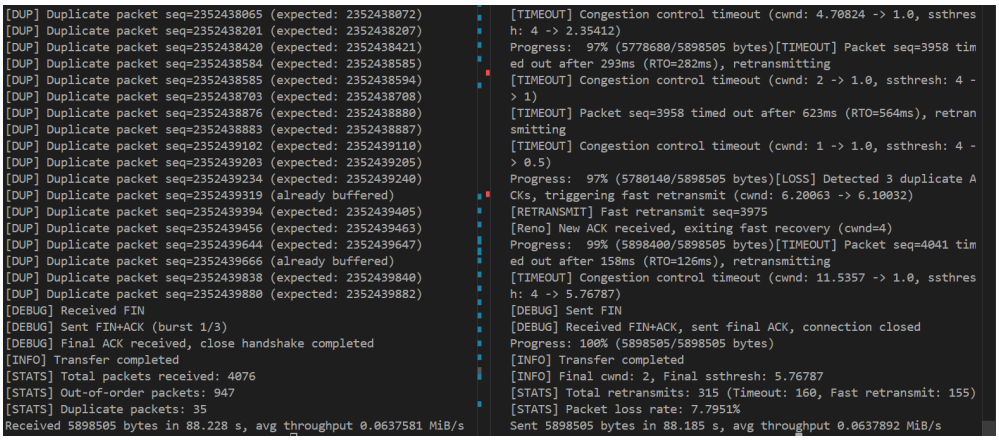


Figure 10: 弱网环境下传输 2.jpg 的部分日志

1. **接收端出现大量乱序与重复：** 日志中多次出现 [DUP] Duplicate packet，其含义包括：seq < expected（该段序号早于当前期望序号，属于重复到达）以及 already buffered（该段已经在乱序缓冲区中，再次到达说明网络中存在重复包或发送端发生了重传）。这些现象说明弱网下“包会丢、也会乱序”，接收端只能在缓存中等待缺口被补齐后再连续写入文件。

2. **发送端同时触发两类恢复路径：快速重传与超时重传：** 当接收端持续对同一缺口反馈累计 ACK 时，发送端会统计重复 ACK，并在达到阈值后打印 [LOSS] Detected 3 duplicate ACKs，随后执行 [RETRANSMIT] Fast retransmit 对缺口段进行快速重传；若丢包/延迟更严重导致在当前 RTO 内仍未收到推进 ACK，则会出现 [TIMEOUT] Packet seq=... timed out 的超时重传。两者相比，快速重传通常更“及时”，能减少等待 RTO 的空转。

3. **Reno 导致吞吐显著下降但能保证最终正确：** 日志中可见多次 Congestion control timeout (cwnd: ... -> 1.0)，这表示发生超时后 Reno 将拥塞窗口收缩到 1 并降低 ssthresh，发送端进入慢启动重新探测带宽；当缺口被补齐并收到新的累计 ACK 时，会出现 [Reno] New ACK received, exiting fast recovery 等状态切换。由于丢包率较高且频繁触发回退，窗口无法长期维持在较大值，最终表现为表7 中 2.jpg 的吞吐明显低于无丢包场景，但通过 ACK/SACK 与重传机制仍能完成传输并通过哈希一致性验证。

正确性验证 对每次不可靠网络下的传输，我们仍使用 sha256sum 对比发送端原文件与接收端输出文件的摘要，以证明最终文件在字节层面一致。

Figure 11: 弱网环境下的 SHA-256 完整性校验

6.5 不同窗口大小与不同丢包率的对比思路

我们固定测试文件为 `2.jpg`，每个配置重复 10 次并取平均值，可以有效降低偶然抖动对单次测量的影响；同时把变量拆分为“窗口大小”和“丢包率”两条轴分别对比，能够更清晰地解释吞吐、重传与 Reno 回退的变化来源。下文按这两部分给出具体的实验设置、记录表与折线图。

6.5.1 不同窗口大小对比

控制变量：固定测试文件为 `2.jpg`，固定 clumsy 网络条件（丢包率 5%、延迟 5ms），每个窗口配置重复 10 次取平均值；只改变发送/接收窗口大小（1、2、4、8、16、32）。**预期现象：**窗口较小时流水线深度不足，链路难以被充分填满，吞吐受限；窗口增大后吞吐上升，但在较高丢包率下会更频繁触发重传与 Reno 回退，收益可能出现边际递减。

Table 8: 不同窗口大小对传输性能的影响 (5% 丢包率, 5ms 延迟)

窗口大小	实测丢包率 (%)	传输用时 (s)	吞吐量 (MiB/s)	相对提升
1	13.8332	427.866	0.01315	基准
2	8.339 52	207.977	0.02705	105.7 %
4	7.349 67	94.498	0.05953	352.7 %
8	7.671 37	82.340	0.06832	419.5 %
16	6.879 49	79.809	0.07048	435.9 %
32	7.176 44	77.709	0.07239	450.5 %

* 注：测试文件均为 2.jpg (5,898,505 Bytes)。相对提升基于窗口为 1 时的吞吐量计算。

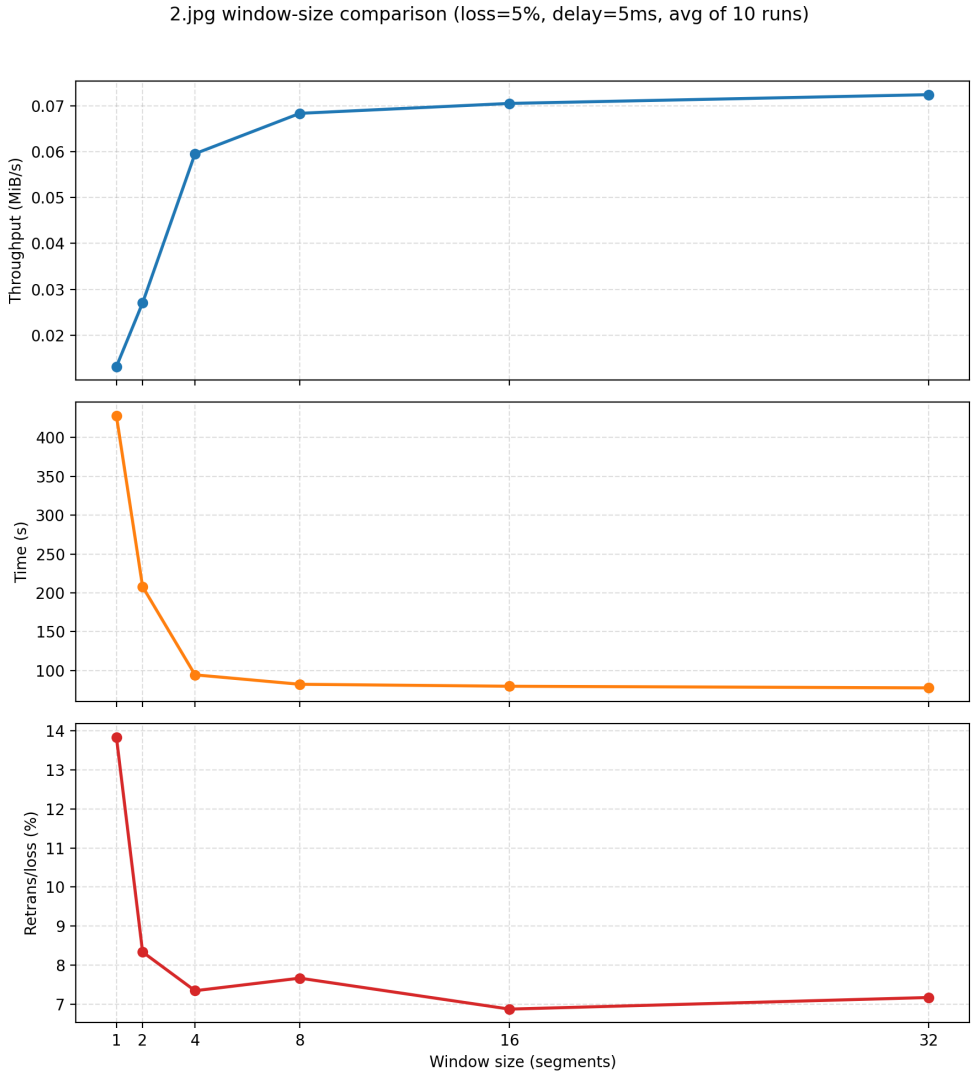


Figure 12: 不同窗口大小对吞吐与耗时的影响

现象与原因分析 结合表8 与图12，可以观察到窗口大小从 1 增大到 4 时吞吐显著提升、耗时显著下降；当窗口继续增大到 8 及以上时，吞吐仍有提升但增幅变小，表现为边际收益递减。其原因主要有三点：

1. **流水线深度决定链路利用率：**窗口为 1 时接近“停等”，每个 RTT 只能确认极少量数据，难以把链路填满；窗口增大后，在途段增多，发送端能在等待 ACK 的同时持续发送，从而显著提高链路利用率。

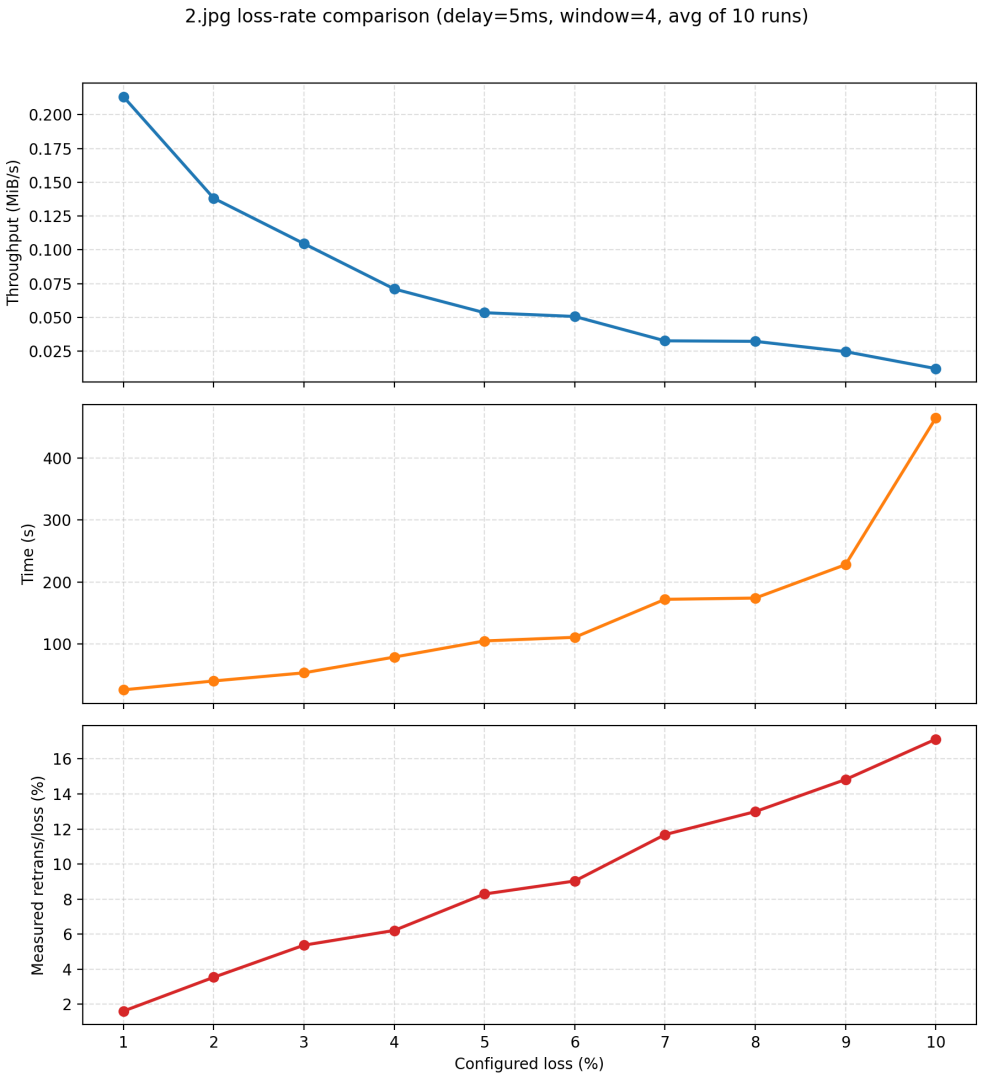
2. **超过带宽-时延积后收益趋缓：**当窗口足以覆盖端到端时延造成的“确认空窗期”后，进一步增大窗口并不会线性提升吞吐，此时性能更多受限于链路带宽、接收端处理能力以及丢包导致的恢复开销，因此曲线逐渐趋于平缓。

3. **丢包与 Reno 回退带来上限：**在 5% 丢包环境下，窗口越大代表同时在途的段越多，出现缺口需要重传的概率也更高。虽然 SACK 与快速重传能减少等待 RTO 的空转，但当丢包触发超时或频繁重复 ACK 时，Reno 会降低 `cwnd` 并限制有效发送窗口，导致吞吐的进一步提升受限。

6.5.2 不同丢包率对比

控制变量：固定测试文件为 `2.jpg`，固定窗口大小为 32，固定延迟（5ms），每个丢包率配置重复 10 次取平均值；只改变 clumsy 的丢包率（1% 到 10%）。

预期现象：丢包率升高会带来更多缺口与重传，重复 ACK 与超时更频繁，从而使 Reno 更频繁回退（`cwnd` 收缩），吞吐下降、耗时上升；SACK 与快速重传能在中等丢包下减少等待超时的空转，但在高丢包下仍会受限于拥塞回退与重传开销。



现象与原因分析 由图6.5.2 与表9 可见：当丢包率从 1% 上升到 10% 时，传输用时整体呈上升趋势，而吞吐量呈明显下降趋势，且下降幅度在高丢包区间更为剧烈（例如 10% 时吞吐量仅约为 1% 时的 5.7%）。主要原因包括：

1. **必然重传开销随丢包率上升而上升：**丢包率越高，缺口段出现越频繁，发送端需要发送更多重传段才能补齐缺口；在固定带宽下，更多的重传意味着更少的“有效新数据”占比，吞吐自然下降。
2. **RTO 等待与 Reno 回退放大耗时：**在中高丢包下，除了快速重传外还更容易触发超时重传。超时不仅会引入“等待 RTO 到期”的空转时间，还会触发 Reno 的强退让（`cwnd` 收缩到 1 并重新

Table 9: 丢包率对传输性能的影响 (固定延迟: 5ms, 窗口大小: 4)

设置丢包率	实测丢包率 (%)	传输用时 (s)	吞吐量 (MiB/s)	性能下降比
1%	1.6085	26.374	0.21329	–
2%	3.5387	40.717	0.13816	-35.2%
3%	5.3700	53.800	0.10456	-51.0%
4%	6.2113	79.183	0.07104	-66.7%
5%	8.2900	105.198	0.05347	-74.9%
6%	9.0324	111.000	0.05068	-76.2%
7%	11.6803	172.229	0.03266	-84.7%
8%	12.9918	174.249	0.03228	-84.9%
9%	14.8231	228.107	0.02466	-88.4%
10%	17.1245	464.438	0.01211	-94.3%

* 注：性能下降比以 1% 丢包率时的吞吐量为基准计算。测试文件为 2.jpg。

慢启动），使得一段时间内在途数据很少，进一步拉长总耗时。

3. **乱序与重复 ACK 增多导致发送端恢复路径更频繁：**丢包引发的乱序会使接收端持续回累计 ACK 并携带 SACK 位图，发送端据此进行缺口修复。虽然 SACK 能减少无谓的重复发送，但在高丢包率下“缺口本身更多”，仍会导致恢复过程频繁发生、稳定窗口难以维持，从而出现明显的性能下降。

7 总结与反思

7.1 实验收获

本次实验把“课堂上 TCP 可靠传输的概念”落到了工程实现上。通过在 UDP 上补齐序号、确认、重传、窗口与拥塞控制，可以直观看到：网络只要出现丢包与延迟，很多看似细碎的机制（SACK、快速重传、窗口限制、握手与关闭等）都会变得必要，否则不是传不完，就是传完但内容错误。

7.2 设计取舍与可能的改进方向

本项目用 32 位 SACK 位图在“带宽开销”和“表达能力”之间做了折中：它覆盖了最常见的少量乱序/丢包，但如果窗口更大或乱序跨度更大，SACK 信息可能不足。另一个改进点是更精细的 RTO 估计与更严格的 Karn 规则处理，以减少“超时重传过早/过晚”对吞吐和重传次数的影响。

8 GitHub 仓库

本次实验的源代码与相关文件仓库地址如下：

https://github.com/LoveGump/Nku_Computer_Network/tree/main/lab2（已经私密，需要看请联系 2312236）