实验报告：基于 UDP 数据报套接字的面向连接可靠传输（RDT + SACK + 固定窗口 + Reno）

1. 实验目的与要求理解

本实验目标是在用户空间基于 UDP（数据报套接字）实现一个“面向连接、可靠传输”的协议栈，能够在受控网络条件（丢包、延时）下完成文件的单向传输，并输出传输耗时与平均吞吐率。协议需具备连接管理、差错检测、流水线确认重传（选择确认）、固定窗口流控，以及 Reno 拥塞控制。

我采用“TCP 风格的字节序号 + 三次握手/四次挥手思想 + 累计 ACK + SACK 位图 + Reno cwnd/ssthresh”的设计，尽量在结构清晰与可复现之间取得平衡，同时保持实验环境（router.exe）兼容：在router.exe中，仅对 Client→Server 数据方向做丢包/延迟，Server→Client ACK 不做处理，因此 ACK 通道相对稳定，但数据通道会出现随机丢失和固定延迟。

---

2. 实验环境与运行配置

2.1 平台与编译运行

- 操作系统：Windows（Winsock2）
- 编译方式：C/C++ 编译器（我用的 g++），链接 `ws2_32.lib`

```
g++ -std=c++11 receiver.cpp -o receiver.exe -lws2_32
g++ -std=c++11 sender.cpp -o sender.exe -lws2_32
```

Socket 类型：`SOCK_DGRAM`（UDP），不使用 CSocket 等封装类，完全基于基础 API

代码中使用 `ioctlsocket` 将 socket 设为非阻塞，主循环以 `sleep(1)` 降低 CPU 占用。

2.2 Router 实验环境

在 router 环境下，通信链路变为：

Sender(Client) → Router → Receiver(Server)（对该方向的数据包做丢包/延时）
Receiver(Server) → Router → Sender(Client)（ACK/控制包通常不做丢包/延时）

因此我在 sender 端做了两点适配：

1. sender 程序参数中显式传入 `client_ip/client_port` 并 `bind()` 固定端口，保证 router 转发回包能准确回到 sender；
2. sender 的 `peer` 设置为 router 的 IP/端口（即 `sendto()` 直接发给 router）。

receiver 端同样按 router 的规则：receiver 实际收到的来源地址会是 router，因此 receiver 只接受一个“peer”（首个 SYN 的来源），后续控制包与数据包都要求来自同一 peer。

2.3 运行步骤

我本地实验的顺序为（避免握手阶段收不到包）：

1. **启动 receiver（监听 server 端口）**
   `receiver.exe <bind_ip> <bind_port> <output_file> <fixed_wnd_segments>`
2. **启动 router（配置丢包率/延迟，并绑定其转发端口）**
   router 的具体参数以课程提供的程序说明为准；总体逻辑是 router 监听一个端口接收来自 sender 的包，并转发到 receiver；对 Client→Server 做 loss/delay。
3. **启动 sender（绑定 client 端口并把 peer 指向 router）**
   `sender.exe <client_ip> <client_port> <router_ip> <router_port> <input_file> <fixed_wnd_segments>`

---

3. 协议总体设计

3.1 报文格式（RdtHeader + Payload）

协议头在 `rdt.h` 中定义为：

- `seq`：本段起始字节序号（SYN/FIN 也占用 1 个序号）
- `ack`：累计确认号（下一个期望字节）
- `flags`：SYN/ACK/FIN/DATA/RST
- `wnd`：固定窗口大小（单位：分片数）
- `len`：payload 长度
- `cksum`：16-bit Internet checksum（header+payload）
- `sack_mask`：SACK 位图（对 ack 之后 32 个分片的接收情况）

我选用“**字节序号**”而不是“分片序号”，主要原因是：

- 更接近 TCP，FIN/SYN 可自然占用 1 个序号；
- 与 payload 长度直接对应，累计 ACK 的推进更直观；
- 理论上可扩展实现：对不同大小 payload 也能统一处理（尽管本实验先简单按 MSS 切片）。

3.2 连接管理状态机

receiver 侧状态机：`R_CLOSED → R_SYN_RCVD → R_EST → R_FIN_WAIT`
sender 侧主要通过握手循环/FIN 状态变量体现。

握手与挥手借鉴 TCP 思路，但做了简化：

- 三次握手：SYN → SYN|ACK → ACK
- 关闭：DATA 完成后 sender 发 FIN；receiver 收 FIN 后回 ACK 并发自身 FIN；sender 收到 FIN 后回 ACK 并退出。
- 控制包（SYN/FIN）的重传超时采用 `RDT_HANDSHAKE_RTO_MS`。

---

4. 核心功能实现说明

4.1 连接管理（建立/关闭/异常处理）

4.1.1 三次握手

- sender：周期性发送 SYN（超时重传），等待收到 `(SYN|ACK)` 且 `ack == isn_send + 1`，然后回最终 ACK。
- receiver：在 `R_CLOSED` 收到 SYN 后记录 peer，回 SYN|ACK，进入 `R_SYN_RCVD`；收到最终 ACK 后进入 `R_EST`。

此设计能在丢包环境下通过 SYN 重传保证建连成功，同时 receiver 只绑定一个 peer，避免多源干扰。

4.1.2 连接关闭（FIN）

- sender：当"所有数据段都被确认"后发送 FIN，并等待对端 FIN/ACK 流程完成；若 FIN 未被确认则按 handshake RTO 重传 FIN。
- receiver：收到 FIN 后先回 ACK，再发送自己的 FIN，进入 FIN_WAIT，等待对端 ACK 结束。

4.1.3 异常处理

- checksum 校验失败直接丢弃包；
- 重传次数超过 `RDT_MAX_RETX` 触发退出（防止死循环）；
- 对于非 peer 来源的包（receiver 侧）直接忽略，保证链路唯一性。

4.2 差错检测：校验和

我使用 Internet checksum（16-bit one's complement），实现于 `rdt.h`：

- `checksum16()`：对 header+payload 计算校验；
- `fill_checksum()`：发送前将 cksum 置 0 后计算并写入；
- `verify_checksum()`：接收后将 cksum 清零重算并比对。

发送流程为：**先在 host 序计算 checksum → 再 hton(header) → sendto()**。
接收流程为：**recv → ntoh(header) → verify_checksum()**，失败则 drop。

4.3 确认重传：流水线 + SACK（选择确认）

4.3.1 流水线发送与发送缓冲

sender 维护一个发送缓冲 `out: map<seq, OutSeg>`，每个 `OutSeg` 保存：

- 分片 seq/len/data
- acked 标记
- last_sent_ms（用于 RTO）
- retx（用于限制重传次数）

主循环中计算当前在途未确认分片数 `inflight`，并计算有效窗口：

eff_wnd = min(cwnd, fixed_wnd)

随后 while 循环不断切片发送，直到窗口填满，从而形成流水线。

4.3.2 receiver 侧乱序缓存 + SACK 位图

receiver 使用 `ooo: map<seq, SegmentBuf>` 缓存乱序段：

- 若 `h.seq == expected_ack`：按序写入文件并推进 expected_ack，同时尝试把 ooo 中连续段一并"吃掉"；
- 若 `h.seq > expected_ack` 且在接收窗口范围内：缓存到 ooo；
- 若 `h.seq < expected_ack`：认为重复段，忽略 payload。

每次发送 ACK 时，receiver 会把 `ack=expected_ack` 并附上 `sack_mask`：

- `build_sack_mask()` 逐 bit 检查 `expected_ack + (i+1)*MSS` 是否存在于 ooo 中，存在则置位。

4.3.3 sender 侧利用 SACK 标记确认

sender 在处理 ACK 时做两步确认：

1. **累计 ACK**：将所有满足 `(seg.seq + seg.len) <= ackno` 的段标记为 `acked=true`
2. **SACK**：`mark_sack_acked(ackno, sack_mask, out)` 把 ackno 之后已到达的乱序段也标记 acked

这样当累计 ACK 卡住时，sender 仍能知道后面哪些段已经到达，从而避免对这些段的重复重传，重传策略更接近选择重传。

4.4 流量控制：发送/接收固定窗口一致

实验要求发送窗口与接收窗口固定且一致，因此我使用同一个参数 `fixed_wnd_segments`：

- sender：用 fixed_wnd 参与 `eff_wnd=min(cwnd,fixed_wnd)` 限制在途段数；
- receiver：用 fixed_wnd 限制乱序缓存接受范围：`expected_ack + fixed_wnd*MSS`。

这保证了发送端不会把接收端缓存完全压爆；同时也方便在实验中改变 fixed_wnd 来观察吞吐变化。

4.5 拥塞控制：Reno（cwnd / ssthresh / dupACK）

Reno 仅在 sender 端实现，变量含义如下：

- `cwnd`：拥塞窗口（分片数），反映网络可承载的在途数据量
- `ssthresh`：慢启动阈值，用于决定指数/线性增长阶段
- `dup_ack_cnt`：重复 ACK 计数，用于触发快速重传/快速恢复

有效发送窗口始终是 `min(cwnd, fixed_wnd)`：
固定窗口体现端系统/接收缓存上限；cwnd 体现网络拥塞上限。

4.5.1 慢启动（Slow Start）

当收到新 ACK 且 `cwnd < ssthresh`：

- `cwnd += 1`（每个 ACK 增加 1 个分片）
  由于一个 RTT 内约收到 cwnd 个 ACK，因此 cwnd 近似 RTT 级翻倍。

4.5.2 拥塞避免（Congestion Avoidance）

当 `cwnd >= ssthresh` 且收到新 ACK：

- 采用 `cwnd += 1/cwnd` 的近似实现
  我用 `ca_acc` 累加 1/cwnd，累计到 1 再使 cwnd++，从而达到“每 RTT 约 +1”的线性增长。

4.5.3 快速重传 + 快速恢复（3 dupACK）

当 `ackno == last_ack` 时，认为是重复 ACK，`dup_ack_cnt++`：

- 当 `dup_ack_cnt == 3`：触发快速重传
  - 找到 `out` 中最早未确认段（map 有序，首个 unacked 即 oldest）
  - `ssthresh = max(1, cwnd/2)`
  - `cwnd = ssthresh + 3`
  - 立即重传 oldest 段
- 当 `dup_ack_cnt > 3`：快速恢复阶段每个额外 dupACK 执行 `cwnd += 1`，保持管道不空、避免吞吐骤降。

4.5.4 超时重传（Timeout）

在每一轮循环末尾我都会检查超时（不依赖是否收到 ACK）：

- 找到最早未确认段 oldest
- 若 `now - oldest.last_sent_ms >= RDT_RTO_MS`：
  - `ssthresh = max(1, cwnd/2)`
  - `cwnd = 1`
  - 重传 oldest

这里的定时策略是“**每段记录 last_sent_ms，但只对 oldest 段进行超时判断**”，等价于一个全局 RTO 定时器挂在最早未确认段上，符合 TCP 常见做法，同时实现简单可控。

---

5. 端到端网络交互链路过程（从建连到结束）

我在本地跑一次完整传输时，链路从逻辑上经历如下阶段：

5.1 建立连接

sender 向 router 发送 SYN，router 转发给 receiver。receiver 收到 SYN 后回 SYN|ACK，ACK 通道通常不做丢包/延迟，因此 sender 很快收到 SYN|ACK 并回 ACK，连接进入 EST。

这一阶段我的日志重点观察：

- SYN 是否在重传（若一直重传说明链路/端口配置可能错误）

- receiver 是否打印 `RX SYN -> TX SYN|ACK`
- sender 是否打印 `RX SYN|ACK -> TX ACK. Connected.`

5.2 数据阶段（流水线 + SACK + Reno）

sender 按 `eff_wnd=min(cwnd,fixed_wnd)` 不断填满窗口发送 DATA；receiver 进行按序写入与乱序缓存，同时回 ACK + SACK。
当 router 引入 3% 丢包时，sender 侧会出现：

- 连续 dupACK（累计 ACK 不前进）→ 触发 fast retransmit；
- 或在更糟情况下出现 TIMEOUT → cwnd 归 1 再慢启动。

我在日志中主要看：

- cwnd 是否经历慢启动到阈值，再进入拥塞避免；
- 是否出现 `3 dupACK -> Fast Retransmit`；
- 是否出现 `TIMEOUT -> Retransmit`（出现说明丢包更严重或缺口重传仍丢）。

5.3 关闭连接

当 sender 判断“所有数据段均 acked”后发送 FIN；receiver 收到 FIN 回 ACK 并发送自己的 FIN；sender 收到 FIN 后回 ACK 并退出。
receiver 在 FIN_WAIT 收到 ACK 后打印接收耗时并退出。

---

6. 输出与统计指标

6.1 时间与吞吐率

sender 记录：

- `start_ms = now_ms()`（握手完成后开始）
- `end_ms = now_ms()`（数据完成并关闭后）
- `sec = (end_ms-start_ms)/1000`
- `throughput = file_size_MB / sec`

receiver 记录：

- `start_ms`（握手完成后进入 EST）
- `end_ms`（关闭完成）
- 输出接收时间

6.2 日志策略

我统一使用 `LOG()` 输出带时间戳的日志：

- 时间戳来自 `steady_clock` 的毫秒数，用于体现相对时间顺序；
- 日志覆盖握手、Reno 阶段、dupACK/超时重传、FIN 重传、最终吞吐等关键点；
- 由于每个 ACK 都打印会很“吵”，我在 receiver 侧默认关闭 ACK 日志（可按需打开）。

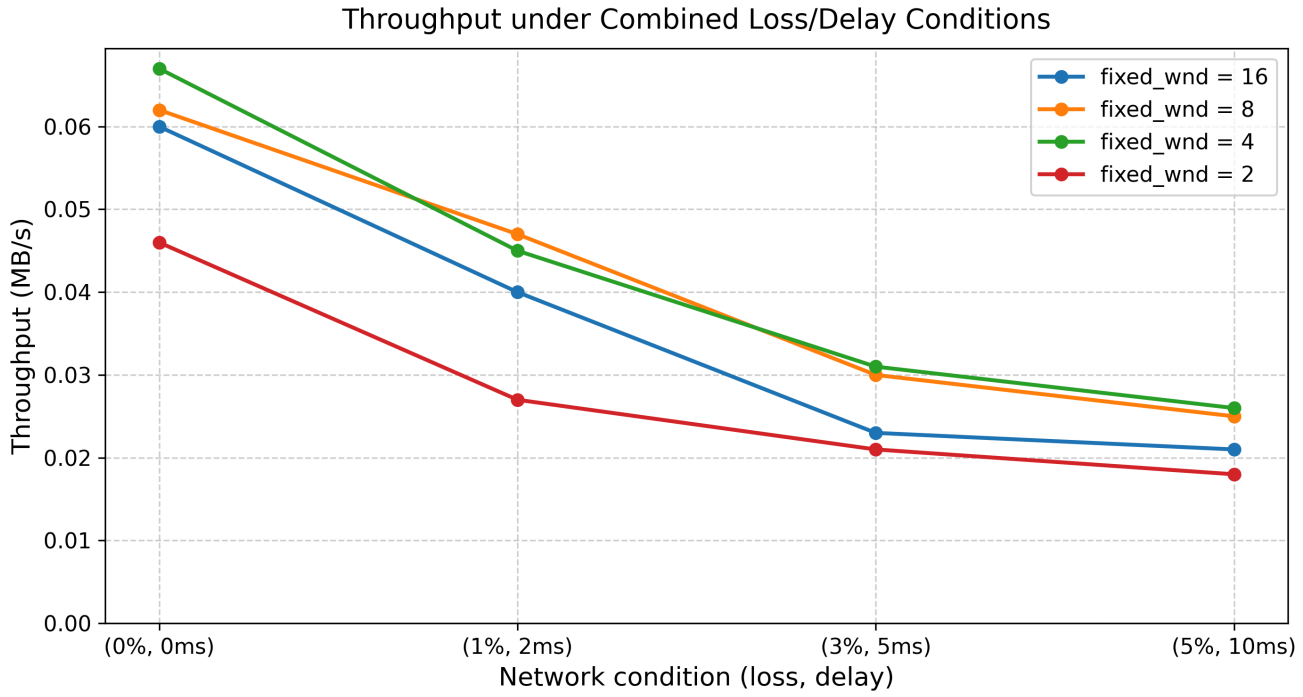
7. 本地实验方法与现象分析（fixed_wnd × loss/delay）

我在主要功能代码完成后，额外记录了实验数据并用python脚本进行了可视化绘制。我记录了在各种 **窗口大小 fixed_wnd** 和 **网络条件（loss%, delayms）**下的 **吞吐量** 和 **cwnd 随时间的变化趋势**

本实验固定文件（采用helloworld.txt）与实现参数，测试参数：窗口大小 **fixed_wnd ∈ {16, 8, 4, 2}** ，四档网络条件 **(loss%, delayms) = (0,0)、(1,2)、(3,5)、(5,10)**

7.1 fixed_wnd 对吞吐的影响（见如下吞吐折线图）

吞吐量MB/s		窗口大小 fixed_wnd			
		16	8	4	2
网络情况 loss/delay	0%，0ms	0.060	0.062	0.067	0.046
	1%，2ms	0.040	0.047	0.045	0.027
	3%，5ms	0.023	0.030	0.031	0.021
	5%，10ms	0.021	0.025	0.026	0.018

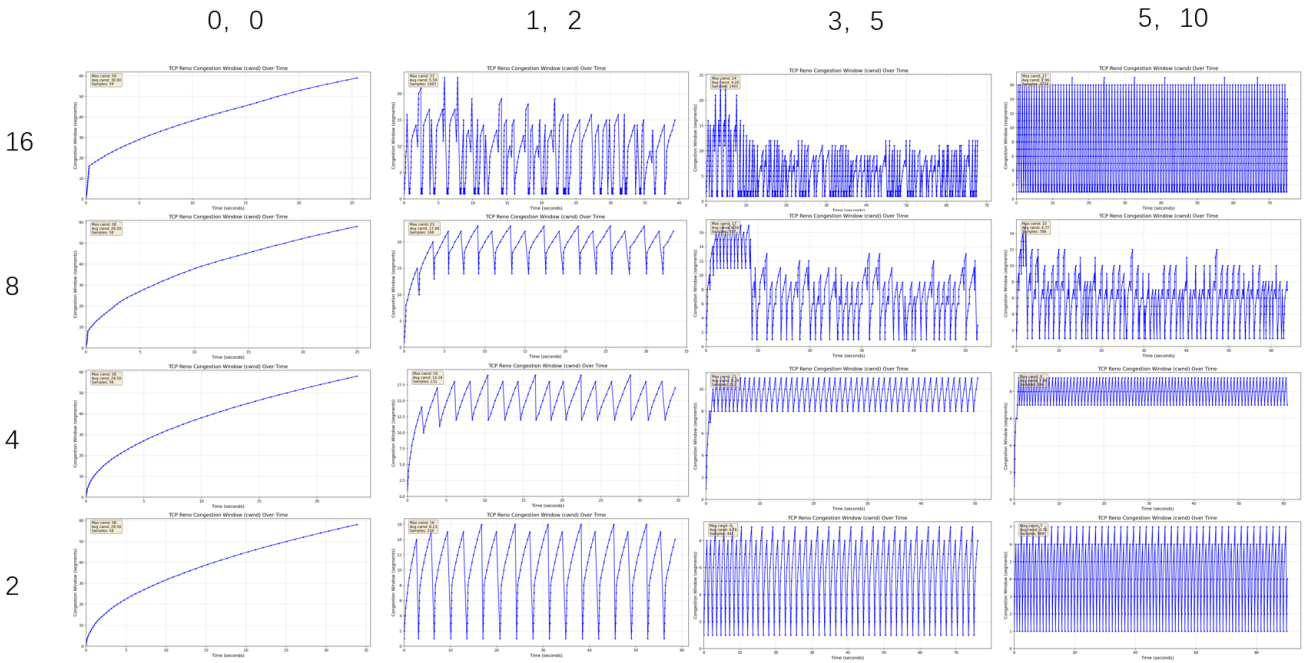


整体上，网络从 (0,0) 恶化到 (5,10) 时，四条曲线均明显下降（吞吐从约 0.06 降到约 0.02 MB/s 量级）。不同窗口的对比呈现出“中等窗口更优”的特征：

- 在 **(0,0)** 下吞吐最高出现在 **fixed_wnd=4**（0.067 MB/s），16/8 接近；
- 在 **(1,2)** 下 **fixed_wnd=8** 最优（0.047 MB/s）；
- 在更差网络 **(3,5)、(5,10)** 下 **fixed_wnd=4** 始终最高（0.031、0.026 MB/s）；
- **fixed_wnd=2** 全程偏低，说明窗口过小会限制并发度，难以填满链路。

这表明：在存在丢包/时延时，过大的 fixed_wnd 并不一定带来收益，最佳窗口通常落在 **4～8**。

7.2 结合 cwnd 曲线解释原因（见如下 cwnd 随时间图）



- **(0,0)**：四种 fixed_wnd 下的 cwnd 曲线整体平滑上升、几乎无明显回退，说明链路稳定时拥塞窗口能持续增长，因此吞吐主要由发送节奏/实现开销等因素决定，窗口差距不大。
- **(1,2) 到 (5,10)**：随着丢包和 RTT 增大，cwnd 曲线逐渐转为明显的锯齿/频繁下跌，反映出更高的丢包触发（快速重传/恢复）甚至更激烈的窗口回退，导致有效在飞数据下降、吞吐降低。尤其在 **fixed_wnd=16** 时，波形更“乱”、回退更频繁，对应吞吐下降最明显；而 **fixed_wnd=4/8** 的波动相对更可控，平均吞吐反而更高。

结论：

- 吞吐折线图给出结论：网络越差吞吐越低、最佳 fixed_wnd 在 4～8；
- cwnd 曲线进一步说明原因：在有损/有延时条件下，窗口过大更容易诱发频繁回退与不稳定恢复，使平均吞吐被拉低。

---

8. 总结与实现取舍

本实验中我在用户空间实现了一个面向连接的可靠传输协议，核心机制覆盖了实验要求：三次握手/挥手的连接管理、checksum 的差错检测、流水线传输与 SACK 选择确认、固定窗口流控、以及 Reno 拥塞控制（慢启动、拥塞避免、快速重传/恢复、超时重传）。

为了可读性与实验可控性，我做了一些简化：

- RTO 使用固定值（300ms），未实现 RTT 估计与自适应 RTO；
- 超时定时器采用“最早未确认段单定时器语义”，但保留每段 last_sent_ms；
- receiver 的 ACK 日志默认不全量输出，保证主要机制更易观察。

整体上该实现能在 router 模拟丢包/延时环境下完成给定文件传输，并通过日志清晰呈现 Reno 状态变化与重传行为，满足实验对“可靠性、可观测性与可分析性”的要求。