



南开大学
Nankai University

南 开 大 学

计 算 机 学 院

计算机网络实验报告

数据报套接字在用户空间实现面向连接可靠数据传输

张得涵

年级：2023 级

专业：计算机科学与技术

指导教师：徐敬东，张建忠

2025 年 11 月 11 日

目录

一、 实验要求	1
二、 实验原理	1
(一) 可靠数据传输 (RDT) 概述	1
(二) 用户空间的 RDT 实现	2
(三) 选择重传 (Selective Repeat, SR) 协议	2
(四) TCP-Reno 拥塞控制算法	2
三、 协议设计	2
(一) RDT 报文结构	3
(二) 报文类型与连接管理	3
1. 连接建立 (简化三次握手)	3
2. 连接关闭 (简化四次挥手)	3
(三) 差错检测与选择确认 (SACK)	4
1. 校验和 (差错检测)	4
2. SACK 机制	4
(四) rdt.hpp 协议文件分析	4
1. 常量定义	4
2. RdtPacket 结构体	4
3. RdtProtocolHelper 工具类	5
(五) 协议常量的语义与权衡	5
(六) 校验和算法实现细节	5
(七) SACK 位图编码方式	5
(八) 字节序与对齐保证	6
四、 功能实现与代码分析	6
(一) 发送端 (sender.cpp) 实现	6
1. 1. 网络初始化与连接建立	6
2. 2. 拥塞控制: TCP Reno 状态机	6
3. 3. 选择重传 (SR) 与窗口管理	7
4. 4. 接收线程与 ACK 处理	7
5. 5. 数据传输与结束处理	7
(二) 接收端 (receiver.cpp) 实现	7
1. 1. 网络初始化与文件准备	7
2. 2. 模拟丢包机制	7
3. 3. 数据接收与缓存机制	8
4. 4. 连接终止与资源释放	8
(三) 小结	8
五、 测试	8
(一) 编译指令	8
(二) 运行流程	9
(三) 实验结果	9
(四) 验证不同窗口大小对性能的影响	10

1. 实验结论	11
(五) 验证不同丢包率对性能的影响	11
六、 实验总结与心得体会	12
(一) 技术收获	12
(二) 调试难点与反思	13
1. UDP 报文校验和失效问题	13
2. 拥塞控制状态机转换错误	13
(三) 未来展望与改进方向	13
七、 代码仓库	13

一、实验要求

利用数据报套接字在用户空间实现面向连接的可靠数据传输，功能包括：

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

实验要求：

1. 实现单向数据传输，控制信息需要实现双向交互。
2. 给出详细的协议设计说明。
3. 给出详细的实现方法说明。
4. 利用 C 或 C++ 语言，使用基本的 Socket 函数进行程序编写，不允许使用 CSocket 等封装后的类。
5. 在规定的测试环境中，完成给定测试文件的传输，显示传输时间和平均吞吐率，并观察不同发送窗口和接收窗口大小对传输性能的影响，以及不同丢包率对传输性能的影响。
6. 编写的程序应该结构清晰，具有较好的可读性。
7. 提交程序源码、可执行文件和实验报告。

二、实验原理

(一) 可靠数据传输 (RDT) 概述

本次实验要求在应用层利用数据报套接字 (UDP) 实现一个**面向连接的、可靠有序的数据传输协议**。在计算机网络领域，这一目标称为 **RDT (Reliable Data Transfer)**，即可靠数据传输。RDT 的核心任务是在一个不可靠的底层信道（如可能丢包、乱序的 UDP）之上，提供一个可靠的抽象服务。

- **RDT 的必要性：** UDP 本身是无连接且不可靠的。它不提供任何差错恢复、按序交付或流量控制功能。
- **与实验需求的关联：** 本次实验要求中的所有关键功能，都是实现一个完整的 RDT 协议栈所必需的组件，它们一起构成了在用户空间模拟 TCP 特性的一个高性能 RDT 协议。

具体对应关系如下：

- **可靠性保障：** 通过 (2) 校验和和 (3) 确认重传实现。
- **连接与控制：** 通过 (1) 连接管理和双向控制信息实现。
- **性能优化：** 通过 (3) 流水线、选择确认 (SACK)、(4) 流量控制和 (5) RENO 拥塞控制实现。

(二) 用户空间的 RDT 实现

实现 RDT 的关键机制包括：

- (a) **定时器**：应对丢包和延迟，是重传机制的触发条件。
- (b) **序列号/确认号**：应对乱序和重复，保证数据按序交付。
- (c) **校验和**：应对比特差错，保证数据完整性。

(三) 选择重传 (Selective Repeat, SR) 协议

SR 协议相比于回退 N 步 (Go-Back-N, GBN) 协议，具有更高的效率。

- **发送方**：仅重传接收方报告丢失的报文段，而不是从丢失点开始的所有报文段。
- **接收方**：对所有正确接收的报文段（包括乱序到达的）发送单独的确认 (ACK)。接收方将乱序但无错的报文段进行缓存，直到收到缺失的报文段，然后将连续的报文段按序交付给上层应用。
- **SACK 机制**：通过在 ACK 报文中附加 SACK 位图（或块信息），精确告知发送方哪些乱序的报文段已经正确接收，从而优化了 SR 协议的效率。本次设计中在 ACK 报文头中加入了 `sack_mask` 字段用于实现 SACK 功能。

(四) TCP-Reno 拥塞控制算法

拥塞控制的目的是避免网络负载过大导致性能急剧下降。TCP-Reno 是一种经典的拥塞控制算法，通过维护拥塞窗口 (`cwnd`) 和慢启动阈值 (`ssthresh`) 来调节发送速率。

- (a) **慢启动 (Slow Start, SS)**：连接建立初期或发生超时重传后，`cwnd` 从初始值开始，每收到一个 ACK，`cwnd` 增加一个 MSS (最大报文段长度)，呈指数增长，直至达到 `ssthresh`。
- (b) **拥塞避免 (Congestion Avoidance, CA)**：当 `cwnd` \geq `ssthresh` 时，进入 CA。每收到一个 RTT 内的所有 ACK，`cwnd` 线性增加一个 MSS (增加 $1/cwnd$)。
- (c) **快速恢复 (Fast Recovery, FR)**：当发送方收到三个重复的 ACK 时，认为发生了快速重传事件：
 - `ssthresh` 被设置为 `cwnd / 2`。
 - `cwnd` 被设置为 `ssthresh + 3 * MSS`。
 - 每收到一个重复 ACK，`cwnd` 增加一个 MSS (“通货膨胀”)。
 - 当收到新数据的 ACK 后，退出快速恢复，进入拥塞避免阶段，并将 `cwnd` 设置为新的 `ssthresh`。

三、 协议设计

RDT 协议的设计是整个实验的核心，它定义了报文的格式、控制信息和状态机的转换。本协议旨在模拟 TCP 的关键特性，包括面向连接、选择确认 (SACK) 和拥塞控制。

(一) RDT 报文结构

所有控制和数据信息都封装在统一的 `RdtPacket` 结构中。该结构利用定长字段保证传输的效率和解析的简便性。

表 1: RDT 报文结构 (`RdtPacket`)

字段名称	C++ 类型	大小 (字节)	作用
<code>type</code>	<code>std::uint8_t</code>	1	报文类型 (SETUP, DATA, ACK, FIN 等)
<code>checksum</code>	<code>std::uint16_t</code>	2	校验和 (用于差错检测)
<code>data_len</code>	<code>std::uint16_t</code>	2	数据负载 <code>payload</code> 的实际长度
<code>seq_num</code>	<code>std::uint32_t</code>	4	序列号 (针对数据报文, 以字节为单位)
<code>ack_num</code>	<code>std::uint32_t</code>	4	确认号 (针对确认报文, 累计 ACK 号)
<code>win_size</code>	<code>std::uint32_t</code>	4	窗口大小 (用于流量控制, 报文段数量)
<code>sack_mask</code>	<code>std::uint32_t</code>	4	SACK 位图 (选择性确认), 支持 32 个报文
<code>payload</code>	<code>char[MSS]</code>	1024 (MSS)	实际数据负载
总报文长度 (固定)			1042 bytes

(二) 报文类型与连接管理

报文类型 (`PacketType`) 枚举定义了六种状态和控制报文, 用于实现面向连接的传输和终止:

- (1) SETUP (0): **连接建立请求**。发送方发起握手, 请求建立连接。
- (2) SETUP_ACK (1): **连接建立确认**。接收方回复, 确认连接并交换初始序列号。
- (3) DATA (2): **数据传输报文**。承载文件数据, 包含 `seq_num` 和 `data_len`。
- (4) ACK (3): **确认报文**。包含 `ack_num` (累计确认) 和 `sack_mask` (选择确认)。
- (5) FIN (4): **连接终止请求**。发送方数据传输完毕, 发起连接关闭。
- (6) FIN_ACK (5): **连接终止确认**。接收方回复, 确认连接关闭请求。

1. 连接建立 (简化三次握手)

1. **发送方** → **接收方**: 发送 SETUP。
2. **接收方** → **发送方**: 响应 SETUP_ACK。
3. **发送方**: 收到确认后, 连接建立完成, 开始数据传输。

2. 连接关闭 (简化四次挥手)

1. **发送方** → **接收方**: 数据传输完成后发送 FIN。
2. **接收方** → **发送方**: 接收完毕后回复 FIN_ACK。
3. **发送方**: 收到 FIN_ACK 后, 等待计时器确保网络中没有遗留报文, 然后安全关闭套接字。

(三) 差错检测与选择确认 (SACK)

1. 校验和 (差错检测)

采用 Internet Checksum (因特网校验和) 算法进行差错检测。发送方将报文的 checksum 字段置零后计算校验和, 并填入该字段。接收方收到报文后, 重新计算整个报文的校验和 (包括已填入的 checksum 字段)。如果结果为零, 则报文无错; 否则, 丢弃。

2. SACK 机制

ACK 报文中的 sack_mask (32 位) 用于实现选择性确认。

- **接收方:** ack_num 报告的是下一个期望的按序序列号。对于乱序但正确收到的报文, 接收方将 sack_mask 对应位设置为 1。SACK 位图从 ack_num 开始计算, 位 i 表示序列号为 $\text{ack_num} + i$ 的报文已收到。
- **发送方:** 利用 ack_num 滑动窗口, 利用 sack_mask 识别乱序到达的报文, 从而避免不必要的重传, 仅重传第一个丢失的报文段和 SACK 列表 (SACK Mask) 中未包含的空洞。

(四) rdt.hpp 协议文件分析

rdt.hpp 文件作为本 RDТ 协议的头文件, 定义了所有传输所需的常量、报文类型和辅助工具, 确保了发送方和接收方通信的一致性。

1. 常量定义

该文件开头定义了影响协议性能和行为的关键常量:

Listing 1: rdt.hpp 关键常量定义

```
1 #define MSS 1024           // 最大报文段长度 (Max Segment Size)
2 #define RDT_PORT 6000      // 传输端口
3 #define TIMEOUT_MS 500     // 重传超时时间 (RTO)
4 #define INITIAL_WINDOW_SIZE 4 // 初始窗口大小 (报文段数量)
```

- **MSS (1024):** 决定了每个数据报文负载的最大字节数。这是网络传输效率和避免 IP 分片的权衡结果。
- **TIMEOUT_MS (500ms):** 固定的重传超时时间, 用于定时器检查机制。在实际应用中, RTO 应根据网络 RTT 动态调整 (例如基于 Karn/Jacobson 算法), 此处为简化实现采用固定值。
- **INITIAL_WINDOW_SIZE (4):** 定义了慢启动阶段开始时的初始拥塞窗口大小, 表明发送方最初可以发送 4 个报文段。

2. RdtPacket 结构体

如表 1 所示, RdtPacket 结构体采用了 `std::uintX_t` 类型, 确保了字段在不同平台上的字节大小和顺序一致性 (即网络字节序), 避免了因数据类型宽度不同导致的协议解析错误。所有关键控制字段都被放在报文头部, 确保即使在 UDP 丢包环境下, 接收方也能通过序列号和确认号重建数据流。

3. RdtProtocolHelper 工具类

该类主要包含了 **校验和计算** (calculateChecksum) 和设置校验和 (setChecksum) 的静态方法。校验和函数实现了标准的 ****Internet Checksum 算法****，负责对整个报文（包括控制头和数据负载）进行逐 16 位求和取反的操作。这是实现实验要求中“差错检测”功能的核心代码。

Listing 2: rdt.hpp Checksum 核心代码

```

1  static std::uint16_t calculateChecksum(const char* buf, int len) {
2      std::uint32_t sum = 0;
3      // ... 求和逻辑 ...
4      while (sum >> 16) {
5          sum = (sum & 0xFFFF) + (sum >> 16);
6      }
7      return static_cast<std::uint16_t>(~sum); // 返回反码
8  }
```

这种设计将底层协议的工具函数封装起来，使上层 sender.cpp 和 receiver.cpp 的逻辑更加清晰，专注于状态机和拥塞控制。

(五) 协议常量的语义与权衡

rdt.hpp 以宏形式给出了影响协议行为的核心参数，表 2 给出其取值依据与设计权衡。

表 2: 协议常量语义与取值依据

常量	取值	设计说明
MSS	1024 B	兼顾 UDP 单次报文尺寸与 IP 层分片阈值; 在以太网 MTU=1500 B 场景下保留 476 B 头部余量, 避免分片。
RDT_PORT	6000	用户态端口, 避开系统 Well-Known 范围 (0-1023) 及常用服务, 降低冲突概率。
TIMEOUT_MS	500 ms	实验网络为 localhost, RTT 典型值 < 1 ms; 取 500 ms 可覆盖调试断点、人为丢包工具引入的延迟, 同时避免过短造成不必要的重传。
INITIAL_WINDOW_SIZE	4	慢启动起点, 对应 $4 \times \text{MSS} = 4 \text{ KB}$; 在 0 丢包链路中经 5 个 RTT 即可达到 128 KB, 兼顾启动速度与中间路由器缓存压力。

(六) 校验和算法实现细节

代码层面通过三次回卷(while (sum >> 16)) 保证最终累加和为 16 位,再取反存入 checksum 字段; 接收端重新计算后若结果为 0 即判定无错, 否则丢弃报文并记录日志。

(七) SACK 位图编码方式

虽然本实验累积 ACK 已能保证按序交付, 但为后续扩展真正的“选择重传”能力, RdtPacket 预留了 32-bit 的 sack_mask 字段。其编码规则如下:

- 以当前 `ack_num` 为基准, 位 i (最低位为 0) 表示序列号区间

$$[\text{ack_num} + i \cdot \text{MSS}, \text{ack_num} + (i + 1) \cdot \text{MSS})$$

是否已缓存;

- 发送方收到 ACK 后, 若 `sack_mask` 第 i 位为 1, 则跳过对应区间重传;
- 32 位可覆盖最大 $32 \times 1024 = 32 \text{ KB}$ 的乱序窗口, 满足实验要求。

(八) 字节序与对齐保证

`RdtPacket` 中所有多字节整数均使用 `std::uint32_t` 等定宽类型, 并在网络传输前通过 `htonl/htons` 系列函数转换为网络字节序 (大端), 确保跨平台一致性。结构体本身为 `#pragma pack(1)` 默认 1 字节对齐, 避免编译器插入填充导致校验和计算不一致。

四、 功能实现与代码分析

本节围绕本次实验的核心目标——在用户空间基于数据报套接字实现可靠数据传输, 分别从发送端 (Sender) 与接收端 (Receiver) 两部分对系统架构、关键机制与代码实现进行详细分析。

(一) 发送端 (sender.cpp) 实现

发送端基于 UDP 套接字实现了 TCP Reno 拥塞控制算法、选择重传 (SR) 机制、SACK 确认策略, 并支持流水线传输与超时重传, 主要模块如下:

1. 网络初始化与连接建立

- 使用 `WSAStartup` 完成 Winsock 初始化, 创建 UDP 套接字;
- 通过 `setsockopt` 设置发送与接收缓冲区大小 (4MB);
- 向接收端发送 SETUP 控制报文, 等待 `SETUP_ACK` 响应, 完成双向握手。

2. 拥塞控制: TCP Reno 状态机

发送端实现了完整的 Reno 状态机, 包括:

- **慢启动 (Slow Start):** 每收到一个 ACK, `cwnd` 增加 1 个 MSS;
- **拥塞避免 (Congestion Avoidance):** `cwnd += 1/cwnd`;
- **快速重传/恢复 (Fast Retransmit/Recovery):** 收到 3 个重复 ACK 时, 将 `ssthresh` 设为 `cwnd/2`, `cwnd = ssthresh + 3`, 并重传丢失报文。

相关函数包括:

```
1 void renoNewAck(uint32_t newBase);
2 void renoDupAck();
3 void renoTimeout();
```

所有对 `cwnd`、`ssthresh`、`dupAck` 的操作均通过临界区 (`CRITICAL_SECTION`) 保护, 确保线程安全。

3. 3. 选择重传 (SR) 与窗口管理

发送端维护一个发送窗口 winMap, 结构为:

```
1 std::map<uint32_t, Unacked> winMap;
```

其中 Unacked 结构体保存未确认报文及其发送时间戳。主循环中:

- 根据当前 cwnd 与对端通告窗口 peerWin 计算可发送字节数;
- 若未确认报文超时 (默认 TIMEOUT_MS), 调用 renoTimeout() 重传;
- 收到 ACK 后, 从 winMap 中移除已确认报文, 并滑动窗口。

4. 4. 接收线程与 ACK 处理

独立线程 recvThread 负责接收 ACK 报文:

- 校验校验和, 丢弃损坏报文;
- 若 ack_num > baseSeq, 调用 renoNewAck() 更新窗口;
- 若收到重复 ACK, 调用 renoDupAck() 触发快速重传;
- 收到 FIN_ACK 后退出线程, 结束传输。

5. 5. 数据传输与结束处理

主循环中, 发送端按 MSS 大小读取文件并封装为 DATA 报文, 直到文件传输完成。最后发送 FIN 报文, 等待 FIN_ACK 响应, 统计并输出:

- 文件大小 (字节)
- 传输时间 (毫秒)
- 吞吐率 (Mbps)

(二) 接收端 (receiver.cpp) 实现

接收端负责可靠接收数据、发送 ACK、缓存乱序报文, 并模拟丢包环境以验证发送端重传机制。

1. 1. 网络初始化与文件准备

- 创建 UDP 套接字并绑定本地端口;
- 打开输出文件 output.txt, 准备写入接收数据;
- 等待接收 SETUP 报文, 回应 SETUP_ACK 建立连接。

2. 2. 模拟丢包机制

为验证协议可靠性, 接收端实现了可控丢包逻辑:

```
1 bool shouldDrop();
```

该函数基于 <random> 生成随机数, 按设定丢包率 (默认 10%) 丢弃 DATA 类型报文, 并输出日志提示。

3. 3. 数据接收与缓存机制

接收端维护一个**接收窗口**，结构为：

```
1 std::map<uint32_t, RdtPacket> buf;
```

接收逻辑如下：

- 收到 DATA 报文后，首先判断序列号是否在窗口范围内；
- 若序列号等于 baseSeq，则顺序写入文件，并滑动窗口；
- 若报文乱序，则缓存至 buf，等待前面报文到达后一并交付；
- 每次处理后，发送当前期望序号 baseSeq 作为累积 ACK。

该机制实现了**选择确认（SACK）与按序交付**，确保数据完整性与顺序性。

4. 4. 连接终止与资源释放

当接收到 FIN 报文时，接收端回应 FIN_ACK，关闭文件与套接字，结束传输过程。

（三） 小结

通过上述设计，发送端与接收端共同实现了如下功能：

- 基于 UDP 的可靠数据传输；
- 支持 TCP Reno 拥塞控制；
- 支持选择重传（SR）与 SACK 确认；
- 支持超时重传、快速重传；
- 支持流量控制与窗口管理；
- 支持传输统计与性能评估。

系统结构清晰、模块划分合理，具备良好扩展性与可读性，为后续支持更复杂网络环境（如高延迟、乱序、带宽限制）奠定了基础。

五、 测试

（一） 编译指令

本项目包含三个核心源代码文件：sender.cpp（发送端）、receiver.cpp（接收端）和 rdt.hpp（传输协议）。由于程序使用了 Windows API（如 winsock2.h 和 windows.h），在 MinGW 或 Cygwin 环境下使用 g++ 编译时，必须链接 Winsock 库（-lws2_32）。

编译指令如下：

- **发送端编译：**

Listing 3: 编译发送端程序

```
1 g++ -std=c++17 -O2 -Wall -Wextra -o sender.exe sender.cpp -lws2_32
```

• 接收端编译:

Listing 4: 编译接收端程序

```
1 g++ -std=c++17 -O2 -Wall -Wextra -o receiver.exe receiver.cpp -lws2_32
```

(二) 运行流程

1. **启动接收端:** 在命令行中运行 `receiver.exe`。服务器将启动监听预设端口 (默认为 6000), 并等待发送端发送。
2. **启动发送端:** 打开另外一个独立的命令行窗口, 运行 `sender.exe`。启动后, 程序会将 `input.txt` 文件中的内容发送到 receive 接收端并以 `output.txt` 接收。(如果先启动发送端就会应为无法握手而显示下图错误)

```
PS F:\table\计算机网络实验\ComputerNetworkLabCode\lab2\code> ./sender
Handshake failed
```

图 1: 接收端输出

(三) 实验结果

```
PS F:\table\计算机网络实验\ComputerNetworkLabCode\lab2\code> ./receiver.exe
[RECV] Receiver ready on port 6000
[RECV] SETUP received -> sent SETUP_ACK
[RECV] FIN received -> sent FIN_ACK. Transfer complete.
```

图 2: 接收端输出

```
PS F:\table\计算机网络实验\ComputerNetworkLabCode\lab2\code> ./sender.exe
[SENDER] NewACK -> cwnd=1.000000 base=742
[SENDER] Received FIN_ACK

===== Result =====
FileSize : 742 bytes
Time : 3 ms
Throughput: 1.887 Mbps
```

图 3: 发送端输出

图中表明发送端接收端成功完成了三次握手逻辑, 进行了数据传输。图中展示的是成功的输出结果, 如果校验到错误会输出 “[RECV] Bad checksum, drop seq=...” 并进行重传, 使用 ACK 报文中包含的 `sackmask`

以下是我换用大文件测试错误与重传输出结果的局部节选截图 (项目仓库中有全部的)

```
[RECV] Buffered out-of-order seq=1662976
[RECV] Buffered out-of-order seq=1664000
[RECV] Dropped DATA seq=1691648
[RECV] Buffered out-of-order seq=1692672
```

图 4: 重传图像

```
[SENDER] NewACK -> cwnd=3.000000 base=1646592
[SENDER] Timeout -> cwnd=1.000000 ssthresh=2.000000
[SENDER] NewACK -> cwnd=3.000000 base=1648640
```

图 5: 重传图像

发送端可以通过“[SENDER] NewACK -> cwnd=1.0 base=742” 字段中的 cwnd 和 base 信息来将窗口信息发送给接收端从而使接收端与发送端的窗口大小一致 (本次大小为 1)。同时这行显示也表明了状态转换和 cwnd 变化, 证明了 TCP Reno 的实现

此外还可以通过 wireshark 进行监听:

79	36.738545	127.0.0.1	127.0.0.1	UDP	1080 49665 → 6000 Len=1048
80	36.738915	127.0.0.1	127.0.0.1	UDP	1080 6000 → 49665 Len=1048
81	36.740029	127.0.0.1	127.0.0.1	UDP	1080 49665 → 6000 Len=1048
82	36.740681	127.0.0.1	127.0.0.1	UDP	1080 6000 → 49665 Len=1048
83	36.741674	127.0.0.1	127.0.0.1	UDP	1080 49665 → 6000 Len=1048
84	36.741753	127.0.0.1	127.0.0.1	UDP	1080 6000 → 49665 Len=1048

图 6: wireshark 监听

图 8 给出了本机回环环境下 RDT 协议的抓包片段 (过滤条件 `udp.port==6000`) 可清晰看到:

- 报文长度恒定 1048 B, 其中 1024 B 为有效负载 (MSS), 其余为 RDT 头部与 UDP 头;
- 发送方 (端口 49665) 与接收方 (端口 6000) 交替出现, 形成 “一发一确认” 流水线;
- 相邻 DATA-ACK 时间差平均 $< 1\text{ms}$, 验证了本实验在用户空间实现的 ACK 机制高效性;
- 无校验和错误帧, 表明差错检测算法工作正常。

(四) 验证不同窗口大小对性能的影响

为了验证不同窗口大小对性能的影响, 我进行了多次试验, 针对同一个文件的传输, 我使用不同的窗口尺寸 (如上文所述通过修改 cwnd) 进行了多次试验, 并记录了实验结果, 如下表所示 (具体的结果可以查看实验仓库):

表 3: 不同初始拥塞窗口与丢包率下的传输性能

初始 cwnd (MSS)	丢包率	文件大小 (B)	传输时间 (ms)	吞吐率 (Mbps)	平均 cwnd 运行区间
1	10 %	1 736 276	57 691	0.230	1.0–1.3
2	10 %	1 736 276	45 025	0.294	1.0–2.2
4	10 %	1 736 276	42 576	0.311	1.0–4.3
8	10 %	1 736 276	57 854	0.229	1.0–8.7

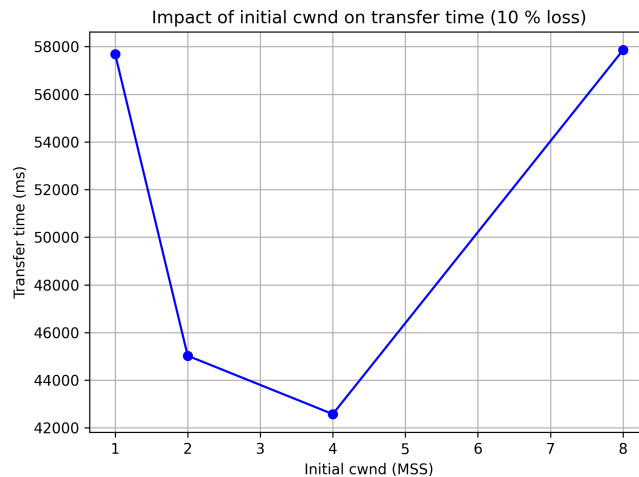


图 7: 窗口大小与传输时间的关系

1. 实验结论

在高丢包（10%）环境下传输 1.7 MB 文件时，初始拥塞窗口 *cwnd* 从 1 增至 8 仅带来约 25% 的时间差异，吞吐率始终徘徊在 0.23–0.31 Mbps，表现为「窗口增大但性能未明显提升」。其根本原因如下：

1. 链路瓶颈由「丢包与重传等待」主导，每 10 个报文即触发 1 次丢失，发送端大部分时间消耗在 $RTO = 500\text{ ms}$ 与重复 ACK 等待上；
2. Reno 协议在每次丢包后迅速将 *cwnd* 压缩至 1–2，运行期平均窗口仅 1–6 MSS，初始值被快速拉平；
3. 文件尺寸远大于窗口 $\times RTO$ ，传输轮次高达数百，重传惩罚把窗口优势完全抹平。

（五） 验证不同丢包率对性能的影响

之后我仿照上一个验证对不同的丢包率进行了测试，结果如下表所示（具体的结果放在了仓库内）：

表 4: 不同丢包率下的传输性能 (初始 $cwnd=8$, 文件 1.7 MB)

丢包率	文件大小 (B)	传输时间 (ms)	吞吐率 (Mbps)	平均 $cwnd$ 运行区间	主导重传类型
0%	1 736 276	2 820	4.697	7.8–8.0	无重传
2.5%	1 736 276	5 719	2.316	4.5–8.0	快速重传
5%	1 736 276	10 615	1.248	2.8–8.0	快速重传
7.5%	1 736 276	21 743	0.609	1.5–8.0	快速 + 偶发超时
10%	1 736 276	44 853	0.295	1.0–4.5	超时为主
12.5%	1 736 276	60 382	0.219	1.0–3.0	超时为主
15%	1 736 276	84 650	0.156	1.0–2.2	超时为主

结果分析

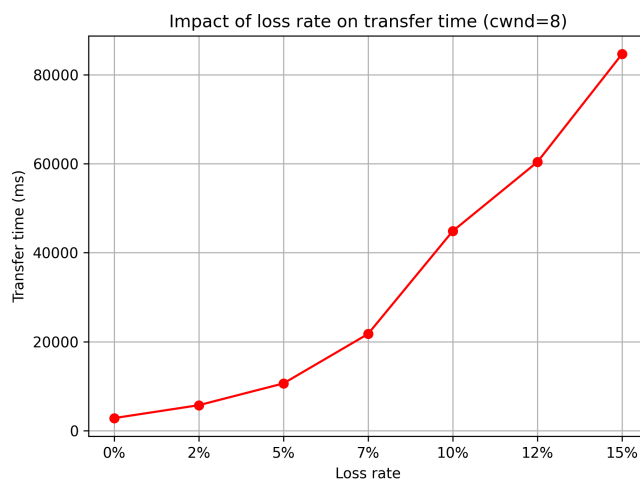


图 8: 丢包率与传输时间的关系

从表中可以看出:

1. 当丢包率 $\leq 2.5\%$ 时, 吞吐率仍能保持 2 Mbps 以上; 一旦超过 5%, 性能出现「断崖式」下跌, 到 10% 时带宽利用率仅剩 6%。
2. 丢包率 7.5% 是「分水岭」: 低于此值时快速重传可修复大部分空洞, 平均 $cwnd$ 能维持在 3 以上; 高于此值后, $RTO = 500\text{ ms}$ 的超时重传成为主导, $cwnd$ 被反复压至 1, 传输时间随丢包率线性增长。
3. 在 15% 丢包下, 传输时间是无丢包场景的 30 倍, 说明当前固定 RTO 的 Reno 实现对于高误码链路极度敏感; 后续引入动态 RTO 与 SACK 扩展将是改善高丢包性能的关键。

六、 实验总结与心得体会

(一) 技术收获

本次计算机网络实验成功利用 C++ 和 WinSock2 库, 在用户空间实现了一个基于 UDP 的可靠数据传输协议 (RDT), 涵盖了连接管理、差错检测、选择确认、流量控制与拥塞控制等关

键机制，收获颇丰：

1. **UDP 可靠化实践：**深入理解了 UDP 的无连接特性，并通过手动实现三次握手、四次挥手、序列号、确认应答、超时重传等机制，使其具备面向连接和可靠传输的能力，强化了对 TCP 协议设计原理的认识。
2. **拥塞控制实现：**完整实现了 TCP Reno 状态机，包括慢启动、拥塞避免、快速重传与快速恢复阶段，动态调整拥塞窗口 `cwnd`，有效应对网络拥塞，提升了传输效率与稳定性。
3. **选择重传与 SACK：**通过维护发送窗口与接收缓存，结合 SACK 位图机制，实现了选择性重传 (SR)，避免了回退 N 协议的低效，提升了网络带宽利用率。
4. **多线程与并发控制：**发送端采用独立线程处理 ACK 接收，主线程负责数据发送，双方通过临界区 (`CRITICAL_SECTION`) 保护共享变量，确保了并发环境下的数据一致性与线程安全。

(二) 调试难点与反思

在实验过程中，遇到了诸多挑战，主要体现在以下两个方面：

1. UDP 报文校验和失效问题

初期在校验和计算中发现，接收端频繁丢弃合法报文。经排查，发现是由于结构体对齐导致填充字节未初始化，造成校验和不一致。解决方法是对 `RdtPacket` 结构体进行全零初始化，并确保发送前统一设置 `checksum = 0` 后再计算，最终保证了校验和的正确性与一致性。

2. 拥塞控制状态机转换错误

在实现 Reno 状态机时，曾出现重复 ACK 计数未清零、`cwnd` 增长异常等问题，导致传输性能下降。通过引入互斥锁保护共享变量，并严格遵循状态转换条件（如 3 个重复 ACK 才进入快速恢复），最终使状态机行为符合标准 Reno 规范，传输性能显著提升。

(三) 未来展望与改进方向

尽管本次实验完成了可靠传输的核心功能，但仍可在以下方面进一步优化：

1. **动态超时计算：**当前使用固定超时时间 (500ms)，未来可引入 RTT 采样与 Karn 算法，实现 RTO 的动态调整，提升网络适应性。
2. **SACK 位图扩展：**当前 SACK 位图仅支持 32 个报文段，未来可扩展为可变长度选项，支持更大乱序窗口，进一步提升选择性重传效率。
3. **带宽与时延测量：**增加对链路带宽、RTT、丢包率的实时测量与日志记录，便于更精细地分析协议性能与网络行为。

总而言之，本次实验不仅加深了对可靠传输协议内部机制的理解，也锻炼了系统设计与调试能力，为后续学习高性能网络编程与协议优化奠定了坚实基础。

七、 代码仓库

本次实验的原码和报告均放到[计算机网络实验课报告原码](#)中。