南开大学
Nankai University

南　　开　　大　　学

计　算　机　学　院

计算机网络实验报告

实验 2：设计可靠传输协议并编程实现

贾景顺

年级：2022 级

专业：计算机科学与技术

指导教师：张建忠　徐敬东

2025 年 12 月 27 日

摘要

针对传输层 UDP 协议无连接、不可靠的特性，本实验旨在应用层构建一套面向连接的可靠文件传输系统。实验首先设计了包含序列号、确认号及校验和的自定义报文结构，并基于有限状态机完整实现了三次握手建立连接与四次挥手断开连接的逻辑。核心工作在于实现了 TCP Reno 拥塞控制算法，通过维护拥塞窗口（cwnd）和慢启动门限（ssthresh），精确模拟了慢启动、拥塞避免及快恢复三个阶段的状态流转。

在传输机制上，采用滑动窗口配合累积确认策略以提升信道利用率，接收端利用缓冲区有效处理了数据包的乱序到达问题。同时，系统引入了超时重传与基于重复 ACK 的快速重传双重保障机制，以应对网络丢包。实验结果表明，该系统能够在模拟高延迟和丢包的网络环境中，动态调整发送速率，实现高效、完整且准确的文件数据传输。

关键字：UDP；可靠传输；TCP Reno；拥塞控制；滑动窗口；快速重传

目录

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。

二、 运行环境

本实验的编译与测试在 Windows11 系统下进行，采用 Visual Studio Code 并配置 Microsoft Visual C++ (MSVC) 编译器（cl.exe）进行代码的构建。

另外,在网络编程中,链接器必须将程序与 Windows Sockets 2 库($ws2\_32.lib$)进行链接才能解析 *socket*, *bind*, *connect* 等符号。在本实验的代码实现中，通过在源码头部添加预处理指令 $\#pragma\ comment(lib, "ws2\_32.lib")$，指示链接器自动在默认库路径中搜索并链接该静态库。

三、 协议设计与实现

3.1 报文设计

为了在无连接、不可靠的 UDP 协议之上实现面向连接的可靠数据传输，本实验设计了一套自定义的应用层报文格式，模拟 TCP 协议段的行为特征。该报文结构严格遵循字节紧凑排列原则，在 C++ 实现中通过 `#pragma pack(1)` 指

令强制取消编译器的内存对齐填充，以确保二进制数据在网络传输过程中的一致性与跨平台兼容性。

报文整体由固定长度的头部和可变长度的数据载荷两部分组成。头部长度固定为 16 字节，包含维持可靠传输所需的关键控制信息。首部的前两个字段分别为 32 位的序列号（Sequence Number）与 32 位的确认号（Acknowledgment Number）。序列号用于标识数据流中每一个字节的顺序，解决了 UDP 传输中常见的乱序到达与重复接收问题；确认号则配合累积确认机制（Cumulative ACK），告知发送端接收方期望收到的下一个字节序号，从而实现丢包检测与滑动窗口推进。

紧随其后的是 16 位的标志位字段（Flags）、16 位的校验和（Checksum）、16 位的窗口大小（Window）以及 16 位的数据长度（Length）。标志位采用位掩码设计，分别定义了 SYN (0x01)、ACK (0x02)、FIN (0x04) 和 DATA (0x08) 四种状态。校验和字段采用标准的互联网 16 位反码求和算法，覆盖头部与数据部分，接收端通过校验该字段判定数据在传输过程中是否发生比特翻转，一旦校验失败则直接丢弃报文。数据长度字段指示了载荷的实际字节数，本实验将最大段大小（MSS）设定为 14600 字节，以减少头部开销并适应实验环境的高吞吐需求。

<table>
<tr><td colspan="32">31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td></tr>
<tr><td colspan="32">Sequence Number (32 bits)</td></tr>
<tr><td colspan="32">Acknowledgment Number (32 bits)</td></tr>
<tr><td colspan="16">Flags (16 bits)</td><td colspan="16">Checksum (16 bits)</td></tr>
<tr><td colspan="16">Window Size (16 bits)</td><td colspan="16">Data Length (16 bits)</td></tr>
<tr><td colspan="32">Data (Variable Length, Max 14600 Bytes)
...</td></tr>
</table>

图 1: 自定义可靠传输协议报文格式

3.2 三次握手

连接建立是可靠数据传输的前提，也是 TCP 协议的核心特征之一。本实验在正式传输文件数据之前，设计并实现了一套严格的三次握手（Three-way

Handshake）机制，旨在让通信双方在不可靠的 UDP 信道上确认彼此的存活状态，并同步初始序列号（Initial Sequence Number, ISN）。该过程由发送端（Sender）发起，接收端（Receiver）被动响应，通过交换带有特定标志位的控制报文来协商连接状态。

握手的第一阶段始于发送端。发送端构造一个标志位 SYN 置 1 的报文，将初始序列号设定为 0，发送给接收端，接收端在绑定端口后一直处于监听模式，一旦收到 SYN 报文且校验和验证通过，便认为有新的连接请求。接收端随即初始化本地的乱序缓冲区与状态变量，计算期望序列号为接收到的序列号加 1。

紧接着进入第二阶段，接收端向发送端回复一个 SYN+ACK 报文。该报文同时置位 SYN 和 ACK 标志，确认号（Ack）填充为 1，表明接收端已准备好接收序号为 1 的数据。

在第三阶段，发送端在验证 SYN+ACK 无误后，向接收端发送最后一个 ACK 报文，该报文序列号为 1，确认号为接收到的序列号加 1。发送完该确认报文后，发送端将自身的发送窗口基准（Base）和下一个发送序号（NextSeqNum）均更新为 1，标志着连接正式建立。至此，双方完成了状态同步，随后立即进入文件传输阶段。

下图展示了该三次握手过程的时序交互逻辑：

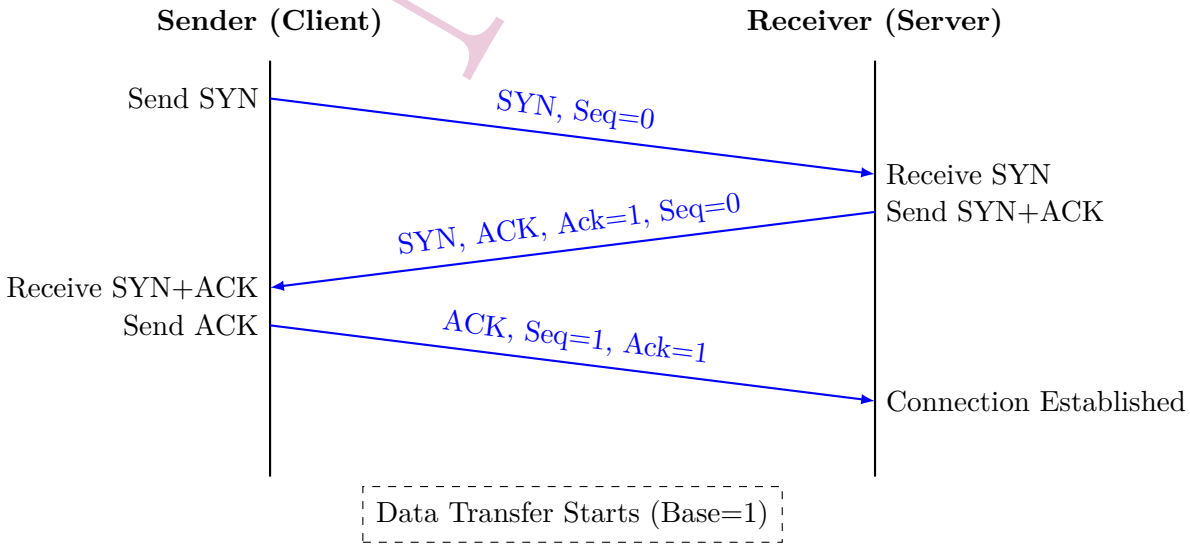

图 2: 基于 UDP 的三次握手时序图

3.3 文件传输

在三次握手成功建立连接后，系统进入核心的文件传输阶段。本实验采用了基于滑动窗口（Sliding Window）的可靠传输机制，以解决 UDP 协议固有的丢包和乱序问题，同时利用流水线技术提高信道利用率。发送端将待传输的文件视为一个字节流，根据最大段大小（MAX_MSS）将其切分为若干个数据包，每个数据包头部携带标志位 DATA 及对应的序列号。

发送端的传输逻辑由拥塞窗口（cwnd）和窗口基准（Base）共同控制。在传输循环中，只要下一个待发送的序列号（NextSeqNum）位于允许的发送窗口内（即小于 Base + cwnd），发送端便会持续从文件中读取数据，封装成报文并立即发送。为了支持重传机制，所有已发送但尚未收到确认的报文会被暂时存入发送缓冲区（Send Buffer）队列中。发送端同时采用非阻塞或极短超时的轮询机制接收来自接收端的 ACK 确认包，一旦收到确认号大于当前 Base 的 ACK，表明该确认号之前的所有数据均已被接收端正确接收，发送端随即右移滑动窗口（更新 Base），从缓冲区中移除已确认的报文，并尝试发送新的数据包。

接收端维护一个期望序列号变量（ExpectedSeq），用于指示当前等待接收的字节流起始位置。每当收到一个 DATA 报文，接收端首先校验其校验和与序列号。如果收到的序列号严格等于 ExpectedSeq，说明数据按序到达，接收端将其直接写入目标文件，并检查乱序缓冲区（Out-of-Order Buffer）中是否存在后续连续的数据块，若存在则一并写入并更新 ExpectedSeq。如果收到的序列号大于 ExpectedSeq，说明中间发生了丢包或乱序，接收端将该报文暂存于乱序缓冲区中，不进行文件写入。无论接收情况如何，接收端始终向发送端回复当前 ExpectedSeq 作为确认号，这种累积确认机制确保了发送端能够准确感知数据流的连续接收状态，是触发快速重传的基础。

下图展示了发送端滑动窗口的逻辑结构：

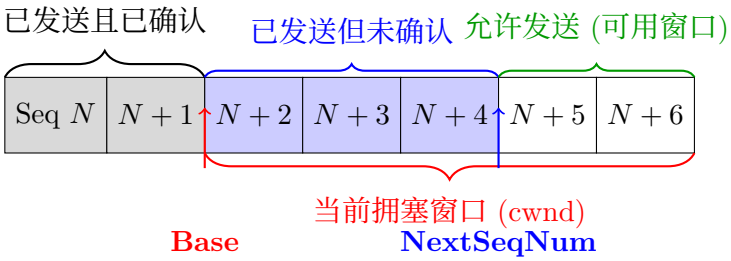

图 3: 发送端滑动窗口机制示意图

3.4 Reno 算法

拥塞控制是保证网络共享资源公平性与传输稳定性的关键机制。本实验在应用层完整实现了 TCP Reno 拥塞控制算法，通过动态调整发送端的拥塞窗口（cwnd）和慢启动门限（ssthresh）来适应不断变化的网络带宽。系统维护了一个有限状态机，包含慢启动（Slow Start）、拥塞避免（Congestion Avoidance）和快恢复（Fast Recovery）三种核心状态。

传输伊始，发送端处于 ** 慢启动 ** 状态，拥塞窗口初始化为 1 个最大段大小（MSS），慢启动门限设为 64 KB。在该阶段，每收到一个确认新数据的 ACK，拥塞窗口便增加 1 MSS，从而实现发送速率的指数级增长，以快速探查网络可用带宽。当拥塞窗口增长至超过慢启动门限时，状态机自动迁移至 ** 拥塞避免 ** 阶段。在此阶段，算法采取加性增（Additive Increase）策略，每收到一个 ACK，窗口仅增加 $MSS \times (MSS/cwnd)$，即每个往返时间（RTT）窗口仅增加约 1 MSS，从而避免过快触发网络拥塞。

为了区分不同程度的数据丢失，本实验实现了 Reno 算法标志性的 ** 快速重传与快速恢复 ** 机制。当发送端连续收到 3 个重复的 ACK（Duplicate ACKs）时，系统判定网络发生了轻微拥塞或随机丢包，而非严重阻塞。此时，算法不等待重传计时器超时，而是立即执行快速重传：将慢启动门限减半（$ssthresh = cwnd/2$），并将拥塞窗口设置为新的门限值加 3 MSS（$cwnd = ssthresh + 3MSS$），随即进入 ** 快恢复 ** 状态并立即重传丢失的数据包。在快恢复期间，每收到一个重复 ACK，窗口增加 1 MSS；一旦收到确认新数据的 ACK，说明丢包已修复，算法将拥塞窗口重置为慢启动门限值，并退回拥塞避免状态。

与之相对，若发生超时事件（Timeout），则被视为网络严重拥塞的信号。无论当前处于何种状态，系统都将执行严厉的惩罚措施：将慢启动门限设为当前窗口的一半，并将拥塞窗口重置为 1 MSS，状态强制回退至慢启动，重新开始网络探测过程。

下图展示了本实验实现的 Reno 算法状态转移逻辑：

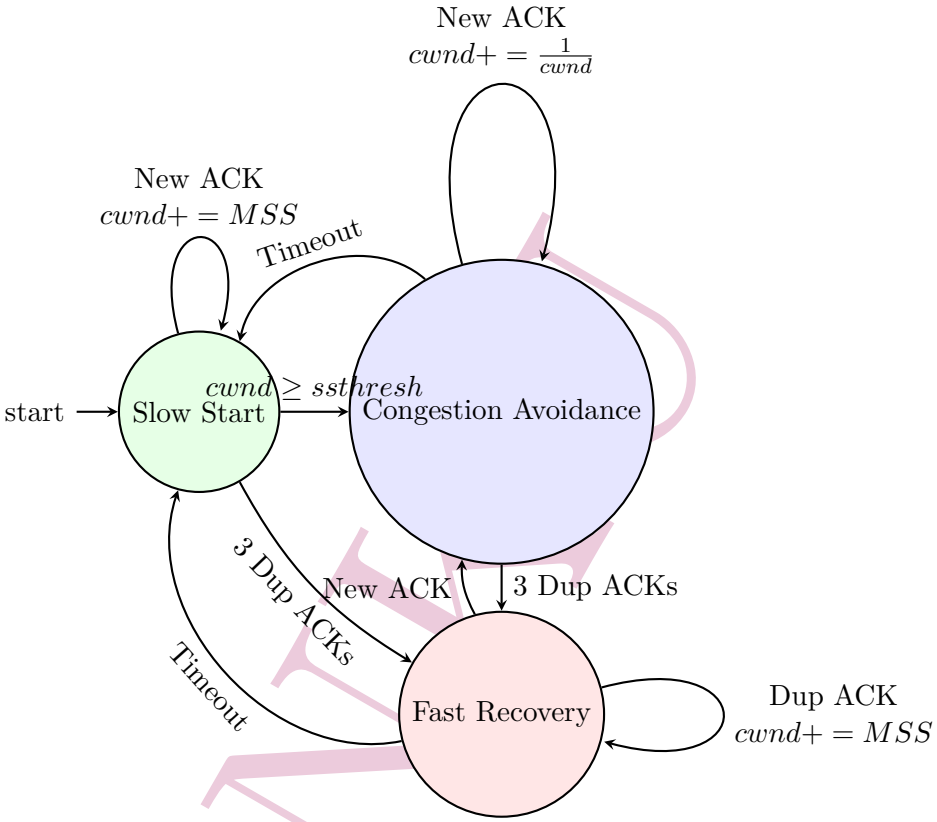

图 4: TCP Reno 拥塞控制状态转移图

3.5 四次挥手

数据传输任务完成后，为了确保通信双方都能优雅地释放资源并确认对方已停止发送数据，本实验设计了严格的四次挥手（Four-Way Handshake）断开连接机制。该过程由主动关闭方（通常为发送端）发起，被动关闭方（接收端）响应，通过交换四个控制报文来终止全双工连接。

断开过程的第一阶段由发送端在文件数据发送完毕后触发。发送端构造一个置位 FIN 标志的报文，其序列号紧接在最后一个数据报文之后，发送给接收端，

随即进入等待确认状态。接收端收到该 FIN 报文后，识别出这是连接终止请求，立即向发送端回复一个 ACK 报文，确认号为收到的 FIN 序列号加 1，表明已知晓发送端结束了数据发送。此时，连接处于“半关闭”状态，发送端不再发送数据，但仍可接收数据。

紧接着，接收端执行自身的关闭流程。在关闭文件输出流并释放相关缓冲区资源后，接收端向发送端发送自己的 FIN 报文，请求关闭反方向的连接。发送端在收到接收端的 FIN 报文后，向接收端回复最后一个 ACK 报文。为了应对不可靠网络环境下的丢包风险，代码逻辑中在此阶段引入了超时重传机制：发送端在发出 FIN 后若在 1000ms 内未收到 ACK，会触发重传；同样，接收端若未收到最后的 ACK，也会重传其 FIN 报文。只有当发送端发出最后的 ACK 且接收端收到该确认后，双方的 Socket 资源才会被彻底释放，至此连接完全关闭。

下图展示了该四次挥手过程的时序交互逻辑：

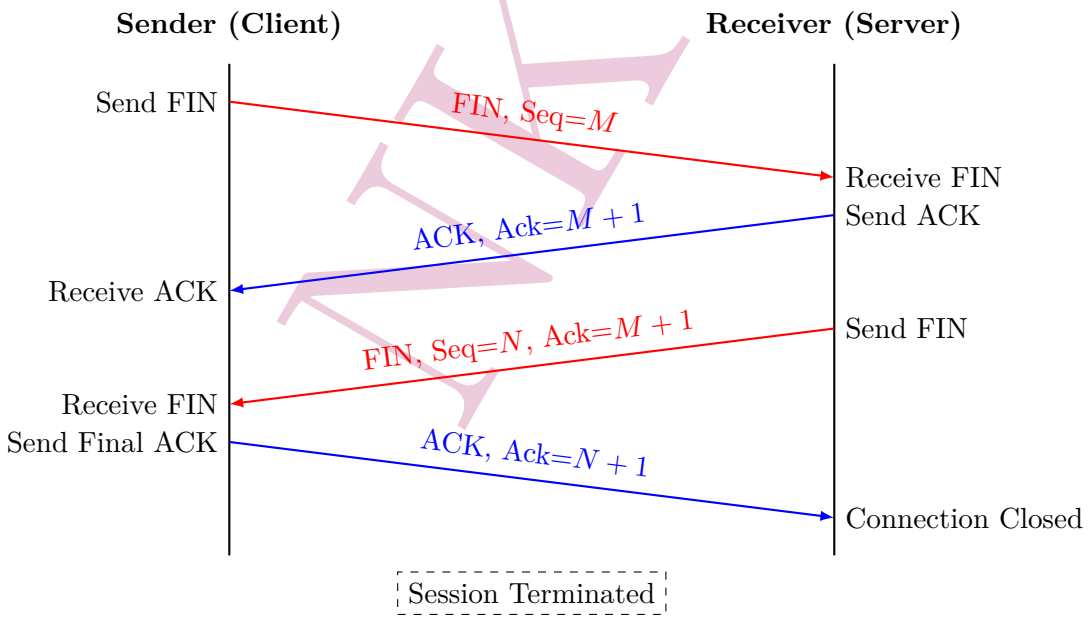

图 5: 基于 UDP 的四次挥手时序图

四、 实现方法

4.1 IP 及端口定义

在本实验的通信模型中，发送端与接收端均采用 IPv4 协议族进行寻址。为了便于程序的配置与移植，系统通过预处理宏定义指定了服务器（接收端）的目标 IP 地址与监听端口号。在代码实现层面，使用了 Windows Sockets API (Winsock2) 提供的 `sockaddr_in` 结构体来封装网络地址信息。

相关核心代码实现如下：

Listing 1: 网络地址结构配置代码

```
1
2  #define SERVER_IP "127.0.0.1"
3  #define SERVER_PORT 8080
4
5  sockaddr_in recvAddr;
6  recvAddr.sin_family = AF_INET;
7  recvAddr.sin_port = htons(SERVER_PORT);
8  recvAddr.sin_addr.s_addr = INADDR_ANY;
9
10 memset(&recvAddr, 0, sizeof(recvAddr));
11 recvAddr.sin_family = AF_INET;
12 recvAddr.sin_port = htons(SERVER_PORT);
13
14 inet_pton(AF_INET, SERVER_IP, &recvAddr.sin_addr);
```

使用 router 程序来模拟路由器的数据传输，设置如下，服务器以及路由器设置如下

图 6: router 设置

4.2 报文类实现

本实验封装了 Packet 类作为应用层协议数据单元的核心载体。为了保证网络传输的二进制兼容性，首先定义了 PacketHeader 结构体来描述报文头部。

在数据完整性保障方面,Packet 类内置了校验和计算逻辑。calculate_checksum 函数实现了标准的互联网校验和算法（16 位反码求和）。该算法首先将整个报文（包含头部和数据部分）视为一个 16 位整数序列，通过指针类型转换进行遍历求和。在遍历过程中，若报文总长度为奇数，算法会特判处理最后一个字节。所有加法运算结束后，通过循环移位将溢出的进位加回到低位，最后对结果取反码。通过这种方式，接收端只需对收到的报文再次执行同种算法，若结果为 0，则证明数据在传输中未发生比特错误。

报文结构定义与校验和计算的核心代码实现如下:

Listing 2: 报文头部定义与校验和算法实现

```
#pragma pack(1)
struct PacketHeader
{
    uint32_t seq;
    uint32_t ack;
    uint16_t flags;
    uint16_t checksum;
    uint16_t window;
    uint16_t length;
};
#pragma pack()

class Packet
{
public:
    PacketHeader head;
    char data[MAX_DATA_SIZE];
    // ... (其他成员函数声明)
private:
    uint16_t calculate_checksum();
};

uint16_t Packet::calculate_checksum() {
    uint32_t sum = 0;
    uint16_t* ptr = (uint16_t*)&head;
    for (size_t i = 0; i < sizeof(PacketHeader) / 2; ++i) {
        sum += *ptr++;
    }
```

```
29      ptr = (uint16_t*)data;
30      for (size_t i = 0; i < head.length / 2; ++i) {
31          sum += *ptr++;
32      }
33      if (head.length % 2) {
34          sum += data[head.length - 1];
35      }
36      while (sum >> 16) {
37          sum = (sum & 0xFFFF) + (sum >> 16);
38      }
39      return static_cast<uint16_t>(~sum);
40  }
```

4.3 三次握手实现

本实验在代码层面通过状态标志位与超时机制共同完成了三次握手逻辑。握手过程由发送端（Sender）的主动发起与接收端（Receiver）的被动响应交替进行。

发送端通过 `handshake` 函数执行连接建立流程。首先构造一个标志位为 `FLAG_SYN` 的报文,并将其序列号初始化为 0。为了应对 UDP 的不可靠性,发送端在一个死循环中发送 SYN 报文,并利用 `setsockopt` 将接收超时时间(`SO_RCVTIMEO`)设置为 2000ms。如果在规定时间内未收到接收端的响应，`recvfrom` 将返回错误，循环将触发重传逻辑；若收到的报文通过了校验和检查且标志位包含 `FLAG_SYN | FLAG_ACK`，则跳出循环，进入最后确认阶段。

接收端在主循环中检测到未连接状态下的 `FLAG_SYN` 报文后，将其标记为连接建立请求。为了遵循 TCP 规范，接收端认为 SYN 报文消耗一个序列号，因此将期望接收的下一个序列号 `expectedSeq` 设置为收到 SYN 报文序列号加 1。随后，接收端清空乱序缓冲区，并立即回复一个带有 `FLAG_SYN | FLAG_ACK` 的确认报文，确认号即为计算出的 `expectedSeq`。

发送端在收到 SYN+ACK 后，发送最后的 ACK 报文（标志位为 `FLAG_ACK`），并将自身的发送窗口基准 `base` 与下一个发送序号 `nextSeqNum` 均初始化为 1。这一步至关重要，它确保了随后传输的文件数据包序列号从 1 开始，完成了双方的状态同步。

核心交互代码实现如下：

Listing 3: 发送端与接收端握手核心逻辑

```
// Sender端：发送SYN并等待响应（超时重传机制）
sendPkt.head.flags = FLAG_SYN;
sendPkt.head.seq = 0;
sendPacket(sendPkt);

// 设置接收超时，防止死锁
int timeout = 2000;
setsockopt(senderSocket, SOL_SOCKET, SO_RCVTIMEO, (const char*)
    &timeout, sizeof(timeout));

while (true) {
    int ret = recvfrom(senderSocket, (char*)&recvPkt, sizeof(
        Packet), 0, (sockaddr*)&recvAddr, &addrLen);
    if (ret > 0) {
        if (recvPkt.check_checksum() && (recvPkt.head.flags & (
            FLAG_SYN | FLAG_ACK))) {
            break;
        }
    } else {
        // 超时未收到SYN+ACK，重传SYN
        sendPacket(sendPkt);
    }
}

// Receiver端：处理SYN并初始化状态
if (recvPkt.head.flags & FLAG_SYN) {
    isConnected = true;
    outOfOrderBuffer.clear();
    // SYN消耗一个序列号，期望序号+1
    expectedSeq = recvPkt.head.seq + 1;

    sendPkt.reset();
    sendPkt.head.ack = expectedSeq;
    sendPkt.head.flags = FLAG_SYN | FLAG_ACK;
    sendto(receiver, (char*)&sendPkt, sizeof(PacketHeader), 0,
        (sockaddr*)&senderAddr, senderAddrSize);
    continue;
}

// Sender端：完成握手，初始化序列号为1
sendPkt.head.flags = FLAG_ACK;
sendPkt.head.seq = 1;
sendPkt.head.ack = recvPkt.head.seq + 1;
base = 1;
nextSeqNum = 1;
```

4.4 模块初始化

在三次握手建立连接后，发送端进入文件传输的准备阶段。此时需要对拥塞控制算法的状态变量进行初始化，并调整底层 Socket 的行为模式以适应高频的数据交互。

首先，函数将 Reno 算法的状态机置为 `SLOW_START`（慢启动），并将拥塞窗口 `cwnd` 初始化为 1 个最大段大小（MAX_MSS），慢启动门限 `ssthresh` 设定为 64 倍的 MSS。这符合 TCP 协议的标准启动行为，即从低速率开始探测网络带宽。

其次，为了提高传输效率，代码对 Socket 的接收超时选项进行了关键调整。在握手阶段为了保证可靠性使用了较长的超时时间，而在传输阶段，为了避免发送逻辑被 `recvfrom` 阻塞，通过 `setsockopt` 将 `SO_RCVTIMEO` 设置为极短的 50ms。这使得发送端能够在等待 ACK 的间隙快速响应超时事件或发送新数据，实现了近似非阻塞的传输效果。

最后，在发送实际文件内容之前，发送端首先构造并发送一个包含文件名的特殊数据包。该报文序列号使用当前的 `nextSeqNum`，其负载部分携带了纯文件名字符串。该报文被立即推入发送缓冲区，随后计时器启动，这标志着可靠传输流程的正式开始。

相关初始化代码实现如下：

Listing 4: 拥塞控制参数初始化与文件名发送

```
// 初始化 Reno 状态变量
state = SLOW_START;
cwnd = MAX_MSS;
ssthresh = 64 * MAX_MSS;
dupAckCount = 0;

// 设置极短的接收超时，实现近似非阻塞模式
int nonBlockTimeout = 50;
setsockopt(senderSocket, SOL_SOCKET, SO_RCVTIMEO, (const char*)
    &nonBlockTimeout, sizeof(nonBlockTimeout));

// 构造并发送文件名报文
namePkt.reset();
namePkt.head.seq = nextSeqNum;
```

```
namePkt.head.flags = FLAG_DATA;
strcpy(namePkt.data, pureName.c_str());
namePkt.head.length = (uint16_t)pureName.length();

sendPacket(namePkt);
sendBuffer.push_back(namePkt);

// 启动计时器并更新序列号
if (base == nextSeqNum) startTimer();
nextSeqNum += namePkt.head.length;
```

4.5 数据传输实现

数据传输阶段采用了经典的流水线协议设计。发送端的核心逻辑位于主循环中，它严格受控于当前拥塞窗口（cwnd）的大小。只要下一个待发送的序列号 nextSeqNum 未超出 base + cwnd 的范围，且文件尚未读取完毕，发送端便会持续构造并发送新的数据包。

在发送过程中，程序利用文件流的 seekg 函数精确定位读取位置。读取的数据被封装进 Packet 对象后，首先通过 UDP 套接字发出，随即被推入 sendBuffer 队列。这个队列起到了“滑动窗口”的作用，保存了所有已发送但未被确认的副本，以便在发生丢包时能够从内存中快速提取并重传，而无需再次访问磁盘 IO。若当前发送的是窗口内的第一个报文（即 base == nextSeqNum），系统会同步启动重传计时器。

接收端的实现重点在于处理乱序到达的数据报文。当收到带有 FLAG_DATA 的报文时，系统首先校验其序列号。若序列号等于期望的 expectedSeq，说明数据按序到达，程序将其直接写入目标文件，并立即检查 outOfOrderBuffer（一个基于红黑树实现的 map 容器）中是否存在后续连续的报文。若存在，则循环取出并写入文件，同时更新 expectedSeq，从而实现乱序报文的重组。若收到的序列号大于期望值，则将其暂存入缓冲区。无论何种情况，接收端最后都会发送一个携带当前 expectedSeq 的 ACK 报文，实现累积确认。

发送端的数据发送循环与接收端的乱序处理逻辑如下所示：

Listing 5: 发送端窗口控制与接收端乱序缓冲逻辑

```
// Sender: 窗口允许范围内持续发送数据
while (nextSeqNum < base + (int) cwnd && nextSeqNum < fileSize)
    {
    Packet pkt;
    pkt.reset();
    pkt.head.seq = nextSeqNum;
    pkt.head.flags = FLAG_DATA;

    file.seekg(nextSeqNum - nameLen - 1);
    int bytesToRead = min((long long)MAX_MSS, fileSize -
       nextSeqNum);
    file.read(pkt.data, bytesToRead);
    pkt.head.length = bytesToRead;

    sendPacket(pkt);
    sendBuffer.push_back(pkt);
    if (base == nextSeqNum) {
        startTimer();
    }
    nextSeqNum += bytesToRead;
}

// Receiver: 处理按序到达与乱序缓冲
if (recvPkt.head.seq == expectedSeq) {
    outFile.write(recvPkt.data, recvPkt.head.length);
    totalBytesReceived += recvPkt.head.length;
    expectedSeq += recvPkt.head.length;

    // 检查缓冲区是否有后续数据
    while (outOfOrderBuffer.count(expectedSeq)) {
        Packet bufferedPkt = outOfOrderBuffer[expectedSeq];
        outFile.write(bufferedPkt.data, bufferedPkt.head.length
           );
        totalBytesReceived += bufferedPkt.head.length;
        expectedSeq += bufferedPkt.head.length;
        outOfOrderBuffer.erase(bufferedPkt.head.seq);
    }
}
else if (recvPkt.head.seq < expectedSeq) {
    // 收到重复包, 忽略或仅记录
}
else {
    // 收到乱序包, 存入缓冲区
    outOfOrderBuffer[recvPkt.head.seq] = recvPkt;
}

// 发送累积确认 ACK
sendPkt.reset();
sendPkt.head.seq = 0;
sendPkt.head.ack = expectedSeq;
sendPkt.head.flags = FLAG_ACK;
```

```
49 sendPacket(sendPkt);
```

4.6 Reno 算法

本实验将 TCP Reno 拥塞控制算法的核心逻辑嵌入在发送端的主循环中，主要由 ACK 接收事件和超时事件驱动状态机的迁移。代码中使用枚举类型 `RenoState` 定义了 `SLOW_START`、`CONGESTION_AVOIDANCE` 和 `FAST_RECOVERY` 三种状态，并通过浮点型变量 `cwnd` 维护拥塞窗口大小。

当发送端收到一个新的确认报文（即 `ack > base`）时，意味着数据被成功接收，此时算法根据当前状态更新窗口。若处于慢启动状态，`cwnd` 增加 1 个 MSS，呈现指数增长趋势；当 `cwnd` 超过 `ssthresh` 时，自动切换至拥塞避免状态。若已处于拥塞避免状态，代码通过公式 `cwnd += MAX_MSS * (MAX_MSS / cwnd)` 实现窗口的线性增长（每个 RTT 增加约 1 MSS）。此外，收到新 ACK 还标志着快恢复阶段的结束，系统将状态重置为拥塞避免，并将 `cwnd` 恢复为 `ssthresh`。

对于重复 ACK 的处理，系统维护了一个计数器 `dupAckCount`。当收到 `ack <= base` 的报文时，计数器递增。一旦计数器达到 3，即触发快速重传机制：系统立即将 `ssthresh` 减半，将 `cwnd` 设置为 `ssthresh + 3 * MSS`，并切换至快恢复状态。最为关键的是，发送端无需等待计时器超时，而是直接访问 `sendBuffer` 的头部（即 `base` 对应的报文），立即重传丢失的数据包，从而显著降低了丢包带来的延迟。

作为最后的保障,代码在主循环末尾检查应用层计时器。若 `isTimerExpired()` 返回真，说明网络发生严重拥塞。此时执行超时重传逻辑：将 `ssthresh` 骤降为当前窗口的一半，`cwnd` 重置为 1 MSS，状态强制回退至慢启动，并重传缓冲区头部的报文，以此大幅降低发送速率来缓解网络压力。

核心算法实现代码如下：

Listing 6: Reno 拥塞控制与重传逻辑

```
1 // 处理新 ACK
2 if (ack > base) {
```

```
    base = ack;
    if (state == SLOW_START) {
        cwnd += MAX_MSS;
        if (cwnd >= ssthresh) {
            state = CONGESTION_AVOIDANCE;
        }
    }
    else if (state == CONGESTION_AVOIDANCE) {
        cwnd += MAX_MSS * ((double)MAX_MSS / cwnd);
    }
    else if (state == FAST_RECOVERY) {
        state = CONGESTION_AVOIDANCE;
        cwnd = ssthresh;
    }
    dupAckCount = 0;
    stopTimer();
}
// 处理重复 ACK
else {
    dupAckCount++;
    if (state == FAST_RECOVERY) {
        cwnd += MAX_MSS;
    }
    else if (dupAckCount == 3) {
        ssthresh = max((double)10 * MAX_MSS, cwnd / 2);
        cwnd = ssthresh + 3 * MAX_MSS;
        state = FAST_RECOVERY;

        if (!sendBuffer.empty()) {
            sendPacket(sendBuffer[0]);
        }
    }
}

// 超时处理逻辑
if (isTimerExpired()) {
    ssthresh = max((double)2 * MAX_MSS, cwnd / 2);
    cwnd = MAX_MSS;
    state = SLOW_START;
    dupAckCount = 0;
    stopTimer();
    startTimer();
    if (!sendBuffer.empty()) {
        sendPacket(sendBuffer[0]);
    }
}
```

4.7 四次挥手实现

当文件传输完成且所有数据包均被确认后，发送端调用 teardown 函数发起断开连接流程。代码实现采用了标准的 TCP 四次挥手逻辑，并针对 UDP 的特性增加了超时重传保障。

发送端首先构造一个标志位为 FLAG_FIN 的报文发出，随后进入一个循环等待状态。在等待接收端的 ACK 确认时，代码通过设置 SO_RCVTIMEO 启用了接收超时机制。若在规定时间内未收到 ACK（即 recvfrom 返回值小于等于 0），发送端会不断重传 FIN 报文，直到收到确认为止。这一设计有效防止了因 FIN 报文丢失而导致的接收端一直处于挂起状态。

接收端在主循环中检测到 FLAG_FIN 后，立即进入关闭流程。它首先回复一个 ACK 报文确认发送端的关闭请求，随即发送自己的 FIN 报文请求关闭反向连接。之后，接收端进入循环等待发送端的最后一次确认。

发送端在收到接收端的 FIN 报文后，回复最后一个 ACK 报文（标志位 FLAG_ACK），并打印连接关闭日志。至此，双方确认彼此均已停止发送并释放了连接资源，程序安全退出。

四次挥手的核心交互代码如下：

Listing 7: 发送端主动关闭与接收端响应逻辑

```
// Sender: 发送 FIN 并循环等待 ACK (超时重传)
sendPkt.head.flags = FLAG_FIN;
sendPkt.head.seq = nextSeqNum;
sendPacket(sendPkt);

while(true) {
    int ret = recvfrom(senderSocket, (char*)&recvPkt, sizeof(
        Packet), 0, (sockaddr*)&recvAddr, &addrLen);
    if (ret > 0 && (recvPkt.head.flags & FLAG_ACK)) {
        break;
    }
    // 超时重传 FIN
    if (ret <= 0) sendPacket(sendPkt);
}

// Sender: 等待 Server 的 FIN
while(true) {
    int ret = recvfrom(senderSocket, (char*)&recvPkt, sizeof(
```

```
        Packet), 0, (sockaddr*)&recvAddr, &addrLen);
    if (ret > 0 && (recvPkt.head.flags & FLAG_FIN)) {
        break;
    }
}

// Sender: 发送最后的 ACK
sendPkt.reset();
sendPkt.head.flags = FLAG_ACK;
sendPkt.head.ack = recvPkt.head.seq;
sendPacket(sendPkt);

// Receiver: 处理 FIN 报文
if (recvPkt.head.flags & FLAG_FIN) {
    // 发送 ACK
    sendPkt.head.flags = FLAG_ACK;
    sendto(receiver, (char*)&sendPkt, sizeof(PacketHeader), 0,
        (sockaddr*)&senderAddr, senderAddrSize);

    // 发送 FIN
    sendPkt.head.flags = FLAG_FIN;
    sendto(receiver, (char*)&sendPkt, sizeof(PacketHeader), 0,
        (sockaddr*)&senderAddr, senderAddrSize);

    // 等待最后的 ACK
    while (true) {
        int ret = recvfrom(receiver, (char*)&recvPkt, sizeof(
            Packet), 0, (sockaddr*)&senderAddr, &senderAddrSize)
            ;
        if (ret > 0 && (recvPkt.head.flags & FLAG_ACK)) {
            break;
        }
        else {
            sendto(receiver, (char*)&sendPkt, sizeof(
                PacketHeader), 0, (sockaddr*)&senderAddr,
                senderAddrSize);
        }
    }
}
```

五、 性能测试

5.1 有拥塞控制和无拥塞控制的性能比较

在本部分中，测试组共分为：

- 无拥塞控制：固定传送窗口大小。

- 有拥塞控制：Reno 拥塞控制算法。

窗口大小设为 30。

5.1.1 不同丢包率下的性能对比

延时为 0ms 下，测试不同丢包率下的性能，如表1所示：

丢包率		0%	2%	4%	6%	8%	10%
有拥塞控制	传输时间 (s)	0.708	0.745	0.817	0.889	2.930	9.034
	吞吐量 (KB/s)	2623	2493	2273	2089	633.9	205.6
无拥塞控制	传输时间 (s)	0.641	0.729	0.752	0.856	0.948	1.265
	吞吐量 (KB/s)	2898	2547	2470	2170	1959	1468

表 1: 不同丢包率下的性能对比

将对比结果绘制成图片，如图7所示。

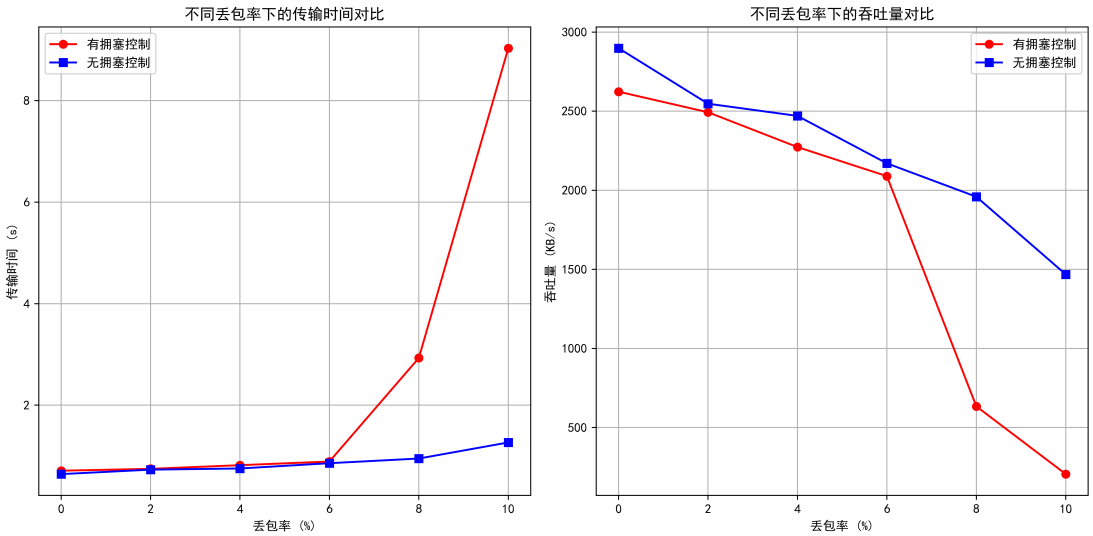

图 7: 不同丢包率下的性能对比

分析：随丢包率增大，各方法均出现了传输时间增加，吞吐量下降的情况。然而，在丢包率为 6% 以下，有拥塞控制的方法与无拥塞控制的方法差距并不显著。但当丢包率提升后，由于频繁发生的丢包情况，拥塞窗口大小一直位于较低

水平，若在窗口大小小于 3 的时候发生了丢包，则会造成超时，返回慢启动阶段，因此产生了比较大的开销。

总体来说，由于无拥塞控制的性能已经很高了，有拥塞控制不但没有提升性能，反而产生了性能的降低。

5.1.2 不同延时下的性能对比

丢包率为 0% 下，测试不同延时下的性能，如表2所示：

传输延时 (ms)		0	20	40	60	80	100
有拥塞控制	传输时间 (s)	0.708	10.407	17.931	19.955	22.971	25.996
	吞吐量 (KB/s)	2623	178.4	103.5	93.08	80.86	71.45
无拥塞控制	传输时间 (s)	0.641	18.304	21.234	25.545	31.368	34.461
	吞吐量 (KB/s)	2898	101.5	87.47	72.70	59.21	53.89

表 2: 不同延时下的性能对比

将对比结果绘制成图片，如图8所示。

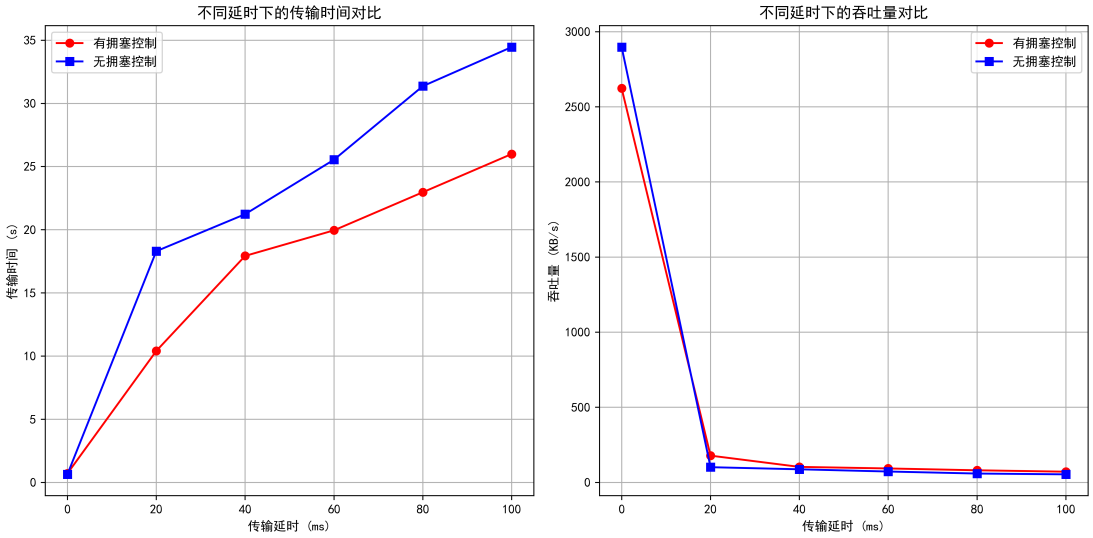

图 8: 不同延时下的性能对比

分析：随传输延时的增加，有无拥塞控制的传输时间都发生了较为显著的增加。然而，相比于有拥塞控制的测试结果，无拥塞控制的测试组的优化随传输延

时的增加而越发显著。这是因为，在发生拥塞时（重复 ACK 或超时），Reno 算法会变换状态，改变窗口大小，从而减少不必要的包的发送，进而成功起到提升性能的效果。

六、 总结

6.1 功能实现总结

本实验基于 UDP 协议设计并实现了一套完整的应用层可靠文件传输系统。通过自定义报文格式与状态机逻辑，系统成功克服了 UDP 不可靠、无连接的特性，达到了以下主要目标：

- **连接管理**：实现了基于 SYN/FIN 标志位的严格三次握手建立连接与四次挥手断开连接流程，确保了通信双方状态的同步。
- **可靠传输**：利用序列号与确认号机制实现了数据的有序交付，通过接收端的乱序缓冲处理解决了网络乱序问题，并结合超时重传与累积确认保证了数据的完整性。
- **拥塞控制**：完整复现了 TCP Reno 算法的核心逻辑。实现了慢启动阶段的指数增长、拥塞避免阶段的线性增长，以及基于 3 次重复 ACK 的快速重传与快速恢复机制。实验结果表明，该算法能够根据网络状况动态调整发送速率，有效缓解网络拥塞。

6.2 局限性分析与改进方向

尽管本系统实现了 TCP 协议的大部分核心特性，但对比标准 TCP 协议规范，在**窗口管理与流量控制**方面仍存在一定的简化与不足，具体体现在以下两点：

1. **缺乏接收端窗口通告（Flow Control）**：标准 TCP 协议中，发送端的实际发送窗口大小（Send Window）应由拥塞窗口（cwnd）和接收端通告窗口

（rwnd）的最小值决定，即 $Window = \min(cwnd, rwnd)$。本实验中，发送端的发送速率仅完全受控于拥塞窗口 `cwnd`，即只考虑了网络的拥塞状况，而忽略了接收端的处理能力与缓冲区剩余空间。

2. **接收端窗口反馈机制缺失**：在接收端的实现中，虽然定义了报文头部中的 `window` 字段，但并未并在 ACK 报文中实时计算并填充本地缓冲区的剩余大小。接收端仅是被动地接收数据并确认，未向发送端提供“反压（Backpressure）”信号。若接收端的磁盘写入速度远低于网络传输速度，可能会导致接收端应用层缓冲区溢出，而发送端对此一无所知仍持续高速发送。

综上所述，本实验成功构建了一个具备拥塞控制能力的可靠传输模型，验证了 Reno 算法的有效性。在未来的改进中，应引入接收端滑动窗口（rwnd）的动态反馈机制，实现完整的流量控制，使系统在网络拥塞和接收端处理能力受限时均能保持鲁棒性。