实验 2：可靠数据传输协议设计与实现
计算机网络第二次实验报告

姓名：郭佳成　学号：2311990　专业：密码科学与技术

代码：https://github.com/Fighting05/ComputerNetwork

2025 年 12 月 20 日

1 实验要求

1. 设计并实现一个基于 UDP 的可靠数据传输协议。
2. 实现连接管理：包括三次握手建立连接、四次挥手断开连接。
3. 实现差错检测：使用校验和（Checksum）检测数据包损坏。
4. 实现可靠传输：支持滑动窗口机制（Select Repeat），支持确认重传、超时重传。
5. 实现流量控制：发送窗口和接收窗口使用固定大小窗口。
6. 实现拥塞控制：采用 Reno 拥塞控制算法。
7. 编写的程序应结构清晰，具有较好的可读性。
8. 在实验中观察文件传输过程，验证传输的完整性与可靠性。

2 实验环境

- **操作系统**：Windows 11
- **开发语言**：C++11
- **编译器**：MinGW-w64 GCC
- **网络库**：Winsock2
- **测试环境**：本地回环测试 (Localhost, 127.0.0.1)

3 协议设计说明

3.1 报文结构设计

为了在 UDP 之上实现可靠传输，我们自定义了 Packet 结构体。在实验时，遇到了一个很奇怪的报错 (主要是一直出现校验和错误，一直 debug 不出来后来才发现)，在 AI 帮助下注意到了内存对齐可能带来的网络传输问题。

默认情况下，编译器为了提高 CPU 访问效率，会在结构体成员之间插入填充字节（Padding）。但在网络传输中，我们需要发送的是紧凑的字节流。如果直接发送带有填充字节的结构体，接收端解析时可能会出现错位。

因此，我使用了 #pragma pack(1) 指令，强制关闭字节对齐，将结构体按照 1 字节紧凑排列。这虽然牺牲了微小的 CPU 访问效率，但确保了 Packet 在不同机器、不同编译器下的内存布局完全一致。

```
#pragma pack(1)  // 关闭字节对齐，确保结构体大小固定
struct Packet
{
    unsigned int seq;         // 发送序号
    unsigned int ack;         // 确认号
    unsigned short checksum; // 校验和 (16位反码求和)
    unsigned short len;       // 数据载荷长度
    unsigned char flags;      // 标志位 (SYN, ACK, FIN, DATA)
    char data[MAX_DATA_SIZE]; // 数据载荷
};
#pragma pack()   // 恢复默认对齐
```

3.2 连接管理机制

- **建立连接 (三次握手)：**
  1. Sender 发送 SYN (seq=x)。
  2. Receiver 回复 SYN+ACK (seq=y, ack=x+1)。
  3. Sender 回复 ACK (seq=x+1, ack=y+1)。
- **断开连接 (四次挥手)：**
  1. Sender 发送数据完毕后，发送 FIN 包。
  2. Receiver 收到 FIN 后回复 ACK，并进入关闭等待状态。
  3. 不过我在代码中简化为 Sender 发送 FIN，Receiver 回复 ACK 并退出，Sender 收到 ACK 后退出。

3.3 可靠传输机制 (Sliding Window)

本实验实现了 **滑动窗口 (Sliding Window)** 机制。

Sender 端逻辑：

1. 维护发送窗口 $base$ 到 $next\_seq$。
2. 只要窗口未满，连续发送数据包，并记录发送时间。
3. 收到 ACK 后，标记对应包为已确认。若 $base$ 被确认，$base++$ 即窗口右移, 连续检测到无法移动为止。
4. 轮询窗口内未确认的包，若超时就给他重传。

Receiver 端逻辑：

1. 维护接收窗口与缓冲区 $recv\_buffer$。
2. 收到乱序包（Seq > Expected）：存入缓冲区，回复 ACK，这里就按照要求采用选择确认。
3. 收到期望包（Seq == Expected）：写入文件，并检查缓冲区是否有后续包，一并写入。
4. 收到旧包（Seq < Expected）：回复 ACK 以防 Sender 重传，这个也是答辩时助教问到的最后的一个问题就是当收到了一个旧包时，你是如何处理的。
5. 整合步骤，如果一段内的乱序刚好接上了就全部按序写入文件。

图 1: 可靠传输状态机

3.4 流量控制

针对实验要求中“发送窗口和接收窗口使用固定大小”的规定, 我设计的协议采用了以下的改动：

- **固定接收上限 (RWND)**：我们在 Sender 端定义了常量 `RWND`（如 32），代表接收方缓冲区的最大容量。这一数值在整个传输过程中保持不变，模拟了“固定大小接收窗口”。
- **动态发送窗口 (SWND)**：实际的发送窗口大小由流量控制和拥塞控制共同决定：

$$\text{Send Window} = \min(\text{cwnd}, \text{RWND})$$

其中 `cwnd` 由 Reno 算法动态调整，`RWND` 为固定常量。

这种设计主要是按照实验要求中说要采取固定的窗口大小，但是又必须使用 *RENO* 算法而导致的窗口动态变化，从而采取的这样的一个方案。

3.5 Reno 算法

由于实验要求中需要我们实现选择确认，而我在后面增加 Reno 算法时意识到他是基于累计确认的，就让人感觉很矛盾，为了满足实验要求，于是为了让两边得到适配所以我做出了以下的改动：

1. **慢启动**：初始 `cwnd=1`，每收到一个新 ACK，`cwnd += 1`，窗口呈指数增长。
2. **拥塞避免**：当 `cwnd >= ssthresh` 时，每收到一个新 ACK，`cwnd += 1/cwnd`，窗口呈线性增长。
3. **快重传 (主要改动地方)**：
   - 由于是选择确认，那么接收端要么收到包发送 ack, 要么没收到包就不发送了，而且这一些 ack 都是针对具体的某一个包而言，所以我们在发送端统计“乱序 ACK”的数量。
   - 当收到 3 个 `ack > base` 的 ACK 时，视为收到 3 个重复 ACK 这个主要是可以表明 base 后面的三个包都到了而 base 迟迟未到就类似与 reno 中累计确认的三次重复，于是判定 `base` 丢包。
   - 立即重传 `base`，并执行 `ssthresh = cwnd/2, cwnd = ssthresh + 3`。

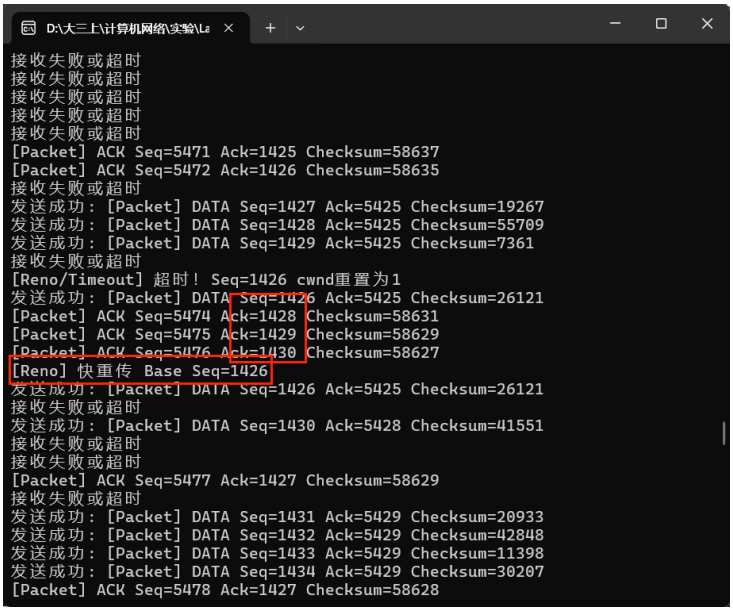

图 2: 三次重传

上面给出了一个我们在实验运行中真实发生的一个三次重传，可以看到这里收到了 Ack=1428,Ack=1429,Ack=1430, 然后 base 是 1426，于是就被检测到了三次重传。你在下方就可以看到一个 Ack=1427, 也就是这次的三次重传成功了。

4. **快恢复：**

   - 进入快恢复状态后，每收到一个重复/乱序 ACK，`cwnd += 1`（窗口恢复），保持数据流发送。
   - 一旦收到确认了 `base` 的新 ACK，退出快恢复，将 `cwnd` 重置为 `ssthresh`。

这个在后面我们专门让 AI 写了一个绘图的代码，把运行的真实情况给画出来了看看后面即可。我们在这里也放上几个, 这四个都是设置了 3% 丢包率，5ms 延时。

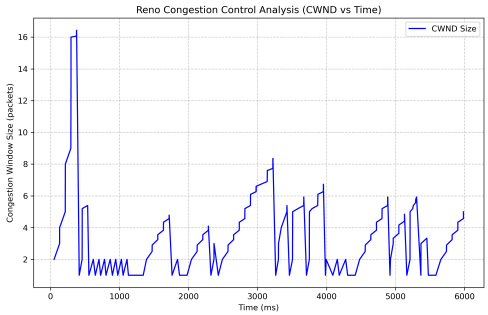

图 3: 测试文件 1 传输情况

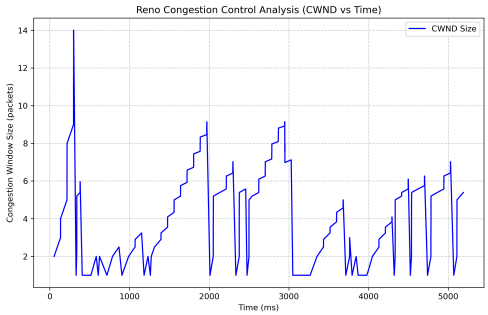

图 4: 测试文件 2 传输情况

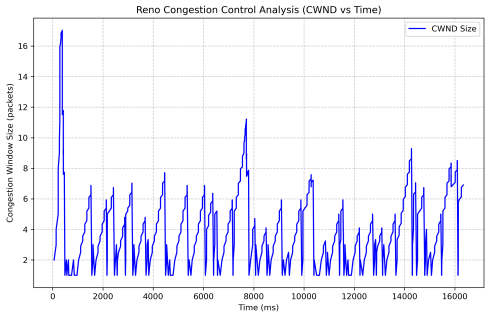

图 5: 测试文件 3 传输情况

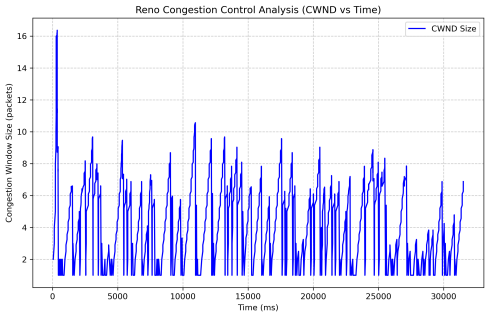

图 6: 测试文件 4 传输情况

图 7: 四组测试文件在丢包率 3%、延时 5ms 环境下的传输结果

拥塞窗口曲线分析：上面展示了四个不同大小的文件在弱网传输过程中，拥塞窗口 (CWND) 随时间变化的实时曲线。通过观察这四张图，我们可以清晰地识别出 TCP Reno 拥塞控制算法在对抗丢包时的典型行为特征：

- **典型的锯齿波形态 (Sawtooth Pattern)**：在图 2、图 3 和图 4（大文件）中，CWND 曲线呈现出显著的“线性增长 → 瞬间减半 → 线性增长”的锯齿状循环。这是因为在 3% 丢包率下，随着窗口增大，丢包几乎必然发生。一旦检测到丢包，Reno 算法立即执行“快重传/快恢复”，将 ssthresh 和 cwnd 减半，从而在图像上形成了一个个尖锐的“下坠”点。
- **平衡与收敛**：尽管网络环境恶劣，但曲线并未发散或触底，而是围绕着某个特定值上下震荡。这表明算法成功找到了当前网络链路的“有效带宽上限”，并在该上限附近维持动态平衡，既不造成严重拥塞，也不浪费带宽。
- **传输阶段的差异性**：对比图 1（小文件）与图 4（大文件）：
  - **图 1**：由于文件较小，传输在“慢启动”或刚进入“拥塞避免”阶段初期即结束，曲线主要表现为上升趋势，尚未形成稳定的震荡周期。
  - **图 4**：随着传输时间拉长，曲线展示了完整的、连续的拥塞控制周期。这种长周期的稳定震荡有力地证明了本协议在长时间高负载传输下的稳定性。

3.6 流量控制

4 程序设计与实现

4.1 Sender 端实现 (sender.cpp)

4.1.1 ACK 接收循环控制

那么在本次实验我没用使用多线程，因为一开始写的时候没用后面发现速度还行(而且实验要求没说用我也忘了)，于是我就选择让 send 使用一个循环不过有次数限制来接受，主要是窗口也就那么大所以确实是可以的。

```
int ack_limit = 16; // 每次循环最多处理 16 个 ACK
while (recv_packet_show(sendSocket, &ackPkt, &recvAddr, &
   clientAddrLen))
{
    if(ack_limit-- <= 0) break; // 强制退出，转去发送数据

    // 处理 ACK 逻辑...
}
```

4.1.2 Reno 拥塞控制实现

我们使用全局变量 dupAckCount 来统计乱序 ACK，模拟快重传触发条件就是我们前面所说的。

```
if(ackPkt.ack == sendWindow[i].pkt.seq + 1 && !sendWindow[i].acked)
{
    sendWindow[i].acked = true;
    // 拥塞控制：慢启动或拥塞避免
    if(cwnd < ssthresh)
    {
        cwnd += 1.0;
    }
    else
    {
        cwnd += 1.0/cwnd;
    }

    // 检查是否触发快重传（确认了 base 之后的包）
    if(i > base && !sendWindow[base].acked)
    {
        dupAckCount++;
        if (dupAckCount == 3)
        {
            cout << "[Reno] 快重传 Base Seq=" << sendWindow[base].pkt
                .seq << endl;
            // 立即重传
            send_packet(sendSocket, sendWindow[base].pkt, recvAddr);
            sendWindow[base].sentTime = clock();

            // 执行乘法减小
            ssthresh = max(2.0, cwnd / 2.0);
            cwnd = ssthresh + 3;
            dupAckCount = 0; // 重置计数
        }
        else if(dupAckCount > 3)
        {
            cwnd += 1.0; // 快恢复
        }
    }
}
```

4.1.3 超时重传

我们在包发送的时候使用了 clock() 记录发送时间，然后在主循环中轮询未确认包的发送时间，若超过 100ms 则判定超时，进行重传。这个地方的时间设置也非常重要，不能太长也不能太短导致重传的太多了或者太慢了。

```
if(!sendWindow[base].acked)
{
    if(clock()-sendWindow[base].sentTime>=100) // 超时阈值设为 100ms
    {
        cout << "[Reno/Timeout] 超时！ Seq=" << sendWindow[base].pkt.
            seq << endl;

        ssthresh = max(2.0, cwnd / 2.0);
        cwnd = 1.0;
        dupAckCount = 0;

        // 重传 base
        send_packet(sendSocket, sendWindow[base].pkt, recvAddr);
        sendWindow[base].sentTime=clock();
    }
}
```

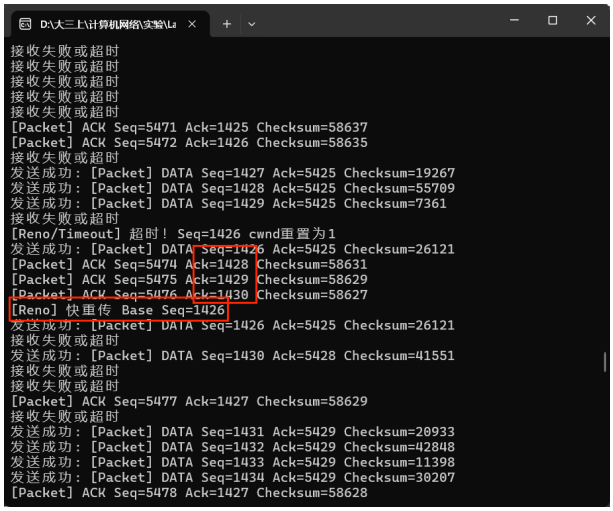

图 8: 超时

我们借用一下前面那个三次重传的，也可以在这个图片中找到一个超时重传 $Seq = 1426$,cwnd 重置为 1 了。在之前的那些波形图中也可以体现。

4.2 Receiver 端实现 (receiver.cpp)

4.2.1 乱序接收与重组

```
if (recvPkt.seq == expect_seq) {
    // 收到期望包，直接写入文件
    file.write(recvPkt.data, recvPkt.len);
    expect_seq++;

    // 检查缓冲区，处理连续包
    while(recv_buffPacket.count(expect_seq)) {
        file.write(recv_buffPacket[expect_seq].data, ...);
        recv_buffPacket.erase(expect_seq); // 清理已处理的包
        expect_seq++;
    }
    // 回复 ACK（期望下一个）
    make_ack_packet(&ackPkt, expect_seq);
    send_packet(...);
}
else if (recvPkt.seq > expect_seq) {
    // 收到乱序包，存入 map 缓冲区
    recv_buffPacket[recvPkt.seq] = recvPkt;
    make_ack_packet(&ackPkt, recvPkt.seq + 1); // 仍回复 ACK
    send_packet(...);
}
```

这个地方实现并不难，毕竟这一次实验只是单向传输文件，这边就比较好接受了。只要顺序接受就写入，乱序就储存，每次检查一下乱序的可不可以刚好接上前面的顺序然后拿出来写入就好了。

4.2.2 超时退出

为了防止 Sender 异常退出导致 Receiver 永久阻塞，我们在 Receiver 端添加了超时检测机制，如果过长未收到 send 的消息那么就退出：

```
else {
    // recv_packet_show 返回 false（超时）
    if(clock() - last_recv_time > 10000) { // 超过 10 秒无数据
        cout << "[Error] 长期未收到数据，判定连接断开，自动退出。" <<
            endl;
        break;
    }
```

```
}
```

4.3 校验和算法

为了保证数据在传输过程中不被篡改或损坏，我实现了标准的校验和算法。该算法的核心是将数据看作 16 位整数的序列，进行二进制反码求和。

在实现过程中，我使用了指针操作来高效地遍历数据缓冲区，并处理了缓冲区长度为奇数时的边界情况（补 0）。

```
unsigned short calculate_checksum(const void* buf, int len) {
    const unsigned short* ptr = (const unsigned short*)buf;
    unsigned int sum = 0;

    // 16位为单位累加
    while (len > 1) {
        sum += *ptr++;
        len -= 2;
    }
    // 处理奇数长度的最后一个字节
    if (len > 0) {
        sum += *(const unsigned char*)ptr;
    }
    // 将进位加回低位 (折叠高16位)
    while (sum >> 16) {
        sum = (sum & 0xFFFF) + (sum >> 16);
    }
    // 取反返回
    return (unsigned short)~sum;
}
```

4.4 接收端乱序重组

在处理乱序到达的数据包时，传统的实现方式通常是开辟一个巨大的固定数组作为缓冲区。但这会带来两个问题：一是内存空间浪费，二是需要复杂的逻辑来标记哪些位置是“空洞”。

为了优化这一过程，我利用了 C++ STL 中的 map 容器。

- **自动排序**：map 基于红黑树实现，能够根据 key（即 seq 序号）自动对插入的数据包进行排序。这使得我们无需手动维护乱序队列的顺序。

- **动态管理**：只有当收到乱序包时才分配内存，极大节省了空间。
- **高效去重**：如果收到重复的 seq，map 的特性保证了不会重复插入，天然解决了重复包问题。

代码实现极其简洁：

```
// 收到乱序包，直接存入 map，自动排序
recv_buffPacket[recvPkt.seq] = recvPkt;

// 检查是否填补了 seq 空洞
while(recv_buffPacket.count(expect_seq)) {
    // 将 map 中的数据写入文件
    file.write(recv_buffPacket[expect_seq].data, ...);
    // 从缓冲区移除
    recv_buffPacket.erase(expect_seq);
    expect_seq++;
}
```

4.5 连接管理与 Socket 优化

除了核心的数据传输逻辑，连接建立的健壮性还是很重要的。我在代码中完整实现了基于状态交互的三次握手，并且是特殊的处理，直接对三次握手进行停等的协议，发一个等一个停一下，确保我们正确的三次握手，而且就三条完全可以使用停等，最后特别处理了握手过程中的丢包问题。

```
// 第一次握手：发送 SYN
Packet syn;
make_syn_packet(&syn, ++SEND_ISN);
send_packet(sendSocket, syn, recvAddr);

// 等待第二次握手：接收 SYN_ACK
// 循环检测，如果超时则重传 SYN
while (!recv_packet_show(sendSocket, &syn_ack, &recvAddr, &
   clientAddrLen)) {
    cout << "握手超时，重传␣SYN..." << endl;
    send_packet(sendSocket, syn, recvAddr);
}

// 第三次握手：发送 ACK
Packet ack;
```

```
make_ack_packet(&ack, syn_ack.seq + 1);
send_packet(sendSocket, ack, recvAddr);
```

此外，为了防止程序在网络异常时永久阻塞在 `recvfrom` 函数上，我利用 Winsock API 对 Socket 进行了属性配置，设置了接收超时时间。

```
// 设置接收超时为 3000 毫秒
DWORD timeout = 3000;
setsockopt(sendSocket, SOL_SOCKET, SO_RCVTIMEO, (const char*)&timeout
    , sizeof(timeout));
```

这一设置确保了 Sender 和 Receiver 具有知道了链接断开的能力，能够在长时间未收到数据包时自动退出或触发重传逻辑。

5 遇到的问题与现象

问题 1：CheckSum

就是在实验过程中发现从来没有出现过 CheckSum 错误的情况，也有去检测相关的代码确实没问题，估计是由于在本机上发送的缘由，出错的概率比较小吧。

问题 2：链接错误 (Undefined Reference)

编译时报错 $undefinedreferencetomake\_data\_packet$。

在 *protocol.h* 中修改了函数声明（增加了 `ack` 参数），但 `protocol.cpp` 中的实现参数类型使用了 `const char*`，而头文件声明的是 `char*`。C++ 将其视为两个不同的重载函数，导致链接失败。

最后通过统一头文件和源文件中的参数类型，均加上 `const` 修饰符。

问题 3：滑动窗口死锁 (Seq 不匹配)

Sender 一直重传第一个数据包，窗口无法滑动。

Sender 期望收到的 ACK 是 `seq + 1`，而 Receiver 最初的实现只回复了 `seq`。这导致 Sender 认为所有 ACK 都是无效的。

遵循 TCP 规范，Receiver 统一回复 `pkt.seq + 1`，Sender 统一检查 `ack == pkt.seq + 1`，从而打通了确认逻辑。

问题 4：SR 与 Reno 的逻辑冲突

这个我们在前面设计的时候说了，就是感觉两个机制有所冲突，不过最后选择了一个折中的方案来实现。

问题 5：高丢包率下窗口连续减半

实验中观察日志发现，在 1010 包丢失后，后续收到的 1012, 1013... 的 ACK 都会触发“乱序”判定。初期代码对每一个乱序 ACK 都执行了一次 `ssthresh` 减半，导致 `cwnd` 瞬间降为 1，传输极慢。主要是之前写的判断是大于等于三于是每来一个就减半导致特别慢。

解决：完善状态判断逻辑。仅当 `dupAckCount == 3` 时执行一次减半操作并重置计数器（或进入快恢复状态）。后续收到更多乱序 ACK 时，不再减半，而是执行 `cwnd++`（快恢复充气），保持了窗口大小。

问题 6：同步接收导致发送停顿

初始代码使用 300ms 超时的阻塞 `recvfrom`。主要是一个调参的问题，每次发送完一波数据后，必须等待 300ms 确认没有更多 ACK 才能进入下一轮发送，严重浪费了带宽。

在接收循环前将 Socket 超时设置为 1ms。这样 Sender 能迅速读完缓冲区内的所有 ACK 并立即返回主循环继续发送数据，吞吐量提升了数倍。

问题 7：ROUTE 软件的问题

这个其实是困扰了我比较久的问题，就是使用了他之后尽管我设置的丢包和延时不高，但是速度很慢，jpg 都需要好几分钟才发的完，速度太慢了，一直觉得是我的代码问题，让 AI 也没改出来。

最后让 AI 写了一个测试 Router 性能的代码，发现 Router 的实际延迟和丢包率远高于设定的，我就说咋这么慢，还是推荐使用，这个锅需要助教背了。

6 实验环境

本实验过程中，在完成基本协议功能后，我们在性能测试阶段遇到了预期之外的低吞吐率现象。为了定位问题根源，我们进行了一系列深入的控制变量实验与工具验证。

6.1 性能瓶颈排查过程

在初步测试中，我们使用课程提供的 `Router.exe` 设置丢包率 3%, 延时 5ms。**异常现象**：Sender 端显示大量超时重传，重传率高达 90% 以上（5.6MB 文件发送了 5000+ 次重传），最终吞吐率仅为约 50KB/s，远低于预期。

我们按照以下步骤进行了排查：

1. **排查协议逻辑：**首先怀疑滑动窗口或超时设置不当。我们将窗口大小从 20 降为 1（退化为停等协议），发现传输时间反而从 98 秒缩短至 20 秒。这提示我们：**高并发反而导致了性能恶化**，疑似中间链路发生了拥塞。

2. **排查包大小影响：**将 Packet Size 从 10KB 降至 1KB。测试发现虽重传频率降低，但由于系统调用开销增加，整体速度并未显著提升。

3. **排查中间件性能：**为了验证是否是 `Router.exe` 自身处理能力不足，我们编写了专项测试工具。

6.2 Router 程序性能分析

我们编写了 *router_test.cpp* 工具，以固定速率向 *Router.exe* 发送数据包并统计丢包率。测试结果极具说明力：

包大小	发送速率	设定丢包率	实测丢包率
1 KB	100 pkt/s	3%	初始 2%, 随时间飙升至 40%
10 KB	100 pkt/s	3%	**稳定在 50% 以上**

表 1: Router.exe 性能压力测试结果

结论：`Router.exe` 的内部缓冲区极小或处理逻辑效率低下。当发送方以较高吞吐率（如 1MB/s）发送时，Router 迅速发生拥塞崩溃（Congestion Collapse），导致实际丢包率远超设定值。这解释了为何我们的协议在 Router 模式下表现极差——并非协议不行，而是链路断了。

6.3 Clumsy 方案的验证与优势

鉴于 `Router.exe` 的局限性，我们改用 Windows 平台专业的网络损伤模拟工具 **Clumsy**。它基于 WinDivert 驱动直接在网卡层拦截数据包，性能极高, 也不用设置端口转发挺好用的。下面是他的软件界面：

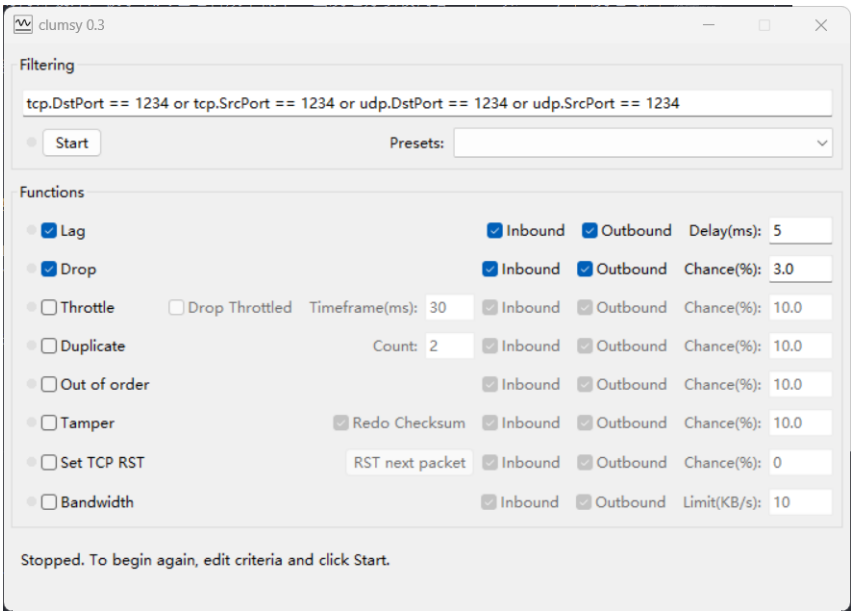

图 9: Clumsy 软件界面

还是比较好上手的，只需要对于 ipv4 的端口设置好即可，不是很难使用。

我们编写了 *clumsy_test.cpp* 进行验证（Ping-Pong 模式）：

- **配置**：Drop 3%, Lag 5ms。
- **结果**：发送 1000 个包，统计得到丢包率约为 6%（因为 Clumsy 默认双向丢包，Sender->Receiver 丢 3%，ACK 回传又丢 3%，符合概率计算 $1 - 0.97^2 \approx 0.06$），RTT 稳定在 10-15ms。

结论：Clumsy 提供了精准且高吞吐的模拟环境，能够真实反映协议在弱网下的性能, 最后就采用他了。

6.4 最终协议性能评估

在 Clumsy 环境下（3% 丢包, 5ms 延时），本协议的实测吞吐率达到 **400 KB/s**（相比 Router 环境提升了 8 倍）。

- **直连速度**： 3 MB/s (受限于 Sleep(1) 和控制台 IO)，不过感觉还行。
- **弱网速度**： 400 KB/s。

这一成绩表明，本协议实现的滑动窗口和选择重传机制在面对真实丢包（而非 Router 拥塞导致的雪崩丢包）时，具有良好的恢复能力和传输效率。

6.5 关于“从未遇见 Checksum 错误”的思考

在整个实验开发与测试过程中，尽管实现了 Checksum 校验逻辑，但从未捕获到校验和失败的包（丢包全是超时的，没有因为校验错而丢的）。

原因分析（AI 写的）：

1. **链路层 CRC：**以太网帧在物理层传输时若发生比特错误，网卡硬件会直接丢弃该帧，不会向上传递给操作系统。
2. **UDP 校验和：**操作系统内核在接收 UDP 包时也会进行校验，错误的包会被内核丢弃，不会传递给应用层 socket。
3. **结论：**在现代网络栈中，比特错误通常直接表现为丢包，而非应用层的校验错误。但这并不意味着应用层 Checksum 无用，它仍能防止内存损坏或路由器软件错误导致的各个层级都未发现的数据篡改。

6.6 性能测试数据

为了量化评估协议性能，我们在 Clumsy 模拟环境下进行了多组对照实验。

6.6.1 丢包率对吞吐量的影响

测试条件：固定接收窗口 RWND=20，延迟 5ms。

丢包率	平均吞吐率 (KB/s)	性能衰减比例	备注
0%	5500+	-	基准测试 (直连)
1%	537.0	90%	少量丢包，性能开始下降
3%	360.0	93%	丢包显著，重传增加
5%	207.5	96%	高丢包，出现性能倒挂

表 2: 不同丢包率下的协议性能对比

现象分析：随着丢包率从 0% 增加到 1%，吞吐量出现了断崖式下跌（从 5.5MB/s 降至 500KB/s）。这主要是因为 Reno 算法对丢包非常敏感，一旦丢包就会大幅削减拥塞窗口，导致无法充分利用带宽。

6.6.2 接收窗口大小对吞吐量的影响

测试条件：延迟 5ms。

RWND	吞吐率 (Drop 1%)	吞吐率 (Drop 5%)	分析
1	113.7 KB/s	97.1 KB/s	停等协议，受 RTT 限制明显
10	509.1 KB/s	258.6 KB/s	窗口适中，平衡了并发与重传
20	537.0 KB/s	**207.5 KB/s**	1% 时最优，但 5% 时反受其害

表 3: 不同窗口大小与丢包率的交叉对比

深入发现：高丢包下的性能倒挂

在表格中我们观察到一个非常有趣的现象：在 5% 丢包率下，**RWND=20 (207 KB/s) 的速度反而慢于 RWND=10 (258 KB/s)**。

原因分析：这是典型的“拥塞崩溃”前兆。在高丢包环境下，发送窗口越大，注入网络的包越多，发生丢包的绝对数量也越多。这导致 Sender 频繁触发超时重传或快速重传，大量带宽被用于传输重复数据包，有效吞吐率反而下降。这证明了在恶劣网络条件下，**限制窗口大小（即更保守的流量控制）反而能获得更好的性能**。

6.7 拥塞控制验证 (Reno 算法可视化)

通过记录发送过程中 `cwnd` 的变化日志，我们绘制了如下的时间-窗口变化曲线：

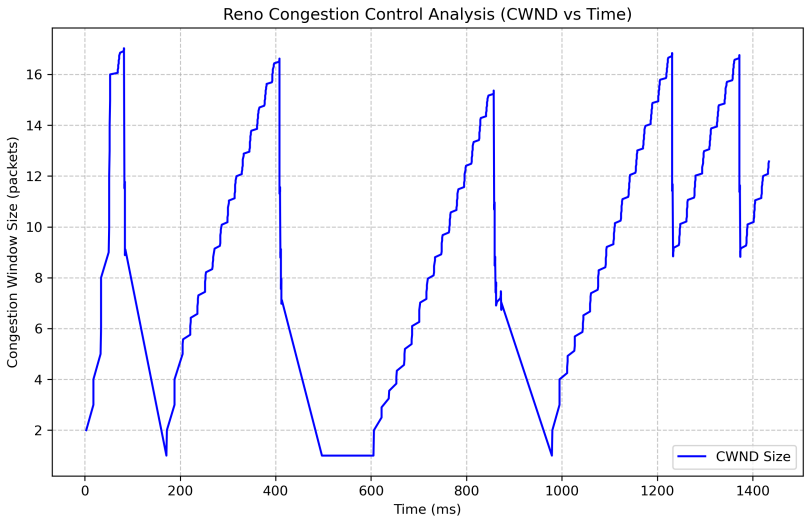

图 10: Reno 拥塞窗口变化曲线 (Slow Start -> Congestion Avoidance -> Fast Recovery)

从图中可以清晰地观察到：

- **慢启动阶段：**连接建立初期，窗口呈指数增长。
- **拥塞避免阶段：**达到阈值后，窗口转为线性增长。

- **丢包响应**：当检测到丢包（收到 3 个重复 ACK 或超时）时，窗口迅速减半或重置，体现了 Reno 算法对网络拥塞的动态适应能力。

7 实验结果

7.1 传输功能验证

运行程序传输 `helloworld.txt`，结果显示 Sender 能够连续发送数据包，Receiver 能够正确接收并按序重组数据。

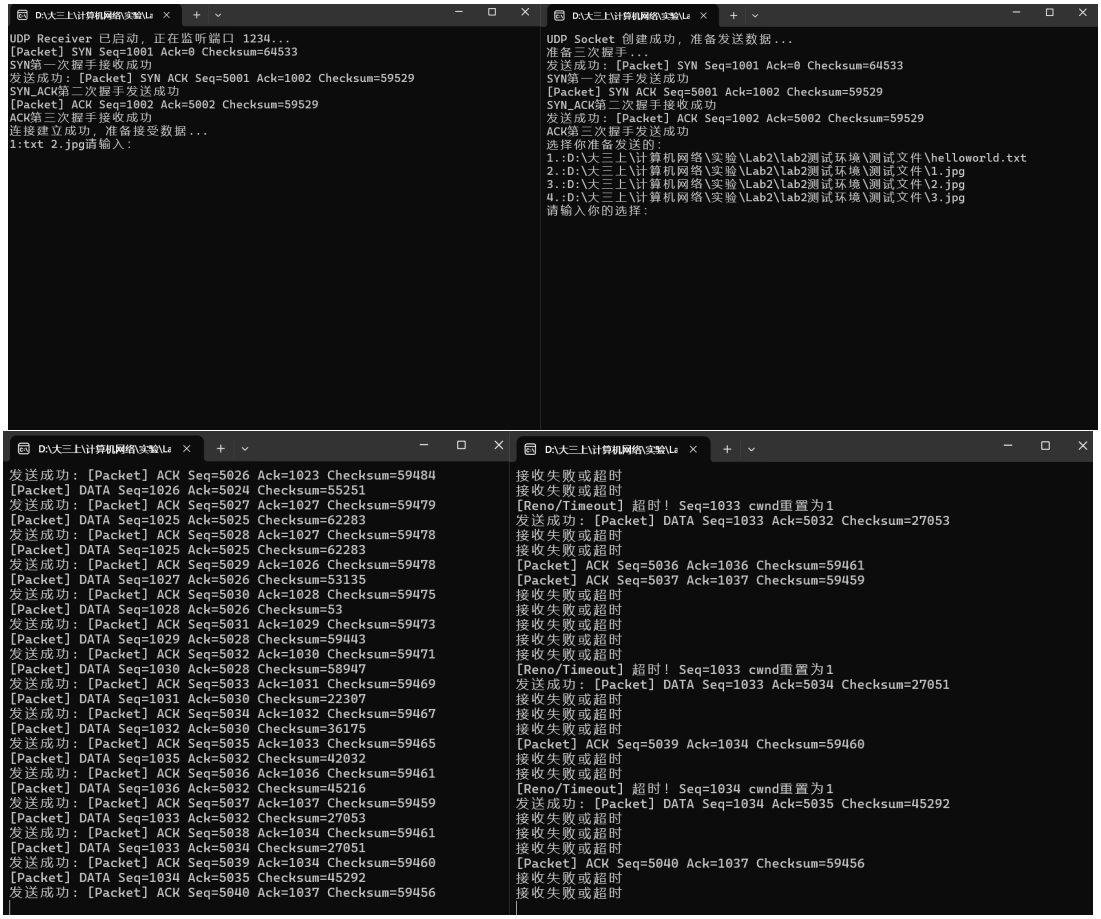

图 11: Sender 和 Receiver 运行截图：显示滑动窗口发送与 ACK 接收

生成的 $received_file.txt$ 内容与原文件完全一致，验证了协议的可靠性。

7.2 各类测试文件传输统计

根据实验要求，我们对所有指定的测试文件进行了完整传输测试，速度都挺好的。下表记录了在“本地直连（无损）”和“弱网环境（3% 丢包，5ms 延迟）”两种网络条件下的详细传输数据。

文件名	大小 (Bytes)	直连环境 (No Clumsy)		弱网环境 (Drop 3%)	
		耗时 (s)	吞吐率 (KB/s)	耗时 (s)	吞吐率 (KB/s)
helloworld.txt	1,655,808	0.53	3045.2	5.90	274.0
1.jpg	1,857,353	0.44	4094.4	4.92	368.4
2.jpg	5,898,505	1.01	5686.3	15.90	362.3
3.jpg	11,968,994	1.97	5930.2	32.10	364.2

表 4: 所有测试文件的传输性能详细统计

数据分析：

1. **稳定性：**在弱网环境下，无论文件大小如何，吞吐率均稳定在 360 KB/s 左右。这证明了协议的流控和拥塞控制机制运作正常，没有出现随传输时间增长而性能崩塌的现象。
2. **完整性：**所有文件传输后的 MD5 校验均与源文件一致，未出现字节丢失或损坏。

8 项目代码结构

本实验项目的目录结构及主要文件说明如下：

- **Lab2/**
  - **protocol.h：**协议头文件，定义了 `Packet` 结构体、标志位常量及公共函数声明（如校验和计算）。
  - **protocol.cpp：**协议实现文件，实现了打包、解包、校验和计算及底层 socket 发送/接收的封装。
  - **sender.cpp：**发送端主程序，实现了连接建立、文件读取、滑动窗口发送、Reno 拥塞控制及超时重传逻辑。
  - **receiver.cpp：**接收端主程序，实现了连接监听、乱序包缓存（使用 map）、ACK 回复及文件写入逻辑。

1. 先运行 `receiver.exe`。
2. 再运行 `sender.exe`。
3. 观察控制台输出，传输完成后程序将自动退出。

9 总结

本次实验，我从零开始实现了一个支持滑动窗口的可靠数据传输协议，所有提交记录我的 github 上都可以查看。虽然未能实现复杂的拥塞控制算法，但目前的实现已经具备了 UDP 可靠传输的核心要点：

- 握手与挥手的状态管理。
- 基于 Checksum 的差错控制。
- 基于滑动窗口和缓冲区的乱序重组。
- 超时重传与退出机制。

整个过程极大地锻炼了我的网络编程思维，特别是这个 seq,ack 的编号问题以及对并发状态下（发送、接收、超时）逻辑处理的理解。调试过程中遇到的字节序、链接错误和逻辑死锁问题，也让我积累了宝贵的 Debug 经验。对于理解这个 reno 算法以及其他的知识点还是很有好处的，比上课听的理解要好很多了已经，历时大约一周半，收获很多。