计算机网络第3-1次实验报告

邢清画 2211999 物联网工程

一、实验目的

基于 UDP 服务设计可靠传输协议并编程实现（3-1）

二、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：建立连接、差错检测、接收确认、超时重传等。流量控制采用停等机制，完成给定测试文件的传输。

- 数据报套接字：UDP
- 协议设计：数据包格式，发送端和接收端交互，详细完整
- 建立连接、断开连接：类似 TCP 的握手、挥手功能
- 差错检验：校验和
- 接收确认、超时重传：rdt2.0、rdt2.1、rdt2.2、rtd3.0 等，亦可自行设计协议
- 单向传输：发送端、接收端
- 日志输出：收到/发送数据包的序号、ACK、校验和等，传输时间与吞吐率
- 测试文件：必须使用助教发的测试文件（1.jpg、2.jpg、3.jpg、helloworld.txt）

三、实验过程

3.1 协议设计

本实验基于 UDP 数据报套接字实现了可靠传输协议，设计了一套类似于 TCP 的可靠数据传输机制，包括连接建立（三次握手）、数据传输（停等协议）以及连接关闭（四次挥手）。协议通过自定义的数据包格式和可靠性增强机制，确保数据的完整性和有序性，旨在在不可靠的 UDP 通信上构建可靠的数据传输机制。协议设计包括以下主要部分：

1. 数据包格式定义

协议中定义了自定义的数据包头部 `Head`，用于控制数据传输的状态和管理。数据包由头部和数据部分组成。

```
struct Head
{   u_short CheckSum;//校验和 16位
    u_short DataLen;//所包含数据长度 16位
    unsigned char Sign;//后三位表示数据包类型,8位
    unsigned char Seq;//序列号（用于验证是否出错及出错的序列号，便于修正）,8位
};
```

协议数据包由以下几个字段组成，每个字段的含义如下：

字段	类型	长度（字节）	说明
`CheckSum`	`u_short`	2	数据包的校验和，用于校验数据的完整性
`DataLen`	`u_short`	2	数据包中实际数据的长度（字节）
`Sign`	`unsigned char`	1	标志位，后三位用于指示数据包的类型和控制信息
`Seq`	`unsigned char`	1	序列号，用于数据包的顺序控制，范围为 0-255
数据内容	`char[]`	可变	数据包携带的实际数据（仅在数据传输时有效）

2. 标志位的使用

标志位 `Sign` 定义了数据包的类型和控制信息。采用 8 位无符号字符，使用其中的后三位表示不同的标志。根据数据包的值是否为1与命名对应，如当ACK为1时命名为ACK，ACK和SYN均为1命名为ACK_SYN。

标志位	值（`hex`）	说明
`SYN`	`0x1`	表示连接建立请求，三次握手的第一步
`ACK`	`0x2`	确认收到数据包或握手请求
`ACK_SYN`	`0x3`	同时表示连接请求和确认（包含SYN和ACK，三次握手的第二步）
`FIN`	`0x4`	表示连接关闭请求，四次挥手的第一步
`FIN_SYN`	`0x5`	同时表示连接关闭请求和同步（包含FIN和SYN,未使用，可扩展）
`ACK_FIN`	`0x6`	确认连接关闭请求（包含ACK和FIN，第三次挥手）
`END`	`0x7`	传输结束标志，用于通知接收端传输完成

3. 传输流程

3.1 连接建立（三次握手）

1. 第一次握手：
    - 发送端发送一个 `SYN` 标志的数据包，表示请求建立连接。
    - 数据包：`Sign= SYN`。
2. 第二次握手：
    - 接收端接收到 `SYN` 包后，回复一个 `ACK` 标志的数据包，表示确认收到连接请求。
    - 数据包：`Sign= ACK`。
3. 第三次握手：
    - 发送端接收到 `ACK` 包后，发送一个 `ACK_SYN` 标志的数据包，表示确认连接建立。
    - 数据包：`Sign= ACK_SYN`。

连接建立成功后，双方进入数据传输阶段。

3.2 数据传输

1. **数据包发送：**

- 发送端将文件分割成多个数据包，每个数据包包含：
  - **头部：**`Head`，包含 `CheckSum`、`DataLen`、`Seq`、`Sign`。
  - **数据部分：**实际传输的文件内容。
- 每个数据包的 `Sign` 为 `0x0`，表示普通数据包。
- 序列号 `seq` 用于确认数据包的顺序，发送后自增并对 256 取模。

2. **数据包接收与确认：**

- 接收端收到数据包后，计算校验和并验证数据完整性。
- 验证通过后，回复一个 `ACK` 数据包，`seq` 与收到的数据包序列号一致。
- 若校验失败或序列号不匹配，接收端丢弃数据包并发送上一个正确的 `ACK` 确认。

3.3 连接关闭（四次挥手）

1. **第一次挥手：**

- **发送端**发送一个 `FIN` 标志的数据包，表示请求关闭连接。
- 数据包：`Sign = FIN`。

2. **第二次挥手：**

- **接收端**收到 `FIN` 包后，回复一个 `ACK` 标志的数据包，表示确认关闭请求。
- 数据包：`Sign = ACK`。

3. **第三次挥手：**

- **接收端**再发送一个 `ACK_FIN` 标志的数据包，表示准备关闭连接。
- 数据包：`Sign = ACK_FIN`。

4. **第四次挥手**（可选，根据实际实现可能未包含）：

- **发送端**收到 `ACK_FIN` 后，回复一个 `END` 标志的数据包，表示连接已关闭。
- 数据包：`Sign = END`。

4. 错误控制与重传机制

1. **校验和验证：**使用 16 位校验和对头部和数据部分进行完整性校验。接收端计算收到数据包的校验和，若不匹配则认为数据损坏。
2. **序列号控制：**序列号 `seq` 用于确认数据包的顺序，防止数据包丢失或重复。接收端**仅接受与期望序列号匹配**的数据包。
3. **超时重传：**发送端在发送数据包后启动定时器，等待接收端的 `ACK` 确认。若在设定时间内未收到确认，则重传数据包，重传次数有限制（如 5 次）。接收端收到重复的数据包时，根据序列号判断，避免数据重复处理。

4. **丢包处理**：通过超时重传机制，尽可能保证数据包最终被成功接收。接收端检测到序列号不匹配时，发送上一个正确的 `ACK`，提示发送端重传。

5. 连接管理（建立与关闭）

1. **初始化**：在通信开始前，双方初始化套接字，设置为非阻塞模式，准备进行握手。
2. **三次握手建立连接**：保证双方同步，建立可靠的传输通道。
3. **数据传输过程**：发送端按照窗口大小和序列号发送数据包。接收端按照序列号顺序接收，进行校验和确认。
4. **四次挥手关闭连接**：确保双方都同意关闭连接，释放资源。

6. 数据包格式示意图

```
+----------------+----------------+----------------+----------------+
|    CheckSum    |    DataLen     |      Sign      |      Seq       |
|    16 bits     |    16 bits     |     8 bits     |    8 bits      |
+----------------+----------------+----------------+----------------+
|                      Data (variable length)                       |
+-------------------------------------------------------------------+
```

- CheckSum：16 位，头部和数据的校验和。
- DataLen：16 位，数据部分的长度。
- Sign：8 位，控制标志位。
- Seq：8 位，序列号。
- Data：可变长度的数据部分，实际传输的文件内容。

协议的主要特性

- **基于 UDP 的可靠传输**：在不可靠的 UDP 协议上，通过自定义的确认、重传和序列号机制，实现可靠的数据传输。
- **简单的头部设计**：头部仅包含必要的控制信息，减少了数据包的开销。
- **支持大文件传输**：通过数据分片和序列号控制，可以传输大于单个数据包大小的文件。
- **超时重传机制**：设置了重传次数和超时时间，确保在网络不稳定的情况下尽可能完成传输。
- **连接的建立和关闭**：通过三次握手和四次挥手，模拟 TCP 的连接管理，确保通信双方的状态同步。

3.2 关键内容

1. 数据包结构定义

协议头部 struct Head 是整个传输协议的核心数据结构，用于描述数据包的元信息。它包含了校验和、数据长度、标志位以及序列号等字段，确保数据包的完整性、正确性和有序性。在整个文件传输过程中，每个数据包都附带一个 Head 信息，接收端通过解析 Head 可以了解数据包的属性并进行相应的处理。

通过 `flag` 的不同设置，该结构既可以用于数据包的传输，也可以用于握手、确认等控制信号的传递。

```
struct Head
{
    u_short CheckSum;//校验和 16位
    u_short DataLen;//所包含数据长度 16位
    unsigned char Sign;//后三位表示数据包类型,8位
    unsigned char Seq;//序列号（用于验证是否出错及出错的序列号，便于修正）,8位
    Head()
    {
        CheckSum = 0;
        DataLen = 0;
        Sign = 0;
        Seq = 0;
    }
};
```

1. `CheckSum` （16位）：校验和，用于检测数据包在传输过程中是否被篡改或损坏。发送端在发送数据包前计算整个数据包的校验和，并将其填入 `checksum` 字段；接收端收到数据包后重新计算校验和，并与接收到的值对比，以验证数据包的完整性。
2. `DataLen` （16位）：表示数据部分的长度（以字节为单位）。发送端根据数据的实际大小动态设置 `DataLen`，接收端通过该字段知道当前数据包中有效数据的大小。
3. `Sign` （8位）：标志位，用于标识数据包的类型。协议中定义了多种标志位值：

- `SYN` （0x1）：同步信号，用于连接建立的第一次握手。
- `ACK` （0x2）：确认信号，用于数据包接收后的确认。
- `END` （0x7）：传输结束信号，用于通知接收端所有数据包已发送完成。

4. `seq` （8位）：序列号，用于标识数据包在整个文件中的位置，确保数据包的有序性。发送端为每个数据包分配一个递增的序列号，接收端通过序列号检查是否按顺序接收数据。

- 序列号的长度为 8 位（0-255），采用模 256 的方式循环使用，能够满足一般文件传输的需求。

2. 校验和计算

2.1 计算方法

`u_short CalculateChecksum(u_short* head, int size)` 用于计算校验和，它根据数据内容计算一个 16 位的校验和，用于验证数据在传输过程中是否被篡改或损坏。发送端在数据包发送前计算校验和并填入协议头，接收端收到数据后重新计算校验和并与接收到的值对比，从而判断数据包是否完整。

```
u_short CalculateChecksum(u_short* head, int size)
{
    int count = (size + 1) / 2;//计算循环次数，每次循环计算两个16位的数据
    u_short* buf = (u_short*)malloc(size + 1);//动态分配字符串变量
    memset(buf, 0, size + 1);//数组清空
    memcpy(buf, head, size);//数组赋值
    u_long checkSum = 0;
    while (count--) {
        checkSum += *buf++;//将2个16进制数相加
        if (checkSum & 0xffff0000) {//如果相加结果的高十六位大于一，将十六位置零，并将最低位加
一
```

```
            checkSum &= 0xffff;
            checkSum++;
        }
    }
    return ~(checkSum & 0xffff);//对最后的结果取反
}
```

1. **输入参数：**

- `head`：指向要进行校验和计算的数据缓冲区。
- `size`：数据缓冲区的大小（以字节为单位）。

2. **计算流程：**

- **动态分配缓冲区**：分配 `size + 1` 字节的内存空间，并初始化为 0。这是为了确保数据的对齐与便于处理长度为奇数的数据。
- **逐字（16 位）累加**：将数据内容两两合并（16 位为一组）并累加到 `checkSum` 中。
- **处理溢出**：如果累加过程中发生了高 16 位溢出，则将高位进位到低 16 位中。
- **取反返回**：对最终结果取反，作为校验和。

3. **溢出处理**：在累加过程中，如果 `checkSum` 的高 16 位存在非零值（溢出），说明累加超出了 16 位范围。这时，需要手动将高位加到低位中，从而实现“环绕加法”（carry-around addition），这是校验和算法的核心步骤。
4. **返回值**：是一个 16 位的无符号整数，表示输入数据的校验和。发送端和接收端的校验和计算结果应一致，如果不一致，则说明数据在传输中被修改或损坏。

在发送端：每次发送数据前，通过调用 `CalculateChecksum()` 函数计算数据包的校验和，并将结果填入协议头部的 `checksum` 字段中。

在接收端：收到数据后，再次调用 `CalculateChecksum()` 函数对数据包的所有内容（包括协议头）进行校验和计算。如果返回值为 0，说明数据包未发生篡改或损坏；否则，数据包可能有误。

2.2 校验和出错处理

1. **发送端处理：**
   当接收端发现校验和出错时，它会通过 ACK 请求发送端重发对应序列号的数据包。发送端在收到 ACK 包后，会检查其中的序列号并判断需要重传哪些数据包。
2. **接收端处理：**
   接收端会继续等待，直到收到序列号匹配且校验和正确的数据包。

主要体现在数据接收的部分（`Recievefile` 函数中）。如果校验和出错，接收端会拒绝确认当前数据包，并通过重新发送 ACK 的方式请求发送端重传对应的数据包。

2.2.1 校验和出错的检测逻辑

在 `Recievefile` 函数中（接收端），校验和检测的代码如下：

```
if (head.Sign == unsigned char(0) && CalculateChecksum((u_short*)buf, recvlength - sizeof(head)))//校验和不为0且Sign是无符号字符
    // 校验和出错或者数据包有误的处理
    if (seq != int(head.Seq))
            {//seq不相等，数据包接收有误
                Newpacketcheck(head, ACK, buf, seq);
                //重新发送ACK
        continue; // 丢弃当前数据包，等待重传 }
}
```

```
void PrintPacketLog(const Head& head, const string& action)
{//重新发送ACK,根据不同情况action进行调用
    string SIGN;
    switch (head.Sign) {
    case 1: SIGN = "【SYN】"; break;
    case 2: SIGN = "【ACK】"; break;
    case 3: SIGN = "【SYN ACK】"; break;
    case 4: SIGN = "【FIN】"; break;
    case 5: SIGN = "【FIN SYN】"; break;
    case 6: SIGN = "【FIN ACK】"; break;
    case 7: SIGN = "【RESEND】"; break;
    case 8: SIGN = "【END】"; break;
    default: SIGN = "【SEND】"; break;
    }

    cout << "【" << action << "】标志位 = " << SIGN << " 序列号 = " << int(head.Seq) << " 校验和 = " << int(head.CheckSum) << endl;
}
```

1. 校验和检查：
   调用 `CalculateChecksum` 函数对接收到的数据包进行校验和验证。如果**校验和结果不为零**，说明数据包在传输过程中出现了数据损坏。
2. 序列号验证：
   即使校验和正确，也需要进一步验证**序列号是否和预期的序列号匹配**。如果序列号不匹配，说明该数据包是**乱序到达或重复**的数据包。

2.2.2 校验和出错的处理流程

1. **构造重传请求：** 调用 `Newpacketcheck` 函数创建一个 ACK 数据包，ACK 的序列号等于接收端当前的 `seq`（接收端期望的下一个数据包序列号）。

```
Newpacketcheck(head, ACK, buf, seq); // 创建 ACK 包
```

2. **发送重传请求：** 通过 `sendto` 函数将 ACK 包发送回发送端。

```
sendto(socket, buf, sizeof(head), 0, (sockaddr*)&addr, addrlength);
```

3. **丢弃错误数据包：** 当前数据包被丢弃，代码通过 `continue` 跳过本次循环，等待发送端重新传输该数据包。

```
continue; // 丢弃当前数据包，等待重传
```

3. 三次握手实现（连接建立）

模拟 TCP 的三次握手过程，建立发送端与接收端之间的可靠连接。这一过程在传输开始之前完成，用于确认双方的通信能力和基本参数一致性。

代码解析：

1. 函数输入参数：
   - `socket`：表示用于通信的 UDP 套接字。
   - `addr`：接收端的网络地址信息，包括 IP 和端口号。
2. 本地变量：
   - `length`：存储地址的长度，用于 `sendto` 和 `recvfrom` 函数。
   - `head`：协议头，用于发送和接收报文时携带状态信息。
   - `buff`：动态分配的缓冲区，用于存储要发送或接收的报文。

具体实现步骤：

第一次握手：

发送端向接收端发送一个 `SYN` 标志的数据包，请求建立连接。如果发送成功，输出提示信息：“第一次握手成功【SYN】”。

1. 初始化变量：

- 首先将 `Handshakesuc` 初始化为 0，用于跟踪握手的成功状态。
- 创建并初始化一个 `Head` 结构体对象 `head`，该结构体用于存储数据包的头部信息，包括标志位、校验和等。

2. 设置数据包的标志位和计算校验和：

- 数据包的标志位 `Sign` 被设置为 `SYN`，这是 TCP 协议中用来发起连接请求的标志。
- 然后计算该数据包头部的校验和，并存储到 `head.CheckSum` 中，确保数据在传输过程中能够检测到是否发生了错误。

3. 创建缓冲区并将数据包头复制到缓冲区：

- 创建一个缓冲区 `buff`，大小与数据包头部大小一致，并将 `head` 结构体中的数据复制到该缓冲区，以便后续发送。

```
    Handshakesuc = 0;    //Handshakesuc 被初始化为 0，用于跟踪握手的成功状态。
    Head head = Head(); //数据首部
    head.Sign = SYN; //将数据包的标志位设置为 SYN（同步请求），这是 TCP 握手的第一步
    head.CheckSum = CalculateChecksum((u_short*)&head, sizeof(head)); //计算校验和
    char* buff = new char[sizeof(head)];//创建了一个大小为 sizeof(head) 的缓冲区 buff，用来存放数据包头部信息。
```

```
    memcpy(buff, &head, sizeof(head));//使用 memcpy 将 head 结构体中的数据复制到 buff 缓冲区中。
    if (sendto(socket, buff, sizeof(head), 0, (sockaddr*)&addr, length) == SOCKET_ERROR)//使用 sendto 函数发送数据包，buff 中包含了第一次握手的头部信息。通过 socket 套接字将数据发送到指定的 addr 地址。
    {//发送失败
        cout << "【第一次握手失败】" << endl;
        return;
    }
    cout << "第一次握手成功【SYN】" << endl;
    clock_t handstime = clock(); //记录发送第一次握手的时间，以便后续进行超时控制和统计。
    u_long mode = 1;
    ioctlsocket(socket, FIONBIO, &mode); //设置非阻塞模式,在非阻塞模式下，recv 或 send 等函数调用不会阻塞程序执行，这对于处理超时重传和异步接收数据很有帮助。
    int handscount1 = 0;//记录超时重传次数，如果握手在一定时间内未完成，程序会尝试重新发送数据包。
```

4. 发送数据包头（第一次握手）：

- 使用 `sendto` 函数将数据包头发送到指定的服务器地址。如果发送失败（`sendto` 返回 `SOCKET_ERROR`），则输出握手失败的信息并退出函数。
- 如果发送成功，输出日志“第一次握手成功【SYN】”，表示客户端已成功发送了第一次握手的数据包。

5. 记录发送时间并设置非阻塞模式：

- 使用 `clock()` 函数记录发送第一次握手的时间，用于后续的超时控制和统计。
- 使用 `ioctlsocket` 函数将套接字设置为非阻塞模式，以便后续操作不会因阻塞而影响程序的执行。

6. 初始化超时重传计数器：

- 创建 `handscount1` 计数器，用于记录第一次握手过程中的超时重传次数。如果握手未在规定时间内完成，程序将尝试重新发送数据包。

总结：

第二次握手：

确认接收端已经收到了发送端的 `SYN` 报文，并发送了确认包 `ACK`。如果接收到符合条件的包，输出提示信息：“第二次握手成功【ACK】”。

1. 等待接收服务器响应：

- 使用 `recvfrom` 函数等待接收来自服务器的响应。该函数接收数据包并将其存储到 `buff` 缓冲区中。如果接收到的数据包大小为零或发生错误（即返回值小于等于零），则表示接收失败，需要继续等待。

2. 超时重传机制：

- 在等待接收过程中，如果超出了设定的超时时间 `RETIME`（通过 `clock()` 与 `handstime` 的时间差计算），则触发重传机制。

- 在超时后，将数据包头部 `head` 重新复制到缓冲区 `buff`，然后使用 `sendto` 函数重新发送数据包到服务器，表示重试请求。
- 重传时，重新记录发送时间 `handstime`，并输出提示信息“连接超时！等待重传……”。同时，增加 `handscount1` 的计数，记录已发生的超时重传次数。
- 如果超时重传次数超过了预定的最大重试次数 `Handshakecount`，则输出“等待超时”，并终止握手过程。

```
while (recvfrom(socket, buff, sizeof(head), 0, (sockaddr*)&addr, &length) <= 0)
    {//等待接收
        if (clock() - handstime > RETIME)//超时重传
        {
            memcpy(buff, &head, sizeof(head));//将首部放入缓冲区
            sendto(socket, buff, sizeof(head), 0, (sockaddr*)&addr, length); //再次发
送
            handstime = clock(); //计时
            cout << "【连接超时！等待重传……】" << endl;
            handscount1++;
            if (handscount1 == Handshakecount) {
                cout << "【等待超时】" << endl;
                return;
            }
        }
    }
    memcpy(&head, buff, sizeof(head)); //ACK正确且检查校验和无误
    if (head.Sign == ACK && CalculateChecksum((u_short*)&head, sizeof(head) == 0))
    {
        cout << "第二次握手成功【SYN ACK】" << endl;
        handscount1 = 0;
    }
    else
    {
        cout << "【第二次握手失败】" << endl;
        return;
    }
```

3. 接收正确响应并校验：

- 如果 `recvfrom` 成功接收到数据包，接下来会将接收到的响应数据包头部从 `buff` 缓冲区复制到 `head` 结构体中。
- 接收的数据包头的标志位 `Sign` 应该是 `ACK`，并且通过 `CalculateChecksum` 函数计算出的校验和必须为零，表示数据包在传输过程中没有损坏。

4. 确认握手成功或失败：

- 如果接收到的响应数据包头部的标志位为 `ACK` 且校验和正确，则表示第二次握手成功，客户端输出“第二次握手成功【SYN ACK】”的日志，并将重传计数器 `handscount1` 归零，准备进入后续的握手过程或数据传输。
- 如果接收到的响应不符合预期（即标志位不是 `ACK` 或校验和不正确），则表示第二次握手失败，客户端输出“第二次握手失败”的日志，并终止握手过程。

第三次握手：

发送端确认接收端的 `ACK` 报文，并向接收端发送 `ACK_SYN` 报文，表明连接建立成功。输出提示信息：“第三次握手成功【ACK_SYN】”。

1. 发送握手请求：

- 在第三次握手中，客户端准备发送 `ACK_SYN` 标志位的握手请求包。
- 首先，设置数据包头部 `head.Sign = ACK_SYN`，表示这是一个带有 `ACK` 和 `SYN` 标志位的数据包，符合 TCP 协议中三次握手的第三次握手包的要求。
- 接着，调用 `CalculateChecksum` 函数计算数据包头的校验和，确保数据包在传输过程中不会出现损坏。
- 使用 `sendto` 函数发送数据包到服务器，表示客户端发起的第三次握手请求。

2. 等待服务器回应：

- 设置一个标志变量 `win = 0`，用于检查是否成功接收到服务器的回应。
- 通过 `clock()` 获取当前时间并与之前记录的握手发送时间 `handstime` 比较，如果时间差小于等于超时时间 `RETIME`，则继续等待服务器回应。
- 如果 `recvfrom` 函数成功接收到服务器的回应数据包，设置 `win = 1`，表示握手成功，跳出等待循环。

```
    head.Sign = ACK_SYN; //ACK=1 SYN=1
    head.CheckSum = CalculateChecksum((u_short*)&head, sizeof(head)); //计算校验和
    sendto(socket, (char*)&head, sizeof(head), 0, (sockaddr*)&addr, length); //发送握手请求
    bool win = 0; //检验是否连接成功的标志
    while (clock() - handstime <= RETIME)
    {//等待回应
        if (recvfrom(socket, buff, sizeof(head), 0, (sockaddr*)&addr, &length))
        {//收到报文
            win = 1;
            break;  }
        //选择重发
        memcpy(buff, &head, sizeof(head));
        sendto(socket, buff, sizeof(head), 0, (sockaddr*)&addr, length);
        handstime = clock();
        handscount1++;
        if (handscount1 == Handshakecount) {
            cout << "【等待超时】" << endl;
            return;}}
    if (!win)
    {cout << "【第三次握手失败】" << endl;
        return;}
    cout << "第三次握手成功【ACK】" << endl;
    Handshakesuc = 1;
    cout << "【连接成功】" << endl;
}
```

3. 超时重传机制：

- 如果在超时时间内没有收到服务器回应，则触发重传机制。
- 将数据包头部 `head` 重新复制到缓冲区 `buff`，并通过 `sendto` 函数重新发送握手请求包。
- 重新记录发送时间 `handstime`，并增加 `handscount1` 计数，记录已发生的重传次数。
- 如果超时重传次数超过了预定的最大重试次数 `Handshakecount`，则输出“等待超时”的提示，并终止握手过程。

4. 连接成功确认：

- 如果成功收到服务器回应并且不超时，表示第三次握手成功，客户端输出“第三次握手成功【ACK】”的日志。
- 设置 `Handshakesuc = 1`，表示连接已经成功建立。
- 输出“连接成功”的提示，表示客户端与服务器之间的连接已经建立。

5. 第三次握手失败：

- 如果在超时重试后仍然没有收到服务器回应，即 `win` 仍然为 `0`，则表示第三次握手失败，客户端输出“第三次握手失败”的日志并终止连接。

4. 数据传输

4.1 发送端

`Sendpacket` 函数实现了将单个数据包发送至接收端的核心逻辑。完成了数据包的组装、校验和发送，还实现了超时重传机制以确保数据包能够可靠传输到接收端并获得确认（ACK）。

4.1.1 数据包的组装

- 初始化头部变量 `Head` 并填充关键信息：数据长度（`DataLen`）标明数据部分的长度。序列号（`seq`）用来标识当前数据包，确保接收端能够正确识别包的顺序。
- 使用缓冲区 `buf`，将头部信息和数据部分按顺序存放：头部拷贝到缓冲区的前面。数据部分紧随头部之后。
- 计算校验和（`checksum`）：校验和用于验证数据包的完整性。将整个数据包（包括头部和数据部分）传入 `CalculateChecksum` 函数，计算校验和。将计算结果填充到头部的 `checksum` 字段，并更新缓冲区中的头部内容。

```
void Sendpacket(SOCKET& socket, SOCKADDR_IN& addr, char* data, int length, int& seq)
{
    Head head;
    char* buf = new char[MAXLEN + sizeof(head)];
    head.DataLen = length; //使用传入的data的长度定义头部datasize
    head.Seq = unsigned char(seq);//序列号
    memcpy(buf, &head, sizeof(head)); //拷贝首部的数据
    memcpy(buf + sizeof(head), data, sizeof(head) + length); //数据data拷贝到缓冲数组
    head.CheckSum = CalculateChecksum((u_short*)buf, sizeof(head) + length);//计算数据部分的校验和
    memcpy(buf, &head, sizeof(head)); //更新后的头部再次拷贝到缓冲数组
    ...
}
```

4.1.2 数据包发送

组装完成的数据包需要通过 UDP 套接字发送至接收端：使用 `sendto` 函数将缓冲区中的数据包发送到接收端。打印信息确认当前发送包的序列号，便于观察发送进度。

```
    sendto(socket, buf, length + sizeof(head), 0, (sockaddr*)&addr, addrlength);//发送
    PrintSendLog(head); // 打印日志
```

4.1.3 超时重传机制

可靠传输的核心是确保接收端收到数据包并发送确认（ACK）。如果超时未收到 ACK，发送端需要重新发送数据包。

- **记录发送时间**：使用 `clock_t` 记录每次发送的起始时间。
- **等待接收确认**：使用 `recvfrom` 检查是否收到 ACK。
  - 如果接收到 ACK，退出等待循环。
  - 如果超过预设的超时时间（`RETIME`）仍未收到 ACK，则重新发送数据包，并更新发送时间。
  - 打印超时重传的提示信息。

该机制保证了数据包在可能的网络丢包情况下，仍然能够最终到达接收端。

```
...
clock_t starttime = clock(); // 记录发送时间
while (recvfrom(socket, buf, MAXLEN, 0, (sockaddr*)&addr, &addrlength) <= 0){ // 等待接收 ACK
    if (clock() - starttime > RETIME) { // 超过重传时间
        sendto(socket, buf, length + sizeof(head), 0, (sockaddr*)&addr, sizeof(addr)); // 重传数据包
        starttime = clock(); // 重置发送时间
        cout << "超时重传" << endl; // 打印重传提示
    }
}
```

4.2 接收端

`Recievefile` 函数是接收端用来接收发送端传输的文件数据的核心部分。该函数通过循环接收数据包并逐一处理，确保数据包的完整性和顺序，同时在接收到正确的数据包后向发送端回传确认（ACK）。如果接收到结束标志（END），则终止接收并返回接收的数据总长度。

代码逻辑：

4.2.1 函数结构与初始化

- `addrlength`：存储发送端地址长度，用于与 `recvfrom` 和 `sendto` 函数配合使用。
- `Head head`：用于存储接收到的数据包的头部信息，包括校验和、标志位、序列号等。
- `buf`：为接收数据包准备的缓冲区，其大小足以容纳头部信息和数据部分。

```
int Recievefile(SOCKET& socket, SOCKADDR_IN& addr, char* data) {
    int addrlength = sizeof(addr);
    Head head;
    char* buf = new char[maxsize + sizeof(head)];
    int seq = 0, sum = 0;
    ...}
```

4.2.2 循环接收数据包

函数通过 `recvfrom` 获取来自发送端的每个数据包，并对其内容进行解析和处理。

- 数据接收：
  调用 `recvfrom` 接收发送端的数据包，并将其存储到缓冲区 `buf` 中，同时解析头部信息到 `head`。

```
 while (1) {
        int recvlength = recvfrom(socket, buf, sizeof(head) + maxsize, 0,
(sockaddr*)&addr, &addrlength);
        memcpy(&head, buf, sizeof(head));
```

- 结束标志检测：
  检查接收到的标志位是否为 `END` 且校验和无误。如果满足条件，则表示文件传输完成，打印提示并跳出循环。

```
if (head.Sign == END && CalculateChecksum((u_short*)&head, sizeof(head)) == 0)//END
标志位，校验和为0，结束
{ cout << "【传输成功】" << endl;
  break; //结束跳出while循环
}
```

- 数据包校验与顺序检查：
  检查数据包是否通过校验和验证，同时验证其序列号是否与接收端当前期望的序列号一致：
  - 如果通过校验且序列号匹配，则将数据部分拷贝到目标缓冲区 `data` 中，更新接收总长度 `sum` 和下一个期望序列号 `seq`。
  - 如果校验或序列号不匹配，则丢弃数据包并继续等待。

```
// 处理正确的数据包
     PrintPacketLog(head, "接收");
     char* buf_data = new char[recvlength - sizeof(head)]; //数组的大小是接收到的报文长度
减去头部大小
     memcpy(buf_data, buf + sizeof(head), recvlength - sizeof(head)); //从头部后面开始
拷贝，把数据拷贝到缓冲数组
     memcpy(data + sum, buf_data, recvlength - sizeof(head));
     sum = sum + int(head.DataLen);
//初始化首部
     Newpacketcheck(head, ACK, buf, seq);
//发送ACK
     sendto(socket, buf, sizeof(head), 0, (sockaddr*)&addr, addrlength);
     PrintPacketLog(head, "发送");
     seq++;//序列号加
```

4.2.3 发送 ACK 确认

接收端在处理数据包后，无论是否通过校验和检查，都会向发送端发送一个 ACK 数据包以通知接收状态。这通过以下步骤实现：

- 调用 newbag 函数生成一个包含 ACK 标志的头部数据包。
- 使用 sendto 函数将生成的 ACK 数据包发送回发送端。

```
Newpacket(head, END, buf);
sendto(socket, buf, sizeof(head), 0, (sockaddr*)&addr, addrlength); // 发送 ACK 确认
```

5. 四次挥手实现（连接关闭）

由于第二次挥手和第三次挥手可以重合在一次，因此代码只写了三次挥手。第一次挥手 由发送端发起，组装数据包后进行发送。确保了双方在连接关闭时的状态一致性，避免了通信资源的浪费。

```
void Fourway_Wavehand(SOCKET& socket, SOCKADDR_IN& addr)
{   int addrlength = sizeof(addr);
    Head head;
    char* buff = new char[sizeof(head)];
 //具体四次挥手......
}
```

第一次挥手（FIN）

1. 设置数据包头部：

- 在第一次挥手阶段，客户端准备发送一个 `FIN` 标志位的数据包，表示客户端已经没有数据发送，准备关闭连接。
- `head.Sign = FIN;` 将数据包头部的标志位设置为 `FIN`，按照 TCP 协议的规定，`FIN` 表示客户端请求断开连接。
- 计算数据包头的校验和，调用 `CalculateChecksum` 函数确保数据在传输过程中不会发生损坏。
- 使用 `memcpy` 将数据包头部内容拷贝到缓冲区 `buff` 中。

2. 发送数据包：

- 使用 `sendto` 函数将包含 `FIN` 标志位的数据包发送到服务器端。该包的长度为数据包头部的长度（即 `sizeof(head)`）。
- 如果发送失败（即返回值为 `SOCKET_ERROR`），则输出错误信息“【第一次挥手失败】”并返回，终止连接过程。

```
//第一次挥手
    head.Sign = FIN;
    //head.checksum = 0;//校验和置0
    head.CheckSum = CalculateChecksum((u_short*)&head, sizeof(head));
    memcpy(buff, &head, sizeof(head));
    if (sendto(socket, buff, sizeof(head), 0, (sockaddr*)&addr, addrlength) ==
SOCKET_ERROR)
    {
```

```
        cout << "【第一次挥手失败】" << endl;
        return;
    }
    cout << "第一次挥手【FIN ACK】" << endl;
    clock_t byetime = clock(); //记录发送第一次挥手时间

    u_long mode = 1;
    ioctlsocket(socket, FIONBIO, &mode);
```

3. 记录发送时间：

- 如果数据包成功发送，输出“第一次挥手【FIN ACK】”的日志。
- 使用 `clock()` 函数记录发送第一次挥手数据包的时间，存储在 `byetime` 变量中，用于后续判断超时和重传的处理。

4. 设置非阻塞模式：

- 使用 `ioctlsocket(socket, FIONBIO, &mode)` 设置套接字为非阻塞模式。非阻塞模式下，`recvfrom` 等接收函数在没有数据可接收时不会阻塞程序，而是立即返回。

第二、三次挥手（ACK）

1. 等待接收响应：

- 在第二次挥手阶段，客户端等待服务器响应。在此过程中，客户端使用 `recvfrom` 函数接收服务器返回的数据包。`recvfrom` 会阻塞等待直到收到数据包或超时。
- 如果 `recvfrom` 返回的字节数小于等于 0，表示没有成功接收到数据包（可能是因为超时或错误），则进入超时重传机制。

2. 超时重传机制：

- 如果客户端在一定时间内没有接收到服务器的响应数据包（即超时），则启动超时重传机制。
- 使用 `clock()` 获取当前时间，与之前发送的 `byetime` 进行比较。如果超时（即当前时间减去发送时间超过设定的超时时间 `RETIME`），则重新发送 `FIN ACK` 数据包。
- 重传前，将数据包头部 `head` 重新复制到缓冲区 `buff`，并使用 `sendto` 发送数据包。
- 重传后，更新发送时间 `byetime`，并继续等待接收响应。如果超时重传次数超过设定的最大次数，程序会退出。

```
//第二次挥手
    while (recvfrom(socket, buff, sizeof(head), 0, (sockaddr*)&addr, &addrlength) <=
0)
    {//等待接收
        if (clock() - byetime > RETIME)//超时重传
        {
            memcpy(buff, &head, sizeof(head));//将首部放入缓冲区
            sendto(socket, buff, sizeof(head), 0, (sockaddr*)&addr, addrlength);
            byetime = clock();
        }
    }
    //进行校验和检验
    memcpy(&head, buff, sizeof(head));
```

```
    if (head.Sign == ACK && CalculateChecksum((u_short*)&head, sizeof(head) == 0))
    {
        cout << "第二、三次挥手【FIN ACK】" << endl;
    }
    else
    {
        cout << "【第二、三次挥手失败】" << endl;
        return;
    }
```

3. 校验和检验：

- 一旦接收到数据包，客户端将收到的数据包头部复制到 `head` 变量中，并进行校验和验证。调用 `CalculateChecksum` 函数计算接收到的数据包的校验和。
- 如果数据包的标志位 `Sign` 为 `ACK` 且校验和正确（即没有出错），则认为第二次和第三次挥手成功，输出“第二、三次挥手【FIN ACK】”日志。
- 如果校验失败或标志位不是 `ACK`，则输出“【第二、三次挥手失败】”日志，表示挥手失败，程序返回。

第四次挥手（ACK_FIN）

1. 准备发送ACK_FIN包：

- 在第四次挥手阶段，客户端向服务器发送确认的 `ACK_FIN` 数据包，表示客户端已经完成连接的终止。
- 首先设置数据包的标志位 `Sign` 为 `ACK_FIN`，这是挥手的确认包，标志着客户端在连接关闭过程中所处的最后一步。
- 然后通过调用 `CalculateChecksum` 计算校验和，确保数据包的完整性和准确性。

2. 发送数据包：

- 使用 `memcpy` 函数将数据包头 `head` 复制到缓冲区 `buff` 中，以便发送。
- 调用 `sendto` 函数将数据包发送到指定的服务器地址 `addr`。如果 `sendto` 返回值为 `-1`，表示发送失败。
- 如果发送失败，程序输出错误日志“【第四次挥手失败】”，并返回，终止连接关闭过程。

```
    head.Sign = ACK_FIN;
    head.CheckSum = CalculateChecksum((u_short*)&head, sizeof(head));//计算校验和
    memcpy(buff, &head, sizeof(head));
    if (sendto(socket, (char*)&head, sizeof(head), 0, (sockaddr*)&addr, addrlength)
== -1)
    {
        cout << "【第四次挥手失败】" << endl;
        return;
    }
    cout << "第四次挥手【ACK】" << endl;
    cout << "【结束连接】" << endl;
```

6. 文件发送与性能统计

6.1 传输时间

`Sendpacket` 函数实现了发送端将文件数据完整地分包并发送至接收端的核心逻辑。它负责将大文件按照一定的包大小（`MAXLEN`）拆分为多个数据包，通过调用 `Sendpacket` 函数依次发送，并在传输完成后输出总耗时。

```
void sendfile(SOCKET& socket, SOCKADDR_IN& addr, char* data, int data_len) {
    int bagsum = data_len / MAXLEN + (data_len % MAXLEN ? 1 : 0);
    int seq = 0;

    clock_t starttime = clock();
    for (int i = 0; i < bagsum; i++) {
        int len = (i == bagsum - 1) ? data_len % MAXLEN : MAXLEN;
        Sendpacket(socket, addr, data + i * MAXLEN, len, seq);
        seq = (seq + 1) % 256;
    }
    clock_t endtime = clock();
    cout << "传输总时间：" << endtime - starttime << "ms" << endl;
}
```

1. 计算数据包总数

在传输前，根据文件数据长度 `data_len` 和每个数据包的最大长度 `MAXLEN`，计算出需要分成的总数据包数 `bagsum`：

- 如果文件长度可以整除最大包大小，则总包数为 `data_len / MAXLEN`。
- 如果有剩余数据，则需额外增加一个包存放剩余的数据。

```
int bagsum = data_len / maxlength + (MAXLEN % MAXLEN ? 1 : 0);
```

2. 循环发送数据包

通过一个循环，将文件数据分块发送：

确定当前包的长度：如果是最后一个包，长度为文件剩余部分的字节数。否则，长度为 maxlength。

```
int len = (i == bagsum - 1) ? data_len % MAXLEN : MAXLEN;
```

调用 `sendbag` 发送：每次从 `data` 指针偏移 `i * maxlength`，获取当前包的数据地址。调用 `sendbag` 函数发送当前数据包，传入序列号 `seq`。

```
Sendpacket(socket, addr, data + i * MAXLEN, len, seq);
```

更新序列号：序列号递增并对 256 取模，保持在 0~255 范围内循环使用。

```
seq++; seq = (seq + 1) % 256;
```

3. 记录传输时间

在开始传输前记录起始时间 `starttime`，传输完成后记录结束时间 `endtime`，并计算总耗时：

- 耗时单位为毫秒（`ms`）。
- 打印传输总时间，便于评估传输性能。

```
clock_t starttime = clock(); // 记录起始时间
clock_t endtime = clock();   // 记录结束时间
cout << "传输总时间：" << endtime - starttime << "ms" << endl; // 输出传输时间
```

6.2 吞吐率

1. **在每个包发送时计算吞吐率：** 你可以在每个包的发送和确认过程中，分别记录时间并实时计算吞吐率。为了做到这一点，你可以把 `Sendpacket` 函数中的 `clock()` 计时和吞吐率计算提取到每个数据包的传输过程中。
2. **为每个包添加吞吐率统计：** 在 `Sendpacket` 函数内加入吞吐率的实时计算。在每个数据包发送后，记录传输时间和吞吐量。

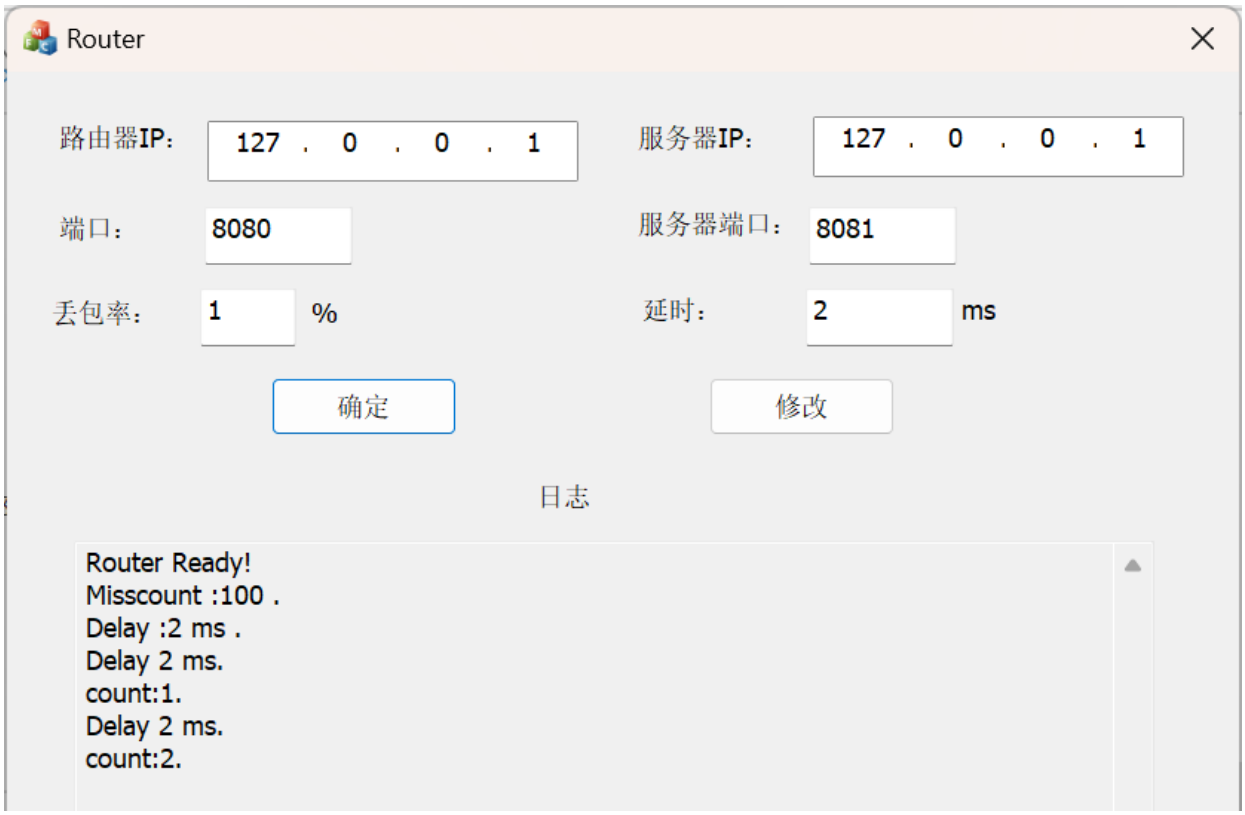
```
 clock_t starttime = clock(); // 计时
 // 打开文件进行写入（以追加模式打开）
    std::ofstream logfile("transfer_log.csv", std::ios::app);
    if (!logfile) {
        std::cerr << "无法打开日志文件！" << std::endl;
        return;
    }
    // 如果是第一次写入（文件为空），写入表头
    if (logfile.tellp() == 0) {
        logfile << "传输数据量 (MB), 传输时间 (秒), 吞吐率 (MB/s)\n";
    }
     for (int i = 0; i < bagsum; i++)
    {   int len;
        if (i == bagsum - 1)
        { // 最后一个数据包是向上取整的结果，因此数据长度是剩余所有
            len = data_len - (bagsum - 1) * MAXLEN;
        }
        else
        { // 非最后一个数据长度均为maxlength
            len = MAXLEN;
        }
        // 记录每个包的开始时间
        clock_t packet_start_time = clock();
        // 计算每个包的传输时间
        double packet_duration = double(clock() - packet_start_time) /
CLOCKS_PER_SEC; // 秒数
        double throughput = (double)len / packet_duration / 1024 / 1024; // 吞吐率
(MB/s)
        std::cout << "包吞吐率: " << throughput << " MB/s" << std::endl;
    }
    // 计算总传输时间
    double duration = double(clock() - starttime) / CLOCKS_PER_SEC; // 秒数
```

```
    // 计算总吞吐率
    double throughput = (double)data_len / duration / 1024 / 1024; // 吞吐率 (MB/s)
    // 将数据写入日志文件
    logfile << data_len / 1024.0 / 1024.0 << ", " << duration << ", " << throughput
<< "\n";
    logfile.flush();  // 确保数据立即写入文件
    // 输出结果
    std::cout << "【传输成功】" << std::endl;
    std::cout << "总数据量: " << data_len / 1024 / 1024 << " MB   " << "传输时间: " <<
duration << " 秒   " << "吞吐率: " << throughput << " MB/s" << std::endl;
```

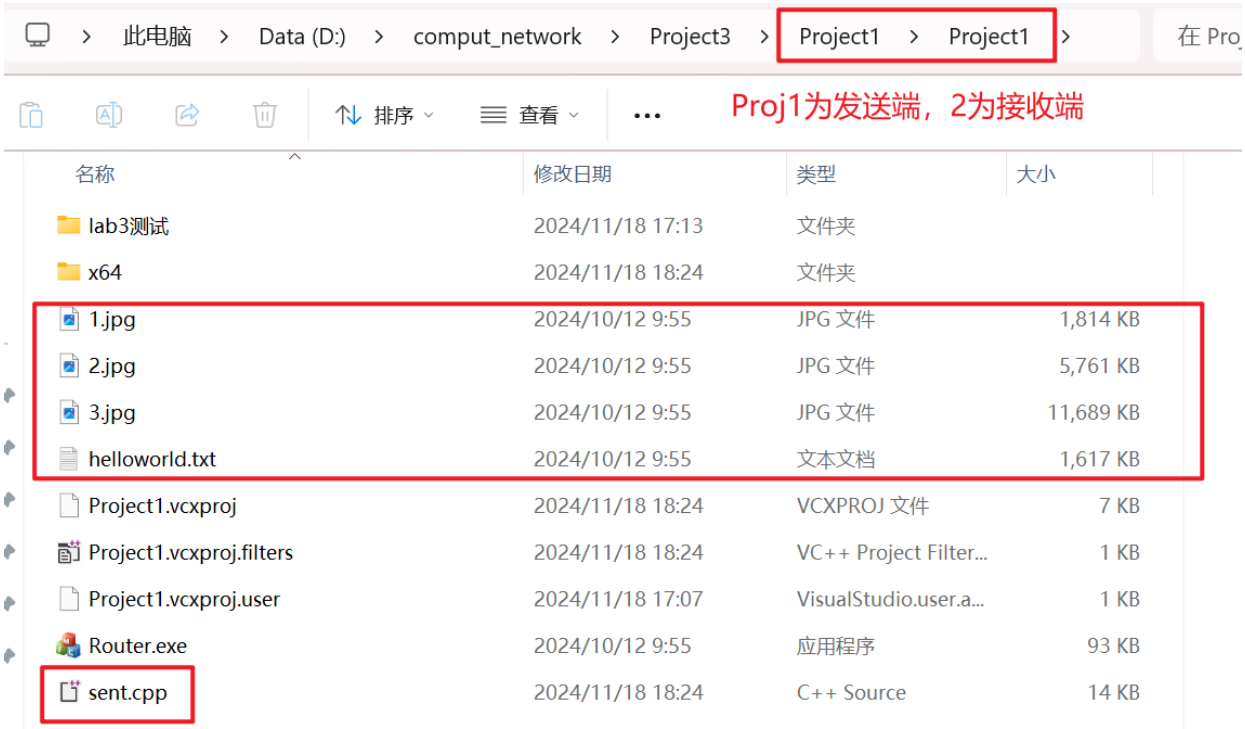
五、实验结果

1. 运行截图

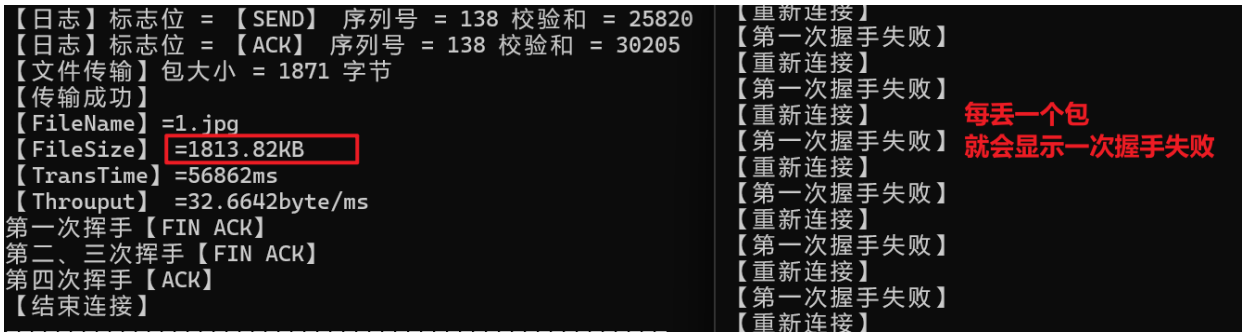
（1）将路由器设置端口号与 IP 地址，并设置丢包率为 1%，延迟 2ms。

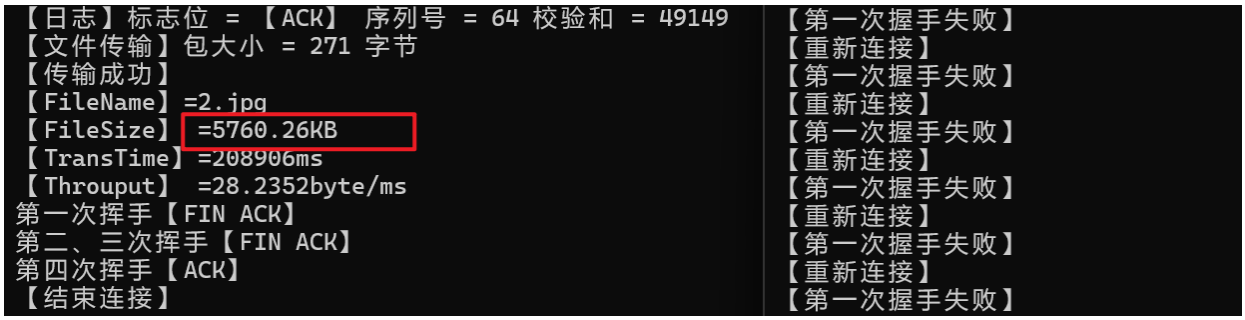


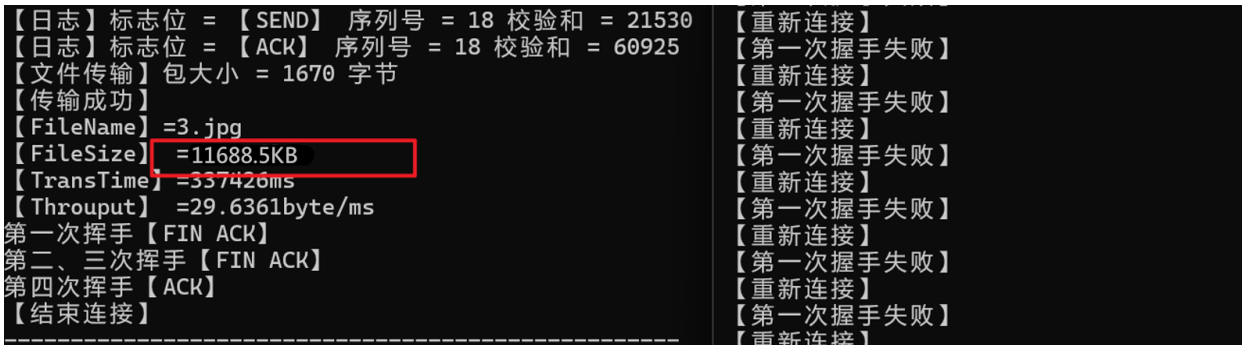
（2）将测试文件放置于发送端的程序目录下



（3）发送 1.jpg,2.jpg,3.jpg,helloworld.txt 文件测试



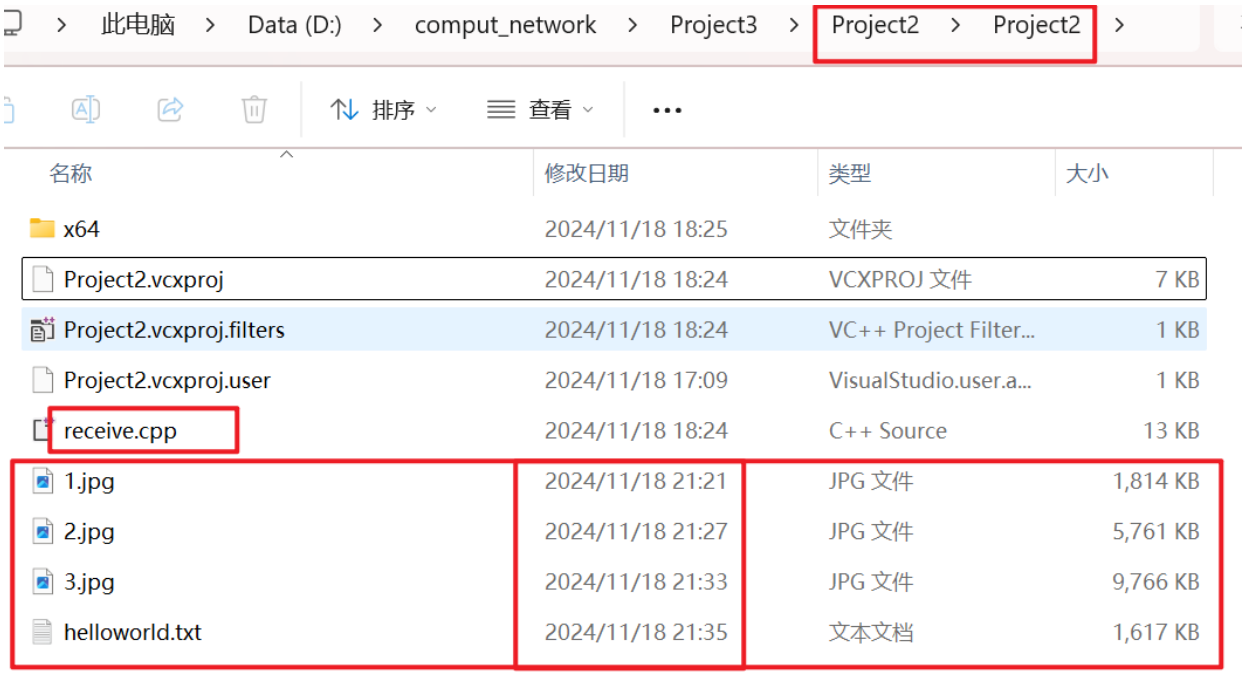




```
【日志】标志位 = 【ACK】 序列号 = 40 校验和 = 55293
【文件传输】包大小 = 1030 字节
【传输成功】
【FileName】=helloworld.txt
【FileSize】=1617KB
【TransTime】=69769ms
【Throuput】 =23.7327byte/ms
第一次挥手【FIN ACK】
第二、三次挥手【FIN ACK】
第四次挥手【ACK】
【结束连接】
```

```
【第一次握手失败】
【重新连接】
【第一次握手失败】
【重新连接】
【第一次握手失败】
【重新连接】
【第一次握手失败】
【重新连接】
【第一次握手失败】
【重新连接】
【第一次握手失败】
```

发现四个文件都传输完成，并且文件大小和系统显示的都相同，经检测也都能正常打开 文件，传输成功。



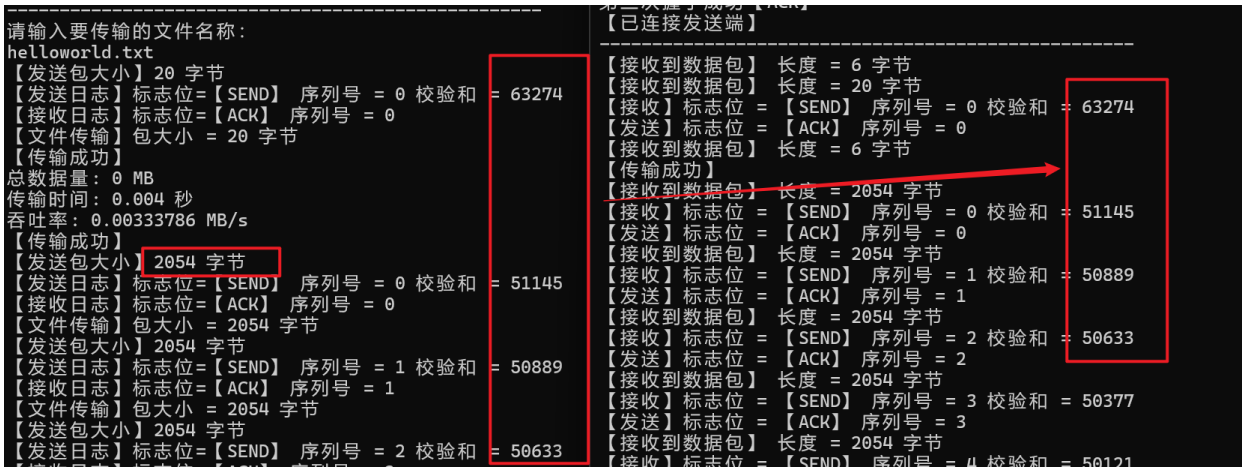
2. 传输结果分析

（1）使用刚才设定的 router 程序来模拟丢包和延迟，输入对应的路由器 IP 和服务器 IP 等信息，实现连接。

```
||                    Sender-->                    ||
||【系统通知】:网络环境初始化成功！！
请输入本机IP地址: 127.0.0.1
请输入端口号: 8080
第一次握手成功【SYN】
第二次握手成功【SYN ACK】
第三次握手成功【ACK】
【连接成功】
请输入要传输的文件名称:
```

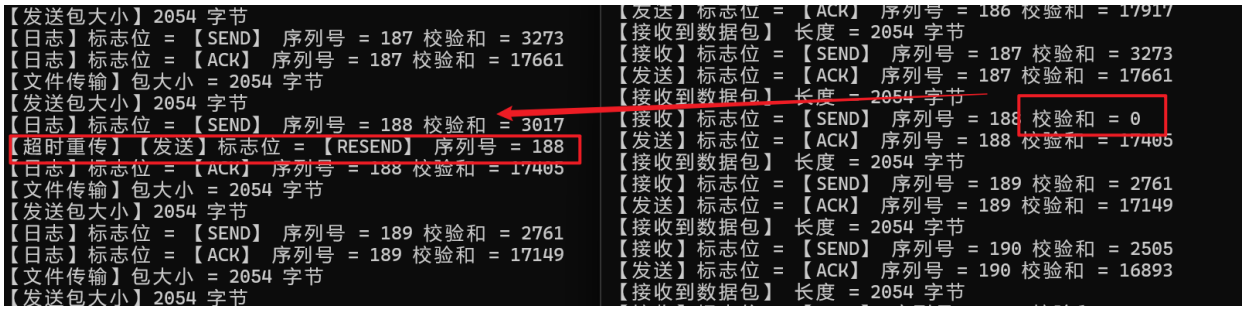
```
||            -->Receiving_End                    ||
||【系统通知】:网络环境初始化成功！！
请输入本机的IP地址: 127.0.0.1
请输入端口号: 8081
【等待发送端连接...】
第一次握手成功【SYN】
第二次握手成功【SYN ACK】
第三次握手成功【ACK】
【已连接发送端】
【接收到数据包】 长度 = 6 字节
```

（2）传输测试文件，输入文件名，开始传输会显示标志位、序列号、包大小、校验和等信息



（ps:上图为只输出单个校验和版本，提交代码在此基础上增加了其他日志（可以在代码中启用输出，可以更明显的观察校验和的变化），简易版本较容易观察对比发送端和接收端的数据）

（3）产生丢包的情况，则会在发送端显示出超时重传，接收端也会发现有校验和为 0 的情况出现，此时接收端向发送端发送数据包告知正确的序列号应当为多少，并请求发送端重新发送。



（4)传输完毕打印最终信息，握手失败代表一次丢包，与router数量保持一致。

3. 性能分析

以发送端向接收端发送helloworld.txt为例：

	传输数据量 (MB)	传输时间 (秒)	吞吐率 (MB/s)
2	2.005859375	1.377123666	1.456557188
3	2.005859375	1.414240913	1.418329336
4	2.005859375	1.379820462	1.453710414
5	2.005859375	1.392093505	1.440894141
6	2.005859375	1.41209713	1.42048258
7	2.005859375	1.413060614	1.419514036
801	2.005859375	1.413221363	1.419352571
802	2.005859375	1.420837407	1.411744486
803	2.005859375	1.375316153	1.458471473
804	2.005859375	1.404466388	1.428200342
805	2.005859375	1.391377958	1.441635153
806	2.005859375	1.411561599	1.421021496
807	1.005859375	0.713645799	1.409465839

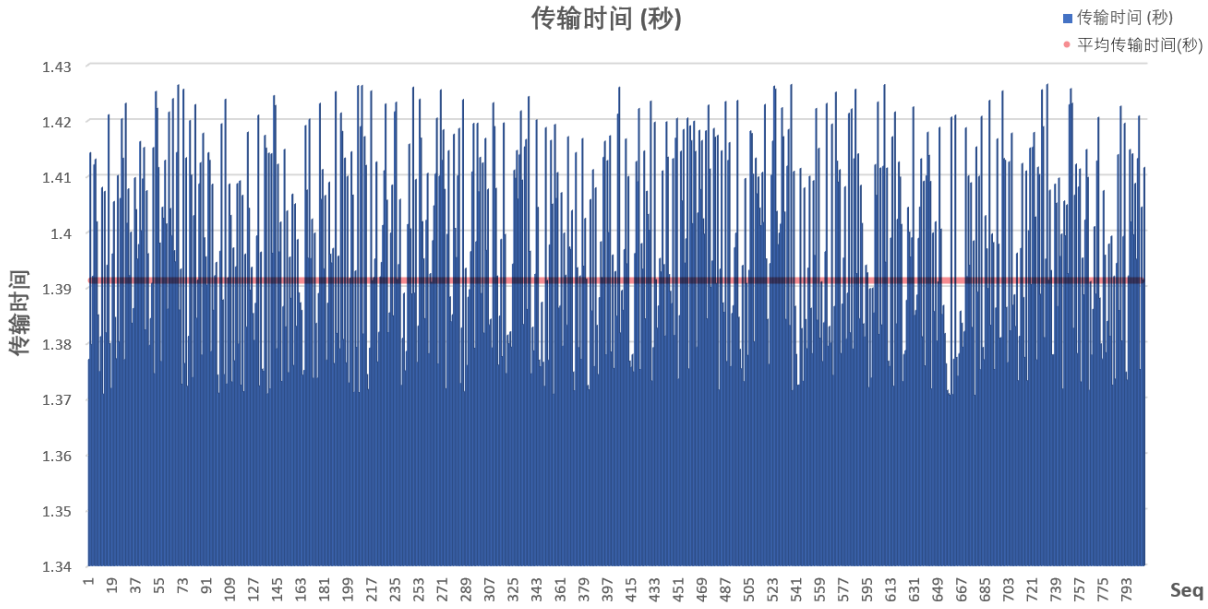
传输过程每次发送数据包会写入到csv文件，除了最后一个数据包是1030字节之外，其他包大小都与设定值MAXLEN保持一致。

3.1 平均传输时间

传输时间（TransTime）是文件从源到目标传输所需要的时间，通常会受到**带宽、延迟、丢包、网络拥塞**等因素的影响。

在一个典型的网络传输场景中，传输时间通常会受到以下因素的影响：

- **带宽限制**：网络的最大传输速率会影响吞吐率。带宽过低会导致吞吐率降低，从而增加传输时间。
- **延迟**：网络中不同的节点之间的传播延迟、排队延迟等都会导致传输时间增加。
- **丢包和重传**：在不稳定的网络环境中，丢包会导致数据包需要重传，从而增加传输时间和降低吞吐率。
- **拥塞控制**：在高流量的网络中，拥塞控制算法（如TCP的慢启动、拥塞避免等）会限制吞吐率，导致传输时间增大。

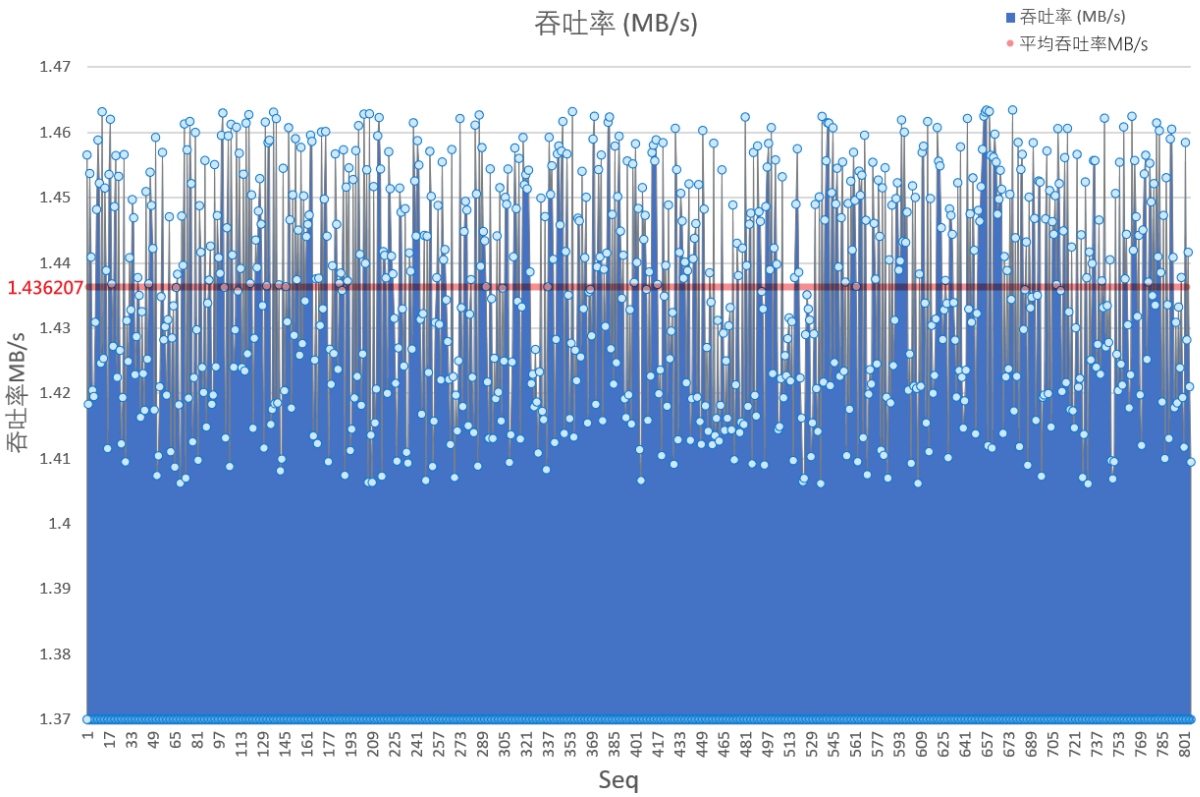


- 图中有多个传输时间的**峰值**，这些峰值表明在某些传输过程中，传输时间显著增加。例如，在 Seq（序列）值大约为 200、400 和 600 的位置，传输时间达到了 1.43 秒左右，这是图中的最高传输时间。这些峰值可能是由于网络拥塞、系统资源紧张或其他外部因素导致的。
- 图中也有一些传输时间的**谷值**，这些谷值表明在某些传输过程中，传输时间相对较短。例如，在 Seq 值大约为 300 和 500 的位置，传输时间接近 1.34 秒，这是图中的最低传输时间。这些谷值可能是由于网络状况较好、系统资源充足等因素导致的。

3.2 吞吐率

吞吐率曲线有一定波动。这种波动在实际网络环境中是常见的，可能受到以下几个因素的影响：

- **网络延迟**：可能因为某些数据包的传输时间较长，导致吞吐率出现短暂的下降。
- **丢包重传**：如果某些数据包丢失或出现错误，可能导致重传，重传过程通常比首次传输需要更多时间，影响吞吐率。
- **带宽抖动**：如果网络带宽在某些时刻发生变化，吞吐率也会随之波动。带宽的抖动可能是由网络拥塞、路由器负载变化等因素引起的。



能看到吞吐率随着传输过程逐渐波动，但整体上不会有很大的起伏。说明**网络带宽基本稳定**，偶尔会因为上面提到的因素而出现轻微的波动。

吞吐率的低点通常意味着传输过程中发生了较大的延迟或丢包现象。吞吐率的高点表明网络能够在短时间内有效地传输数据，这通常是在带宽充足、没有丢包或延迟抖动的情况下发生的。

平均吞吐率是一个重要的性能指标，它决定了总体传输过程中的效率。假如图中的**波动不大**，说明整体的吞吐率较为稳定，网络传输效率较高。

六、实验总结

本次实验主要涉及了基于UDP协议的可靠数据传输的实现，并通过模拟TCP的四次握手和挥手过程来完成连接的建立与断开。通过实验，我深刻理解了协议栈中的连接管理机制，尤其是TCP连接的三次握手和四次挥手过程，虽然实验中使用的是UDP协议，但模拟的这些机制在保证数据可靠传输中的重要性不可忽视。

1. **数据传输的可靠性：**

   本实验中的主要目标是通过UDP协议实现可靠的连接和数据传输。UDP协议本身不保证数据的可靠性，因此需要通过手动处理丢包、重传、校验和等机制来确保数据传输的可靠性。

   在握手和挥手过程中，我学会了如何利用校验和、序列号等手段来确保数据的完整性和顺序性。特别是在第二次握手、第三次握手和挥手过程中，数据的完整性校验和状态的变化是实现可靠传输的关键。

2. **三次握手与四次挥手的实现：**

   通过这次实验，我更加熟悉了TCP协议中的三次握手和四次挥手过程。在实验中模拟的每一步都涉及到通过发送特定的数据包进行确认，且每次的发送和接收都需要校验和、序列号等验证机制来保证数据传输的可靠性。

   通过不断检查每次握手和挥手的状态，确保数据包正确接收并根据状态做出相应的动作，这是实现连接管理的核心。

3. **非阻塞模式和超时机制：**

在实验中，我使用了非阻塞模式（`FIONBIO`）来避免在数据发送或接收时出现阻塞。通过超时机制（`RETIME`）控制重传的次数和时间，确保在发生网络延迟或数据丢失时能够重新发送数据包，从而增强连接的可靠性。

这种超时和重传机制对我来说是一个非常重要的学习点，它帮助我理解了如何在不阻塞主线程的情况下实现数据传输，并且能有效应对网络抖动和丢包问题。

不足之处：

1. 性能优化不足：
    - 在当前的实验中，使用了简单的循环和计时方法来实现重传机制，尽管保证了可靠性，但在性能方面有一定的瓶颈。例如，频繁的内存分配和缓冲区操作会增加程序的运行开销，影响效率。尤其是在高频率的数据传输中，可能会造成性能问题。
    - 如果需要处理大规模的数据传输，应该考虑对内存的优化管理，例如使用缓冲池和优化数据结构，减少内存分配的开销。
2. 不够灵活的超时重传机制：
    - 在实验中，超时重传的时间（`RETIME`）是固定的，没有根据实际情况进行动态调整。在实际网络中，网络状况是不稳定的，可能会出现延迟波动。若超时重传的时间可以根据网络延迟进行动态调整，将使得传输更加智能和高效。