计算机网络实验报告

Lab3.1 可靠数据传输代码实现

2112338 李威远 计算机学院 计算机卓越班

一、协议设计：

（一）实验要求

回顾我们的实验要求，本次实验中，我们需要实现基于UDP数据报套接字，运用socket编程，实现如下四个功能：

1. **稳定连接：**类似于TCP的握手、挥手功能来建立连接、断开连接
2. **差错检验：**使用校验和之类的方式，对可能出现的部分位出错进行检验
3. **接收确认：**对发送端的每个数据报，需要回复一个ACK数据报进行确认
4. **超时重传：**若一段时间收不到ACK，发送端需要重传。

此外，需要注意只需要实现客户端到服务器端的单向传输即可，要有收到、发送的数据包的**序号、ACK、校验和**等信息，还需要统计数据传输的**总传输时间和吞吐率。**

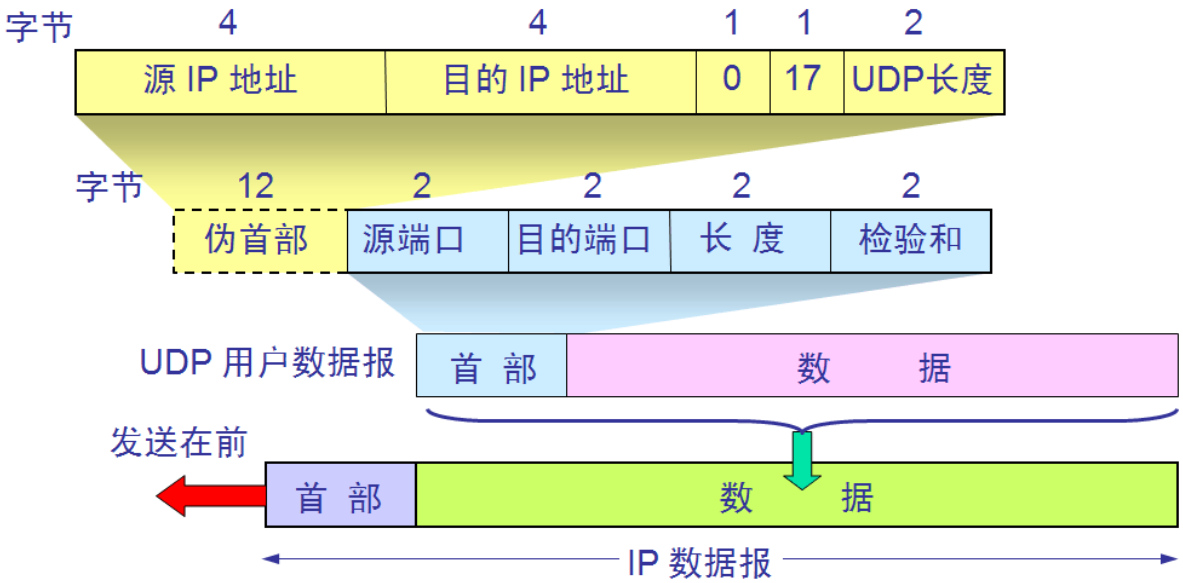
结合我们的课程所学，本次实验的任务其实十分明了，其中差错检验、接收确认、超时重传都是rdt3.0具有的功能，也就是需要我们实现一个拥有类似于TCP三次握手、四次挥手功能的、类似于rdt3.0协议的单向可靠数据传输程序即可，具体的实现可以效仿rdt3.0，也可以结合情况自己设计新的协议。

因此，我们首先从协议的设计开始，为我们的整个程序设计一个合适的报文结构，实现整个可靠数据传输程序。整个程序的具体代码实现已经上传到我的github当中

（二）设计思路

1、从UDP报文结构出发

本实验中，需要我们使用UDP套接字展开设计（虽然实际的报文结构不需要严格仿照UDP报文结构，但是可以作为参考）。因此，我们首先回顾UDP的报文结构，在它的基础上来增加我们所需的报文信息。



UDP报文结构中，含有如下的一些基础信息：

- **伪头部：** 只是为了提取 IP 数据报中的源IP，目的IP信息并加上协议等字段构造的数据。在实际传输中并不会发送，仅起到校验和计算使用，因此称之为伪首部。
- **源端口号：**一般是客户端程序请求时,由系统自动指定,端口号范围是 0 ~ 65535，0~ 1023为知名端口号。
- **目的端口：** 一般是服务器的端口，一般是由编写程序的程序员自己指定，这样客户端才能根据ip地址和 port 成功访问服务器

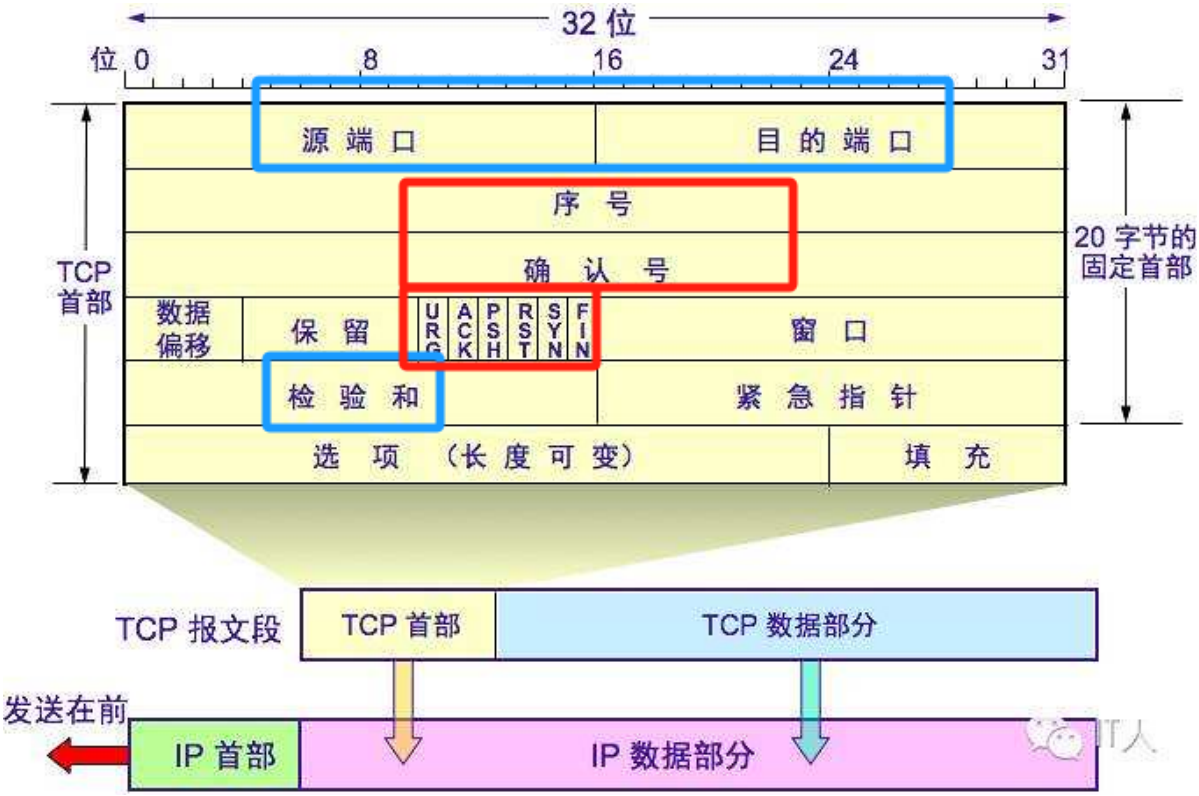
- **UDP长度：** 是指整个UDP数据报的长度，包括 报头 + 载荷
- **UDP校验和：** 用于检查数据在传输中是否出错，是否出现bit反转的问题，当进行校验时，需要在UDP数据报之前增加临时的伪首部。

其中，需要注意的是，UDP校验和这个结构，能够帮助我们完成数据错误情况下的**差错检验**功能的实现，当然还需要实现对应的维护结构，因此我们需要认识校验和机制的原理，我们放到后面再说。

考察以上的UDP结构后，我们认识到，仅仅有UDP这些结构，想要实现包含三次握手四次挥手的可靠稳定数据传输程序，是十分困难的。

2、结合TCP报文结构完善

因为我们需要实现三次握手四次挥手，所以一个直观的想法是回顾我们上次实验中详细考察的TCP三次握手和四次挥手中所必须的几个报文结构，将其补充到UDP报文结构中，即可实现我们所需的**建立连接**功能



上图为我们上次实验中已经熟悉的TCP报文结构，考察其内容，我们认识到，要想完整的复现TCP类似的三次握手和四次挥手，FIN、SYN、ACK这类标志位是不可缺少的（事实上，ACK标记位在数据传输中也有重要作用）。

此外，**序号Seq和确认号Ack**是否需要呢？这是我思考了很久的问题，最终还是将其加入了我们的报文结构，还增加了一个**标志位RDT_SEQ**，具体的原因需要和rdt3.0协议的一些特性相关，因此我们也放到后面的功能设计及其原理中讲解。

3、根据功能实现需求按需增加

但是，回顾我们先前的数据传输，比如上次实验中的**HTTP请求和响应**形式的双向数据传输，我们同样也需要一个标志来帮助服务器端判断客户端发来的数据报文是什么类型，是否是数据传输相关，因此，我又增加了一个**标记位FILE_TAG**，含该标记的数据报被我称为预告信息，用于告诉服务器端接下来需要进行数据传输，让服务器做好准备。

实际设计中，我还遇到了一个问题：**关于数据报的大小计算**，这个信息和Seq、Ack位的计算相关，由于涉及图片的传输计算，我的存储方式是**利用了char数组来进行逐字节的读入。**然后将其放进数据报的data段，传输给接收端。

这样会造成一个问题，我们在接收端难以获取char数组的长度，这是因为图片二进制文件按字节读入，很可能存在0数据。而常用的获取char数组方法即strlen()方法，该方法的原理即是读到0数据结束，这样就显然会造成读取长度的不正确。

因此，我还维护了一个size字段，该字段具有以下两个功能：

- **数据传输的正常情况下：**该字段存储数据报的大小，用于维护Seq和Ack字段的值，方便进行校验，保证传输正常。
- **此外，当出现FILE_TAG标记时：**该字段的意义改变，不再是整个数据报数据的大小，而是我们整个文件的大小，服务器端将根据这个字段的值来进行缓冲区的设置，为传输预留空间。

（三）协议展示

1、消息类型

在我们的协议设计中，不同的消息类型实际上是由标记位决定的，其内部的结构是统一的，作用会根据标记位的不同产生不同。结合三次握手和四次挥手、数据传输的整个过程，我们按照标记区分不同的消息类型，总共分为以下几种：

- **连接请求消息（SYN）：**由客户端向服务器端发送，即三次握手的第一次握手的消息，表明希望建立一个连接。
- **连接同意消息（SYN、ACK）：**由服务器向客户端发送，即三次握手的第二次握手的消息，表明同意建立这个连接。
- **断开请求信息（FIN）：**由客户端向服务器端发送，即四次挥手的第一次挥手的消息，表明希望断开连接。
- **断开同意信息（FIN、ACK）：**由服务器向客户端发送，即四次挥手的第三次挥手的消息，表明同意断开这个连接。
- **数据传输预告信息（FILE_TAG）:**由客户端向服务器端发送，即数据传输开始的消息，表明开始数据的传输。
- **数据传输信息（无）：**由客户端向服务器端发送，即数据传的数据报文，其中包含文件的数据信息
- **确认消息（ACK）：**双方都可能发送，用于确认对方的上一条数据报已经成功收到，且结果准确无误。

2、报文结构（语法及其语义）

基于以上设计，我们给出了如下的报文结构设计，所有的消息类型的结构都如下所示，但其标记位（flags）不同，功能也不同。

首先是UDP的伪头部，仿照IP头进行设计，增加了一个数据包长度字段。

```
#pragma pack(1)  //禁止优化存储结构进行对齐，保证结构体内部数据连续排列
struct IP_HEADER
{
    unsigned int SrcIp;     //源ip 32位
    unsigned int DestIp;     //目的ip 32位
    unsigned short TotalLenOfPacket;    //数据包长度 16位
};
#pragma pack()
```

其中，各字段在实验中用途较少，这里不多赘述。

然后是UDP头的设计，这是**整个报文结构设计的核心**，我们重点讲一下这部分：

```
#pragma pack(1)
struct UDP_HEADER
{
    unsigned short SrcPort, DestPort;//源端口号 16位、目的端口号 16位
    unsigned int Seq;//序列号 32位
    unsigned int Ack;//确认号 32位
    unsigned int size;//数据大小 32位
    char flag;//标志  8位
    char empty; //填充位，以保证对齐，无实际意义 8位
    unsigned short other; //16位，计数
    unsigned short checkNum;//校验和 16位
};
#pragma pack()
```

其中，各字段的功能（语义）在先前的思路设计中大部分都已经提到，这里补充说明一下增加的empty和other字段。

- **empty**：empty字段其实只是填充位，为了保证我们的报文结构是16位对齐，所以增加了这一段作为填充位，具体原因同样放到对校验和的原理介绍中说明。
- **other**：该字段的名称起的有误，事实上，我一开始用1字节的other字段（开始）作为填充位，然后意识到需要输出数据包号，因此就顺便将1字节的other字段作为数据包序号。但是发现这样**数据包号容量很小，超过127就会溢出**，因此我将其改为了一个16位的字段，用于记录数据包号，也就保留了"other"这个不太准确的名字，实际上应该是叫做package_num字段才对。

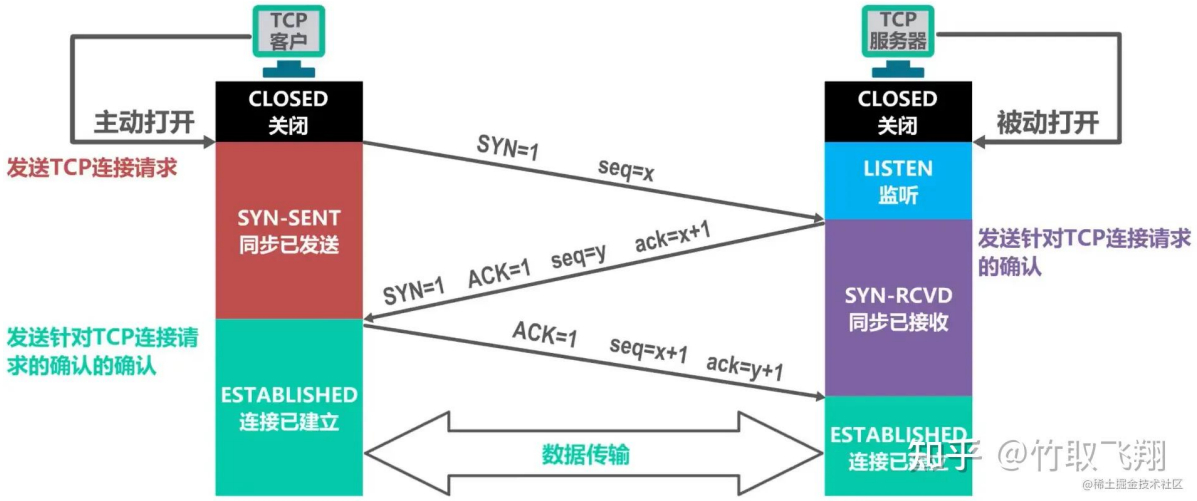
接下来，我们来看看最后的封装，也就是我们的数据包结构，它封装了我们刚刚给出的IP头和UDP头，以及一个char类型的报文数据，其中宏MaxMsgSize是指最大报文数据长度，用于后续的文件数据分组传输。

这里，我们对先前给出的字段内容都初始化为0。

```
#pragma pack(1)
struct DATA_PACKAGE
{
    IP_HEADER Ip_Header;
    UDP_HEADER Udp_Header;

    //报文数据
    char Data[MaxMsgSize];

    DATA_PACKAGE() {
        memset(&Ip_Header, 0, sizeof(Ip_Header));
        memset(&Udp_Header, 0, sizeof(Udp_Header));
        memset(&Data, 0, sizeof(Data));
    };
};
#pragma pack()
```

3、时序

对于三次握手和四次挥手的时序，可以参考我们上次实验中的三次握手和四次挥手的时序逻辑关系图，如三次挥手的过程如下，都已经在上次实验中详细介绍，不多赘述：



这里需要注意不同的是，由于我们是基于UDP数据报套接字，所以**上图中的服务器进入监听状态实质上在我们的实现中是不存在的**，客户端和服务器端都是如我们接下来展示的功能设计原理中的UDP套接字连接过程来建立连接，没有出现监听状态。

此外，还需要注意我们的Seq和Ack的变化，稍后会提到，我们这里按照TCP连接的方式维护了Ack和Seq两个字段，每次增加数据报的长度。因此，在三次握手和四次挥手中，我们也严格按照了上图进行Seq和Ack的维护。

如下是四次挥手的时序图，也已经在上次实验中详细介绍，这里同样不过多解释：

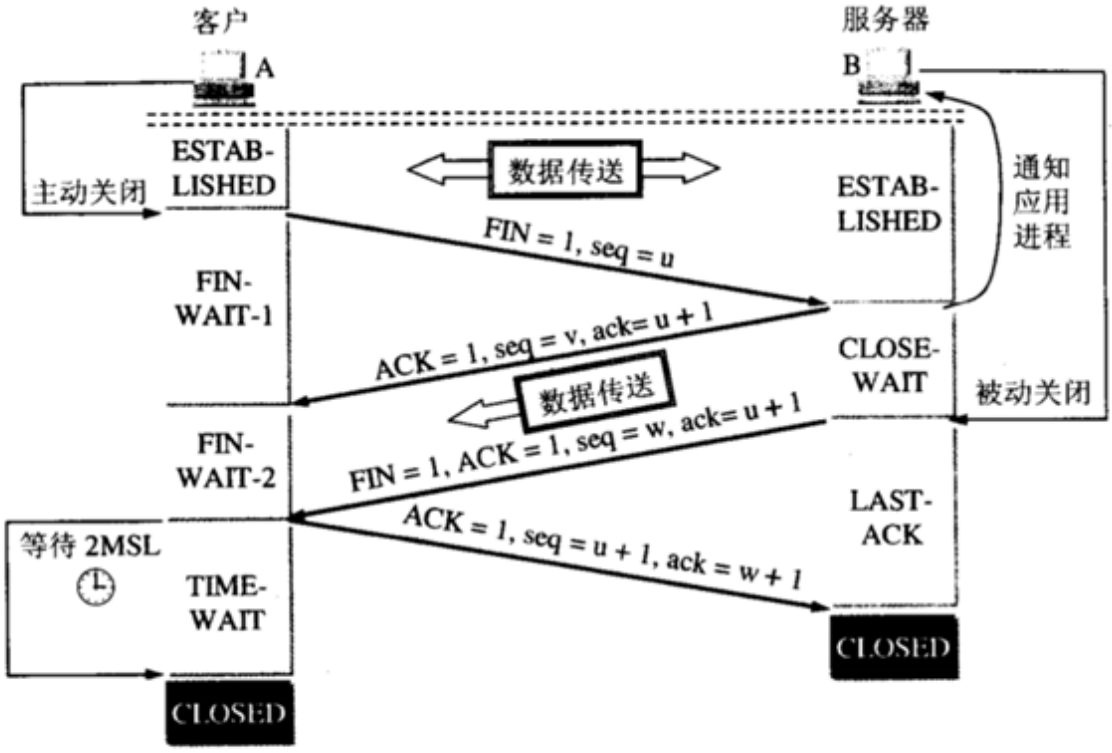
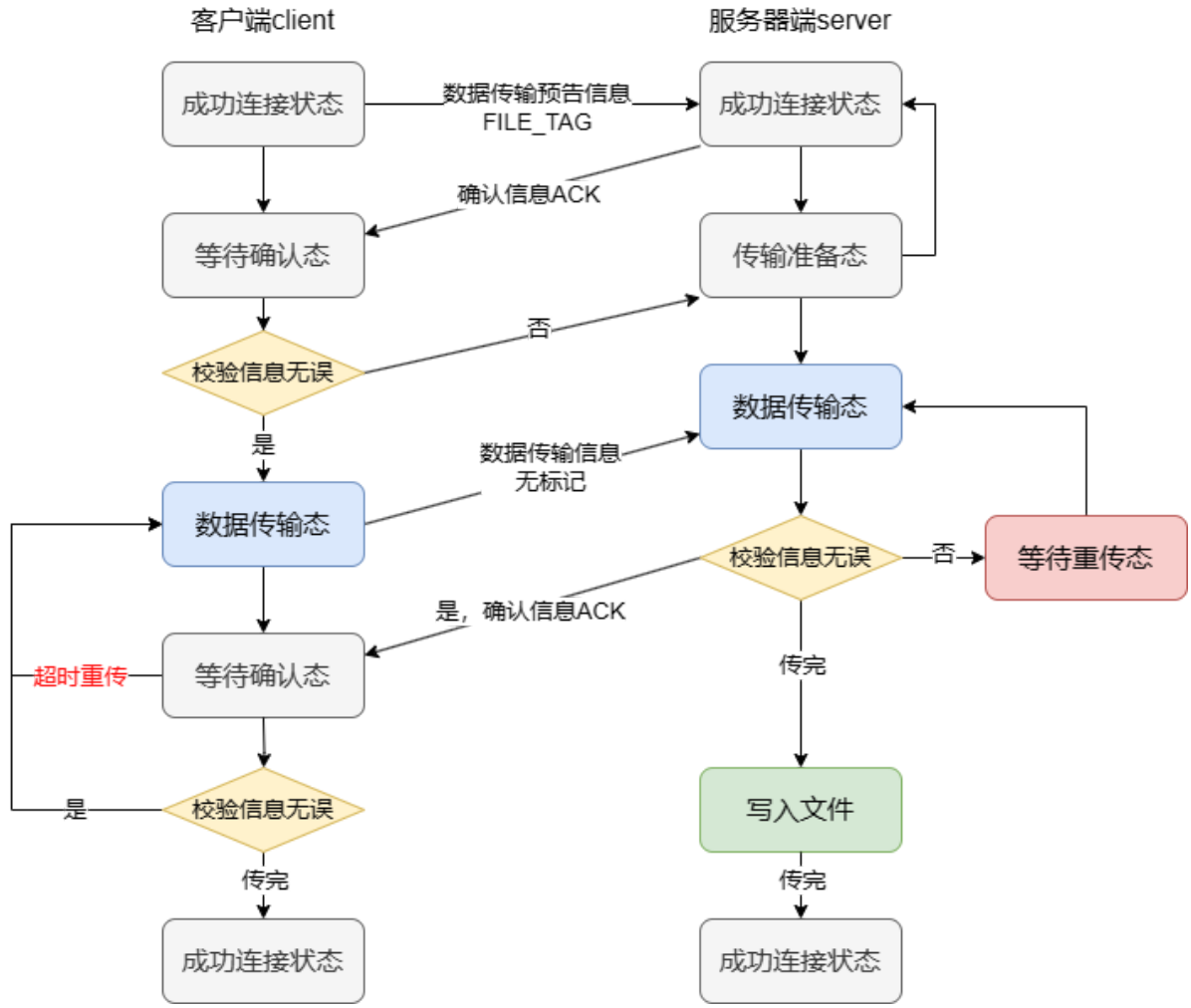


图 5-32　TCP 连接释放的过程

需要注意的是，事实上由于我们只是单向数据传输，且由发起数据传输的客户端主动提出四次挥手的断开，**即服务器端不需要向客户端发送任何的数据**。这样的话，我们的第二次挥手实际上丧失了它的意义，因为服务器不再需要等待自己传完数据，可以省略，但我们为了完整的模拟四次挥手，仍然保存了第二次挥手。

最后，如下图所示是建立连接后的数据传输的时序图，根据我的设计形成：



从上图可以看出，数据传输的过程中，我们通过之前定义的协议类型和特定功能实现（后续给出代码分析），完成了**超时重传、差错检验、接收确认**三项核心功能，接下来，我们来介绍这三个功能和先前的tcp三次握手四次挥手的**建立连接**这四个功能的具体实现思路和原理知识点，并给出对应的代码分析。

二、功能设计与原理

（一）从rdt协议思考功能实现

抛开类似于TCP的三次握手和四次挥手实现，剩下的**超时重传、差错检验、接收确认**三项核心功能其实都是rdt协议中逐步实现的过程，我们回顾rdt协议的发展，以此来确定我们需要实现什么具体功能：

1、经完全可靠信道的可靠数据传输：rdt 1.0

首先考虑最简单的情况，即底层信道是完全可靠的，我们称该协议为rdt 1.0，该协议非常简单，只需要实现发送端的send功能和接收端的recv功能即可实现。

2、经具有比特错误信道的可靠数据传输：rdt 2.0

假设底层**信道仅可能产生位错误**，即可能将某比特由 0 翻转为 1，由 1 翻转为 0，仍然假定所有发送的分组（虽然有些比特可能受损）将按其发送的顺序被接收。

这时候，我们需要实现以下功能，来对错误进行检验，并给予错误处理：

- **差错检测。**可以通过**校验和**来检测位错误。
- **接收确认。**接收方提供明确的反馈信息给发送方，rdt 2.0 协议将从接收方向发送方回送 ACK 与 NAK 分组，接收方通过 ACK 显示地告知发送方分组已正确接收，NAK 则显示地告知发送方分组有错误。

- **重传。**接收方收到有差错的分组时，发送方将重传该分组。

这里的差错检测和接收确认，正是我们所需实现的功能之一，不过我们没必要按照rdt2.0来进行设计，因为rdt协议后续还有改进。

3、经具有比特错误信道的可靠数据传输：rdt 2.1

rdt 2.0 存在一个致命的缺陷，即没有考虑 ACK / NAK 分组受损的可能性，可以给出以下的解决方法：

1. 可以为 ACK / NAK 增加校验和，检错并纠错，难度较高，且伴随着较高的代价；
2. 当发送方收到含糊不清的 ACK 或 NAK 分组时，只需重传当前数据分组即可。

然而，第二种方法可能会产生**重复分组。**重复分组的根本困难在于接收方不知道它上次所发送的 ACK 或 NAK 是否被发送方正确地收到，也就无法事先知道接收到的分组是新的还是一次重传。

解决重复分组问题的一个简单方法（几乎现有的数据传输协议，包括 TCP，都采用了这种方法）是在数据分组中添加**序列号（Sequence Number）**：发送方给每个分组增加序列号。接收方只需要检查序列号即可确定收到的分组是重传的还是新的分组。

可以看到，这里我们对**接收确认**进行了改进，增加了Seq序列号，使其更为可靠。

4、经具有比特错误信道的可靠数据传输：rdt 2.2

其实不需要使用 ACK 和 NAK 两种确认消息，只使用一种确认消息就可以实现 rdt 2.1 的功能，这就是 rdt 2.2，它删去了 NAK 消息协议。

接收方通过 ACK 告知最后一个被正确接收的分组序列号，这就需要在 ACK 确认消息中显示地加入被确认分组的序列号。发送方接收到对同一个分组的两个 ACK（即重复 ACK）后，就知道接收方没有正确接收到跟在被确认两次的分组后面的分组。

rdt 2.2 和 rdt 2.1 的细微变化在于，接收方此时必须包括**由一个 ACK 报文所确认的分组序号**，接收方此时**必须检查接收到的 ACK 报文中被确认的分组序号。**

可以看到，我们先前的协议设计中也没有包含NAK，这正是基于rdt2.2的认知。

5、经具有比特错误的丢包信道的可靠数据传输：rdt 3.0

现在底层信道还会丢包，假定发送方传输一个数据分组，该分组或者接收方返回对该分组的 ACK 发生了丢失，这样发送方都收不到应当到来的接收方的响应。

协议现在必须处理另外两个问题：怎样检测丢包以及发生丢包后该做什么。通过使用检验和、序列号、ACK 分组和重传等，我们能够给出第二个问题的答案，为了解决第一个问题，还需要增加一种新的协议机制：发送方等待“合理”时间：

- 如果没收到 ACK，发送方重传分组。
- 如果分组或 ACK 只是延迟而不是丢失，那么重传会产生重复分组，但 rdt 2.2 中**通过序列号机制已经能够处理重复分组**（需要在 ACK 中显示地加入上一个被正确接收的分组序列号）。

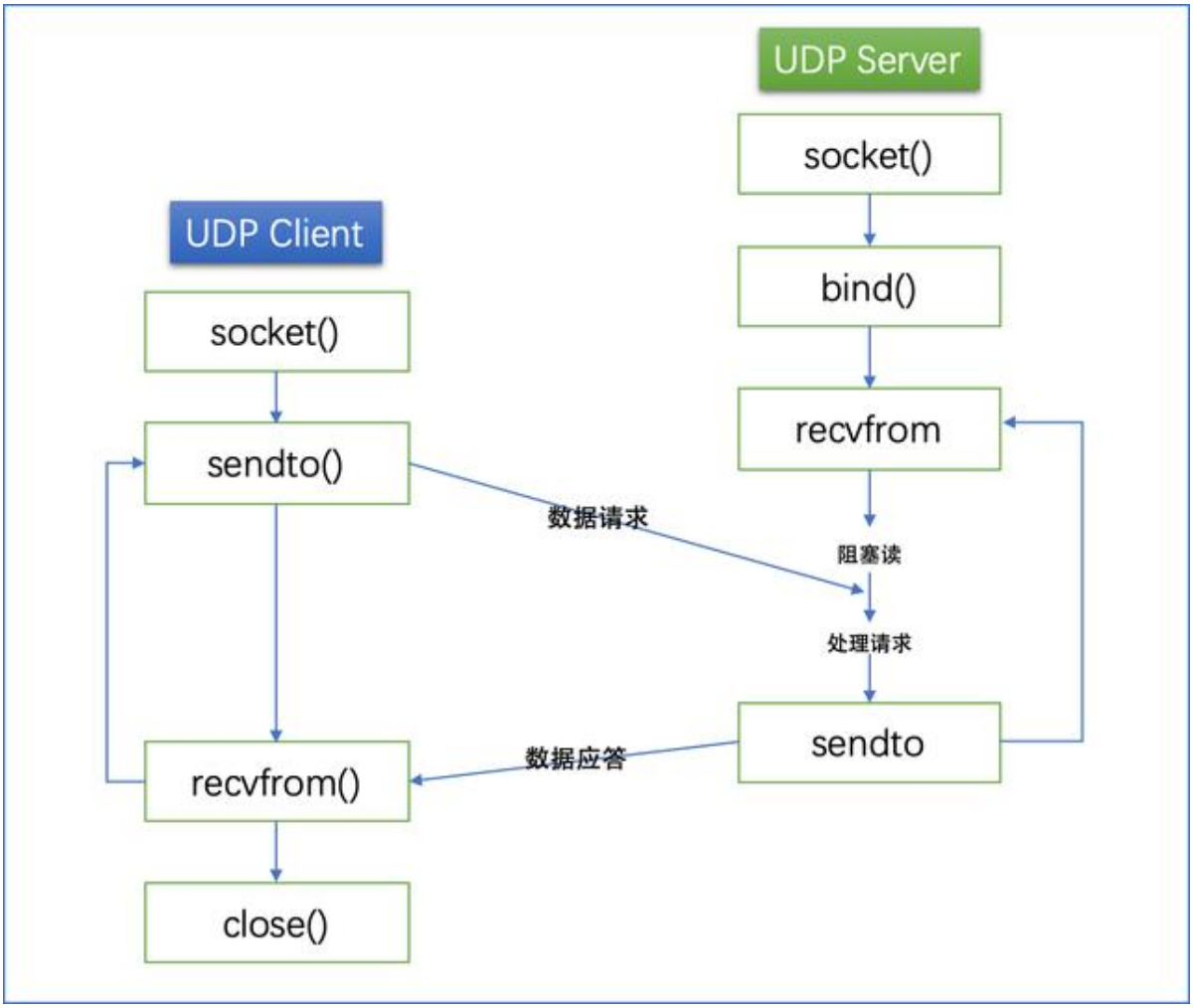
发送方不知道是一个数据分组丢失，还是一个 ACK 丢失，或者只是该分组或 ACK 过度延迟，在所有这些情况下，发送方的动作是同样的：重传。为了实现基于时间的重传机制，需要一个**倒计数定时器（countdown timer）**。

注意 rdt 3.0 发送方与 rdt 2.2 发送方有一个细微的差别，当发送方收到重复 ACK 时，**rdt 3.0 什么都不做**，直到定时器到时间后才会重传分组，而 rdt 2.2 的发送方收到重复 ACK 时会直接重传分组。

考察我们所需要实现的功能和rdt3.0，我们意识到，我们需要实现的恰恰是一个类似于rdt3.0的具有TCP三次握手、四次挥手的可靠数据传输程序，因此，我们可以**结合rdt3.0协议和TCP三次握手、四次挥手**的思路进行我们整个程序的设计。

（二）建立连接、断开连接

具体的功能设计原理上，建立连接、断开连接是**完全基于我们上次实验中的三次握手和四次挥手的形式。**但是事实上，我们是先基于UDP套接字进行实质上的连接，而三次握手和四次挥手实质上只是确认收发功能正常，并不是建立连接的操作，这部分操作需要类似第一次实验中的socket编程来实现，但是需要注意是UDP的形式，也就是如下图的连接：



可以看到，正如我们先前协议设计中提到，不同于TCP套接字的连接方式，**服务器端不需要进入监听态，然后对不同服务端来进行不同的connect，**UDP套接字的连接过程中，服务器和客户端经过的过程是类似的，都是经过如下几步：

1. 首先初始化Winsock库，建立自己的Socket，将其绑定到地址上。
2. 绑定对方的ip地址和进程端口号。
3. 利用sendto将各类消息发送到对方那边。
4. 利用recvfrom将对方的各类消息接收。

这是整个的实质性过程，但是我们在第三步的最开始，需要进行三次握手以确认连接的稳定性，并且在第四步的最后，进行四次挥手确保能够正常断开连接，随后再进行socket的关闭等操作。

（三）差错检验（校验和实现）

对于差错的校验，我们首先基于UDP校验和来进行实现，该方法能够确保高层的数据传输中，出现如”0变1“、”1变0“之类的异常错误能够及时检查到。此外，选用校验和而不是循环冗余校验法等方法的原因是：

冗余校验法主要采用除的计算方式，比累加的（校验和）计算方式效率要低，而数据链路层用这种方法可以用硬件实现，但是**网络层和传输层一般只能通过软件**实现，那么**效率就降低**了。

接下来，我们来看看校验和的原理：

检验和是存在于各种报文中的一个字段，它存在的目的是验证报文在网络传输过程中的完整性（有的数据可能在链路传输时发生0-1数据翻转，从而导致报文出错）。

因此，在报文的发送端，会根据报文中的数据来计算一个检验和，一旦接收端接受到相应报文，接收端也会对报文的首部或数据进行一次检验和计算，如果接收端算出来的检验和和发送端发送的不一样，那么接收端认为报文在传输过程中出了错，进行错误处理。

算法实现上：我们按照如下步骤实现:

客户端在发送数据时，为了计算数据报的校验和。应该按如下步骤：

1. 把IP数据报的校验和字段置为0；
2. 把首部看成以16位为单位的数字组成，依次进行二进制反码求和；
3. 把得到的结果存入校验和字段中。

服务端在接收数据时，计算数据报的校验和相对简单，按如下步骤：

1. 把首部看成以16位为单位的数字组成，依次进行二进制反码求和，包括校验和字段；
2. 检查计算出的校验和的结果是否等于零（反码应为16个1）；
3. 如果等于零，说明被整除，校验是和正确。否则，校验和就是错误的，协议栈要抛弃这个数据包，采取措施让对方重传。

这期间，计算和的过程中若发生溢出的现象，需要进行**回卷**（即把溢出的位加到末位上即可）。基于此算法实现，我们即可完成校验和的处理。事实上，基于差错检验的实现，我们完成了rdt1.0到rdt2.0的第一步处理，之后再增加接收确认等功能即可实现rdt2.0。

（四）接收确认

1、ACK报文实现接收确认

首先，根据我们在rdt2.0协议的认知，我们在收到数据报文后，需要回应一个ACK报文，这一点我们在TCP的三次握手中也有所体现，需要回复一个ACK报文来表示确定，因此实际上代码实现只需要在每次接收后，返回一个含有ACK标记的报文即可。

需要注意的是，因为我们这次实验是**基于停等机制**进行实现，所以需要保证收到ACK之前，客户端需要被阻塞卡住，直至收到ACK才能进行下一步的数据传输。

此外的实现思路都很简单，我们不多赘述，重点讨论一下序列号的维护。

2、序列号的维护：防止重复分组

在rdt2.1中，为了处理错误的ACK，我们需要进行重传等处理，这可能造成数据报文的**重复分组现象**，为了避免这种现象带来的影响，包括TCP在内的等协议都实现了序列号这一字段，来对数据报文进行检验，保证不会接收重复分组。

这里，我比较纠结的地方是，选取什么样的方法来设计我们序列号的检验规则。

实质上，对于我们的rdt协议，仅仅只需要一位序列号就能够实现检验，这是由于我们的实现实质上是**基于停等机制**，而对于停等协议这种简单的情况，1 比特序列号就足够了，因为它可以让接收方知道发送方是否正在重传前一个发送分组或是一个新分组。

这也是为什么rdt 3.0 有时被称为**比特交替协议（alternating-bit protocol）**。

但是，我个人的一个思考或者说纠结的点在于，我需要实现类似于TCP的三次握手和四次挥手，这期间为了更好地模拟，我对Seq和Ack进行了维护。既然如此的话，为什么不直接效仿TCP协议的序列号维护方式，对每次序列号增加发送的报文的大小，以此来进行校验呢？这样的话还会易于我们之后的几次滑动窗口的实验的实现。

俗话说，小孩子才做选择，我全都要。因此我对rdt的单个seq和TCP的Seq和ACK都进行了维护，也就是我先前提到的标志位RDT_SEQ等字段，在每次数据传输中，对这两类序列号都进行维护，保证不会出现重复分组等问题。

```
#define RDT_SEQ 128

char flag;//标志  8位 其中第八位是上面的RDT_SEQ，为RDT的序列号
unsigned int Seq;//TCP序列号 32位
unsigned int Ack;//TCP确认号 32位

//获取数据包的RDT_SEQ
int get_RdtSeq(DATA_PACKAGE& message) {
    return ((message.Udp_Header.flag & 0x80) >> 7);
}
```

（五）超时重传

超时重传的实现也十分简单，原理和原因在最一开始的从rdt协议思考功能实现的rdt3.0部分都已经详细介绍，这里我们通过对send后计时，若超时就进行重传来实现。

一个问题是，这里面临了多线程和非阻塞模式的二选一抉择，因为我们接收ACK的recvfrom方法默认是阻塞模式，如果没有接收到ACK就会一直卡住，也就没法进行超时处理的条判断。这里有两种思路进行解决：

- **多线程**：很简单，新建一个线程用recvform方法接收ACK，主线程用while卡住，在while中进行超时重传的检测处理，若接收到设置全局变量标志为正，结束循环
- **非阻塞模式**：强行设置非阻塞模式，这时候recvfrom没接收到ACK也会返回-1，需要用while管理，且我总感觉不太安全。

因此，我选择了多线程的方式解决这个问题。

三、核心功能代码分析

> **写在前面**：本次实验的代码量较大，我的client客户端写了547行，server服务器端写了468行，message报文头文件也有一百多行，报告肯定是难以展示全部的代码，全部的都已经在github中给出，就不再过多展示代码，简单列出要点。具体的数据传输部分代码因为不是核心功能，我就没有单独列出来作为一个模块，而是作为例子放入了各个核心功能实现的代码中。

在展开核心代码实现介绍前，先罗列一下我们基于UDP套接字进行连接的过程，下面代码为我们服务器端进行的设置，可见是基于UDP套接字进行设计的**（为了减少报告的代码量，我删去了一些错误处理和输出的代码，并非没有做）**。

```
    // 1、初始化Winsock库
    WSADATA wsadata;
```

```
    int res = WSAStartup(MAKEWORD(2, 2), &wsadata);

    // 2、创建服务器端的socket
    Server_Socket = socket(AF_INET, SOCK_DGRAM, 0);

    // 3、服务端绑定ip地址和进程端口号
    Server_Addr.sin_family = AF_INET;
    // 端口号设置
    Server_Addr.sin_port = htons(ServerPORT);
    // ip地址设置为我们的环回ip地址
    inet_pton(AF_INET, "127.0.0.1", &Server_Addr.sin_addr.S_un.S_addr);
    res = bind(Server_Socket,(SOCKADDR*)&Server_Addr,sizeof(SOCKADDR));

    //设置路由器的端口和地址号
    Router_Addr.sin_family = AF_INET; //地址类型
    Router_Addr.sin_port = htons(RouterPORT); //端口号
    inet_pton(AF_INET, "127.0.0.1", &Router_Addr.sin_addr.S_un.S_addr);

    //4、进行TCP类似的三次握手连接确认。
    bool ret = TCP_Connect();
```

（一）建立连接、断开连接

1、三次握手

（1）客户端的处理

在三次握手的具体实现上，代码量也是比较大，为了不使得报告过于长，我只展示了核心部分，具体的一些实现可以在github中找到：

第一次握手处理：首先，客户端会初始化第一次握手的报文段信息，这里我把所有的初始化都展示了，之后只展示核心的部分，如标志和Seq的处理等：

```
    //-----------第一次握手处理开始（SYN=1，seq=x）-----------
    //实际上应该随机分配一个seq，这里简化了
    message1.Udp_Header.SrcPort = ClientPORT;
    message1.Udp_Header.DestPort = RouterPORT;
    message1.Udp_Header.flag += SYN;//设置SYN
    message1.Udp_Header.Seq = now_seq++;//设置序号seq
    set_checkNum(message1);//设置校验和
```

这里的now_seq是初始化的全局变量，按照我们的TCP连接设置，实际上应该是随机的，不过这里我做了简化。

初始化的过程中，因为我们在构造函数中把所有字段置为零了，这里需要设置目标端口和源端口、必须的标志位、维护Seq和Ack和一些别的特殊信息等等，此外需要注意最后需要调用**设置校验和**的函数（之后说明），在三次握手和四次挥手中也注意**可靠稳定数据传输**！

随后，调用sendto进行数据传输，检查有没有发送，即完成客户端的第一次握手：

```
int send1 = sendto(Client_Socket, (char*)&message1, sizeof(message1), 0,
(sockaddr*)&Router_Addr, addr_len);

// 设置时钟，用于接收的超时重传
clock_t message1_start = clock();

//检查有没有成功发送
if (send1 == 0) return false;
cout << "客户端第一次握手成功发送！" << endl;
```

第二次握手处理：客户端的第二次握手是接收SYN、ACK的同意建立连接报文，这里需要注意我们超时重传的问题，还需要注意差错的校验。

这里实质上也是一种接收的确认，也需要对Seq进行确认，具体的一些操作和数据传输是类似的，我们先不展开讲，放到后面三种功能代码分析中来展示。

因此我们不展示了，一些相关处理到后面的功能分析中来说明，这里，我们通过超时和校验等操作，对第二次握手接收的报文进行检查，来看是否需要对第一次握手进行重传。

第三次握手处理：客户端第三次握手就比较简单，和第一次握手是一样的处理，只不过标记位改为ACK，同时需要对seq进行一点维护，我们只展示核心处理：

```
    message3.Udp_Header.flag += ACK;//设置ACK
    message3.Udp_Header.Seq = now_seq;//设置序号seq=x+1
    message3.Udp_Header.Ack = message2.Udp_Header.Seq + 1;
    now_ack = message3.Udp_Header.Ack;
    set_checkNum(message3);//设置校验和
    int send3 = sendto(Client_Socket, (char*)&message3, sizeof(message3), 0,
(sockaddr*)&Router_Addr, addr_len);
```

(2) 服务端的处理

实质上有些操作和客户端是雷同的，我们不讲太多，具体代码可以在github找到。

第一次握手处理：由于是第一次接收，不需要进行超时重传的处理，也不需要进行什么seq的检验，只需要检查校验和即可，这里需要注意的是，如果在这时候发生错误，需要不作回复，等待客户端重传：

```
//----------- 接收第一次握手的消息（SYN=1，seq=x）-----------
int recv1 = recvfrom(Server_Socket, (char*)&message1, sizeof(message1), 0,
(sockaddr*)&Router_Addr, &addr_len);
if (recv1 == 0) return false;
else if (recv1 > 0) {
    cout << "服务器端第一次握手成功收到！";
    if ((message1.Udp_Header.flag & SYN)) {
        if (!check(message1)) return false;
        cout << "且信息准确无误" << endl;
    }
    else return false;
}
else return false;
```

第二次握手处理：第二次握手时，服务器段需要维护服务器段的Seq和设置Ack，这里由于TCP的规定，设置增加一个序列号即可，此外的操作和客户端第一次握手几乎没有差别，也就是一个单一的发送消息操作，我们不过多展示：

```
message2.Udp_Header.Seq = now_seq;  //服务器回复的seq=服务器的seq+1
message2.Udp_Header.Ack = message1.Udp_Header.Seq+1;  //服务器回复的ack=客户端发来的
seq+1
now_ack = message2.Udp_Header.Ack;
message2.Udp_Header.flag += SYN;
message2.Udp_Header.flag += ACK;
set_checkNum(message2);//设置校验和
int send2 = sendto(Server_Socket, (char*)&message2, sizeof(DATA_PACKAGE), 0,
(sockaddr*)&Router_Addr, addr_len);
```

第三次握手处理：和客户端第二次握手处理的操作几乎完全一样，不再展示了，这里我其实本来很想做一个封装，写着写着写忘了，这些超时重传的检查处理其实本质上都是差不多的，我们不再多看。

第三次握手的检验的while结束后，TCP_connect函数就结束了，此时返回一个true，表明三次握手成功建立，接下来可以进行数据的传输。

这里，客户端会进入到main函数的一个while处理中，即选择断开连接还是传输数据，代码很简单不多展示，这里我们可以进行之后的一些选择处理。

```
    // 5、接收用户指令来选择传输或者中断
    while (1)
    {
        int choice;
        cout << "请输入您的操作：" << endl << "（终止连接--1      传输文件--0）" <<
endl;
        cin >> choice;
        if (choice == 1)break;
        else if (choice == 0) {
            string filename;
            cout << "请输入文件路径：" << endl;
            cin >> filename;
            if (!Data_Send(filename)) {
                cout << "数据传输出错！请重试" << endl << endl;
            }
        }
```

2、四次挥手

在看四次挥手的实现前，需要思考何时会进入四次挥手的处理，对于客户端，是用户输入了断开连接的指令，那么对于服务器端呢？

我在main函数中增加了对消息类型的检查，如果是FIN类型（表明客户端开始断开连接了）会中断循环，进行断开连接的四次挥手，代码如下所示，其中传入参数nowMessage是为了防止重传的发送：

```
    // 5、服务器端处理消息，根据类型做不同处理
    DATA_PACKAGE NowMessage;
    while (1) {
        int recv = recvfrom(Server_Socket, (char*)&NowMessage,
sizeof(NowMessage), 0, (sockaddr*)&Router_Addr, &addr_len);
        if (recv == 0) {
            cout << "数据传输:接收消息失败!" << endl;
```

```
        }
        else if (recv > 0) {
            //先看看是不是FIN标志，如果是进入到关闭TCP函数中处理
            if (NowMessage.Udp_Header.flag & FIN) break;
            else if(NowMessage.Udp_Header.flag & FILE_TAG) {
                int ret = Data_Recv(NowMessage);
                if (ret == false) {
                    cout << "数据传输:传输过程失败!" << endl;
                    exit(EXIT_FAILURE);
                }
            }
        }
    }
    ret = Close_TCP_Connect(NowMessage);
    if (ret == false) {
        cout << "四次挥手:服务器响应失败!" << endl;
        exit(EXIT_FAILURE);
    }
```

四次挥手实质上的代码实现和三次握手也是十分类似的，我们不再过多展示，只说明其中哪些是类似的，有哪些不同：

其中，第一次挥手和第一次握手的处理基本相同、第二次握手和第二次挥手基本相同，只是标记位的处理和检查不同，我们从第三次挥手开始看：

在第二次挥手之后，服务端会检查自己的数据有没有传输完，实质上这里服务器端没有数据传输，会立刻进行第三次挥手，这里客户端不再需要等待超时处理，因为之前的第二次挥手中已经确保了第一次挥手成功发送：

```
//-----------第三次挥手处理开始（FIN=1，ACK=1，seq=w，ack=u+1）-----------
recv_mark = recvfrom(Client_Socket, (char*)&message3, sizeof(message3), 0,
(sockaddr*)&Router_Addr, &addr_len);
//不为负1的时候表明收到了消息
if (recv_mark == 0) return false;
//成功收到消息
else if (recv_mark > 0) {
    //检查标记位和序列号ack，保证可靠性传输
    cout << "客户端第三次挥手成功收到！";
    if ((message3.Udp_Header.flag & ACK) && (message3.Udp_Header.flag & FIN)&&
(message3.Udp_Header.Ack == message1.Udp_Header.Seq + 1)) {
        if (!check(message3)) return false;
        cout << "且信息准确无误" << endl;
    }
    else return false;
}
recv_mark = -1;
```

随后，第四次挥手实际上和第三次握手也是类似的，但是有所不同的是我增加了等待2MSL的处理，这期间服务器端如果一直没有收到ACK，又向客户端发送信息（实质上该是NAK），客户端就会触发重传，也就是如下的代码实现：

```
//创建最后的接收线程线程
HANDLE wait = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ReceiveThread, &tmp,
0, 0);
```

```
while (clock() - close_clock < 2 * MaxWaitTime)
{
    if (recv_mark == 0) return false;
    else if (recv_mark > 0)
    {
        //丢失的ack
        send4 = sendto(Client_Socket, (char*)&message4, sizeof(message4), 0,
(sockaddr*)&Router_Addr, addr_len);
        cout << "异常恢复：重新发送结束ACK" << endl;
    }
}

//回收线程
TerminateThread(wait, EXIT_FAILURE);
```

基于以上处理，我们成功实现了三次握手和四次挥手，二者的实现逻辑实际上和后续的数据传输也差不多，都是基于之后的差错检验、接收确认和超时重传的几个功能来实现的。

（二）差错检验

对于差错检验的实现，其实就是基于先前给出的算法，设计两个接口，在每次接收数据报的时候增加check的调用、在每次发送数据的时候增加set_check的调用即可。两个函数的实现分别如下：

```
void set_checkNum(DATA_PACKAGE& message) {
    int sum = 0;
    // 用short指针划分，保证按照两个字节两个字节的读取
    unsigned short* msg = (unsigned short*)&message;

    // 注意将checkNum置为0，方便求和
    message.Udp_Header.checkNum = 0;
    // 计算和
    for (int i = 0; i < sizeof(message) / 2; i++) {
        sum += *msg;
        msg++;
        // 判断是否存在高16位，否则将高位加到低位上
        if (sum & 0xFFFF0000) {
            sum &= 0xFFFF;
            sum++;
        }
    }
    // 取反
    message.Udp_Header.checkNum = ~(sum);
}
```

```
bool check(DATA_PACKAGE& message) {
    int sum = 0;
    unsigned short* msg = (unsigned short*)&message;
    for (int i = 0; i < sizeof(message) / 2; i++) {
        sum += *msg;
        msg++;
        // 判断是否存在高16位，否则将高位加到低位上
        if (sum & 0xFFFF0000) {
            sum &= 0xFFFF;
```

```
            sum++;
        }
    }

    if ((sum & 0xFFFF) == 0xFFFF) {
        return true;
    }
    return false;
}
```

这里，我们通过一个**unsigned short*类型**的msg指针（其每次指向两个字节的字段），来遍历message这个报文段结构体，由于我们先前的设计，所有的结构体都是**16位对齐**的，且用**#pragma pack(1)宏**保证了其**不会出现编译器的优化。**

基于此，我们累加各个16位字段，并对其出现溢出的情况进行回卷处理，set_check在最后将值取反增加到checkNum字段上（先把这个字段置为0了，保证不累加它），这样的话就能保证**如果不出现数据错误的话，**能够在check检验中所有字段的累加和为0xFFFF，也就实现了差错的校验。

我们在每次接收到数据和发送数据前，都会调用这两个函数进行校验和字段的处理，具体的一些调用可以在之前的实现看到。

（三）接收确认

1、ACK确认报文的处理

接下来，我们还需要做好接收确认的处理，首先是ACK确认报文的实现，如下代码所示，**即为我后续封装的在数据传输过程中应用的returnACK，该函数用于服务器端对传入的数据报文进回应**：

```
bool returnACK(DATA_PACKAGE& now_Message) {
    //设置一些基本信息
    ……
    nameMessage.Udp_Header.flag |= ACK;
    nameMessage.Udp_Header.Seq = now_seq;
    nameMessage.Udp_Header.Ack = now_Message.Udp_Header.Seq +
now_Message.Udp_Header.size;
    now_seq++;
    //完成数据包的构成后不能忘了设置校验和！！！
    set_checkNum(nameMessage);

    int send0 = sendto(Server_Socket, (char*)&nameMessage, sizeof(nameMessage),
0, (sockaddr*)&Router_Addr, addr_len);
    //检查有没有成功发送
    if (send0 == 0) return false

    //剩下的其实都是一些输出信息，为了减少代码量就不展示了，可以在github找到
    ……
}
```

可以看出，这个returnACK函数的逻辑和先前的握手返回ACK其实是相似的，不多赘述了，我们会在每次接收完一个数据包，确认信息无误，不需要等待重传，并写入到缓冲区后调用该函数，完成对数据报文的确认。

2、Seq序列号的处理

这里，我们首先以先前没有展示的第二次握手的处理中检验的处理为例，这里，通过按位与的方式来检查标记位是否符合预期，然后会根据收到报文的Ack是否和下一次的Seq相同来进行预测，实质上应该与now_seq比较，但这里还没有更新now_seq。

```
if ((message2.Udp_Header.flag & ACK) && (message2.Udp_Header.flag & SYN)  &&
(message2.Udp_Header.Ack == message1.Udp_Header.Seq+1)){
    if (!check(message2)) return false;
    cout << "且信息准确无误" << endl;
    break;
}
else return false;
```

然后我们来看看实际数据传输中是如何处理Seq序列号的，这里我启用了RDT_SEQ的处理，同时还对之前的returnACK函数进行了调用：

```
                cout << "服务器端成功收到" << int(Data_message.Udp_Header.other) <<
"号数据包！";
                //检查标记位和序列号ack，保证可靠性传输
                if ((rdt_seq == get_RdtSeq(Data_message)) &&
(Data_message.Udp_Header.Ack == now_seq)) {
                    if (!check(Data_message)) return false;
                    ……一些输出信息
                    //返回确认信息
                    returnACK(Data_message);
                    rdt_seq = !rdt_seq;
                    for (int j = 0; j < left_size; j++)
                    {
                        file_message[package_num * MaxMsgSize + j] =
Data_message.Data[j];
                    }
                    break;
```

再来看看TCP的**Seq和Ack，以及RDT的seq是怎么维护的吧**：

服务端是如此维护TCP的Seq和Ack的，即Seq自增1用于记录这是第几个数据包（是我的个人规定，实际上因为服务端没有发送数据所以没有Seq自增），而Ack通过计算大小来计算预测客户端的Seq：

```
nameMessage.Udp_Header.Seq = now_seq;
nameMessage.Udp_Header.Ack = now_Message.Udp_Header.Seq +
now_Message.Udp_Header.size;
now_seq++;
```

客户端也是类似的：

```
Data_message.Udp_Header.Seq = now_seq;
Data_message.Udp_Header.Ack = now_ack;
now_seq += (*message).Udp_Header.size;
now_ack++;
```

此外，RDT的Seq的维护，不管服务器端还是客户端，只需要每次取反即可：

```
if (rdt_seq == 0) {
    Data_message.Udp_Header.flag += RDT_SEQ;
    rdt_seq = 1;
}
else rdt_seq = 0;
```

（四）超时重传

最后是超时重传的处理，之前已经提到，我选用了新建一个线程的方式来进行超时重传的处理，原因也已经解释过，这里还是以我们刚刚没有给出的第二次握手为例，给出我们的超时重传的一个实现，其他地方的超时重传处理都是同样的逻辑：

```
DWORD WINAPI ReceiveThread(LPVOID pParam) {
    DATA_PACKAGE* message = (DATA_PACKAGE*)pParam;
    recv_mark = recvfrom(Client_Socket, (char*)message, sizeof(*message), 0,
(sockaddr*)&Router_Addr,&addr_len);
    return 0;
}

//-----------第二次挥手处理开始（ACK=1，seq =v ，ack=u+1）-----------
//建立新线程，用于监听有无收到第二次握手，主线程用于超时重传处理
CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ReceiveThread, &message2, 0, 0);

//检验和超时重传
while (1) {
    //不为负1的时候表明收到了消息
    if (recv_mark == 0) return false;
    //成功收到消息
    else if (recv_mark > 0) {
        //检查标记位和序列号ack，保证可靠性传输
        ……刚刚已经给出了
    }
    //第一次握手的message1超时，准备重传
    if (clock() - message1_start > MaxWaitTime)
    {
        cout << "第一次挥手超时，准备重传......" << endl;
        send1 = sendto(Client_Socket, (char*)&message1, sizeof(message1), 0,
(sockaddr*)&Router_Addr, addr_len);
        message1_start = clock();
        if (send1 == 0) return false;
    }
}
recv_mark = -1;
```

可以看到，多线程处理的超时重传，实际上只是分为了**接收和计时**两个线程，当接收完毕后，会修改全局变量recv_mark为-1，随后再进行消息的校验等操作，如果校验正常，就可以结束循环，进行之后的操作；如果校验不正常，就还需要等待超时进行重传，以获取正确的数据内容。

实质上，这种多线程在逻辑上**还是停等机制的实现**，用while阻塞了数据的传输继续进行，跟采用非阻塞模式进行的单线程处理没有任何区别。

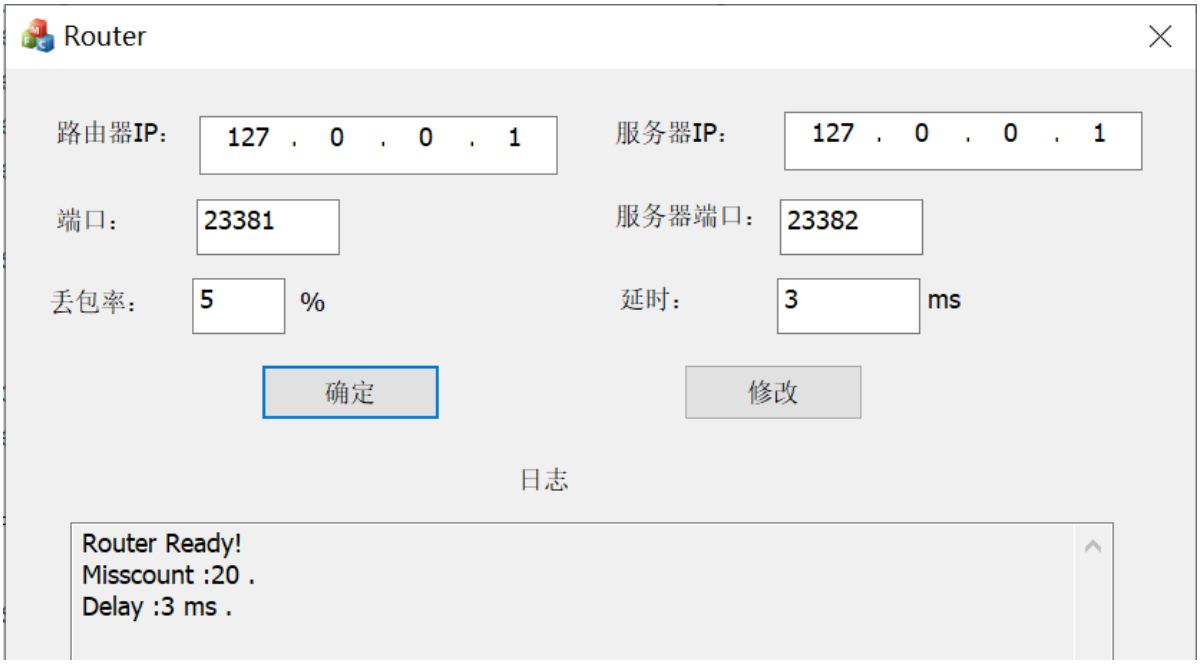
基于以上功能的实现，我们正确保障了四个核心功能的设计。一些基本的操作，如怎么进行的数据传输，这里就不做多介绍了，实质上和先前介绍中的一些代码都是大同小异的。

四、运行截图与传输结果分析

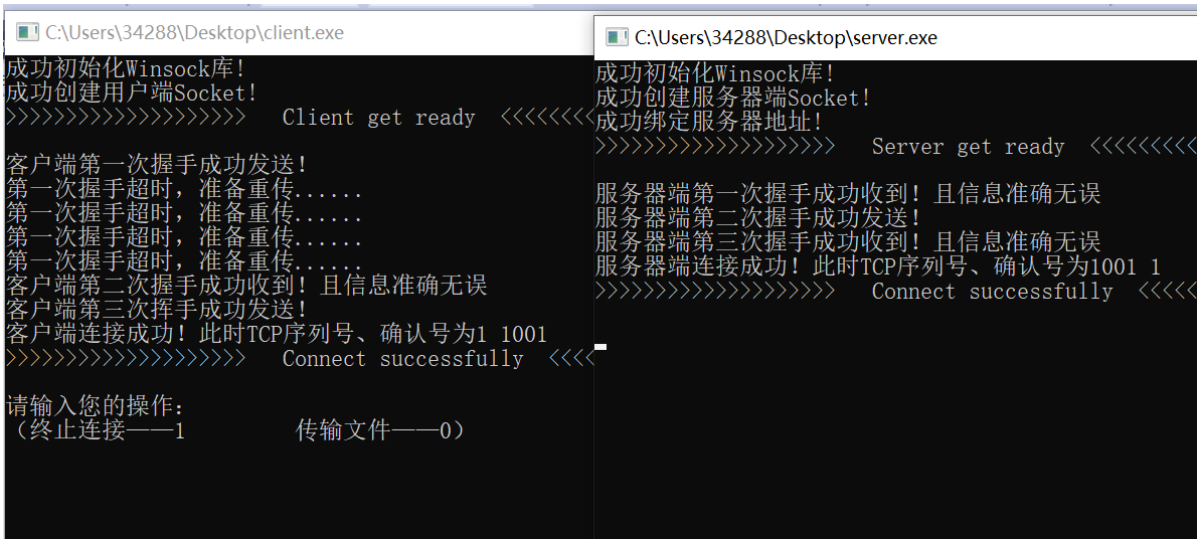
接下来，我们来看看正确运行的截图，和传输结果的分析：

（一）运行截图

首先我们先进行路由器的设置，设置信息如下：



这里，我们故意先运行客户端，再运行服务器端，看看三次握手的重传能不能正常实现，我们发现客户端可以正常重传，且能够正常输出序列号确认号信息：



选择我们要传输文件，并设置传输文件的地址，随后在服务器端设置写入文件的地址，为桌面上的recv文件夹，先来看看客户端运行的结果：

```
C:\Users\34288\Desktop\client.exe
客户端成功收到129号数据包的ACK！且信息准确无误
其中：RDT确认号为:0,校验和为(对为1错为0):1,TCP序列号为:1130,TCP确认号为:1806006

客户端成功发送130号数据包！
其中：RDT序列号为:0,校验和为(对为1错为0):1,TCP序列号为:1806006,TCP确认号为:1131
客户端成功收到130号数据包的ACK！且信息准确无误
其中：RDT确认号为:1,校验和为(对为1错为0):1,TCP序列号为:1131,TCP确认号为:1820006

客户端成功发送131号数据包！
其中：RDT序列号为:1,校验和为(对为1错为0):1,TCP序列号为:1820006,TCP确认号为:1132
客户端成功收到131号数据包的ACK！且信息准确无误
其中：RDT确认号为:0,校验和为(对为1错为0):1,TCP序列号为:1132,TCP确认号为:1834006

客户端成功发送132号数据包！
其中：RDT序列号为:0,校验和为(对为1错为0):1,TCP序列号为:1834006,TCP确认号为:1133
客户端成功收到132号数据包的ACK！且信息准确无误
其中：RDT确认号为:1,校验和为(对为1错为0):1,TCP序列号为:1133,TCP确认号为:1848006

客户端成功发送133号数据包！
其中：RDT序列号为:1,校验和为(对为1错为0):1,TCP序列号为:1848006,TCP确认号为:1134
客户端成功收到133号数据包的ACK！且信息准确无误
其中：RDT确认号为:0,校验和为(对为1错为0):1,TCP序列号为:1134,TCP确认号为:1857359

>>>>>>>>>>>>>>>>>>>>   Data transmission successfully   <<<<<<<<<<<<<<<<<<<<
总体传输时间为:25s
吞吐率:74294.1byte/s

请输入您的操作：
（终止连接——1          传输文件——0）
```

这里，我们正确输出了每次数据包传输的详细信息，包含数据包号、RDT序列号、TCP序列号、TCP确认号、ACK、校验和等等信息，此外，在最后还输出了总体的传输时间和吞吐率。

对于超时重传，我们也有相应的输出信息：

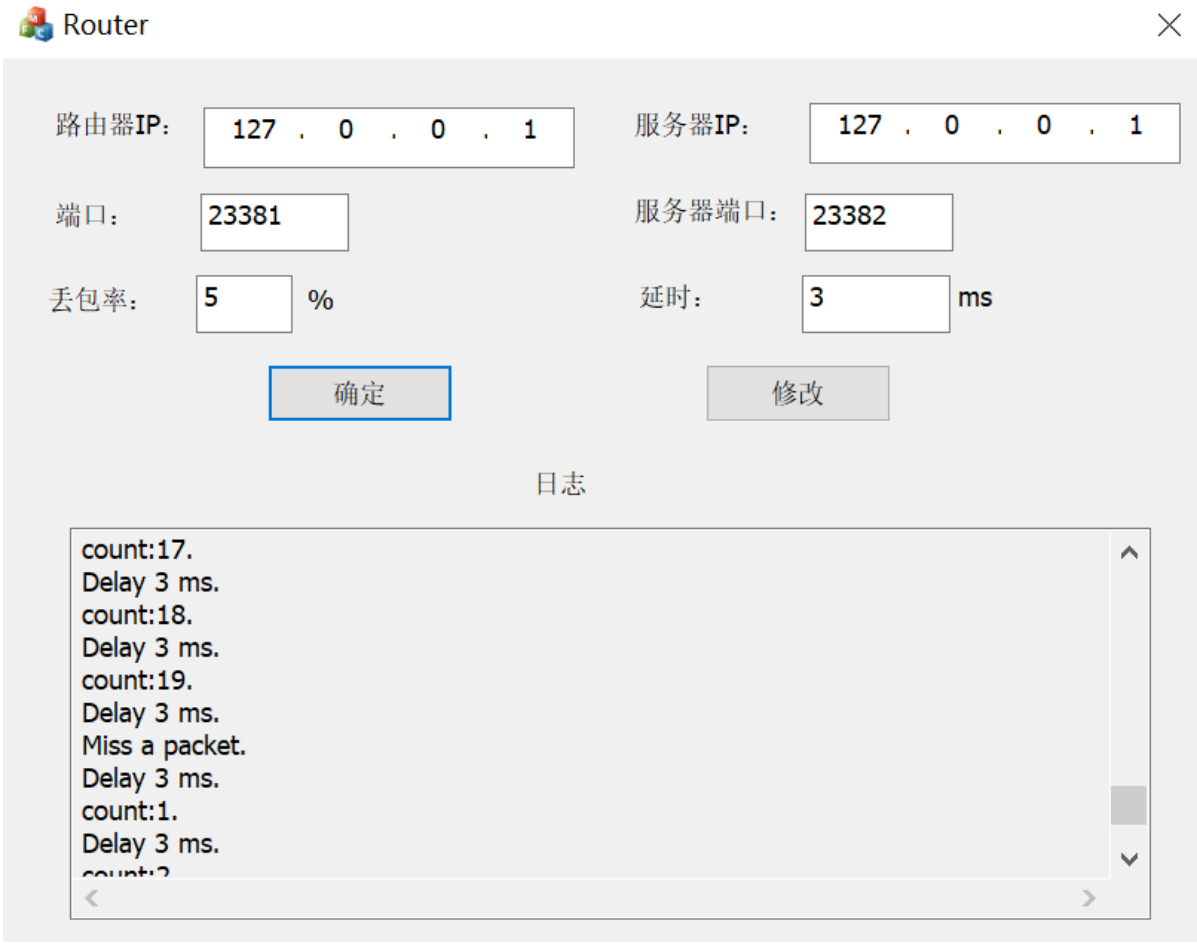
```
客户端成功发送106号数据包！
其中：RDT序列号为:0,校验和为(对为1错为0):1,TCP序列号为:1470006,TCP确认号为:1107
106号数据包超时0，准备重传.....
客户端成功收到106号数据包的ACK！且信息准确无误
其中：RDT确认号为:1,校验和为(对为1错为0):1,TCP序列号为:1107,TCP确认号为:1484006
```

再考察我们的服务器端输出结果，上述的一些数据包信息是同样能够输出处理的。此外，可以看到对于0号数据包预告信息，服务器端能够顺利解析，能够获取文件名、文件大小，并对此进行分组接收。

而且其中由于用户输入存储地址会延迟，导致0号数据包重传，我们能够看到，程序是可以正常处理忽视该重复分组的：

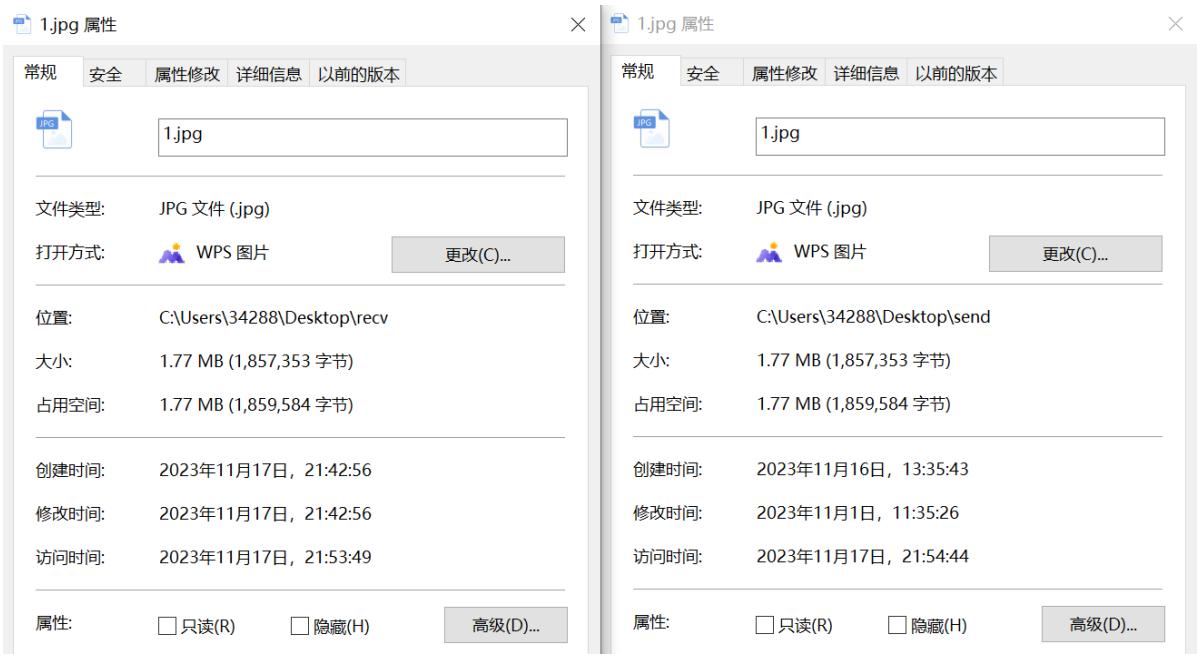
```
选择 C:\Users\34288\Desktop\server.exe
>>>>>>>>>>>>>>>>>>>>   Connect successfully   <<<<<<<<<<<<<<<<<<<<

服务器端0号数据包成功收到！且信息准确无误
其中：RDT序列号为:0,校验和为(对为1错为0):1,TCP序列号为:1,TCP确认号为:1001
收到新文件！文件名为：1.jpg，大小为：1857353
请输入文件存储目录：
C:\Users\34288\Desktop\recv\
服务器端成功发送0号数据包的ACK！
其中：RDT确认号为:1,校验和为(对为1错为0):1,TCP序列号为:1001,TCP确认号为:6

开始接收数据段，共 133 个报文段
132服务器端成功收到0号数据包！出现【重复接收报文段】！
其中：RDT序列号为:0,校验和为(对为1错为0):1,TCP序列号为:1,TCP确认号为:1001
服务器端成功收到0号数据包！出现【重复接收报文段】！
其中：RDT序列号为:0,校验和为(对为1错为0):1,TCP序列号为:1,TCP确认号为:1001
服务器端成功收到0号数据包！出现【重复接收报文段】！
其中：RDT序列号为:0,校验和为(对为1错为0):1,TCP序列号为:1,TCP确认号为:1001
服务器端成功收到1号数据包！且信息准确无误
其中：RDT序列号为:1,校验和为(对为1错为0):1,TCP序列号为:6,TCP确认号为:1002
服务器端成功发送1号数据包的ACK！
其中：RDT确认号为:0,校验和为(对为1错为0):1,TCP序列号为:1002,TCP确认号为:14006

服务器端成功收到2号数据包！且信息准确无误
其中：RDT序列号为:0,校验和为(对为1错为0):1,TCP序列号为:14006,TCP确认号为:1003
服务器端成功发送2号数据包的ACK！
其中：RDT确认号为:1,校验和为(对为1错为0):1,TCP序列号为:1003,TCP确认号为:28006

服务器端成功收到3号数据包！且信息准确无误
其中：RDT序列号为:1,校验和为(对为1错为0):1,TCP序列号为:28006,TCP确认号为:1004
服务器端成功发送3号数据包的ACK！
```

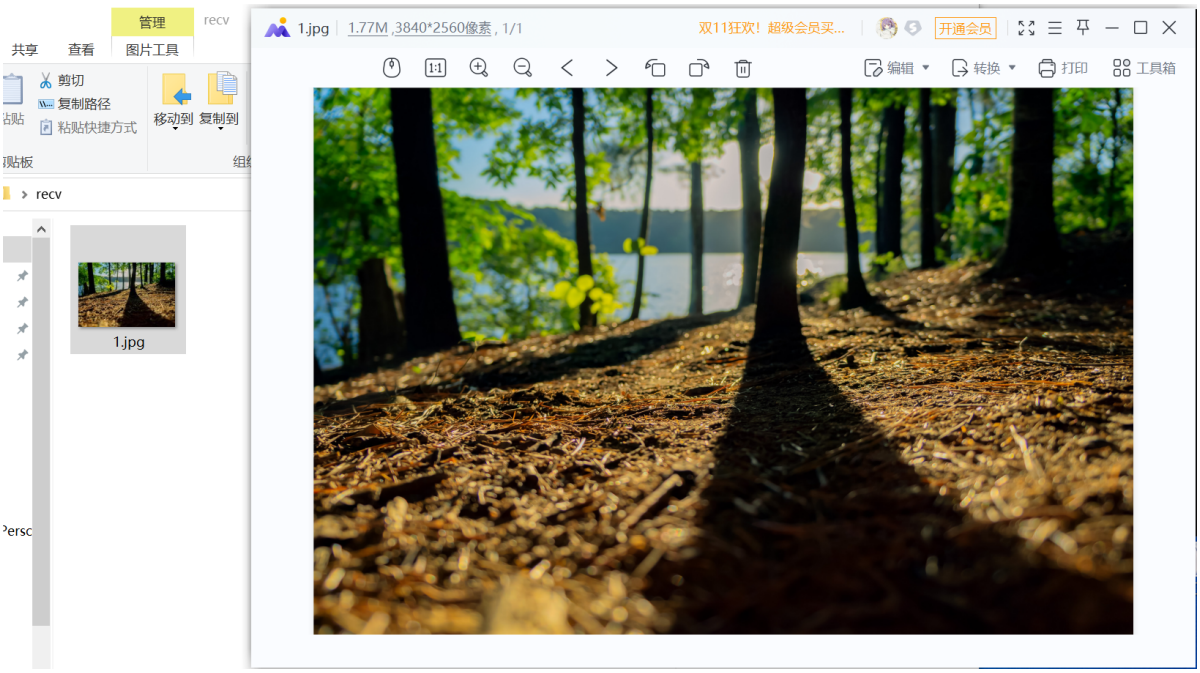
最后，来看看我们路由器的传输过程，可以看到，设置丢包率为5的情况下，实质上每20个会发生一次Miss a package，也是确实经过路由器进行有延时的传输的：



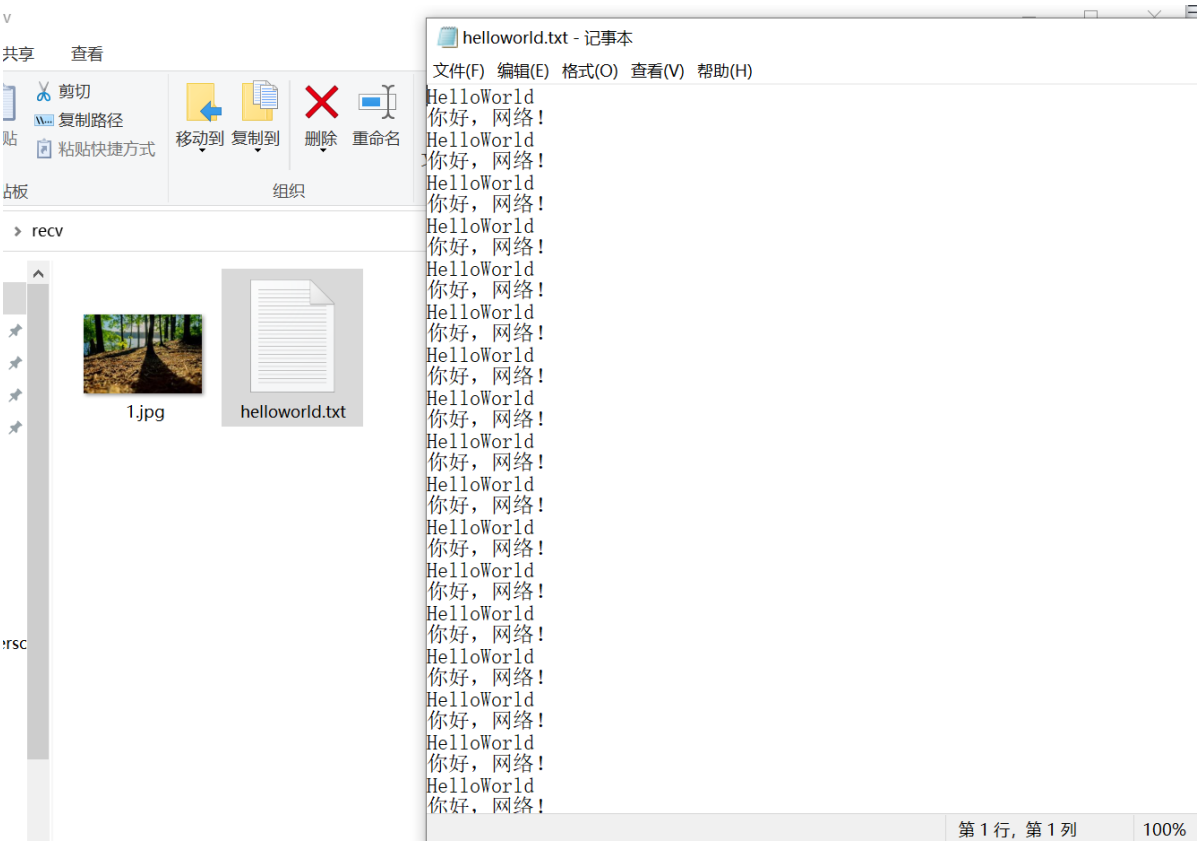
（二）传输结果分析

分析我们传输前和传输后的图片，发现文件大小不变，且能都够正常打开





同时，对文本文件，也可以正常打开并显示中英文：



五、扩展思考：实验总结与问题分析

本次实验中，实际上遇到了不少问题，比如先前提到的非阻塞模式和多线程模式啊，rdt协议的单比特的Seq处理啊等等，这些问题都最后被自己尝试思考和解决了。

但是还是有一个问题是本次实验的遗留问题，**就是对于三次握手的最后一次握手，这个ACK如果丢失，该怎么处理呢?**

这时候，因为服务器不会对该ACK响应，客户端不会等待重传，直接就会进入传输数据的状态，所以如果这个ACK发生丢失，就无法处理。这个问题的解决让我有点头疼，一个不太方便但是有效的方法是：

服务端也设置一个定时器，如果一段时间内没有收到第三次握手的信息，就发送一个NAK的字段我们在客户端的数据传输阶段，如果接收到NAK报文，客户端就立马补发一个特定的应用于三次握手的ACK确认报文。

这个想法好像是可行的，但是必须增加NAK报文，就没有给出实现。