**100071011 Computer Networks 2023-2024-2**

# Project-1

# Reliable File Transfer using Go-Back-N protocol

# Specification

| | |
|---|---|
| 学号 (Student ID) | 1120210964 |
| 姓名 (Name) | 王英泰 |
| 班号 (Class No.) | 07112102 |
| 授课教师 (Instructor) | 李凡 |

**School of Computer**
**Beijing Institute of Technology**
**April 12, 2024**

# 目录

# 一. Requirement Analysis

## 1. 项目需求

- 使用 **GBN** 协议实现可靠的文件传输，所实现的 **GBN** 协议应该支持全双工，实现双向文件传输：这就要求我们编写的程序需要同时具备发送端和接收端的功能
- 帧需要在其末尾添加 **checksum** 字段，采用 **CRC-CCITT** 标准
- 使用 **UDP Socket** 模拟并实现 **PDU** 的发送和接受，每个 **UDP** 数据包封装一个 **PDU**
- 通过配置文件配置通信和协议参数：在程序中需要编写相应的代码去读取配置文件并将其应用到代码中
- 记录通信状态写到日志文件中，并编写一个程序读取日志文件，对通信状态记录数据并进行统计分析：这需要我们在传输文件时向日志文件中写入内容，在传输文件后，使用 Python 或其他语言编写的代码读取日志文件并获得分析数据
- 支持多台主机之间的文件通信：这需要我们可以为来自不同端口的 PDU 都有一个单独的 GBN 协议来处理数据
- 考虑采用多进程、多线程、队列等技术实现并发文件传输

## 2. GBN 可靠传输文件的原理

Go-Back-N 协议，是一种在计算机网络中用于可靠地传输数据的流量控制协议。它属于数据链路层协议，主要功能是确保数据在发送方和接收方之间的可靠传输。

在 Go-Back-N 协议中，发送方可以发送多个分组，而无需等待每个分组的确认。然而，发送方受限于在流水线中未确认的分组数不能超过某个最大数 N。这个 N 代表了滑动窗口的大小，也就是发送方未确认的数据包数目。发送方按照窗口的大小将数据包发送给接收方，而接收方在正确收到数据包后会发送确认消息。

接收方只接收有序的数据包并会丢弃乱序的数据包。一旦发送方收到确认消息，它将滑动窗口向前滑动，继续发送下一个数据包。然而，如果在某个时间点，发送方没有收到预期内的确认消息，它会启动一个计时器。当计时器超时时，发送方会重传所有未确认的分组，即从第一个未确认的分组开始，直到最新的分组。这就是"回退 N"这个名

称的由来，因为在重传时，发送方可能需要回退到之前的分组，重新发送它们以确保数据的可靠传输。

Go-Back-N协议通过一系列机制来确保传输文件的可靠性。以下是这些机制的主要方面：

- 序列号机制：每个数据包都包含一个唯一的序列号，这样接收方就可以根据序列号来识别数据包的顺序。如果数据包丢失或乱序到达，接收方可以通过序列号来通知发送方。发送方则根据这些信息来决定是否需要重传数据包。
- 滑动窗口机制：发送方和接收方各维护一个滑动窗口，用来管理已经发送但尚未确认的数据包。发送方可以发送窗口内的所有数据包，而无需等待每个数据包的确认。接收方则只接收窗口内的数据包，并对每个正确接收的数据包发送确认消息。
- 确认与重传机制：接收方在接收到数据包后，会向发送方发送一个确认消息（ACK），表示已经成功接收到该数据包。如果发送方在一段时间内没有收到某个数据包的确认消息，它会认为该数据包已经丢失，并触发重传机制。发送方会重传所有未确认的数据包，直到收到它们的确认消息为止。
- 超时定时器：发送方为每个未确认的数据包设置一个超时定时器。如果定时器超时而仍未收到确认消息，发送方就认为该数据包丢失并启动重传过程。这种机制确保了即使在网络延迟或丢包的情况下，数据也能可靠地传输。
- 流量控制：通过调整滑动窗口的大小，Go-Back-N 协议可以实现流量控制。接收方可以根据其处理能力来调整窗口大小，从而避免发送方发送过多的数据导致接收方缓冲区溢出。

综合这些机制，Go-Back-N 协议能够在复杂的网络环境中实现可靠的数据传输。它能够在数据包丢失、乱序或网络拥塞的情况下，通过重传和确认机制来确保数据的完整性和顺序性。同时，通过滑动窗口和流量控制机制，它还能够有效地利用网络资源，提高传输效率。

# 3. 主要问题

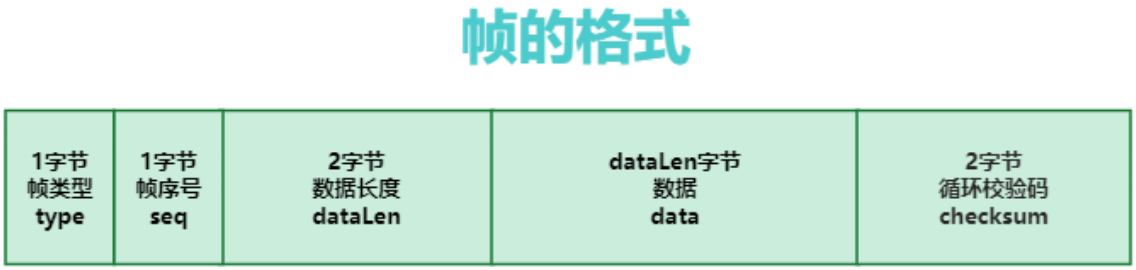
- UDP 通信
- GBN 协议的发送端

- GBN 协议的接收端
- 定时器和超时处理
- 日志文件的写入
- 配置文件的读取

# 二. Design and Implementation

## 1. 帧：对应于 Frame 类

### 1.1 帧格式




设计的帧格式如上图，可以看到该帧是变长帧，其长度会根据数据的长度发生变化，各字段的具体解释如下：

- 帧类型 type：具体类型有 data（0）、ack（1）、nak（2）
- 帧序号 seq：对于 data 类型的帧，seq 表示当前正在发送的帧序号；对于 ack 类型的帧，seq 表示期望接收到的下一帧的帧序号；对于 nak 类型的帧，seq 表示当前出错帧的帧序号，即希望发送端重新传输的帧序号
- 数据长度 dataLen：表示数据区的数据字节数，正常情况下 ack 和 nak 类型的帧该字段应该为 0；这里采用小端表示法。
- 数据 data：对于 data 类型的帧存在该字段，用于存储数据；对于 ack 和 nak 类型的帧不存在该字段
- 循环校验码 checksum：采用 CRC-CCITT 标准的循环校验码，其校验范围为 type、seq、dataLen、data

```
enum FrameType {

   data, ack, nak
```

```
};
class Frame {
public:

   FrameType type;

   unsigned char seq;

   unsigned short source;

   unsigned short dataLen;

   std::vector<char> data;

   unsigned short checksum; // 采用 CRC-CCITT 标准
}
```

## 1.2 帧校验和检查

循环校验码（CRC，Cyclic Redundancy Check）是一种在数据通信领域中常用的差错校验码。它通过对数据进行多项式除法运算来生成一个固定长度的校验码，附加在数据的末尾进行传输。接收方在收到数据后，再次进行同样的多项式除法运算，如果得到的结果为零，则表明数据在传输过程中没有发生错误；如果不为零，则说明数据存在错误。

CRC 码的生成过程大致如下：

- 选择生成多项式：生成多项式是 CRC 算法的核心，它决定了 CRC 码的特性。生成多项式通常表示为 G(x)，其最高位和最低位必须为 1。
- 转换数据为多项式：将要发送的数据（信息字段）转换为多项式 C(x)。每个二进制位对应多项式的一个项，例如，二进制数 1101 对应的多项式为 x^3 + x^2 + 1。
- 进行多项式除法：使用选定的生成多项式 G(x)对数据多项式 C(x)进行模 2 除法运算（即异或运算）。模 2 除法运算的特点是，运算过程中不产生进位或借位，每一位的运算都是独立的。运算的结果是一个余数，这个余数就是 CRC 校验码。
- 附加校验码：将计算得到的 CRC 校验码附加在原始数据的末尾，形成带有 CRC 校验码的数据包进行发送。

在接收端，接收方收到数据后，使用同样的生成多项式对接收到的数据进行模 2 除法运算。如果运算结果为 0，说明数据在传输过程中没有发生错误；如果不为 0，则说明数据存在错误，接收方可以根据余数进一步确定错误的位置并进行纠正。

```
unsigned short Frame::crc_ccitt() {
    const unsigned short polynomial = 0x1021; // 循环校验码对应的多项式：x^16+x^12+x^5+1
    unsigned short crc = 0xFFFF;

    crc ^= (unsigned short)this->type << 8;
    for (int i = 0; i < 8; ++i) {
        if (crc & 0x8000) {
            crc = (crc << 1) ^ polynomial;
        }
        else {
            crc <<= 1;
        }
    }
    crc ^= (unsigned short)this->seq << 8;
    for (int i = 0; i < 8; ++i) {
        if (crc & 0x8000) {
            crc = (crc << 1) ^ polynomial;
        }
        else {
            crc <<= 1;
        }
    }
    crc ^= (unsigned short)(this->dataLen && 0xFF) << 8;
    for (int i = 0; i < 8; ++i) {
        if (crc & 0x8000) {
            crc = (crc << 1) ^ polynomial;
        }
        else {
            crc <<= 1;
        }
    }

    crc ^= (unsigned short)(this->dataLen >> 8 && 0xFF) << 8;
    for (int i = 0; i < 8; ++i) {
        if (crc & 0x8000) {
            crc = (crc << 1) ^ polynomial;
        }
        else {
            crc <<= 1;
```

```
        }
    }


    for (auto byte : this->data) {
        crc ^= (unsigned short)byte << 8;
        for (int i = 0; i < 8; ++i) {
            if (crc & 0x8000) {
                crc = (crc << 1) ^ polynomial;
            }
            else {
                crc <<= 1;
            }
        }
    }

    return crc;
}
```

## 1.3 帧的其他方法

为了方便程序对帧的使用，这里我编写了两个方法来从 Frame 对象序列化为字节数组和从字节数组反序列化为 Frame 对象：分别为类成员方法 getByteArray 和静态方法 fromByteArray。

```
void Frame::getByteArray(char* bytes, int* len) {
    if (*len < this->getLength()) {
        // 防止缓存区溢出
        *len = 0;
        return;
    }
    int index = 0;
    bytes[index++] = (this->type) & 0xFF;
    bytes[index++] = (this->seq) & 0xFF;
    bytes[index++] = this->dataLen & 0xFF;
    bytes[index++] = (this->dataLen >> 8) & 0xFF;

    for (int i = 0; i < this->dataLen; i++) {
        bytes[index++] = this->data[i];
    }

    bytes[index++] = this->checksum & 0xFF;
    bytes[index++] = (this->checksum >> 8) & 0xFF;
    *len = index;
```

```
}

static Frame Frame::fromByteArray(char* bytes) {
    int index = 0;
    Frame frame;
    frame.type = (FrameType)bytes[index++];
    frame.seq = bytes[index++];
    frame.dataLen = *((unsigned short*)(bytes+index));
    index += 2;
    for (int i = 0; i < frame.dataLen; i++) {
        frame.data.push_back(bytes[index++]);
    }

    frame.checksum = *((unsigned short*)(bytes+index));
    index += 2;
    return frame;
}
```

# 2. 包：对应于 Packet 类

为了方便标识文件传输开始和结束，并更好模拟实际网络系统，向帧中添加的数据段实际上是 Packet 对象对应的字节数据。其包含的字段如下：

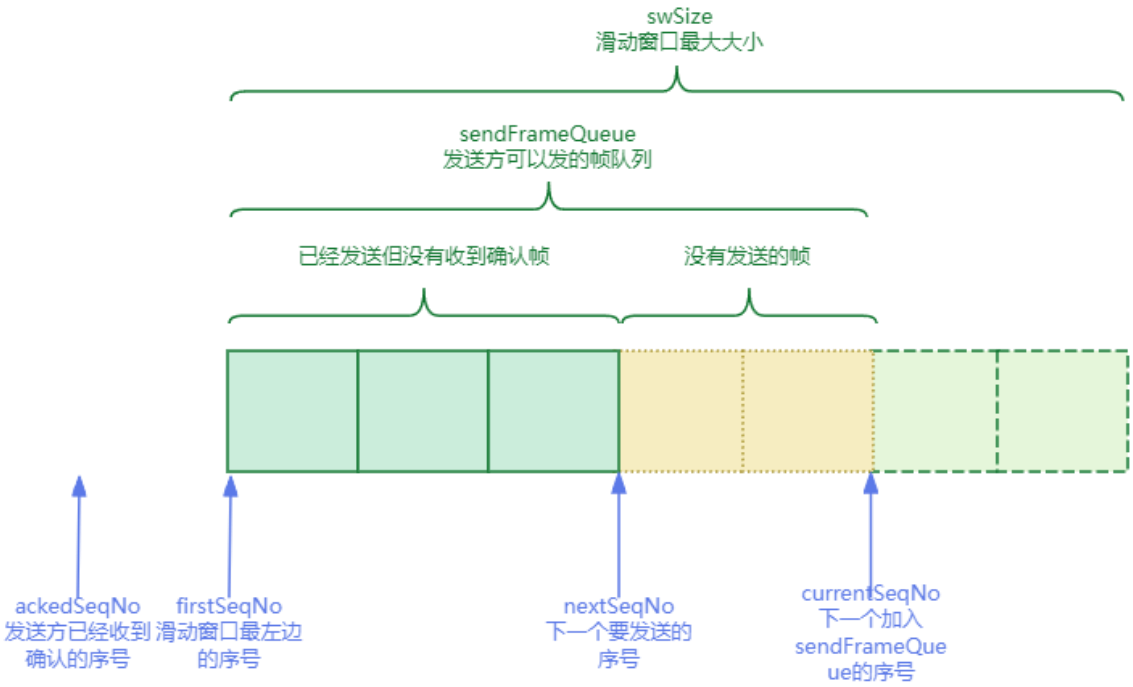
- 1 字节类型：type
- 2 字节数据大小：dataLen
- dataLen 字节数据：data

其中 type 类型有 fileStart、fileConent、fileEnd，分别表示文件开始传输、文件传输内容、文件传输结束。fileStart 类型的 Pakcet 的 data 字段存放文件后缀名，方便接收段处理接受到的文件并创建文件，fileContent 类型的 Packet 的 data 字段存放文件内容，fileEnd 类型的 Packet 告诉接收端文件接收完毕，可以释放文件句柄。

# 3. GoBackN 协议：对应于 DataLinkLayer 类

## 3.1 发送方滑动窗口控制

我采用了 C++中的双端队列 deque 来表示滑动窗口，对应 DataLinkLayer 类的 sendFrameQueue 属性。为了控制滑动窗口相关状态，设置了 firstSeqNo、nextSeqNo、currentSeqNo、ackedSeqNo、expectedSeqNo 属性，具体解释如下：




- firstSeqNo：滑动窗口最左边的帧序号
- nextSeqNo：发送方下一个要发送的帧序号
- currentSeqNo：下一个加入发送方帧队列的帧序号
- ackedSeqNo：发送方已经收到确认帧的帧序号
- expectedSeqNo：接收方期望接收到的帧序号
- swSize：滑动窗口的最大大小

```
int initSeqNo; // 起始 PDU 的序号
int swSize; // 发送窗口大小
std::deque<Frame> sendFrameQueue; // 发送方的帧队列
int firstSeqNo = initSeqNo; // 滑动窗口最左边的序号
int nextSeqNo = initSeqNo; // 下一个要发送的序号
int currentSeqNo = initSeqNo; // 下一个加入发送方帧队列的序号
int ackedSeqNo = this->decSeqNo(initSeqNo); // 发送方已经确认的帧序号
int expectedSeqNo = initSeqNo; // 接收方期望接受到的帧序号
```

## 3.2 从网络层获取 Packet 添加到滑动窗口中

当滑动窗口还没有满时，从 Packet 队列中获取 Packet 转换为字节数组，然后构建 Frame 对象，添加到滑动窗口中，这里要对添加到滑动窗口中的帧设置正确的序号。

```
// 从包队列取出包转换为帧加入到滑动窗口中
```

```
while (this->sendFrameQueue.size() < this->swSize) { // 当前滑动窗口没满，可以向滑动窗口中添加数据
    Packet packet;
    if (this->sendPacketQueue.try_pop(packet)) {
        int bufferLen = packet.getLength();
        char* buffer = new char[bufferLen];
        memset(buffer, 0, bufferLen);
        packet.getByteArray(buffer, &bufferLen);

        this->sendFrameQueue.push_back(Frame(FrameType::data, this->currentSeqNo, buffer, bufferLen));
        this->currentSeqNo = this->incSeqNo(this->currentSeqNo);
        delete[] buffer;
    }
    else {
        break;
    }
}
```

## 3.3 从滑动窗口中取出没有发送的帧向物理层发送

当没有未确认帧存在于滑动窗口中时才能发送没有发送过的帧。这里我用了一个定时器来控制一组帧的发送超时，当第一个帧开始发送时开始计时。

```
// 从滑动窗口中取出没有发送的帧向物理层发送
if (firstSeqNo == nextSeqNo) { // 当未确认帧都确认后才进行发送
    bool flag = false;
    for (auto& frame : this->sendFrameQueue) {
        // 发送滑动窗口中所有没有发送的帧
        if (!flag && frame.seq == this->nextSeqNo) {
            flag = true;
            // 开始计时
            this->timer.addTimer(0);
        }
        if (flag) {
            std::string log = "[send] pdu_to_send=" + std::to_string(frame.seq) + ",status=New,ackedNo=" + std::to_string(this->ackedSeqNo);
            Logger::write(log);
            this->physicalLayer->send(frame);
            this->nextSeqNo = this->incSeqNo(this->nextSeqNo);
        }
    }
}
```

## 3.4 从物理层中取出接收到的帧进行处理

从物理层中取出帧进行处理。对帧的不同类型采用不同的处理：

- 类型为 **data** 的帧：首先判断接收到的帧的序号是否等于 expcetedSeqNo，如果不等于则说明接受到的数据顺序错误，要忽略；否则说明接收到的数据顺序正确，接着进行循环校验码的验证，如果循环校验码验证正确，则向物理层发送确认帧，否则向物理层发送否认帧
- 类型为 **ack** 的帧：对滑动窗口中的第一个帧弹出，并增加 firstSeqNo 和 ackedSeqNo；如果滑动窗口所有已经发送的帧都收到了确认帧，则停止定时器。
- 类型为 **nak** 的帧：停止原来的定时器并开启新的定时器，并发送滑动窗口中已经发送但是还没有收到确认帧的所有帧。

```
// 从物理层中取出帧进行处理
Frame frame;
if (this->physicalLayer->receive(frame)) {
   switch (frame.type) {
   case FrameType::data:
       // 判断从物理层接收到的帧是否符合预期
       if (frame.seq == this->expectedSeqNo) {
           // 向物理层发送确认帧或否者帧
           if (frame.isRight()) {
               std::string log = "[recv] pdu_exp=" +
std::to_string(this->expectedSeqNo) + ",status=OK,pdu_recv=" +
std::to_string(frame.seq);
               Logger::write(log);

   this->receivePacketQueue.push(Packet::fromByteArray(frame.data.data
()));
               this->expectedSeqNo =
this->incSeqNo(this->expectedSeqNo);
               this->physicalLayer->send(Frame(FrameType::ack,
this->expectedSeqNo, nullptr, 0));
           }
           else {
               this->physicalLayer->send(Frame(FrameType::nak,
this->expectedSeqNo, nullptr, 0));
               std::string log = "[recv] pdu_exp=" +
std::to_string(this->expectedSeqNo) + ",status=DataErr,pdu_recv=" +
std::to_string(frame.seq);
               Logger::write(log);
```

```
                std::cout << "[datalink] 接收到的帧校验错误" << std::endl;
            }
        }
        else {
            std::string log = "[recv] pdu_exp=" +
std::to_string(this->expectedSeqNo) + ",status=NoErr,pdu_recv=" +
std::to_string(frame.seq);
            Logger::write(log);
        }
        break;
    case FrameType::ack:
    {
        // 如果收到了所有已发送但没有确认帧的确认帧，则停止计时器
        if (frame.seq == nextSeqNo) {
            this->timer.removeTimer(0);
        }
        this->sendFrameQueue.pop_front();
        this->firstSeqNo = this->incSeqNo(this->firstSeqNo);
        this->ackedSeqNo = this->incSeqNo(this->ackedSeqNo);
        break;
    }
    case FrameType::nak:
    {
        // 停止计时器并开始新的计时器
        this->timer.removeTimer(0);
        this->timer.addTimer(0);
        for (auto& frame : this->sendFrameQueue) {
            // 发送滑动窗口中所有没有收到确认帧的帧
            if (frame.seq == this->nextSeqNo) {
                break;
            }
            this->physicalLayer->send(frame);
            std::string log = "[send] pdu_to_send=" +
std::to_string(frame.seq) + ",status=RT,ackedNo=" +
std::to_string(this->ackedSeqNo);
            Logger::write(log);
        }

        break;
    }
    default:
        std::cout << "[datalink][warning] 不期望的帧类型" << std::endl;
    }
}
```

### 3.5 对超时进行处理

当收到超时事件后，开启新定时器，并重新发送滑动窗口中所有发送但是没有收到确认帧的帧。

```
EventType eventType;
if (this->eventQueue->try_pop(eventType)) {
    switch (eventType) {
    case EventType::timeout:
    {
        this->timer.addTimer(0);
        for (auto& frame : this->sendFrameQueue) {
            // 发送滑动窗口中所有收到确认帧的帧
            if (frame.seq == this->nextSeqNo) {
                break;
            }
            this->physicalLayer->send(frame);
            std::string log = "[send] pdu_to_send=" +
std::to_string(frame.seq) + ",status=TO,ackedNo=" +
std::to_string(this->ackedSeqNo);
            Logger::write(log);
        }
        std::cout << "[datalink] 超时" << std::endl;
        break;
    }
    default:
        std::cout << "其他事件类型" << std::endl;
    }
}
```

## 4. UDP 通信：对应于 PhysicalLayer 类

### 4.1 判断是服务端还是客户端

服务端和客户端有不同的行为，服务端需要绑定端口，并记录客户端的端口地址，而客户端不需要绑定端口。所以这里我们需要通过互斥量来区分是服务端还是客户端，具体代码如下：

```
bool isServer = false;
// 通过互斥信号量来判断该程序是否该为服务端
HANDLE mutexHandle = OpenMutex(MUTEX_ALL_ACCESS, FALSE, L"gobackn");
if (mutexHandle == NULL) {
    mutexHandle = CreateMutex(NULL, FALSE, L"gobackn");
```

```
    this->server = true;
}
else {
    this->server = false;
}
```

## 4.2 初始化 Socket

对于服务端，要绑定端口，对于客户端，要在启动时向服务端发送“client”，从而让服务端得知客户端的端口号。无论服务端还是客户端，都需要设置 Socket 属性为接收时非阻塞。

```
bool init() {
    // 初始化 Winsock
    if (WSAStartup(MAKEWORD(2, 2), &this->wsaData) != 0) {
        std::cerr << "WSAStartup failed" << std::endl;
        return false;
    }

    // 创建套接字
    this->sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (sockfd == INVALID_SOCKET) {
        std::cerr << "Socket creation failed" << std::endl;
        WSACleanup();
        return false;
    }

    u_long iMODE = 1; // 1 为非阻塞，0 为阻塞
    ioctlsocket(this->sockfd, FIONBIO, &iMODE); // 设置 recvfrom 是否为阻塞

    // 服务端需要绑定端口；而客户端不需要绑定端口，但需要向服务器发送消息让服务器记住客户端
    if (this->server) {
        // 绑定套接字
        if (bind(this->sockfd, (struct sockaddr*)&this->serverAddr, sizeof(this->serverAddr)) != 0) {
            std::cerr << "Bind failed" << std::endl;
            closesocket(this->sockfd);
            WSACleanup();
            return false;
        }
    }
    else {
```

```
        char message[] = "client";
        sendto(this->sockfd, message, sizeof(message), 0, (struct
sockaddr*)&this->serverAddr, sizeof(this->serverAddr));
    }

    return true;
}
```

## 4.3 模拟帧丢失和帧出错

这里我们根据 PhysicalLayer 的属性 errorRate 和 lostRate 来产生随机数，从而决定是否丢失和出错。

```
// 模拟丢失
if (this->lostRate > 0) {
    // 初始化随机数生成器
    std::random_device rd;  // 用于获得种子
    std::mt19937 gen(rd()); // 使用随机设备种子初始化 Mersenne Twister 生成
器
    std::uniform_int_distribution<> distrib(1, this->lostRate); // 定义
分布范围

    // 获取一个随机数
    int randomNumber = distrib(gen);

    if (randomNumber == 1) {
        std::cout << "[physical] 帧丢失" << std::endl;
        goto sendFinish;
    }
}
// 模拟错误
if (this->errorRate > 0 && frame.type==FrameType::data) {
    // 初始化随机数生成器
    std::random_device rd;  // 用于获得种子
    std::mt19937 gen(rd()); // 使用随机设备种子初始化 Mersenne Twister 生成
器
    std::uniform_int_distribution<> distrib(1, this->errorRate); // 定义
分布范围

    // 获取一个随机数
    int randomNumber = distrib(gen);

    if (randomNumber == 1) {
        buffer[2] = 'w';
        std::cout << "[physical] 帧出错" << std::endl;
```

```
    }
}
```

### 4.4 发送和接收数据报

这里主要调用 socket 接口 sendto 和 recefrom 来实现数据报的发送和接收。

## 5. 配置：对应于 Config 类

Config 类的配置字段都设置为静态属性，具体如下：

```
static unsigned short UDPPort; // UDP 端口
static unsigned short DataSize; // PDU 中数据字段的长度
static int ErrorRate; // PDU 错误率，如 ErrorRate=10 表示每 10 帧中一帧出错
static int LostRate; // PUD 丢失率，如 LostRate=10，表示每 10 帧丢一帧
static int SWSize; // 发送窗口大小
static int InitSeqNo; // 起始 PDU 序号
static int Timeout; // 超时定时器值，单位为毫秒
```

在程序启动时 main 函数会调用 Config::readConfigFile()，如果读取配置文件失败则使用默认配置。

在初始化 PhysicalLayer、DatalinkLayer、NetworkLayer 时，我们会传入配置属性来初始化系统。

```
Config::readConfigFile();
ThreadSafeQueue<EventType> eventQueue;
PhysicalLayer physicalLayer(&eventQueue, Config::UDPPort,
Config::ErrorRate, Config::LostRate);
bool isServer = physicalLayer.isServer();
if (isServer) {
    Logger::createLogFile("server");
    cout << "我是服务端" << endl;
}
else {
    Logger::createLogFile("client");
    cout << "我是客户端" << endl;
}
DataLinkLayer dataLinkLayer(&eventQueue, Config::InitSeqNo,
Config::SWSize, Config::Timeout, &physicalLayer);
NetworkLayer networkLayer(&eventQueue, &dataLinkLayer);
```

## 6. 日志：对应于 Logger 类和外部的 log.ipynb 文件

写入日志的操作很简单，就是在程序启动时创建日志文件，然后在程序运行过程中向日志文件中写入内容，在程序运行完毕后关闭日志文件。这里主要介绍一下日志文件的分析过程：编写 python 程序，分析各种类型的帧的数量，并计算出传输文件的时间。

```python
log_file_name = "log-client-1714135838335.txt"

lines = []
timeout_num = 0 # 超时次数
repeat_num = 0 # 重传次数
with open(log_file_name, "r") as file:
    lines = file.readlines()

for line in lines:
    if line.find("status=TO") != -1:
        timeout_num += 1
    elif line.find("status=RT") != -1:
        repeat_num += 1

print(f"通信总次数 {len(lines)}")
print(f"超时次数 {timeout_num}")
print(f"重传次数 {repeat_num}")
print(f"PDU 总数量 {len(lines) - timeout_num - repeat_num}")
total_time = int(lines[-1][1:14]) - int(lines[0][1:14])
print(f"总时间 {total_time} ms")
```

## 7. 线程安全的队列：对应于 ThreadSafeQueue 类

由于 C++标准库提供的队列是线程不安全的，所以需要我们自己实现线程安全的队列。这里主要用到了 C++标准库提供的互斥量和条件变量。

```cpp
template<typename T>
class ThreadSafeQueue {
private:
    std::queue<T> q;
    std::mutex mut;
    std::condition_variable cond;

public:
    void push(T new_value);

    // 如果队列为空则会阻塞
    void wait_and_pop(T& value);
```

```
    // 非阻塞的，如果队列为空则返回 false
    bool try_pop(T& value);

    int size();
};

template<typename T>
void ThreadSafeQueue<T>::push(T new_value) {
    std::lock_guard<std::mutex> lock(mut);
    q.push(new_value);
    cond.notify_one();
}

template<typename T>
void ThreadSafeQueue<T>::wait_and_pop(T& value) {
    std::unique_lock<std::mutex> lock(mut);
    cond.wait(lock, [&] { return !q.empty(); });
    value = q.front();
    q.pop();
}

template<typename T>
bool ThreadSafeQueue<T>::try_pop(T& value) {
    std::lock_guard<std::mutex> lock(mut);
    if (q.empty()) return false;
    value = q.front();
    q.pop();
    return true;
}

template<typename T>
int ThreadSafeQueue<T>::size() {
    std::lock_guard<std::mutex> lock(mut);
    auto size = this->q.size();
    return size;
}
```

# 三. Development

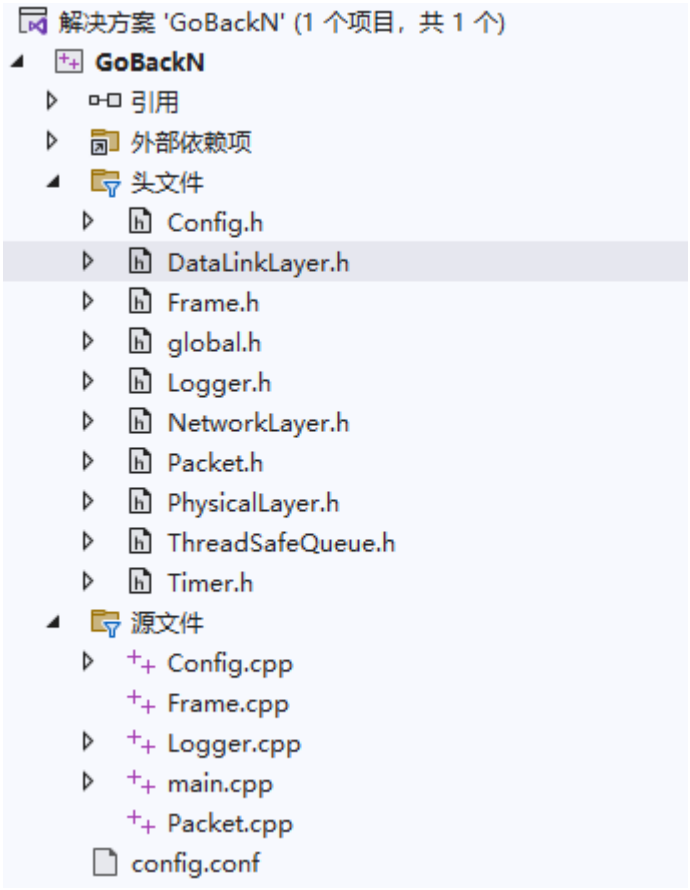
## 1. 开发环境

- 操作系统：Windows 11
- 编程语言：C++

- 库：ws2_32.lib
- IDE：Visual Studio 2022

## 2. 项目结构

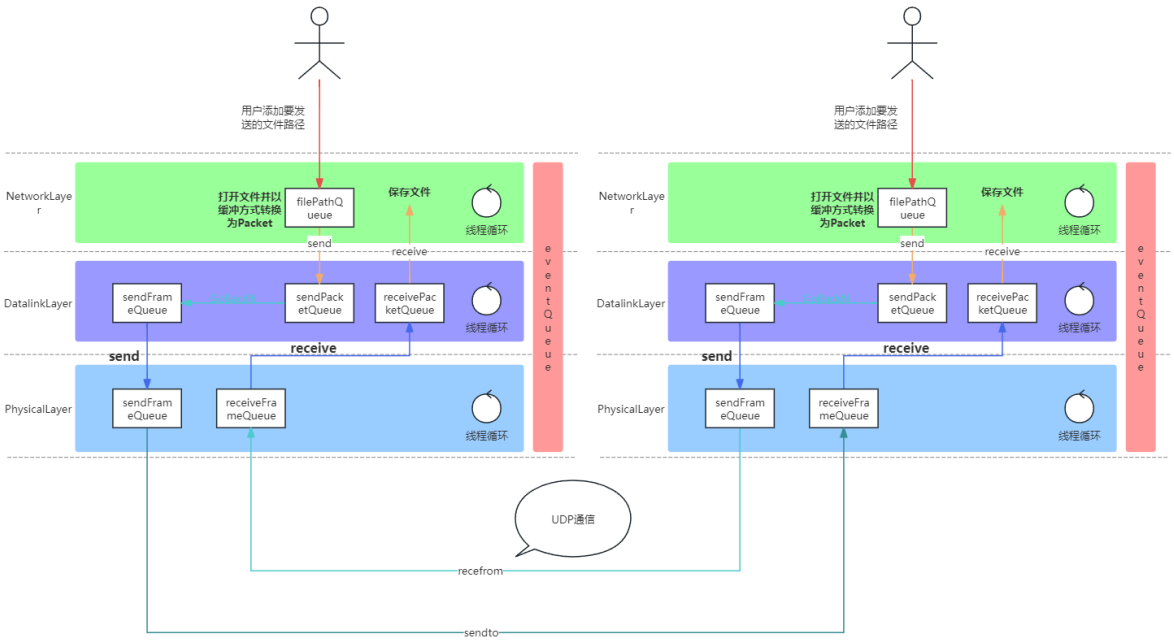
本项目采用了面向对象的编写方式，具有清晰的项目结构和功能分类，




- PhysicalLayer：该类模拟了物理层，实际采用 UDP Socket 来进行模拟并实现帧的发送和接收，每个 UDP 数据报封装一个 Frame
- DataLinkLayer：该类模拟了数据链路层，是实现 GoBackN 协议所在的类
- NetworkLayer：该类模拟了网络层，是对文件相关的操作所在的类
- Frame：该类表示帧，用于在数据链路层和物理层之间进行通信
- Packet：该类表示包，用于网络层和数据链路层之间进行通信
- Timer：该类表示定时器
- Logger：该类用于记录日志文件

- Config：该类用于读取配置文件
- ThreadSafeQueue：线程安全的队列

# 3. 总体结构




在设计这个项目时，我充分模拟实际主机之间的通信过程，将程序分成了物理层、数据链路层和网络层。其中物理层主要通过 UDP 通信模拟了物理层之间的通信，在通信过程中也会模拟丢包和错误；数据链路层即为 GoBackN 协议所在的层，在层中我们会保证数据的顺序和正确性；网络层主要负责具体的文件操作，如将文件转换为 Packet，将 Packet 缓存为文件。总之，我模拟了计算机网络通信协议中数据传输的详细过程，展现了发送方与接收方之间如何通过多个层次来传输数据，不仅包括了数据的封装、传输和解封，还涉及到了多个层次的协作和数据处理。

在图片的左侧，我们看到的是主机 1 的操作流程。首先，用户添加要发送的文件路径，网络层接收到这一指令后，会打开文件并将其内容以某种方式转换为数据包（Packet）。这些数据包被添加到数据链路层中的队列中，等待进一步的处理。同时，网络层也会从数据链路层中取出 Packet 并存入接收文件中。

随后，数据链路层会将 Packet 转换为 Frame，并以 GoBackN 方式计算出帧的 seq 和 checksum，然后添加到物理层的要发送的 Frame 队列中，等待物理层进行发送。同时，数据链路层也会从物理层的接收帧队列中取出帧进行处理，根据协议要求判断是接收并转换为 Packet 且存入自身的接收包队列中，并发送确认帧，还是发送否认帧或丢弃。

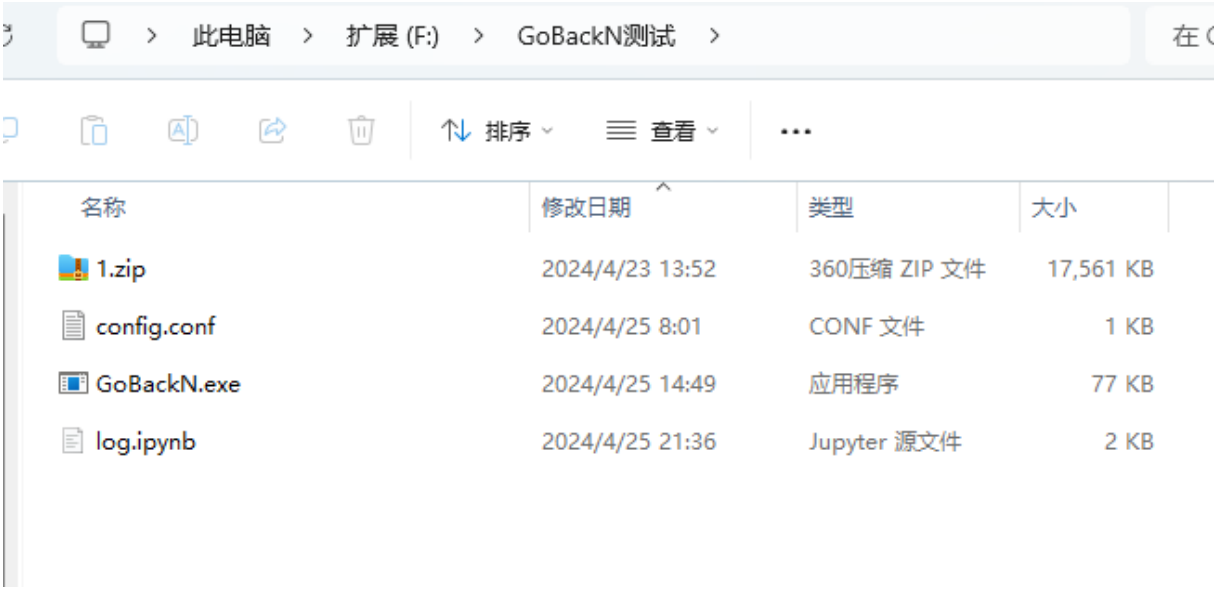
在物理层，我们通过 UDP 通信方式来将 Frame 以数据包的形式发送给接收方，并同时接收其他主机的数据并转换为 Frame，存入物理层的接收帧队列中。

由于我们实现的全双工，所以图片右侧的主机 2 的处理流程和主机 1 完全相同。

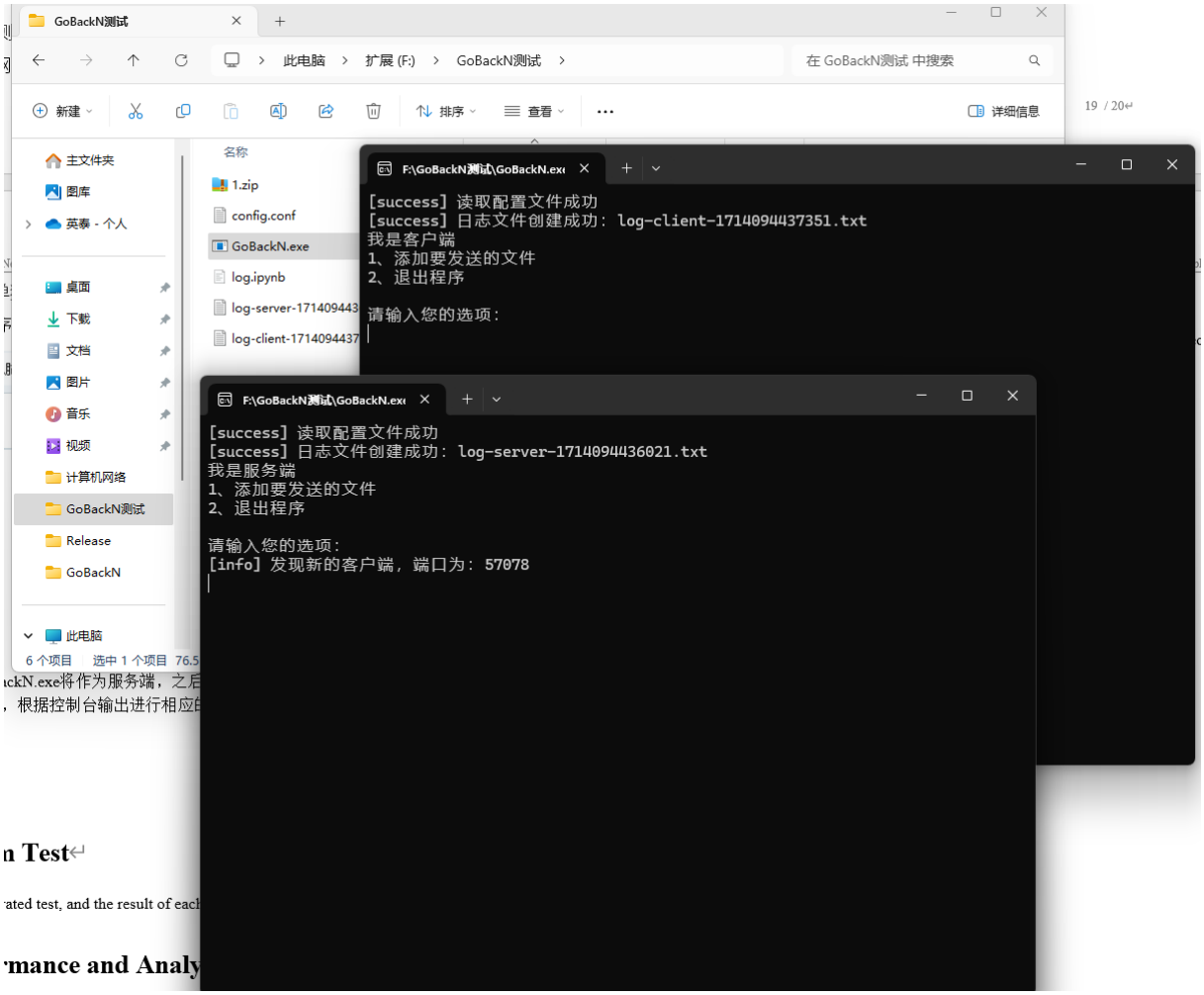
# 四. System Deployment, Startup, and Use

在编写完代码后，打包 Release 版，就得到了最终生成的 exe 程序。

将 exe 程序单独移动到一个文件夹中，并和配置文件 config.conf 放在同一目录中，方便我们测试使用。然后双击 exe 程序即可启动。




第一次启动的 GoBackN.exe 将作为服务端，之后所有启动的 GoBackN.exe 将作为客户端。服务端和客户端启动之后，根据控制台输出进行相应的操作即可。

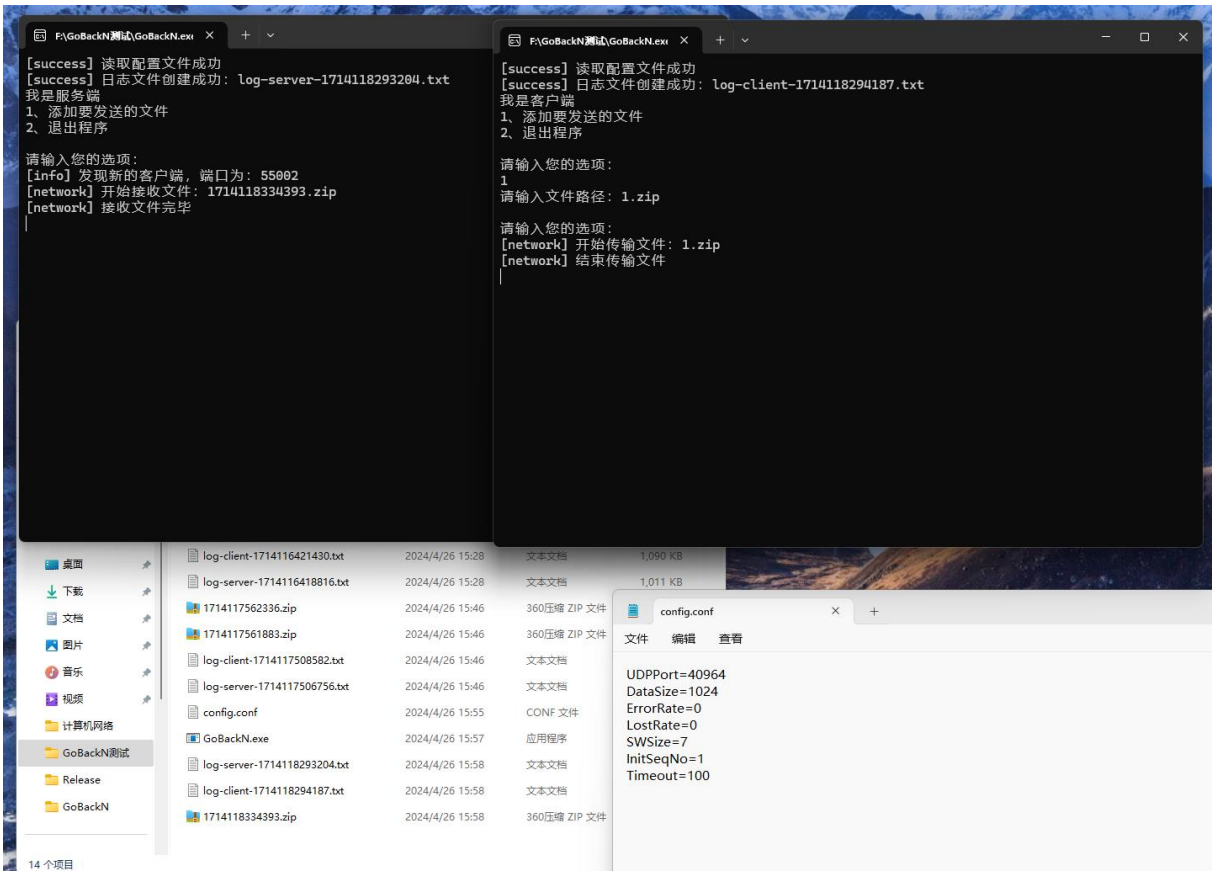



# 五. System Test

单元测试、集成测试在编写程序过程中已经完成，所以这里不再进行赘述。接下来主要呈现一些系统测试的结果。

## 1. 配置文件是否生效检查

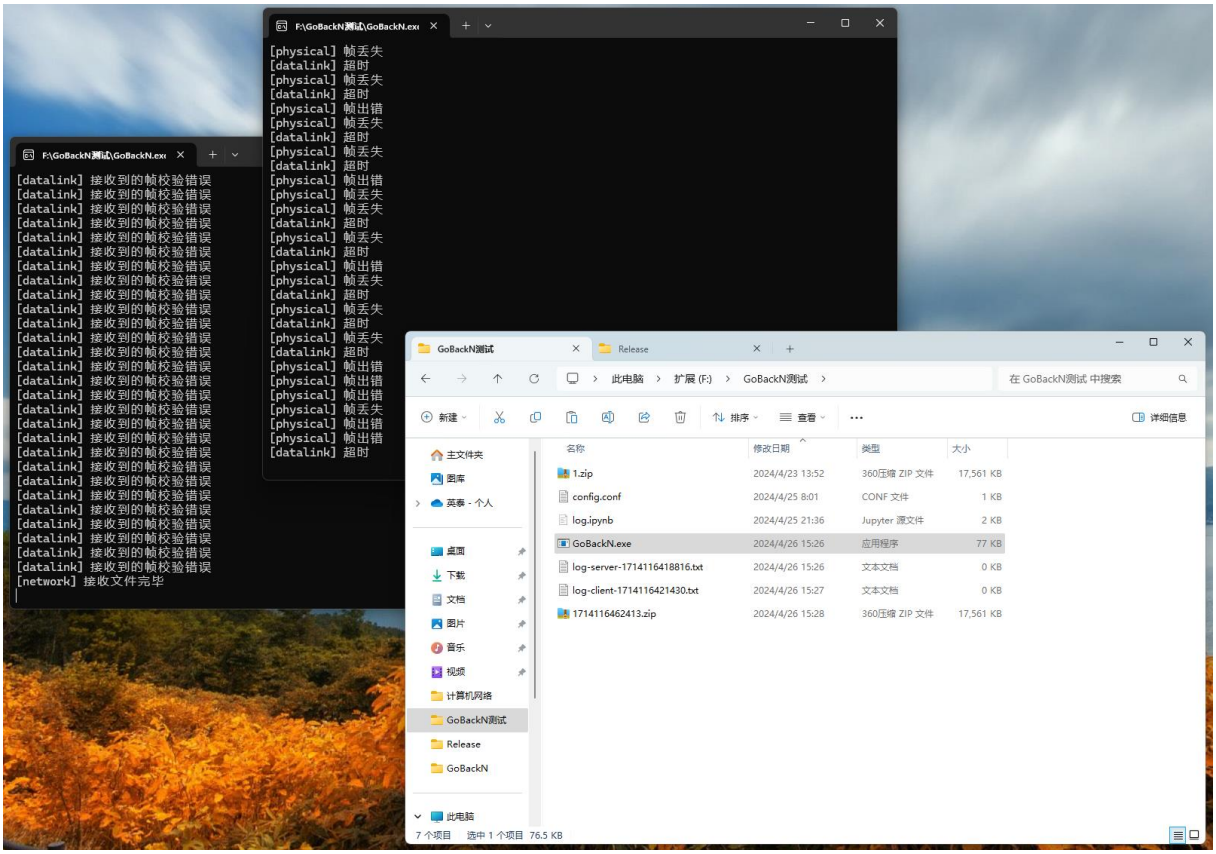
修改配置文件中的 LostRate 和 ErrorRate 都为 0，然后启动 GoBackN 传输文件，可以发现控制台输出是干净的，表示并没有发生丢帧和错误帧。




当我们修改配置文件中的 LostRate 和 ErrorRate 都为 100 时，即有 1%的概率发生帧丢失和帧出错，然后启动 GoBackN 传输文件，可以发现控制台会输出“[physical] 帧丢失”、“[physical] 帧出错”、“[datalink] 超时”、“[datalink] 接收到的帧校验错误”。

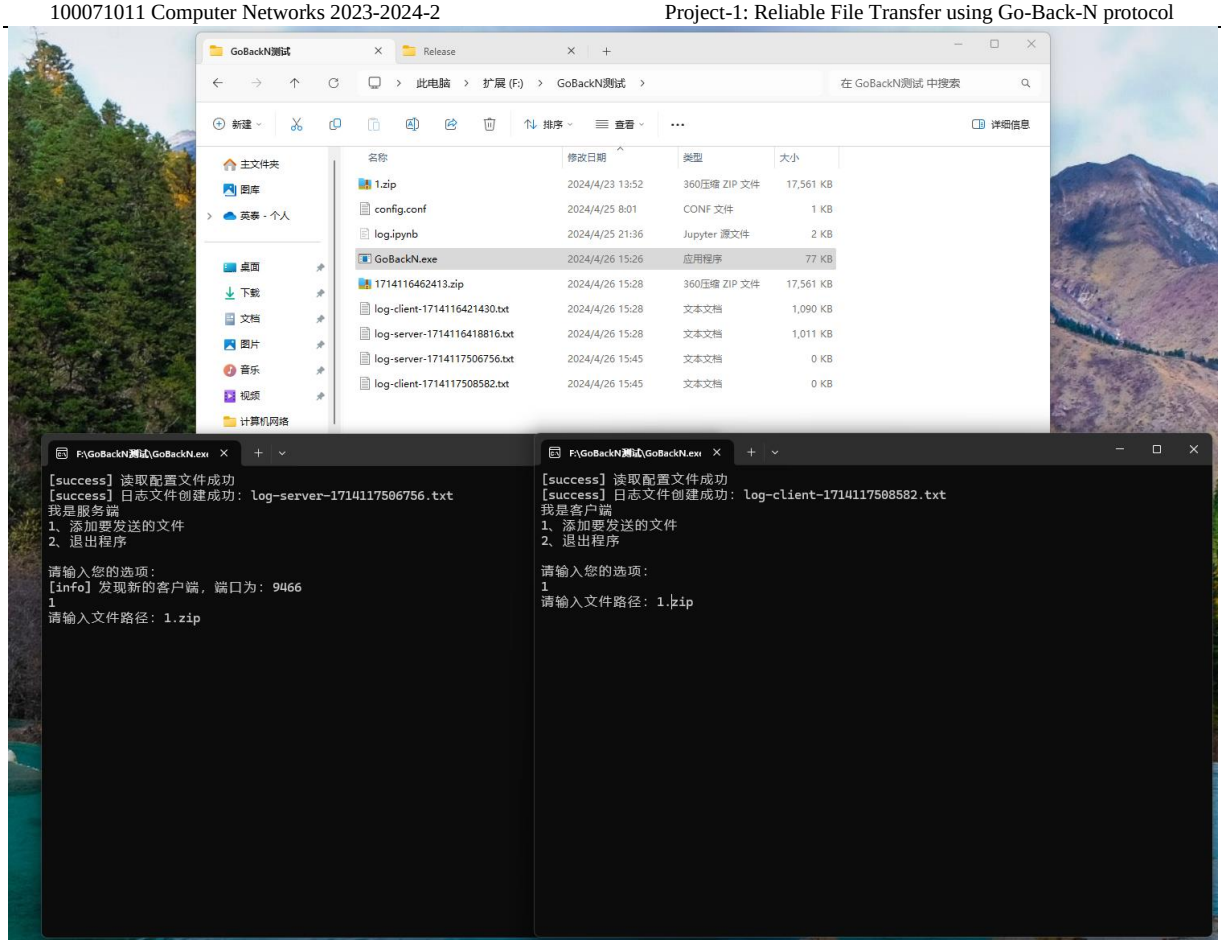
## 2. 大文件传输

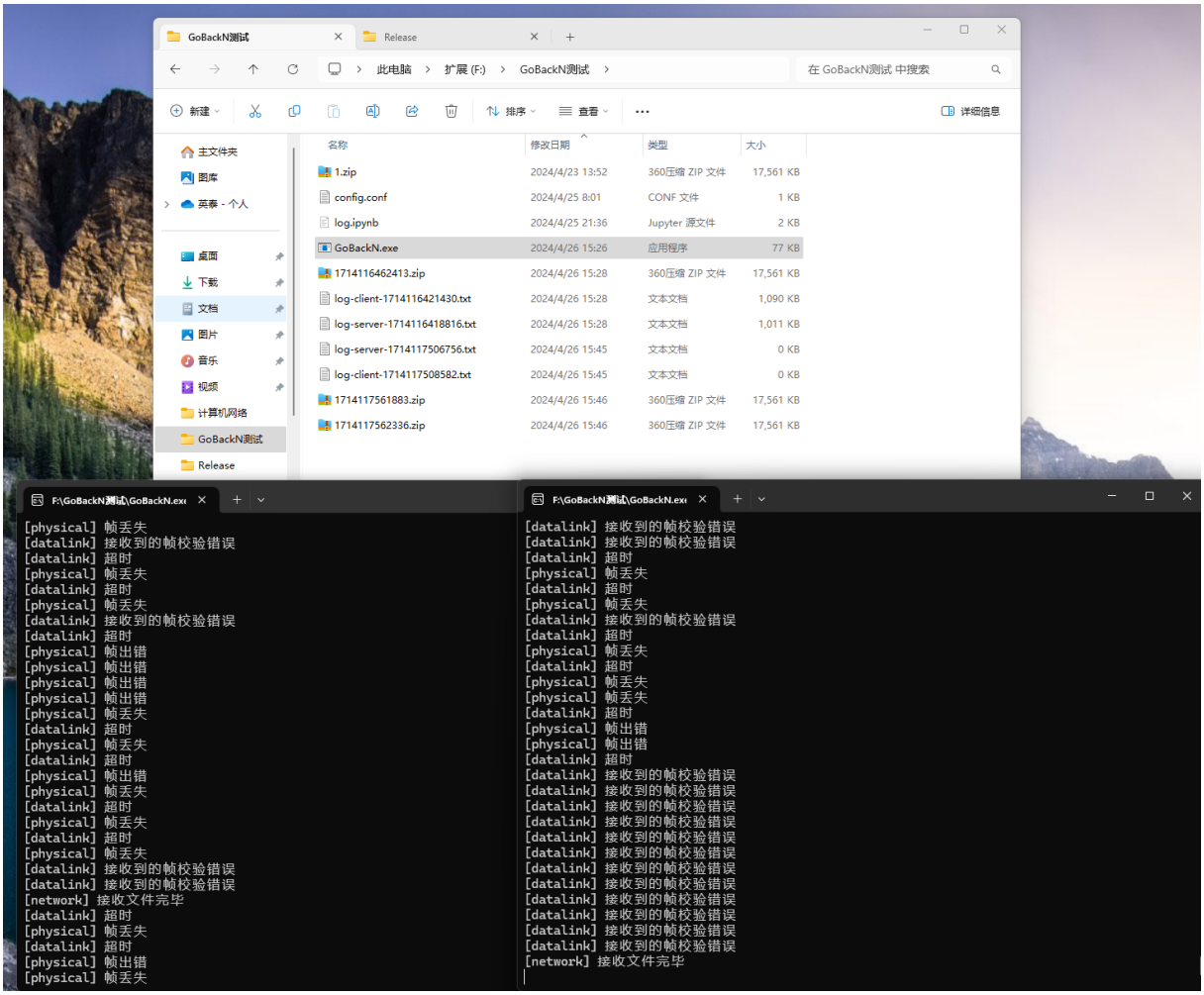
打开两个 GoBackN 进程，然后输入命令和文件路径，传输一个 17.6MB 大小的文件，经过一段时间后，接收方显示“[network] 接收文件完毕”。然后查看接收到的文件大小，可以看到两个文件大小相同，且接收到的文件可以正常打开。




# 3. 全双工传输

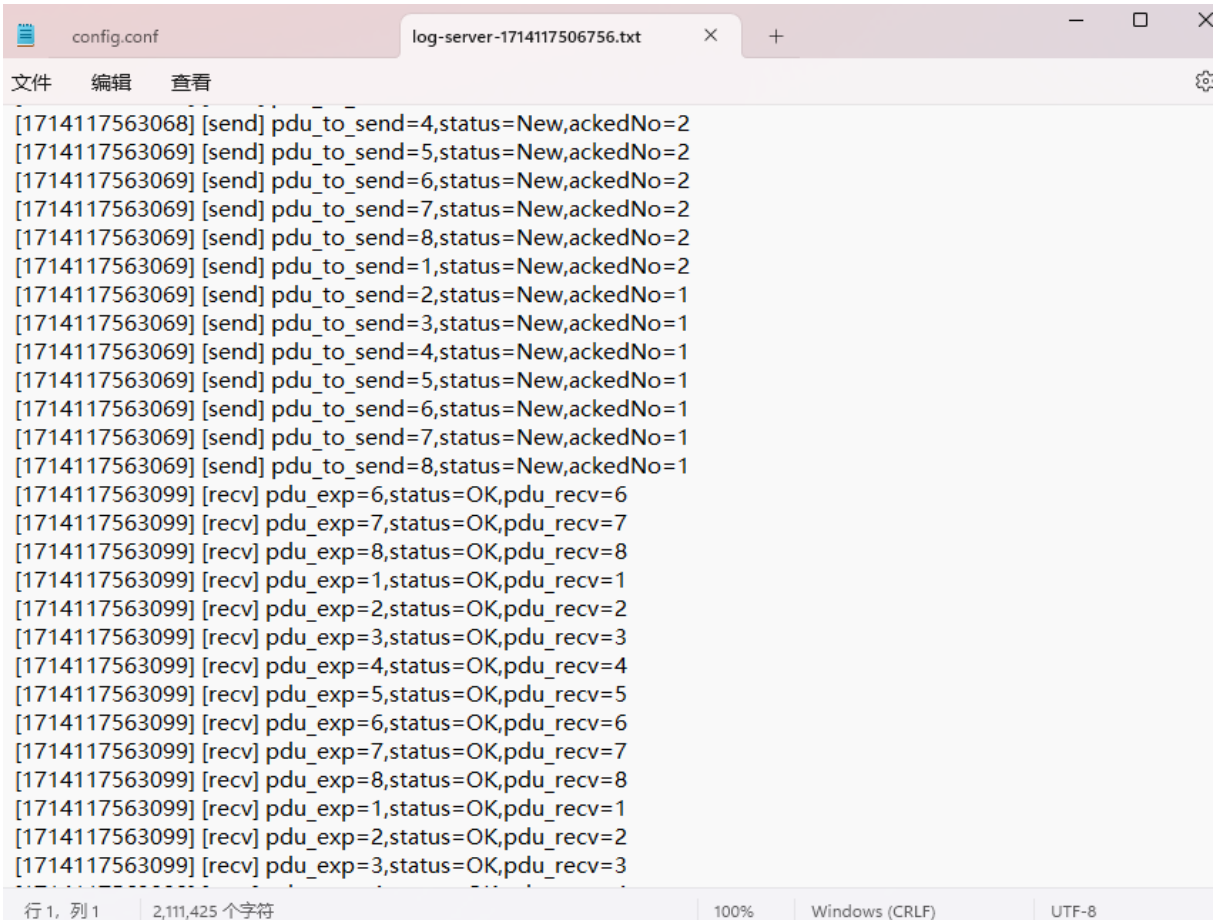
打开两个 GoBackN 进程，在这两个进程中同时输入命令和文件路径，相互传输一个 17.6MB 大小的文件，经过一段时间后，两方都显示“[network] 接收文件完毕”，检查接收到的文件大小，和原文件大小一致，且可以正常打开。


GoBackN测试
Release
此电脑 › 扩展 (F:) › GoBackN测试
在 GoBackN测试 中搜索
新建
排序
查看
详细信息
名称 修改日期 类型 大小
1.zip 2024/4/23 13:52 360压缩 ZIP 文件 17,561 KB
config.conf 2024/4/25 8:01 CONF 文件 1 KB
log.ipynb 2024/4/25 21:36 Jupyter 源文件 2 KB
GoBackN.exe 2024/4/26 15:26 应用程序 77 KB
1714116462413.zip 2024/4/26 15:28 360压缩 ZIP 文件 17,561 KB
log-client-1714116421430.txt 2024/4/26 15:28 文本文档 1,090 KB
log-server-1714116418816.txt 2024/4/26 15:28 文本文档 1,011 KB
log-server-1714117506756.txt 2024/4/26 15:45 文本文档 0 KB
log-client-1714117508582.txt 2024/4/26 15:45 文本文档 0 KB
[success] 读取配置文件成功
[success] 日志文件创建成功: log-server-1714117506756.txt
我是服务端
1、添加要发送的文件
2、退出程序
请输入您的选项:
[info] 发现新的客户端, 端口为: 9466
1
请输入文件路径: 1.zip
[success] 读取配置文件成功
[success] 日志文件创建成功: log-client-1714117508582.txt
我是客户端
1、添加要发送的文件
2、退出程序
请输入您的选项:
1
请输入文件路径: 1.zip


GoBackN测试
Release
此电脑 › 扩展 (F:) › GoBackN测试
在 GoBackN测试 中搜索
新建
排序
查看
详细信息
名称 修改日期 类型 大小
1.zip 2024/4/23 13:52 360压缩 ZIP 文件 17,561 KB
config.conf 2024/4/25 8:01 CONF 文件 1 KB
log.ipynb 2024/4/25 21:36 Jupyter 源文件 2 KB
GoBackN.exe 2024/4/26 15:26 应用程序 77 KB
1714116462413.zip 2024/4/26 15:28 360压缩 ZIP 文件 17,561 KB
log-client-1714116421430.txt 2024/4/26 15:28 文本文档 1,090 KB
log-server-1714116418816.txt 2024/4/26 15:28 文本文档 1,011 KB
log-server-1714117506756.txt 2024/4/26 15:45 文本文档 0 KB
log-client-1714117508582.txt 2024/4/26 15:45 文本文档 0 KB
1714117561883.zip 2024/4/26 15:46 360压缩 ZIP 文件 17,561 KB
1714117562336.zip 2024/4/26 15:46 360压缩 ZIP 文件 17,561 KB
[physical] 帧丢失
[datalink] 接收到的帧校验错误
[datalink] 超时
[physical] 帧丢失
[datalink] 超时
[physical] 帧丢失
[datalink] 接收到的帧校验错误
[datalink] 超时
[physical] 帧出错
[physical] 帧出错
[physical] 帧出错
[physical] 帧出错
[physical] 帧丢失
[datalink] 超时
[physical] 帧丢失
[datalink] 超时
[physical] 帧出错
[physical] 帧丢失
[datalink] 超时
[physical] 帧丢失
[datalink] 超时
[physical] 帧丢失
[datalink] 接收到的帧校验错误
[datalink] 接收到的帧校验错误
[network] 接收文件完毕
[datalink] 超时
[physical] 帧丢失
[datalink] 超时
[physical] 帧出错
[physical] 帧丢失
[datalink] 接收到的帧校验错误
[datalink] 接收到的帧校验错误
[datalink] 超时
[physical] 帧丢失
[datalink] 超时
[physical] 帧丢失
[datalink] 接收到的帧校验错误
[datalink] 超时
[physical] 帧丢失
[datalink] 超时
[physical] 帧丢失
[physical] 帧丢失
[datalink] 超时
[physical] 帧出错
[physical] 帧出错
[datalink] 超时
[datalink] 接收到的帧校验错误
[datalink] 接收到的帧校验错误
[datalink] 接收到的帧校验错误
[datalink] 接收到的帧校验错误
[datalink] 接收到的帧校验错误
[datalink] 接收到的帧校验错误
[datalink] 接收到的帧校验错误
[datalink] 接收到的帧校验错误
[datalink] 接收到的帧校验错误
[datalink] 接收到的帧校验错误
[datalink] 接收到的帧校验错误
[network] 接收文件完毕

## 4. 日志文件检查

打开日志文件，发现内容正常且符合预期。




# 六. Performance and Analysis

这部分我会通过控制变量法来探究配置文件各配置项对传输文件的影响，被传输文件的大小为 17.6MB。

## 1. 帧中数据字段最大长度：DataSize

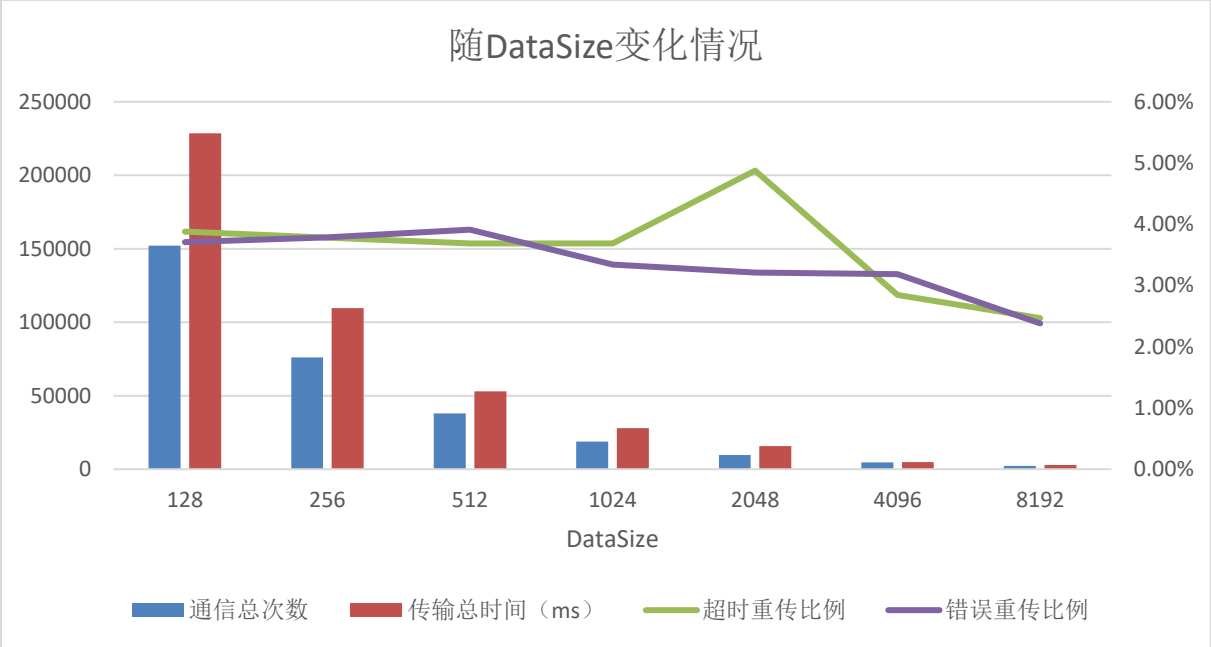
其他参数如下：

- UDPPort=40964
- ErrorRate=100
- LostRate=100
- SWSize=7
- InitSeqNo=1

- Timeout=100

更改 DataSize 的大小为 128B、256B、512B、1024B、2048B、4096B、8192B，测试结果如下：

| DataSize（B） | 通信总次数 | PDU总数量 | 超时重传次数 | 错误重传次数 | 传输总时间（ms） | 超时重传比例 | 错误重传比例 |
|---|---|---|---|---|---|---|---|
| 128 | 152031 | 140487 | 5903 | 5641 | 228497 | 3.88% | 3.71% |
| 256 | 75993 | 70245 | 2873 | 2876 | 109704 | 3.78% | 3.78% |
| 512 | 38013 | 35124 | 1402 | 1487 | 53052 | 3.69% | 3.91% |
| 1024 | 18890 | 17563 | 696 | 631 | 27949 | 3.68% | 3.34% |
| 2048 | 9556 | 8783 | 466 | 307 | 15809 | 4.88% | 3.21% |
| 4096 | 4675 | 4393 | 133 | 149 | 4917 | 2.84% | 3.19% |
| 8192 | 2310 | 2198 | 57 | 55 | 2804 | 2.47% | 2.38% |




可以看到，随着帧的数据字段长度的增大，通信总次数和传输总时间都呈现下降趋势。超时重传比例和错误重传比例基本保持不变。在实际通信过程中，过长的帧会导致帧冲突加剧，进而导致传输速率降低。

## 2. 滑动窗口大小：SWSize

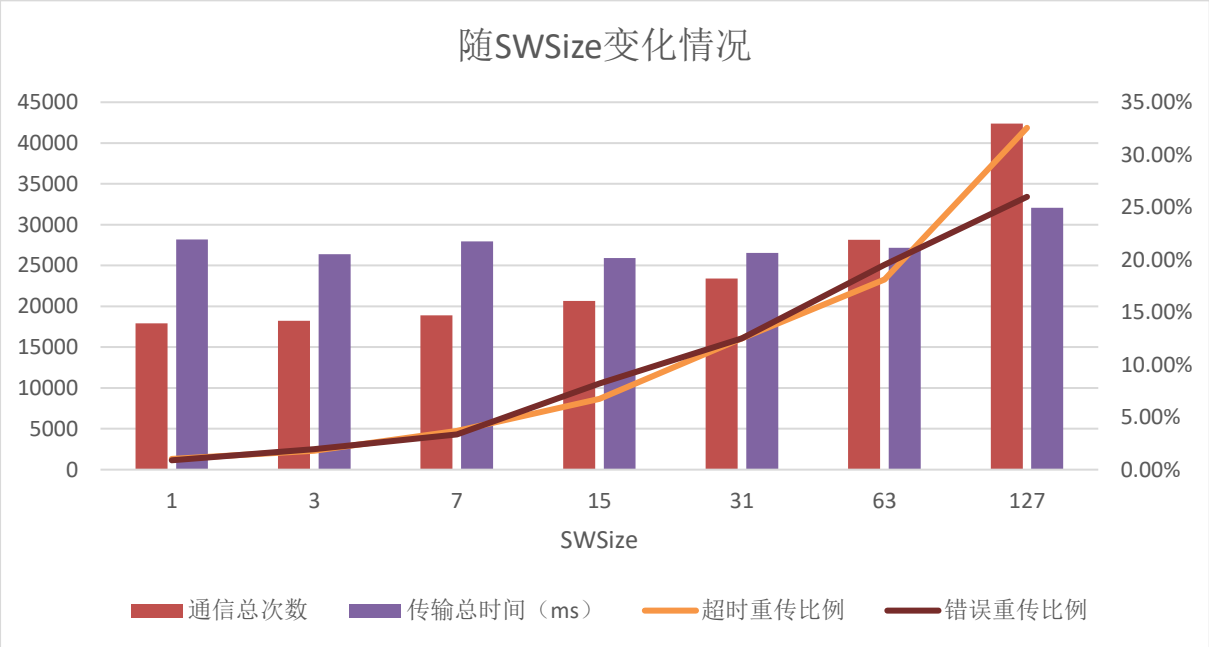
其他参数如下：

- UDPPort=40964
- DataSize=1024
- ErrorRate=100

- LostRate=100
- InitSeqNo=1
- Timeout=100

更改 SWSize 的大小为 1、3、7、15、31、63、127，测试结果如下：

| SWSize | 通信总次数 | PDU总数量 | 超时重传次数 | 错误重传次数 | 传输总时间（ms） | 超时重传比例 | 错误重传比例 |
|---|---|---|---|---|---|---|---|
| 1 | 17906 | 17563 | 183 | 160 | 28195 | 1.02% | 0.89% |
| 3 | 18252 | 17563 | 331 | 358 | 26378 | 1.81% | 1.96% |
| 7 | 18890 | 17563 | 696 | 631 | 27949 | 3.68% | 3.34% |
| 15 | 20648 | 17563 | 1394 | 1691 | 25906 | 6.75% | 8.19% |
| 31 | 23423 | 17563 | 2929 | 2931 | 26555 | 12.50% | 12.51% |
| 63 | 28161 | 17563 | 5104 | 5494 | 27186 | 18.12% | 19.51% |
| 127 | 42374 | 17563 | 13798 | 11013 | 32084 | 32.56% | 25.99% |




可以看到，滑动窗口的大小不是越大越好，而是要根据根据网络带宽和延迟进行选择。较高的带宽和较小的延迟允许选择较大的窗口大小，以提高数据传输效率；而较低的带宽和较大的延迟则需要选择较小的窗口大小，以避免过多的数据堆积和丢失。当网络出现拥塞时，发送方应减小窗口大小以降低数据发送速率，以避免进一步加剧拥塞。而当网络负载较轻时，发送方可以增大窗口大小以提高数据传输速率。

在测试中，由于丢包率和错误率较高，为 1%，所以过大的滑动窗口大小会导致重传次数增多，导致传输速率降低；相反，过小的滑动窗口大小导致在等待确认帧时浪费时间，也会降低传输速率。

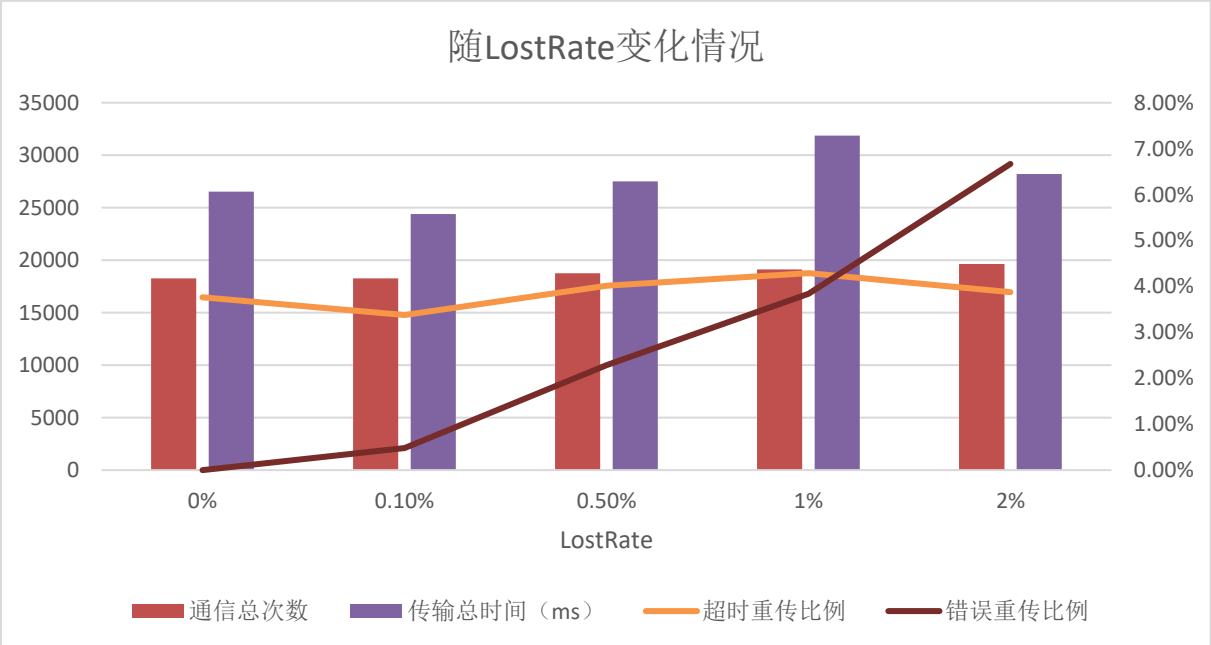
# 3. 丢失率：LostRate

其他参数如下：

- UDPPort=40964
- DataSize=1024
- ErrorRate=100
- SWSize=7
- InitSeqNo=1
- Timeout=100

更改 LostRate 的大小为 0（0%）、50（2%）、100（1%）、200（0.5%）、1000（0.1%），测试结果如下：

| LostRate | 通信总次数 | PDU总数量 | 超时重传次数 | 错误重传次数 | 传输总时间（ms） | 超时重传比例 | 错误重传比例 |
|---|---|---|---|---|---|---|---|
| 0% | 18252 | 17563 | 0 | 689 | 495 | 0.00% | 3.77% |
| 0.10% | 18221 | 17563 | 63 | 595 | 2663 | 0.35% | 3.27% |
| 0.50% | 18720 | 17563 | 399 | 758 | 14697 | 2.13% | 4.05% |
| 1% | 19026 | 17563 | 666 | 797 | 25487 | 3.50% | 4.19% |
| 2% | 19672 | 17563 | 1442 | 667 | 53824 | 7.33% | 3.39% |




可以看到，随着丢失率的升高，传输文件的速率降低，耗时大幅增加，这主要是用于超时时间太大导致的。

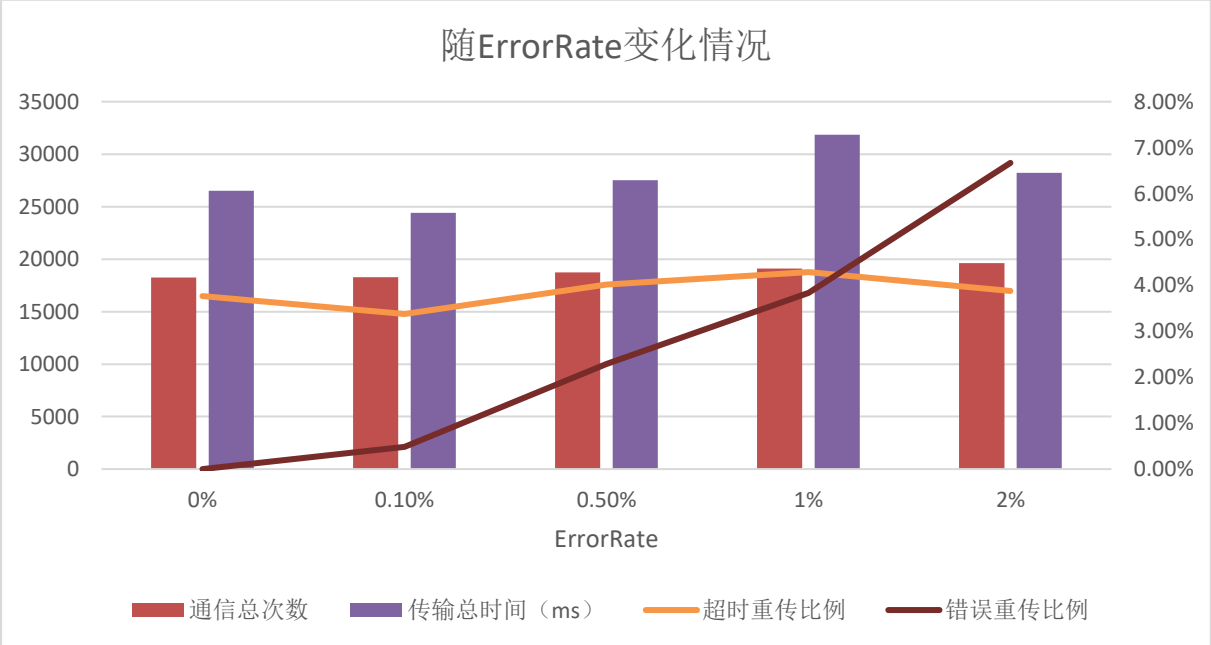
# 4. 错误率：ErrorRate

其他参数如下：

- UDPPort=40964
- DataSize=1024
- LostRate=100
- SWSize=7
- InitSeqNo=1
- Timeout=100

更改 ErrorRate 的大小为 0（0%）、50（2%）、100（1%）、200（0.5%）、1000（0.1%），测试结果如下：

| ErrorRate | 通信总次数 | PDU总数量 | 超时重传次数 | 错误重传次数 | 传输总时间（ms） | 超时重传比例 | 错误重传比例 |
|---|---|---|---|---|---|---|---|
| 0% | 18250 | 17563 | 687 | 0 | 26527 | 3.76% | 0.00% |
| 0.10% | 18268 | 17563 | 617 | 88 | 24399 | 3.38% | 0.48% |
| 0.50% | 18745 | 17563 | 753 | 429 | 27511 | 4.02% | 2.29% |
| 1% | 19117 | 17563 | 820 | 734 | 31859 | 4.29% | 3.84% |
| 2% | 19633 | 17563 | 761 | 1309 | 28216 | 3.88% | 6.67% |




可以看到，错误率与传输速率的关系不是特别明显，与 LostError 呈现不同的效果。分析原因发现，由于我们对于发生错误的帧不是仅丢弃而是返回 nak，所以不会等待超时，所以错误率越大，传输时间增大地非常不明显。

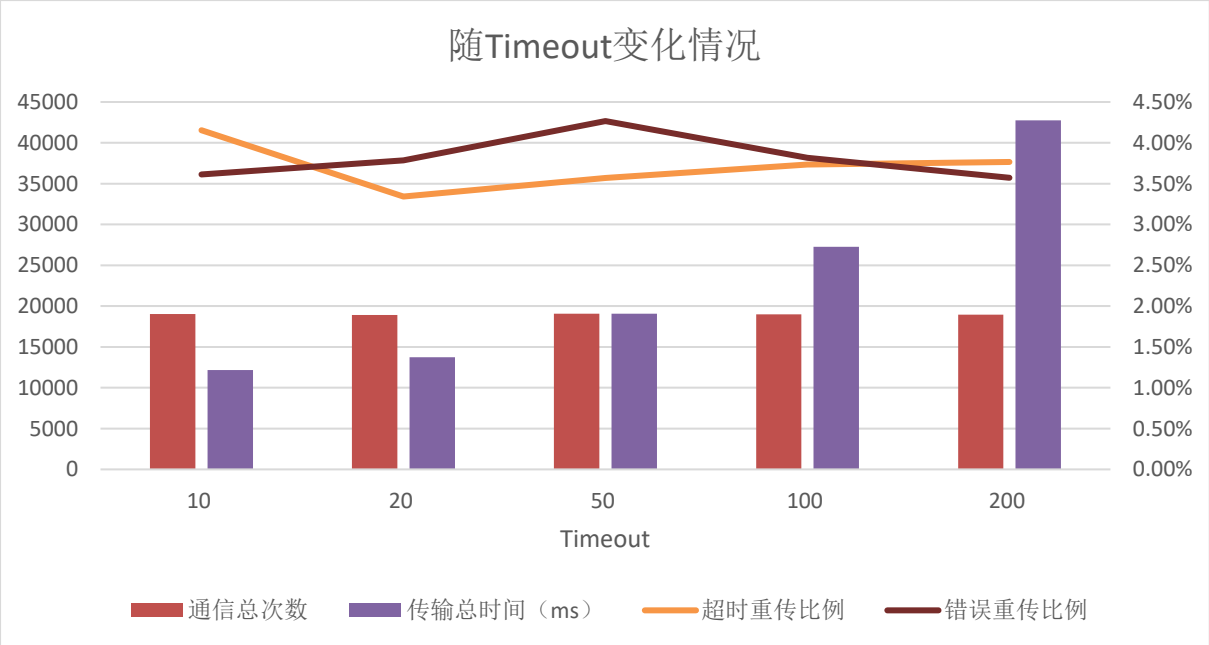
## 5. 超时时间：Timeout

其他参数如下：

- UDPPort=40964
- DataSize=1024
- LostRate=100
- ErrorRate=100
- SWSize=7
- InitSeqNo=1

更改 Timeout 的大小为 10、20、50、100、500，测试结果如下：

| Timeout（ms） | 通信总次数 | PDU总数量 | 超时重传次数 | 错误重传次数 | 传输总时间（ms） | 超时重传比例 | 错误重传比例 |
|---|---|---|---|---|---|---|---|
| 10 | 19042 | 17563 | 791 | 688 | 12159 | 4.15% | 3.61% |
| 20 | 18911 | 17563 | 632 | 716 | 13718 | 3.34% | 3.79% |
| 50 | 19056 | 17563 | 680 | 813 | 19053 | 3.57% | 4.27% |
| 100 | 18997 | 17563 | 709 | 725 | 27246 | 3.73% | 3.82% |
| 200 | 18954 | 17563 | 714 | 677 | 42749 | 3.77% | 3.57% |




可以看到，超时时间越大，传输速率越慢，传输时间越长，所以我们的超时时间不能太大；但是，当超时时间很小时，会导致超时重传比例增大，浪费带宽，甚至导致传输速率减慢，实际中需要进行折中。

# 七. Summary or Conclusions

本次实验我主要研究和分析了 GoBackN 协议的实现和各参数的影响。从上面的分析可以看出来，影响 GoBackN 传输效率的因素有很多，而大多数因素都对传输速率的影响都不是线性的，需要我们根据实际情况进行均衡调节。

在用代码实现 GoBackN 时，我最开始采用的事件队列模式来将一系列事件串联起来，但这样做了之后发现和实际情况相差很远。最后我改为了通过模拟实际传输过程的方式来编写代码，这样我的代码逻辑清晰完整。同时，也加深了对课程所学内容的理解。

在实现定时器时，最开始我采用了 Windows 的系统调用 SetTimer。但是发现它的精度很低，导致删除定时器并不总是成功，出现了很多莫名其妙的超时重传帧。最后改为通过线程来手动实现定时器，同时由于我将定时器封装在了 Timer 类中，其他用到 Timer 地方代码不用更改，只需要重新实现 Timer 类内方法即可。这体现了接口的好处。

在编写程序的时候，我采用了面向对象和组合的编程方式，将程序分为物理层、数据链路层、网络层，并确保上一层只会调用下一层提供的功能，具有很好的分层结构。同时，自己实现了具有编号功能的定时器和线程安全的队列，方便了多线程之间相互操作数据。

在使用多线程时，我也根据条件决定线程是否休眠一段时间，从而降低系统开销。

总之，本程序最大的特色就是全面模拟实际网络通信过程，并很好地遵循了分层的设计思想，上层只能调用下层所提供的服务，不能跨层调用。同时使用了组合的方式来实现分层，即在初始化上层类时传入下层类对象，这样就可以在上层类中使用下层类所提供的方法。

# 八. References

- 课程课件：数据链路层 PPT
- 课程教材：《计算机网络》

# 九. Comments

本次实验从零开始来实现 GoBackN 协议，确实让我深入理解数据链路层的工作，完成实验后，对 PPT 中的内容有更好的理解。同时，本次实验让我思考了课堂上所没有讲解的内容，如滑动窗口大小的对传输速率的影响。在编写代码时，我深入践行了网络中最重要的分层结构，确实让代码更加清晰，这是无法通过课本上干瘪的文字所能体会到的。