

# 项目一报告

姓名	学号	分工	贡献率
蔡与望	2020010801024	程序编写调试、报告撰写	40%
党一琨	2020140903010	共享信道仿真实验、程序调试	20%
郭培琪	2020030701003	拓扑与帧结构设计	20%
陶砚青	2020040401013	共享信道仿真实验、程序调试	20%

- [项目一报告](#)

- [二、整体架构](#)
  - [1.1 网元设计](#)
  - [1.2 帧设计](#)
  - [1.3 源码总览](#)
    - [1.3.1 目录结构](#)
    - [1.3.2 网元层模块设计](#)
- [二、物理层](#)
  - [2.1 连接网元](#)
  - [2.2 模拟误码](#)
  - [2.3 模拟MTU](#)
  - [2.4 模拟冗余](#)
- [三、应用层](#)
  - [3.1 决定网元模式](#)
  - [3.2 信息I/O](#)
  - [3.3 文本与图片的编解码](#)
    - [3.3.1 文本编解码](#)
    - [3.3.2 图片编解码](#)
    - [3.3.3 代码实现](#)
  - [3.4 代码框架](#)
  - [3.5 阶段一测试](#)
    - [3.5.1 随机数产生与应答](#)
    - [3.5.2 心得与反思](#)
- [四、主机网络层](#)
  - [4.1 帧同步与定位](#)
    - [4.1.1 基本原理](#)
    - [4.1.2 代码实现](#)
  - [4.2 地址读写](#)
    - [4.2.1 取16位的原因](#)
    - [4.2.2 代码实现](#)
  - [4.3 序号读写](#)
    - [4.3.1 取8位的原因](#)
    - [4.3.2 代码实现](#)

- [4.4 差错检测](#)

- [4.4.1 基本原理](#)
- [4.4.2 代码实现](#)

- [4.5 差错控制](#)

- [4.5.1 基本原理](#)
- [4.5.2 采用停等协议的原因](#)
- [4.5.3 Keepalive机制](#)
- [4.5.4 代码实现](#)

- [4.6 流量控制](#)

- [4.6.1 基本原理](#)
- [4.6.2 代码实现](#)

- [4.7 代码框架](#)

- [4.8 阶段二测试](#)

- [4.8.1 Unicode字符I/O](#)
- [4.8.2 文本传输容量](#)
- [4.8.3 图片I/O](#)
- [4.8.4 检错与差控](#)
- [4.8.5 Keepalive机制](#)
- [4.8.6 各项性能指标](#)
  - [4.8.6.1 吞吐量](#)
  - [4.8.6.2 误码承受能力](#)
  - [4.8.6.3 传输延时](#)
- [4.8.7 心得与反思](#)

- [五、交换机网络层](#)

- [5.1 维护端口地址表](#)

- [5.2 单播广播支持](#)

- [5.3 代码框架](#)

- [5.4 阶段三测试](#)

- [5.4.1 单播支持](#)
- [5.4.2 广播支持](#)
- [5.4.3 各项性能指标](#)
  - [5.4.3.1 吞吐量](#)
  - [5.4.3.2 误码承受能力](#)
  - [5.4.3.3 传输延时](#)
- [5.4.4 心得与反思](#)

- [六、路由器网络层](#)

- [6.1 路由表](#)

- [6.1.1 基本原理](#)
- [6.1.2 代码实现](#)

- [6.2 数据定向投递](#)

- [6.2.1 基本原理](#)
- [6.2.2 代码实现](#)

- [6.3 代码框架](#)

- [6.4 阶段四调试](#)

- [6.4.1 路由表计算](#)
- [6.4.2 跨网单播](#)
- [6.4.3 局域网广播](#)
- [6.4.4 各项性能指标](#)
  - [6.4.4.1 吞吐量](#)
  - [6.4.4.2 误码承受能力](#)
  - [6.4.4.3 传输延时](#)
- [6.4.5 心得与反思](#)

- [七、总结](#)

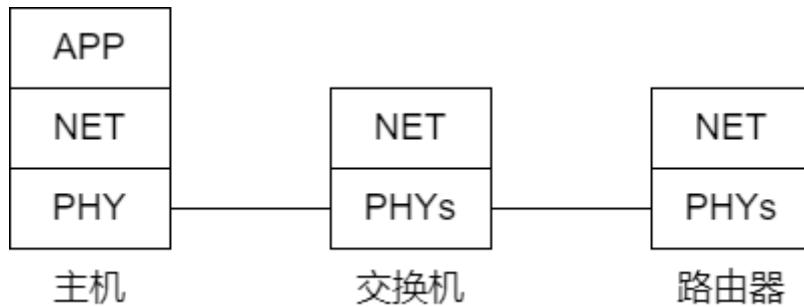
- [7.1 指导书需求完成情况](#)
- [7.2 项目心得体会](#)

- [八、附录](#)

- [8.1 编写与运行环境](#)
- [8.2 项目开源地址](#)

# 一、整体架构

## 1.1 网元设计



在我们的网络模型中，共有三种网元，它们分别是：**主机、交换机、路由器**。

其中，主机的网元分为三层，分别是**应用层、网络层、物理层**；而路由器与交换机，由于不需要与用户进行I/O操作，只设**网络层、物理层**。各层间通过套接字进行通信。

网元各层的主要功能如下：

- 物理层
  - 连接网元
  - 模拟误码
  - 模拟MTU
  - 模拟冗余
- 主机应用层
  - 决定网元模式
  - 信息I/O
  - 文本与图片的编解码
- 主机网络层
  - 帧同步与定位
  - 地址读写
  - 序号读写
  - 差错检测
  - 差错控制
  - 流量控制
- 交换机网络层
  - 维护端口地址表
  - 单播广播支持
- 路由器网络层
  - 维护路由表
  - 数据定向投递

## 1.2 帧设计

在我们的设计中，单次发送的信息将被分为多个帧，网元间以帧为单位交换信息；帧内除了用户发送的数据，还有网络层添加的各种控制信息，用于实现差控、流控、判收等功能。



上图是我们组设计的帧结构。它包括以下这几部分：

- 帧头、帧尾 (8位)
- 源地址、目的地址 (16位)
- 帧序号 (8位)
- 数据 (不多于32位)
- CRC校验码 (16位)
- 冗余 (位数不等，帧同步的副产物)

具体每一部分的功能、原理与实现见第3章。

## 1.3 源码总览

程序使用 `Python` 编写，环境见附录8.1节。

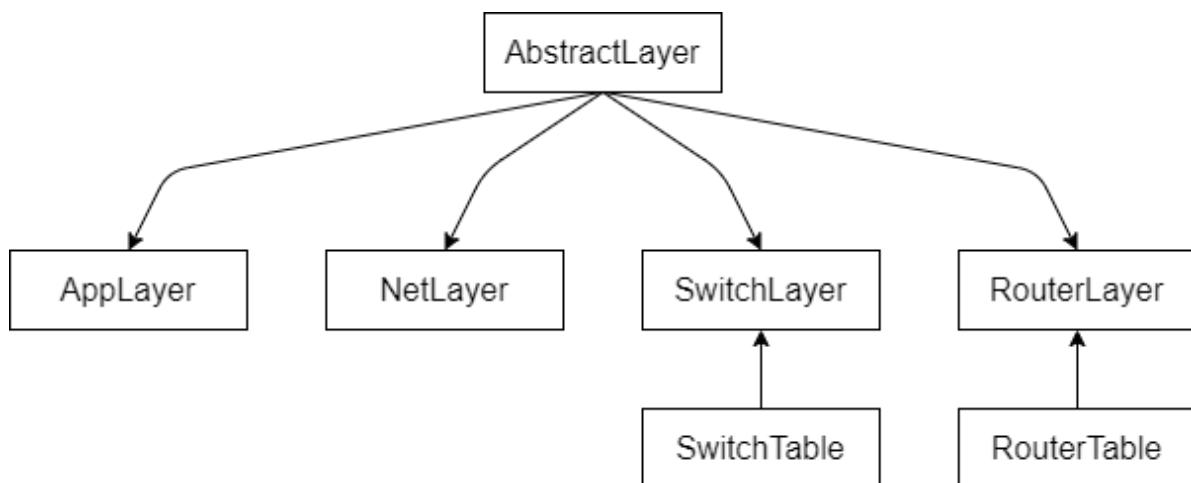
### 1.3.1 目录结构

```

1 Project
2   └ bin      可执行文件
3   └ config   配置与备份文件
4   └ resource 资源文件
5   └ src       源代码文件
6     └ layer   网元层模块
7     └ utils   工具模块

```

### 1.3.2 网元层模块设计



虽然网元的各层都肩负着不同的职责，但它们身为数据的传递者，必定存在一些共性，比如都有确定的端口号和套接字、都可以收发消息等。我们将这些共性抽象出来，创建了**抽象层** (`AbstractLayer`)。它可以实现最简单的收发与 `select` 功能，其收发API只对子类开放，供各层实现进一步分化。在此基础上：

- **主机应用层** (AppLayer) 增加了与用户、主机网络层交互的功能；
- **主机网络层** (NetLayer) 增加了与主机应用层、主机物理层交互的功能；
- **交换机网络层** (SwitchLayer) 增加了与主机物理层交互的功能；同时继承**端口地址表** (SwitchTable)，可以对其进行CRUD操作。
- **路由器网络层** (RouterLayer) 增加了与物理层交互的功能；同时继承**路由表** (RouterTable)，可以对其进行查找与合并。

## 二、物理层

在整个网元中，物理层主要有四种功能：

- 连接网元
- 模拟误码
- 模拟MTU
- 模拟冗余

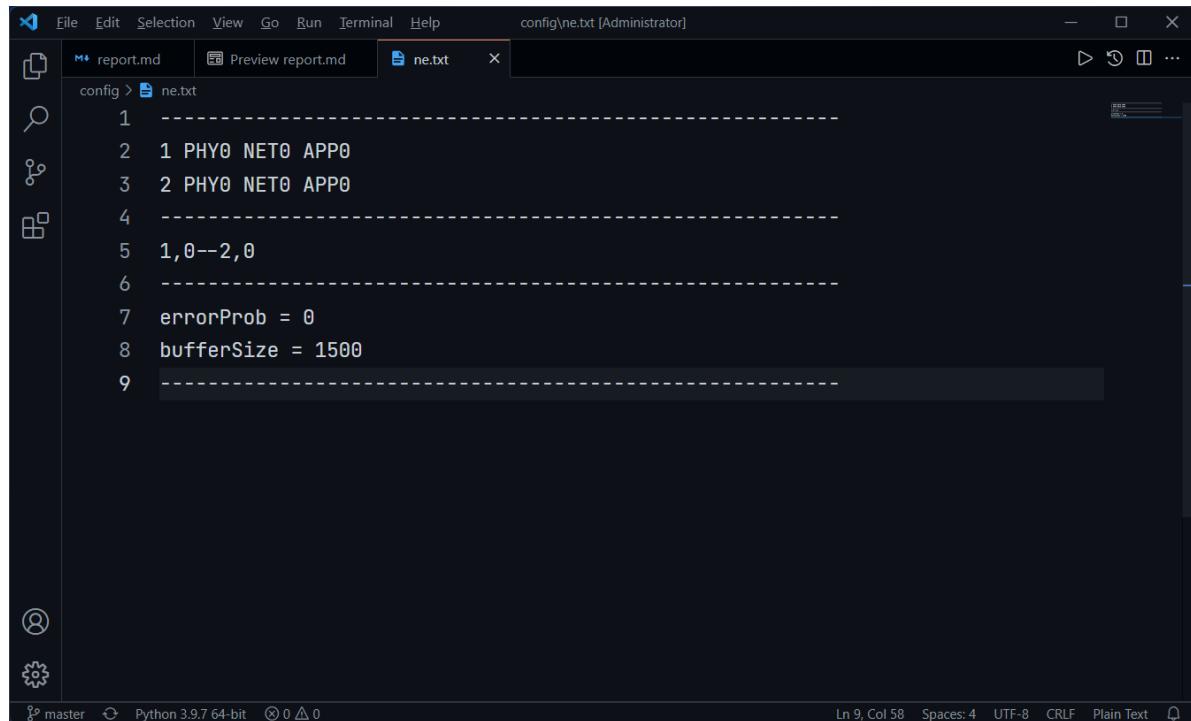
下面将分别展示这四种功能。

### 2.1 连接网元

在[网元设计图](#)中，我们可以看到不同网元间有一根根线将其互相连接。这些网元间的连接——也就是收发消息的桥梁——是物理层模拟软件提供的。在本项目中，我们将物理层设定为只能传输**bit数组**，即任意两网元在底层上只能传输0和1的比特流。

而为了连接我们拓扑中的各个网元，我们有时将打开十几个物理层软件，而一个个去手动配置它们的参数显然是不现实的。所以我们根据各阶段的拓扑图，编写了相应的配置文件 `ne.txt`；在物理层启动时，它可以告诉每个物理层各自的参数。

如下图，我们实现了阶段二的，主机1与主机2相连的拓扑。



```

File Edit Selection View Go Run Terminal Help config\ne.txt [Administrator]
report.md Preview report.md ne.txt

config > ne.txt
1 -----
2 1 PHY0 NET0 APP0
3 2 PHY0 NET0 APP0
4 -----
5 1,0--2,0
6 -----
7 errorProb = 0
8 bufferSize = 1500
9 -----

```

The screenshot shows a terminal window titled "config\ne.txt [Administrator]". The window contains a configuration file named "ne.txt" with the following content:

```

1 -----
2 1 PHY0 NET0 APP0
3 2 PHY0 NET0 APP0
4 -----
5 1,0--2,0
6 -----
7 errorProb = 0
8 bufferSize = 1500
9 -----

```

The terminal interface includes standard navigation keys like backspace, arrow keys, and enter, along with tabs for file selection and help.

除此之外，我们还编写了批处理文件 `OneTouchToGo-stageX.bat` 和 `OneTouchToClose.bat`，分别可以一键启动和关闭各阶段拓扑内的所有网元；这也大大降低了我们调试的难度。

下面是一键启动的原理，这套配置能够一键启动阶段二的网元拓扑。

```

File Edit Selection View Go Run Terminal Help
src\start.py (Administrator)
... start.py ...
src > start.py > ...
1 import sys
2
3 from utils.io import cover_batch, cover_phynum, cover_r
4
5 if __name__ == "__main__":
6     # 读取阶段。
7     if len(sys.argv) != 2:
8         print("[Error] Stage required")
9         exit(-1)
10    stage = sys.argv[1]
11
12    # 写入阶段配置。
13    cover_batch(stage) 2. start.py覆盖旧配置
14    cover_phynum(stage)  运行一键启动bat文件
15    cover_ne(stage)
16
17    # 运行批处理文件。
18    run_batch()
19

1 cd %~dp0
2 start python ..\src\start.py 2
1. 入口bat文件指定当前所处阶段，运行start.py

batch.bat
config > batch.bat
1 @REM ----- cd to /config 3. 一键启动bat文件根据新配置启动所有网元
2 cd %~dp0
3 @REM ----- Host 1
4 start python ..\src\app.py 1
5 start python ..\src\net.py 1
6 start ..\bin\phy.exe 1 PHY 0
7 @REM ----- Host 2
8 start python ..\src\app.py 2
9 start python ..\src\net.py 2
10 start ..\bin\phy.exe 2 PHY 0

```

1. 入口bat文件指定当前所处阶段，运行start.py

2. start.py覆盖旧配置

3. 一键启动bat文件根据新配置启动所有网元

## 2.2 模拟误码

对误码的承受能力，是衡量一个通信系统性能的重要指标。物理层模拟软件可以轻松地提供对误码模拟的支持，只需要在主窗GUI内更改误码率即可。

通过网上查阅资料、小组讨论，以及项目四调制解调的经验，我们依误码率将信道划分为三个等级：

误码率（十万分之）	信道环境
0-500	优质
500-2000	一般
>2000	恶劣

## 2.3 模拟MTU

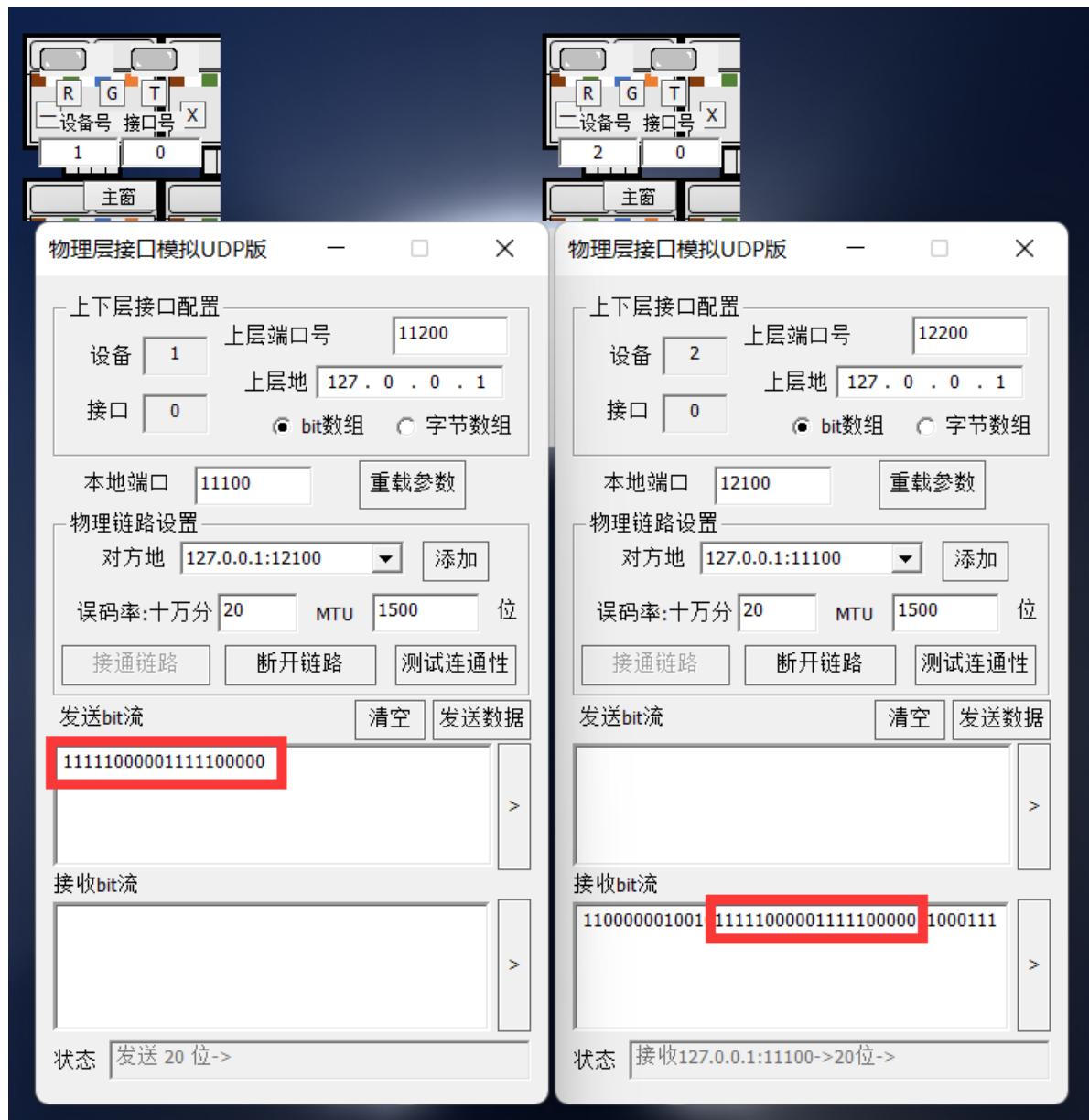
MTU (Maximum Transmission Unit)，最大传输单元，也即单次允许传输的最大字节数。对于当下广泛使用的以太网标准来说，这个值为1500字节。

但实际上，我们的帧只占100余位左右，远远低于这一限制；所以我们组在此标准的基础上进行了缩小，即设物理层模拟软件的MTU为1500位，而非1500字节。（理论上，150位也是足够充裕的。）

## 2.4 模拟冗余

当我们在真实的信道中传输数据时，接收端得到的只有01比特流，而并不知道帧的起始在哪里；这就需要通信双方约定协议来解决定位问题。

为了模拟这一现象，物理层模拟软件会在传输的信息前后各添加若干比特的冗余位，如下图所示。



### 三、应用层

在整个网元中，应用层主要有三种功能：

- 决定网元模式
- 信息I/O
- 文本与图片的编解码

下面将分别展示这三种功能。

#### 3.1 决定网元模式

由于多线程代码的编写具有一定难度，本项目中的网元只采用单线程、半双工模式，即同时只能处于接收、单播、广播中的一种。这一选择将通过用户手动输入来激活，然后应用层负责将用户的选择通知到下层。

#### 3.2 信息I/O

应用层作为与用户实际交互的门户，它不仅负责向用户呈现出完整、合理的信息（Output），还负责从用户处获取正确、可传输的信息（Input）。

为了操作的便利，本项目中将使用 `str` 类型，进行绝大部分信息的操作。相应地，信息的I/O只需要调用 `input()` 和 `print()` 就能够实现，唯一的注意点就是需要在用户输入时，判断输入的合理性。

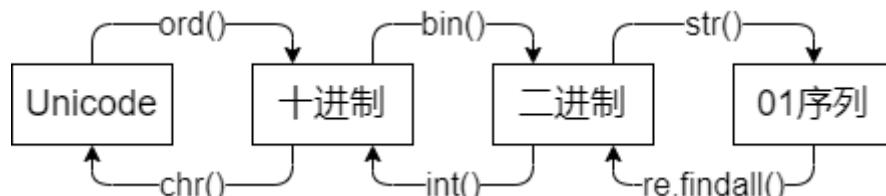
这些行为虽然只是用户在终端进行的，并不涉及套接字，但我们仍然将其封装在了 AppLayer 类内，意图在和“与网络层交互”的代码形成呼应，强调应用层也有与其“上层”（用户）交互的功能。

## 3.3 文本与图片的编解码

由于物理层模拟软件只能传输01比特流，所以我们需要设计一套编解码方案，建立起文本/图片与01比特流的映射规则。

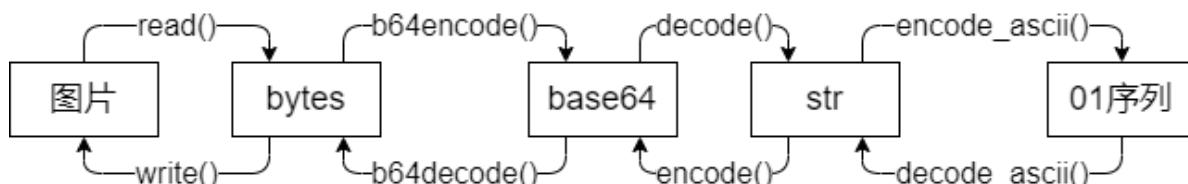
### 3.3.1 文本编解码

由于项目需要提供对中文I/O的支持，所以显然ASCII码无法满足项目的需求，而是需要针对Unicode字符设计编解码方案，具体流程如下：



### 3.3.2 图片编解码

我们采用**base64编解码方案**，这是一种常见的图片编解码方案。为了加快运行速度、提高代码简洁性，我们直接调用内置库 base64 的编解码函数进行图片操作，具体流程如下。



其中的 `encode_ascii()` 和 `decode_ascii()` 与 [Unicode编解码方案](#)类似，只是单次操作位数略有不同。

### 3.3.3 代码实现

`utils/coding.py` 提供了较为丰富的编解码API，简述如下：

- `dec_to_bin()`：将十进制整型数转化为01字符串；
- `bin_to_dec()`：将01字符串转化为十进制整型数；
- `string_to_bits()`：将01字符串转化为01比特流；
- `bits_to_string()`：将01比特流转化为01字符串；
- `encode_ascii()`：将Unicode字符串编码为01字符串；
- `decode_ascii()`：将01字符串解码为Unicode字符串。
- `encode_unicode()`：将ASCII字符串编码为01字符串；
- `decode_unicode()`：将01字符串解码为ASCII字符串。
- `encode_pic()`：将图片编码为01字符串；
- `decode_pic()`：将01字符串解码为图片。

## 3.4 代码框架

```
1 while True:
2     mode = get_mode_from_user()
3
4     if mode == QUIT:
5         break
```

```
7 elif mode == RECV:  
8     receive_msgtype_from_net()  
9     receive_message_from_net()  
10    show_message_to_user()  
11    continue  
12  
13 else:  
14     if mode == UNICAST:  
15         get_dst_from_user()  
16         send_dst_to_net()  
17  
18         get_msgtype_from_user()  
19         send_msgtype_to_net()  
20  
21         get_message_from_user()  
22         send_message_to_net()
```

## 3.5 阶段一测试

### 3.5.1 随机数产生与应答

为了给接下来的阶段做铺垫，我们先编写了两个简单应用层，对网元工作的模式、网元间通信的方式有了一定的理解。根据项目指导书中的需求，我们创建了一个服务器、一个客户端，下面是我们测试的结果。

## 项目需求:

客户端定时每500ms向服务器发送一个随机整数，范围在1~500之间；

服务器每收到一份数据也同时产生随机整数与收到的数据相加，只有在结果大于100时才会把计算结果返回给客户端，而客户端收到超出100的结果则立即产生一个新的数据，而不是在间隔500ms后。

客户机需要产生20份数据，如果有超过100的结果，总运行时间应接近 $10 - N * 0.5$ ，N为超过100的结果的数量。

```
File Edit Selection View Go Run Terminal Help Project1-MinNE-python\client.py [Administrator] - + x

PROBLEMS DEBUG CONSOLE TERMINAL

PS D:\Codes\uestc-CNTProject\Project1-MinNE-python> python server.py
-----Server-----
1 Recv: 401 Generated: 17
2 Recv: 78 Generated: 446
3 Recv: 207 Generated: 27
4 Recv: 322 Generated: 333
5 Recv: 455 Generated: 32
6 Recv: 51 Generated: 25
7 Recv: 24 Generated: 428
8 Recv: 419 Generated: 262
9 Recv: 238 Generated: 180
10 Recv: 294 Generated: 391
11 Recv: 162 Generated: 213
12 Recv: 298 Generated: 433
13 Recv: 496 Generated: 69
14 Recv: 407 Generated: 163
15 Recv: 280 Generated: 28
16 Recv: 431 Generated: 20
17 Recv: 448 Generated: 321
18 Recv: 367 Generated: 94
19 Recv: 243 Generated: 321
20 Recv: 68 Generated: 372
-----Client-----
1 Send: 401 Recv: 418
2 Send: 78 Recv: 524
3 Send: 207 Recv: 234
4 Send: 322 Recv: 655
5 Send: 455 Recv: 487
6 Send: 51 Timeout
7 Send: 24 Recv: 452
8 Send: 419 Recv: 681
9 Send: 238 Recv: 418
10 Send: 294 Recv: 685
11 Send: 162 Recv: 375
12 Send: 299 Recv: 723
13 Send: 496 Recv: 565
14 Send: 407 Recv: 570
15 Send: 280 Recv: 508
16 Send: 431 Recv: 451
17 Send: 448 Recv: 769
18 Send: 367 Recv: 461
19 Send: 243 Recv: 564
20 Send: 68 Recv: 440
-----Cost 0.5 second(s).-----PS D:\Codes\uestc-CNTProject\Project1-MinNE-python>
```

可以看到，服务器与客户端之间能够进行稳定的通信，客户端通过 `select()` 实现了超时的判断，实际运行时间与预期时间（ $10-N*0.5$ ）相符。

### 3.5.2 心得与反思

通过阶段一的程序，我们了解了控制超时的两种方法：`settimeout()` 和 `select()`，同时也知道如何基于不同事件做出不同的响应，对网元间通信的形式、时序等有了进一步了解。

我们原本想要基于 `PyQt5` 为应用层添加 GUI，方便用户的观测与操作；但由于时间紧张，并且涉及到较多架构的改动，并没有实现。

## 四、主机网络层

在整个网络中，主机网络层的功能最多，主要分为六部分：

- 帧同步与定位
- 地址读写
- 序号读写
- 差错检测
- 差错控制
- 流量控制

下面我们将分别展示这六种功能。

### 4.1 帧同步与定位

如同[2.4节](#)所说，发送端和接收端需要约定协议，让双方能在杂乱的比特流中，找到对方发来的信息。

我们采用**面向位的首尾定界法**进行帧同步与定位。

#### 4.1.1 基本原理

- 发送端：变换，添加帧头帧尾。
  1. 在一帧的首尾加上 `01111110`，以标识帧的始末位置；
  2. 帧内的信息也有可能出现 `01111110` 的序列，所以为了防止接收端把帧内信息误当作帧尾，发送端还要在帧内的每个 `11111` 后面插一个 `0`，以免帧内出现 `01111110` 子序列。
- 接收端：找到帧头，反变换。
  1. 在物理层收到的乱码中，找到帧头 `01111110`，然后把帧头剥落；
  2. 对于接下来出现的每个 `11111` 子序列：
    1. 如果接下来出现的是 `0`，那这个 `0` 肯定是发送端插的，将其删除。
    2. 如果接下来出现的是 `1`，那这就是帧尾 `01111110`。（因为发送方已经保证了帧内不可能出现连续6个 `1`。）

#### 4.1.2 代码实现

我们在 `Frame` 类内添加了两个私有静态方法，实现帧同步与定位功能：

- `__add_locator()`：实现上述发送端的任务，在写入信息时调用；
- `__extract_message()`：实现上述接收端的任务，在读入信息时调用。

具体的代码可以在 `utils/frame.py` 的 `Frame` 类中找到。

### 4.2 地址读写

发送端需要在帧内写入源与目的地址，用于后续交换、路由的实现；接收端需要从帧内读取源与目的地址，知道信息从哪来、是不是给自己的。

我们采用**16位二进制数**来标识地址。

### 4.2.1 取16位的原因

由于本项目的网元间通信只在本机（127.0.0.1）实现，所以只需要封装源与目的地的端口即可。又因为端口范围是0~65535，所以每个端口需要用16位二进制表示。

### 4.2.2 代码实现

发送端只需要使用简单的 `f-string` 字符串拼接，即可把地址写入帧；接收端也只需要用字符串切片，就可以提取地址信息。不再展开叙述。

## 4.3 序号读写

在帧内写入序号可以防止传送时的帧间乱序或重复，对于差错控制协议至关重要。

我们采用**8位二进制数**来标识序号。

### 4.3.1 取8位的原因

项目需求提出，传输数据上限约50个字符；根据[应用层的Unicode编码方案](#)，一个字符会占16位比特，所以单次传输的数据最多约800位比特。

又由于，我们的[帧结构设计](#)中，一帧最多传输32位数据，所以一段消息最多要用25帧，才能传输完毕。

又为了CRC校验码在代码实现上的便利，序号位数需要是4的倍数；于是我们可以选择4位或8位，即0~15或0~255。理论上，4位就已经足够使用；但我们想要为单次发送过程中的每一帧都标上独特序号，加之4位的开销也并不算很大，所以采用了8位序号。

### 4.3.2 代码实现

序号读写与地址读写相似，只需要简单的拼接和切片即可实现。不再展开叙述。

## 4.4 差错检测

差错检测是整个网络层代码中至关重要的一步，它需要检查信息有没有传错，如果出错就要求重传。

我们采用**CRC-16校验码**，只检错、不纠错。

### 4.4.1 基本原理

- 发送端：产生校验码。
  1. 将包含源地址、序号、数据、目的地址这四部分信息拼接为01字符串；
  2. 使用生成式 $x^{16} + x^{15} + x^2 + 1$ ，对其进行模2除法；
  3. 得到的余数即为校验码。
- 接收端：检验校验码。
  1. 提取出源地址、序号、数据、目的地址这四部分信息；
  2. 使用与发送端同样的方法进行模2除法；
  3. 与校验码比较是否相同，相同即验证通过。

根据原理，我们可以得到，该校验码可以检测出：

- 所有两位差错；
- 任意奇数个比特的差错；
- 所有长度不超过16的突发差错。

当突发差错多于16位，其漏检率约为0.003%，而这对于我们88位的帧而言，基本可以忽略不计。所以在此我们认为，帧内的CRC-16校验码可以捕获到所有差错。

## 4.4.2 代码实现

我们在 Frame 类内添加了私有静态方法 `__generate_crc()`，实现了对任意（长度为4的倍数的）01字符串的CRC-16校验码生成。

具体的代码可以在 `utils/frame.py` 的 Frame 类中找到。

## 4.5 差错控制

当接收端通过CRC-16校验码检测到帧内有错误，它就需要以某种方式告诉发送端，这帧错了，需要重传；而发送端也相应地需要等待接收端的回复，以确定是否需要重传。

我们采用**停等协议**来进行差错控制。

### 4.5.1 基本原理

1. 发送端：发送了一帧消息，等待各接收端回复；
2. 接收端：
  1. 如果超时，则什么都不做；
  2. 如果不是给自己的，则什么都不做；
  3. 如果序号重复，则丢弃并回复ACK；
  4. 如果校验失败，则回复NAK；
  5. 如果校验通过，则回复ACK；
3. 发送端：
  1. 如果是单播，则只有收到对方ACK，才能发下一帧；
  2. 如果是广播，则只有回复全是ACK，才能发下一帧；
  3. 否则重传；
4. 回到第1步，直到传完所有帧。

### 4.5.2 采用停等协议的原因

- 可以顺便控制流量：发送端需要等回复，所以不会发得太快；
- 编程难度大大降低：只需要实现简单时序逻辑。

### 4.5.3 Keepalive机制

在差控的同时，我们还引入了Keepalive机制，这一机制将在任意一端连续多次接收超时的时候触发，用于终止无限的等待或重传。这一机制有三个作用：

1. 如果是发送端多次超时：
  1. 如果是由于信道环境恶劣引起的，那么发送端将暂时停止重传，要求用户在信道环境恢复良好时，重新输入信息并发送。（就好像QQ如果断网了，发出去的消息会在几十秒后显示红色叹号，告诉用户网络不好，没发出去。）
  2. 如果是由于目的端口不存在引起的，那么发送端停止发送就能及时止损，让用户重新输入正确的端口。
2. 如果是接收端多次超时：说明发送端可能已经发完了，但自己还在进行无谓的等待，那么 Keepalive机制能够结束等待，准备下一次接收。

### 4.5.4 代码实现

根据4.5.1节所展示的时序，我们可以搭建出双端代码差错控制协议的框架：

- 接收端

```
1 | while True:
```

```
2 receive_from_phy()
3
4 if timeout:
5     continue
6 if too_many_timeout:
7     break
8 if not send_to_me:
9     continue
10 if is_repeated:
11     send_ack_to_phy()
12     continue
13 if not verified:
14     send_nak_to_phy()
15     continue
16
17 accept_message()
18 send_ack_phy()
19
20 break_if_this_is_the_last_frame()
```

- 发送端

```
1 while True:
2     send_to_phy()
3
4     if mode == UNICAST:
5         receive_from_phy()
6         if timeout:
7             pass
8         else:
9             if is_ack:
10                 resend_flag = False
11             else:
12                 resend_flag = True
13
14     else:
15         while True:
16             receive_from_phy()
17             if timeout:
18                 break
19             if is_ack:
20                 resend_flag = False
21             else:
22                 resend_flag = True
23
24     if too_many_timeout:
25         break
26     if not resend_flag:
27         send_cnt += 1
28     if all_frames_are_sent:
29         break
```

## 4.6 流量控制

由于我们不可能做到每两台主机间都连一条链路，所以主机间必然会有转发的中介，同时也必然带来一定的延时。这就要求双方不能发得太快，导致中间的交换机、路由器来不及处理，对方来不及读取……等等后果。

我们采用 `time.sleep()` 函数来进行流量控制。

### 4.6.1 基本原理

如果发送端发的速度过快，那么有可能导致：

- 发送端口来不及发；
- 网络来不及传；
- 接收端口来不及收；
- 接收端来不及处理。
- .....

所以，在调用 `sendto()` 函数前，让程序先睡眠适当的时间，就可以做到：等上一波信息完全发出去之后，再发这一波信息。

### 4.6.2 代码实现

只需要在 `sendto()` 的上一行调用 `time.sleep()` 即可。

这一操作被封装在了 `NetLayer` 类内，具体的代码可以在 `utils/layer.py` 中找到。

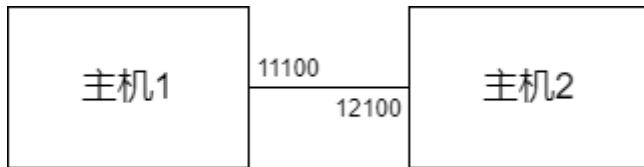
## 4.7 代码框架

将以上所有的功能配合起来，我们就可以得到网络层的代码框架：

```
1 while True:  
2     mode = get_mode_from_app()  
3  
4     if mode == QUIT:  
5         break  
6  
7     elif mode == RECV:  
8         receive_codes_shown_above()  
9         send_msgtype_to_app()  
10        send_message_to_app()  
11        continue  
12  
13    else:  
14        get_dst_from_app()  
15        get_msgtype_from_app()  
16        get_message_from_app()  
17        capsule_all_frames()  
18        send_codes_shown_above()
```

## 4.8 阶段二测试

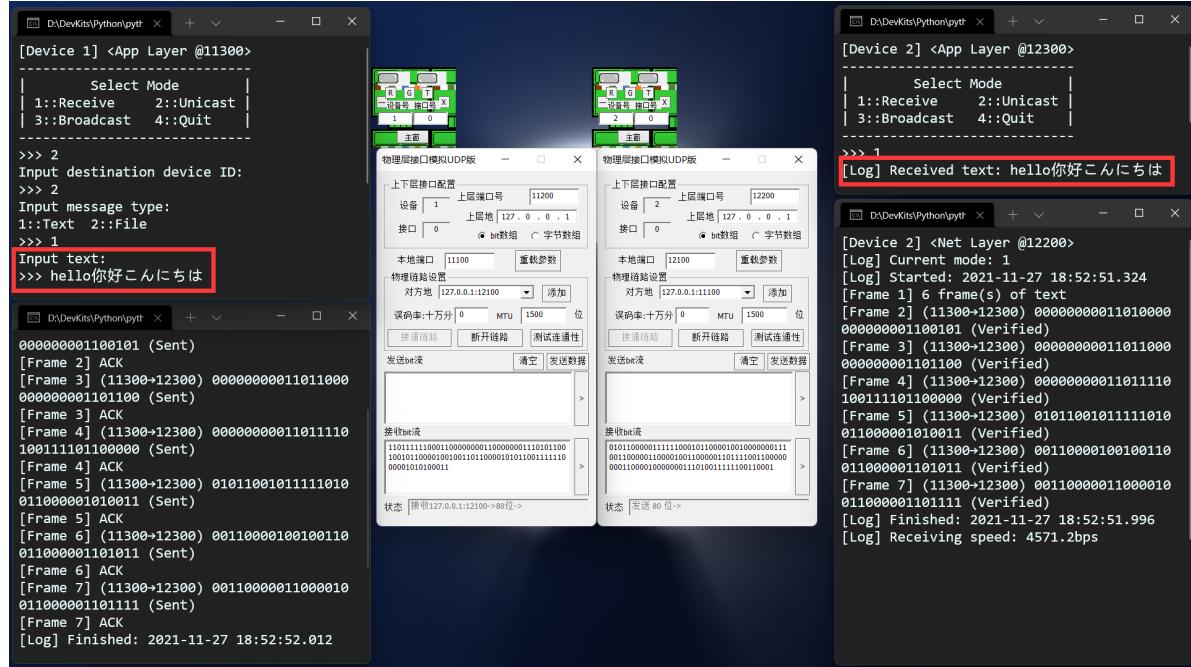
根据上面的代码框架，我们在阶段二写出了应用层与网络层，并使用物理层模拟软件模拟了信道，在两个网元间进行了测试。下面是阶段二的拓扑。



测试结果如下。

#### 4.8.1 Unicode字符I/O

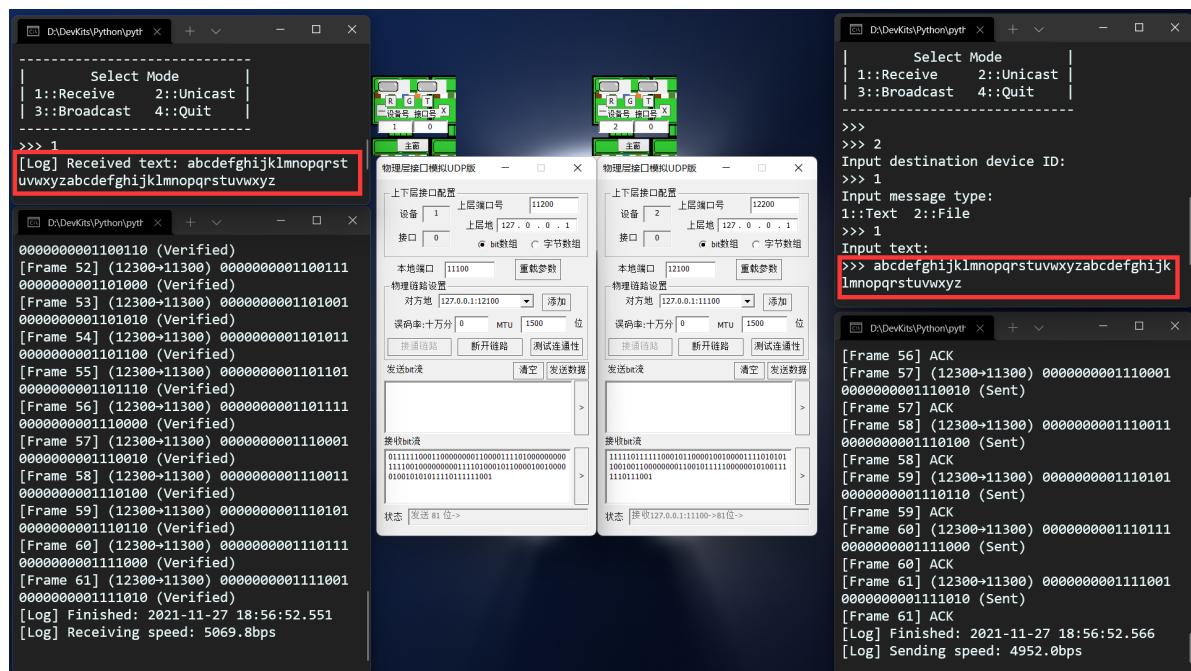
在本测试中，我们从设备1向设备2发送文本“hello你好こんにちは”，这一字符串中同时包含了中、英、日文。



可以看到，双方应用层能够正确编解码，最终在设备2呈现出原本、完整的字符串。由此可以说，我们成功实现了Unicode字符的I/O。

#### 4.8.2 文本传输容量

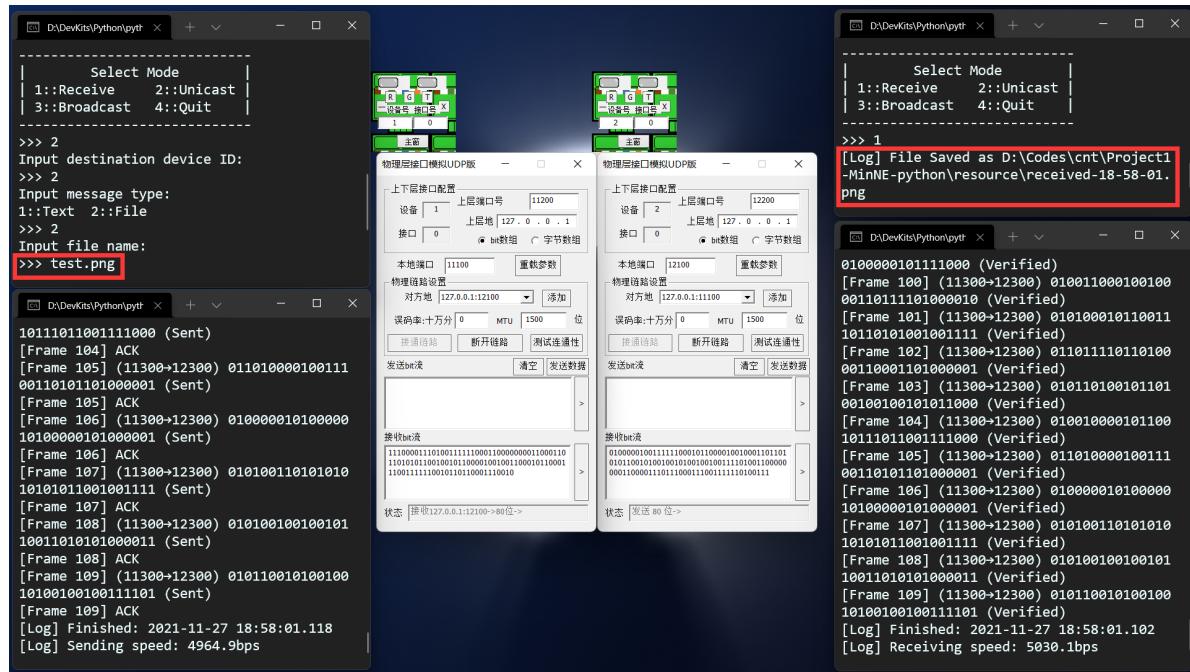
根据指导书要求，双方至少需要支持50个中英文字符的传输。于是在本测试中，我们从设备2向设备1发送文本“abcd...zabcd...z”，即两遍字母表（52个英文字符）。



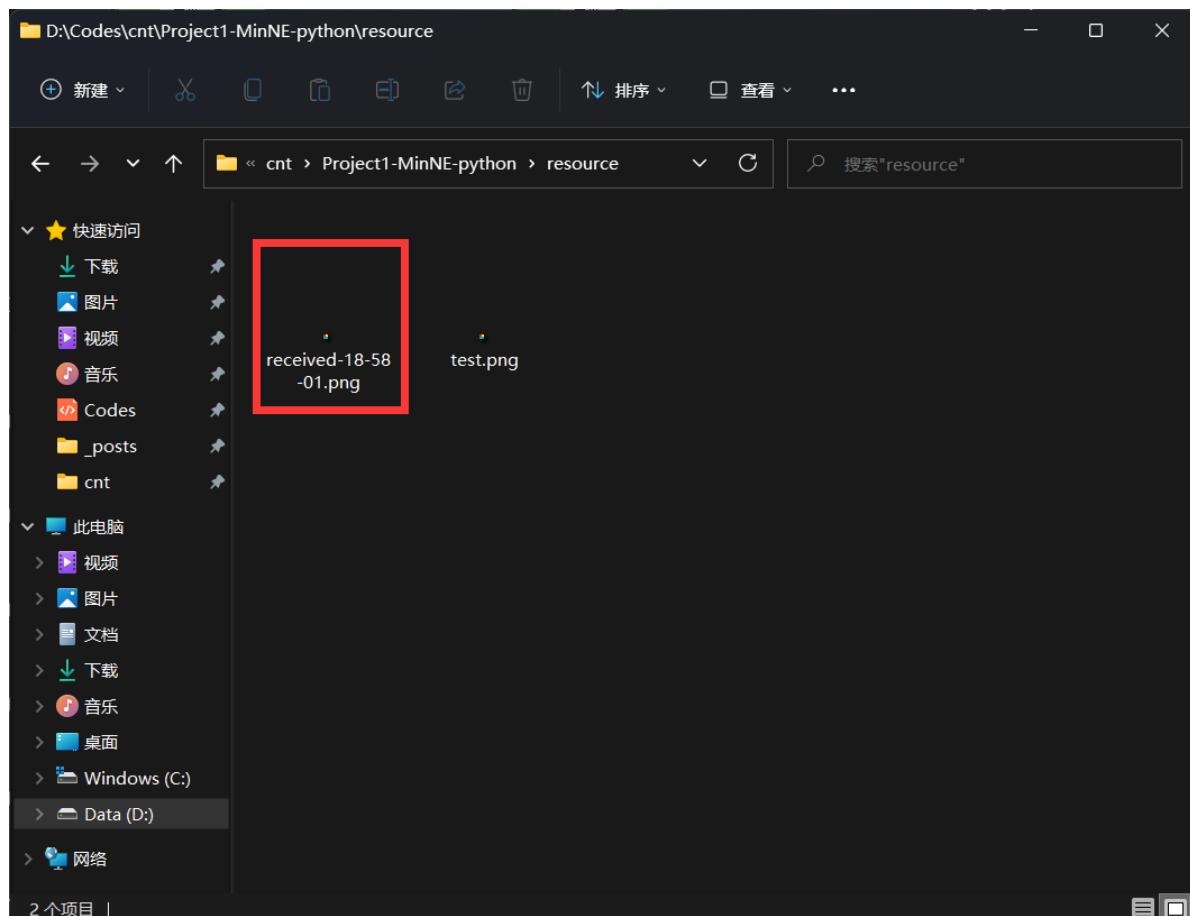
可以看到，设备1成功收到了这一字符串。由此可以说，我们成功实现了指导书的文本传输容量需求；并且在理论上，这一容量是没有上限的，因为我们物理层单次发送的数据远小于MTU限制。接下来的图片I/O测试，也能侧面反映这一点。

### 4.8.3 图片I/O

在本测试中，我们从设备1向设备2发送图片“test.png”。由于我们平时生活中的图片大多都有上百万位比特，测试耗时实在过长，所以我们自行绘制了一幅4\*4像素的微型图片，用于简单测试图片传输效果。



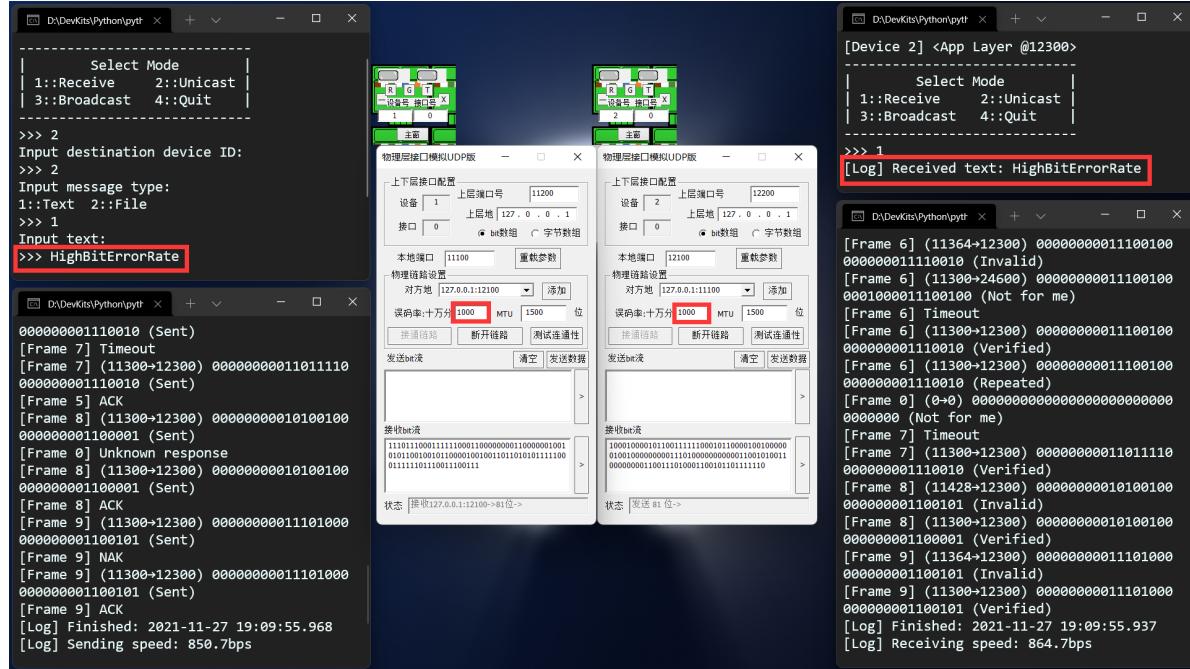
可以看到，图片成功地以比特流形式从一端传到了另一端，并提示保存在了/resource目录下。打开目录，我们可以发现一张一模一样的图片“received-XX-XX-XX.png”，这就是接收端收到的图片。



由此可以说，我们成功实现了小型图片的I/O，但中、大型图片的I/O需要性能的进一步优化（尤其是差错控制协议的优化），来加速比特流的传输。

#### 4.8.4 检错与差控

在本测试中，我们设置两边的物理层误码率为十万分之1000（1%），从设备1向设备2发送文本“HighBitErrorRate”。



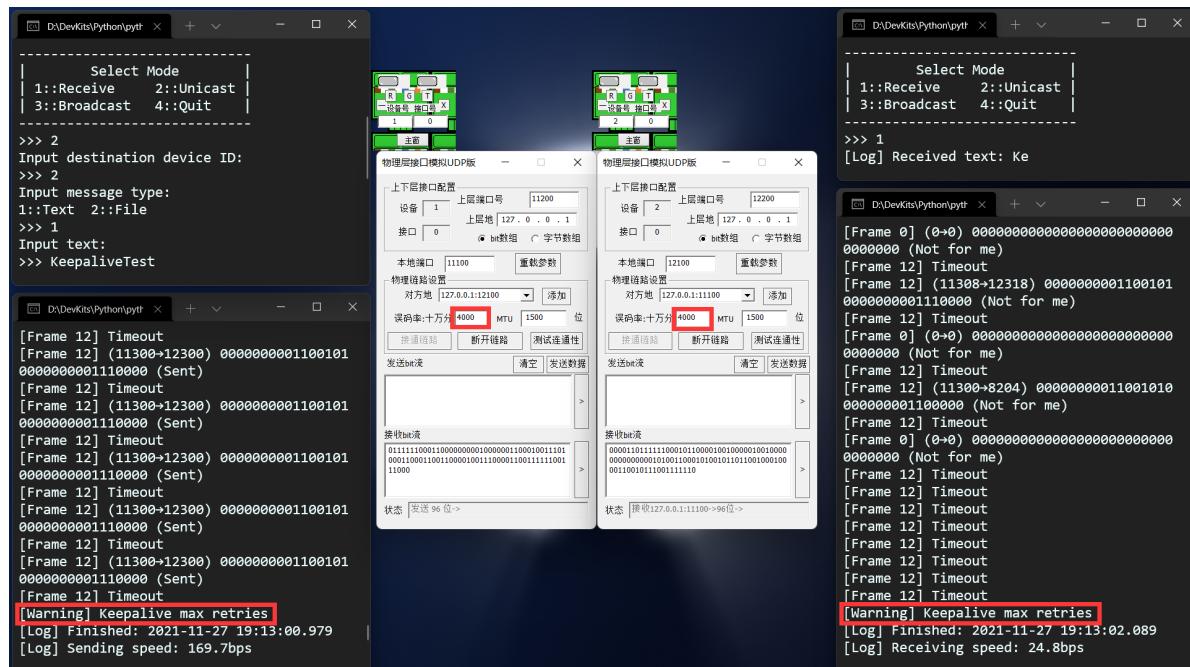
可以看到，传输过程中出现了多次超时、负确认、乱码等情形；但通过双方网络层的检错与差控，接收端应用层最终能够呈现出正确、完整的字符。由此可见，我们成功实现了CRC-16校验、停等协议这两个基本的方法。

#### 4.8.5 Keepalive机制

在本测试中，我们将对两种可能触发Keepalive机制的情况分别进行测试。

##### 第一种情况，信道环境恶劣。

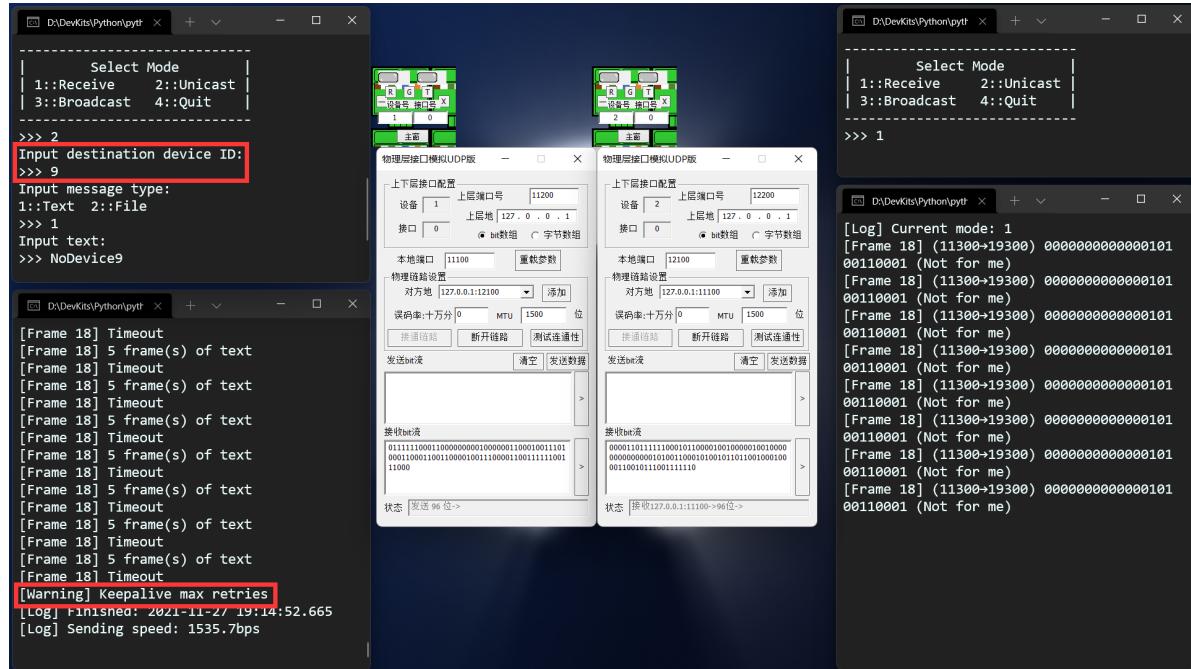
我们设置两边的物理层误码率为十万分之4000（4%），来模拟恶劣的信道环境；然后从设备1向设备2发送“KeepaliveTest”。



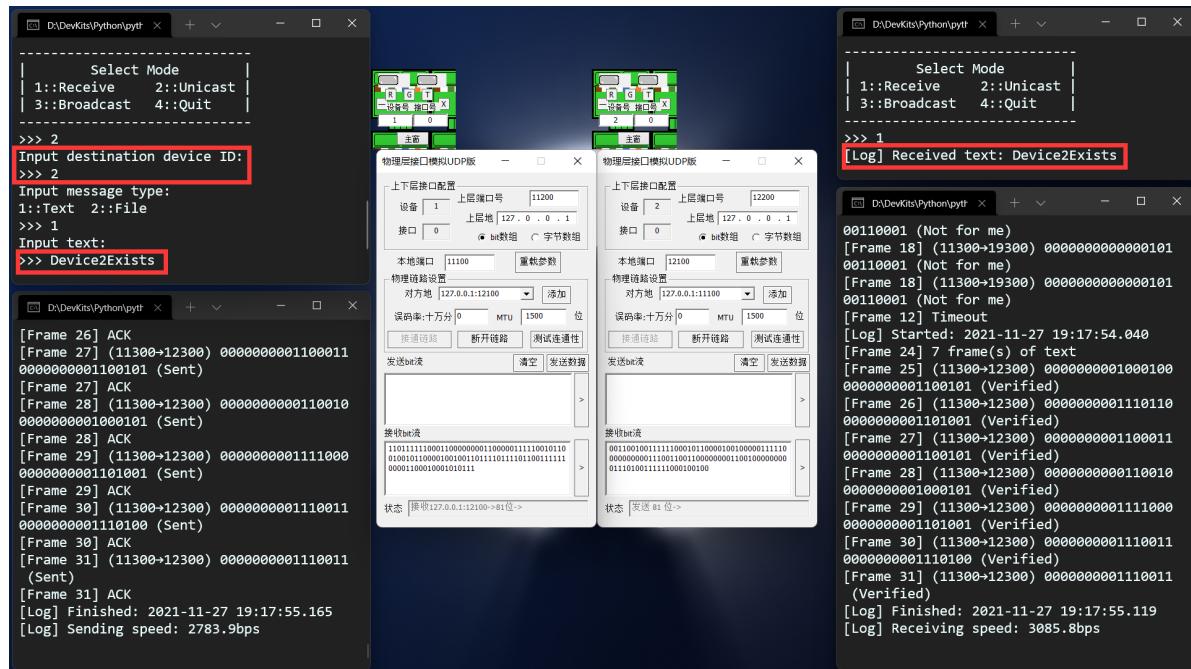
在收发双方多次的超时过后，双方都触发了Keepalive机制，停止了收发过程。

## 第二种情况，目的地不存在。

我们恢复误码率为0，但设备1向不存在的设备9发送文本“NoDevice9”。



在发送端多次超时过后，发送端触发了Keepalive机制，停止了重传，而接收端继续等待接收。我们再次在设备1发送文本“Device2Exists”，这次目的地为设备2。



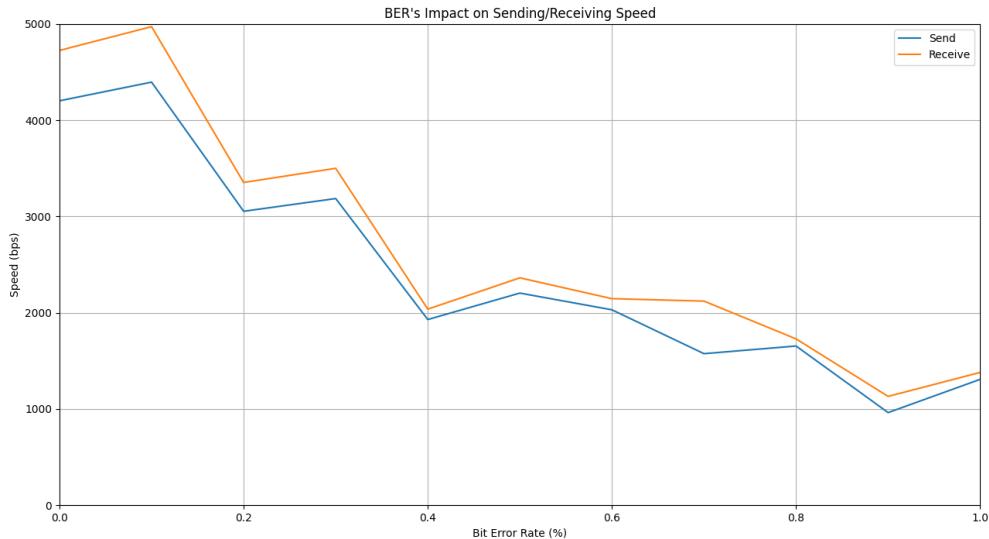
接收端此时能够正确地接收消息，就好像发送端没有出错一样。由此可以说，我们成功实现了简单的Keepalive机制。

## 4.8.6 各项性能指标

### 4.8.6.1 吞吐量

在本测试中，我们会以比特速率 (bps) 来衡量系统的性能。每次测试会在不同误码率下，发送信息“helloworld”，然后由网络层计算出每次发送的网速；如果接收端成功接收，就记录下这次的网速。

两边的物理层误码率从0到1%每0.1%为一档，每档测试3次取平均值。使用 Python+Matplotlib 绘制出测试结果曲线如下：



在信道环境良好的情况下，吞吐量最高可以达到4500bps左右；随着误码率提高，吞吐量总体呈下降趋势。

#### 4.8.6.2 误码承受能力

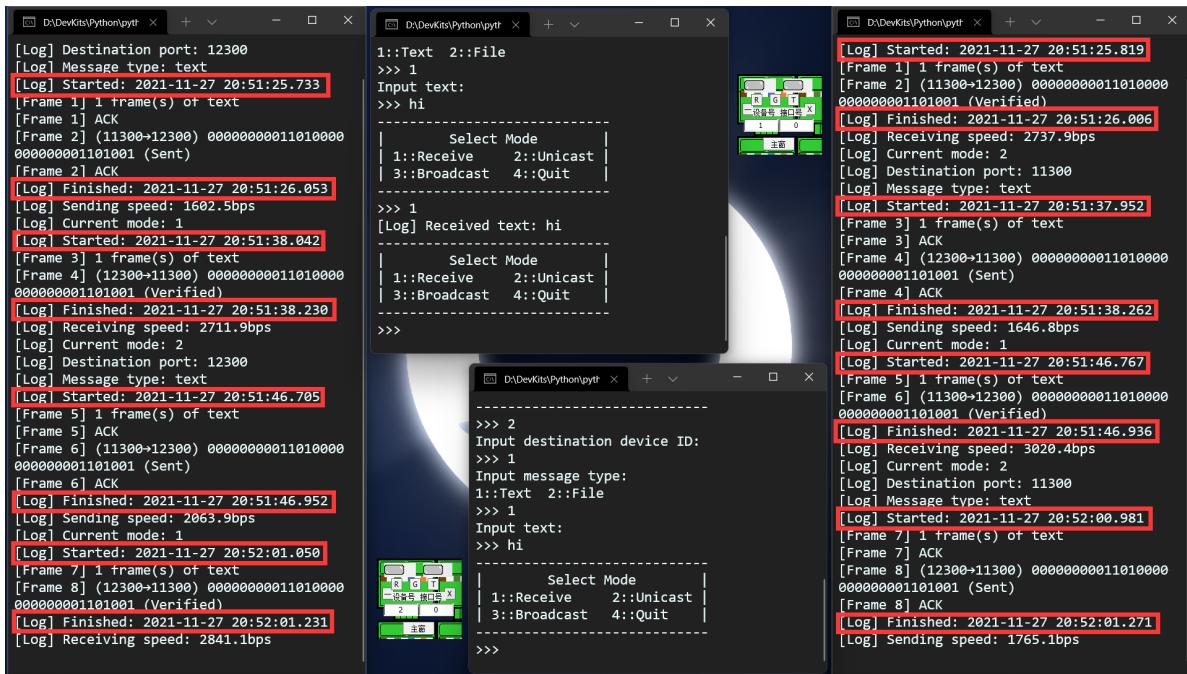
在本测试中，我们将测试系统对误码的承受能力。我们将逐步提高误码率，步进0.5%；当连续3次触发双方的Keepalive机制时，则认为系统达到误码承受能力上限。测试结果如下表：

误码率	发送端	接收端
0 – 3.0%	成功发送	成功接收
$\geq 3.5\%$	触发Keepalive	触发Keepalive

由此可以说，该系统对误码的承受能力在3.5%左右；高于3.5%时，用户需要等待信道恢复良好，再重新输入消息。

#### 4.8.6.3 传输延时

在本测试中，我们从设备1向设备2发送文本“hi”，通过网络层显示的时间信息测算传输延时。记 $t_1$ 为发送端发送第一帧到接收端接收第一帧的间隔， $t_2$ 为接收端发送最后一帧回复到发送端接收最后一帧回复的间隔。



4次测试中,  $t_1$ 平均值为76.75ms,  $t_2$ 平均值为33.75ms。这一数值显然并不令人满意, 我们推测主要是停等协议造成的, 因为等待接收端回复的时间实在白白浪费了太多时间。

#### 4.8.7 心得与反思

通过阶段二的程序编写, 我们深入了解了网络层(与数据链路层)各种功能的基本原理, 同时对停等协议下双端的交互时序有了深刻的体会。

我们的主机网络层仍存在不尽人意之处, 例如停等协议实在太简单粗暴, 大大限制了发送与接收的速度; 流量控制也过分简陋, 不能根据网路状态进行自适应。但令我们感到满意的是, 我们在阶段二的几次联调中, 均未发现任何的bug或异常。

## 五、交换机网络层

在网络中, 交换机负责在多个主机间交换信息, 从而让广播成为可能, 同时也减少了P2P通路的数量。它主要有两种功能:

- 维护端口地址表
- 单播广播支持

下面我们将分别展示这两种功能。

### 5.1 维护端口地址表

对于主机而言, 它的信息只有一条路可走——就是发到自己的物理层, 然后交给交换机。但对于交换机而言, 它的信息有不止一条路可走。

直接广播给所有端口然后让它们自行判收, 显然是浪费信道资源的一种做法, 我们更希望交换机自己能够记住, 发给谁的信息要走哪个端口。这就需要它内部维护一张对照表, 将本地自己的物理层端口与远程其他主机的应用层端口联系起来。

同时, 端口地址表, 在高误码率情况下, 有可能记录错误的端口; 我们还希望端口地址表能够自行清理这些错误的端口。这就需要交换机为每个端口设置一个生存周期, 当端口过期, 就自动清除它。

我们使用`collection.defaultdict`类型对此进行管理。使用JSON描述端口地址表内部数据如下:

```

1  {
2      "14100": {           // 交换机的物理层14100。
3          "11300": 100,    // 从这个端口可以去往应用层11300, 当前寿命为100。

```

```

4     "65535": -1 // 广播端口65535的寿命无限。
5,
6     "14101": { // 交换机的物理层14101。
7         "12300": 99, // 从这个端口可以去往应用层12300, 当前寿命为99。
8         "65535": -1 // 广播端口65535的寿命无限。
9     },
10    "14102": { // 交换机的物理层14102。
11        "13300": 57, // 从这个端口可以去往应用层13300, 当前寿命为57。
12        "65535": -1 // 广播端口65535的寿命无限。
13    }
14 }

```

在主程序中，每一次交换机接收到信息，它都会：

1. 刷新一次端口地址表，剔除过期的端口，同时反向学习；
2. 根据发来的帧的目的端口进行查询，看要发到哪个物理层端口。

我们在 `switchLayer` 类中封装了这两个方法，分别是：

- `update()`：上述任务1；
- `search_locals()`：上述任务2。

具体代码实现逻辑较复杂，不在此叙述，源文件内有详细注释。

## 5.2 单播广播支持

为了实现对单播广播的支持，交换机需要截获并读取帧，判断帧内消息发送的形式（单播或广播），然后据此采取相应的行动。

- 如果是单播消息，就在端口地址表中查询应该送到哪个端口；
  - 如果查到了，就发向这个端口；
  - 如果没找到，就发向各个物理层，接收端回复时就能反向学习了；
- 如果是广播消息，就直接发向各个物理层。

## 5.3 代码框架

将以上的两部分结合起来，就是交换机网络层的代码框架。

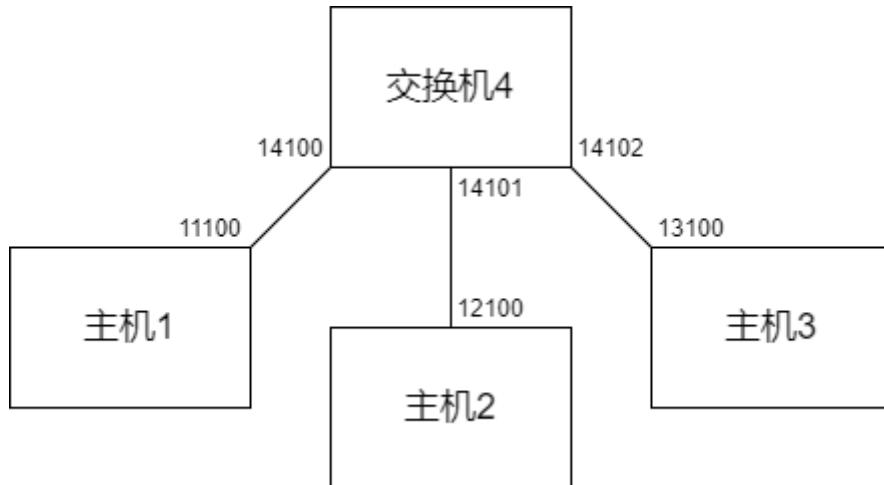
```

1 while True:
2     if not has_message():
3         continue
4
5     receive_frame()
6
7     update()
8     search_where_should_I_send_this_frame()
9
10    if search_success:
11        unicast_to_this_port()
12    else:
13        broadcast_to_every_port()

```

## 5.4 阶段三测试

根据上面的代码框架，我们在阶段三写出了交换机网络层，并在交换机下接入了多台主机，在这些网元间进行了测试。下面是阶段三的拓扑。



测试结果如下。

### 5.4.1 单播支持

在测试中，我们从设备2向设备1单播文本“HelloHost1”。

The screenshot shows four terminal windows. The top-left window [Device 1] shows Host1 receiving the message "[Log] Received text: HelloHost1". The top-right window [Device 3] shows Host3 sending the message. The bottom-left window [Device 2] shows Host2 sending the message "HelloHost1". The bottom-right window [Device 4] shows Switch4's internal state tables. The first table (Local 14100) has entries for 11100 (port 65535, life inf) and 12300 (port 11300, life 100). The second table (Local 14102) has entries for 11300 (port 65535, life inf) and 12300 (port 11300, life 99). Both tables also have an entry for 14101 (port 65535, life inf).

可以看到，交换机能够正确的学习并更新端口地址表，并充当两主机发送、回复、重传的桥梁，消息最终完好地传递到了设备1。

同时，我们对交换机实际表现出的行为做如下分析：

1. 交换机从14101收到一个包，它读取包得知源端口号为12300。此时端口地址表内还没有14101-12300的关系，于是它将其加入表内；
2. 交换机读取包得知目标端口号为11300，但此时它还不知道要到11300该从哪里走，所以它向14100、14102广播了这个包；
3. 交换机从14100收到一个包，它读取包得知源端口号为11300。此时端口地址表内还没有14100-11300的关系，于是它将其加入表内；
4. 交换机读取包得知目标端口号为12300，此时它的表内已经有14101-12300的关系，所以它直接向14101单播了这个包；
5. 之后的过程中，交换机已经完全学会了两边的关系，所以之后全部都是单播。

这与我们在[5.2节](#)中的理论分析完全一致。由此我们可以说，我们成功实现了交换机的单播功能，同时也很难知道，我们的单播判收和反向学习也是成功的。

## 5.4.2 广播支持

在测试中，我们从设备3向所有设备广播文本“HelloEveryone”。

```
>>> 1
[Log] Received text: HelloHost1
| Select Mode
| 1::Receive 2::Unicast
| 3::Broadcast 4::Quit
>>> 1
[Log] Received text: HelloEveryone
| Select Mode
| 1::Receive 2::Unicast
| 3::Broadcast 4::Quit
>>>
>>> 1
[Log] Received text: HelloEveryone
| Select Mode
| 1::Receive 2::Unicast
| 3::Broadcast 4::Quit
>>>
>>> 3
Input message type:
1::Text 2::File
>>> 1
Input_text:
>>> HelloEveryone
| Select Mode
| 1::Receive 2::Unicast
| 3::Broadcast 4::Quit
>>>
>>>
>>> 1
Input text:
>>> HelloHost1
| Select Mode
| 1::Receive 2::Unicast
| 3::Broadcast 4::Quit
>>> 1
[Log] Received text: HelloEveryone
| Select Mode
| 1::Receive 2::Unicast
| 3::Broadcast 4::Quit
>>>
```

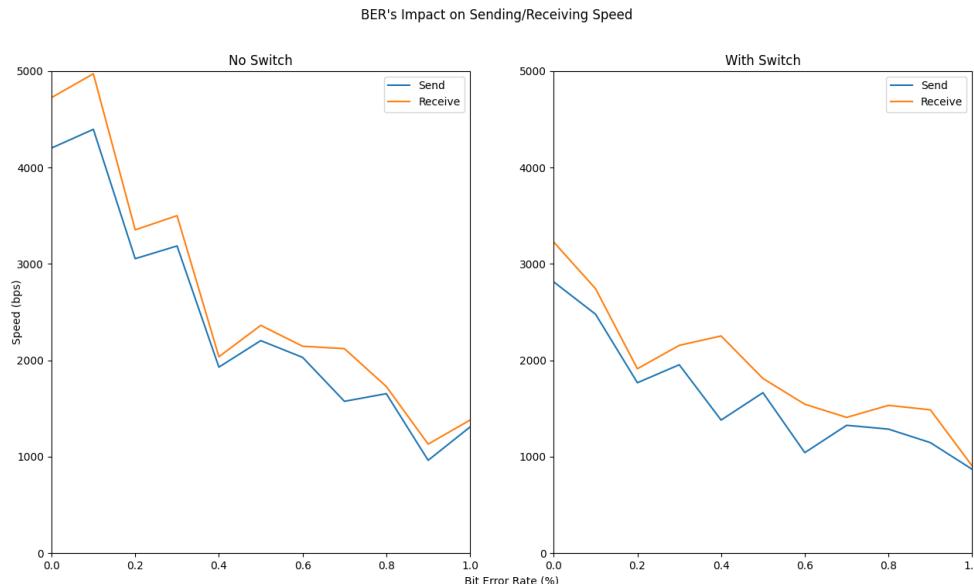
可以看到，在设备3发送时，交换机能够正确识别出这是广播消息，并广播给设备1和2；在设备1和2回复时，交换机也能够正确识别出这是单播消息，并单播给设备3。最终，消息完好地传递到了所有设备。由此我们可以说，我们成功实现了交换机的广播功能。

## 5.4.3 各项性能指标

### 5.4.3.1 吞吐量

测试方法与[阶段二相应测试](#)基本一致，但本次测试调整的是交换机两个物理层端口的误码率，从0到1%，每0.1%为一档。

同样绘制曲线如下：



左边是阶段二无交换机的曲线，作为对照；右边是本次有交换机的吞吐量测试结果。在信道环境优质的情况下，吞吐量最高可以达到3000bps左右；随着误码率提高，吞吐量总体呈下降趋势。

这一现象显然有其合理性：加入交换机后，消息的传输过程就额外增加了交换机解包、查表的时间开销；其传输时间必然多于无交换机的情况。

### 5.4.3.2 误码承受能力

测试方法与[阶段二相应测试](#)基本一致，但本次测试调整的是交换机两个物理层端口的误码率。测试结果如下表：

误码率	发送端	接收端
0 – 3.0%	成功发送	成功接收
$\geq 3.5\%$	触发Keepalive	触发Keepalive

由此我们可以说，该系统对误码的承受能力在3.5%左右，这与无交换机时一致；高于3.5%时，用户需要等待信道恢复良好时，再重新输入消息。

这一现象亦有其合理性：交换机做的只是简单的消息转发，而并不对消息本身做任何更改；而改变误码率的物理层，和阶段二一样，只有两处，所以其误码承受能力理应没有太大变化。

### 5.4.3.3 传输延时

测试方法与[阶段二相应测试](#)基本一致。

对于单播，我们测试了3次， $t_1$ 平均为99ms， $t_2$ 平均为42ms，相比于阶段二均有所延长。

对于广播，我们测试了3次， $t_1$ 平均为152ms， $t_2$ 平均为182ms，相比于阶段二大幅延长。

这一大幅延长亦有其合理性：第一，发送端需要接收来自多方的回复，而每一次都是采用停等协议这一笨重的方法；第二，在接收完所有回复后，发送端还必须等待一次超时时间，来保证回复全部接收。这导致了广播耗时的大幅增加。

### 5.4.4 心得与反思

通过阶段三的程序编写，我们对端口地址表的实现有了深刻的体会，同时也了解了“反向学习”，这一个初听神秘、现在熟知的功能。同时，我们也在编写广播功能的同时，认识到了广播风暴的根源。

同时我们也清楚，我们的交换机只适用于最简单的拓扑网络；由于我们并没有实现生成树算法，我们的交换机无法应对环形拓扑的死循环症状。因此，我们的交换机只适用于简单的拓扑结构，而不能适应环形交换机拓扑。这是我们程序的一大问题。

## 六、路由器网络层

相比起交换机，路由器有更全局的视野、更智能的实体；交换机解决不了的投递定向问题，路由器可以解决。在整个网络中，路由器主要有两种功能：

- 维护路由表
- 数据定向投递

下面将依次展示这两种功能。

### 6.1 路由表

路由表是路由器的核心所在：后者正是依靠对前者的不断更新与维护，才能做到对网络全局的把控、对数据包的精准投递。

我们采用[链路状态算法](#)来维护路由表。

### 6.1.1 基本原理

链路状态算法主要采用**链路延时**作为衡量链路状态的标准：链路的延时越高，意味着“走这条路到达别的路由器的代价”就越高。每台路由器只关心自己周围的链路状态，据此更新自己的路由表，然后将自己的路由表扩散到全网的路由器。

当路由器接收到来自其它路由器的路由表时，它会采用**Dijkstra算法**计算出到达全网各点的最佳路径，即“费用最低”的路径。具体实现如下：

1. 一开始，只记录相邻链路的费用，而认为其它所有节点的费用均为无穷；
2. 选择路由表中费用最低的路径P，将其标记为“最优化”，不再对其进行更新；
3. 接收P的终点路由器T的路由表，然后在此基础上，判断“先走T，再走T的邻路”是否能让自己更快地到达T的邻居。如果是，则更新自己的路由表；
4. 重复2、3，直到路由表中的每一条路径都被标记为“最优化”。

### 6.1.2 代码实现

我们将Dijkstra算法等重要函数封装在了RouterTable类内；路由表通过merge()函数即可迅速实现本地路由表与新路由表的合并。对于路由器间的信息交互，我们设置了pack()和unpack()函数，以便捷地打包、解包路由表。具体代码见layer/router.py。

## 6.2 数据定向投递

我们常说，路由表“划分了广播域”，所有的广播消息都不会穿过路由器。这一叙述的实际含义就在于，路由器总是知道帧的目的地在哪里、应该投递给谁；而不会像交换机那样，遇到不认识的端口就直接广播。

### 6.2.1 基本原理

在实际的生活中，寻址是通过ARP（地址解析协议）完成的，它能够将IP地址映射到MAC地址。但由于本项目的网络拓扑模型较小，且限于本机套接字间通信，加之物理层模拟软件限制了端口号各位的意义，我们可以——并且也只能——自定义一种映射方案来简化这一过程。方案如下：

- 端口号定义：
  - 第1位统一为1；
  - 第2位为设备号；
  - 第3位为层次号：应用层为3，网络层为2，物理层为1；
  - 第4-5位为实体号，从00开始编号。
- 例如，设备2的物理层实体3的端口号为12103。
- 路由器寻址：
  - 记任意类型设备m的设备号为 $D_m$ ，路由器i的设备号为 $R_i$ ，交换机j的设备号为 $S_j$ ，主机k的设备号为 $H_k$ 。定义 $D_s \in D_t$ 为“设备s隶属于设备t”。
    - $k < j, \forall H_k \in S_j$
    - $k_1 < k_2, \forall H_{k_1} \in S_{j_1}, H_{k_2} \in S_{j_2}, j_1 < j_2$
    - $j < i, \forall S_j, H_k \in R_i$
    - $j_1 < j_2, \forall S_{j_1} \in R_{i_1}, S_{j_2} \in R_{i_2}, i_1 < i_2$

即，对于任意一台有上级的设备A，其上级B的设备号必然大于A；但是在这一前提下，所有可能成为A上级的设备中，B的设备号是最小的。

- 投递过程：
  - 用户指定设备号，而非端口号；
  - 网元将目的设备号映射为应用层端口号。

例如，用户指定向设备4投递消息，则网元将其映射为14300。

### 6.2.2 代码实现

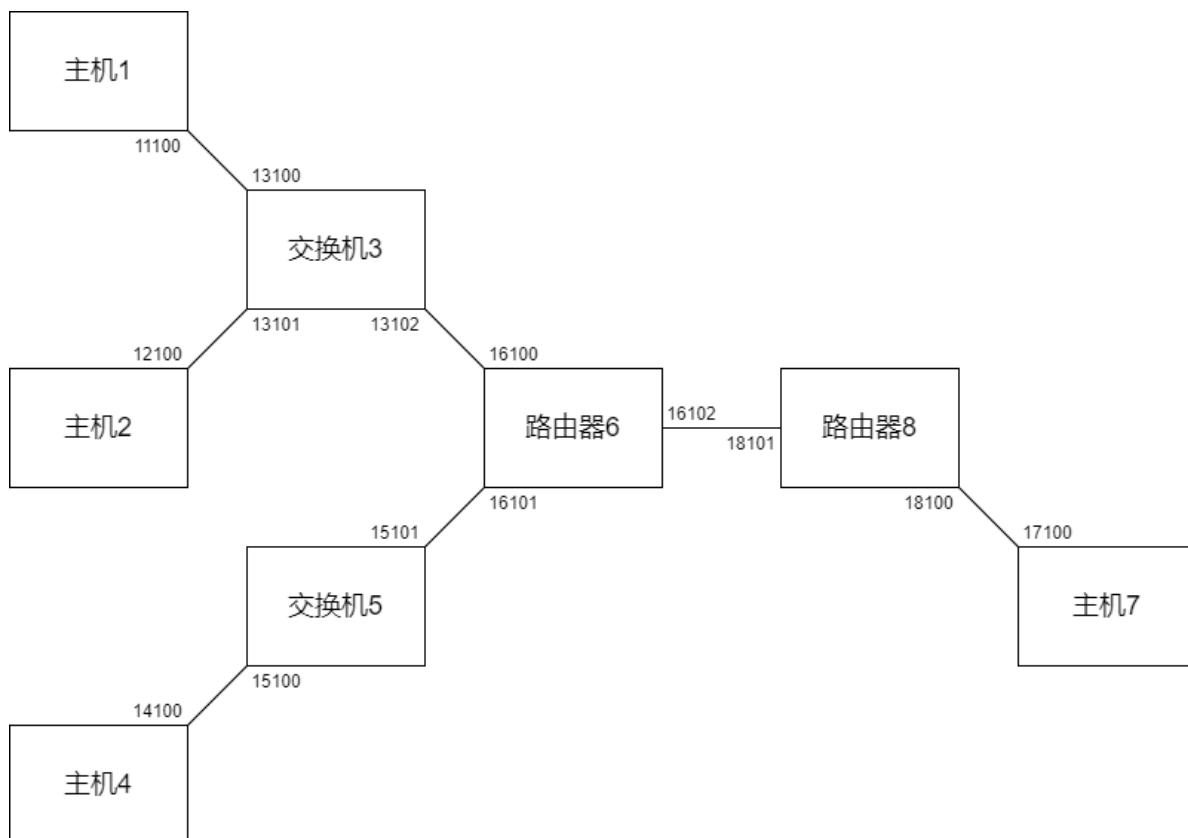
这一映射与反射需要主机应用层和路由器网络层的共同努力。我们在 AppLayer 类内封装了 `_get_dst_from_user()` 方法，支持设备号到端口号的映射；在 RouterLayer 类内封装了 `search()` 方法，支持端口号到设备号的反射。具体代码见 `layer/app.py` 和 `layer/router.py`。

## 6.3 代码框架

```
1 while True:
2     broadcast_table_at_regular_interval()
3     if not has_message():
4         continue
5
6     receive_frame()
7     if send_to_me():
8         merge_this_table()
9     else:
10        search_exit()
11        if no_result:
12            continue
13        else:
14            send_frame_to_this_exit()
```

## 6.4 阶段四调试

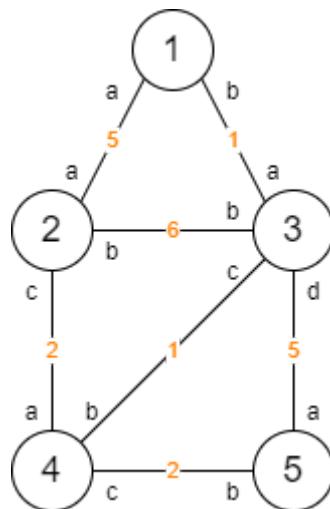
根据上面的代码框架，我们在阶段四写出了路由器网络层，并建立了混合组网拓扑，实现了不同子网的主机交互。下面是阶段四的拓扑。



测试结果如下。

### 6.4.1 路由表计算

在本测试中，为了测试我们编写的路由表在拥有较多路由器的拓扑网络中的表现，我们另外编写了一套拓扑配置，拓扑图如下：



其 JSON 文件见附录。我们只导入路由表类 (RouterTable)，进行不依赖套接字的测试。测试结果如下：

Router table 1			
Destination	Next hop	Exit port	Cost
2	4	b	4
3	3	b	1
4	3	b	2
5	4	b	4

Router table 2			
Destination	Next hop	Exit port	Cost
1	3	c	4
3	4	c	3
4	4	c	2
5	4	c	4

Router table 3			
Destination	Next hop	Exit port	Cost
1	1	a	1
2	4	c	3
4	4	c	1
5	4	c	3

Router table 4			
Destination	Next hop	Exit port	Cost
1	3	b	2
2	2	a	2
3	3	b	1
5	5	c	2

Router table 5			
Destination	Next hop	Exit port	Cost
1	3	b	4
2	4	b	4
3	4	b	3
4	4	b	2

对照拓扑图，我们很容易验证其正确性。由此可以说，我们成功实现了路由表的计算与合并更新。

### 6.4.2 跨网单播

在本测试中，我们将测试不同子网下的主机间能否正常通信。我们从隶属于路由器6的设备4，向隶属于路由器8的设备7发送文本“HelloHost7”。

```

[Device 1] <App Layer @11300>
-----
| Select Mode |
| 1::Receive   2::Unicast |
| 3::Broadcast 4::Quit    |
-----
>>> |
```

```

[Device 2] <App Layer @12300>
-----
| Select Mode |
| 1::Receive   2::Unicast |
| 3::Broadcast 4::Quit    |
-----
>>>
```

```

[Device 4] <App Layer @14300>
-----
| Select Mode |
| 1::Receive   2::Unicast |
| 3::Broadcast 4::Quit    |
-----
>>> 2
Input destination device ID:
>>> 7
Input message type:
1::Text 2::File
>>> 1
Input text:
>>> HelloHost7
[Log] Received text: HelloHost7
```

```

[Device 7] <App Layer @17300>
-----
| Select Mode |
| 1::Receive   2::Unicast |
| 3::Broadcast 4::Quit    |
-----
>>> 1
[Log] Received text: HelloHost7
```

由于终端数量过多，我们只展示各主机的应用层。可以看到，设备7正确地接收到了来自设备4的消息。由此可以说，我们成功实现了路由器的跨网单播；同时也不难知道，我们的端口映射方案也是成功的。

### 6.4.3 局域网广播

在本测试中，我们将测试主机能对局域网内主机进行广播。我们从设备2向路由器6下的所有设备发送文本“HelloUnder6”。

```

[Device 1] <App Layer @11300>
-----
| Select Mode |
| 1::Receive   2::Unicast |
| 3::Broadcast 4::Quit    |
-----
>>> 1
[Log] Received text: HelloUnder6
```

```

[Device 2] <App Layer @12300>
-----
| Select Mode |
| 1::Receive   2::Unicast |
| 3::Broadcast 4::Quit    |
-----
>>> 3
Input message type:
1::Text 2::File
>>> 1
Input text:
>>> HelloUnder6
```

```

[Device 4] <App Layer @14300>
-----
| Select Mode |
| 1::Receive   2::Unicast |
| 3::Broadcast 4::Quit    |
-----
>>> 1
[Log] Received text: HelloUnder6
```

```

[Device 7] <App Layer @17300>
-----
| Select Mode |
| 1::Receive   2::Unicast |
| 3::Broadcast 4::Quit    |
-----
>>> 1
```

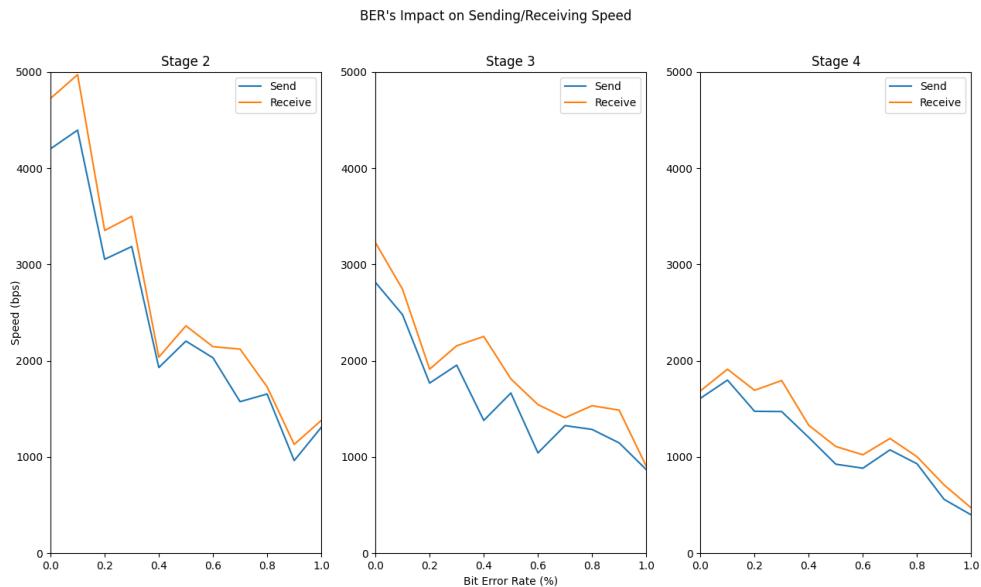
可以看到，同属于路由器6下的主机1和4都收到了广播消息；而主机7由于不属于该局域网，并没有收到任何消息。由此可以说，我们成功实现了路由器的局域网广播。

### 6.4.4 各项性能指标

#### 6.4.4.1 吞吐量

测试方法与阶段三相应测试基本一致，但本次测试调整的是路由器对应两主机的出口物理层的误码率，从0到1%，每0.1%为一档。

同样绘制曲线如下：



最右是本次路由器的吞吐量测试结果。在信道环境优质的情况下，吞吐量最高可以达到1800bps左右；随着误码率提高，吞吐量总体呈下降趋势。

#### 6.4.4.2 误码承受能力

测试方法与阶段三相应测试基本一致，但本次测试调整的是路由器对应两主机的出口物理层的误码率。测试结果如下表：

误码率	发送端	接收端
0 – 2.0%	成功发送	成功接收
$\geq 2.5\%$	触发Keepalive	触发Keepalive

由此我们可以说，该系统对误码的承受能力在2.5%左右；高于2.5%时，用户需要等待信道恢复良好时，再重新输入消息。

相较于阶段三，加入路由器后的误码承受能力下降，我们组推测其原因是，路由器增加了端到端的延时，本来可以到达另一端的消息因为该延时被误判为超时，于是白白增加了超时的次数。

#### 6.4.4.3 传输延时

测试方法与阶段三相应测试基本一致。

对于单播，我们测试了3次， $t_1$ 平均为167ms， $t_2$ 平均为47ms，相比于阶段三均有所延长。

对于广播，我们测试了3次， $t_1$ 平均为209ms， $t_2$ 平均为390.3ms，相比于阶段三大幅延长。

#### 6.4.5 心得与反思

通过对阶段四的程序编写，我们组对链路状态路由表的计算算法有了深入的了解与实践，对路由表交换信息的方式有了基本的了解，同时也明白了构建好的拓扑图、映射关系的重要性。

同时，我们的阶段四程序并不很令人满意。虽然我们能够实现跨网单播、内网广播等基本功能，但我们的路由表做的过于简陋，主要表现为以下两点：

- 没有通过测算链路延时，来计算链路状态，而是将链路状态（费用）设置为一个固定值；
- 只编写了路由表扩散与接收的函数，但没有真正实现与其它路由器的交流。

因此，我们的路由器其实并没有做到完全的链路状态算法，只是能够算出全网的拓扑与费用而已，但实际上都是静态的。这是我们阶段四最大的遗憾。

同时，我们也发现我们的流控超时时间存在一定的瑕疵：虽然在应用层上，我们的确能完整地收到消息，但翻阅网络层的日志，我们发现存在着许多不该出现的超时与乱序，这需要我们对超时参数进行进一步的调整。

## 七、总结

---

### 7.1 指导书需求完成情况

综上所述，我们组编写的网元系统能够完成指导书上的基本要求，以及部分进阶要求，具体列出如下：

- 阶段二

- 基本要求

- 帧同步与定位：面向位的首尾定界法
    - 差错检测：CRC-16校验码
    - 差错控制协议：停等协议
    - 流量控制：睡眠固定时间
    - 性能测试：误码率-吞吐量曲线

- 进阶要求

- 端到端差错控制协议
    - 面向连接协议
    - 有测量计算的流量控制
    - 分帧：帧封装，单次传输100位
    - 多角度性能测试：误码承受能力、延时
    - 其它相关：Keepalive机制、模块化编程

- 阶段三

- 基本要求

- 单播目的地址判收：成功实现
    - 支持广播：成功实现
    - 有端口地址表：成功实现
    - 反向地址学习：成功实现
    - 性能测试：误码率-吞吐量曲线、误码承受能力、延时

- 进阶要求

- 环形交换机拓扑效果：广播死循环
    - 生成树算法
    - 友好的应用层：输入容错高，界面美观
    - 其它相关：端口地址表引入老化过期机制

- 阶段四

- 基本要求

- 实体编址：成功实现
    - 路由表：成功实现
    - 按路由转发：成功实现
    - 端到端信息交换：成功实现单播、广播

- 性能测试：误码率-吞吐量曲线、延时
- 进阶要求
  - 相互独立的地址映射技术
  - 动态路由算法：只实现了 Dijkstra 算法，但不交流
  - 多种多层网元体系完整：建立了混合组网拓扑
  - 多角度性能测试：误码承受能力
  - 应用层图形化界面
  - 统一友好的观测平台

## 7.2 项目心得体会

该项目是课程四个项目中最难的一个，它涵盖了课程中的许多知识点：定位、检错、差控、流控、共享、交换、路由、转发、广播……但相应地，它也是给我们小组最多收获的一个。

通过这一项目，我们小组真正地将课堂所学的内容，应用到了实际的编程当中，并且也对这门课程的授课大纲有了更深层次的把握。我们不再只是知识的接收者，而是成为设计者，通过实操真正理解了为什么、怎么样；也因此，我们的专业知识水平、程序设计水平、团队合作水平得到了极大的提高。

虽然我们小组并没有能够将代码写得十全十美、令人满意，但这一项目给我们的收获——我们认为——是远远超过代码本身的。

感谢老师们在我们组项目推进的过程中，提供的悉心教导与陪伴！

## 八、附录

---

### 8.1 编写与运行环境

- OS: Windows 11
- IDE: Microsoft Visual Studio Code
- Python version: 3.9.7 64-bit

Minimum version 3.7 required, 3.9 or higher recommended.

### 8.2 项目开源地址

<https://github.com/MrCaiDev/cnt>