

合肥工业大学

《计算机网络课程设计》报告

设计题目：多线程模拟以太网帧的发送

学生姓名：丁盛鹏

学 号：2021214813

专业班级：计算机科学与技术 3 班

2024 年 1 月

目录

一、设计要求	4
1.1 设计要求:	4
1.2 设计描述.....	4
二、开发环境与工具	5
三、设计原理	5
3.1 以太网数据链路层.....	5
3.2 以太网帧.....	5
3.3 CSMA/CD.....	6
3.4 二进制指数退避算法.....	6
3.5 交换机工作原理.....	7
四、系统功能描述及软件模块划分	8
4.1 系统功能描述.....	8
4.2 系统模块划分.....	10
4.3 系统模块结构介绍.....	11
4.3.1 CSMA	11
4.3.2 frame	12
4.3.3 PC.....	13
4.3.4 Exchange.....	14
五、模块详细设计与实现	14
5.1 CSMA	15
5.1.1 Backoff.....	15
5.1.2 Transmission line	15
5.1.3 Mutex	16
5.2 frame	16
5.2.1 Get_frame	16
5.2.2 Reverseframe	17
5.2.3 Decode	18
5.2.4 Oddcheck	18

5.3 PC.....	19
5.3.1 Listen_and_send	19
5.3.2 Get send_frame	23
5.3.3 Receive frame	24
5.3.4 Decode frame	27
5.4 Exchange.....	29
5.4.1 交换机 switch 工作	29
六、关键问题及其解决方法	35
七、设计结果	38
7.1 程序运行初始设置.....	38
7.2 PC 运行结果及介绍.....	40
7.3 switch 运行结果及说明	42
7.4 综合结果说明.....	43
7.5 主机日志记录.....	45
八、参考资料	47
九、验收时间及验收情况	47
十、设计体会	49

一、设计要求

1.1 设计要求：

模拟共享网络中 Ethernet 帧的发送过程 使用至少 2 个线程模拟 Ethernet 上主机的数据发送流程。发送流程必须遵循 CSMA/CD 协议。

1.2 设计描述

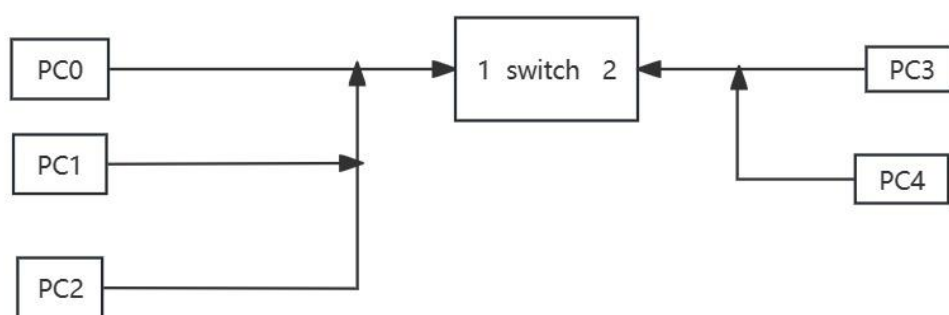


图 1 网络结构

设计完成模拟了以太网中数据链路层的功能，具体描述如下：

- Ethernet 帧的定义与结构实现
- 以太网上个人主机的定义与实现
- 以太网上交换机的定义与实现
- Ethernet 帧在发送过程中的差错控制
- 以太网发生冲突时冲突信号的定义与设置
- Ethernet 帧在发送与接收时在主机上对数据的包装与解包
- 各主机在监听是遵循 1 坚持 CSMA 算法
- 各主机在发送 Ethernet 帧时遵循 CSMA/CD 协议
- 主机发现冲突时遵循二进制退避算法等待
- 以太网上各主机发送与接受时的流量控制
- 各主机在进行数据交换时日志的记录
- 以太网上使用交换机对冲突域进行隔离
- 交换机对来自不同的冲突域的数据的转发与转发表的维护

二、开发环境与工具

开发环境：Windows10

开发语言：C++20

开发工具：Microsoft Visual Studio

三、设计原理

3.1 以太网数据链路层

以太网数据链路层主要完成以下三个功能：

- 1) 将网络层交下来的 IP 数据报添加首部尾部封装成帧；
- 2) 将封装好的帧发送给接受方的数据链路层；
- 3) 收到的帧无差错则从中提取到 IP 数据报交给网络层，否则丢弃

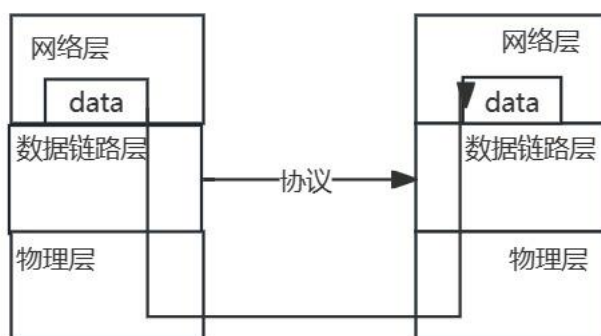


图 2 数据链路层功能

3.2 以太网帧

以太网帧就是将网络层交付的数据添加报头信息后，此时的数据以帧的形式传递。以太网帧结构如下所示。

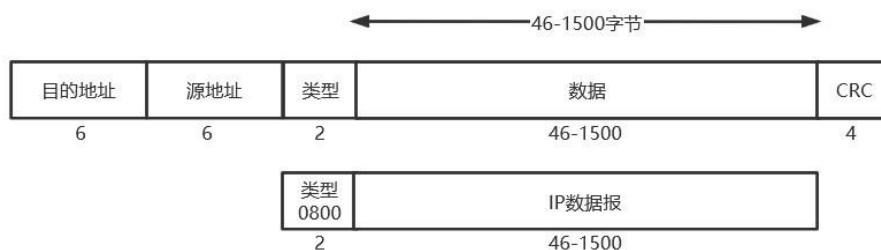


图 3 以太网帧格式

目的地址与源地址这里都指的是 MAC 地址。每一个主机对应唯一的一个 MAC 地址，是由网卡决定的，长度是 48 位，所以这里的地址都是 6 个字节，也就是 48 个比特位。在以太网帧的最后，还有一个 CRC 校验码，来校验数据是否异常。在中间，有一个两个字节的类型标识。这个类型字段有三种值，分别对应 IP、ARP、RARP。如果类型码为 0800 那么在数据链路层解包完毕后，将该数据交付给网络层的 IP 协议来处理该报文。

3.3 CSMA/CD

- 1) 传输协议节点发送数据之前需要持续监听信道，若信道 busy，就一直监听，一旦节点发现信道空闲，则立刻发送数据。在发送数据的同时，节点持续监听信道，"探测" 是否有别的节点也在该时刻发送数据。
- 2) 若传输过程中没有检测到别的节点的传输，那么成功传输。在成功传输后，节点需要等待帧间间隔 IFG (interframe gap) 时间后，可以进行下一次传输。
- 3) 若在传输过程中，探测到别的节点也在传输，那么则检测到冲突。发生冲突后，节点立刻停止当前的传输，并且发送特定的干扰序列 (JAM 序列)，用以加强该次冲突 (用以保证其余所有节点都检测到该次冲突)，在 JAM 序列发送完之后，节点随机选择一个时间倒数进行 backoff。当 backoff 完成之后，节点可以尝试再次重传。

3.4 二进制指数退避算法

在数据发生冲突后，站点需要立即停止数据的发送，并等待一定的时间后再次发送，这个时间由二进制指数退避算法得来。二进制指数退避算法的具体表现为：

- 1) 把争用期 2τ 作为基本退避时间

- 2) 从整数集合 $\{0, 1, \dots, (2^k) - 1\}$ 中抓取一个整数，记为 r ，推迟发送的时间为 $r \cdot 2\tau$ ，其中 k 的取值规则为：当重传次数不超过十次时， k 等于重传次数，超过 10 次时， k 不再增大，一直等于 10。
- 3) 当重发次数到达 16 次还不能成功时，抛弃当前数据包，向上层应用报告。

3.5 交换机工作原理

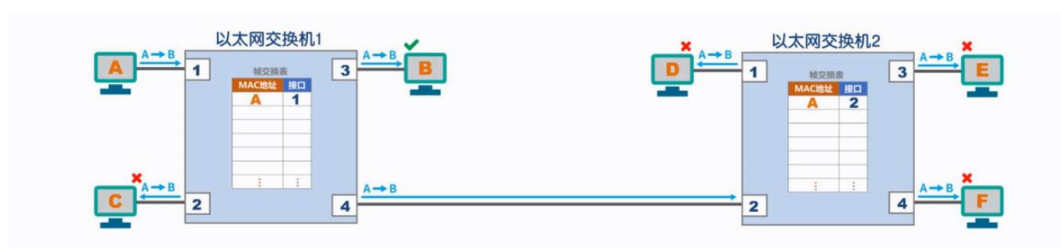


图 4 交换机工作 1

转发表初始为空，当 $A \rightarrow B$ 发送帧：

- 1) 交换机 1 首先记录 A 的 MAC 地址及对应的接口 1，查找交换机表，无 B 的 MAC，则除 A 的接口，其他接口全部转发帧（盲目泛洪）；
- 2) 接口 4 将帧转发到交换机 2，交换机 2 记录 A 的 MAC 地址及对应接口号 2，查找 B 的 MAC 地址，没有，则盲目泛洪，D、E、F 主机接收到帧，根据目的 MAC 判断不是自己的帧，不理睬；
- 3) 交换机 1 转发的帧到达 B 主机，B 回应帧，在交换机 1 中记录 B 的 MAC 地址及接口号 3，查找 A 的 MAC 地址，能找到，转发到接口 1。

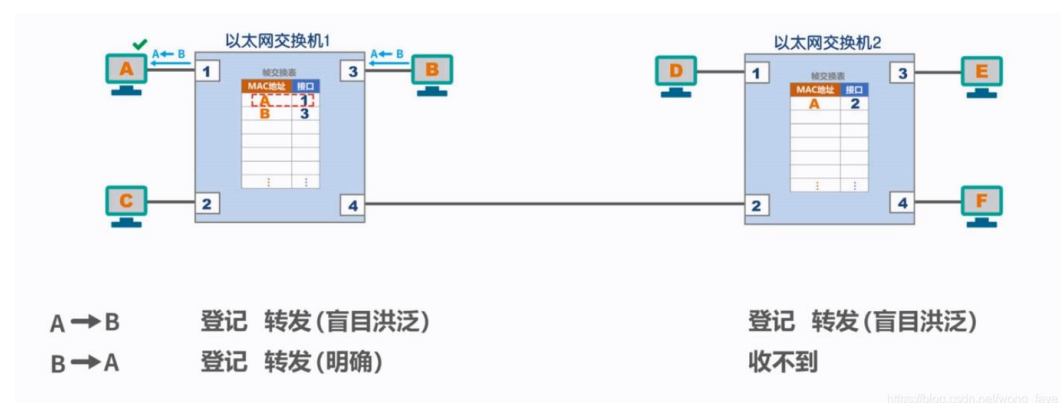


图 5 交换机工作 2

当 B 到 A 发送帧：

- 1) 帧从主机 B 出发到接口 3，交换机在帧交换表中登记，之后查询目的 MAC

地址，找到 MAC 地址为 A，接口为 1，于是从 1 接口转发出去到达主机 A。

四、系统功能描述及软件模块划分

4.1 系统功能描述

本单元中将详细描述设计的系统所实现的功能。

1. Ethernet 帧的定义与结构实现

定义 Ethernet 帧的结构与实际的帧结构略有差别，具体表现在 MAC 地址和 CRC 循环冗余码上。实际 Ethernet 帧中 MAC 地址有 48 位之多，由于考虑到 MAC 地址的功能与特征，其仅在传输过程中仅区别一个独立的主机，不需要过多冗余位数，故在设计中使用 3 位二进制码代替。具体定义的帧结构如下：

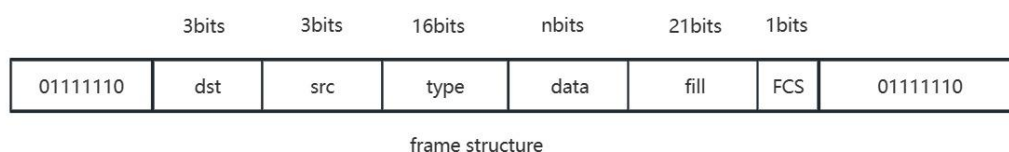


图 6 帧结构的设计

前后八位为面向比特传输协议的八位标识符；接着是 3 位的目的地址和源地址；之后 16 位的上层协议类型；n 位的数据；21 位的填充码用来模拟以太网中帧的最小长度。以太网中最小长度也相对较长，设计中模拟为最小帧长 60bit。当数据字段为空时，其余所有字段的长度和为 39bit， $60-39=21\text{bit}$ ；1 位的奇校验码用来错误检测。

2. 以太网上个人主机的定义与实现

主机作为网络上一个独立接发帧的个体，需要有接收帧、发送帧、监听主线、解码帧等功能。相应地，接收应存在接收缓冲区，发送应遵循 CSMA/CD 协议，监听应使用 1 坚持算法，解码应能将帧中包含的原数据还原。

3. 以太网上交换机的定义与实现

交换机为以太网数据链路层的设备，有转发帧、隔离冲突域的功能。所以设计中交换机应具有接收帧、维护转发表、多端口接发帧等功能。相应地，接

收应监听主线，维护转发表应遵循学习-扩散算法，同时应能在多端口之间接收并发送帧且不互相影响，且并不需要遵循 CSMA/CD 协议。

4. Ethernet 帧在发送过程中的差错控制

Ethernet 帧中存在 FCS 校验位，设计中使用奇校验实现。在校验位添加一位二进制使得整个帧的 1 个数为奇数，同理在接收方若发现 1 的个数为偶数则说明出错。

5. 以太网发生冲突时冲突信号的定义与设置

检测到冲突时主机会向总线上发送一个错误帧来告知其他主机已发生冲突，需要退避。设计中使用 32 位 1 来模拟这个错误信号帧，若主机在发送过程中发现有冲突则向总线上发送此信号，其他主机检测到此信号时得知现在已发生冲突，需要等待。

6. 各主机在发送 Ethernet 帧时遵循 CSMA/CD 协议

CSMA/CD 协议要求主机在发送数据时边发送边监听，若发现产生冲突，则立即停止发送，并产生一个信号帧告知其他主机发生了冲突，同时其应进行退避等待一段时间后再次监听发送。若冲突次数超过某一阈值，则判断为发送失败。

7. 主机发现冲突时遵循二进制退避算法等待

主机遵循 CSMA/CD 协议发生冲突时需要退避等待，等待时间由此算法给出，取 10 和 2 的冲突次数次幂中的较小者 n ，计算 $2^n - 1$ ，在 $0 \sim 2^n - 1$ 中随机选取一个值作为退避时间，过后再次监听。

8. 以太网上各主机发送与接受时的流量控制

在发送和接受帧的过程中，难免遇到发送方发送速率较快而接收方接收较慢的情况，此时接收方不得不舍弃一部分数据。在设计中各主机存在一个给定容量的接收缓冲区，缓冲区用来存放接收到的帧。当接受区未滿时可继续接收数据，接受区满时拒绝接收数据。

9. 各主机在进行数据交换时日志的记录

主机在接收数据时的数据出错、发现检测到是冲突信号 jam、接收区满不得不舍弃数据时会将相关信息记录日志中；同样的发送方在发送冲突和成功时也会写入日志文件。

10. 以太网上使用交换机对冲突域进行隔离

交换机属于数据链路层的设备，可以隔离冲突域，不同冲突域的主机可同时发送并不会产生冲突。设计中使用交换机隔离了两个冲突域，其间的交流需要先通过交换机进行转发。

11. 交换机对来自不同的冲突域的数据的转发与转发表的维护

不同冲突域的主机交换数据时需要通过交换机的转发，主机将数据送往交换机对应的端口处，交换机收到数据后检查交换表中对应的表项，表项若不存在则需要学习或扩散，若存在则根据表中记录的内容向目的主机所在的端口转发。

4.2 系统模块划分

本单元中将介绍系统各模块的划分。

系统主要分为五个部分，分别为

- **CSMA**: 模拟控制以太网中传输数据的总线，以及实现一些基础协议与算法。
- **Frame**: 模拟实现 Ethernet 帧的结构，以及对帧的一些操作，如打包解包等。
- **PC**: 模拟实现共享网络中个人主机，包含主机上关于数据链路层一些必备的结构如 MAC 地址、当前状态、接受缓冲区等，以及一些对这些结构访问操作的功能。
- **Exchange**: 模拟实现以太网中数据链路层的交换机，包括交换机上一些必备的结构如交换表、端口号等以及对交换机功能的实现。
- **Main**: 模拟构建以太网数据链路层结构，声明了创建一些线程对象来运行主机或交换机对象的功能函数。

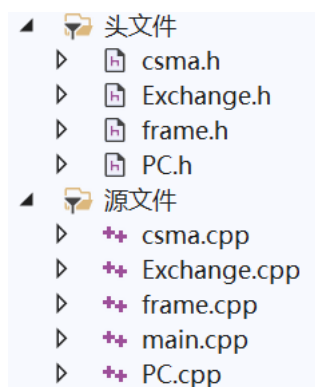


图 7 系统模块划分

4.3 系统模块结构介绍

本单元中将根据之前介绍的模块划分来描述各模块结构的详细设计，为之后模块功能实现介绍铺垫。

4.3.1 CSMA

CSMA 模块中定义并实现了以下内容

表格 1 csma 模块结构设计表

名称	类型	返回类型	描述
transmitting_data1	extern vector<int>	--	第一条总线（冲突域）上传输的数据
transmission_status1	extern int	--	第一条总线（冲突域）的状态
transmitting_data2	extern vector<int>	--	第二条总线（冲突域）上传输的数据
transmission_status2	extern int	--	第二条总线（冲突域）的状态
One	Extern mutex	--	控制总线 1（冲突域）的数据状态一致性
Out	Extern mutex	--	控制总线 2（冲突域）的数据状态一致性

min(int a, int b)	--	Int	求两个整型数的最小值并返回
backoff(int collision_count)	--	Int	二进制退避算法，传入冲突次数，返回整型数。

4.3.2 frame

Frame 模块定义并实现了以下内容

表格 2 frame 模块结构设计表

名称	返回类型	描述
getframe(vector<int>data,int source,int receiver,vector<int>type)	vector<int>	将主机中要发送的数据打包到帧中，传入的参数 data 指定了数据内容，source 指定了本机的 MAC 地址，receiver 指定了接受方的 MAC 地址，type 指定使用的协议类型。这些均为帧中必备的字段
reverseframe(vector<int> frame)	vector<int>	将发送的帧转向，发送的帧（在传输线上的帧）与其本身是相反的
decode(vector<int> frame)	vector<int>	对帧进行解包，将其包装的数据返回。
get_sender(vector<int> frame)	Int	返回帧中包含的源地址字段的值
odd_check(vector<int> frame)	Bool	对帧进行奇偶校验，返回值为奇偶校验检查的帧是否出错

4.3.3 PC

PC 模块定义并实现了以下内容

表格 3 PC 模块结构设计表

类名		描述
PC		模拟共享网络主机
成员变量		
名称	类型	描述
status	Int	主机所处状态发送、等待
send_frame	vector<int>	打包数据后要发送的帧
send_data	vector<int>	一次要发送的数据
receiver	Int	指定帧的接收者
collision_count	Int	一次发送中的冲突次数
Mac	int	本机唯一的 MAC 地址
receive_buffer	queue<vector<int>>	本机接受数据的缓冲区
receive_buffer_size	Int	缓冲区大小
switch_port	Int	本机所连交换机的端口
成员函数		
名称	返回类型	描述
PC(int MAC,int ReceiveBufferSize,int)	--	构造函数，传入指定的 MAC 地址、接受缓冲区大小、本机所连交换机端口号
get_send_frame()	Void	对本次要发送的数据进行打包，得到要发送的帧
listen_and_send()	Void	监听并发送帧
receive_frame()	Void	接受帧到缓冲区中
decode_frame()	void	对接受到的帧解包出数据

4.3.4 Exchange

Exchange 模块定义并实现了以下内容

表格 4 exchange 模块结构设计表

类名	描述	
Exchange	模拟数据链路层的交换机	
成员变量		
名称	类型	描述
Tablemtx	static mutex	多个端口对象访问交换表的锁
port1	static mutex	互斥访问总线 1 的锁
port2	static mutex	互斥访问总线 2 的锁
data1	vector<int>	与总线 1 交换的数据
data2	vector<int>	与总线 2 交换的数据
table	static vector<int*>	交换表, 表项<mac,port>
成员函数		
名称	类型	描述
Exchange(int);	--	构造函数, 指定端口
exchange()	Void	工作函数, 主要执行转发与转发表的维护
printtable()	void	打印转发表

五、模块详细设计与实现

本单元将详细介绍各模块/分模块中的详细设计, 包括文字描述、程序流程、算法伪代码描述、代码实现与讲解。

5.1 CSMA

5.1.1 Backoff

二进制指数退避算法的实现

```
int backoff(int collision_count) {
    int k = min(collision_count, 10);
    int m = pow(2, k) - 1;
    int randnum = rand() % m + 1;
    return randnum * SLOT;
};
```

图 8 二进制指数退避算法

当冲突次数 `collision_count` 小于 16 时, 选取冲突次数和 10 中较小者 `k`, 计算 $m=2^k-1$ 的 k 次幂-1, 在 $[1,m]$ 中随机选取一个整数与时间单位 `SLOT` 的乘积作为本次退避的时间, 返回值为时间单位数。

5.1.2 Transmission line

```
//data on transmission line
extern vector<int> transmitting_data1;
//free 0 busy 1 jam 2;
extern int transmission_status1;
//line2
extern vector<int> transmitting_data2;
extern int transmission_status2;
```

图 9 总线设计

设计中设置了两个冲突域, 每个冲突域可看作用总线式的连接方式, 所以设置两条总线, `line1` 和 `line2`。每条总线上用关键的总线数据和总线状态来代表, 分别对应 `transmitting_data` 和 `transmission_status`。由于总线需要全局访问, 由主机和交换机共同访问, 所以使用 `extern` 声明为全局变量。总线数据使用 `vector<int>` 类型, 总线状态声明为 `int` 类型, 其中 0 代表总线空闲, 1 代表总线忙, 2 代表总线上发生了冲突。

5.1.3 Mutex

```
//ensure consistence in transmission1 info
extern mutex one;
//ensure consistence in transmission2 info
extern mutex two;
//ensure complete window/file print info
extern mutex out;
```

图 10 mutex

由于设计中使用多线程，在主机和交换机访问总线并向上传输数据且更改总线状态时，由于总线由数据线和状态位两部分组成，为了避免数据和状态在不同的线程中修改，使用 `one` 和 `two` 分别控制在修改总线时操作的原子性。

`Out` 锁作为整个设计程序中的全局变量，控制程序在任何地方的打印输出的原子性，避免打印结果不完整。

同样的，其使用 `extern` 声明为全局变量共整个程序访问。

5.2 frame

帧结构定义如下，使用 `vector<int>` 类型，其中每一位使用二进制表示。

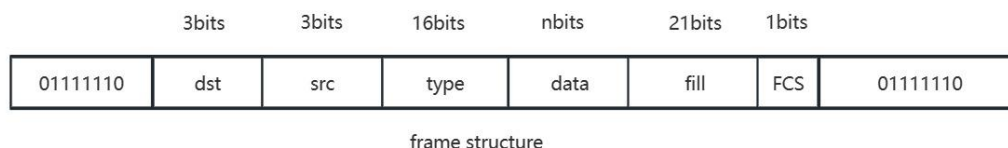


图 11 帧结构

5.2.1 Get_frame

```
vector<int> getframe(vector<int>data, int source, int receiver, vector<int>type);
```

图 12 getframe

`Get_frame` 打包帧结构，按帧定义的顺序依次将各字段送入 `frame` 中。首先将帧前 8 位的标识符‘01111110’送入；再设置目的地的 `MAC` 地址，由于真实的 `MAC` 有 48 位之多，且相对固定，本次设计使用三位的二进制模拟；之后送入源 `MAC` 地址；之后设置协议类型，使用传入参数中的 `type` 字段；之后设置帧中的数据，数据由主机产生使用 `data` 字段传参，注意在打包数据时需要在五位连续的 1 后

添加一个 0 用来实现面向比特的同步协议，与前后标识符区别开。代码如下

```
//set data
int count=0;//记录5个数据
for (int i = 0; i < data.size(); i++) {
    int x = data[i];
    if (x == 1) {
        count++;
        onecount++;
    }
    else
        count = 0;
    frame.push_back(x);
    bitcount++;
    //每5个连续的1后加0
    if (count == 5){
        frame.push_back(0);
        count = 0;
    }
}
```

图 13 set data

之后设置 21 位的填充码段，用来模拟以太网中帧的最小长度。以太网中最小长度也相对较长，设计中模拟为最小帧长 60bit。当数据字段为空时，其余所有字段的长度和为 39bit， $60-39=21$ bit。本函数中起初设置整形变量 onecount 用于记录所包入的内容中 1 的个数，在填充为后根据 onecount 的值判断插入 0 或者 1 的奇偶校验位以实现帧的奇偶校验。最后插入八位的标识符‘01111110’。

5.2.2 Reverseframe

```
vector<int> reverseframe(vector<int> frame)
```

图 14 reverse frame

将所传入的帧 frame 逆序后返回。

5.2.3 Decode

```
vector<int> decode(vector<int>frame)
```

图 15 decode

将接收到的数据帧解包, 返回原先的数据。根据帧结构截取特定长度的二进制位, 注意由于使用‘01111110’作为前后标识符, 解包时应将五个连续 1 后的 0 去掉。

具体实现代码如下。

```
vector<int> data;
vector<int> rframe;
int count=0;
rframe = reverseframe(frame);
for (int i = 30; i < rframe.size() - 30; i++) {
    if (rframe[i] == 1) {
        count++;
    }
    else {
        count = 0;
    }
    //每五个连续的1后的0去掉
    if (count == 5 && rframe[i] == 0) {
        count = 0;
        continue;
    }
    data.push_back(rframe[i]);
}
return data;
```

图 16 decode 函数实现

5.2.4 Oddcheck

```
bool odd_check(vector<int> frame)
```

图 17 odd_check

对得到的帧进行奇校验, 对接受到的帧逐位进行异或操作, 结果为 0 说明出错, 若结果为 1 说明没有发生偶数位错误。具体实现代码如下。

```
bool result=0;
for (int i = 0; i < frame.size(); i++) {
    result ^= frame[i];
}
return result;
```

图 18 odd_check 函数实现

5.3 PC

PC 模块包括主机的一些数据，例如 MAC 地址、需要发送的数据、缓冲区等。PC 模块实现了主机在以太网中一系列操作。PC 根据 CSMA/CD 协议进行发送帧。同时在监听总线时使用 1-坚持的监听算法，即发现总线不空闲时一直监听。当然在遇到冲突时应该使用二进制退避算法等待一定时间。除了发送外，主机还应具备接受功能，在接受帧时，总线上各主机监听总线上的数据，如果发现是发送给自己的数据就将其保存下来，若发现不是就丢弃。保存的数据发到主机的接受缓冲区中，接受缓冲区设置一定的大小，当接受主机的接受速度不及发送方发送速度 时，应通过缓冲区有所展示，即缓冲区满不得不将多出的帧丢弃。另外，主机应对接收到的帧解包，将其原本的数据还原。

根据以上描述，应对各主机设置三个不同的线程分别执行不同的功能。其分别为监听与发送线程、接收和过滤线程、解包与还原线程。

5.3.1 Listen_and_send

程序流程图

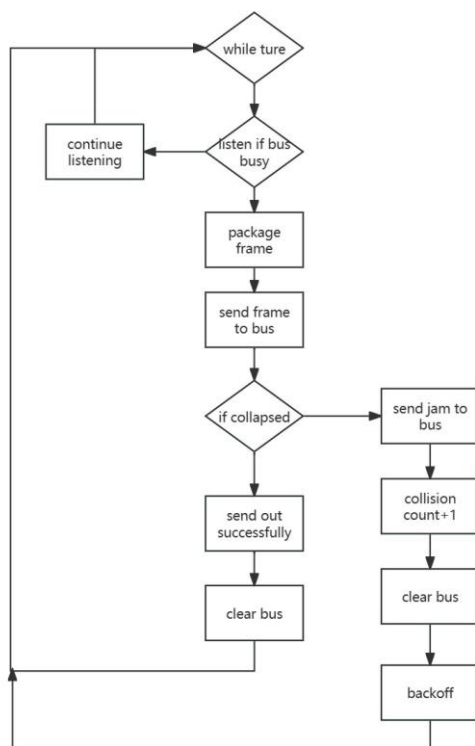


图 19 listen and send 流程图

算法伪代码

Listen and send

```
Define jam_data 32{1}
```

While true:

Host state = listen

If data on bus is empty: //bus is free

Host state = send

Send data to bus

Update bus state //to which bus depends on port number

Wait to and from time

If data on bus is data sent://no collision

Send out successfully

Clear bus // which bus depends on port number

Else // collapsed

Detected collision

```

Bus data = jam
Update bus state
Collision count +1
Clear bus //which bus depends on port number
If collision count < max
    backoff
    Host state = wait
Else send failure
    Collision count reset
    Clear bus //which bus depends on port number

Else 1-persist listen

Return
    
```

算法 1 PC 监听并发送流程

代码实现与介绍

```

status = HOST_LISTEN;
vector<int> transmitting_data;
if (switch_port == 1)
    transmitting_data = transmitting_data1;
else
    transmitting_data = transmitting_data2;
if (transmitting_data.empty()) {
    
```

图 20 listen

首先主机在相连的主线上监听，查看总线是否空闲，具体监听对象取决于主机位于哪一个冲突域中，即连接交换机的哪一个端口，若总线上数据为空，说明总线空闲，可以发送数据；若总线数据不空，则继续 while 循环持续监听，实现 1 坚持的监听算法。

```

status = HOST_SEND;
//sending
std::this_thread::sleep_for(std::chrono::milliseconds(10)); //simulate transmitting time
//package frame
get_send_frame();
if (switch_port == 1) {
    one.lock();
    transmitting_data1 = send_frame;
    transmission_status1 = 1;
    one.unlock();
}
else {
    two.lock();
    transmitting_data2 = send_frame;
    transmission_status2 = 1;
    two.unlock();
}
std::this_thread::sleep_for(std::chrono::milliseconds(10)); //simulate transmitting time

```

图 21 send 并监听

若总线空闲，则进入发送状态。首先打包要发送的数据到帧中，根据所处总线将数据送往总线上，并更新总线状态为 busy。对总线操作时数据部分和状态部分应设置为原子操作，避免一个主机修改了 data 字段另一主机修改了 status 字段的不一致现象。

```

if (switch_port == 1)
    transmitting_data = transmitting_data1;
else
    transmitting_data = transmitting_data2;
//success
if (transmitting_data == send_frame) {

```

图 22 得到总线数据

过段时间后，取下总线上的数据，若此时总线上的数据依旧等于要发送的数据，则判断发送成功。这里时模拟在总线上发送数据时边发送边监听，若在往返时间内依旧没有发现冲突（没有其他主机发送数据改变主线上的数据），则证明此次发送成功。发送成功后打印 send success

```

one.lock();
transmitting_data1.clear();
transmission_status1 = 0;
collision_count = 0;
one.unlock();

```

图 23 清空总线数据

之后将总线 reset，代表此次发送数据完成，模拟防止帧在总线上的反复传递。当然发送成功后冲突次数应该清空。同样，访问总线时应该使用锁。

```

//collision
else {
    status = HOST_WAIT;
    if (switch_port == 1) {
        one.lock();
        transmission_status1 = 2;//jam
        transmitting_data1 = jam_data;
        one.unlock();
    }
    else {
        two.lock();
        transmission_status2 = 2;//jam
        transmitting_data2 = jam_data;
        two.unlock();
    }
    collision_count++;
}

```

图 24 发送 jam 信号

若发生冲突主机应该等待，并发送一个冲突帧提醒其他主机此时发生了冲突。同时应该更改总线状态为冲突。

```

//recover
one.lock();
transmitting_data1.clear();
transmission_status1 = 0;
one.unlock();

```

图 25 清空总线

冲突过后将总线上 jam 清空，与之前类似，模拟防止帧在总线上反复传播。

```

std::this_thread::sleep_for(std::chrono::milliseconds(backoff(collision_count)));

```

图 26 二进制指数退避

之后主机使用二进制退避算法等待一定的时间后在重新进入循环监听状态。可注意到此处的等待时间换为 backoff()的返回值。

5.3.2 Get send_frame

```

void PC:: get_send_frame()

```

图 27 get_send_frame

用于得到主机本次要发送的帧。

```

vector<int> protocol_type;

```

图 28 protocol_type

首先定义结构存储协议类型，用作填充帧中的协议字段，协议可自行指定也可使用默认协议（用于模拟此字段，并非真正协议）。之后根据协议的代号将其每一位送入存储结构中。

```
//set data
int data_size = rand() % 10 + 1;
int bit;
for (int i = 0; i < data_size; i++) {
    bit = rand() % 2;
    data.push_back(bit);
}
```

图 29 打包数据

在主机中自动生成数据字段，本次设计为更好的独立出数据链路层的功能实现，在这里使用 01 二进制位的随机产生，用于模拟真实网络中上层数据包。

```
//set receiver
int RandomReceiver;
RandomReceiver = rand() % 5; //0-4
```

图 30 设置接收者

设置接受主机的 MAC 地址，这里与数据字段类似，使用随机数模拟上层协议指定的主机地址。

```
vector<int> frame = getframe(data, mac, RandomReceiver, protocol_type);
receiver = RandomReceiver;
send_data = data;
send_frame = reverseframe(frame);
```

图 31 保存信息

得到帧所需的字段后，将其作为参数传入打包帧的函数，得到要发送的帧。并将上述一系列数据给到主机类的成员数据结构中，便于之后使用。

5.3.3 Receive frame

程序流程图

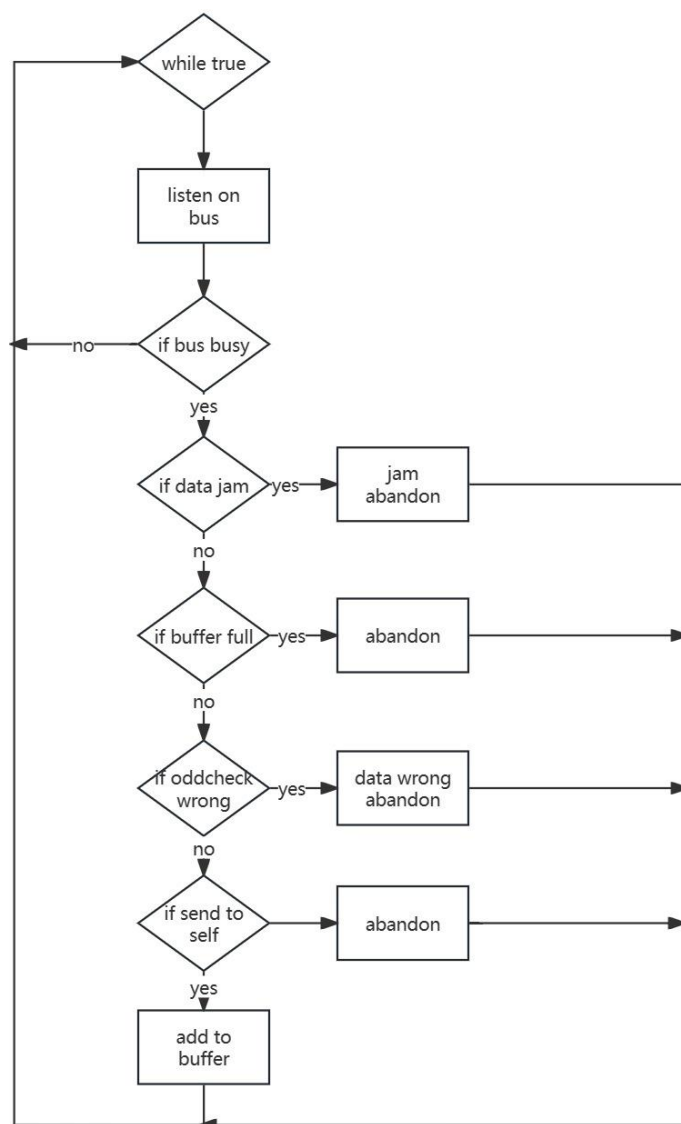


图 32 receive frame 流程图

算法伪代码

Receive frame

Define jam data

While true:

 Get data from bus

 If data on bus is not empty:

 If data is not jam data:

 If buffer out of range:

 Buffer full, abandon

```

Else
    If oddcheck data wrong:
        Abandon
    Else
        If receiver is self:
            Receive data into buffer
        Else abandon
    Else abandon jam
Else continue detecting

```

算法 2 PC 接收帧流程

代码实现与介绍

```

if(switch_port==1)
    received_data = transmitting_data1;
else
    received_data = transmitting_data2;

//if data detected
if (!received_data.empty()) {

```

图 33 监听总线

首先不断在总线上进行监听，若有数据则接受下来。使用 `received_data` 获取总线上的数据，若数据不为空说明监听到总线上有数据传输。

```

//if not jam : receive frame
if (received_data != jam_data) {
    buffer[mac].lock();
    int currentBufferSize = receive_buffer.size();
    buffer[mac].unlock();
    //buffer out of range
    if (currentBufferSize >= receive_buffer_size) {

```

图 34 判断不是 jam 且缓冲区不满

其次判断接受的数据是否为冲突控制信号 `jam`，若不是则判断本主机缓冲区是否已经满了，若时冲突控制信号则应该丢弃，若本主机接受缓冲区已满则无法接受也应该丢弃。

```

//there is buffer space
else {
    //odd check -- frame is wrong
    if (!odd_check(received_data)) {
        //abandon

```

图 35 奇校验

若缓冲区有空间则判断是否出差错，若奇校验出错，则判断帧中数据必然出差错，将其丢弃。

```
//frame data is intact
else {
    //data = decode(received_data);
    vector<int>reversed_data = reverseframe(received_data);
    int receiver = 0;
    receiver += reversed_data[8] * 4;
    receiver += reversed_data[9] * 2;
    receiver += reversed_data[10] * 1;
```

图 36 得到接收者

若判断数据帧奇校验没有出错，则得到该帧的接收者判断该帧是否是发送给自己的。

```
//receiver is self
if (receiver == mac) {
    buffer[mac].lock();
    receive_buffer.push(received_data);
    buffer[mac].unlock();
```

图 37 判断接收者

若该帧是发送给自己的，就将该帧保存到本机的缓冲区内。注意访问本机缓冲区需要互斥访问，互斥对象是数据解包线程，因为接受帧的线程和解包的线程都需要访问同一个主机对象的接受缓冲区，接收方需要向里写入接受的帧，解包方需要从里面读取数据来解码。

5.3.4 Decode frame

```
void PC::decode_frame()
```

图 38 decode_frame

程序流程图

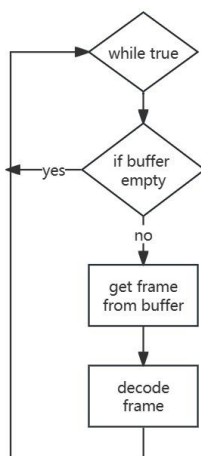


图 39 decode frame 流程图

算法伪代码

Decode frame

While true:

 If receive_buffer not empty:

 Get data from buffer front;

 Update buffer

 Decode data;

 Print info

 Else: persist on detecting

Return

算法 3 PC 解码帧流程

代码实现和介绍

```

buffer[mac].lock();
int bsize = receive_buffer.size();
buffer[mac].unlock();
if (bsize > 0) {
    ..
}
    
```

图 40 判断接收缓冲区是否未满足

不断获取缓冲区大小并判断是否大于零，即缓冲区中是否有接受的帧。注意缓冲区的访问需要加锁，具体原因与接收类似。

```

received_data = receive_buffer.front();
receive_buffer.pop();
buffer[mac].unlock();
reversed_data = reverseframe(received_data);
data = decode(reversed_data);
int sender = get_sender(reversed_data);

```

图 41 取帧解包

若缓冲区中有帧，则取出并进行解包，得到原来的数据。这里得到发送方是为了之后打印帧信息。注意缓冲区中得到的帧是原本总线上的数据，是与帧结构的定义相反的（由于帧的传输需要逆向传输），所以需要先使用 `reverseframe` 将帧反向。

```

out.lock();
cout << "PC" << mac << " received data:";
for (int i = 0; i < data.size(); i++) {
    cout << data[i];
}
buffer[mac].lock();
cout << " spared space:" << receive_buffer_size - bffsize + 1 << endl;
buffer[mac].unlock();
cout << endl;
out.unlock();

```

图 42 解包成功打印信息

对帧解包后的一些信息打印，比如接收的数据是什么、本机接收后剩余缓冲区的大小为多少。注意打印输出时需要使用锁保证输出完整性，访问缓冲区应用锁与接收线程互斥访问。

5.4 Exchange

5.4.1 交换机 switch 工作

程序流程图

```

//switcher work
void Exchange::exchange()

```

图 43 exchange

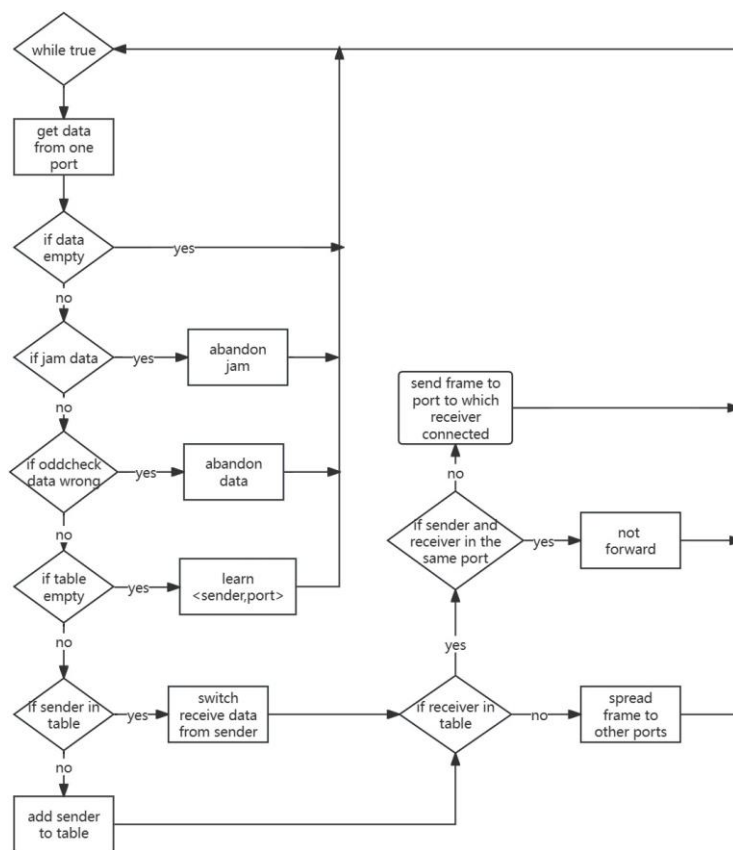


图 44 switch 工作流程图

算法伪代码

Switcher work

Define jam data

While true:

 If data on port not empty:

 If data not jam:

 If data oddcheck wrong:

 Abandon

 Else:

 Ge size t sender and receiver

 If table == 0:

 Table add <sender,port>

 For $i \leftarrow 1$ to table size:

 If table[i].sender == sender

Switch receive data from sender

If sender not in table:

Table add <sender, port>

For $i \leftarrow 1$ to table size:

If table[i].receiver == receiver:

If receiver.port == sender.port

Sender and receiver in the same port, not forward

Else: forward frame to receiver's port

If receiver not in table:

Spread frame to other ports

Return

算法 4 switch 工作流程

代码实现与介绍

```
//always at working
while (true) {
    //received data from line
    vector<int> received_data;
    //decide which data to get by self port
    if (port == 1) { //get data from line connected to port1
        Exchange::port1.lock();
        received_data = transmitting_data1;
        Exchange::port1.unlock();
    }
    else {
        Exchange::port2.lock();
        received_data = transmitting_data2;
        Exchange::port2.unlock();
    }

    //if there is data on line
    if (!received_data.empty()) {
```

图 45 监听总线是否有数据

不断循环判断总线上是否有数据，具体对应的总线取决于本端口与那根总线相连。如果总线上有数据，即接收到的数据不为空，则进行下一步操作。

注意无论在访问总线时，应对总线进行互斥访问，这里的互斥不是与其他工作模块互斥，而是交换机各端口间的互斥，因为一个交换机包含多个对象（线程），每个对象（线程）控制一个端口，需要防止交换机其中一个端口刚将数据送往一根总线上，另一个端口就误拿来作为刚刚接收的数据。所以 port1\2 两个互斥锁

设置为 static 类成员变量，需要各端口对象共享一把锁，避免上述这种自产自销的行为。

```
//if not jam warning
if (received_data != jam_data) {
    //if is damaged
    if (!odd_check(received_data)) {
```

图 46 判断是否 jam 或出错

若端口发现自己对应的总线上有数据，接着判断这个数据是否时 jam，若不是则是数据帧，判断其奇校验是否出错，若出错应该将其丢弃，不作任何处理。

```
else { //oddcheck right
    //reverse frame
    vector<int>reversed_data = reverseframe(received_data);
    //get receiver in frame
    int receiver = 0;
    receiver += reversed_data[8] * 4;
    receiver += reversed_data[9] * 2;
    receiver += reversed_data[10] * 1;
    //get source in frame
    int sender = 0;
    sender += reversed_data[11] * 4;
    sender += reversed_data[12] * 2;
    sender += reversed_data[13] * 1;
```

图 47 得到 sender 和 receiver

若校验正确，则将接收者和送者从帧中相应字段取出。

接下来应该根据交换表进行工作了，主要分为学习、扩散、转发等步骤。

```
tablemtx.lock();
//first get data from other host table empty now
if (Exchange::table.size() == 0) {
    int b[2];
    b[0] = sender;
    b[1] = port;
    //learn
    Exchange::table.push_back(b);
    out.lock();
    cout << "未知主机 添加PC"<<sender<<"进表" << endl;
    printtable();
    out.unlock();
}
tablemtx.unlock();
```

图 48 交换机首次学习

首先初始状态，交换表为空，遇到任何外来的帧其发送者都未在交换表中，应该将其加入表中，完成学习操作。注意对于交换表的范围需要互斥，因为各线程控制不同端口需要访问共享数据交换表。


```

//search table
flag = 0;
tablemtx.lock();
for (int i = 0; i < Exchange::table.size(); i++)
{
    int* a = table[i];
    //如果发送者在表内 if source in table
    if (a[0] == sender) {
        flag = 1;
        out.lock();
        cout << "switch receive data from PC" << sender<<endl;
        printtable();
        out.unlock();
    }
}

```

图 49 查找发送者是否在表

逐项查找交换表，查看发送者是否在表中，若在则不需要学习，只需等之后判断接收者是否存在与表中，但下面首先我们完成对发送者的判断。

```

//不在表内 learn
if(!flag){
    int* b=new int[2];
    b[0] = sender;
    b[1] = port;
    table.push_back(b);
    out.lock();
    cout << "未知主机 添加PC" << sender << "进表" << endl;
    printtable();
    out.unlock();
}

```

图 50 发送者不在表中则学习

若发送者不在表中，需要将其学习到表中，将发送者的端口和 MAC 地址加入表中，格式为<MAC,Port>，其中每个表项使用一个 2 位整型数组表示，两位分别表示 MAC 和 port。

上面我们完成了对发送者的判断，下面我们进行对接收者的判断。也即上面完成了学习维护转发表，下面进行转发或扩散。

```

//search table
flag = 0;
tablemtx.lock();
for (int i = 0; i < Exchange::table.size(); i++)
{
    int* a = table[i];
    //如果接收者在表内
    if (a[0] == receiver) {
        flag = 1;
    }
}

```

图 51 查找接收者是否在表

与发送者类似的是，这里也需要查找表项，查看接收者是否在表中，先看在表中的情况。

```
int tport = a[1]; //接收者的port
//如果接受者和发送者同一端口 ignore
//port 为此switch监听的端口
if (tport == port) {
    out.lock();
    cout << "接收者和发送者同一端口" << port << " 不转发" << endl;
    printtable();
    out.unlock();
}
```

图 52 接收者与发送者同端口

如果接收者在表中且和发送者处于同一端口，则交换机不进行转发，因为两者处于同一冲突域，两者相互发消息是可以看见的。

```
//否则将数据送往另一端口(接收者的端口)
else {
    if (port == 1) {
        Exchange::port2.lock();
        transmitting_data2 = received_data;
        Exchange::port2.unlock();
        out.lock();
        cout << "转发 port1 to port2 " << endl;
        printtable();
        out.unlock();
    }
}
```

图 53 接收者和发送者不同端口

若接收者在表中且不饿发送者处于同一端口，则需要转发。这里以 1 号端口为例，若其发现满足上述条件，则将其向接收者所在端口发送数据，这里的接收者处于 2 号端口。

```
//接收者不在表内，扩散到其他端口
if(!flag)
{
    //port1端口则扩散到2口
    if (port == 1) {
        Exchange::port2.lock();
        transmitting_data2 = received_data;
        Exchange::port2.unlock();
        out.lock();
        cout << "接收者" << receiver << "不在表内，扩散" << endl;
        printtable();
        out.unlock();
    }
}
```

图 54 接收者不在表中

结束接收者在表中，再看接收者不在表中的情况。若在表中查找不到接收者对应

的表项，则需要将接到的帧向其他所有端口扩散。这里以端口 1 为例，在端口 1 的帧中接收者不在交换表内，则向除端口 1 外的所有端口扩散该帧，这里只有端口 2。

六、关键问题及其解决方法

本单元将说明设计中一些关键的问题是如何解决的，同时也会描述一些设计过程中存在的错误以及是如何解决的。

1. 判断总线是否空闲

设置三种总线状态，分别为忙、空闲、冲突。总线设计成由两个部分组成，分别为状态和总线上的数据，且两者对应。总线状态为忙时，总线上存在数据且不为 jam；总线状态为空闲时总线上的数据为空；总线状态为冲突时，总线上的数据为 jam。当主机监听时，只需将总线上的数据取下并根据上述条件判断即可。

Define Transmission line

If transmission line is free:

 Transmitting data is none;

Else if transmission line is busy:

 Transmitting data is frame;

Else if transmission line is jam:

 Transmitting data is jam data;

算法 5 定义总线状态

2. 监听是否发生冲突

CSMA/CD 协议要求发送时便发送边监听是否发生冲突。设计中先将待发送的数据帧发往总线，即让总线数据部分等于发送帧，之后发送方等待一段时间（在总线上传输的往返时延），等待后再将总线上的数据取下与发送帧对比，若两者相等说明帧在总线上传输了往返路程后依旧没有错误，传输成功；若总线上数据与本机发送帧不同则说明发生了冲突。算法描述如下

Detect collision

Transmitting frame = sent frame

Wait for to and from time

If transmitting frame == sent frame:

No collision

Else collision happened

算法 6 检测冲突

3. 二进制指数退避算法

传输遇到冲突时, 主机需要向总线发送冲突信号, 并且遵循二进制退避算法等待。描述如下。

Collision detected

If collapsed:

Collision count++;

Send jam data to transmission line;

Wait for time (backoff*time slot);

算法 7 冲突后二进制退避

4. 1 坚持 CSMA 监听

主机进行监听时, 若总线忙, 则一直监听, 遇到空闲时立即发送

1 - persist csma

If transimission line is busy:

Persist on listening

Else send frame immediately;

算法 8 1 坚持监听

5. 接收数据帧

接收总线上的数据帧时需要判断帧是否发给自己、帧是否出错、帧是否为 jam、若一系列检查通过则将其放到自己的接收缓冲区。具体描述在设计讲解处已给出, 这里不在赘述。

6. 解码数据帧

解码帧时从本地缓冲区拿到帧, 对帧解包, 具体描述在设计讲解处已给出, 这里不再赘述。

7. 收发双方速度不一致

真实情况下收发双方由于网络时延、拥塞程度等不一致，会导致双方速度不同，设计中使用缓冲区来模拟。在接收时将帧放到缓冲区中，若本机解码速度慢会导致缓冲区被迅速填满，填满后无法接收发送来的帧。描述如下

Simulate inconsistency in speed of sender and receiver

Receiver:

While true:

 Receive frame;

 If buffer full:

 Abandon

 Else add to buffer

End while

算法 9 模拟接收与发送速度不一致

8. 设置多线程模拟各设备同时运行

由于现实情况下网络上各主机和交换机同时运行，而单机上的程序不能，故设置多个线程模拟同时运行，尽管部分还会有相差。本次设置描述如下。

Multithread to simulate simultaneously working devices

For $i \leftarrow 0$ to 4:

 Start thread of PC;

For $i \leftarrow 0$ to 1:

 Start thread of switch port;

算法 10 多线程模拟多设备运行

9. 设置多线程模拟各设备同时运行多个功能

现实中主机会同时向网络上发送数据和接收并解码数据，交换机可以多个端口同时收发数据。设置多个线程模拟一台设备上的多个功能同时运行。

Multithread to simulate multifunctional devices

PC:

Start thread of sending frame;

Start thread of receiving frame;

Start thread of decoding frame;

Switch:

For $i \leftarrow 1$ to n :

 Start thread of each port;

算法 11 多线程模拟单设备多功能

10. 交换机各端口间共享交换表

由于一个交换机由多个端口，各个端口设置为一个对象，其应共享同一个交换机的交换表，起初设计中并未考虑此问题，只是启动了两个端口对象的线程函数，发现交换表打印时时长时短，且内容不同。分析后发现两个端口对象维持了类数据变量交换表的两个副本。所以将成员变量设置为 `static` 类型，所有对象维持一个副本。当然此交换表应设置互斥锁。

11. 交换机各端口互斥访问同一总线

一个交换机包含多个对象（线程），每个对象（线程）控制一个端口，需要防止交换机其中一个端口刚将数据送往一根总线上（转发），另一个端口就误拿来作为刚刚接收的数据（接收）。所以在类中加入 `port1\2` 两个互斥锁成员变量并设置为 `static` 类成员变量，需要各端口对象共享一把锁，避免上述这种自产自销的行为。

12. 各总线状态和数据一致性

由于设计中使用多线程，在主机和交换机访问总线并向上传输数据且更改总线状态时，由于总线由数据线和状态位两部分组成，为了避免数据和状态在不同的线程中修改，使用 `one` 和 `two` 分别控制在修改总线时操作的原子性。

七、设计结果

7.1 程序运行初始设置

本单元将说明程序运行的基础设置与初始化，包括主机的设置、交换机的设置，

以及各功能线程的启动。

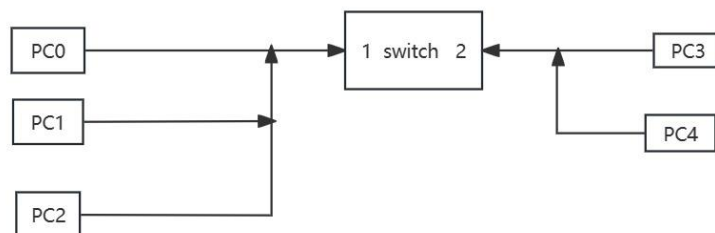


图 55 网络结构图

1) 在主函数中设置如下工作设备，完成初始化。

包含五个主机编号依次为 0、1、2、3、4，接收缓冲区大小均为 3，PC0、1、2 处于交换机的 1 号端口，PC3、4 处于交换机的 2 号端口。

一个交换机，包含 2 个端口对象，分别初始化端口号为 1、2。

```

PC PC0(0, 3, 1);
PC PC1(1, 3, 1);
PC PC2(2, 3, 1);
PC PC3(3, 3, 2);
PC PC4(4, 3, 2);
Exchange switcher1(1);
Exchange switcher2(2);
  
```

图 56 创建主机和交换机对象

2) 每个主机创建 3 个线程分别用来模拟监听与发送帧、接受帧到缓冲区、从缓冲区读取接收的帧并解包。

一个交换机包含两个端口对象，每个对象启动一个线程运行。

```
//warning use 'ref' to ensure unique object
thread port1(&Exchange::exchange, ref(switcher1));
thread port2(&Exchange::exchange, ref(switcher2));
//每台主机三个线程 发送帧 接受帧 解包帧
thread sender0(&PC::listen_and_send, ref(PC0));
thread receiver0(&PC::receive_frame, ref(PC0));
thread decoder0(&PC::decode_frame, ref(PC0));

thread sender1(&PC::listen_and_send, ref(PC1));
thread receiver1(&PC::receive_frame, ref(PC1));
thread decoder1(&PC::decode_frame, ref(PC1));

thread sender2(&PC::listen_and_send, ref(PC2));
thread receiver2(&PC::receive_frame, ref(PC2));
thread decoder2(&PC::decode_frame, ref(PC2));

thread sender3(&PC::listen_and_send, ref(PC3));
thread receiver3(&PC::receive_frame, ref(PC3));
thread decoder3(&PC::decode_frame, ref(PC3));

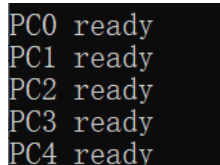
thread sender4(&PC::listen_and_send, ref(PC4));
thread receiver4(&PC::receive_frame, ref(PC4));
thread decoder4(&PC::decode_frame, ref(PC4));
```

图 57 创建工作线程

7.2 PC 运行结果及介绍

本模块将介绍 PC 主机运行时独自的运行结果，包括创建 PC、接收数据正确性、发送数据正确性、解码数据帧的正确性。

- 1) 各主机在创建对象时在构造函数中进行初始化。在初始化时会将主机各成员变量赋以初始值，比如在创建时传入构造函数的 MAC 地址、接收缓冲区大小、所在交换机的端口号。构造完成后打印“PC ready”



```
PC0 ready
PC1 ready
PC2 ready
PC3 ready
PC4 ready
```

图 58 PC ready

- 2) 当 PC 由于同时发送遇到冲突时会监听到冲突。冲突表现为 PC_i 向主线上传数据，过段时间总线上数据与本身所传不一致。下图为 PC₁、PC₂、PC₃ 发送的数据与总线上不一致，即检测到了冲突。


```
PC1 detected collision
PC2 detected collision
PC3 detected collision
```

图 59 PC detected collision

- 3) PC 成功发送。当发送的数据与过段时间传输的数据一致，说明没有被冲突，发送成功。会显示主机发送成功的信息以及数据内容和源、目的地址。下图为 PC4、PC1 发送成功的输出。

```
PC4 send out successfully
10 from PC4 to PC0

PC0 send out successfully
10 from PC0 to PC0
```

图 60 PC send out successfully

- 4) PC 接收时发现总线上发生了冲突，会接受到 jam 信号并丢弃。Jam 信号设计为 32 位全一。若接收线程收到此信号会将其丢弃并打印相关信息。下图为 PC0-4 的输出。

```
PC1 received jam signal! abandon
PC2 received jam signal! abandon
PC4 received jam signal! abandon
PC3 received jam signal! abandon
PC0 received jam signal! abandon
```

图 61 PC received jam

- 5) 当 PC 成功接收并解码帧后会显示解码后的数据及源和目的主机 MAC，以及当前接收缓冲区还剩余多少。下图为 PC1 成功解码接收 0000011111 的打印输出。

```
PC1 received data:0000011111 spared space:3
```

图 62 PC received data

- 6) 当 PC 接收缓冲区满时会将帧丢弃。若发送过快而解码过慢会导致接收方的缓冲区迅速被填满，再次接收时会显示 buffer full 的信息并丢弃该帧。下图为 PC2 缓冲区满丢弃帧的输出。

```
PC2 receive buffer full! abandon
```

图 63 PC Buffer full

7.3 switch 运行结果及说明

本模块将介绍 switch 交换机的工作结果，包括自学习算法的执行正确性、根据转发表转发帧的正确性。

- 1) 交换机各端口对象在起初会初始化。在构造函数中设置端口号并打印输出“switcher ready”

```
switcher ready
switcher ready
```

图 64 switch ready

- 2) Switch 遇到发送方不在交换表内的情况时，会学习主机及对应端口（将其加入表），随着时间推移，各主机进行发送帧，交换表会学习的逐渐完整。这里的各主机发送帧应有快速随机性，确保在较短时间内交换表能够学习到足够的信息，为满足此在主机包帧时以随机数的形式产生目的 MAC 地址。

```
未知主机 添加PC3进表
mac:3 port:2
```

```
未知主机 添加PC1进表
mac:3 port:2
mac:1 port:1
```

```
未知主机 添加PC4进表
mac:3 port:2
mac:1 port:1
mac:4 port:2
```

```
未知主机 添加PC0进表
mac:3 port:2
mac:1 port:1
mac:4 port:2
mac:0 port:1
```

```
未知主机 添加PC2进表
mac:3 port:2
mac:1 port:1
mac:4 port:2
mac:0 port:1
mac:2 port:1
```

图 65 交换表学习过程图

- 3) Switch 遇到目的地址不在表内情况时，会将帧向其他所有端口扩散。下图是

接收者 0 和 1 不在表中时交换机将帧向其他端口扩散的情况。

```

接收者0不在表内，扩散
mac:3 port:2
mac:2 port:1
mac:4 port:2

接收者1不在表内，扩散
mac:3 port:2
mac:2 port:1
mac:4 port:2
    
```

图 66 接收者不在表中

- 4) Switch 遇到接收者和发送者处于同一端口的情况时，不会将帧转发。下图是 PC3 向总线上发送数据，不久后被交换机接收，查表后发现接收者和发送者位于同一端口，不需要转发。

```

PC3 send out successfully
001111001 from PC3 to PC4

接收者和发送者同一端口1 不转发
mac:3 port:2
mac:2 port:1
mac:4 port:2
    
```

图 67 接收者和发送者同端口

- 5) Switch 遇到接收者在转发表内，且接收者和发送者不在同一端口的情况，会将其转发。下图是 PC3 发送一帧给 PC2，不久后被交换机接收，查表后发现存在发送者 PC3，不需要学习；而接收者也在表中但位于 1 端口（和 PC3 不同端口），于是将帧转发到 1 端口。

```

PC3 send out successfully
1000 from PC3 to PC2

转发 port2 to port1
mac:3 port:2
mac:2 port:1
mac:4 port:2
    
```

图 68 接收者发送者不同端口，转发帧

7.4 综合结果说明

本单元将展示各主机和交换机功能线程间协调工作的正确性，包括接收的数据是否由另一主机发送等。

- 1) 仅仅单独查看各主机/交换机端口的运行结果显然不能验证程序的正确性，因

为并不知道主机/交换机中的数据是否真的是在总线上传输过来的。下面我们将综合上下文来看看输出结果是否能匹配。

```
PC3 detected collision
PC2 detected collision
PC0 detected collision
PC1 detected collision
PC4 received jam signal! abandon
PC1 received jam signal! abandon
PC0 received jam signal! abandon
PC3 received jam signal! abandon
PC2 received jam signal! abandon
```

图 69 collision and jam

- 2) 起初应该是 PC0、1、2、3、4 同时向上发送数据从而导致两条总线发生冲突，所以五个主机的接收线程在接收时会收到 jam 信号丢弃。

```
PC2 send out successfully
0000011111 from PC2 to PC1
PC3 detected collision
PC1 detected collision
PC4 detected collision
PC0 detected collision
PC2 detected collision
PC4 detected collision
PC1 send out successfully
11101010 from PC1 to PC2
PC3 detected collision
PC1 received data:0000011111 spared space:3
```

图 70 send and receive

- 3) 当 PC2 向 PC1 发送数据 0000011111 后，在不久后 PC1 收到了同样的数据 0000011111，说明了 PC1 成功从 PC2 接收到了数据。

```
PC4 send out successfully
010 from PC4 to PC3

PC3 detected collision
PC0 detected collision
PC2 detected collision

未知主机 添加PC4进表
mac:4 port:2
```

图 71 send and learn

- 4) PC4 向 PC3 发送数据，不久后交换机接收到了此帧，检查后发现表内没有发送者 PC4，将 PC4 加入转发表。

```
PC4 send out successfully
11 from PC4 to PC4

PC1 send out successfully
101 from PC1 to PC1

PC4 send out successfully
100011 from PC4 to PC0

PC1 received data:10111110 spared :

switch receive data from PC4
mac:4 port:2
mac:1 port:1
接收者2不在表内，扩散
mac:4 port:2
mac:1 port:1

switch receive data from PC4
mac:4 port:2
mac:1 port:1
接收者和发送者同一端口1 不转发
mac:4 port:2
mac:1 port:1
```

图 72 send and switch receive

- 5) PC4 向 PC4 发送数据帧，交换机收到此帧发现两者处于同一端口，不转发该帧。至于发送方 PC4，由于之前已经将 PC4 加入表中，故此次接收不需要学习，可以看到显示 switch receive data 说明正确接收数据没有学习。

7.5 主机日志记录

本单元将展示主机中对于工作日志的记录是否正确。

- 1) 若程序开始前选择写入日志，在运行结束后在工程目录会产生五个主机对应的日志。Error_log 记录了主机发送和接收的错误；received_log 记录了主机成功接收数据的描述；send_log 记录了主机成功发送数据的描述，
















 error_log0.txt	2024/1/21 9:40	文本文档	4 KB
 error_log1.txt	2024/1/21 9:40	文本文档	4 KB
 error_log2.txt	2024/1/21 9:40	文本文档	4 KB
 error_log3.txt	2024/1/21 9:40	文本文档	4 KB
 error_log4.txt	2024/1/21 9:40	文本文档	5 KB
 received_log0.txt	2024/1/21 9:40	文本文档	5 KB
 received_log1.txt	2024/1/21 9:40	文本文档	4 KB
 received_log2.txt	2024/1/21 9:40	文本文档	5 KB
 received_log3.txt	2024/1/21 9:40	文本文档	5 KB
 received_log4.txt	2024/1/21 9:40	文本文档	5 KB
 send_log0.txt	2024/1/21 9:40	文本文档	1 KB
 send_log1.txt	2024/1/21 9:40	文本文档	5 KB
 send_log2.txt	2024/1/21 9:40	文本文档	18 KB
 send_log3.txt	2024/1/21 9:40	文本文档	17 KB
 send_log4.txt	2024/1/21 9:40	文本文档	12 KB

图 73 logs

- 2) Send_log 记录了主机发送成功的记录，会显示数据的起始和终止，以及发送成功的信息。查看 PC0 的记录如下

**PC0 send out successfully
00 from PC0 to PC1**

**PC0 send out successfully
0100101 from PC0 to PC0**

**PC0 send out successfully
0101110010 from PC0 to PC4**

**PC0 send out successfully
0 from PC0 to PC1**

图 74 PC0 send log

- 3) Receive_log 记录了主机接收成功的记录，会显示主机接收并解码的数据内容，以及源主机地址，并打印剩余的缓冲区数量。查看 PC0 的 receive_log 如下。

PC0 received data:1110010000 from PC1 spared space:2

PC0 received data:0010010 from PC3 spared space:2

PC0 received data:100101100 from PC3 spared space:2

PC0 received data:100000 from PC3 spared space:1

图 75 PC0 receive log

- 4) Error_log 记录了主机从发送到接收所有的失败与错误，会显示具体错误类型与信息，包括 jam、collision、buffer full、oddcheck wrong。查看 PC2 的 error_log 如下。

PC2 received jam signal! abandon

PC2 receive buffer full! abandon

PC2 received jam signal! abandon

PC2 receive buffer full! abandon

图 76 PC2 error log

八、参考资料

[1]高传善. 计算机网络教程（第二版）. 高等教育出版社，2009.

九、验收时间及验收情况

验收时首先从**网络拓扑结构**开始讲解，完成了五主机一交换机的双冲突域网络结构，交换机具有两个端口，每个端口分别与三个和两个主机相连。之后是每个主机使用三个线程用来分别实现监听和发送功能、接收帧、解码帧的功能；交换机使用两个线程分别代表两个端口完成根据交换表转发功能，若在转发过程中发现表中数据不足则会根据自学习算法进行学习表项。

之后再到具体实现。首先是**主机的监听和发送功能**，主机使用 1 坚持 CSMA

算法对所连总线进行监听，若监听到空闲则立即发送数据，边发送边监听是否发生了冲突，若发生了冲突主机会向所连总线上发送 jam 信号，当然 jam 信号只会在自己的冲突域内传播因为交换机会隔离冲突，发送完 jam 信号后会使用二进制指数退避算法选择合适的时间退避等待，等待结束后再进行监听，若没有冲突则判断为发送成功。

其次是**主机的接收功能**。主机接收线程不断监听总线，若总线上有数据则将其接收，然后判断是否为 jam、是否出错、缓冲区是否满、是否发给自己，若上述条件全部通过则将其放到自己的接收缓冲区内。

再次是主机的解包帧功能。主机解帧线程不断查看本机缓冲区内是否有接收到的帧，若有则将帧解包成原始数据。

交换机的实现：交换机有两个端口，每一个端口为一个 exchange 对象，且占有一个线程。这样实现的原因是交换机需要多个端口可以同时工作，其中一个端口在接收的同时，另一个端口也应该能工作。交换机各端口不断监听总线，若存在数据则接收，接受下来查看源地址和目的地址，再从交换表中找，若没有找到则使用自学习算法加入表项或扩散到其他端口；若找到则根据表项记录进行转发。随着时间推移，接收交换的次数增多，每个主机都在交换表中占有一个表项。

最后讲解了特定功能的实现。首先是可以**模拟实现接发速率不一致带来的数据帧丢失**，但并不会进行重发，因为以太网并不会可靠传输。这里是用接收缓冲区实现的，每台主机存在一个接收缓冲区（使用队列实现），在初始化时设置其大小。每次接收数据时先不着急解码，将帧放入缓冲区，启用另一个线程从缓冲区内取帧解码，若缓冲区满则丢弃再来的帧。

其次是讲解了**多线程引发的数据不一致问题和解决方法**。使用多个互斥锁对共享数据访问，确保多个线程访问时不会冲突。

然后验收老师询问接收发送为何要设置两个线程，如何实现边发边听？

这里设置两个线程并不是分别用来监听和发送的。主机在监听发送的期间会收到来自其他主机发送过来的数据，接收线程是用来接收此数据帧的。而监听和发送处在一个线程中，首先发送是将数据送到总线上，使用延时函数模拟帧在总线上传输过程的时间，若传输时间到再取下总线上的数据（边发边听）依旧是本机送上去的数据，则判断帧在总线上已经往返了一个来回依旧没有与其他帧冲

突，而总线上的其他主机一定监听到了该帧（因为一个来回一定经过其他主机），则判断发送的帧没有冲突。

十、设计体会

本次设计完成了一个简单的网络结构拓扑并在其上实现了模拟以太网数据链路层的功能。起初并没有实现交换机，但交换机是属于数据链路层必不可少的设备，且主机间发送数据需经过它。若没有交换机则可以实现但冲突域简单的网络结构，若实现交换机则可以扩展实现更加复杂的网络结构。同时本次设计中以太网帧的定义也不够完全，比如采用简短版的 **MAC** 地址以及以太网中采用的 **CRC** 循环冗余码也替换成奇偶校验码。但众多线程的使用和控制增进了多线程程序编写的能力；同时对网络中数据链路层的原理以及实现有更深刻的理解。