**北京邮电大学《计算机网络》课程实验报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **实验**  **名称** | 数据链路层滑动窗口协议的设计与实现 | | **学 院** | 计算机 | **指导教师** | 高占春 |
| **班 级** | **班内序号** | **学 号** | | **学生姓名** | **成绩** | |
| **2023211303** | **02** | **2023210710** | | **陈子容** |  | |
|  |  |  | |  |  | |
|  |  |  | |  |  | |
| **实**  **验**  **内**  **容** | 利用所学数据链路层原理，自行设计一个滑动窗口协议，在仿真环境下编程实现有噪音信道环境下两站点之间无差错双工通信。信道模型为8000bps全双工卫星信道，信道传播时延270毫秒，信道误码率为10-5，信道提供帧传输服务，网络层分组长度固定为256字节。  本次实验选用的滑动窗口协议有未使用NAK通知机制的回退N步协议与使用了NAK通知机制的选择重传协议。 | | | | | |
| **学生**  **实验**  **报告** | （详见“实验报告和源程序”册） | | | | | |
| **课**  **程**  **设**  **计**  **成**  **绩**  **评**  **定** | **评语**:  **成绩**:  指导教师签名：    年 月 日 | | | | | |

注：评语要体现每个学生的工作情况，可以加页。

目录

[1. 实验内容和实验环境描述 1](#_Toc6880)

[1.1. 实验任务 1](#_Toc21991)

[1.2. 实验内容 1](#_Toc22109)

[1.3. 实验环境 1](#_Toc30732)

[2. 软件设计 1](#_Toc17083)

[2.1. 数据结构 1](#_Toc18603)

[2.2. 模块结构 3](#_Toc25940)

[2.3. 算法流程 5](#_Toc8864)

[3. 实验结果分析 6](#_Toc5331)

[3.1. 基本功能实现 6](#_Toc21541)

[3.2. 程序健壮性分析 6](#_Toc22498)

[3.3. 协议参数选取 6](#_Toc15685)

[3.4. 理论分析 9](#_Toc23097)

[3.5. 实验结果分析 10](#_Toc32561)

[3.6. 存在的问题 11](#_Toc13121)

[4. 研究和探索的问题 11](#_Toc10969)

[4.1. CRC 校验能力 11](#_Toc21156)

[4.2. CRC 校验和的计算方法 12](#_Toc4540)

[4.3. 程序设计方面的问题 15](#_Toc19070)

[4.4. 软件测试方面的问题 17](#_Toc12748)

[4.5. 对等协议实体之间的流量控制 18](#_Toc11283)

[4.6. 与标准协议的对比 18](#_Toc15830)

[5. 实验总结和心得体会 18](#_Toc4917)

[5.1. 实际上机调试时间 18](#_Toc3540)

[5.2. 编程工具相关问题 18](#_Toc7604)

[5.3. 编程语言相关问题 18](#_Toc24329)

[5.4. 协议相关问题 19](#_Toc14236)

[5.5. 开发库相关问题 21](#_Toc18914)

[5.6. 总结 21](#_Toc10757)

[6. 源程序文件 22](#_Toc17385)

[6.1. Go-Back-N 协议 22](#_Toc2498)

[6.2. 选择重传协议 26](#_Toc30292)

[参考文献 31](#_Toc19513)

# 实验内容和实验环境描述

## 实验任务

利用所学数据链路层原理，自己设计一个滑动窗口协议，在仿真环境下编程实现有噪音信道环境下两站点之间无差错双工通信。

信道模型为 8000bps 全双工卫星信道，信道传播时延 270 毫秒，信道误码率为 ，信道提供字节流传输服务，网络层分组长度固定为 256 字节。

滑动窗口机制的两个主要目标：(1) 实现有噪音信道环境下的无差错传输; (2)充分利用传输信道的带宽。

可选的协议类型为“不搭载 ACK 的 Go-Back-N 协议”，“使用搭载 ACK 技术的 Go-Back-N 协议”，“选择重传协议”，要求必须是全双工通信协议。

## 实验内容

选择“使用搭载 ACK 技术的 Go-Back-N 协议”与“选择重传协议”。

根据信道实际情况合理地为协议配置工作参数，包括滑动窗口的大小和重传定时器时限以及 ACK 搭载定时器的时限，从而充分利用传输信道的带宽。

## 实验环境

Windows 环境 PC 机

Microsoft Visual Studio 2022集成化开发环境。

# 软件设计

## 数据结构

### 帧结构

#### 帧结构图示

*/\**

*DATA Frame*

*+=========+========+========+===============+========+*

*| KIND(1) | SEQ(1) | ACK(1) | DATA(240~256) | CRC(4) |*

*+=========+========+========+===============+========+*

*ACK Frame*

*+=========+========+========+*

*| KIND(1) | ACK(1) | CRC(4) |*

*+=========+========+========+*

*NAK Frame*

*+=========+========+========+*

*| KIND(1) | ACK(1) | CRC(4) |*

*+=========+========+========+*

*\*/*

#### 自定义结构体

源程序中帧结构如下

struct FRAME {

unsigned char kind;

unsigned char ack;

unsigned char seq;

unsigned char data[PKT\_LEN];

unsigned int padding;

};

应按如下定义方式理解

struct FRAME {

unsigned char kind;

unsigned char ack;

union {

struct {

unsigned char seq;

unsigned char data[PKT\_LEN];

unsigned int padding;

} frame\_data;

struct {

unsigned int padding;

} frame\_ack;

struct {

unsigned int padding;

} frame\_nak;

};

};

成员用途同教材。

### Go-Back-N 协议相关变量

static unsigned char frame\_nr = 0;

static unsigned char buffer[MAX\_SEQ + 1][PKT\_LEN];

static unsigned char nbuffered = 0;

static unsigned char ack\_expected = 0;

static boolean phl\_ready = false;

static unsigned char frame\_expected = 0;

变量名和变量所起的作用同教材。

### 选择重传协议相关变量

static unsigned char out\_buf[NR\_BUFS][PKT\_LEN];

static unsigned char nbuffered = 0;

static unsigned char ack\_expected = 0;

static boolean phl\_ready = false;

static unsigned char in\_buf[NR\_BUFS][PKT\_LEN];

static unsigned char arrived[NR\_BUFS] = { 0 };

static unsigned char frame\_expected = 0;

static boolean no\_nak = true;

变量名和变量所起的作用同教材。相比教材我们删去了用于表示窗口上界的变量，因为其可用的方式表示。但在Go-Back-N 协议中，我们保留了变量frame\_nr，其在选择重传协议中用于表示发送窗口上界。具体原因见5.4.1。

## 模块结构

### between()函数

static boolean between(unsigned char a, unsigned char b, unsigned char c)

{

return (((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a)));

}

该函数用于检查帧b是否位于窗口[a, c)内。

### put\_frame()函数

static void put\_frame(unsigned char\* frame, int len)

{

\*(unsigned int\*)(frame + len) = crc32(frame, len);

send\_frame(frame, len + 4);

phl\_ready = false;

}

该函数用于在长度为len的帧frame末尾加上校验字段，并发送至物理层。

### framing()函数

static void framing(unsigned char kind, unsigned char arg)

{

struct FRAME s;

s.kind = kind;

s.ack = (frame\_expected + MAX\_SEQ) % (MAX\_SEQ + 1);

if (kind == FRAME\_DATA) {

s.seq = arg;

memcpy(s.data, out\_buf[arg % NR\_BUFS], PKT\_LEN);

put\_frame((unsigned char\*)&s, 3 + PKT\_LEN);

start\_timer(arg, DATA\_TIMER);

}

else {

if (kind == FRAME\_NAK)

no\_nak = false;

put\_frame((unsigned char\*)&s, 2);

}

stop\_ack\_timer();

}

该函数用于发送kind类型帧。若发送的帧为数据帧，则参数arg表示发送帧编号。

Go-Back-N协议不会发送NAK帧。若发送数据帧，则arg总为frame\_nr，NR\_BUFS为MAX\_SEQ + 1，函数结束后frame\_nr应循环自增。故简化代码如下

static void framing(unsigned char kind)

{

struct FRAME s;

s.kind = kind;

s.ack = (frame\_expected + MAX\_SEQ) % (MAX\_SEQ + 1);

if (kind == FRAME\_DATA) {

s.seq = frame\_nr;

memcpy(s.data, buffer[frame\_nr % (MAX\_SEQ + 1)], PKT\_LEN);

put\_frame((unsigned char\*)&s, 3 + PKT\_LEN);

start\_timer(frame\_nr, DATA\_TIMER);

INC(frame\_nr);

}

else

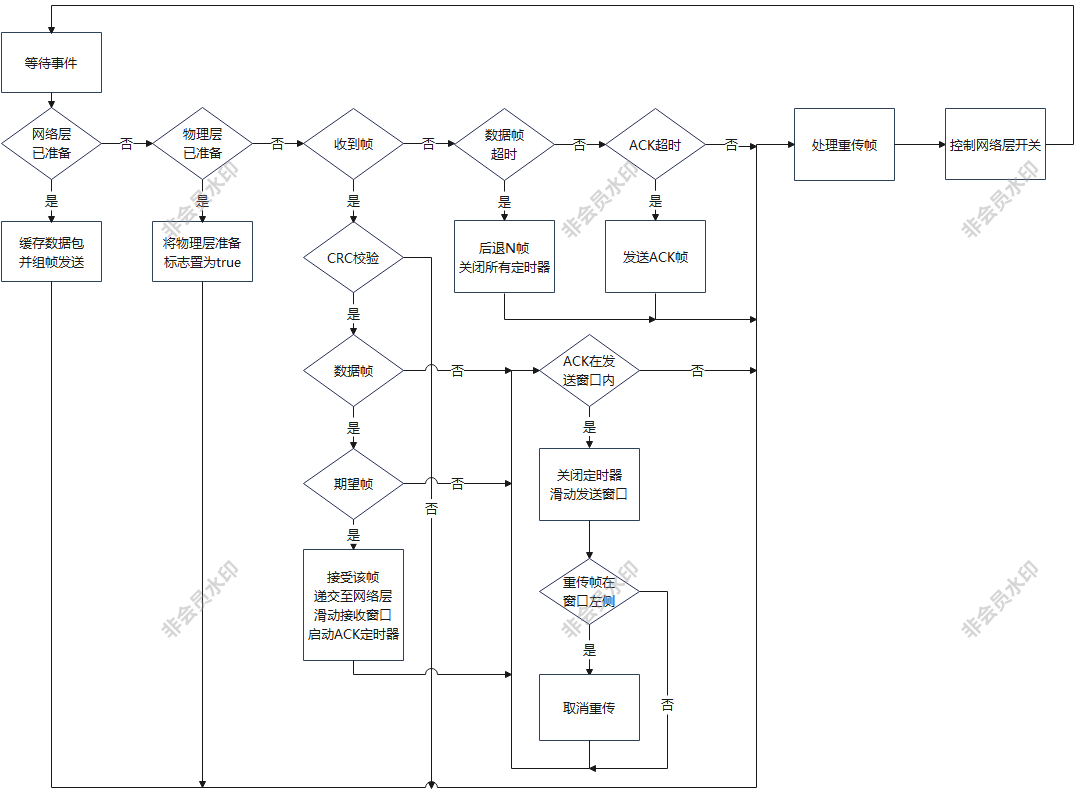
put\_frame((unsigned char\*)&s, 2);

stop\_ack\_timer();

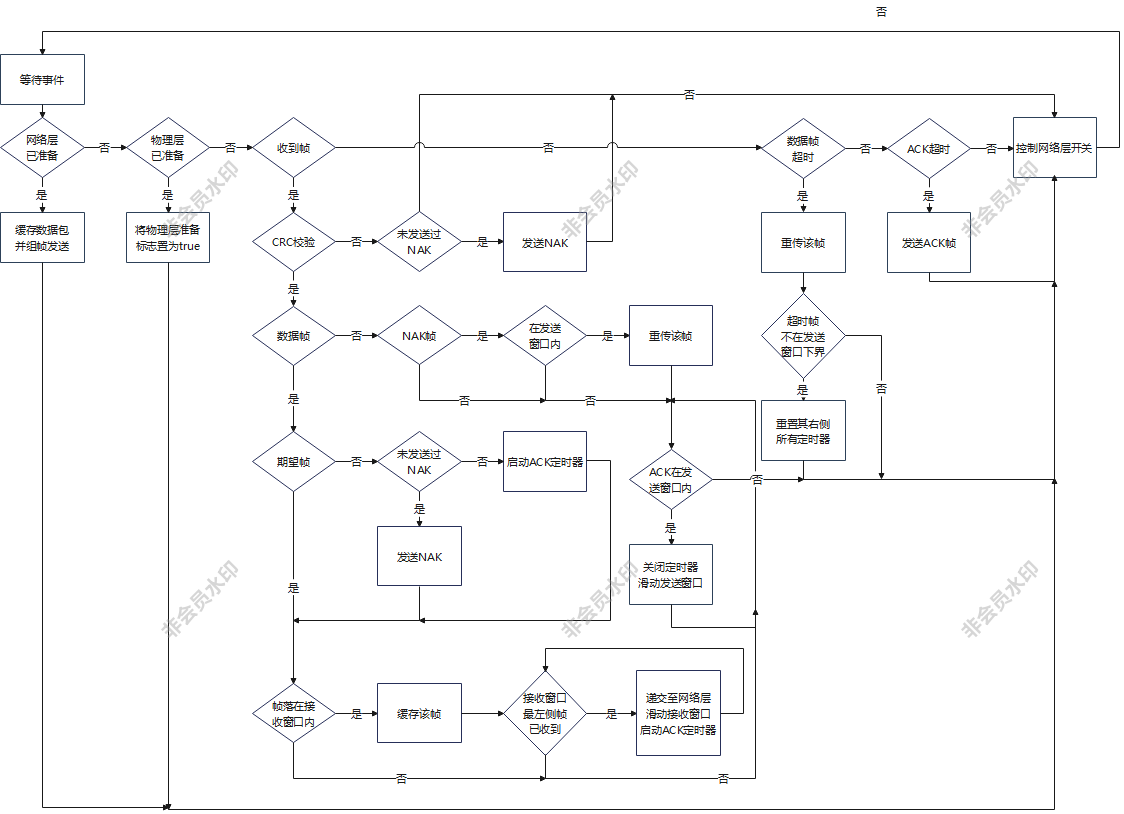
}

## 算法流程

### Go-Back-N协议



### 选择重传协议



# 实验结果分析

## 基本功能实现

两个协议均实现了有误码信道环境中无差错传输功能。

## 程序健壮性分析

两个协议均可在的误码率下运行20000秒以上，且通信双方网络层均能稳定收到数据包。可以初步认为协议能够可靠地长时间运行。

## 协议参数选取

由2.1.1可得，控制帧的传输时延为

远小于信道传播时延。故在后续的分析中有时会忽略控制帧的传输时延，即认为，并将简记为。

控制帧出错的概率为

这是极小概率事件。故在后续的分析中有时会假设控制帧不会出错，即认为，并将简记为。

### 滑动窗口大小

假设所有帧中每一位都是有效位。在无差错信道上有

其中

要使得

解得

考虑到信道为有差错信道，不应取接近的值（例如取）。考虑到实验中有定时器范围的限制，这导致

故在Go-Back-N协议中应有

即

而在选择重传协议用应有

即

为了调试方便，统一取，即，。

### ACK搭载定时器的时限

由3.3.1，

在-f模式下，网络层总是每263ms发送一个数据报。若取使得

则可保证每个ACK都可被搭载。若逆向流量很轻，我们希望当接收方收到数据帧时能够尽可能早地发送ACK（见3.3.3），故应尽可能小，故取。

### 重传定时器的时限

#### Go-Back-N协议

Go-Back-N协议无请求重传机制，接收窗口大小为1。对于一个错误帧，从其开始发送到重传定时器超时中间的时间都是浪费的。故应尽量小。

正确到达的帧不应被重传，这要求重传定时器不应在收到ACK前超时，故当未搭载ACK时有

搭载ACK时有

其中为接收方想要发送ACK时的等待时间。实验测得当时，协议在-f模式下不会发生重传定时器在收到ACK前超时的情况，故取 。

#### 选择重传协议

选择重传协议有请求重传机制，且接收窗口大小充足。对于一个错误帧，其后续帧的正确到达不会收到该帧的影响，故定时器的时限可以适当放宽。但由于累计确认机制，其后续帧的确认帧会受该帧影响，如果定时器时限过短，可能会导致不必要的重传。现考虑如下情况，发送方发送但是丢失，接收方返回，发送方发送正确到达，由于累计确认机制，接收方不会返回。发送方收到后重传成功到达，接收方返回搭载的数据帧（后文在不引起歧义的前提下会将搭载ACK的数据帧简记为ACK）。若的定时器没有在正确到达前超时，就可以防止的不必要的超时重传。这要求

其中表示从错误帧到开始发送之间等待物理层的时间，取

得

故取。

## 理论分析

### 无差错信道

由3.3.1，考虑帧结构（见2.1.1），在无差错信道上有

### 有差错信道

#### Go-Back-N协议

数据帧出错的概率为

该帧发送后重传定时器超时前发送的帧个数为

考虑连续传输5帧，其中出错帧数超过1帧的概率为

这是小概率事件，在信道利用率分析时可不考虑。Go-Back-N协议中，这些帧与错误帧都需要重传，故信道利用率为

#### 选择重传协议

选择重传协议中，由于接收窗口大小充足，若一段时间内只有一个帧出错（见3.4.2.1），则只有该帧需要重传，故信道利用率为

## 实验结果分析

性能测试记录表如下

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 序号 | 命令选项 | 说明 | 运行时间(秒） | GoBackN算法  线路利用率(%) | | Selective算法  线路利用率(%) | |
| A | B | A | B |
| 1 | -–utopia | 无误码信道数据传输 | 1200 | 51.57 | 96.76 | 51.58 | 96.96 |
| 2 | 无 | 站点A分组层平缓方式发出数据，站点B周期性交替“发送100秒，停发100秒” | 1200 | 45.57 | 85.76 | 50.70 | 94.98 |
| 3 | –-flood --utopia | 无误码信道，站点A和站点B的分组层都洪水式产生分组 | 1200 | 96.95 | 96.95 | 96.96 | 96.97 |
| 4 | –-flood | 站点A/B的分组层都洪水式产生分组 | 1200 | 86.66 | 86.35 | 95.23 | 94.91 |
| 5 | --flood -–ber=1e-4 | 站点A/B的分组层都洪水式产生分组，线路误码率设为10-4 | 1200 | 42.07 | 41.61 | 74.12 | 72.62 |

实验得到性能与理论基本吻合，甚至有时会高于理论值，可能是运行时间1200秒仍旧不足导致测试数据不够准确，后续可以考虑延长测试时间或者采用更高效的测试方式。

## 存在的问题

程序在大部分测试方案得到的性能均接近理论值。由于Go-Back-N协议没有引入请求重传机制，当误码率较高时性能低于参考数据。当误码率为时，选择重传协议的性能虽接近理论值，但仍旧较低，此时应考虑采用纠错码技术，如Hamming校验。

# 研究和探索的问题

## CRC 校验能力

### CRC 校验理论上不可能 100%检出所有错误。怎样说服客户相信我的系统能够实现无差错传输？

CRC校验的漏检概率由翻转比特数最小的漏检错误主导[1]。教材中提到，本实验使用的生成多项式，即

的Hamming距离为4，且仅存在一种4位错会被漏检[2]。故对于一帧，发生错误且不被CRC校验检出的概率可用发生该4位错的概率估计，即

如果传输一个分组途中出错却不能被接收端发现，算作一次分组层误码。则发生一次分组层误码事件所需的平均帧数为

若该客户使用本次实验描述的信道，客户的通信系统每天的使用率 50%，即信道利用率50%，则发生一次分组层误码事件所需的平均时间为

这是120倍的宇宙年龄。

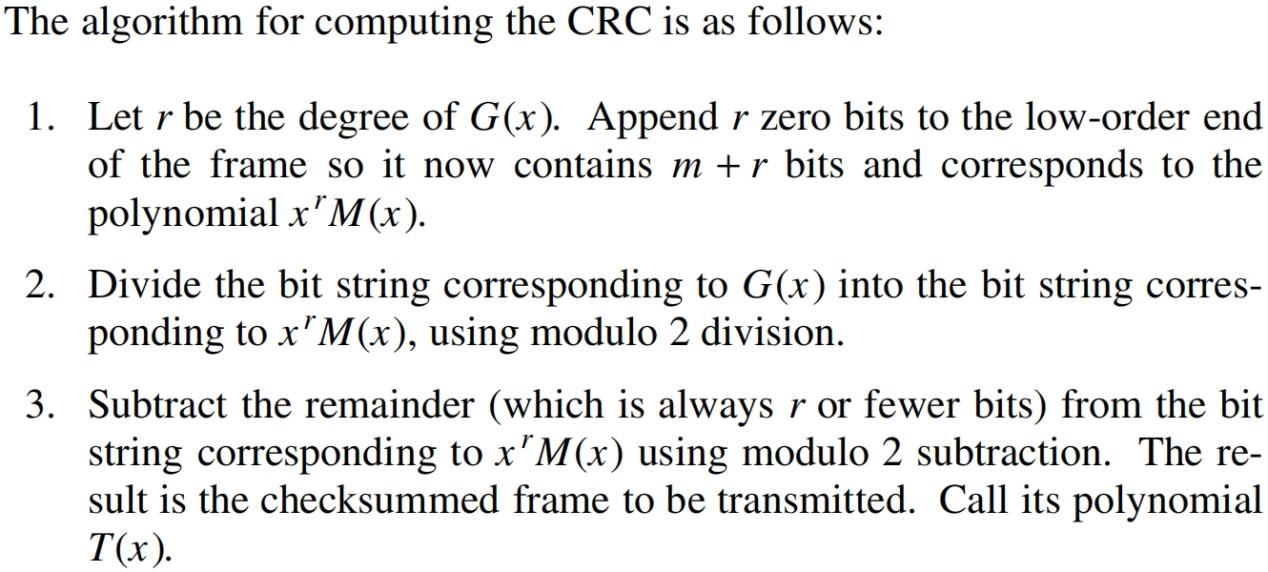
### 如何降低分组层误码率？需要什么代价？

可以考虑使用更强的校验码，如CRC64。代价是会增加冗余信息，导致信道利用率降低。

## CRC 校验和的计算方法

### 库函数crc32()与教材算法等效

教材算法如图所示



据此易得如下代码

unsigned int crc32\_update(unsigned char c) {

unsigned int crc = c;

crc <<= 24;

for (int i = 0; i < 8; i++) {

if (crc >> 31) {

crc <<= 1;

crc ^= 0x04c11db7;

}

else

crc <<= 1;

}

return crc;

}

unsigned int crc32(unsigned char\* buf, int len)

{

unsigned int crc = 0x00000000L;

while (len--)

crc = (crc << 8) ^ crc32\_update((crc >> 24) ^ \*buf++);

return crc;

}

其中函数crc32\_update()用于计算一字节的CRC32值。

注意到由于CRC算法涉及模2除法，该算法总是关注高位数据。使用C语言处理高位数据需要使用繁琐的移位运算，例如上述代码中频繁出现的crc <<= 24，crc >> 31，crc >> 24等。若考虑计算输入字节序列的每个字节的逆序的CRC32值而非直接计算原序列的CRC32值，算法将由关注高位数据变为关注低位数据，可以省去原有的繁琐的移位运算。代码如下

unsigned int crc32\_update(unsigned char c) {

unsigned int crc = c;

for (int i = 0; i < 8; i++) {

if (crc & 1) {

crc >>= 1;

crc ^= 0xedb88320;

}

else

crc >>= 1;

}

return crc;

}

unsigned int crc32(unsigned char\* buf, int len)

{

unsigned int crc = 0x00000000L;

while (len--)

crc = (crc >> 8) ^ crc32\_update((crc ^ \*buf++) & 0xff);

return crc;

}

可以看出，优化后的算法省去了很多移位运算。

同时注意到，程序的主要开销在于函数crc32\_update()的len次调用。事实上，函数crc32\_update()只有256种可能的输入值，且返回值大小只有4字节，故考虑提前存储所有可能的结果于数组crc\_table中，并在需要使用函数时查阅。生成crc\_table的代码如下

static unsigned int crc\_table[256];

unsigned char i = 0x00;

do {

crc\_table[i] = crc32\_update(i);

} while (++i);

使用该数组的crc32()函数代码如下

unsigned int crc32(unsigned char\* buf, int len)

{

unsigned int crc = 0x00000000L;

while (len--)

crc = (crc >> 8) ^ crc\_table[(crc ^ \*buf++) & 0xff];

return crc;

}

考虑到该函数广泛用于以太网链路层校验，crc\_table一般会存于链路层设备的只读存储器中，其余运算依靠硬件电路实现，从而实现快速计算。

库函数crc32()源代码如下

#define DO1(buf) crc = crc\_table[((int)crc ^ (\*buf++)) & 0xff] ^ (crc >> 8);

#define DO2(buf) DO1(buf); DO1(buf);

#define DO4(buf) DO2(buf); DO2(buf);

#define DO8(buf) DO4(buf); DO4(buf);

unsigned int crc32(unsigned char \*buf, int len)

{

unsigned int crc = 0xffffffffL;

while (len >= 8) {

DO8(buf);

len -= 8;

}

if (len) {

do {

DO1(buf);

} while (--len);

}

return crc;

}

该实现与我们提供的实现主要有两点不同。首先，有别于我们将crc初始化为0，库函数将crc初始化为0xffffffffL，这源于CRC32标准。其次，库函数利用一系列宏定义计算crc，一轮循环计算8次，减少了while循环的次数。这有利于提高程序的性能，具体原因参见《Computer Systems: A Programmer's Perspective》第5.8节"Loop Unrolling"（CSAPP, Sect. 5.8）。

### 在 x86 系列计算机上为相同帧计算 16 位 CRC16 校验和，所花费的 CPU 时间会比 CRC32 多一倍吗？

现代 x86 系列计算机使用64位处理器。分析4.2.1代码可知，显然不会多一倍。

### 为什么pppfcs32(fcs, cp, len)设计成需要三个参数？

参数fcs为当前CRC值。该种设计允许函数传入当前CRC值，结合新接收的数据更新CRC值。与实验“先缓存后校验”的思想不同，该设计可实现在数据接收过程中实时计算CRC，减少转发时延。

## 程序设计方面的问题

### 协议软件的跟踪功能有什么意义？

便于调试。

### 我的程序实现这样的功能了吗？

实现了。

### 实现get\_ms()

代码如下

#include <time.h>

unsigned int get\_ms(void)

{

return (unsigned int)(clock() \* 1000 / CLOCKS\_PER\_SEC);

}

### 参考给出的源代码，了解 printf 风格的参数数目不确定的函数的实现方式。

函数lprintf()源代码如下

#include <stdarg.h>

int lprintf(const char \*format,...)

{

int n;

va\_list arg\_ptr;

va\_start(arg\_ptr, format);

n = \_\_v\_lprintf(format, arg\_ptr);

va\_end(arg\_ptr);

return n;

}

实现该类函数需要满足以下三点要求

1. 包含 <stdarg.h> 头文件
2. 参数列表末尾用 ... 表示不定参数，如

int lprintf(const char \*format,...);

1. 采用如下实现逻辑
   1. 使用 va\_start 初始化参数列表指针。
   2. 将参数列表和格式字符串传递给实际处理函数（如 \_\_v\_lprintf）。
   3. 使用 va\_end 清理参数列表。

模仿源代码实现my\_printf()函数如下  
#include <stdio.h>

#include <stdarg.h>

int my\_printf(const char\* format, ...)

{

va\_list arg\_ptr;

va\_start(arg\_ptr, format); // 初始化 arg\_ptr 指向可变参数

int n = vprintf(format, arg\_ptr); // 调用实际处理函数

va\_end(arg\_ptr); // 清理 arg\_ptr

return n; // 返回输出的字符数

}

### 分析start\_timer()与start\_ack\_timer()的设计

#### 为什么start\_timer()等物理层发送完再计时？

在编写协议程序时，我们会非常自然地在调用send\_frame()函数发完帧后调用start\_timer()函数启动重传定时器。如果一切顺利，从该帧完全从物理层发出到收到ACK中间的时间是可以计算的（见3.3），而从调用start\_timer()函数到收到ACK中间的时间与此时物理层队列长度有关。若直接将start\_timer()设计为等物理层发送完再计时，则在设计重传定时器的时限时便无需考虑物理层队列长度，简化协议设计。

#### 为什么start\_ack\_timer()调用后立刻计时？

start\_ack\_timer()函数一般在需要发送ACK时调用，用于等待网络层数据报，超时则单发ACK。该时限与物理层无关，故调用后立刻计时。

#### 为什么start\_timer()按照新的时间计时？

为了避免不必要的重传，协议设计过程中会有需要重置重传定时器的情形（见5.4.3），故重调start\_timer()按照新的时间计时便于协议设计。

#### 为什么start\_ack\_timer()按照先前的时间计时？

我们希望所有ACK都能够尽早发送。若按照新的时间计时，可能延后ACK的发送时间。若按照先前的时间计时，并结合累计确认机制，可以保证ACK的稳定发送。该思想在教材已有体现，不再赘述。

## 软件测试方面的问题

### 设计多种测试方案的目的

验证所完成的程序能否在各种情况下都能够正确工作。

### 每种方案瞄准协议软件中可能出问题的环节

|  |  |
| --- | --- |
| 方案 | 瞄准环节 |
| -–utopia | 基本传输功能 |
| 无 | 差错控制 |
| --flood --utopia | 流量控制与可靠传输 |
| --flood | 大部分主要功能 |
| --flood --ber=1e-4 | 协议在杂乱信道下的健壮性 |

### 程序库的不足

程序库源代码可以考虑添加更多的注释，方便开发协议时更快地理解库函数的具体功能。

## 对等协议实体之间的流量控制

### 程序有没有解决两个站点的数据链路层对等实体之间的流量控制问题？

有。当链路层缓存已满，或物理层正忙时，程序会禁用网络层，以实现流量控制。相关代码如下

if (nbuffered < NR\_BUFS && phl\_ready)

enable\_network\_layer();

else

disable\_network\_layer();

## 与标准协议的对比

### 如果现实中有两个相距 5000 公里的站点要利用本协议通过卫星信道进行通信，还有哪些问题需要解决？

卫星信道具有高带宽，高延迟，高误码率的特点。实验模型参数是固定的，现实网络参数往往具有不确定性，需基于实时网络参数动态调整协议参数（协议参数计算方法见3.3）。

# 实验总结和心得体会

## 实际上机调试时间

40小时以上。

## 编程工具相关问题

无。

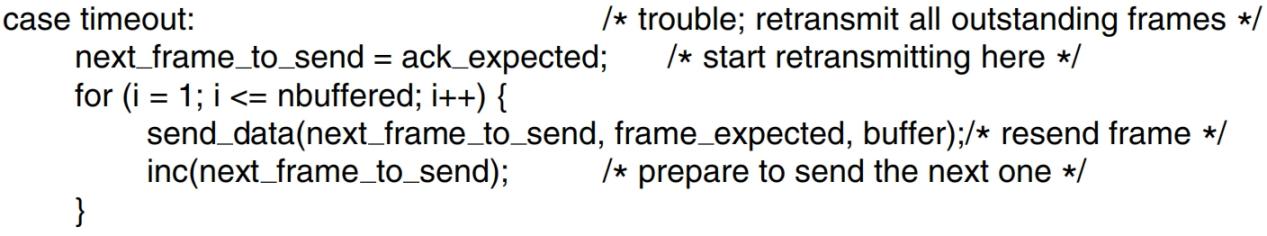
## 编程语言相关问题

无。

## 协议相关问题

### Go-Back-N协议超时重传相关问题

教材伪代码关于Go-Back-N协议超时重传部分如下



当计时器超时时，教材选择使用for循环一次将发送窗口内的所有帧全部重传。考虑如下情况，发送方发送，全部正确到达，接收方返回但丢失。根据3.3.3.1设计的定时器时限，会超时。根据教材设计，会一次全部重传。即使随后接收方返回的正确到达，这次不必要的重传也会发生。如果不是一次重传，而是分多次重传，随后接收方返回的的正确到达就能阻止后续的重传。据此思路能够得出如下伪代码

switch (event) {

/\* handle other event \*/

case FRAME\_RECEIVED:

/\* handle received frame \*/

while (f.ack is in send window) {

/\* handle ack \*/

if (frame\_nr != upper edge of send window + 1 && frame\_nr is not in send window)))

// frame\_nr is received but ack is lost, so stop resending

frame\_nr = ack\_expected;

}

case DATA\_TIMEOUT:

frame\_nr = ack\_expected; // go back n

for (int i = 0; i < nbuffered; i++)

stop\_timer((frame\_nr + i) % WINDOW\_SIZE); // don’t resend together

break;

/\* handle other event \*/

}

if (phl\_ready && frame\_nr != upper edge of send window + 1) {

send\_data\_frame(frame\_nr); // resend one by one

INC(frame\_nr);

}

### Go-Back-N协议请求重传相关问题

与教材一致，本次实验的Go-Back-N协议未引入请求重传机制，故未遇到相关问题。

### 选择重传协议超时重传相关问题

考虑与3.3.3.2中讨论的情况类似的情况，发送方发送，其中丢失，接收方返回，成功到达正确到达，由于累计确认机制，接收方不会返回。发送方收到后重传。现假设再次丢失，则根据3.3.3.2设计的定时器时限，首先会相继超时，发送方会顺序地重传，这四次重传都是不必要的。随后超时重传，若此次成功到达，接收方会返回，从而关闭的定时器，若再一次丢失，则又会有四次不必要的重传。事实上，都出错的概率只有，可以认为不可能发生。而连错两次的情况的概率为，是有可能发生的，实验中也出现过类似情形。因此，因连错两次而导致顺序重传的情况是应该避免的。我们选择在非发送窗口下界定时器超时时，仅重传超时帧，同时重置所有后继定时器，来避免重传多帧的情况。据此思路能够得出如下伪代码

case DATA\_TIMEOUT:

send\_data\_frame(arg);

if (arg != ack\_expected) {

int i = (arg + 1) % (MAX\_SEQ + 1);

while (i is in send window) {

start\_timer(i, DATA\_TIMER);

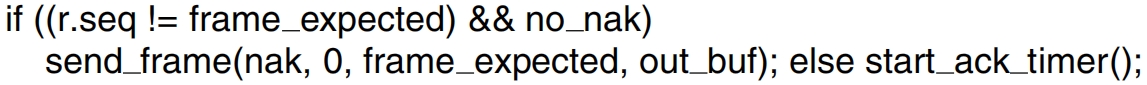
INC(i);

}

}

### 选择重传协议请求重传相关问题

教材中，当收到非期望帧时会返回期望帧的NAK，如图所示



考虑如下情况，发送方发送，正确到达，接收方返回但丢失，发送方重传正确到达，根据教材设计，由于已被收到，此时非期望帧，设此时期望帧序号为，接收方会返回期望帧的NAK，即。随后发送方先发送，正确到达，后收到，故重传，此时非期望帧，根据教材设计，接收方会再次返回期望帧的NAK，如此循环。每一轮循环都会引发一次不必要的重传。为了防止这一系列不必要的循环重传，我们仅在收到落在窗口内的非期望帧时才返回NAK。据此思路能够得出如下伪代码

if (f.seq is in recv window && f.seq != frame\_expected && no\_nak)

send\_nak\_frame();

else

start\_ack\_timer(ACK\_TIMER);

## 开发库相关问题

实验指导书中对函数 phl\_sq\_len()的描述为

“函数 phl\_sq\_len()的描述为返回当前物理层队列的长度。”

建议改为

“函数 phl\_sq\_len()的描述为返回当前物理层队列的长度，单位为字节。”

## 总结

经过本次本实验，我在许多方面有了很大提高。

### C语言方面

通过实现本协议，我增加了系统级编程经验，以及通过查看日志调试函数的能力。同时编码时严格遵循附录一的模块化设计原则，了解了工程化代码规范。

### 协议软件方面

经过本次本实验，我对滑动窗口协议有了更深入的理解。尤其是累计确认机制与捎带确认机制的结合。同时基于对协议的深入理解，对协议进行理论分析，计算协议参数（见3.3），并对教材的代码做出了优化（见5.4），提升了信道利用率。

### 理论学习方面

经过本次实验，我重新回顾了CRC校验相关知识，加深了对CRC校验的理解。同时我阅读了标准协议相关代码，了解了教材算法与工业级算法的区别（见4.2）。

# 源程序文件

## Go-Back-N 协议

#include <stdio.h>

#include <string.h>

#include "protocol.h"

#include "datalink.h"

#define DATA\_TIMER 1010

#define ACK\_TIMER 264

#define MAX\_SEQ 127

#define INC(s) do { s = (s + 1) % (MAX\_SEQ + 1); } while(0)

typedef enum { false, true } boolean;

struct FRAME {

unsigned char kind; /\* FRAME\_DATA \*/

unsigned char ack;

unsigned char seq;

unsigned char data[PKT\_LEN];

unsigned int padding;

};

// as a sender

static unsigned char frame\_nr = 0, buffer[MAX\_SEQ + 1][PKT\_LEN], nbuffered = 0;

static unsigned char ack\_expected = 0;

static boolean phl\_ready = false;

// as a receiver

static unsigned char frame\_expected = 0;

static int between(unsigned char a, unsigned char b, unsigned char c)

{

return (((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a)));

}

static void put\_frame(unsigned char\* frame, int len)

{

\*(unsigned int\*)(frame + len) = crc32(frame, len);

send\_frame(frame, len + 4);

phl\_ready = false;

}

static void framing(unsigned char kind)

{

struct FRAME s;

s.kind = kind;

s.ack = (frame\_expected + MAX\_SEQ) % (MAX\_SEQ + 1);

if (kind == FRAME\_DATA) {

s.seq = frame\_nr;

memcpy(s.data, buffer[frame\_nr % (MAX\_SEQ + 1)], PKT\_LEN);

dbg\_frame("Send DATA %3d %3d, ID %d, Window %3d, PHL Queue %d\n",

s.seq, s.ack, \*(short\*)s.data,

(s.seq + MAX\_SEQ + 1 - ack\_expected) % (MAX\_SEQ + 1), phl\_sq\_len());

put\_frame((unsigned char\*)&s, 3 + PKT\_LEN);

start\_timer(frame\_nr, DATA\_TIMER);

INC(frame\_nr);

}

else {

dbg\_frame("Send ACK %3d\n", s.ack);

put\_frame((unsigned char\*)&s, 2);

}

stop\_ack\_timer();

}

int main(int argc, char\*\* argv)

{

int event, arg;

struct FRAME f;

int len = 0;

protocol\_init(argc, argv);

lprintf("Designed by Chen ZiRong, build: " \_\_DATE\_\_" "\_\_TIME\_\_"\n");

lprintf("Protocol Options: Piggybacking (Enabled), NAK (Disabled)\n");

disable\_network\_layer();

for (;;) {

event = wait\_for\_event(&arg);

switch (event) {

case NETWORK\_LAYER\_READY:

get\_packet(buffer[frame\_nr]);

nbuffered++;

framing(FRAME\_DATA);

break;

case PHYSICAL\_LAYER\_READY:

phl\_ready = true;

break;

case FRAME\_RECEIVED:

len = recv\_frame((unsigned char\*)&f, sizeof f);

if (len < 5 || crc32((unsigned char\*)&f, len) != 0) {

dbg\_event("\*\*\*\* Receiver Error, Bad CRC Checksum\n");

break;

}

if (f.kind == FRAME\_ACK)

dbg\_frame("Recv ACK %d\n", f.ack);

if (f.kind == FRAME\_DATA) {

dbg\_frame("Recv DATA %d %d, ID %d\n", f.seq, f.ack, \*(short\*)f.data);

if (f.seq == frame\_expected) {

put\_packet(f.data, len - 7);

INC(frame\_expected);

}

start\_ack\_timer(ACK\_TIMER);

}

while (between(ack\_expected, f.ack,

(ack\_expected + nbuffered) % (MAX\_SEQ + 1))) {

stop\_timer(ack\_expected);

nbuffered--;

INC(ack\_expected); // slide window

if (frame\_nr != (ack\_expected + nbuffered) % (MAX\_SEQ + 1)

&& !between(ack\_expected, frame\_nr, (ack\_expected + nbuffered) % (MAX\_SEQ + 1))) // frame\_nr is received but ack is lost, so stop resending

frame\_nr = ack\_expected;

}

break;

case DATA\_TIMEOUT:

dbg\_event("---- DATA %d timeout\n", arg);

frame\_nr = ack\_expected; // go back n

for (int i = 0; i < nbuffered; i++)

stop\_timer((frame\_nr + i) % (MAX\_SEQ + 1));

break;

case ACK\_TIMEOUT:

framing(FRAME\_ACK);

break;

}

if (phl\_ready && frame\_nr != (ack\_expected + nbuffered) % (MAX\_SEQ + 1))

framing(FRAME\_DATA); // resend

if (nbuffered < MAX\_SEQ && phl\_ready)

enable\_network\_layer();

else

disable\_network\_layer();

}

}

## 选择重传协议

#include <stdio.h>

#include <string.h>

#include "protocol.h"

#include "datalink.h"

#define DATA\_TIMER 1871

#define ACK\_TIMER 264

#define MAX\_SEQ 127

#define NR\_BUFS ((MAX\_SEQ + 1) / 2)

#define INC(s) do { s = (s + 1) % (MAX\_SEQ + 1); } while(0)

typedef enum { false, true } boolean;

struct FRAME {

unsigned char kind; /\* FRAME\_DATA \*/

unsigned char ack;

unsigned char seq;

unsigned char data[PKT\_LEN];

unsigned int padding;

};

// as a sender

static unsigned char out\_buf[NR\_BUFS][PKT\_LEN], nbuffered = 0;

static unsigned char ack\_expected = 0;

static boolean phl\_ready = false;

// as a receiver

static unsigned char in\_buf[NR\_BUFS][PKT\_LEN];

static unsigned char arrived[NR\_BUFS] = { 0 }, frame\_expected = 0;

static boolean no\_nak = true;

static boolean between(unsigned char a, unsigned char b, unsigned char c)

{

return (((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a)));

}

static void put\_frame(unsigned char\* frame, int len)

{

\*(unsigned int\*)(frame + len) = crc32(frame, len);

send\_frame(frame, len + 4);

phl\_ready = false;

}

static void framing(unsigned char kind, unsigned char arg)

{

struct FRAME s;

s.kind = kind;

s.ack = (frame\_expected + MAX\_SEQ) % (MAX\_SEQ + 1);

if (kind == FRAME\_DATA) {

s.seq = arg;

memcpy(s.data, out\_buf[arg % NR\_BUFS], PKT\_LEN);

dbg\_frame("Send DATA %3d %3d, ID %d, Window %3d, PHL Queue %d\n",

s.seq, s.ack, \*(short\*)s.data,

(s.seq + MAX\_SEQ + 1 - ack\_expected) % NR\_BUFS, phl\_sq\_len());

put\_frame((unsigned char\*)&s, 3 + PKT\_LEN);

start\_timer(arg, DATA\_TIMER);

}

else {

if (kind == FRAME\_ACK) {

dbg\_frame("Send ACK %3d\n", s.ack);

}

else { // NAK

dbg\_frame("Send NAK %3d\n", s.ack);

no\_nak = false;

}

put\_frame((unsigned char\*)&s, 2);

}

stop\_ack\_timer();

}

int main(int argc, char\*\* argv)

{

int event, arg, nerror = 0, nresend = 0;

struct FRAME f;

int len = 0;

protocol\_init(argc, argv);

lprintf("Designed by Chen Zirong, build: " \_\_DATE\_\_" "\_\_TIME\_\_"\n");

disable\_network\_layer();

for (;;) {

event = wait\_for\_event(&arg);

switch (event) {

case NETWORK\_LAYER\_READY:

get\_packet(out\_buf[(ack\_expected + nbuffered) % NR\_BUFS]);

framing(FRAME\_DATA, (ack\_expected + nbuffered) % (MAX\_SEQ + 1));

nbuffered++;

break;

case PHYSICAL\_LAYER\_READY:

phl\_ready = true;

break;

case FRAME\_RECEIVED:

len = recv\_frame((unsigned char\*)&f, sizeof f);

if (len < 5 || crc32((unsigned char\*)&f, len) != 0) {

dbg\_event("Receiver Error %d: Bad CRC Checksum\n", ++nerror);

if (len > 6 && no\_nak)

framing(FRAME\_NAK, 0);

break;

}

if (f.kind == FRAME\_DATA) {

dbg\_frame("Recv DATA %3d %3d, ID %d, Window %3d %3d\n",

f.seq, f.ack, \*(short\*)f.data,

(f.seq - frame\_expected + MAX\_SEQ + 1) % (MAX\_SEQ + 1),

(frame\_expected + NR\_BUFS) % (MAX\_SEQ + 1));

if (between(frame\_expected, f.seq,

(frame\_expected + NR\_BUFS) % (MAX\_SEQ + 1))

&& f.seq != frame\_expected && no\_nak)

framing(FRAME\_NAK, 0);

else

start\_ack\_timer(ACK\_TIMER);

if (between(frame\_expected, f.seq,

(frame\_expected + NR\_BUFS) % (MAX\_SEQ + 1))

&& (arrived[f.seq % NR\_BUFS] == false)) {

arrived[f.seq % NR\_BUFS] = 1;

memcpy(in\_buf[f.seq % NR\_BUFS], f.data, PKT\_LEN);

while (arrived[frame\_expected % NR\_BUFS]) {

put\_packet(in\_buf[frame\_expected % NR\_BUFS], len - 7);

no\_nak = true;

arrived[frame\_expected % NR\_BUFS] = false;

INC(frame\_expected);

start\_ack\_timer(ACK\_TIMER);

}

}

}

if (f.kind == FRAME\_ACK) {

dbg\_frame("Recv ACK %3d\n", f.ack);

}

if (f.kind == FRAME\_NAK) {

dbg\_frame("Recv NAK %3d\n", f.ack);

if (between(ack\_expected, (f.ack + 1) % (MAX\_SEQ + 1),

(ack\_expected + nbuffered) % (MAX\_SEQ + 1)))

framing(FRAME\_DATA, (f.ack + 1) % (MAX\_SEQ + 1));

}

while (between(ack\_expected, f.ack,

(ack\_expected + nbuffered) % (MAX\_SEQ + 1))) {

nbuffered--;

stop\_timer(ack\_expected);

INC(ack\_expected);

}

break;

case DATA\_TIMEOUT:

dbg\_event("Timeout DATA %3d, retransmit %d\n", arg, ++nresend);

framing(FRAME\_DATA, arg);

if (arg != ack\_expected)

for (int i = (arg + 1) % (MAX\_SEQ + 1); between(ack\_expected, i,

(ack\_expected + nbuffered) % (MAX\_SEQ + 1)); ) {

start\_timer(i, DATA\_TIMER);

INC(i);

}

break;

case ACK\_TIMEOUT:

framing(FRAME\_ACK, 0);

break;

}

if (nbuffered < NR\_BUFS && phl\_ready)

enable\_network\_layer();

else

disable\_network\_layer();

}

}

# 参考文献

1. Ramabadran, T. V., & Gaitonde, S. S. (1988). A Tutorial on CRC Computations. IEEE Micro.
2. Koopman, P., & Chakravarty, T. (2004). Cyclic Redundancy Code (CRC) Polynomial Selection for Embedded Networks. International Conference on Dependable Systems and Networks (DSN).