## 实验内容和实验环境描述

### 实验任务

设计数据链路层滑动窗口协议并编程实现

### 实验内容

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 10-5，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。在程序能够稳定运行并成功实现第一个目标之后，运行程序并检查在信道没有误码和存在误码两种情况下的信道利用率。

为实现第二个目标，提高滑动窗口协议信道利用率，需要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。这些参数的设计，需要充分理解滑动窗口协议的工作原理并利用所学的理论知识，经过认真的推算，计算出最优取值，并通过程序的运行进行验证。

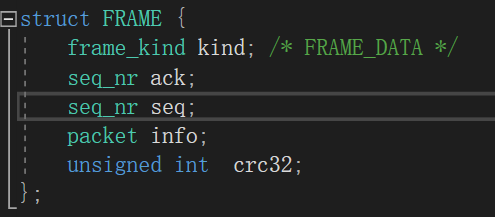
### 实验环境

Windows\_10\_PC机，Microsoft Visual Studio 2019集成化开发环境。

## 软件设计

### 数据结构

1. GOBACKN协议
   1. 数据定义
      1. 帧



struct FRAME {

frame\_kind kind; //帧的种类，分为 Ack和 Data 两种

seq\_nr ack; //上一次成功接收的帧的序号

seq\_nr seq; //本帧的序号

packet info; //数据

unsigned int crc32; //填充字段

};

* + - 1. 数据帧

+=========+========+========+===============+========+

| KIND(1) | ACK(1) | SEQ(1) | DATA(256) | CRC(4) |

+=========+========+========+===============+========+

* + - 1. ACK帧

+=========+========+========+

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

+=========+========+========+

* 1. 定义

NETWORK\_LAYER\_READY：网络层有待发送的分组。此事件发生后才可以调用 get\_packet()得到网络层待发送的下一个分组。

PHYSICAL\_LAYER\_READY：物理层发送队列的长度低于 50 字节。

FRAME\_RECEIVED：物理层收到了一整帧。

DATA\_TIMEOUT：定时器超时，参数 arg 中返回发生超时的定时器的编号。

ACK\_TIMEOUT：所设置的搭载 ACK 定时器超时。

#define DATA\_TIMER 2000 //重传定时器时限

#define MAX\_SEQ 7 //滑动窗口的大小

#define ACK\_TIMER 1100 //ACK 搭载定时器

#define inc(k) if(k < MAX\_SEQ) k = k + 1; else k = 0

typedef unsigned char seq\_nr; //帧序号

typedef unsigned char frame\_kind; //帧的种类

typedef unsigned char packet[PKT\_LEN]; //256字节的数据

static int phl\_ready = 0;//物理层是否准备完毕，1表示物理层准备完毕

* 1. 主函数变量

seq\_nr next\_frame\_to\_send = 0;://将要发送的帧序号，发送方窗口的上界

seq\_nr nbuffered = 0; //当前已发送数据帧的个数

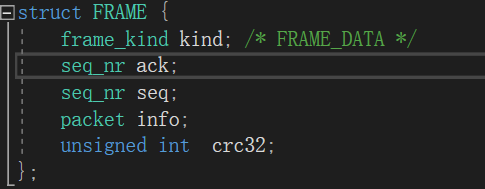
seq\_nr ack\_expected = 0; //期望收到的 ack 帧序号，发送方窗口的下界

seq\_nr frame\_expected = 0; //期望收到的帧序号，接收方窗口的下界

packet buffer[MAX\_SEQ + 1]; //数据

int len = 0; //接受到的帧的字节数

1. 选择重传协议
   1. 数据定义
      1. 帧



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) |

+=========+========+========+

* 1. 定义

#define MAX\_SEQ 43 //帧序号范围

#define DATA\_TIMER 6000

#define ACK\_TIMER 5100

#define NR\_BUFS ((MAX\_SEQ+1)/2) //发送窗口和接收窗口的大小

#define inc(k) if(k < MAX\_SEQ) k = k + 1; else k = 0

typedef unsigned char seq\_nr;

typedef unsigned char frame\_kind;

typedef unsigned char packet[PKT\_LEN];

int no\_nak = 1; //NAK是否发送

static int phl\_ready = 0;

* 1. 局部数据

seq\_nr ack\_expected = 0; //发送窗口下限

seq\_nr next\_frame\_to\_send = 0; //发送窗口上限

seq\_nr frame\_expected = 0; //接收窗口下限

seq\_nr too\_far = NR\_BUFS; //接收窗口上限

struct FRAME f; //数据帧

packet out\_buf[NR\_BUFS]; //发送窗口（发送缓存）

packet in\_buf[NR\_BUFS]; //接收窗口（接收缓存）

int arrived[NR\_BUFS]; //接收窗口接收情况

for (int i = 0; i < NR\_BUFS; i++) arrived[i] = 0;

seq\_nr nbuffered = 0; //已发送等待确认的帧数

int event, arg;

int len = 0; //接收的数据帧的长度

### 模板结构

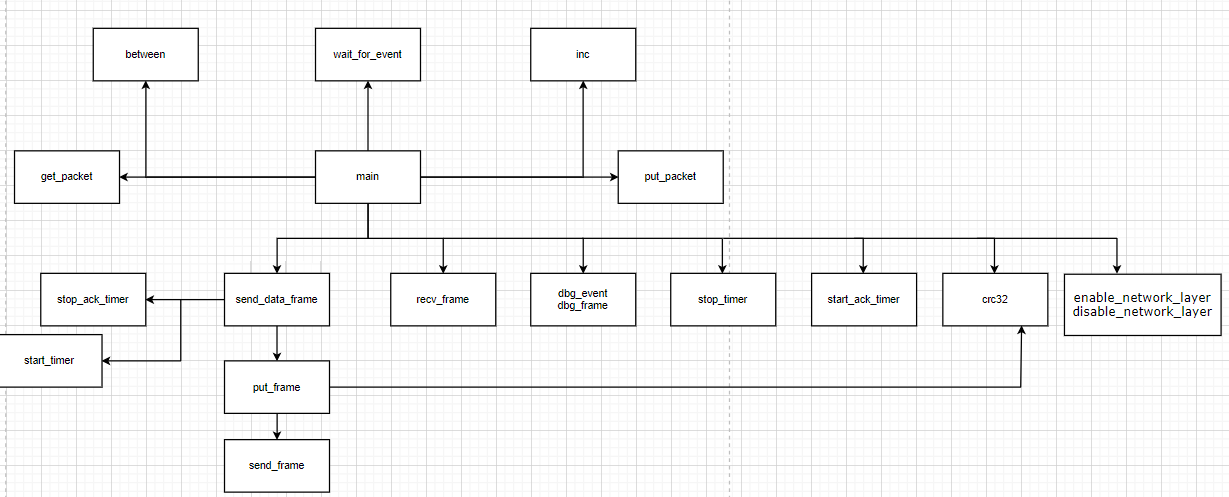


图1.1 GOBACKN协议和选择重传协议中函数调用关系

调用函数简介：

Disable\_network\_layer()

：无参数，数据链路层在缓冲区满等条件下无法发送分组时通过通知网络层

enable\_network\_layer()

：无参数，数据链路层在能够承接新的发送任务时允许网络层发送数据分组

put\_packet()

：供两个参数：存放收到分组的缓冲区首地址和分组长度。

：数据链路层接受到数据帧经过crc32校验，表示正确无误之后调用该函数上传给网络层

get\_packet()

：一个参数：存放分组的缓冲区首地址

：得到网络层待发送的下一个分组

wait\_for\_event()

：无参数

：导致进程等待，直到一个“事件”发生

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

：返回长度为len\*8位的字符串\*buf的crc32校验码

void start\_timer(unsigned int nr, unsigned int ms);

：用于启动一个定时器。两个参数分别为计时器的编号和超时时间值。计时器的编号只允许在 0~63之间，超时时间间隔的单位为毫秒。超时发生时，产生 DATA\_TIMEOUT 事件，并给出超时的定时器编号。在定时器未超时之前直接对同一个编号的定时器执行

start\_timer()调用，将按照新的时间设置产生超时事件。

void stop\_timer(unsigned int nr);

：中止一个定时器。

extern void dbg\_event(char \*fmt, ...);

extern void dbg\_frame(char \*fmt, ...);

extern void dbg\_warning(char \*fmt, ...);

：调试相关函数

#define inc(k) if(k < MAX\_SEQ) k = k + 1; else k = 0

**作用**：使一个字节在0~MAX\_SEQ的范围内循环自增。

static unsigned char between(seq\_nr a, seq\_nr b, seq\_nr c)

**作用**：判断当前帧是否落在发送/接收窗口内。

**参数**：a,b,c，均为字节类型，其中两个分别为窗口的上、下界，一个为帧的编号。其中，发送窗口的上界和下界分别为next\_to\_send和ack\_expected，接收窗口的下界为frame\_expected

static void put\_frame(seq\_nr \*frame, int len)

**作用**：为一个帧做CRC校验，填充至帧的尾部并将其递交给网络层发送。

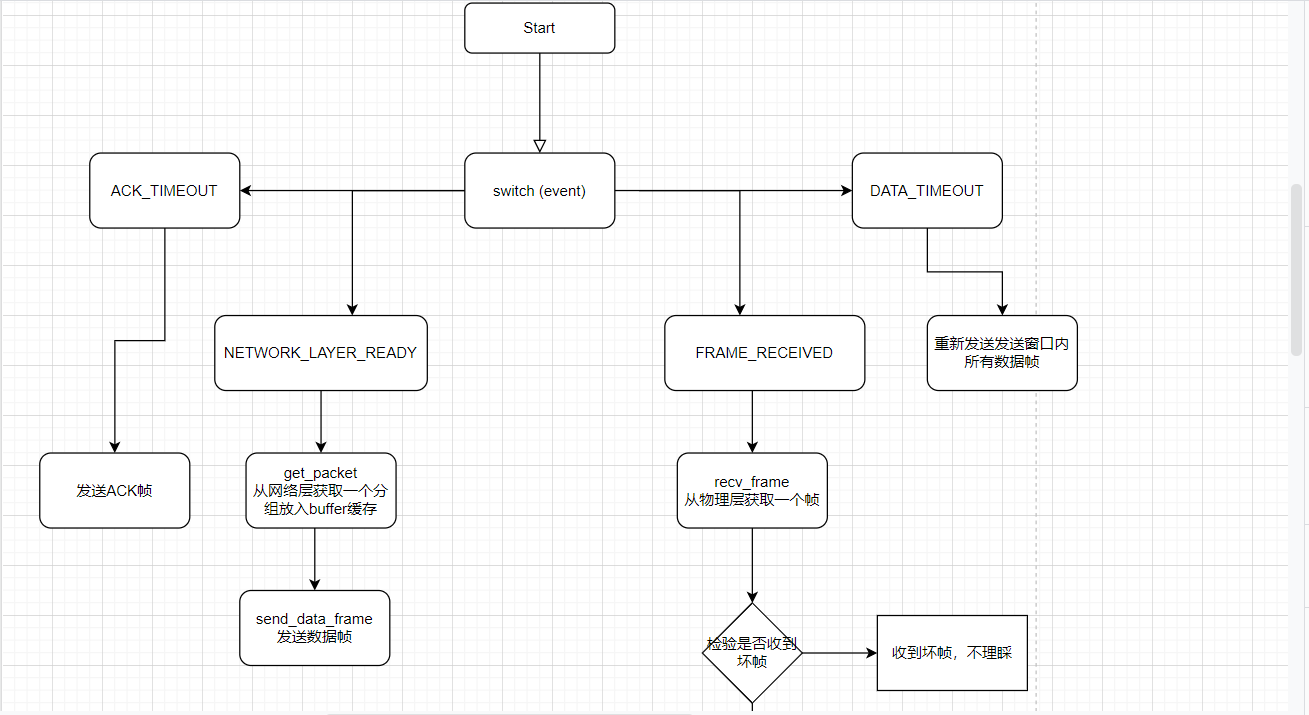
**参数**：frame，字节数组，由除crc32域之外的帧内容转换而来；len，整型，为帧的当前长度。

static void send\_data\_frame(frame\_kind fk, seq\_nr frame\_nr, seq\_nr frame\_expected, packet buffer[])

**作用**：构造一个帧，并将其发送。

**参数**：fk，字节类型，为帧的种类；frame\_nr，字节类型，为帧的编号；frame\_expected，字节类型，为希望收到的帧的编号；buffer[]，二维字节数组，传输帧的内容。

### 算法流程



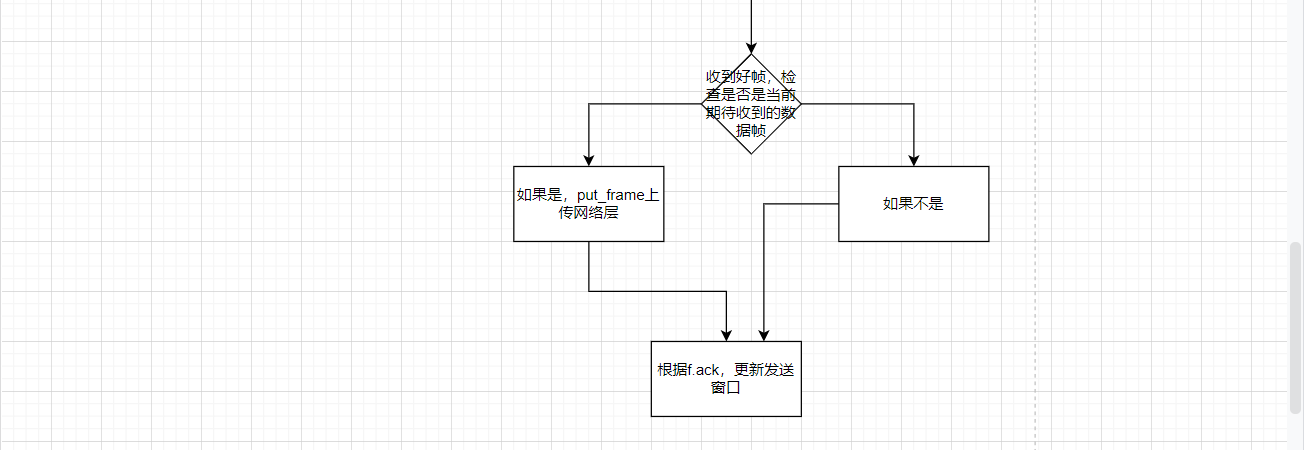


图1.2 GOBACKN算法流程

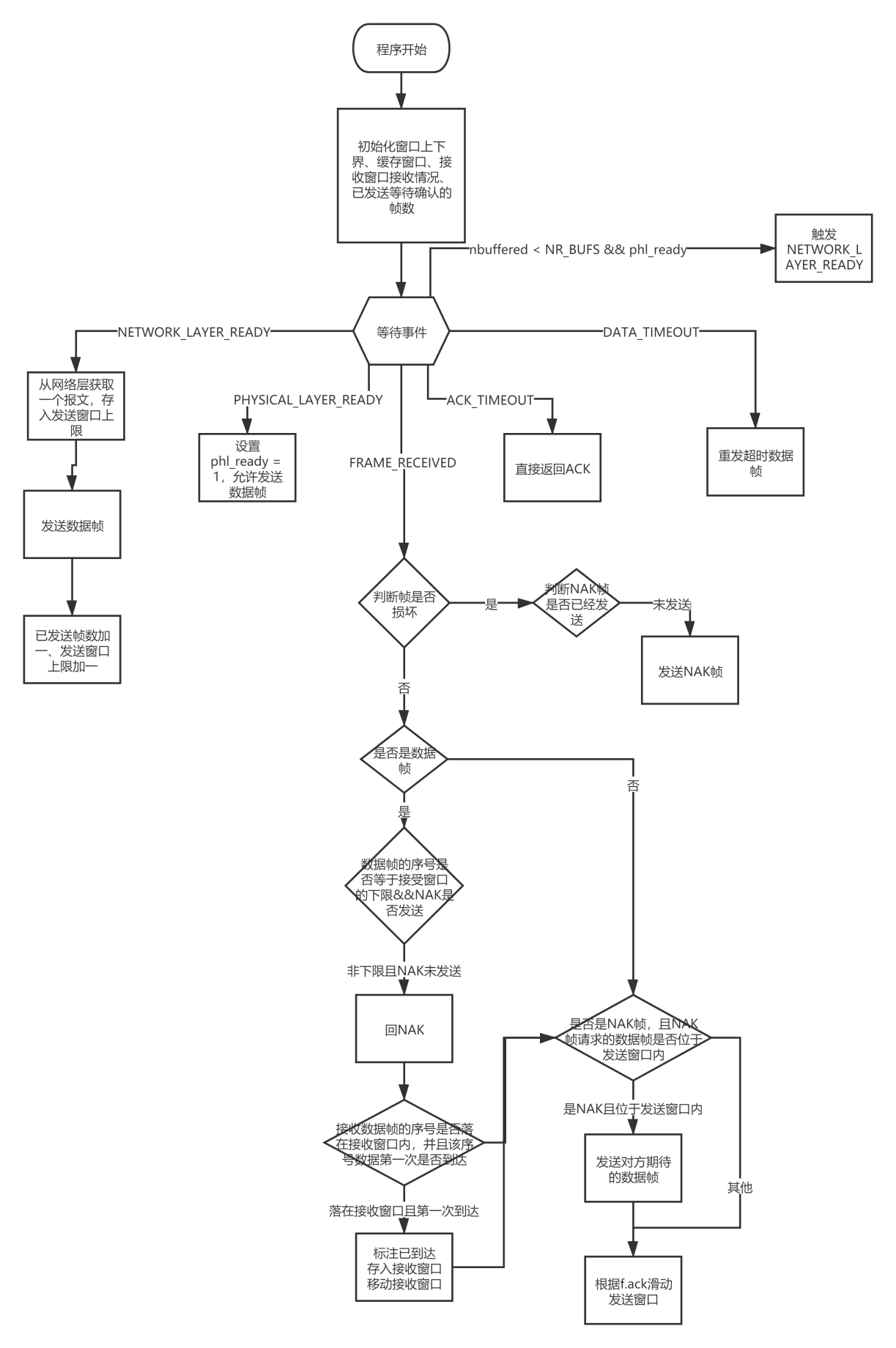


图1.3 选择重传协议算法流程

## 实验结果分析

### 功能描述

分别使用GOBACKN协议和选择重传协议实现有噪音信道环境下的无差错传输，计算调整参各数，并尽可能地充分利用传输信道的带宽。

### 程序健壮性

经过反复调试，程序健壮性逐渐加强。

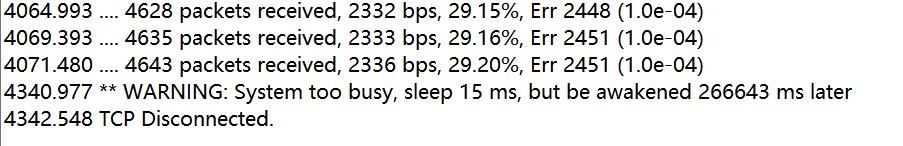


图1.4 GOBACKN协议工作记录截图

实际运行中，GOBACKN协议工作了72分钟，电脑休眠导致程序中断。

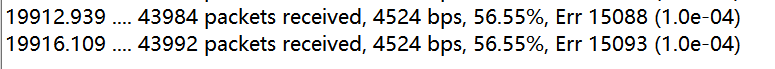


图1.5 GOBACKN协议工作记录截图

实际运行中，选择重传协议工作了336分钟（五小时半），手动关闭。

实践证明，GOBACKN程序和选择重传协议均能稳定运行较长时间。

### 协议参数选取

#### GOBACKN协议参数的选取

##### 数据帧计时器超时时长DATA\_TIMER的选取

先选取MAX\_SEQ。然后由

取1850ms

##### 发送窗口大小MAX\_SEQ的选取

首先MAX\_SEQ = 。分析可知，MAX\_SE并非越大越好，因为在GOBACKN协议中，错误帧的出现会导致DATA计时器的超时，发送窗口需要全部重发，如果发送窗口设置越大，重发的数据越多，这会降低传输速率。所以，选取合适的发送窗口大小必须考量信道误码率。

根据假设，比特误码率为1e-4时，一帧成功传输的可能为0.81。

所以每发送一帧有19%的概率，需要重发MAX\_SEQ个数据帧。所以选择相对较小的MAX\_SEQ有利于提高传输速率。

根据课上所学知识，当

所以 又因为

所以取n = 3 MAX\_SEQ 取 7

##### ACK\_TIMER的选取

先选取MAX\_SEQ = 7，DATA\_TIMER选取应该使得连续发送7帧结束之前，第一帧的应答帧能够回来。并且尽量让ACK\_TIMER更长，这样子更多的ACK能被捎带返回，提高信道利用率。

所以

取770ms

#### 选择重传协议参数的选取

##### 数据帧计时器超时时长DATA\_TIMER的选取

先选取NR\_BUFS = 32，然后根据

选取

##### 发送窗口和接收窗口大小NR\_BUFS的选取

选择重传协议的缓存窗口理论上是越大越好，NR\_BUFS = (MAX\_SEQ+1)/2，所以MAX\_SEQ越大越好。因为typedef unsigned char seq\_nr； seq\_nr seq; 帧序号是单字节无符号整数，所以MAX\_SEQ最大等于 ，此时NR\_BUFS = (MAX\_SEQ+1)/2=64。但是start\_timer()函数序号范围0~63，所以MAX\_SEQ最大也只能取63，此时NR\_BUFS = 32

##### ACK\_TIMER

7350ms > 4000ms 这时即使在IDLE期间大约每 4 秒才发送一帧，也能捎带回去

### 理论分析

#### GOBACKN协议在无差错信道上的最大信道利用率

--flood : flood traffic

因为无差错，所以数据帧都会被正确接收；-flood条件下，所有的ACK都会被捎带回去

此时信道最大利用率=

--ibib : set station B layer 3 sender mode as IDLE-BUSY-IDLE-BUSY-...

BUSY时，信道最大利用率=

IDLE时，四秒发一帧，信道传输速率=，信道利用率=。所以整段时间内，信道最大利用率=

#### GOBACKN协议在有差错信道上的最大信道利用率

--ber=1e-4 --flood : flood traffic

比特误码率为1e-4时，一帧成功传输的可能为0.81。

所有的ACK都会立即被捎带回去

一帧传输失败则回退7帧

对方返回的捎带ACK的数据帧同样可能出错，但如果考虑这种情况则过于复杂，假定超时重传的数据帧的回馈ACK 帧可以 100%正确传输

则一帧平均需要占用帧

所以信道利用率=256/263/2.45=39.73%

由于我们忽略了超时重传的数据帧的回馈ACK 帧也可能出错，真实信道利用率应该低于39.73%

--ber=1e-5 --ibib

Flood时一帧平均需要占用1.1279帧

信道利用率=256/263/1.1279=86.30%

IDLE时，四秒发一帧，我们假设四秒内这一帧经过多次发送后能被正确接受，则信道利用率=

整段时间内信道利用率=（86.30%+6.4%）/2=46.35%

由于我们忽略了超时重传的数据帧的回馈ACK 帧也可能出错，真实信道利用率应该低于46.35%

#### 选择重传协议在无差错信道上的最大信道利用率

和GOBACKN协议分析过程基本类似

#### 选择重传协议在有差错信道上的最大信道利用率

--flood -b1e-4

一帧平均需要传帧

信道利用率=256/263/1.334=72.97%

--ber=1e-5 --ibib

1.02帧

信道利用率=256/263/1.02=95.42%

IDLE时，四秒发一帧，我们假设四秒内这一帧经过多次发送后能被正确接受，则信道利用率=

整段时间内信道利用率=（95.42%+6.4%）/2=50.91%

### 实验结果分析



图1.5 MAX\_SEQ = 7 1850ms ACK\_TIMER = 770ms的GOBACKN协议信息传输速率

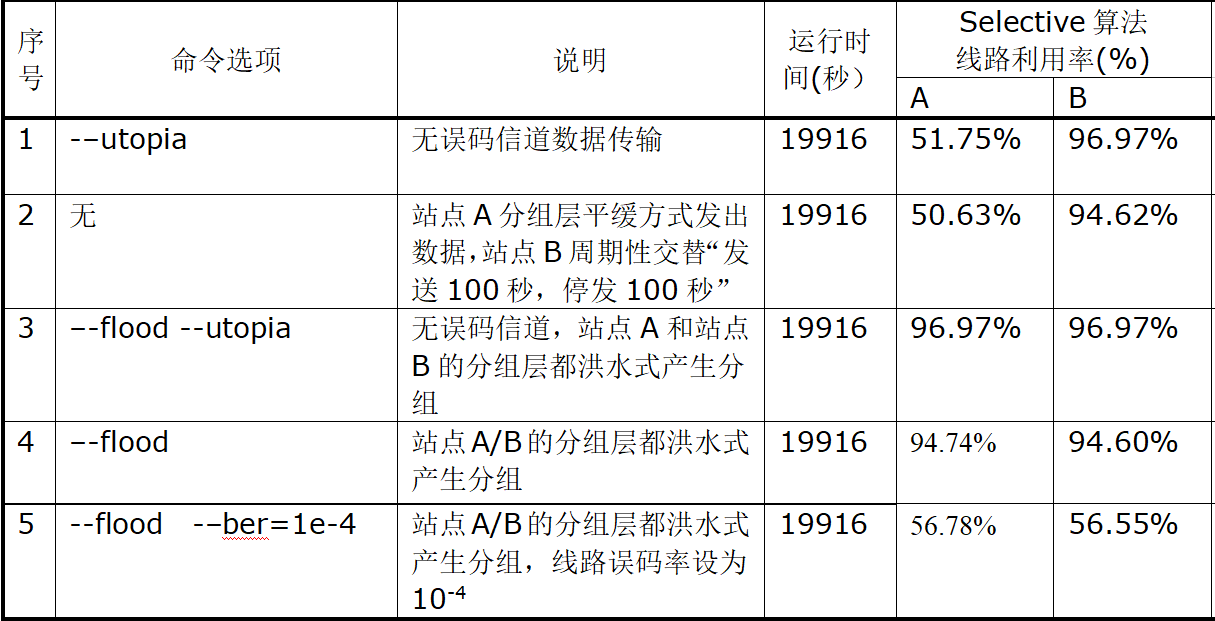


图1.5 MAX\_SEQ = 63 8420ms ACK\_TIMER = 7350ms的选择重传协议信息传输速率

和理论数据对比，除了假设捎带ACK的数据帧回传100%正确导致理论信息传输速率大幅上涨之外，理论值和真实值十分接近，差距几乎在1%以内。

在理论计算中，捎带ACK的数据帧回传100%正确这一假设在误码率较高的情况中，会引起理论值出现较大的误差（例如在GOBACKN -b1e-4的情况下），但是简化状态和准确率不可兼得。在看到实验指导PDF中提醒“捎带ACK的数据帧回传100%正确”这一假设前，我曾尝试计算没有这一假设下，GOBACKN -b1e-4的信息传输速率，思路是可以分为三种状况讨论“发对收对”、“发对收错”、“发错无回”，则情况异常复杂，只好放弃。

在理论值推导过程中，可以十分清晰的知道该协议的“瓶颈”在哪里，例如在GOBACKN协议--ber=1e-4的传输信道中，一帧平均需要占用帧才能被正确接收，所以，如果我们使用数据很短不易出错的NAK帧告知发送方数据出错，则就可以显著提高传输速率。

### 存在的问题

和同学讨论过程中，我发现许多同学在选择重传协议参数选定中，NR\_BUFS多选取16，并且传输速率比我NR\_BUFS选取32来的高。这里面的原因，我没有探究清楚。我猜测可能跟物理层缓存有关，NR\_BUFS选取过大，可能导致发送端数据发送过快，导致物理层缓存溢出，降低了传输速率。

## 研究和探索的问题

### CRC校验能力

CRC校验码的检错能力很强，它除了能检查出离散错外，还能检查出突发错，CRC校验码具有以下检错能力：CRC校验码能检查出全部单个错；CRC校验码能检查出全部离散的二位错；CRC校验码能检查出全部奇数个错；CRC校验码能检查出全部长度小于或等于K位的突发错；CRC校验码能以[1-（1/2）K-1]的概率检查出长度为（K+1）位的突发错。

至少要发生34位的错误CRC太检测不出来，假设一位发生错误的概率是1e-4，则263字节中发生34位错误的概率是

也就是说平均发了个数据帧才会出现crc32检测不出的差错

假设信道传输输入为8000bps，每秒帧，也就是需要秒才会出现一帧异常。年才能出现错误。显然概率极低。

## 实验总结和心得体会

### 实验总结

1. 完成本次实验的实际上机调试时间是多少？

花了一两小时读实验指导PDF，花一两小时该课上PPT的代码（主要在改接口），花十五小时的时间调试，单人完成了GOBACKN和选择重传协议

(2) 编程工具方面遇到了哪些问题？包括 Windows 环境和 VC 软件的安装问题。

这方面并没有遇到太大问题，遇到的小问题借助搜索引擎也都能很快地解决。

(3) 编程语言方面遇到了哪些问题？包括 C 语言使用和对 C 语言操控能力上的问题

没有在编程语言方面遇到问题。不过起初图省事，我直接在datalink.c的源代码删改，没有另写一个datalink.c文件，导致代码格式杂糅，造成代码阅读和理解上的障碍，调试难度和耗费的时间大大增加。

(4) 协议方面遇到了哪些问题？包括协议机制的设计错误，发现协议死锁，或者不能正确工作，协议参数的调整等问题。

在写GOBACKN协议时，因为搞混了ACK\_TIMER和ACK\_TIMEOUT，而把ACK\_TIMEOUT传给start\_ack\_timer()函数，因为这两个宏定义都是unsigned char 类型，编译时不会报错，导致把ACK计时器设为2ms，程序异常。

在写选择重传协议时，不知道帧序号什么时候要%NR\_BUFS和什么时候不用，导致程序运行短时间之后死锁。

(6) 总结本次实验，你在 C 语言方面，协议软件方面，理论学习方面，软件工程方面等哪些方面上有所提高？

在协议软件方面，我了解一个抽象的协议是如何用具体的代码实现的。

C语言方面和软件工程方面，由于实验过程中，我透明了底层实现，只是了解了一下其他文件提高的接口然后编写了datalink.c文件，并没有对整个程序特别了解，提高很小，这也是我在本次实验的不足之处。

### 心得体会

1. 修改别人程序时，应遵循别人程序的代码风格，避免风格杂糅
2. 独学无友则孤落而寡闻。在和同学讨论过程中，我发现别的同学总能发现我没发现的问题，例如本次实验中线路利用率的计算问题。也发现别人用python写了一个脚本自动打开是个命令行分别执行不同参数的datalink.exe文件，而不用手动打开命令行执行，极大方便了调试，也让我知道了掌握一门脚本语言，编写脚本替代自己完成机械枯燥的工作可以显著提高学习和工作效率。和同学们讨论中，我也学习了如何在word文档中插入公式，也提升了自己编写文档的能力。
3. 应该有勇气取学习自己不熟悉的知识和技术，这样子能学到更多知识和技术。例如，在这次实验中，还有许多技术是我十分陌生的，例如，软件工程方面：VS是如何整合多个文件的？CMAKE又是如何整合文件的？C语言方面：ACK\_TIMOUT等事件是如何发生以及被wait\_event()函数接受的？辅助工具：脚本语言的学习。协议方面：底层代码的实现；socket编程。如果能借这次实验的机会掌握上面所提及的知识和技术，自身能力的提升也就不仅仅只是对协议更加深入的理解而已。

//GOBACKN

#include <stdio.h>

#include <string.h>

#include "protocol.h"

#include "datalink.h"

#define DATA\_TIMER 1850

#define MAX\_SEQ 7

#define ACK\_TIMER 770

#define inc(k) if(k < MAX\_SEQ) k = k + 1; else k = 0

typedef unsigned char seq\_nr;

typedef unsigned char frame\_kind;

typedef unsigned char packet[PKT\_LEN];

static unsigned char between(seq\_nr a, seq\_nr b, seq\_nr c)

{

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

}

struct FRAME {

frame\_kind kind; /\* FRAME\_DATA \*/

seq\_nr ack;

seq\_nr seq;

packet info;

unsigned int crc32;

};

static int phl\_ready = 0;

static void put\_frame(seq\_nr \*frame, int len)

{

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

send\_frame(frame, len + 4);

phl\_ready = 0;

}

static void send\_data\_frame(frame\_kind fk, seq\_nr frame\_nr, seq\_nr frame\_expected, packet buffer[])

{

struct FRAME s;

s.kind = fk;

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

stop\_ack\_timer();

if (fk == FRAME\_DATA) {

s.seq = frame\_nr;

memcpy(s.info, buffer[frame\_nr], PKT\_LEN);

dbg\_frame("Send DATA %d %d, ID %d\n", s.seq, s.ack, \*(short \*)s.info);

put\_frame((seq\_nr \*)&s, 3 + PKT\_LEN); //加入四字节crc校验

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

}

else if (fk == FRAME\_ACK) {

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

put\_frame((unsigned char\*)&s, 2); //在s.ack后面插入crc校验

}

}

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

{

seq\_nr next\_frame\_to\_send = 0;

seq\_nr nbuffered = 0;

seq\_nr ack\_expected = 0;

seq\_nr frame\_expected = 0;

packet buffer[MAX\_SEQ + 1];

int event, arg;

struct FRAME f;

int len = 0;

protocol\_init(argc, argv);

lprintf("Designed by Jiang Yanjun, build: " \_\_DATE\_\_" "\_\_TIME\_\_"\n");

lprintf("Process by CWJ, build: " \_\_DATE\_\_" "\_\_TIME\_\_"\n");

disable\_network\_layer();

while(1) {

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

switch (event) {

case NETWORK\_LAYER\_READY:

get\_packet(buffer[next\_frame\_to\_send]);

nbuffered++;

send\_data\_frame(FRAME\_DATA,next\_frame\_to\_send,frame\_expected,buffer);

inc(next\_frame\_to\_send);

break;

case PHYSICAL\_LAYER\_READY:

phl\_ready = 1;

break;

case FRAME\_RECEIVED:

len = recv\_frame((seq\_nr \*)&f, sizeof f);

if (len < 6 || crc32((seq\_nr \*)&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.info);

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

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

inc(frame\_expected);

start\_ack\_timer(ACK\_TIMER);

}

}

while (between(ack\_expected,f.ack,next\_frame\_to\_send)) {

stop\_timer(ack\_expected);

nbuffered--;

inc(ack\_expected);

}

break;

case DATA\_TIMEOUT:

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

next\_frame\_to\_send = ack\_expected;

for (int i = 1; i <= nbuffered; i++)

{

send\_data\_frame(FRAME\_DATA, next\_frame\_to\_send, frame\_expected, buffer);

inc(next\_frame\_to\_send);

}

break;

case ACK\_TIMEOUT:

send\_data\_frame(FRAME\_ACK, 0, frame\_expected, buffer);

break;

}

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 MAX\_SEQ 63

#define DATA\_TIMER 8420

#define ACK\_TIMER 7350

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

#define inc(k) if(k < MAX\_SEQ) k = k + 1; else k = 0

typedef unsigned char seq\_nr;

typedef unsigned char frame\_kind;

typedef unsigned char packet[PKT\_LEN];

static unsigned char between(seq\_nr a, seq\_nr b, seq\_nr c)

{

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

}

struct FRAME {

frame\_kind kind; /\* FRAME\_DATA \*/

seq\_nr ack;

seq\_nr seq;

packet info;

unsigned int crc32;

};

int no\_nak = 1;

static int phl\_ready = 0;

static void put\_frame(seq\_nr\* frame, int len)

{

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

send\_frame(frame, len + 4);

phl\_ready = 0;

}

static void send\_data\_frame(frame\_kind fk, seq\_nr frame\_nr, seq\_nr frame\_expected, packet buffer[])

{

struct FRAME s;

s.kind = fk; //数据帧类型

s.ack = (frame\_expected + MAX\_SEQ) % (MAX\_SEQ + 1); //ack 等于接收窗口下限减一

stop\_ack\_timer(); //发送帧，关闭ack超时时钟

if (fk == FRAME\_DATA) {

s.seq = frame\_nr; //帧序号

memcpy(s.info, buffer[frame\_nr % NR\_BUFS], PKT\_LEN); //将待发送数据从发送窗口缓存存入帧的数据域

dbg\_frame("Send DATA %d %d, ID %d\n", s.seq, s.ack, \*(short\*)s.info);//调试信息

put\_frame((seq\_nr\*)&s, 3 + PKT\_LEN); //在s+3+PKT\_LEN尾加入四字节crc校验，即存入s->crc32，然后发送数据帧

start\_timer(frame\_nr, DATA\_TIMER); //开始数据帧超时计时器

}

else if (fk == FRAME\_ACK) {

dbg\_frame("Send ACK %d\n", s.ack); //调试信息

put\_frame((unsigned char\*)&s, 2); //在s.ack后面插入crc校验，然后发送长度为6字节的ACK帧

}

else if (fk == FRAME\_NAK) {

dbg\_frame("Send NAK %d\n", (s.ack + 1) % (MAX\_SEQ + 1));

no\_nak = 0; //已经发送NAK

put\_frame((unsigned char\*)&s, 2); //在s.ack后面插入crc校验，然后发送长度为6字节的NAK帧

}

}

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

{

seq\_nr ack\_expected = 0; //发送窗口下限

seq\_nr next\_frame\_to\_send = 0; //发送窗口上限

seq\_nr frame\_expected = 0; //接收窗口下限

seq\_nr too\_far = NR\_BUFS; //接收窗口上限

struct FRAME f; //数据帧

packet out\_buf[NR\_BUFS]; //发送窗口（发送缓存）

packet in\_buf[NR\_BUFS]; //接收窗口（接收缓存）

int arrived[NR\_BUFS]; //接收窗口接收情况

for (int i = 0; i < NR\_BUFS; i++) arrived[i] = 0;

seq\_nr nbuffered = 0; //已发送等待确认的帧数

int event, arg;

int len = 0; //接收的数据帧的长度

protocol\_init(argc, argv);

lprintf("Designed by Jiang Yanjun, build: " \_\_DATE\_\_" "\_\_TIME\_\_"\n");

lprintf("Process by CWJ, build: " \_\_DATE\_\_" "\_\_TIME\_\_"\n");

disable\_network\_layer(); //初始时将disable网络层，等待物理层准备好才enable网络层

while (1) {

event = wait\_for\_event(&arg); //等待事件，DATA\_TIMEOUT事件，传出参数arg记录了超时帧的帧序号

switch (event) {

case NETWORK\_LAYER\_READY: //网络层准备好（ 物理层发送缓冲未满 && 发送窗口未满 ）

nbuffered++; //已发送帧数加一

get\_packet(out\_buf[next\_frame\_to\_send % NR\_BUFS]);//从网络层获取一个报文，存入发送窗口上限

send\_data\_frame(FRAME\_DATA, next\_frame\_to\_send, frame\_expected, out\_buf);//发送数据帧

inc(next\_frame\_to\_send); //发送窗口上限加一

break;

case PHYSICAL\_LAYER\_READY:

phl\_ready = 1;

break;

case FRAME\_RECEIVED:

len = recv\_frame((seq\_nr\*)&f, sizeof(f));

if (len < 6 || crc32((seq\_nr\*)&f, len) != 0) { //收到错误的帧

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

if (no\_nak) //如果NAK还没被发送

send\_data\_frame(FRAME\_NAK, 0, frame\_expected, out\_buf); //发送NAK，请求序号为接收下限frame\_expected的数据帧

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.info);

if ((f.seq != frame\_expected) && no\_nak) { //如果收到的帧不是接收窗口下限的数据帧那么，请求序号为接收下限frame\_expected的数据帧

dbg\_event(" Recv frame is not lower bound, NAK sent back\n");

send\_data\_frame(FRAME\_NAK, 0, frame\_expected, out\_buf);

}

if (between(frame\_expected, f.seq, too\_far) && arrived[f.seq % NR\_BUFS] == 0)//接收数据帧的序号落在接收窗口内，并且该序号数据第一次到达

{

arrived[f.seq % NR\_BUFS] = 1; //标注已到达

memcpy(in\_buf[f.seq % NR\_BUFS], f.info, PKT\_LEN); //存入接收窗口

while (arrived[frame\_expected % NR\_BUFS]) //移动接收窗口

{

put\_packet(in\_buf[frame\_expected % NR\_BUFS], PKT\_LEN); //上传到网络层

no\_nak = 1; //重置no\_nak

arrived[frame\_expected % NR\_BUFS] = 0; //重置上传到网络层的数据帧

inc(frame\_expected); //滑动接收窗口

inc(too\_far); //滑动接收窗口

start\_ack\_timer(ACK\_TIMER);

}

}

}

if (f.kind == FRAME\_NAK && between(ack\_expected, (f.ack + 1) % (MAX\_SEQ + 1), next\_frame\_to\_send)) //人家要的是ack+1也就是对方接收窗口下限的数据

{

send\_data\_frame(FRAME\_DATA, (f.ack + 1) % (MAX\_SEQ + 1), frame\_expected, out\_buf);

}

while (between(ack\_expected, f.ack, next\_frame\_to\_send)) { //不管接收到的帧是数据帧、ACK帧还是NAK帧，f.ack都是有效的

nbuffered--;

stop\_timer(ack\_expected);

inc(ack\_expected);

}

break;

case DATA\_TIMEOUT:

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

send\_data\_frame(FRAME\_DATA, arg, frame\_expected, out\_buf); //重发超时数据帧

break;

case ACK\_TIMEOUT:

send\_data\_frame(FRAME\_ACK, 0, frame\_expected, out\_buf); //直接返回ACK

break;

}

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

enable\_network\_layer();

else

disable\_network\_layer();

}

}