计算机网络

实验一 数据链路层滑动窗口协议的设计与实现

实验报告

姓名：--------

学号：--------

班级：--------

学院：--------

一、 实验内容与实验环境描述

1. 实验任务与实验内容

利用所学数据链路层原理，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。

通过该实验，进一步巩固和深刻理解数据链路层误码检测的 CRC 校验技术，以及滑动窗口的工作机理。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。

2. 实验环境

Windows 8.1 环境下的 PC 机，使用 Visual Studio 2013 集成化开发环境。

二、 软件设计

1. 数据结构

帧

```
struct FRAME {                          //帧的内容
    unsigned char kind;                 //帧的种类，分为 Ack, Nak 和 Data 三种
    unsigned char ack;                  //上一次成功接收的帧的序号
    unsigned char seq;                  //本帧的序号
    unsigned char data[PKT_LEN];        //数据
    unsigned int    padding;            //填充字段
};

#define DATA_TIMER    5000              //Data 帧超时时间
#define ACK_TIMER 280                   //Ack 帧超时时间

#define MAX_SEQ 31                      //帧的序号空间，应当是 2^n-1
#define NR_BUFS 16                      //窗口大小，NR_BUFS=(MAX_SEQ+1)/2
```

全局变量

static unsigned char next_frame_to_send = 0, frame_expected = 0, ack_expected = 0;
//next_frame_to_send:将要发送的帧序号，发送方窗口的上界；frame_expected:期望收到的帧序号，接收方窗口的下界；ack_expected:期望收到的 ack 帧序号，发送方窗口的下界。

static unsigned char out_buffer[NR_BUFS][PKT_LEN], in_buffer[NR_BUFS][PKT_LEN],

nbuffered;
//对应输出缓存、输入缓存，以及目前输出缓存的个数。

static unsigned char too_far = NR_BUFS; //接收方窗口的上界。
static int phl_ready = 0; //物理层是否 ready。
bool arrived[NR_BUFS]; //接收方输入缓存的 bit map。
bool no_nak = true; //是否发送了 NAK，true 则不再重复发送。

主函数内变量

nbuffered = 0; //初始没有输出帧被缓存
int i; //临时变量
int event, arg;
struct FRAME f; //定义帧
int len = 0; //收到 data 的长度

2. 模块结构

static bool between(int a, int b, int c)
如果 b 在 a 和 c 组成的窗口之间，则返回 true，否则返回 false
参数：a，b，c 是要进行比较的数字序号。

static void put_frame(unsigned char *frame, int len)
将 frame 做 CRC 校验后附校验码一起发送到物理层
参数：unsigned char * frame：要发送到物理层的帧。int len：帧的长度。

static void send_data_frame(unsigned char frame_nr)
将输出缓存打包成 frame 后调用 put_frame 发送到物理层，同时开始计时 data_timer
参数：unsigned char * frame：要发送到物理层的帧。

static void send_ack_frame(void)
打包 ack 确认帧并发送到物理层

static void send_nak_frame(void)
打包 nak 确认帧并发送到确认帧

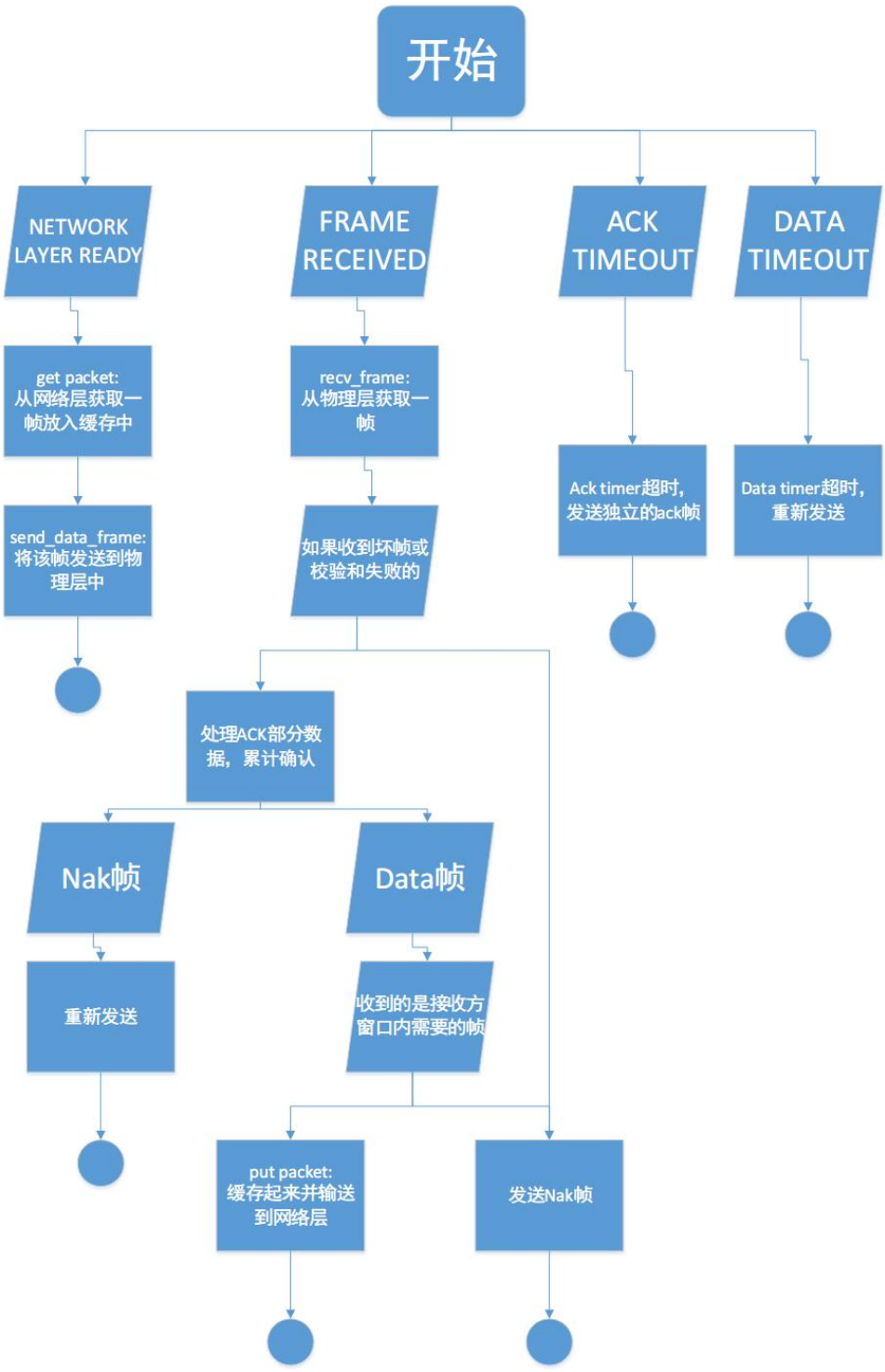
开始
NETWORK LAYER READY
FRAME RECEIVED
ACK TIMEOUT
DATA TIMEOUT
get packet: 从网络层获取一帧放入缓存中
send_data_frame: 将该帧发送到物理层中
recv_frame: 从物理层获取一帧
如果收到坏帧或校验和失败的
Ack timer超时，发送独立的ack帧
Data timer超时，重新发送
处理ACK部分数据，累计确认
Nak帧
Data帧
重新发送
收到的是接收方窗口内需要的帧
put packet: 缓存起来并输送到网络层
发送Nak帧

图 1　算法流程图

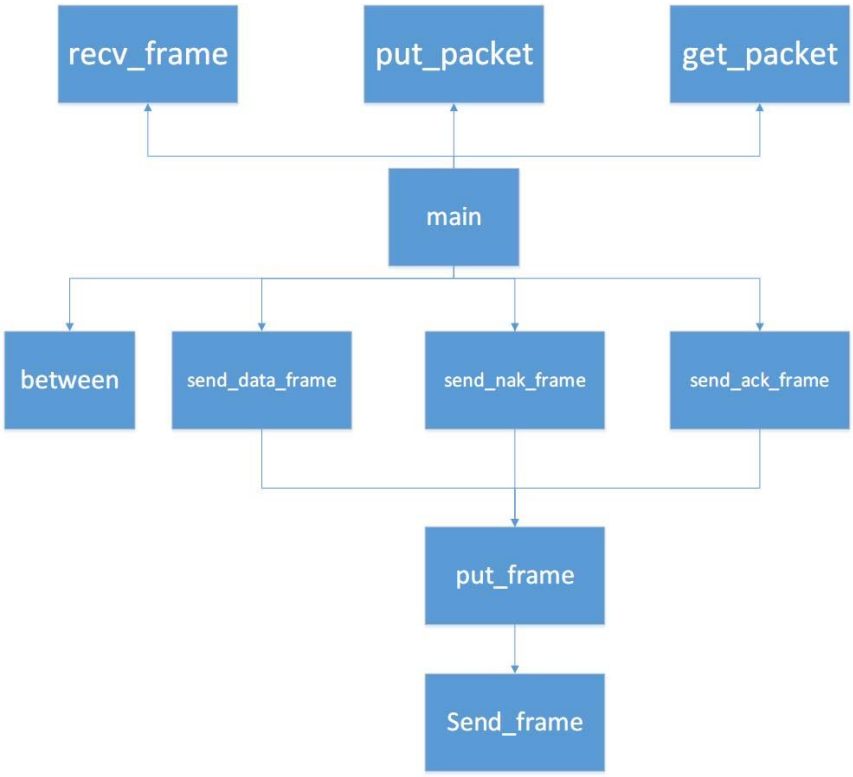

图 2 模块间调用图

三、 实验结果分析

（1） 程序的健壮性

本次实验采用的是选择重传协议，能够在有误码信道环境中实现无差错传输，同时鉴于设计的程序最长运行1个小时，证明其可以长时间运行，健壮性较好。

（2） 协议参数的选取：

经多次实验测得当滑动窗口的大小MAX_SEQ为31，缓冲区的大小NR_BUFS为16，重传定时器data_timer为5000ms,ACK定时器ack_timer为300ms时效率较高。

当数据包从物理层发出时其长度为256字节+3字节（控制信息）+4字节（校验字长）=263字节。发送一帧用时发送一帧用时：t1 = $\frac{263*8}{8000}$ s = 263ms。而数据链路层发送该帧并收到ACK的最短时间是：t2 = t1 + 270 + 300 +270 = 1103ms。所以NR_BUFS应大于等于$\frac{t_2}{t_1} = 1103 \div 263 \approx 4.3$，此时MAX_SEQ=15而NR_BUFS=8。

值得注意的是这只是无误码情况下得出的结果。当有误码时用时会增大，因而MAX_SEQ和NR_BUFS应当相应增大。而重传时限也相应增大为2000ms。

（3） 理论分析

1. 无误码的情况

在数据帧中，每一帧263字节，帧头3个字节，帧尾4个字节，所以信道利用率为:

$$\frac{256}{263} \times 100\% = 97.3\%$$

2. 有误码的情况

这里假设重传操作及时，重传的数据帧的回馈，ACK 帧可以100%正确传输。

当误码率为10^{-5},即发送10^5个比特，出现一个错误，则平均发送$\frac{100\,000}{263*8} = 47.53 \approx 48$个帧会出现一个错误。

此时信道利用率为：$\frac{48}{48+1} * 97.3\% = 95.3\%$

当误码率为10^{-4}，即发送10^4个比特，出现一个错误，则平均发送$\frac{10\,000}{263*8} = 4.75 \approx 5$个帧就会出现一个错误。

此时信道利用率为：$\frac{5}{6}$ * 97.3% = 81.0%

（4） 实验结果分析

序号	命令	说明	运行时间(秒)	效率(%)		备注
				A	B	
1	datalink au datalink bu	无误码信道数据传输	1976.687	52.79	96.87	
2	datalink a datalink b	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”	2164.340	52.79	94.59	
3	datalink afu datalink bfu	无误码信道，站点 A 和站点 B 的分组层都洪水式产生分组	2860.435	96.97	96.97	
4	datalink af datalink bf	站点 A/B 的分组层都洪水式产生分组	2013.335	94.73	94.70	
5	datalink af –ber 1e-4 datalink bf –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}	1876.377	57.40	56.18	

表 1 性能测试记录表

相较于理论计算来说实际实验得到的效率与之相差 1%，由于存在对帧的处理时间以及受到硬件的影响，此误差可以忽略。而在误码率为的 10^{-4} 情况下误差较大，鉴于设计的程序是在无差错环境下测试而设置的滑动窗口大小以及超时时长，需要对原先代码中的参数进行调整才能获得较高的效率。

（5） 存在的问题

目前来说此参数并不是最优参数，其效率未能达到最大值。

四、 研究和探索的问题

1. CRC 校验能力

本次实验使用的 32 位的 CRC 校验码。在理论上，可以检测出：所有的奇数个个错误， 所有双比特错误，所有小于等于 32 位的突发错误。但是检测不错大于 32 位的突

发错误。因此如果出现 CRC32 不能检测出的错误，至少需要出现 33 位突发错误。

而这一概率为 5.160×10^{-93} 可以认为这一事件为不可能事件，客户每天实际可以发送的帧数目为 94.6% * 8000 / 256 / 8* 60 * 60 * 24/ 2 = 159638 帧

所以发生这一事件至少需要 $1.937\times10^{\wedge92}$ / 159638　/ 365 = $5.3091\times10^{\wedge91}$ 年。

2. get_ms()和 log_printf 的实现

C 语言的 time.h 当中提供了一些关于时间操作的函数可以实现 get_ms()函数。可以利用的函数有 clock()函数原型为：clock_t clock()该函数返回程序开始执行后占用的处理器时间，如果无法获得占用时间则返回-1。因为我们计时的起点并不是程序开始之时，而是开始通信之时，所以需要一个静态变量 start_time 来记录通信起始的时间。然后在每次调用 get_ms()后，获取当前的时间 current_time。然后再返回 start_time-current_time 即可。printf 是用变长参数实现的。prinf 的函数原型为 printf(char * fmt,…),后面…即表示变长参数。通过对 fmt 字符串进行解析，将 fmt 字符串转化成最终的字符串，然后通过系统调用输出到屏幕上。

3. 软件测试方面的问题

设置多种测试方案的目的在于测试程序能否在正常环境下运行，通过多种测试方案来模拟现实中的多种情况。

4. 对等协议实体之间的流量控制

鉴于选择重传协议有发送方窗口和接收方窗口限制，当发送方流量过大时接收方窗口将拒绝接收数据进而使发送方计时器超时而选择重传，同时发送方受发送方窗口大小限制不能突发发送大量数据。因此可以认为对等协议实体之间有流量控制。

五、 实验总结与心得体会

本次实验在课下学习理解实验书用了大约 3-4 小时，实际上机撰写代码并调试的时间为 4 小时。由于直接从老师的项目中开始编辑，因此在编程环境上没有出现问题，而编程过程中没有发现 C 语言方面的问题。协议参数用了很长时间才测试出较为理想的结果，而通过本次实验的代码编写对数据链路层各功能的实现有了形象的概念，对选择重传协议以及滑动窗口的概念有了更加深刻的了解。本组实验只有一个人参加，虽然工作量较大，但是通过对整个实验过程的亲身经历，对实验有了更加全面而细致的了解，也深切理解到真正的协议制定及参数调整其不易之处。

```c
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

#include "protocol.h"
#include "datalink.h"

#define DATA_TIMER  5000 //Data帧超时时间
#define ACK_TIMER 280 //Ack帧超时时间

#define MAX_SEQ 31 //帧的序号空间，应当是2^n-1
#define NR_BUFS 16 //窗口大小，NR_BUFS=(MAX_SEQ+1)/2
#define inc(k) if(k < MAX_SEQ)k++;else k=0 //计算k+1

struct FRAME { //帧的内容
    unsigned char kind; //帧的种类，分为Ack, Nak 和 Data 三种
    unsigned char ack; //上一次成功接收的帧的序号
    unsigned char seq; //本帧的序号
    unsigned char data[PKT_LEN]; //数据
    unsigned int  padding; //填充字段
};

static unsigned char next_frame_to_send = 0,
frame_expected = 0, ack_expected = 0;
//next_frame_to_send:将要发送的帧序号，发送方窗口的上界；
frame_expected:期望收到的帧序号，接收方窗口的下界；
ack_expected:期望收到的ack帧序号，发送方窗口的下界
static unsigned char out_buffer[NR_BUFS][PKT_LEN],
in_buffer[NR_BUFS][PKT_LEN], nbuffered;
//对应输出缓存、输入缓存，以及目前输出缓存的个数
static unsigned char too_far = NR_BUFS; //接收方窗口的上界
static int phl_ready = 0; //物理层是否ready
bool arrived[NR_BUFS]; //接收方输入缓存的bit map
bool no_nak = true; //是否发送了NAK，true则不再重复发送

static bool between(int a, int b, int c){ //如果b在a和c组成
的窗口之间，则返回true，否则返回false
    return((a <= b) && (b < c)) || ((c < a) && (a <= b))
|| ((b < c) && (c <a));
}

static void put_frame(unsigned char *frame, int len){
    //将frame做CRC校验后附校验码一起发送到物理层
    *(unsigned int *)(frame + len) = crc32(frame, len);
    send_frame(frame, len + 4);
    phl_ready = 0; //将数据发送到物理层后将物理层置为忙碌状态
}

static void send_data_frame(unsigned char frame_nr){
    //将输出缓存打包成frame后调用put_frame发送到物理层，同时开始
计时data_timer
    struct FRAME s;

    s.kind = FRAME_DATA;
    s.seq = frame_nr;
    s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
    memcpy(s.data, out_buffer[frame_nr % NR_BUFS],
PKT_LEN); //将out_buffer里的缓存复制到帧frame的data域中

    dbg_frame("Send DATA %d %d, ID %d\n", s.seq, s.ack,
*(short *)s.data);

    put_frame((unsigned char *)&s, 3 + PKT_LEN); //将打包好
的帧发送到物理层

    start_timer(frame_nr % NR_BUFS, DATA_TIMER);
    stop_ack_timer();
}

static void send_ack_frame(void){
    //打包ack确认帧并发送到物理层
    struct FRAME s;

    s.kind = FRAME_ACK;
    s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);

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

    put_frame((unsigned char *)&s, 2);
    stop_ack_timer();
}

static void send_nak_frame(void){
    //打包nak确认帧并发送到确认帧
    struct FRAME s;
```

```
    s.kind = FRAME_NAK;
    s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);

    no_nak = false;

    dbg_frame("Send NAK with ACK %d\n", s.ack);

    put_frame((unsigned char *)&s, 2);
    stop_ack_timer();
}

int main(int argc, char **argv)
{
    //初始化
    nbuffered = 0; //初始没有输出帧被缓存
    int i;
    for (i = 0; i < NR_BUFS; i++) //初始时没有输入帧被缓存
        arrived[i] = false;
    int event, oldest_frame;
    struct FRAME f;
    int len = 0; //收到data的长度

    protocol_init(argc, argv); //协议初始化
    lprintf("Designed by Hong Zhilong, based on Teacher
Jiang's code.\nBuild: "__DATE__"  "__TIME__"\n");

    disable_network_layer();

    for (;;)
    {
        event = wait_for_event(&oldest_frame);

        switch (event) //对不同事件进行不同处理
        {
            case NETWORK_LAYER_READY: //当网络层ready时
                get_packet(out_buffer[next_frame_to_send %
NR_BUFS]); //从网络层获取一帧放入输出缓存中
                nbuffered++; //缓存序号+1
                send_data_frame(next_frame_to_send);//发送
该帧
                inc(next_frame_to_send); //将将发送方窗口上限
+1
                break;

            case PHYSICAL_LAYER_READY: //当物理层ready时，将
phy_ready置为1以便之后enable网络层
                phl_ready = 1;
                break;

            case FRAME_RECEIVED: //当物理层收到一帧时
                len = recv_frame((unsigned char *)&f,
sizeof f); //从物理层获取一帧
                if (len < 5 || crc32((unsigned char *)&f,
len) != 0) { //如果接收坏帧或CRC校验失败，则返回nak帧
                    dbg_event("**** Receiver Error, Bad
CRC Checksum\n");
                    if (no_nak)
                        send_nak_frame();
                    break;
                }
                if (f.kind == FRAME_ACK) //如果是ack帧的话，
由于所有帧都含有ack帧，因此统一处理
                    dbg_frame("Recv ACK  %d\n", f.ack);

                if (f.kind == FRAME_DATA) //如果是data帧
                {
                    if ((f.seq != frame_expected) &&
no_nak) //如果收到的是不需要的帧则返回nak
                        send_nak_frame();
                    else
                        start_ack_timer(ACK_TIMER);
                    if (between(frame_expected, f.seq,
too_far) && (arrived[f.seq%NR_BUFS] == false))
                    //如果收到的帧在接收方窗口内且这一帧未被接收
过
                    {
                        dbg_frame("Recv DATA %d %d,
ID %d\n",f.seq, f.ack, *(short *)f.data);
                        arrived[f.seq%NR_BUFS] = true; //
标记该帧为已接受
                        memcpy(in_buffer[f.seq % NR_BUFS],
f.data,PKT_LEN); //将接收到的帧的data域复制至输入缓存中
                        while (arrived[frame_expected %
NR_BUFS])//当收到接收方窗口下界的一帧时，对这一帧以及之后收到的帧进
行处理
                        {
```

```
    put_packet(in_buffer[frame_expected %NR_BUFS], len - 7); //将输入缓存送至网络层
                            no_nak = true;
                            arrived[frame_expected % NR_BUFS] = false;
                            inc(frame_expected); //将窗口前移一位
                            inc(too_far);
                            start_ack_timer(ACK_TIMER); //如果ack_timer超时则发送ack帧
                        }
                    }
                }

                if ((f.kind == FRAME_NAK) && between(ack_expected, (f.ack + 1) % (MAX_SEQ + 1), next_frame_to_send))
                //如果收到的是nak帧且ack的下一帧在发送方窗口里，则发送ack的下一帧
                {
                    send_data_frame((f.ack + 1) % (MAX_SEQ + 1));
                    dbg_frame("Recv NAK with ACK %d\n", f.ack);
                }

                while (between(ack_expected, f.ack, next_frame_to_send))
                //累计确认，只要ack在发送方窗口内就不断地将窗口前移直至未确认的一帧
                {
                    nbuffered--;
                    stop_timer(ack_expected%NR_BUFS);
                    inc(ack_expected);
                }
                break;

            case ACK_TIMEOUT://ack_timer超时时发送独立的ack帧
                dbg_event("---- DATA %d timeout\n", oldest_frame);
                send_ack_frame();
                break;

            case DATA_TIMEOUT: //某帧的data_timer超时说明未收到接收方的ack帧，则重新发送该帧
                dbg_event("---- DATA %d timeout\n", oldest_frame);
                if (!between(ack_expected, oldest_frame, next_frame_to_send))
                    oldest_frame = oldest_frame + NR_BUFS;
                send_data_frame(oldest_frame);
                break;
        }

        if (nbuffered < NR_BUFS && phl_ready) //如果物理层ready且目前缓存未满，则使网络层ready
            enable_network_layer();
        else
            disable_network_layer();
    }
}
```