滑动窗口协议的设计 & 实现

王鹏

学号：2018211341　　班级：2018211308

更新：June 7, 2020

目录

---

1 实验内容及要求

1.1 实验任务

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。相关数据如下：

- 信道模型：8000bps 全双工卫星信道
- 信道传播时延：270 毫秒
- 信道误码率：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

目标：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。

1.2 实验内容

- 实现有噪音信道环境下的无差错传输。
- 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，计算出优取值，并通过程序的运行进行验证。

1.3 实验环境

操作系统：Windows 10

编程语言：C 语言

集成开发环境：Microsoft Visual Studio 2019

2 软件设计与实现

2.1 数据结构

```
#define ACK_TIMER 280                    /*ACK帧超时重传时间*/
#define DATA_TIMER  4100                 /*数据帧超时重传时间*/
#define MAX_SEQ 31                       /*帧的序号空间，应当是2^n-1,序号上限
    */
#define NR_BUFS ((MAX_SEQ+1)/2)          /*窗口大小上限*/

#define inc(k) if(k < MAX_SEQ)k++;else k=0   /*计算k+1*/

struct FRAME {
unsigned char kind;                  /*帧的种类，ACK，NAK，DATA*/
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*/
};

struct ACK_FRAME
{
unsigned char kind;                  /*帧的种类*/
unsigned char ack;                   /*序号*/

unsigned int padding;                /*填充字段*/
};

struct NAK_FRAME
{
unsigned char kind;                  /*帧的种类*/
unsigned char ack;                   /*序号*/

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；
*   fram_expected:期望收到的帧序号，接收方窗口的下界；
```

```
37  *   ack_expected:期望收到的ack帧序号，发送方窗口的下界
38  ************************************************************/
39
40  static unsigned char out_buffer[NR_BUFS][PKT_LEN],
41  in_buffer[NR_BUFS][PKT_LEN], nbuffered;
42  /************************************************************
43  *   对应输出缓存，输入缓存，以及目前输出缓存的个数
44  ************************************************************/
45
46  static unsigned char too_far = NR_BUFS;      /*接收方窗口的上界*/
47  static int phl_ready = 0;                    /*物理层是否ready*/
48  bool arrived[NR_BUFS];                       /*接收方输入缓存的bit map*/
49  bool no_nak = true;                          /*是否发送了NAK，true则不再重复
        发送*/
```

2.2 模块划分

本次我实现的选择重传协议大致可分为五个小模块和一个主模块。下面将详细介绍每个模块内部的具体实现细节。

static void put_frame(unsigned char* frame, int len) 将缓存区中的帧做 CRC 校验后，将 CRC 校验码插入到帧的尾部，即：将附带 CRC 校验码的帧发送至物理层，并将物理层置为忙碌状态。
参数：unsigned char* frame 表示要发送的帧的位置；int len 表示帧的长度大小。

static void send_data_frame(unsigned char frame_nr) 将输出的缓冲区打包成数据帧 data_frame 后通过调用 put_frame 将数据帧发送至物理层。
参数：unsigned char frame_nr 表示要发送的帧的位置。

static void send_ack_frame(void) 将输出的缓冲区打包成 ACK 帧 ack_frame 后通过调用 put_frame 将 ACK 帧发送至物理层。
参数：unsigned char frame_nr 表示要发送的帧的位置。

static void send_nak_frame(void) 将输出的缓冲区打包成 NAK 帧 nak_frame 后通过调用 put_frame 将 NAK 帧发送至物理层。
参数：unsigned char frame_nr 表示要发送的帧的位置。

static bool between(int a, int b, int c) 判断帧的序号是否在滑动窗口的上下沿之间，即：判断是否在滑动窗口内，如果是的话返回 true，否则返回 false。
参数：int a 表示滑动窗口的下沿；int b 表示帧的序号；int c 表示滑动窗口的上沿。

2.3 算法流程

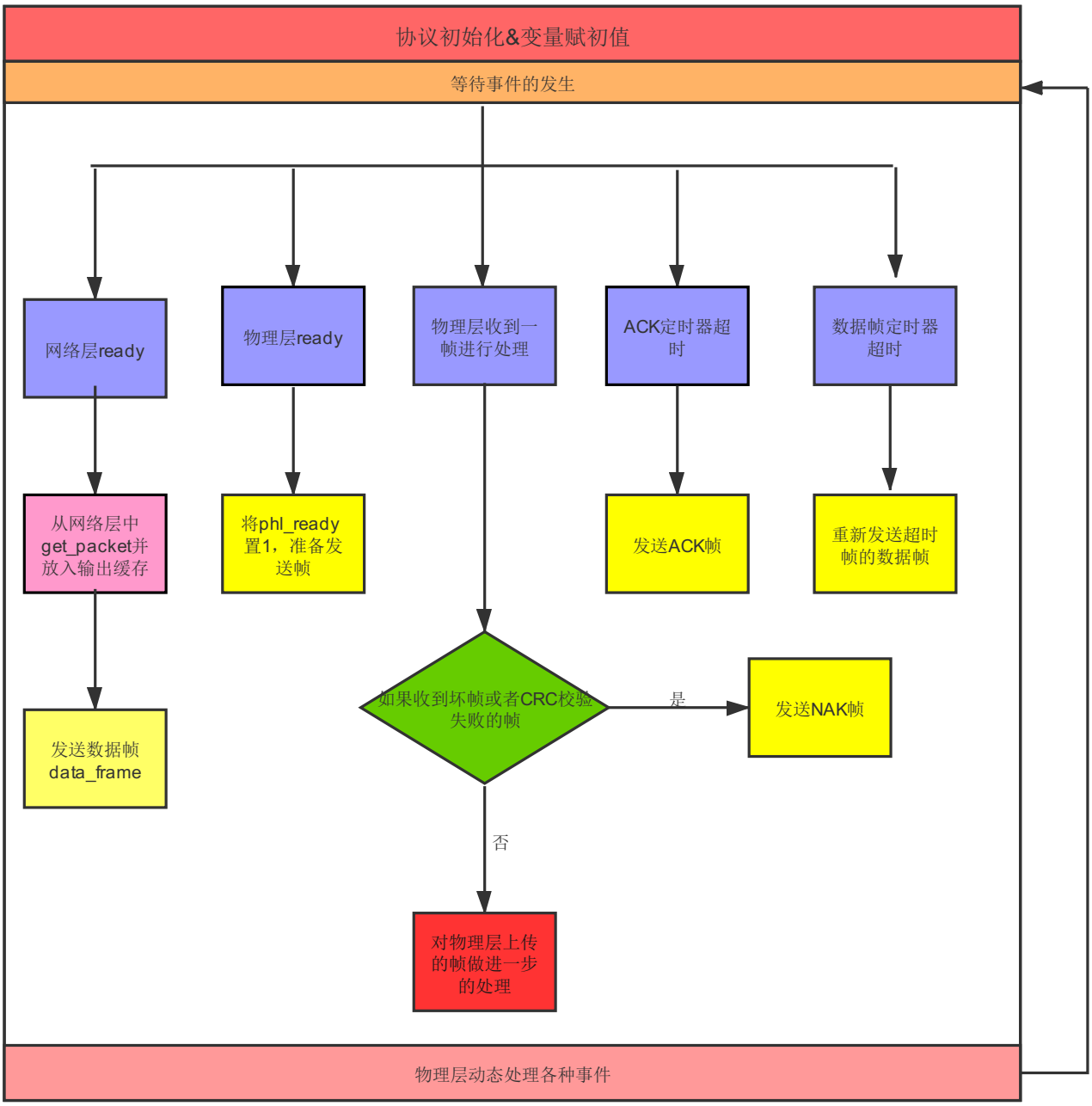

图 1: 选择重传协议框图

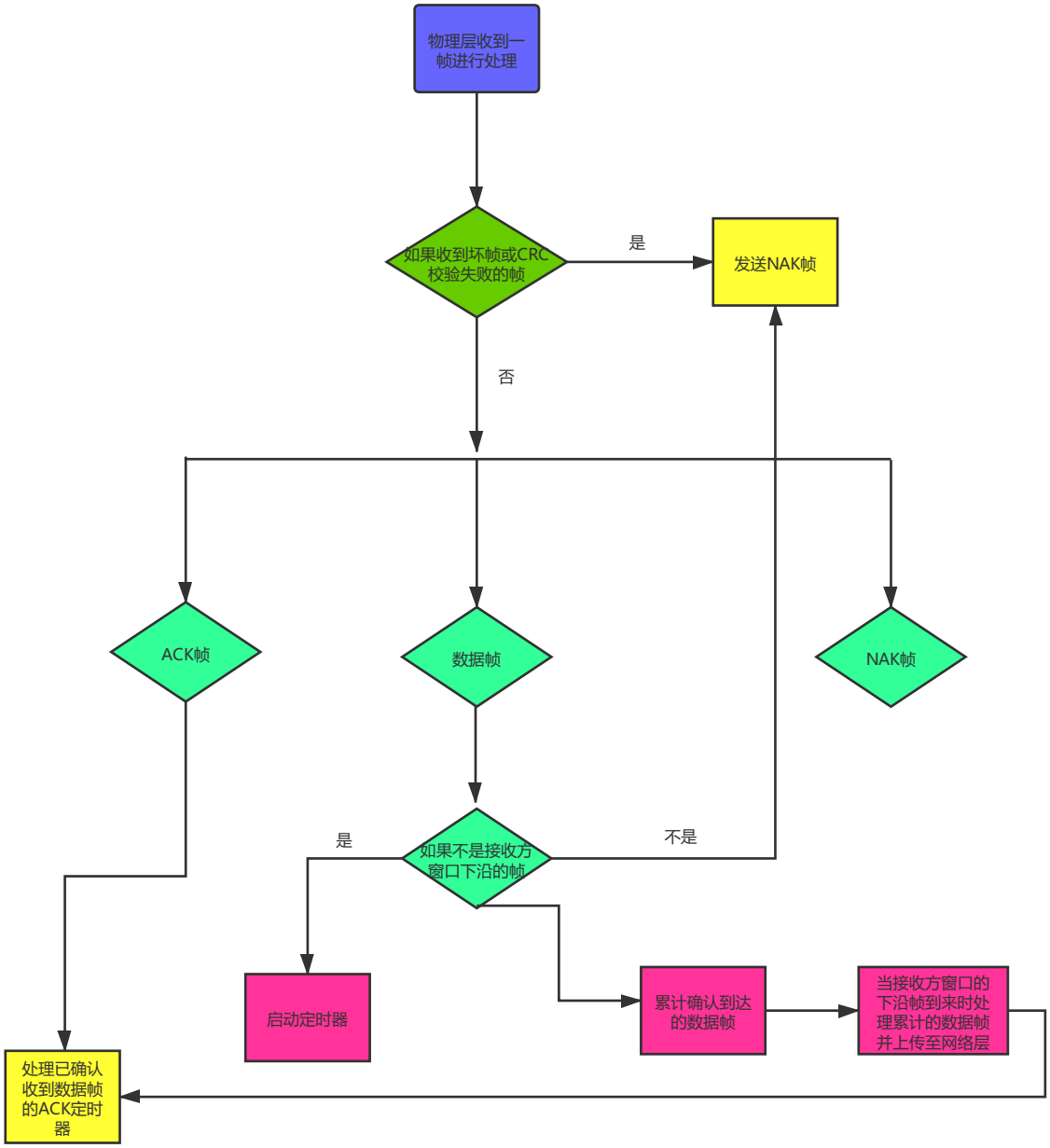

图 2: 从物理层收到一帧的进一步处理

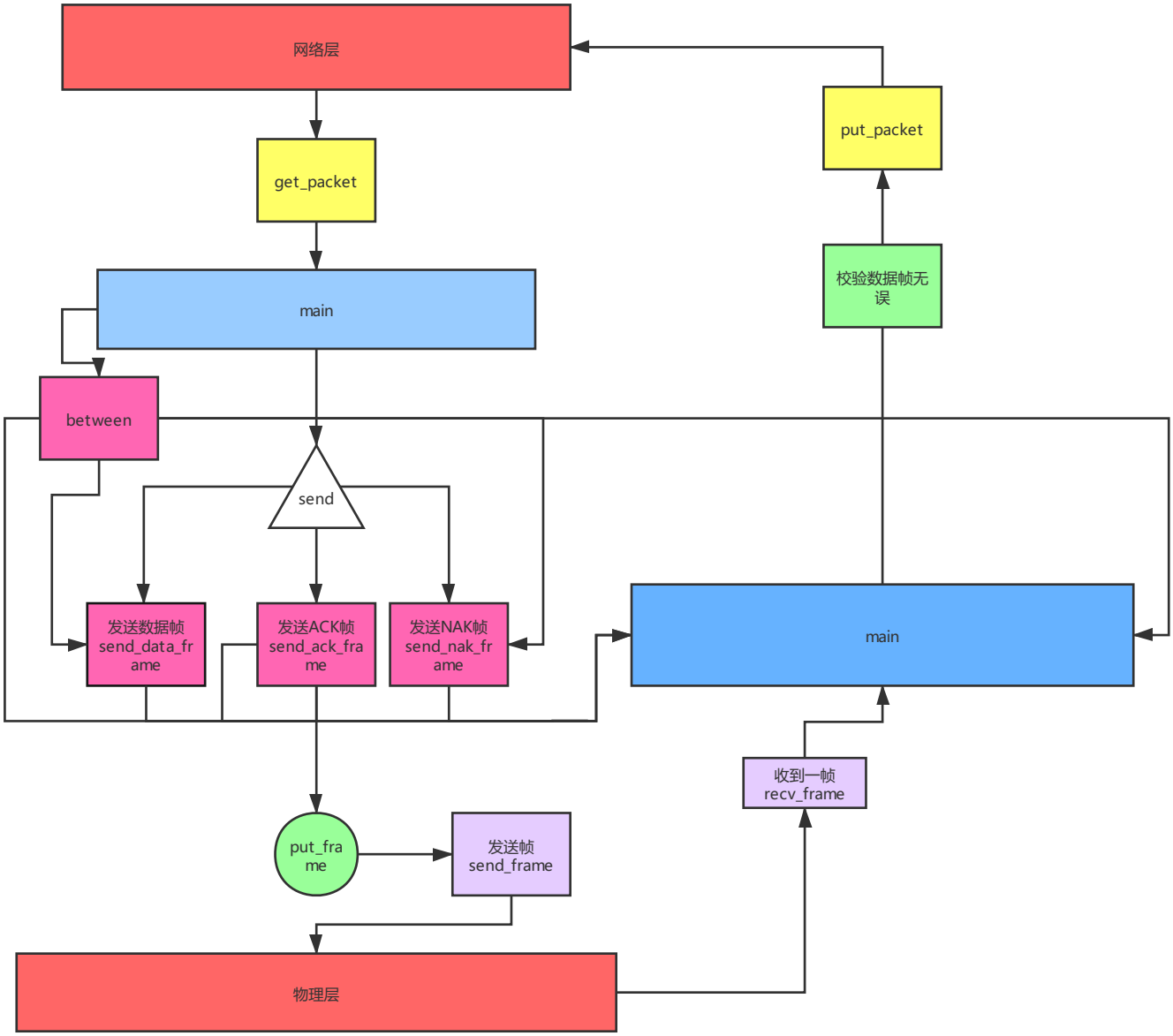
网络层
get_packet
main
between
send
发送数据帧
send_data_fr
ame
发送ACK帧
send_ack_fra
me
发送NAK帧
send_nak_fr
ame
put_fra
me
发送帧
send_frame
物理层
put_packet
校验数据帧无
误
main
收到一帧
recv_frame

图 3: 模块调用关系图

3 实验结果分析

在本次实验中，我前前后后共实现了四五个版本的协议，而在每一版的协议中又具体选择了不同的协议参数，通过不断地调试不同版本的试验协议耗费了我的大量时间和精力，经过我的探索我发现了几个很重要的问题，而对于每一版本的实验，基本上都是围绕着下面的前提进行展开的：

- **没有固定的参数标准**。这里的**固定参数标准**有几层不同的含义。一方面，对于同一协议而言，不同的命令行输入方式下 (不同的信道环境)，协议参数的选择不同，信道的利用率会有很大的不同，另一方面，对于实现的不同版本的协议来说，一种协议的最优参数并不一定适用于另一种版本的协议 (很大程度上根本不适用)。简而言之，就实验本身而言，对于不同的命令行输入 (不同的信道环境)，试图通过固定的参数标准来适应不同的信道环境并且实现信道利用率逼近于理论值的做法过于苛刻且不具有现实意义 (现实信道中，在时间积累达到一定的程度的情况下，信道环境短期内是不变的) 。
- **没有固定的最优信道利用率**。这里的**最优**指的是实验中可以获得的最大的信道利用率。通过多次测试不难发现，实验测出的信道利用率即使在同一操作系统，同一运行环境，同一网络负载，同一工作进程中在不同的时间段范围内信道利用率也是不一样的。由于不清楚软件底层实现的网络层和物理层的具体细节，而且程序中只给出了网络层和物理层的接口，因此测出的信道利用率也只能反映程序一部分的健壮性和鲁棒性，对于程序的优良性能当然也是不能完全反映出来的。
- **没有固定的优良标准**。在实现了不同版本的协议之后，我发现并没有一个固定的标准来评判一种协议相较于另一种协议优在哪里。试图通过同一测试环境来评判不同的协议好坏在我看来是有失偏颇的，因为实现了不同版本的协议之后就不难发现，有的协议更适用于低误码率的信道，有的协议更适用于 flooding 环境下的信道，有的协议更适用于平缓环境下的信道，正如上述所说，即使是同一协议在不同的参数下表现的性能也是参差不齐的。因此，在达到实验基本要求的情况下，在一定范围内信道可以实现可靠的传输就已经满足现实的需求了，也达到了实验的目的 (对选择重传协议的深入理解)。

3.1 程序的健壮性

实验过程中采取滑动窗口的选择重传协议，实验结果证明可以实现在有误码信道环境中的无差错传输。同时，在每种命令行参数环境下，程序均可长时间高效传输运行，多种测试条件都表明了程序具有很好的健壮性。

3.2 协议参数的选取

本次实验中通过理论分析与多次试验测试，我最终选取了 $MAX\_SEQ = 31$ 的滑动窗口，确定了缓冲区 NR_BUFS 的大小为 16，以及其他的一系列参数值。具体的分析过程如下：

- 帧格式的确定。在实验开始最初，我设计的帧格式采用统一的格式，帧格式大致如下图示：

类型（kind）	ack	序号（seq）		数据（data）	CRC
2bits	6bits	6bits		256*8bits	32bits
1 个字节		1 个字节		256 个字节	4 个字节

但是很快我就发现了问题，这样设计帧格式固然有最大效率利用信道带宽的好处，但是出于不浪费字段的原则设计出来的帧在进行数据处理的时候需要以 bits 为单位，这在处理帧的种类，ack 与序号问题时变得极为繁琐，而且由于表示的时候不那么简洁直观，这种编码方式**可扩展性非常差**，且**提高信道利用率的效果有限**，出于让程序变得简洁，直观，易读，易扩展的考虑，在后期我改变了帧的格式，采用了原始的表示方法：

类型（kind）		ack		序号（seq）		数据（data）	CRC
2bits		6bits		6bits		256*8bits	32bits
1 个字节		1 个字节		1 个字节		256 个字节	4 个字节

类型（kind）		ACK		CRC
2bits		6bits		32bits
1 个字节		1 个字节		4 个字节

类型（kind）		NAK		CRC
2bits		6bits		32bits
1 个字节		1 个字节		4 个字节

- 窗口大小的确定。信道传播时延设为 $t_p = 270ms$，发送时延设为 t_f，数据包从物理层发出时的数据长度为 $1+1+1+256+4=263$ 字节，那么可以得出

$$t_f = \frac{263 * 8}{8000} = 263ms$$

数据链路层发送完整一帧的时间为:

$$T_F = t_{tran} + t_{prop} + t_{proc} + t_{ack} + t_{prop} + t_{proc} + t_{tran}$$

由于 t_{proc} 和 t_{ack} 可以忽略不计，考虑

$$U = \frac{W * n * t_{tran}}{T_F} \approx \frac{W * n * t_{tran}}{2 * t_{prop} + 2 * t_{tran}} \geq 1$$

综合有误码率的情况下，相应数据包的重传时间会偏大，因此可以得出在 NR_BUFS 取 16，MAX_SEQ31 的情况下是比较符合理想滑动窗口大小的条件的。

- 超时参数的设计。本次实验有两个时间参数，数据帧超时重传时间 $DATA\_TIMER$，ACK 帧的忍耐时间 ACK_TIMER。

a) 选择重传协议使用了 piggybacking 技术，从 piggybacking 的实际传输角度来看，发送方从发出完整的一个帧，到接收到带这个帧 ACK 的数据，总共需要的时间为 $2*(t_f+t_p)$。然而实际的传输时间为

$$T_F = t_{tran} + t_{prop} + t_{proc} + t_{ack} + t_{prop} + t_{proc} + t_{tran}$$

综合以上考虑，加上多次实验测试，初选定 $DATA\_TIMER$ 的值为 4100。

b) 为了避免 ACK 帧的等待时间过长，在协议的实现中选定了 ACK_TIMER，理论分析来看 $DATA\_TIMER$ 和 ACK_TIMER 应该满足一定的定量关系：

$$ACK\_TIMER \leq DATA\_TIMER - 2(t_p + t_f)$$

然而在实际测试中，这个不等关系的用途并不是很大，因为 ACK_TIMER 可以取定的取值空间太大了。

c) 从 a) 和 b) 的分析来看，似乎 $DATA\_TIMER$ 和 ACK_TIMER 的取值在一个固定的值左右波动，实际上却并不是这样的，我通过多次试验证明了 ACK_TIMER 和 $DATA\_TIMER$ 只要满足一定的定量约束关系，取值范围的波动范围倒是非常地大。这个自然是因为滑动窗口本身的约束式:

$$U = \begin{cases} 1, & W \geq 2a+1 \\ \frac{W}{2a+1}, & W \leq 2a+1 \end{cases}$$

3.3 理论分析

- 无误码的情况下。如果不考虑 ACK 帧和 NAK 帧，那么对于**数据帧**来说，每一帧共 263 字节，其中的数据共占有 256 字节，此时最大理论信道利用率为：

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

- 有误码的情况下。假设信道的误码率为 δ，则对于数据帧而言，长度为 263 字节，一个完整的数据帧成功发送的概率为：

$$P = (1-\delta)^{263*8} = (1-\delta)^{2104}$$

设一帧平均重传的次数为 N_r，那么会有以下等式：

$$N_r = \sum_{i=1}^{\infty} i(1-p)^{i-1}P = \frac{1}{p}$$

此时理论最大信道利用率为：

$$U = \frac{T_{tran}}{2 * N_r(T_{tran} + T_{prop})} = \frac{p}{1+2\alpha}$$

a) 若误码率为 10^{-5}。经计算，此时信道利用率为 94.38%。

b) 若误码率为 10^{-4}。经计算，此时信道利用率为 77.02%。

3.4 实验结果分析

序号	命令选项	说明	运行时间（秒）	selective 算法线路利用率	
				A	B
1	–utopia	无误码信道数据传输	1921.01	52.96	96.65
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”	1901.03	51.28	93.15
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	1754.38	96.56	96.56
4	–flood	站点 A/B 的分组层都洪水式产生分组	1949.76	93.36	93.35
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	2296.83	61.16	57.86

从实验结果来看，可以分析该协议的以下特点：

- 高效性和健壮性。与理论的信道利用率相比，实际的信道利用率与之相差甚微，可以忽略不计，且在各种不同的信道模式下都实现了较高的信道利用率，说明了实现的选择重传协议具有比较优良的性能，可以完成对不同场景下信道的充分适用。
- 稳定性。经过多次试验，反复取不同的参数值证明了实现的协议可以稳定地运行较长时间，且在较长时间利用信道的情况下仍然可以保持较高的信道利用率。
- 在高误码率情况下，信道利用率会降低。在误码率为 10^{-4} 的情况下，A，B 整体的信道利用率都不算特别优秀，没有接近理论的最大信道利用率，不过却也在相同实例程序的演示中达到了较好的效果。分析原因在于，这个信道的整体设计就不是为高误码率信道的情况下所设计的，考虑到老师上课提到过误码率为 10^{-4} 的高误码率信道本身就很少见，即使是**双绞线**这种传输介质非常差的信道的误码率也不至于那么高，因此为高误码率信道设计算法本身就是一个没有太大意义的事情。其在高误码率的信道情况下，信道利用率低的本质原因在于，此时的接受窗口**错帧大量累计**，导致接收方一直在向发送方不停地发 NAK 帧，根本来不及处理 ACK 帧，更不用说发送数据帧了。接受方不停地发送 NAK，发送发不停地重复发送错帧，导致了**成功发送一帧变成了一件非常不容易的事情**的后果。

3.5 存在问题的解决方案

前文已经提到了高误码率情形下，选择重传协议所存在的问题及本质原因，不过鉴于 10^{-4} 甚至大于 10^{-4} 误码率的信道本身的存在就没有太大意义，本次实验的目的也不是单纯地为了增长百分点，因此没有将解决方案进行代码实现，不过还是提出了两种理论上可行的针对**高误码率**信道的解决方案：

- **设置 NAK 帧缓冲区 NAK_BUFFER 和定时发送 NAK 帧的时间 NAK_TIMER**。如前文所述，当误码率较高时，接收方大量累计错帧，不停地发送 NAK 帧，毫无意义地占用了信道带宽，此时应该将错帧放入 NAK_BUFFER 中，并设置 NAK_TIMER，定时发送 NAK 帧，避免 无效的，重复性地 发送 NAK 帧。
- **改变帧的格式**。针对高误码率信道，显然成功发送一帧变成了极为奢侈的一件事情，那么这时接收方在频繁发送 NAK 帧的时候，为了充分利用信道的带宽，应该在发送 NAK 帧的同时发送数据帧，通过设置多个 BUFFER，尽量充分利用每一次 piggybacking 的结果。此时 piggybacking 不再是只有捎带 ACK，当错帧大量淤积时，同样应该捎带 NAK 。

4 研究和探索的问题

4.1 CRC 校验能力

在本次实验中，采用的是 32 位的 CRC 校验码。在理论上，可以检测出：所有的奇数个错误，所有双比特错误与所有小于等于 32 位的突发错误。但是检测不出大于 32 位的突发性错误，考虑到在实际的过程中，如果检测不出大于 32 位的突发性错误，那么势必会在帧传输的过程中出现 33 位以上的突发性错误，这种可能性基本为零。

考虑在误码率为 10^{-5} 的信道环境下，在一帧中出现错误位数的个数的期望：

$$\lambda = 2104 \times \eta = 0.02104$$

若将此过程视为泊松分布的过程，那么在一帧的传输过程中连续出现 33 位以上的错误的概率为：

$$p = \sum_{k=33}^{2104} \frac{\lambda^k}{k!} \times e^{-\lambda} = 5.148 \times 10^{-93}$$

客户每天实际可以发送的帧数目：

$$\frac{94.5\% \times 8000 \times 60 \times 60 \times 24}{256 \times 8 \times 2} = 159468\text{帧}$$

而发生这一事件需要的时间大致估算为 5.25×10^{91} 年。因此实际情况下，CRC 检错能力还是比较厉害的。

4.2 程序设计的问题

- 协议实现中的 get_ms() 函数不是 C 语言中的库函数，可以利用 C 语言库 <time.h> 实现。通过 clock_t clock() 函数返回程序开始执行后的占用 CPU 的时间。
- printf 风格的函数是通过变长参数实现的。通过阅读源代码可以发现，printf 的函数原型为 (char * fmt, • • •)，这其中的 • • • 代表的就是变长参数。通过指针对 fmt 字符串进行解析，将 fmt 字符串转化为最终的字符串，并通过系统的调用将其输出到屏幕上。
- start_timer() 和 start_ack_timer() 两个函数启动的时机不同，在定时器到来之前重新调用函数对原残留时间的处理方式也不同。这是因为两个函数的功能有本质上的区别，start_timer() 考虑的是数据帧以及数据帧对应 ACK 帧的丢包问题，因此需要对发送方发出的每一个数据帧都应进行 start_timer() 记时，以防止因为个别丢包问题导致降低通信效率，数据传输失败以及信道的带宽不能充分利用等问题，这也是为什么每个数据帧都要对应一个 timer()；而 start_ack_timer() 考虑的是在选择重传协议过程中接收方能否**及时合理地**进行判断 ACK 帧到底是应该单独发送还是应该 piggybacking，而 ACK 帧的定时器之所以不能够被覆盖是因为滑动窗口协议本身的局限性，我们考虑的是第一个被成功接收但是却没有及时发送 ACK 确认的数据帧。

4.3 软件测试方面的问题

经多次试验分析，可以得出结论：**本次实验所完成的程序在各种情况下都能够正确工作**。设计多种测试方案的主要目的如下：

- **-u-无误码信道：** 测试协议在理想信道的情况下的最大吞吐量。-u 的设计方案主要受**帧格式**的影响。换言之，帧格式直接决定了在理想信道环境下所能达到的最大传输效率，若控制字段的每一位都充分利用，那么在此信道上的效率就会很高。
- **-fu-洪水分组无误码信道：** 测试信道在大量数据需要传输的情况下，协议的抗击打能力如何，在洪水模式下，信道会源源不断地接收数据，传输数据，在这种信道环境下，协议的效率主要受 ACK_TIMER 和 NAK 帧的设计与发送时机的影响。
- **-ber=1e-4 高误码率信道：** 在高误码率的测试方案下，主要考察了对于 NAK 帧的设计，选择重传的协议参数的设定，NAK 帧和 ACK 帧合理的发送时机等，考虑到现实生活中很多地方都已经实现了光纤到户的政策，信道的环境已经在不断地得到改善，偶尔的噪声影响已经不那么常见，因此该测试方案在实际的应用场景中个人认为并没有太大的意义。

本次实验的测试环节中，老师给出的几种测试方案已经很大程度上反应了协议设计的优良程度，基本上可以完全覆盖实际场景中可能遇到的各种问题，所以在测试方案方面并没有太多东西需要补充。

4.4 对等协议实体间的流量控制

- 协议本身提供了 enable_network_layer() 和 disable_network_layer() 两个函数接口，发送方发送的数据量由于受到滑动窗口自身的局限性，只有在可利用的窗口范围之内，才可以正常发送数据，不可以突然发送大量的数据。
- 接收方突然受到大量数据的时候，如果在滑动窗口的范围之内无法处理，那么就会使得相应数据帧的计时器超时而选择重传，这在一定程度上同样可以实现接收方数据的流量控制。

5 实验总结和心得体会

在本次实验中，由于为了想要比较深刻地理解滑动窗口协议实现的基本原理，我没有选择一次性地将实验做完，而是选择了每天都对协议本身进行思考，写下一些总结，记录一些主要的实验流程和过程。大概的时间线是这样的：

- 花了两三天的课余时间对协议本身的内容进行总体上的把握，包括协议实现的顶层设计，协议中给出了函数接口和参数，协议实现的基本原理和细节，滑动窗口协议和选择重传协议的优缺点比较，实验环境的搭建和配置。这其中，我花了很多时间在琢磨实验指导书上的大致框架，包括函数接口，数据结构，基本原理，伪代码，测试方案等等。
- 又花了两三天的时间用来思考一些非常细节的问题，个人感觉本次实验比较难的地方在于三个方面。其一，选择重传协议本身不是单方的实现，既要考虑发送方，又要考虑接收方，双方在协议的实现过程中用的都是一套的 datalink.c 的代码，这对换位思考的要求就很高；其二，选择重传协议的实现中，时间点特别地多，过程本身很复杂，整个函数的运行是一个动态的过程，需要考虑的场景也很多，流量的大小，时间点的设置，分组或者洪水发送，这都是需要仔细斟酌的问题；其三，细节问题很重要，由于数据帧的格式都是按照位来设计和处理的，如果在协议的实现过程中不能够对位操作进行非常好的处理，那么整个实验的实现基本上就无从谈起，还有其他非常值得注意的细节，在这里不再一一赘述。
- 花了大致一天的课余时间进行代码的实现和 debug。由于前期工作做得比较好，对程序的整体架构和细节实现的方面都想得非常清楚，在代码的实现环节并没有花太多的时间进行 debug，基本上按照前几天想的大致思路和流程很顺利地就将可以跑的代码实现了出来。
- 花了大概一个星期的课余时间进行了协议原理的分析，调试，改变协议的实现，比较不同协议的性能，流程图的绘制以及文档的撰写。本次实验的代码实现本身比较容易，但是每次修改参数，都需要很长时间的实验和运行，如果想要达到非常高的性能，就需要对实验参数进行非常多的尝试，尽管我在横向和纵向的对比过程中发现了很多问题，得到了很多有趣的结论，但是在实验的最后我依然感觉并没有找到最优的参数 (前文已经提到实验本身并不存在唯一的最优方案)。后来，我意识到了如果协议本身的实现没有什么问题，那么通过不断寻找参数来提高信道利用率其实是一件不太有意义的事情，因为对于数据传输来说，多提高一两个百分点并没有太大的实质性的改变。

本次实验的过程中虽然一个人工作量巨大，但是实验过程的本身是一个非常有趣的过程，这其中对于滑动窗口协议实现的理解，对于程序设计思想的体会等过程让我对数据链路层的实现

有了更为深刻的认识。协议本身的实现其实是并不容易的，考虑到实际传输的多种复杂现实情境，让代码的实现又增加了很多难度，但是在这其中，我却感受到了思考协议实现问题所带来的快乐。

最后看到了自己实现的代码能够对各种测试方案都很好地 work，对于问题本质的思考让程序的实现有了很好地保障，理论分析和实验结果的对比和改进让程序变得更加简洁和高效，虽然花费了很多的时间，但其实还是挺快乐的一件事吧。

A 附录代码

注：由于 PDF 文档宽窄有限，附录代码不再缩进，采用取消缩进的形式进行展示代码。

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

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

#define ACK_TIMER 280                    /*ACK帧超时重传时间*/
#define DATA_TIMER  4100                 /*数据帧超时重传时间*/
#define MAX_SEQ 31                       /*帧的序号空间，应当是2^n-1,序号上限
    */
#define NR_BUFS ((MAX_SEQ+1)/2)          /*窗口大小上限*/

#define inc(k) if(k < MAX_SEQ)k++;else k=0  /*计算k+1*/


struct FRAME {
unsigned char kind;                 /*帧的种类，ACK，NAK，DATA*/
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*/
};

struct ACK_FRAME
{
unsigned char kind;                 /*帧的种类*/
unsigned char ack;                  /*序号*/

unsigned int padding;               /*填充字段*/
};

struct NAK_FRAME
{
unsigned char kind;                 /*帧的种类*/
unsigned char ack;                  /*序号*/

unsigned int padding;               /*填充字段*/
};

```

```
40 static unsigned char next_frame_to_send = 0, frame_expected = 0,
       ack_expected =
41 0;
42 /*************************************************************
43 *   next_frame_to_send:将要发送的帧序号，发送方窗口的上界；
44 *   fram_expected:期望收到的帧序号，接收方窗口的下界；
45 *   ack_expected:期望收到的ack帧序号，发送方窗口的下界
46 *
47 *************************************************************/
48
49 static unsigned char out_buffer[NR_BUFS][PKT_LEN], in_buffer[NR_BUFS][
       PKT_LEN],
50 nbuffered;
51 /*************************************************************
52 *对应输出缓存，输入缓存，以及目前输出缓存的个数
53 *
54 *************************************************************/
55
56 static unsigned char too_far = NR_BUFS;      /*接收方窗口的上界*/
57 static int phl_ready = 0;                    /*物理层是否ready*/
58 bool arrived[NR_BUFS];                       /*接收方输入缓存的bit map*/
59 bool no_nak = true;                              /*是否发送了NAK，true则不再
       重复发
60 送*/
61 //bool no_nak_flags[NR_BUFS];                    /*接收方窗口是否发送了NAK*/
62
63 static bool between(int a, int b, int c)     /*判断b是否在a和c组成的窗口之
       间，如果
64 是返回true，否则返回false*/
65 {
66 return((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a))
       ;
67 }
68
69 static void put_frame(unsigned char* frame, int len)         /*CRC校验*/
70 {
71 *(unsigned int*)(frame + len) = crc32(frame, len);       /*将frame进行CRC校验
       后，
72 附校验码发至物理层*/
73 send_frame(frame, len + 4);
74 phl_ready = 0;                                           /*将数据发至物理层后
       将物理
75 层置为忙碌状态*/
76 }
77
```

```
78  static void send_data_frame(unsigned char frame_nr)
79  {
80  struct FRAME s;
81
82  s.kind = FRAME_DATA;
83  s.seq = frame_nr;
84  s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
85  memcpy(s.data, out_buffer[frame_nr % NR_BUFS], PKT_LEN);     /*复制out_buffer
        中的
86  %缓存到data中*/
87
88  dbg_frame("Send DATA %d %d, ID %d\n", s.seq, s.ack, *(short*)s.data);
89
90  put_frame((unsigned char*)& s, 3 + PKT_LEN);         /*将帧发送至物理层*/
91  start_timer(frame_nr % NR_BUFS, DATA_TIMER);
92  stop_ack_timer();
93  }
94
95  static void send_ack_frame(void)
96  {
97  struct ACK_FRAME s;
98
99  s.kind = FRAME_ACK;
100 s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
101
102 dbg_frame("Send ACK  %d\n", s.ack);
103
104 put_frame((unsigned char*)& s, 2);
105 stop_ack_timer();
106 }
107
108 static void send_nak_frame(void)
109 {
110 struct NAK_FRAME s;
111
112 s.kind = FRAME_NAK;
113 s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
114
115 no_nak = false;
116
117 dbg_frame("Send NAK  %d\n", s.ack);
118
119 put_frame((unsigned char*)& s, 2);
120 stop_ack_timer();
121 }
```

```
122
123
124 int main(int argc, char** argv)
125 {
126 nbuffered = 0;                  /*初始时没有输出缓存*/
127 int i;
128 for (i = 0; i < NR_BUFS; i++)
129 {
130 arrived[i] = false;       /*初始时没有输入帧被缓存*/
131 }
132 int event, oldest_frame;          /*定义处理的事件*/
133 struct FRAME f;
134 int len = 0;                    /*data的长度*/
135
136 /*以上为数据初始化*/
137
138 protocol_init(argc, argv);
139 lprintf("Designed□and□implemented□by□Wang□Peng.\n□build:□" __DATE__"
140 "__TIME__"\n");
141
142 /*协议初始化*/
143
144 disable_network_layer();
145
146 for (;;) {
147 event = wait_for_event(&oldest_frame);
148
149 switch (event) {                        /*对事件进行处理*/
150 case NETWORK_LAYER_READY:               /*网络层ready*/
151 get_packet(out_buffer[next_frame_to_send % NR_BUFS]);    /*从网络层获取一帧放
    入输出
152 %缓存*/
153 nbuffered++;
154 send_data_frame(next_frame_to_send);
155 inc(next_frame_to_send);
156 break;
157
158 case PHYSICAL_LAYER_READY:              /*物理层ready*/
159 phl_ready = 1;
160 break;
161
162 case FRAME_RECEIVED:                    /*物理层收到一帧*/
163 len = recv_frame((unsigned char*)& f, sizeof f);
164 if (len < 5 || crc32((unsigned char*)& f, len) != 0) {
165 dbg_event("****□Receiver□Error,□Bad□CRC□Checksum\n");
```

```
166 if (no_nak)
167 send_nak_frame();
168 break;
169 }
170
171 if (f.kind == FRAME_ACK)                    /*如果收到ACK帧，统一处理*/
172 dbg_frame("Recv□ACK□□%d\n", f.ack);
173
174 if (f.kind == FRAME_DATA) {
175 if ((f.seq != frame_expected) && no_nak)
176 {
177 send_nak_frame();
178 }
179 else
180 start_ack_timer(ACK_TIMEOUT);
181 if (between(frame_expected, f.seq, too_far) && (arrived[f.seq % NR_BUFS] ==
182 %false))
183 {
184 dbg_frame("Recv□DATA□%d□%d,ID□%d□\n", f.seq, f.ack, *(short*)f.data);
185 arrived[f.seq % NR_BUFS] = true;
186 memcpy(in_buffer[f.seq % NR_BUFS], f.data, PKT_LEN);
187 while (arrived[frame_expected % NR_BUFS])
188 {
189 put_packet(in_buffer[frame_expected % NR_BUFS], len - 7);
190 no_nak = true;
191 arrived[frame_expected % NR_BUFS] = false;
192 inc(frame_expected);
193 inc(too_far);
194 start_ack_timer(ACK_TIMEOUT);
195 }
196 }
197 }
198 if ((f.kind == FRAME_NAK) && between(ack_expected, (f.ack + 1) % (MAX_SEQ +
    1),
199 %next_frame_to_send))
200 {
201 send_data_frame((f.ack + 1) % (MAX_SEQ + 1));
202 dbg_frame("Recv□NAK□with□ACK□%d\n", f.ack);
203 }
204 while (between(ack_expected, f.ack, next_frame_to_send))
205 {
206 nbuffered--;
207 stop_timer(ack_expected % NR_BUFS);
208 inc(ack_expected);
209 }
```

```
210 break;
211
212 case ACK_TIMEOUT:
213 dbg_event("----□ACK□%d□timeout\n", oldest_frame);
214 send_ack_frame();
215 break;
216
217 case DATA_TIMEOUT:
218 dbg_event("---□DATA□%d□timeout\n", oldest_frame);
219 if (!between(ack_expected, oldest_frame, next_frame_to_send))
220 oldest_frame += NR_BUFS;
221 send_data_frame(oldest_frame);
222 break;
223 }
224
225
226 if (nbuffered < NR_BUFS && phl_ready)
227 enable_network_layer();
228 else
229 disable_network_layer();
230 }
231 }
```