“NAK流”选择重传协议的设计&实现

一、任务要求

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。相关数据如下：

1. 信道模型为 8000bps 全双工卫星信道；
2. 信道传播时延 270 毫秒；
3. 信道误码率为 10^{-5}；
4. 信道提供字节流传输服务，网络层分组长度固定为 256 字节。

设计目标：

1. 实现有误码信道环境下的无差错传输;
2. 充分利用传输信道的带宽

实验环境：Microsoft Visual Stdio Community 2015

二、“W = 4”选择重传协议

1. 帧格式设计

我的数据帧格式如下图所示：

类型	ack	序号	数据	CRC
2bits	3bits	3bits	256*8bits	32bits

FIGURE 1. 帧格式设计图例

类型：表明 frame 的类型，有 nak, ack, data 三类，所以需要2bits 来表示。

ack ： 表明发送方已经收到的帧中最大的序号，用于告知对方此时发送者的接受窗口的下界。（考虑到 ack 帧，将此字段放在序号之前）

序号：表明 frame 的序号，值为0~2^W–1的整数。

数据：即发送方从网络层下载下来的分组，固定大小256B。

CRC：该帧的CRC 校验码，用于检错。

其中窗口大小 W 的计算过程如下：

$$\begin{cases} t_p = 270ms & ① \\ t_f = \dfrac{260 * 8 + 2 + 2 * log_2(W * 2)}{8000} ms & ② \\ \dfrac{W * t_f}{2 * t_f + 2 * t_p} \geqslant 1 & ③ \end{cases}$$

解得：$W \geqslant 4.06$，考虑节约帧大小，**在协议一中取 W = 4。**

其中，式2考虑的是一帧的传输时间，分子的每一项分别代表了数据和 CRC、类型字段、ack 和 seq 字段所用的比特数

式3考虑的是信道的无差错利用率应该要大于1，显然这里的窗口大小会比实际中需要的窗口数小一点，因为必然有一些数据因为传输错误而需要重传。

注：我的帧格式中，将type， seq， ack三个字段整合为一个字节。

2. 超时参数设计

确定了帧格式，我们下面就来计算两个时间参数（超参数）： 发送帧 ack 的等待时间，ack 的忍耐时间：

（1）由于使用了 piggybacking 技术，发送方从发出一个帧，到接收方发出一个带 ack 的数据帧，总共需要的时间是 $2 * (t_f + t_p)$。再考虑接收方的处理时间和排队时间，我初始取 T_{wait_ack} = 1070ms。

（2）为了避免发送方因等不到 ack 帧而重发了一个成功发送的数据帧，应该保证接收方能够及时地发送 ack 帧。如上图所示，即 T_1 应当小于发送方超时时间 T_{wait_ack} 。根据 T_2 与 T_1 的关系，可得：

$$T_{wait\_data} \leqslant T_{wait\_ack} - 2t_p - 2t_f$$

这里的参数设计仅为“W = 4”选择重传协议使用，后面有更合理的设计。

3. 协议一的效率

在实际运行中，W=4的窗口协议效率却并不尽如人意，具体数据如下：

Station A	Station B
Build: May 10 2016 11:32:40 4.476 .... 7 packets received, 4454 bps, 55.67%, Err 1 (3.9e-005) 6.582 .... 13 packets received, 5000 bps, 62.50%, Err 1 (2.4e-005) 8.954 .... 20 packets received, 5322 bps, 66.52%, Err 1 (1.6e-005) 11.075 .... 27 packets received, 5632 bps, 70.40%, Err 1 (1.3e-005) 13.166 .... 33 packets received, 5675 bps, 70.94%, Err 1 (1.1e-005) 15.272 .... 39 packets received, 5699 bps, 71.24%, Err 1 (8.9e-006) 17.643 .... 46 packets received, 5749 bps, 71.87%, Err 1 (7.7e-006) 20.014 .... 53 packets received, 5787 bps, 72.34%, Err 1 (6.7e-006) 22.401 .... 60 packets received, 5812 bps, 72.64%, Err 1 (5.9e-006) 24.491 .... 67 packets received, 5906 bps, 73.82%, Err 1 (5.4e-006) 26.613 .... 69 packets received, 5573 bps, 69.66%, Err 2 (9.9e-006) 28.734 .... 76 packets received, 5665 bps, 70.81%, Err 2 (9.3e-006) 30.840 .... 82 packets received, 5677 bps, 70.96%, Err 2 (8.6e-006) 32.946 .... 88 packets received, 5687 bps, **71.09%**, Err 2 (8.0e-006) … 38.484 TCP Disconnected.	Build: May 10 2016 11:32:40 3.088 .... 8 packets received, 6213 bps, 77.66%, Err 0 (0.0e+000) 5.194 .... 14 packets received, 6045 bps, 75.56%, Err 1 (2.8e-005) 7.316 .... 18 packets received, 5370 bps, 67.12%, Err 1 (1.9e-005) 9.687 .... 25 packets received, 5544 bps, 69.29%, Err 1 (1.4e-005) 11.777 .... 32 packets received, 5786 bps, 72.33%, Err 1 (1.1e-005) 13.883 .... 38 packets received, 5794 bps, 72.42%, Err 1 (9.5e-006) 16.005 .... 44 packets received, 5793 bps, 72.42%, Err 1 (8.2e-006) 18.376 .... 51 packets received, 5827 bps, 72.84%, Err 2 (1.4e-005) 20.732 .... 58 packets received, 5857 bps, 73.21%, Err 2 (1.2e-005) 23.118 .... 65 packets received, 5873 bps, 73.41%, Err 2 (1.1e-005) 25.209 .... 72 packets received, 5956 bps, 74.45%, Err 2 (1.0e-005) 27.845 .... 77 packets received, 5757 bps, 71.96%, Err 2 (9.2e-006) 30.466 .... 82 packets received, 5595 bps, 69.94%, Err 4 (1.7e-005) 32.588 .... 88 packets received, 5608 bps, **70.10%**, Err 4 (1.6e-005) … 38.484 TCP Disconnected.

CHART 1. 协议一的效率

4. 协议一存在的问题

协议一的一切参数都有理论推导，两站的效率却并不尽如人意，其问题出在哪里?

(0) **“W = 4”节省不必要**。节省下来的一字节很难对信道利用率有较大的提升，反倒制约了窗口的扩展。

（1）**T_p的实际值要大于270ms。** 在实验中我发现信道延迟似乎并270ms 来的大，我就设计了以下方法来检测信道的实际传输延迟 T_{real_p}：在接收方接受到第一帧的时候，此时接收方物理层发送队列为空，令其立即发送一个长度为5Byte的ack帧（通过设计时器值为0），记录发送方收到的时间，因为此时发送方物理层的接受队列也为空，所以总时间减去其中帧的发送时间，即为信道真实传输延迟：

Station A	Station B
Build: May 10 2016 11:17:36 0.195 Send DATA 0 7 … **0.991 Recv ACK 0** 1.194 Send DATA 4 7, ID 10004 …	Build: May 10 2016 11:17:36 0.710 Recv DATA 0 7, ID 10000 0.710 ---- ACK for DATA timeout **0.710 Send ACK 0** 0.975 Recv DATA 1 7, ID 10001 …

CHART 2. 测量真实信道延迟

由此可知，**T_p 的真实值应该是在275ms左右，**大于270ms 的原因可能在于物理层接收字节流、同步帧等工作需要时间。

（2）**由于ack的捎带传输，实际中 ack 帧并不能够立即发出。**即上述推导中的式3忽略了 ack 常常需要忍耐一段时间然后才发送（单独发出或捎带），正确的公式应该为：

$$\frac{W * t_f}{2 * t_f + 2 * t_p + T_{wait\_data}} \geqslant 1$$

三、“W = 32”选择重传协议

1. 帧格式设计

2bits 6bits 6bits 256*8bits 32bits

FIGURE 3.版本二格式设计帧

协议二选择了二字节的控制字段。出于不浪费字段的原则，我使用了6bit 的编号空间，意味着版本二的窗口大小为32。

2. 超时参数设计

决定窗口大小为32后，**为了能够尽量地使用信道的带宽，我们考虑在满足在不耗尽发送方的发送窗口的前提下**，最大化我们接收方发送ack的忍耐时间 T_{wait_data}，这需要满足以下两个方程：

$$\begin{cases} T_{wait\_data} \leqslant T_{wait\_ack} - 2t_p - 2t_f \\ W * t_f + W - 1 \geqslant T_{wait\_ack} + 2 * t_f + 2 * t_p \end{cases}$$

其中第一个式子保证了不会因为接收方忍耐 ack 而导致发送方重传；

当接收方长时间没有数据可以发送时，第二个式子避免了因为接收方对ack的忍耐而耗尽了发送方的发送窗口，导致信道被闲置的情况（**W−1是考虑到物理层帧间隔1ms**）。

结合两个式子，我们保守地取：

$$\begin{cases} T_{wait\_data} = 7300ms \\ T_{wait\_ack} = 8400ms \end{cases}$$

3. NAK帧最小发送时间间隔

上述两个式子保证了**在任何的流量条件下**，少发ack帧的同时保证避免了不必要的超时重发。但是当帧在传输出错的时候，上述条件并不能够保证协议的高效，甚至会因为最大化了 T_{wait_ack} 而反而拖延了真正需要重发的帧的发送时间。基于上述考虑，我们考虑通过发送 NAK 帧来告诉发送方，某帧发送失败并尽快重传。与原本只发送一次 nak 帧不同，我的做法如下：

我们为 NAK 帧单独分配一个计时器，**当接收方接收到坏帧或是错误的帧，我们就可以每隔T_{min_nak}再次发送一个 NAK 来催促发送方尽快发送该帧。**那么我们要如何设计 T_{min_nak} 呢?

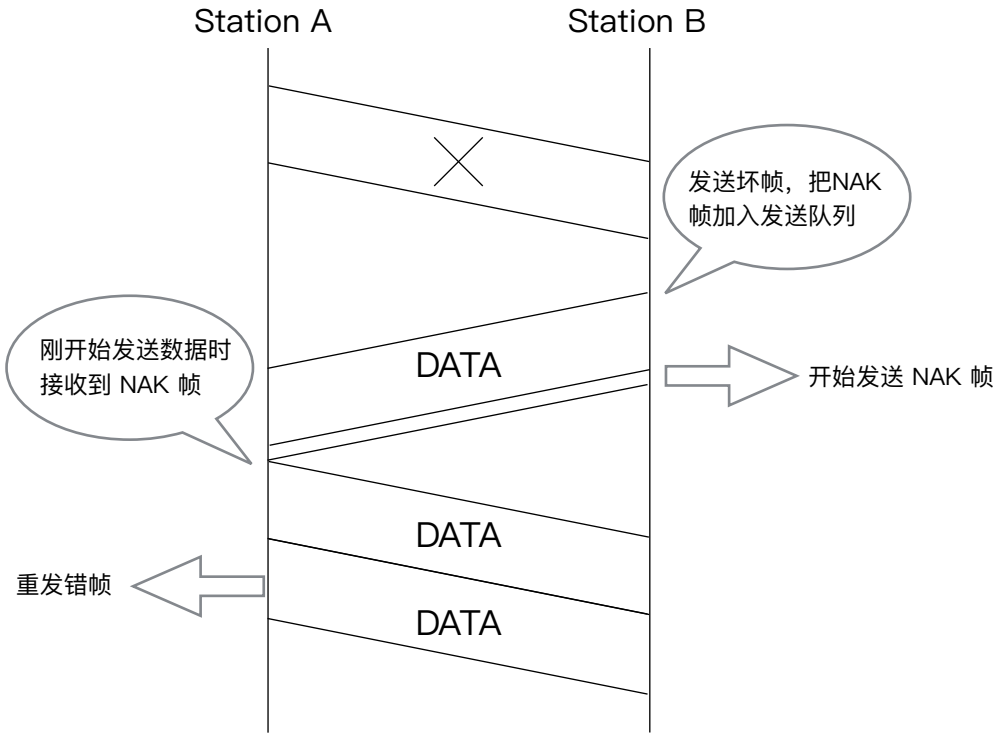

FIGURE 3. NAK帧发送

由上图可知，**T_{min_nak} 不能够太短，以至于发送方无意义地再发送一个包**，故 T_{min_nak} 应该保证：(t_{nf} 表示发送 nak 帧时间)

$$T_{min\_nak} \geqslant 3t_f + t_{nf} + 2t_p$$

在协议二中，我取 T_{min_nak} 为1400ms。

4. 协议二的效率分析

就像我们之前考虑的，协议二在低误码率的情况下可以收获很好的效率，如在误码率为10^{-5}的flood 模式下，**可以收获95.3%信道利用率**。但是在误码率为10^{-4}的时候效率就只能够达到60%，反倒不如大多数普通的选择重传协议，距离理论最大信道利用率81.09%更是相去甚远。

经过分析原因在于：**无意义的重传过多。**考虑下面这个情况：接收方接收窗口为0~31，1~10的数据包早早收到，但是序号为0的包接连发送错误，所以接收方回的 ack 只能为63。经过一段时间后，0包终于姗姗来迟，接收方又花了 $t_{f+}t_p$ 的时间终于把ack 10的消息告诉发送方，此时发送方见 ack 帧迟迟不来，编号为1~10的计时器超时，重发1~10的帧，浪费巨大带宽。上述情况虽有夸张，但是确实真实存在的。**其根本原因在于，接受窗口里面悬而未决的帧过多，导致实际已经成功接收的包的 ack 信息无法到达发送方。**

5. 解决方法

I. **一次发送多个 NAK 帧。**通过增加 NAK 帧的发送次数，使发送方尽快重发错帧，保证接受接受窗口中悬而未决的窗口数目在一个较小的水平。（已实现）

II. **弃用累计确认，对每个成功接收的帧单独发送 ACK。**这要求为每个接收窗口设置一个 start_ack_timer，其功能与原先类似，即为了避免因捎带确认而导致的发送方超时重传。**理论上该方法能根本性地解决了高误码率信道的传输问题**，但是由于具体实现问题较多，老师提供的函数接口也不能很好地支持，故放弃。（未实现）

四、“NAK流”选择重传协议

1. 新NAK帧格式

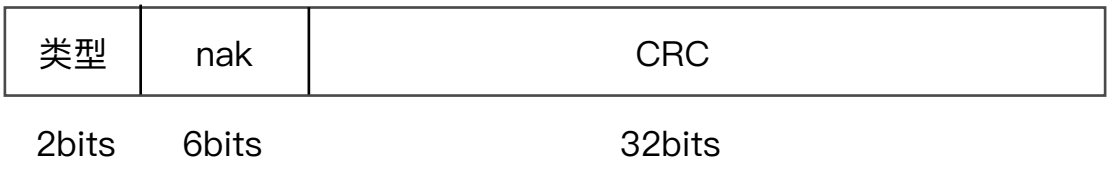

FIGURE 4. NAK帧格式

新的 NAK 帧弃用了原来的 ack 字段，此处 nak 表示的是对应序号的数据包发送失败，催促发送方尽快重发对应帧。再次，此时的NAK帧不再具有提供ack的功能。

2. “NAK流”如何发送

每当CRC校验出错，以及收到一帧不是接收方最想要的帧时，表明存在帧传输出错，这个时候扫描接受窗口，对于每一个传输失败的帧单独发一个NAK帧。那么哪些帧需要发送 NAK 帧呢?

考虑接受窗口中接收到的最新的帧编号为newest_frame，则传输失败的帧即为从frame_expected到newest_frame中，还没有成功接收的帧。因为发送方是顺序发送，所以他们没有收到的唯一解释就是发送失败了。对应代码如下：

```
static void send_nak(void)
{
      int i, newest_frame = frame_expected + 1, count = 0;
      //newest_frame表示成功接收到的最新的帧

      //Find newest_frame
      for (i = frame_expected + 1; between(frame_expected, i,
too_far);)
      {
            if (arrived[i % NR_BUFS] == 1) newest_frame = i + 1;
            if (i < MAX_SEQ) i++;
            else i = 0;
      }

      //send NAKs
      for (i = frame_expected; between(frame_expected, i,
newest_frame);)
      {
```

```
            if (arrived[i % NR_BUFS] == 0)
            {
                  send_nak_frame(i);
                  count++;
            }
            if (i < MAX_SEQ) i++;
            else i = 0;
      }

      //标记发送过nak，并开启对应计时器
      nak_flag = true;
      start_timer(NAK_TIMER, MIN_NAK_WAIT + (count - 1) *
MAX_FRAME_SIZE);

      //我实现的debug输出函数，输出当前接受窗口占用状态
      print_recv_windows(frame_expected, newest_frame, count);
}
```

CHART 2. 发送”NAK流“函数实现

4. 协议性能分析

A. 理论最优信道利用率

设当前信道的误码率为e，则对于每一个长度为262Byte的的包，其一次成功发送的概率为：

$$P = (1-e)^{262*8} = (1-e)^{2096}$$

设一帧需要传输的次数为随机变量X，则对其取期望可得：

$$E[X] = \sum_{i=1}^{\infty} i * (1-P)^{i-1} P = \frac{1}{P}$$

则最终理论最大信道利用率为：

$$\eta = \frac{256}{262 * X}$$

由上述算式可得当误码率为10^{-5}与10^{-4}的理论最大信道利用率分别为95.68%与79.23%。

B. 协议性能的测试结果与分析

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	1901	54.16	97.30	
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”	2511	52.59	95.17	
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	2249	97.34	97.34	
4	datalink af datalink bf	站点A/B的分组层都洪水式产生分组	2209	95.58	95.52	
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}	2129	80.32	80.15	
6	datalink af –ber 2e-4 datalink bf –ber 2e-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}	373	62.19	62.11	测试兼容性
7	datalink af –ber 4e-4 datalink bf –ber 4e-4	线路误码率设为10-4，进行长时间以测试协议容错性	3712	34.21	33.84	测试兼容性

CHART 3. 性能测试记录表

通过上表，我们可以得出该协议的以下优点：

A. **高效。**通过及时催促发送方重发，从而避免了不必要的重传，该协议高误码率10^{-4}的信道中收获了80.32%的超高信道利用率。考虑在该误码率的情况下的理想信道利用率为79.23%，说明协议之高效。

B. **自适应性。**通过动态调整NAK计时器的长度，使得本协议对于误码率较低的信道，不需要更改任何参数，也能够保证高信道利用率。测试结果显示，本协议可以在误码率为10^{-5}的信道中获得95.32%的信道利用率，而理想信道利用率95.68%，体现协议适应能力之强。

C. **能够处理不同数据发送模式的信道。**通过最大化T_{data_wait}来尽可能使用捎带传输，使得协议无论是在“flood模式”下还是在“发送/停发模式”下，都能够收获很好的效率。

D. **稳定**。从序号7的测试中我们可以看出，尽管在$4*10^{-4}$误码率的情况下，协议仍能够**稳定地运行超过一个小时，**足可说明程序的稳定。而其他容错性较差的协议，很容易产生死锁。

然而在测试中我也发现了一点不足，**也就是当误码率高于$3.0*10^{-4}$的时候信道利用率与理论最大值相比开始有所下滑**：当误码率为$4.0*10^{-4}$的时候，协议效率为34.21%，而理论最大值为42.25%。更为严重的是，当误码率大于$1*10^{-3}$的时候，程序基本瘫痪，效率徘徊在8%左右，那么问题出在哪里呢?

因为误码率过高，此时接受窗口中“问题帧”大量堆积，使得真正成功接收到的帧无法及时发出 ack，最终导致发送方经常性的超时，发生不必要的重传。其本根原因在于此时**发送成功一帧才是小概率事件。**考虑$4.0*10^{-4}$的误码率，每一包成功发送的概率为43.23%。此时**应该弃用 ack 的累计确认机制**，因为能够 ack 的帧太少了！而应该对于每一个成功接收到的帧都发送一个 ack 帧，从理论上就可以解决在超高误码率情况下的无意义重传问题。

4. 使用“NAK流”的优缺点

A. **具备自适应不同误码率的信道的能力。**在函数实现中，我累计了此时接受窗口发生传输错误的的个数，并乘以每个数据包的发送时延，据此开始NAK计时器。通过这一点，“NAK流”选择重传协议可以很好地适应不同误码率下的信道，自动的调节计时器的长度，从而最大化信道利用率。

B. **NAK帧不再有提供ACK的能力。**在某种程度上，这一点使得一帧 NAK 所携带的信息量有所减少。但是我们通过最大化T_{data_wait}使得尽量的去使用捎带确认的方法来传输ACK。就算接收方在很长时间内（大于7300ms）没有数据要发，我们也只需要单独发一个ACK帧就可以解决问题，而这仅耗费了总带宽的0.06%，考虑其带来的好处，这一点浪费可以忽略不计。

六、程序设计与实现

1. 数据结构

```
#define MAX_ACK_WAIT      10000      //ack 帧最长等待时间
#define MAX_DATA_WAIT     7300       //piggybacking 最长等待时间
#define MAX_FRAME_SIZE    262        //最大包的大小
#define MIN_FRAME_SIZE    5          //最小包的大小
#define MIN_NAK_WAIT      1400       //NAK最少要等多久再发一个

#define NAK_TIMER         63         //NAK计时器编号

#define FRAME_DATA   0
#define FRAME_ACK    1
#define FRAME_NAK    2

#define KIND_LEN     2               //类型字段长度
#define ACK_LEN      6               //ACK字段长度
#define SEQ_LEN      6               //序号字段长度

#define MAX_SEQ      63
#define NR_BUFS      32

//control[0]前二位表示kind，后六位表示ack/nak，control[1]表示seq
struct FRAME {
     unsigned char control[2];
     unsigned char data[PKT_LEN];
     unsigned int crc;
};

//接受窗口与发送窗口
static unsigned char next_frame_to_send = 0;
static unsigned char frame_expected = 0;
static unsigned char ack_expected = 0;
static unsigned char too_far = NR_BUFS;

//接受缓冲区与发送缓冲区
static unsigned char out_buffer[NR_BUFS][PKT_LEN];
static unsigned char in_buffer[NR_BUFS][PKT_LEN];
static unsigned char nbuffered;

//物理层是否空闲
static bool phl_ready = 0;

//某帧是否已经成功接收
bool arrived[NR_BUFS];

//是否已经发过NAK流
bool nak_flag = false;
```

2. 模块结构示意图

A. 总的程序设计逻辑

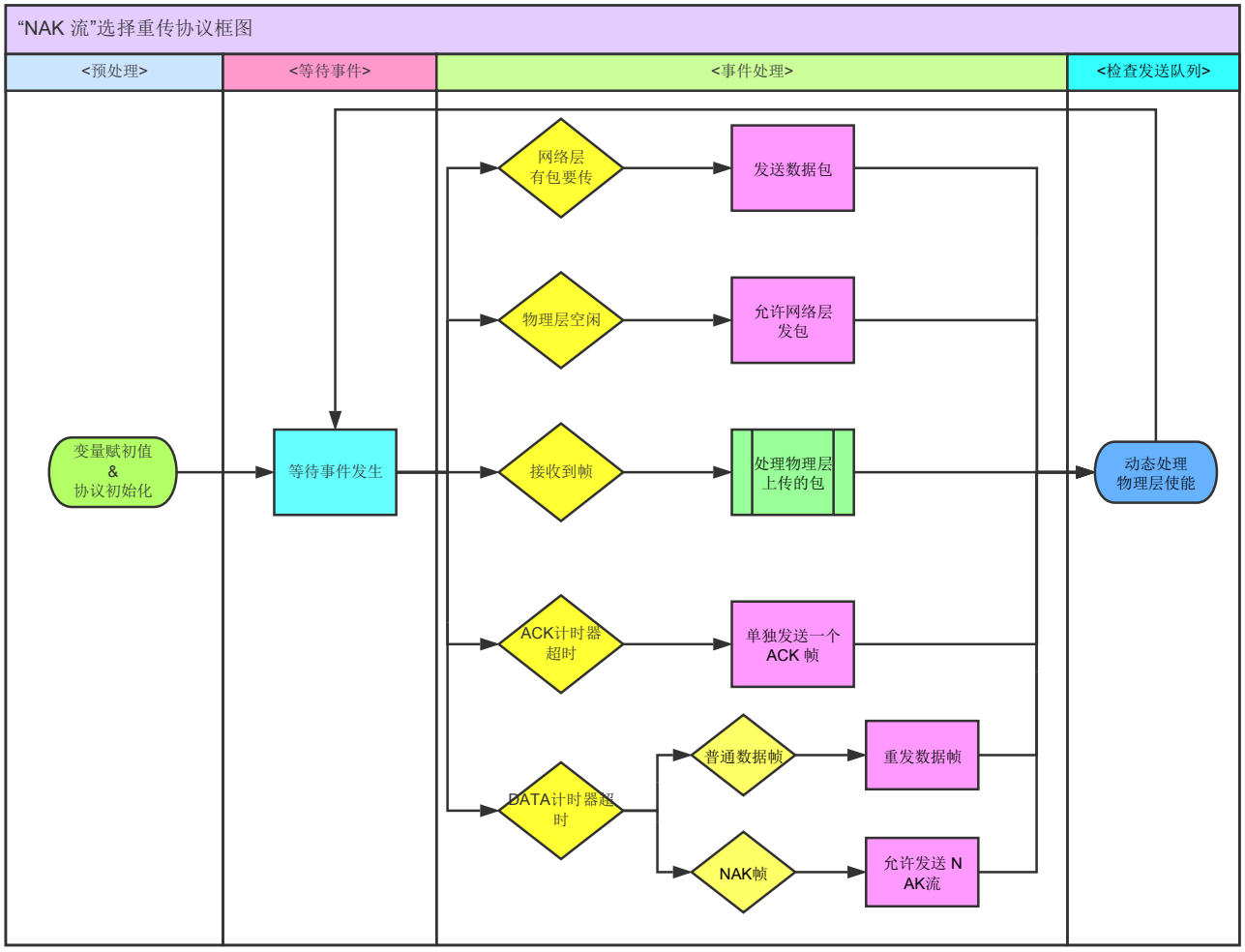

FIGURE 5. “NAK流”选择重传协议框图

B. 接收一个包的处理逻辑

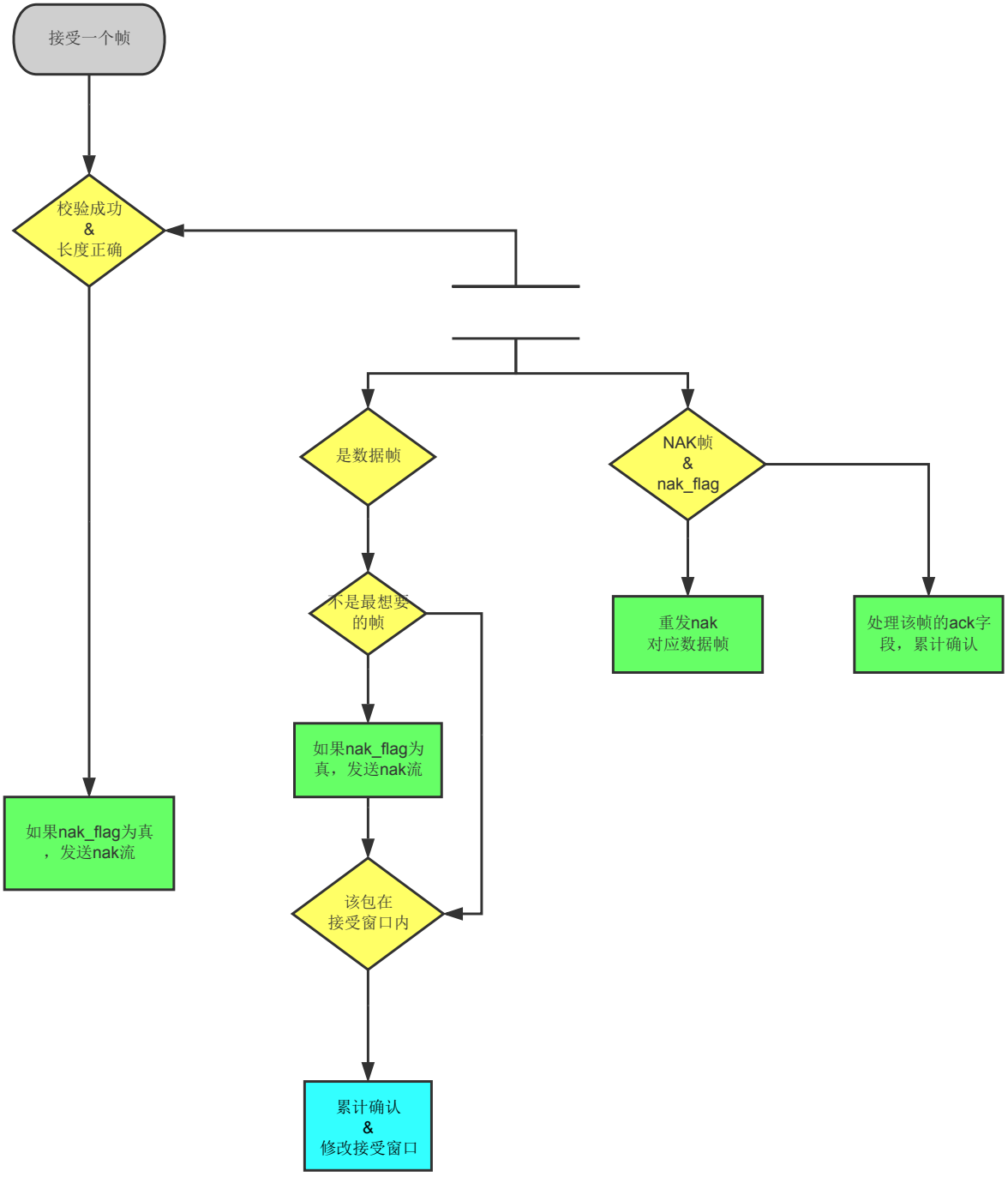

FIGURE 6.接受包的处理逻辑

七、研究与探索

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位以上长度的突发错误。

可以将错误发生的过程视为泊松分布，则在一帧中，出现错误个数的期望是：

$$\lambda = 2096 * 10^{-5} = 0.02096$$

所以一帧中出现33个以上错误的概率为：

$$P\{33 \leqslant X \leqslant 2096\} = \sum_{k=33}^{2096} \frac{\lambda^k}{k!} \times e^{-\lambda} = 4.55398 \times 10^{-93}$$

而实际上，**并不是每一个长度大于32的突发错误都能能够逃过检测，只有突发错误是 G(x)的倍数时才会当成正确帧被接受。**所以真正无法检测的概率为：

$$\begin{aligned} P\{err\ but\ accept\} &= 2^{(-32)} \times P\{33 \leqslant X \leqslant 2096\} \\ &= 1.06 \times 10^{-102} \end{aligned}$$

故平均传送9.932*10^{99}个帧才可以出现一个出错了但却检测不出的坏帧。

而客户每天实际可以发送的帧数目：

$$\frac{95.7\% \times 8000 \times 60 \times 60 \times 24}{256 \times 8 \times 2} = 1.57795 \times 10^7$$

所以，发生一次分组误码事件，需要5.97*10^{92}年。

可见CRC的检验效果之好。

当然有需求的话，可以用更长的CRC的校验多项式来获得更大的校验性能。而代价是，需要更多的比特传输校验和，使信道利用率下降。

2. CRC的计算方法

显然二者是等价的，因为对于给定被除数与除数，“模2”除法的结果是确定的，所以可以通过查表来加快计算过程。

3. 程序设计的问题

get_ms()可以很容易的通过调用<time.h>里面的 clock()来实现。

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。通过指针调用对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。

ack计时器之所以不能够被刷新，是因为我们考虑的是**第一个**我们成功接收，但是却没能及时发出ack的帧。所以ack计时器不能够被覆盖。

而数据计时器是单独考虑每隔发出去的帧，其ack的帧是否及时到达，如果没有到达发送方要重发**计时器对应的数据帧，**所以计时器需要与数据帧一一对应。而刚发送一个新帧时，原来计时器的残余时间毫无意义，自然是要清零重新计算。

4. 软件测试的问题

- **无误码信道**：测试协议的最大吞吐能力。效率与帧格式的设计有关，若控制字段较短，则在此信道上的性能较好
- **发送/停发模式**：测试协议在不同流量模式下的性能。这个模式综合考察了协议对ack帧的发送时机，对计时器超时参数的设计。
- **洪水模式**：测试协议的抗压能力与最大输出能力。这个模式综合考察了协议对ack帧的发送时机、nak帧的设计等

- **高误码率信道**：测试协议在高误码率情况下的性能。这个模式考察了协议的重传机制与对 ack、nak 帧的使用。

5. 对等协议实体之间的流量控制

- 通过控制网络层的使能函数enable_network_layer()/disable_network_layer()

  两个函数来控制从网络层发来的数据包。
- 接收方可以通过把接受窗口大小置为0来拒绝发送方发来的所有包。

八、实验心得与总结

实话实说，我在这次实验作业上大概花费了10~15个小时的时间，其主要花费在于：

– 实验环境的配置

– 前后实现了三个版本的协议，即 “W=4”,“W=32”,与“NAK流”的选择重传协议

因为我用的是 Mac 系统，首选自然是用 Linux 版本的实验平台。但是因为老师给的是32位的版本，我前后一共尝试了：尝试把 mac 系统置为32位，安装32位的通用 gcc 编译器。但是最后都出了一些很奇怪的情况，考虑到时间所剩不多，最后我装了虚拟机在 Win7下完成了作业。前前后后一共花了6个小时左右吧。

我写程序大概花了4个小时，程序写完通过 dbg 函数看协议逻辑，很容易就解决了bug的问题。其余时间大多都花在了参数的计算，协议效率的改进，都是一些理论的分析与计算吧。

一开始协议效率并不好，我先是吧窗口大小从四改为了三十二，大大提高了算法效率。但是在高误码率的情况下效率却与理论最高值相去甚远，此时摆在我眼前的方法有：

- 继续扩大窗口
- 去掉 ack，nak 的 CRC 帧

分析后我明白第一个方法并不是问题的关键，而第二个方法将会导致协议失效，所以我最后自己发明了一个“NAK流”的方法。**看到这么高的信道利用率，心里还是颇为自得的哈哈！**