《计算机网络》实验报告

<table>
<tr><td>实验名称</td><td colspan="2">滑动窗口协议的设计与实现</td><td>学 院</td><td>计算机学院</td><td>指导教师</td><td>程莉</td></tr>
<tr><td>班 级</td><td>班内序号</td><td colspan="2">学 号</td><td>学生姓名</td><td colspan="2">成绩</td></tr>
<tr><td>2022211305</td><td></td><td colspan="2">2022211683</td><td>张晨阳</td><td colspan="2"></td></tr>
<tr><td>2022211305</td><td></td><td colspan="2">2022211124</td><td>梁维熙</td><td colspan="2"></td></tr>
<tr><td>2022211305</td><td></td><td colspan="2">2022211130</td><td>金建名</td><td colspan="2"></td></tr>
<tr><td>实验内容</td><td colspan="6">本实验利用数据链路层原理实现一个滑动窗口协议，在仿真环境下实现有噪音信道两站点间的无差错的双工通信，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。通过本实验达到提升编程能力和实践动手能力的目的，切身体会协议设计和调试中的难题并给出解决方案，将所学得的有关数据链路层和协议设计的理论知识应用于实践，增强对实际系统协议分层体系结构和协议软件设计的理解认知，并完成协议设计。
我小组在本实验中选用了选择重传协议（SR）进行实现，在实验过程中分析 CRC 技术以及通过工作参数的调整提升信道利用率并实现可靠传输。
在本实验中，向张晨阳学负责理论设计与协议设计、程序编码与调试、协议工作参数优化等工作；金建名同学负责理论设计、后期测试、实验结果分析和问题探索等工作；梁维熙同学负责实验相关问题总结和实验报告的撰写等工作，具体工作内容详见报告。</td></tr>
<tr><td>学生实验报告
(附页)</td><td colspan="6"></td></tr>
<tr><td>实验成绩评定</td><td colspan="6">评语:

成绩:

助教签名:
年 月 日</td></tr>
</table>

注：评语要体现每个学生的工作情况，可以加页。

目录

1. 实验内容及实验环境描述 1
   1.1. 实验内容 1
   1.2. 实验环境 1
2. 软件设计 2
   2.1. 帧结构设计 2
   2.2. 数据结构 2
   2.3. 模块结构 3
   2.4. 算法流程 5
3. 实验结果分析 6
   3.1. 程序正确性和健壮性分析 6
   3.2. 协议参数的选取 6
   3.3. 理论分析 11
   3.4. 实验结果分析 12
   3.5. 当前程序存在的问题 12
4. 研究和探索的问题 13
   4.1. CRC 校验能力 13
   4.2. CRC 校验和的计算方法 14
   4.3. 程序设计方面的问题 15
   4.4. 软件测试方面的问题 16
   4.5. 对等协议实体之间的流量控制 16
   4.6. 与标准协议的对比 17
5. 实验总结与心得体会 18
   5.1. 实验总时长 18
   5.2. 实验中所遇到的问题 18
   5.3. 实验心得总结 19
6. 源程序文件 19

1. 实验内容及实验环境描述

1.1. 实验内容

利用所学知识，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、两站点间无差别双工通信。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原理。滑动窗口机制的两个主要目标：

（1）实现有噪音环境下的无差错传输

（2）充分利用信道的带宽

在程序编译完成后，要求能够完成：

（1）在无误码环境下运行测试

（2）在有误码信道环境下完成无差错传输

（3）稳定运行 20 分钟以上，效率不能太低

1.2. 实验环境

windowsSDKversion: 10.0.19041.685

Visual Studio 2022

cStandard: c17

2. 软件设计

2.1. 帧结构设计

参考教材中的 Protocol 6 以及指导书中的要求，使用了如下的帧格式：

DATA Frame

```
+=========+=========+========+============+========+
|  KIND(1)  |  ACK(1)  |  SEQ(1)  |  DATA(256)  |  CRC(4)  |
+=========+=========+========+============+========+
```

ACK Frame

```
+=========+=========+========+
|  KIND(1)  |  ACK(1)  |  CRC(4)  |
+=========+=========+========+
```

NAK Frame

```
+=========+=========+========+
|  KIND(1)  |  ACK(1)  |  CRC(4)  |
+=========+=========+========+
```

```
// 帧的数据结构
typedef struct {
    frame_kind kind;               // 帧的类型
    seq_nr ack;                    // 确认的帧的序号
    seq_nr seq;                    // 帧序号
    unsigned char data[PKT_LEN];   // 帧数据
    unsigned int padding;
} Frame;
```

2.2. 数据结构

```
#define DATA_TIMER 2000               // 数据帧的超时时间
#define ACK_TIMER 300                // 确认帧 ACK 的超时时间
#define MAX_SEQ 63                    // 最大帧序号
#define NR_BUFS ((MAX_SEQ + 1) / 2)   // 帧缓冲区的大小
#define inc(k)       \
    if (k < MAX_SEQ) \
        k++;         \
    else             \
        k = 0  // 序号增加

typedef unsigned char frame_kind;
typedef unsigned char seq_nr;
```

```
typedef unsigned char packet[PKT_LEN];
typedef unsigned char boolean;

static int phl_ready = 0;    // 物理层是否准备好，默认没有
static boolean no_nak = 1;   // 初始化，未发送 NAK

seq_nr frame_expected = 0;   // 下一个准备接收的帧序号,接收窗口下限
seq_nr too_far = NR_BUFS;        // 接收窗口上限+1
seq_nr next_frame_to_send = 0;   // 发送窗口上限+1
seq_nr ack_expected = 0;         // 发送窗口下限

packet in_buf[NR_BUFS];     // 接收窗口缓存
packet out_buf[NR_BUFS];    // 发送窗口缓存
boolean arrived[NR_BUFS];   // 表示 in_buf 的占用情况
seq_nr nbuffered = 0;       // 表示 in_buf 中已接收但未确认的帧个数

int event;     // 事件类型
int arg;       // 超时计时器的序号
Frame f;       // 接收到的数据帧
int len = 0;   // 接收到的数据帧的长度
```

2.3. 模块结构

2.3.1. 函数设计

```
// 判断序号是否正确，区间左闭右开
static boolean between(seq_nr a, seq_nr b, seq_nr c)

// 补充校验和，传送给物理层
static void put_frame(unsigned char* frame, int len)

// 构建并发送一个 data/ACK/NAK 帧
static void send__frame(frame_kind fk, seq_nr frame_nr,
                        seq_nr frame_expected, packet buffer[])
```

函数简要说明：

1. between 函数：

本函数共有三个参数，均为seq_nr类型，用于判断序号是否在传入的上界与下界之间，其中判断标准为左闭右开。三个参数中 a 为下界，b 为待判断的序号，c 为上界。

2. put_frame 函数：

本函数共有两个参数，其中第一个参数为传入的帧，第二个参数为帧的长度。函数实

现的功能为给帧补充 CRC 校验位，调用 send__frame()函数进行帧的发送，同时将 phl_ready 置为 0，假设物理层缓存已满，等待物理层的回应。

3. send__frame 函数：

本函数用于实现发送帧的操作。函数共有四个参数，其中第一个参数为帧的类型，共有 FRAME_DATA,FRAME_ACK,FRAME_NAK 三种类型，函数根据不同的类型进行不同的操作；第二个参数为即将发送的帧的序号；第三个参数为 ACK 的序号，可用于捎带确认；第四个参数为发送方的缓存窗口，用于将需要发送的数据传递。

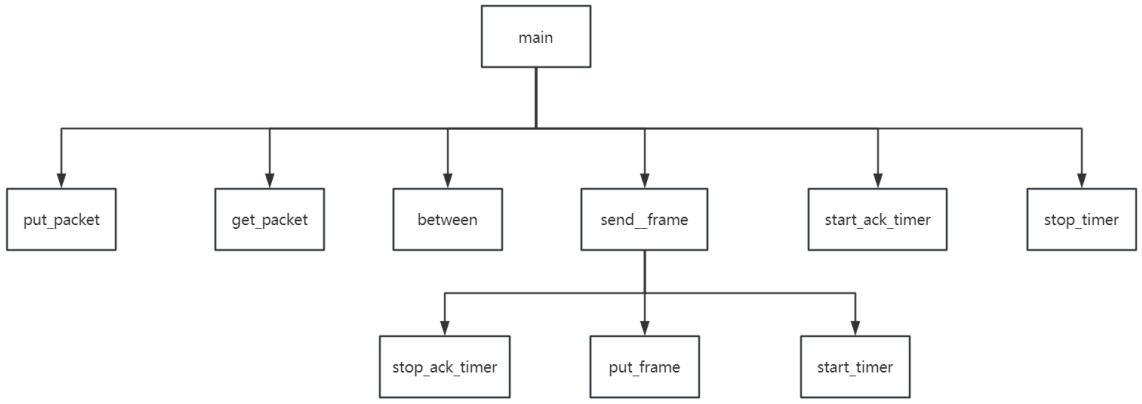
函数先将帧的类型、数据帧的序号、捎带 ACK 的序号填入帧中，然后根据帧的类型进行如下操作：

FRAME_DATA：函数根据需要传递的帧的序号从缓存中取出对应的数据，调用 put_frame()函数填充 CRC 校验位并发送，而后开始计时器；

FRAME_ACK：直接调用 put_frame()函数填充 CRC 校验位并发送，而后开始计时器；

FRAME_NAK：将 no_nak 置为 0，然后调用 put_frame()填充 CRC 校验位并发送，而后开始计时器。

2.3.2. 函数调用关系图

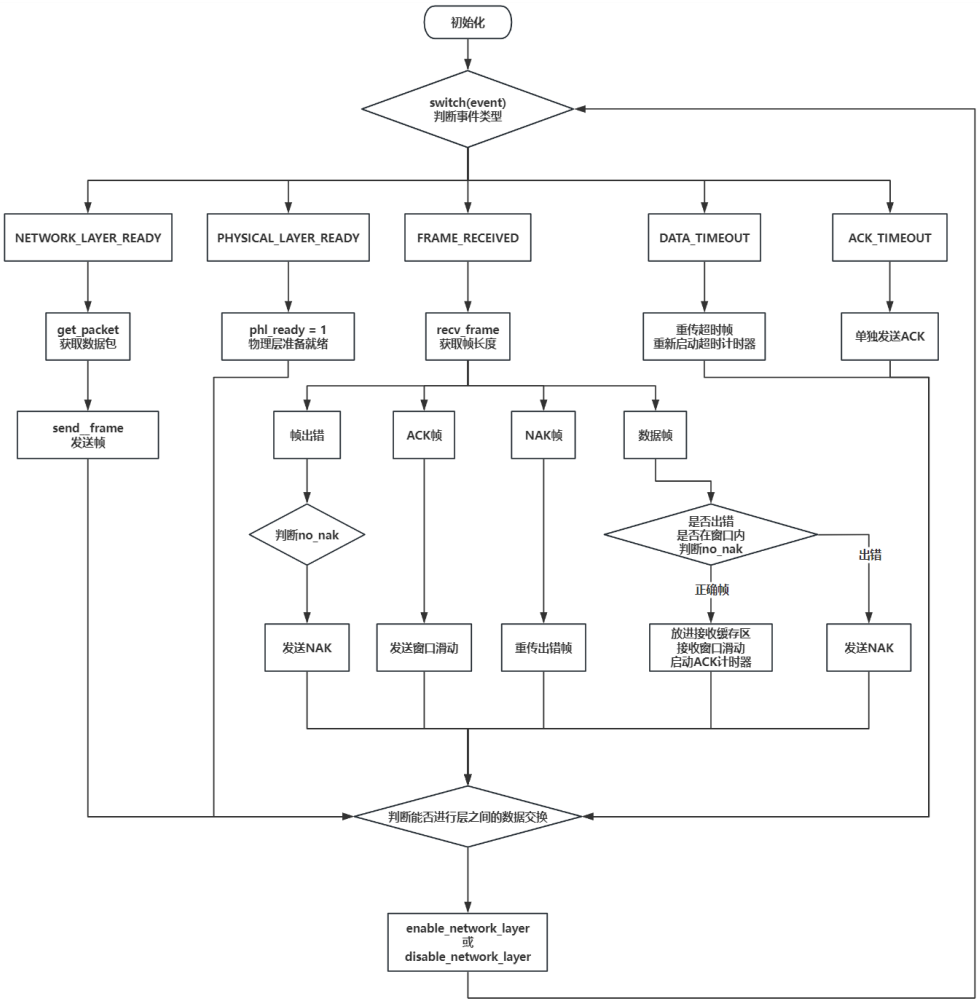


2.4. 算法流程

站点 A 给站点 B 传送一帧，如果该帧无差错，则站点 B 回传给站点 A 一帧，该帧中携带了对 A 传输的上一帧的确认。如果 B 在一定时间内没有要给 A 发送的数据帧，则 B 单独回传一个 ACK 确认帧。

若该帧有差错，则站点 B 回传给站点 A 一个 NAK 帧，站点 A 收到 NAK 帧，选择重传出错的帧。

具体过程如下图：



3. 实验结果分析

3.1. 程序正确性和健壮性分析

程序实现了在有误码信道中的无差错传输功能。程序的健壮性良好，可以长时间稳定运行。

3.2. 协议参数的选取

3.2.1. 滑动窗口的大小计算

首先根据 t_p, t_f 计算出 α：

$$\begin{cases} t_p = 270\ ms \\ t_f = \dfrac{(256\ +7\ )\times 8}{8000} \times 1000 = 263\ ms \\ \alpha = \dfrac{t_p}{t_f} \approx 1.027 \end{cases}$$

在无差错的情况下，根据选择重传协议的性能分析公式有(由于是 piggybacking，所以是 $2+2\alpha$)：

$$U = \frac{w}{2+2\alpha} \geq 1$$

推导出 $w \geq 4.054$ ，取整得 $w = 5$ 。

又由于 $w = 2^{n-1}$ ，所以取 $w = 8$ 。

但随着后续分析重传定时器的时限时发现，随着我们的 DATA_TIMER 从 1500 ms 提升为 2000 ms，过小的（窗口数为 8）的窗口数会导致在出错重传时阻碍后面的帧继续发送，故又将窗口数设置为 16，即 MAX_SEQ = 31.

但通过调试程序发现，有一定的几率可能会用到 16 个以上的发送窗口，由于这点内存开销是十分小的，因此为了达到最大性能，又取得 $w = 32$。

3.2.2. ACK 搭载定时器的时限 ACK_TIMER 计算

由于帧的超时重传需要进行 ACK 计数，故此处先进行 ACK_Timer 的计算。

重传定时器的时限应大于一帧发出并收到它的 ACK 所需的时间，保证不会因为长时间收不到 ACK 而重发帧。同时 ACK 搭载定时器的时限应小于发送窗口被装载满的时间，保证不会因为发送窗口长时间不更新，而导致协议性能降低。

因此可以得到以下不等关系：

$$\begin{cases} T_{data} > 2t_p + 2t_f = 1066\,ms \quad (piggybacking) \\ T_{data} > T_{ack} + 2t_p + 2t_f \quad (\text{单独的}ACK\text{帧}) \\ T_{ack} < w \cdot t_f \end{cases}$$

同时T_{ack}不能设定的太小，否则会使得接收方一直发送单独的 ACK 帧，而不采用 piggyback 机制，导致协议的性能大大降低。

通过搜索阅读网上关于 TCP、稍待确认等的 ACK 定时器时限的设定，我们了解到通常情况下，ACK_TIMER 的值为 200 ms.故我们在 200 的取值范围附近进行了测试。

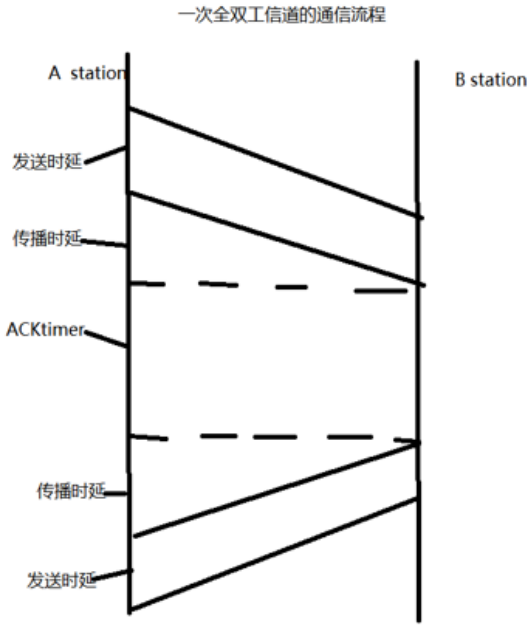
从多次测试我们发现，若 ACK_TIMER 小于 100 ms，会由于经常发出单独的 ACK 帧，导致信道利用率较低，比较明显的结果是：在测试无误码信道时，B 的信道利用率一直居于 90%以下；而当 ACK_TIMER 过大（如 800、1000ms）时，会由于发送方经常接收不到 ACK 而导致信道利用率下降，此时的足以发现问题的测试结果为：在默认的信道情况下，A 的利用率为 50%，而 B 只有 84%。

在上述多种测试下，我们最终决定较为合适的 ACK_TIMER 值为 300 ms.

3.2.3. 重传定时器的时限 DATA_TIMER 计算

3.2.3.1. 初步确定 DATA_TIMER

由帧的结构可以得出，一帧的长度最大是 263 字节，这意味着，在 8000bit 每秒的带宽下，发送时延为 263ms，传播时延为 270ms，忽略 ACK 的发送时间，我们可以得到以下流程：



数据帧的超时时间必须超过一次完整的全双工通信时间，故我们需要求出一次完整的通信需要的最长时间。

经过我们的分析，以上流程是时间最长的通信：首先由 A 发出一个帧，经历发送时延和传播时延完全到达 B 站点，此时 B 站点开始等待，这期间若收到新的帧就将 ACK_Timer 清零，否则发送 ACK，这里由于我们需要求出最大时长，则认定在 ACK 超时前的最后一秒收到对方已经发完的消息，此时轮到 B 站点发送信息，这与 A 发送一个帧的流程无异，故也经历一个发送时延和一个传播时延。

由以上的理论分析可以得到最大时间为 263+270+300+270+263=1366ms，但我们设置的数据帧超时时间必须大于这个值，于是我们暂时将数据帧超时时间设置为 1500ms。

3.2.3.2. DATA_TIMER 的优化计算

但是，在使用 1500ms 作为DATA_TIMER 进行测试的过程中，我们发现在默认条件下，A 的信道利用率为 50%，而 B 的信道利用率只有 84%。从 A 和 B 的日志中，我们发现了问题所在：

为了便于文档的编写，我们只把日志中可以体现问题的部分列出来：

A	B
	205.309 **** Receiver Error, Bad CRC Checksum
	205.309 Send NAK 34
205.864 Recv NAK 34 --6byte	
205.864 Send DATA 34 61, ID 10738	207.176 .... 745 packets received, 7380 bps, 92.25%, Err 18 (1.1e-05)
207.068 ---- DATA 35 timeout, resend ----	207.637 Send DATA 6 41, ID 20390
207.069 Send DATA 35 1, ID 10739	207.669 Recv DATA 35 1, ID 10739
207.144 Recv DATA 2 33, ID 20386	207.669 Recv frame is not lower bound, NAK sent back
207.145 Put packet to network layer seq:2, ID: 20386	207.669 Send NAK 42
207.344 ---- DATA 36 timeout, resend ----	
207.345 Send DATA 36 2, ID 10740	208.952 Send DATA 11 44, ID 20395
	209.014 Recv DATA 42 6, ID 10746
208.252 Recv NAK 42 --6byte	209.014 Recv frame is not lower bound, NAK sent back
208.252 Send DATA 42 6, ID 10746	209.014 Send NAK 45
209.555 Recv NAK 45 --6byte	209.276 .... 750 packets received, 7355 bps, 91.94%, Err 18 (1.1e-05)
209.555 Send DATA 45 11, ID 10749	
	210.268 Send DATA 16 48, ID 20400
210.867 Recv NAK 49 --6byte	210.328 Recv DATA 45 11, ID 10749
210.867 Send DATA 49 16, ID 10753	210.329 Recv frame is not lower bound, NAK sent back
	210.329 Send NAK 49
212.213 Recv NAK 53 --6byte	
212.214 Send DATA 53 21, ID 10757	211.362 .... 757 packets received, 7350 bps, 91.88%, Err 18 (1.1e-05)
	211.611 Send DATA 21 52, ID 20405
	211.641 Recv DATA 49 16, ID 10753
	211.642 Recv frame is not lower bound, NAK sent back
	211.642 Send NAK 53
	213.495 .... 763 packets received, 7334 bps, 91.68%, Err 18 (1.1e-05)
	215.878 .... 770 packets received, 7320 bps, 91.50%, Err 18 (1.0e-05)
	218.008 .... 776 packets received, 7305 bps, 91.31%, Err 18 (1.0e-05)

举一个普遍一点的例子加以说明：

A 连续向 B 发送了 0 号帧、1 号帧、2 号帧……，B 收到了 0 号帧，A 也收到了 B 发来的 0 号帧的 ACK，但是 A 发送的 1 号帧发生了错误（如上表 B 的 Receiver Error, Bad CRC Checksum）。

B 接收到错误的 1 号帧后会向 A 发送 NAK 请求重传 1 号帧。

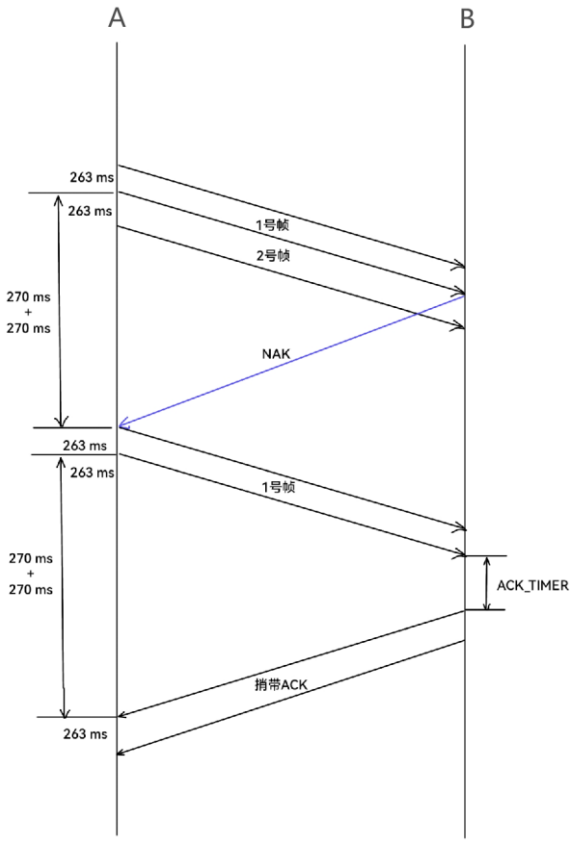
在 B 收到 A 重传的正确的 1 号帧之前，B 发给 A 的数据帧的捎带 ACK 都只能是 0，A 也就无从得知 2 号帧、3 号帧是否被 B 正常接收了。

由于定时器的时限（1500ms）很短，A 的 2 号帧的定时器很快就会超时（甚至 3 号帧的定时器也会超时）（如上表 A 中的 DATA 36 timeout, resend），于是 A 重发 2 号帧，但事实上，此时 B 不仅正确地收到了 2 号帧，也收到了重传的 1 号帧，因此 B 的接收窗口已经向前滑动了，2 号不再处于 B 的接收窗口内。

根据协议的处理逻辑，B 会发送一个 NAK 帧来报告错误。

然而，此时 B 没有任何帧需要重传，这个多余的 NAK 帧会导致 A 发来一个多余的重传帧，这个多余的重传帧到达 B 时又已经不在 B 的接收窗口内了，于是 B 又发送了一个多余的 NAK 帧……如此往复。

信道上偶然的一次错误会导致此后信道上持续出现冗余数据。



那么，我们在上述情况下，考虑增大 DATA_TIMER，求解过程如下：

由 3.2.3.1 可知，当前理论最大延迟为 1366ms。

我们考虑上述情况，当 1 号帧出错，B 发送 NAK，A 接收 NAK 并重发 1 号帧，B 接收 1 号帧并等待 ACK_TIMER，然后结合数据帧发送 ACK。此时的延迟为：

$$263 + 540 + 263 + 270 + 300 + 263 + 270 = 2169 \text{ ms.}$$

但是，因为 A 在接收到 NAK 并重发 1 号帧时，重新启动了 1 号帧的 DATA_TIMER，所以此时，计时最早的 DATA_TIMER 是 2 号帧的。

这个过程中要保证 2 号帧不重发，即 2 号帧的 DATA_TIMER 不超时。那么在计算最大延迟时，需要从 2 号帧开始，即减去 1 号帧的发送时延：

$$2169 - 263 = 1906 \text{ ms}$$

故，我们考虑一些不可避免的延迟，将 DATA_TIMER 的值设置为 2000 ms.

3.3. 理论分析

无差错信道环境下分组层能获得的最大信道利用率：

$$\eta = \frac{256}{256 + 7} \approx 97.34\%$$

下面分析在有误码条件下重传操作及时发生等理想情况下分组层能获得的最大信道利用率。

为了简化有误码条件下的最大利用率推导过程，可以对问题模型进行简化。由于 ACK 帧和 NAK 帧长度很短，出现机率较小，所以假设这些帧能 100% 正确传输。

假定信道误码率为 p ，由于误码率代表每个 bit 出错的几率，因此对于长度为 263 字节的一帧，发送成功的概率为：

$$A = (1 - p)^{263 \times 8} = (1 - p)^{2104}$$

因此成功传输一帧所需要的次数为：

$$\sum_{i}^{\infty} i \cdot (1 - A)^{i-1} \cdot A = \frac{1}{A}$$

因此有误码条件下的最大信道利用率为：

$$\eta = \frac{256A}{263}$$

当信道误码率为 10^{-5} 时：

$$\eta = \frac{256A}{263} \approx 95.31\%$$

当信道误码率为 10^{-4} 时：

$$\eta = \frac{256A}{263} \approx 78.87\%$$

3.4. 实验结果分析

性能测试记录表

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	1500.342	54.58%	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”	909.045	54.90%	94.48%	
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	1249.922	96.95%	96.95%	
4	--flood	站点 A/B 的分组层都洪水式产生分组	1139.417	94.22%	94.86%	
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}	1421.710	58.92%	60.25%	B 站点的线路利用率与理论性能差距较大

运行时间要求超过 10 分钟。

3.5. 当前程序存在的问题

在序号 5 的命令下，协议的信道利用率与理论性能差距较大：

参考值：A：42.0%；B：73.6%

测试值：A：58.9%；B：60.25%

我们查看了高误码率下的日志发现，经常会出现一连串的冗余重传帧。我们猜测很有可能是因为在短时间内发生多次错误，使得双方都忙于错误处理，导致延迟大大增加。

4. 研究和探索的问题

4.1. CRC 校验能力

根据本次实验提供的 crc32.c，CRC 校验采用的生成多项式为：

$$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^{8}+x^{7}+x^{5}+x^{4}+x^{2}+x+1$$

理论上，该校验方法可以检测出所有的一位错误、两位错误、奇数位错误和长度≤32位的突发错误。

对于长度为 33 的突发错误，不被检测到的概率为：

$$p=\frac{1}{2^{32-1}}=\frac{1}{2^{31}}$$

而对于长度大于 33 的突发错误，不被检测到的概率为：

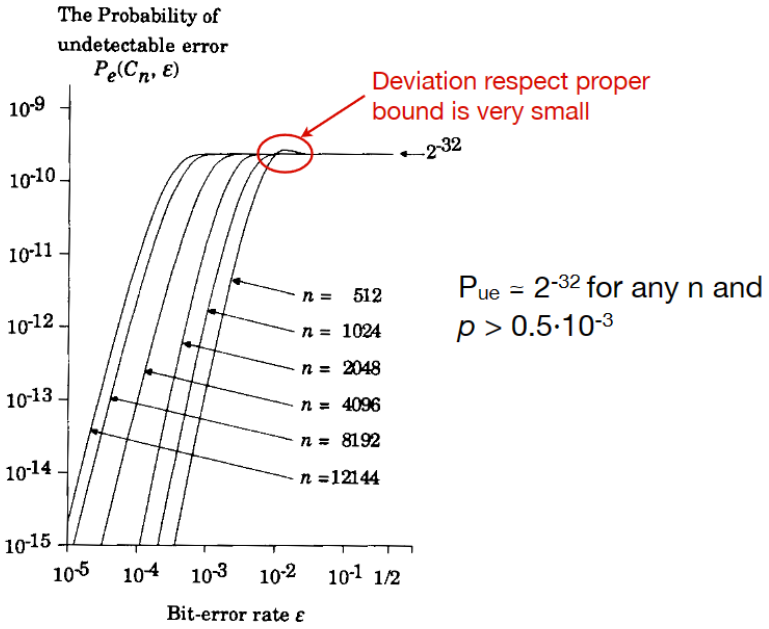
$$\mathrm{p}=\frac{1}{2^{32}}$$

根据在网络上搜索到的相关研究，

（https://www.ieee802.org/3/bv/public/Jan_2015/perezaranda_3bv_2_0115.pdf）

如下图：

- $P_{ue}(p)$ versus p for the IEEE-802 code on the BSC's for $n=2^i$ with $9\le i\le 13$ and n = 12144 [Fujiwara89]



本次实验的帧为 263 字节，即 2104 位，其在误码率为10^{-5}的信道上，出现分组层误码（即上图中 undetectable error）的概率远远小于10^{-15}，我们暂且估计其为10^{-20}.

那么因为客户的通信系统每天的使用率为 50%，我们可以算得：

$$\frac{1}{\frac{1000}{263} \times 60 \times 60 \times 12 \times 365 \times 10^{-20}} \approx 1.67 \times 10^{12}$$

即平均 1 万 6 千亿年发生一次分组层错误。

事实上，现实中的信道更多地发生突发错误而非零散错误，故上述结果是保守计算，因此客户根本无需担心。

若想要降低发生分组层误码事件的概率，还可以使用更长的纠错码，代价是会降低信道利用率。

4.2. CRC 校验和的计算方法

以字节值查表并叠加的方案与手工进行二进制“模 2”除法求余数的算法是等效的。

查表法的原理是预处理出从 0x00 到 0xFF 的每个字节后填充 0 的 CRC32 余数，并存在表里。计算时以字节为单位进行“模 2”除法，利用预先生成的表快速查找，从而加快效率。

我们使用下述代码生成速查表的 256 个数字：

```
#include <stdio.h>

#define CRC 0xEDB88320

int main() {
    printf("static const unsigned int crc_table[256] = {");

    for (int i = 0; i < 256; i++) {
        if (i % 5 == 0)
            printf("\n    ");
        unsigned int v = i;

        for (int j = 0; j < 8; j++)
            v = v & 1 ? (v >> 1) ^ CRC : v >> 1;

        printf("0x%08xL", v);
        if (i < 255)
            printf(", ");
    }

    printf("\n};\n");
```

```
    return 0;
}
```

在 x86 计算机上计算 16 位整数和 32 位整数的速度相同，因此，根据 RFC1662.txt 中的代码，使用查找表法为某一帧计算 32 位 CRC-32 校验和和计算 16 位 CRC-16 校验和所花费的 CPU 时间是一样的。

RFC1662.txt 给出的函数 pppfcs32(fcs, cp, len)中的参数 fcs 用于在已有的 CRC 值基础上进行增量计算，提高效率。

4.3. 程序设计方面的问题

4.3.1. 协议软件的跟踪功能

协议软件的跟踪功能有助于获取详略得当的日志文件用以辅助调试和分析。默认情况下，不产生任何调试信息的输出；通过某种命令可以打开调试开关，边运行边产生协议动作信息的输出。我们的程序适当地使用了 dbg_frame, dbg_event 系列函数，对调试分析有很大的帮助。

4.3.2. get_ms()函数

程序库中获取时间坐标的函数 get_ms()函数，还可以使用 C 标准库中的 time.h 中的标准函数 time 和 clock 实现：

```
#include <time.h>
#include <stdint.h>

unsigned int get_ms(void)
{
    // 使用 clock 获取自程序启动以来的时钟滴答数
    clock_t start = clock();

    // 将时钟滴答数转换为毫秒
   unsigned int ms=(unsigned int)(start * 1000 / CLOCKS_PER_SEC);

    return ms;
}
```

4.3.3. 不定参数的函数

lprintf, dbg_frame, dbg_event, dbg_warning 这些变长参数函数是通过 C 语言的 va_start, va_end, va_arg, va_list 等宏实现的。

4.3.4. 定时器函数

start_timer 函数把截止到调用函数时刻为止已经放入物理层发送队列的数据发送完后才启动计时，这是因为 start_timer 函数用于重传定时器，应该在一个数据帧真正开始发出去的时候才开始定时。定时器超时前调用 start_timer 函数的效果是重新开始计时，这是因为在定时器超时前收到 NAK 帧的情况下需要重新开始重传定时器。

start_ack_timer 函数在调用后立即开始计时，这是因为 start_ack_timer 函数用于 ACK 定时器：如果在函数调用后一段时间内没有数据帧发出则发送单独 ACK 帧。定时器超时前调用 start_ack_timer 函数的效果是维持定时器当前的剩余时间，因为最早的未发送 ACK 的帧到达的时间超过限制就应该发送单独 ACK 帧，不应该因为后续又有数据帧传来而重新开始等待时间。

4.4. 软件测试方面的问题

设计七种测试方案的目的是充分测试程序在各种情况下的性能。比如误码率较高的信道情况，即接收方多次接收到错误帧的情况等多种实际应用中会出现的特殊情况。

4.5. 对等协议实体之间的流量控制

本程序实现了面向物理层的流量控制。

在物理层繁忙时通知网络层暂停发送数据包（disable_network_layer()），在物理层空闲时通知网络层恢复发送数据包（enable_network_layer()）。

本程序没有实现面向网络层的流量控制。本实验假设网络层随时能够不限量地接收数据链路层提交的数据包。但在现实中网络层也有可能拥堵。如果要实现面向网络层的流量控制，可以使用如下两种方案之一：

- 在网络层拥堵时直接丢弃收到的帧。

- 增加一些控制帧，用于在某一方的网络层拥堵时通知另一方停止发送数据或降低发送数据的速率；在某一方的网络层恢复时通知另一方恢复发送数据或提高发送数据的速率。

4.6. 与标准协议的对比

若需要利用我们设计的协议通过卫星通道进行通信，还需要解决以下问题（包括但不限于）：

- 卫星通信的高延迟是否会导致性能的严重下降；
- 各种天气问题是否会对传输产生影响，导致丢包、误码率上升等等问题；

当前实验性协议的设计与成熟的协议标准的差距和遗漏的重要问题如下：

- 本实验隐藏了成帧这一非常重要的过程；
- 须实现面向网络层的流量控制的功能；
- 一个实用协议应该能够根据链路的带宽、延迟和误码率动态地协商各项参数而不是针对特定情景手动设置；
- 一个成熟协议还应该能够传递各种控制信息，控制链路的连接与断开等；
- 需要考虑短时间的突发流量的处理。

5. 实验总结与心得体会

5.1. 实验总时长

本次实验总耗时约 5+4.5+1.5+4.5+1.5=17 小时：

完成初步的代码编写花费大约 5 小时，其中超过一半的时间用于阅读指导书和提供的其他源码；

小组讨论协议所需各参数的具体取值耗时约 4.5 小时，期间包括分析原因、修改代码、测试数据；

测试最终数据花费大约 1.5 小时；

完成报告的基本编写耗时约 4.5 小时，其中主要包括分析源码、撰写报告、研究问题等工作；

完成报告的优化 1.5 小时，包括报告格式、问题检查、添加演示图片等工作。

5.2. 实验中所遇到的问题

在 C 语言编译器、C 程序编写过程中，我们组员都有足够的经验，故未有值得记录的问题。

而在协议的确定、测试过程中，遇到了以下问题：

- ACK_TIMER 设定与算法采取的方法略有出入，导致编写代码时未能将 stop_ack_timer 函数放置位置有出入，出现实际效率与理论效率偏差过大的问题；
- 未能考虑周全实际的发送过程中会出现的情况，导致设置 DATA_TIMER 时偏小，效率偏低；
- 由于开始时未能够明确使用的 DATA_TIMER 和 ACK_TIMER，花费大量时间用于测试不同的数据组合的效率；
- 测试时并没有让 CPU 使用完整性能，导致测试出现的效率低于预期，偶尔出现占用过高需要停止操作等待拥塞解除；

5.3. 实验心得总结

完成本次实验的过程带给我们很多收获。

通过本次实验，我们对选择重传协议（SR 协议）的理解达到了一个新的深度。

SR 协议作为滑动窗口协议的一种，它允许发送方在未收到所有确认的情况下继续发送新的数据包，这在提高网络传输效率方面起到了关键作用。我们不仅学习了 SR 协议的理论基础，还通过实际动手实践，从零开始构建了一个协议模型，这一过程极大地加深了我们的理解。

在实现协议的过程中，我们面临了从理论到实践的挑战。我们必须将抽象的协议概念转化为具体的代码逻辑，这不仅考验了我们的理论知识，也锻炼了我们的编程技巧。逐渐掌握了如何高效地组织代码结构，如何编写清晰、可维护的代码，以及如何通过模块化设计来简化复杂问题。

分析协议的理论性能和实际性能之间的差异，是我们在实验中的另一个重要环节。我们设计了一系列测试用例，模拟了不同的网络条件，以评估协议在各种情况下的表现。通过对比理论预期和实际测试结果，我们发现了实现中存在的一些问题，比如在高丢包率的网络环境下，协议的重传机制可能导致不必要的延迟。

为了提高协议的性能，我们进行了多次调试和优化。优化了重传机制，减少了因各种错误而导致的不必要重传。其中一些改进显著提升了协议在模拟环境中的表现。

总结来说，这次实验不仅让我们更深刻地理解了选择重传协议，也极大地提升了我们的编程实践能力。我们认识到了网络协议设计的精细性，以及在不断变化的网络环境中追求高性能的重要性。我们相信这些经验和技能将对我们未来的学习和职业发展产生积极影响。

6. 源程序文件

见 datalink.c