**计算机网络实验报告**

**回退N协议**

**一、 实验内容和实验环境描述**

数据链路层通信的两个站点分别为A和B，仿真环境利用Windows10环境下的TCP 协议和Socket客户端/服务器机制构建两个站点之间的通信。编译、链接之后生成的可执行程序(.exe 文件)为字符界面命令行程序（不是图形界面程序）。利用dos命令运行协议，使两站点之间顺利通信。实验环境中的物理层为数据链路层提供的服务为8000bps，270ms 传播延时，10^-5误码率的字节流传输通道。

本次编码环境为Codeblocks 16.01 的IDE，在GNU GCC下编译完成。

本次实验实现了滑动窗口协议中的gobackN协议，采用了ACK搭载技术，NAK否定确认（可以在代码中开启此功能，但为了突出gobackN特质，在报告中忽略nak的存在，即一切错误只会引发超时而非nak帧），超时重传等机制。

**二、软件设计**

**（一）、数据结构**

**1、帧结构**

struct FRAME {

unsigned char kind; // frame kind : ack/nak/data

unsigned char ack;

unsigned char seq; // sequence number from 0 to MAX\_SEQ

unsigned char data[PKT\_LEN];

unsigned int padding; // give room for crc32

};

kind为帧的类型字段，占用一个字节，可选择的kind有ack、nak（作为可开启功能，默认是关闭的）和data。即确认帧，否定确认帧和数据帧。

ack为确认信息字段，占用一个字节。由于采用了捎带确认技术，ack字段提供了确认帧的序号。

seq为帧需要字段，占用一个字节，它的范围是从0至MAX\_SEQ。

padding为填充字段,为以后的CRC32校验留出4个字节的字段。

**2、全局变量**

static unsigned char buffer[MAX\_SEQ+1][PKT\_LEN]; //out buffer here

static unsigned char nbuffered; //the use of buffers

static unsigned char frame\_expected = 0; //receiver window with the width of 1

static unsigned char ack\_expected = 0; //lower bound of send window

static unsigned char next\_frame\_to\_send = 0; //upper bound of send window

int no\_nak = 1; //only need one nak call for expect frame

static int phl\_ready = 0;

已为每个变量写好了注释，这里不再赘述。请注意，phl\_ready提供了物理层队列未满的信息。

**3、main函数中变量**

int event, arg;

struct FRAME f;

int len = 0; // the length of frame

event是事件参数，在define中定义了event可能的取值。

arg是命令行控制信息，返回超时定时器的编号，注意到无论是数据帧超时还是ack超时，均会产生一个arg参数。

f是收到的帧。

len是帧长度，初始化为0。

**（二）、函数调用结构**

主程序分为三个部分，packet处理、帧的发送和判定窗口。分别对应put\_packet(get\_packet), send\_frame和between。

get\_packet/put\_packet：与网络层的接口，负责帧的交付与获取。

timer：定时器模块。 start\_timer和start\_ack\_timer是系统的库函数，负责数据帧定时器和单独ack定时器。

send\_frame：完成了向物理层发送一个帧，这个帧分为三大类：正常数据帧，单独ack帧和因超时而重传的数据帧。请注意，当nak功能开启时，发送nak帧是可能的。

between：负责判断帧是否落入窗口。

CRC32：负责计算校验和，如果校验错误，需要跳过piggybacking过程，如果一个帧被损坏，我们要做的只是等待它超时。

**（三）、算法流程**

如图给出在主函数中，无限循环for中的一个过程。

协议要求等待一个事件，事件只可能是下列事件其中之一：

1. **NETWORK\_LAYER\_READY**

此时网络层允许链路层封装一个帧并发送它。此时要做的仅是从发送缓存中取出一帧发送，并更新发送窗口的上界。

1. **PHYSICAL\_LAYER\_READY**

此时帧可以从链路层传递给物理层，只需要给出这个事件发生的标记。

1. **FRAME\_RECEIVED**

此时链路层收到了来自物理层的一个帧。我们假设不开启nak功能。我们首先要判断这个帧是否是完好无损的，否则直接break，等待对方超时。

如果它通过了CRC校验，它只可能是数据帧、ACK帧。若是数据帧，先判别它是否是我们最期待的帧，如果不是，则忽略它，对方最终会超时。如果是ACK帧，则什么也不用做，稍后会处理捎带应答过程。

最后，进行捎带应答过程。请注意，这两种帧都携带了ack信息，因此均有必要处理捎带应答过程。

1. **DATA\_TIMEOUT**

此时某数据帧超时。引发超时的总是发送窗口的下界帧超时，因此回退N的过程在此体现，程序首先将发送窗口的上界回退到下界位置处，然后根据现在的nbuffer，连续重发帧直到恢复到原来的nbuffer数量。请注意，此时是发送方连续发送帧，这些帧将被连续送入物理层对列中。

1. **ACK\_TIMEOUT**

此时ACK超时，它不应该继续等待捎带应答的数据帧，而是发送一个独立的ACK帧以加快确认。

**三、实验结果分析**

**（一）、综述**

该协议至少可以正确运行8小时以上，并稳定在较高的效率（性能测试记录表）。

**（二）、协议参数的选取**

先计算Tf与Td

Tf = ( 256 \* 8 ) / 8K = 256ms

Td = 270ms

1. **滑动窗口大小：**假设ACK每次都被捎带回去，利用公式

η = Ws \* Tf / ( 2\*Td + 2\*Tf ) <= 1

解得Ws < 4 ，但是我们考虑到大一些的窗口几乎总是让效率提升的，并且更加接近稍后分析的极限效率。但是，请注意，我们这里选择了连续重发的gobackN，如果在回退过程中，窗口宽度过大，那么它连续发送帧的上界会变得跟宽度一样大，这个过程本身就是阻碍gobackN效率的过程。因此，窗口在满足让信道充斥帧的情况下，不应该设置得过大，因此我们这里选择发送窗口大小为7（序号宽度为8），接收窗口大小显然只能为1。

1. **DATA\_TIMER：一**个帧被发送直到收到确认，这个数值几乎是个定值。这个值大约为2Tf+2Td即1052ms，如果信道稳定，将重传定时器设置成略大于这个值，可以保证最好的监视效果，但此时NAK几乎起不到作用，因为所有的坏帧都因超时而重传而非收到NAK。但实际测试证明，在不太稳定的信道，这个值应该设置得偏大一些，这样NAK可以加快丢失帧或损坏帧的重传速度。但是，请注意，在回退过程中，由于连续发帧，此时另一端返回的ack很可能被阻塞，因此，这个时间应该设置得尽量宽，以避免回退过程带来的不必要的帧重传。经测试，选取3000ms效果较好。
2. **ACK\_TIMER：**由于这里的ACK\_TIMER使用了立即开始计时的机制，并且在sendframe的同时stop掉acktimer，因此这个值应该尽可能的小，以保证无法捎带应答时，能及时发送ack以提高效率，避免不必要的等待。

**（三）极限效率的理论分析**

* 1. **无差错信道**

分组长度为256字节，seq、ack、kind各占1个字节，CRC校验4个字节。请注意，这里的分母不包括发送成功时的单独ack开销，我们假设所有的ack都可以被捎带应答。（如果不这么做，理论的极限效率会有所下降）

实际效率为

η = 256 / (256 + 1 + 1 + 1 + 4) = 97.34 %

* 1. **误码率为1e-5的信道**

在这里，我们做如下假设：

* + - 1. ack一定会被捎带应答（否则同上，分母会扩大，理论极限效率会下降）。
      2. 重传帧总是会成功到达，并且一定附加了一个同样可以总是会成功到达的nak帧（7字节）。

设成功发送data帧的概率为P，由于每个bit相互独立，我们有

P = (1 – 10^-5)^(h + d)

这里h为帧头开销，为

7\*8 = 56bit

d为数据开销，为

256\*8 = 2048bit

我们将开销分为两类：重传开销和帧头开销。

显然地，伴随着NAK，重传开销的数学期望为

R = (1 – P)\*(h + d + 7)

同样显然地，（成功传送的帧的）帧头开销的数学期望为

H = P\*h

同时，有效数据的数学期望为

D = P\*d

那么，可以计算出效率，表达式如下

η = D / (D + R + H)

带入数据可得，在误码率为1e-5的情况下，η≈95.34%。

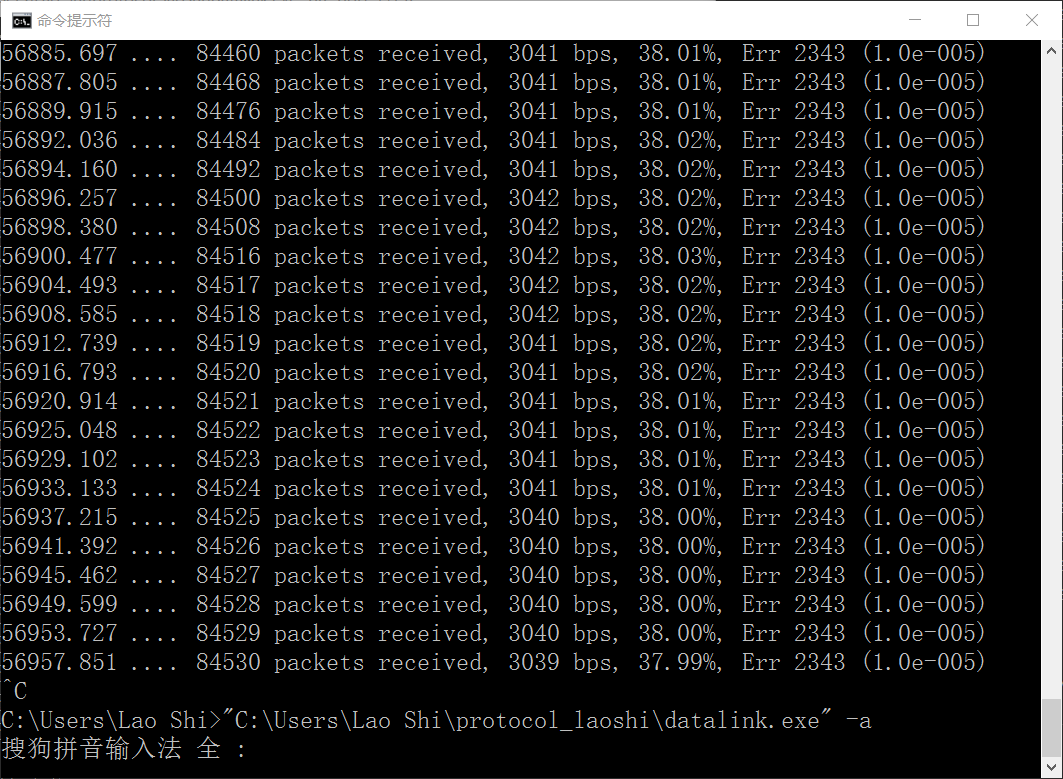
如果在默认的模式下关闭了nak帧的传输，那么分母会减去因错误而导致的单独传输nak的开销，但这样做会耗费额外的时间让对方的帧超时，这里不再做详细的计算。

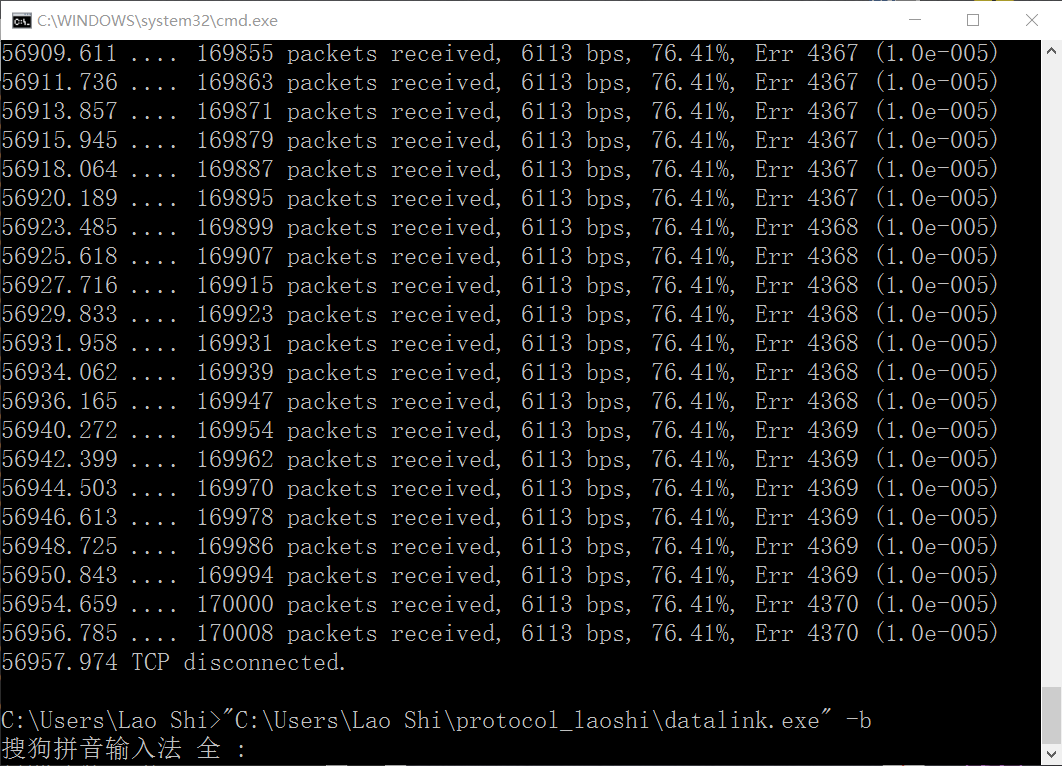
**（四）、实验结果分析**

详细数据请见实验表格。下面给出几个典型模式下的测试样例。

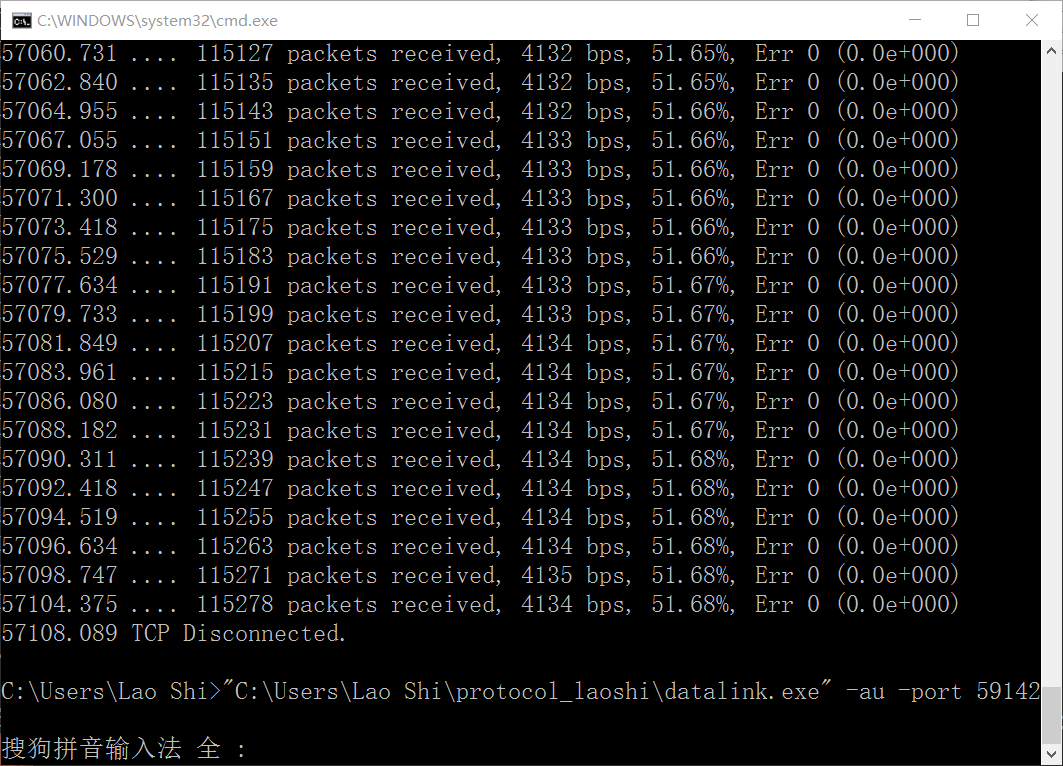
注：命令行命令显示在对话框的标题位置。

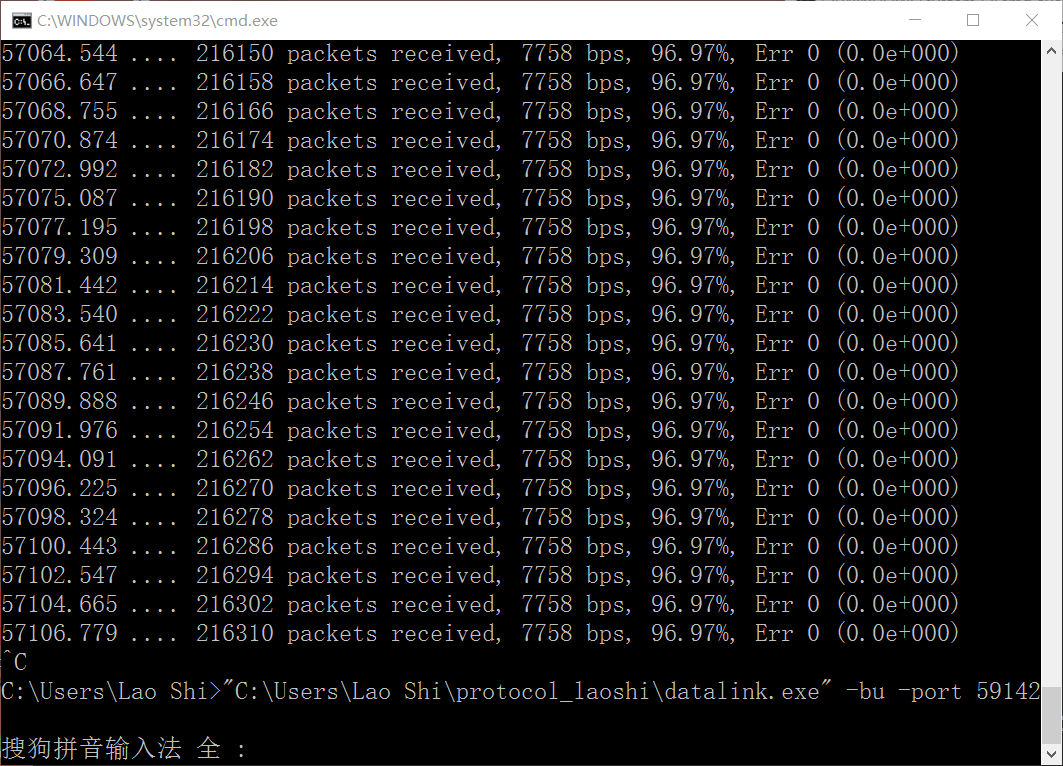
1. **a模式**



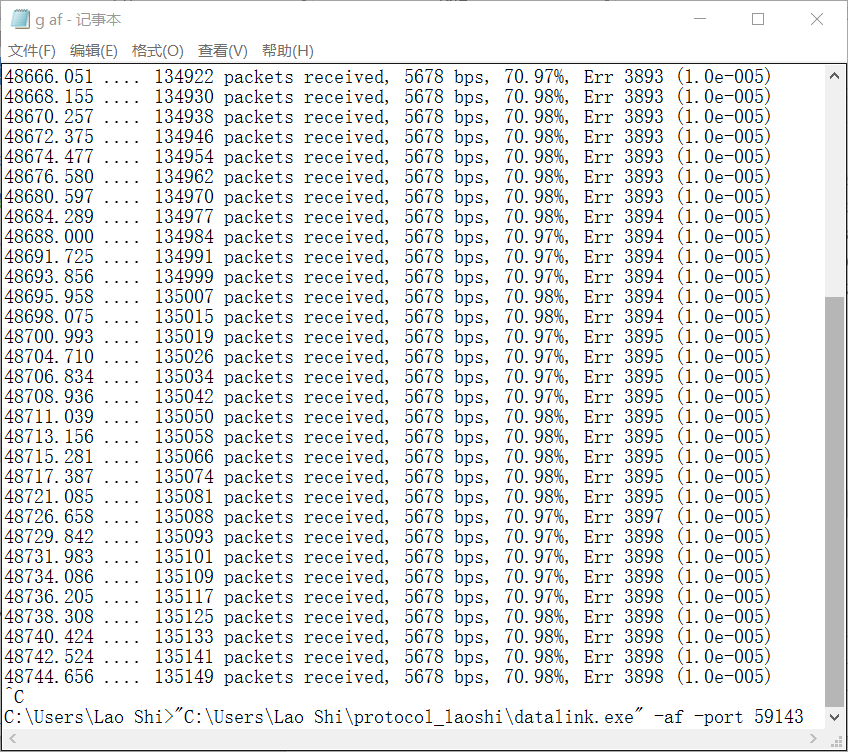


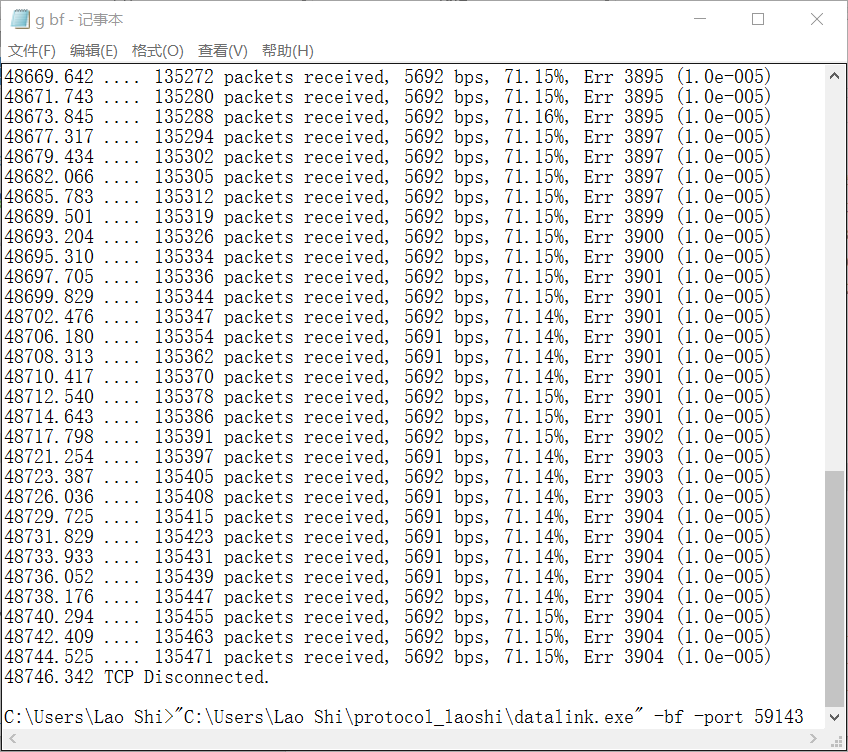
1. **au模式**



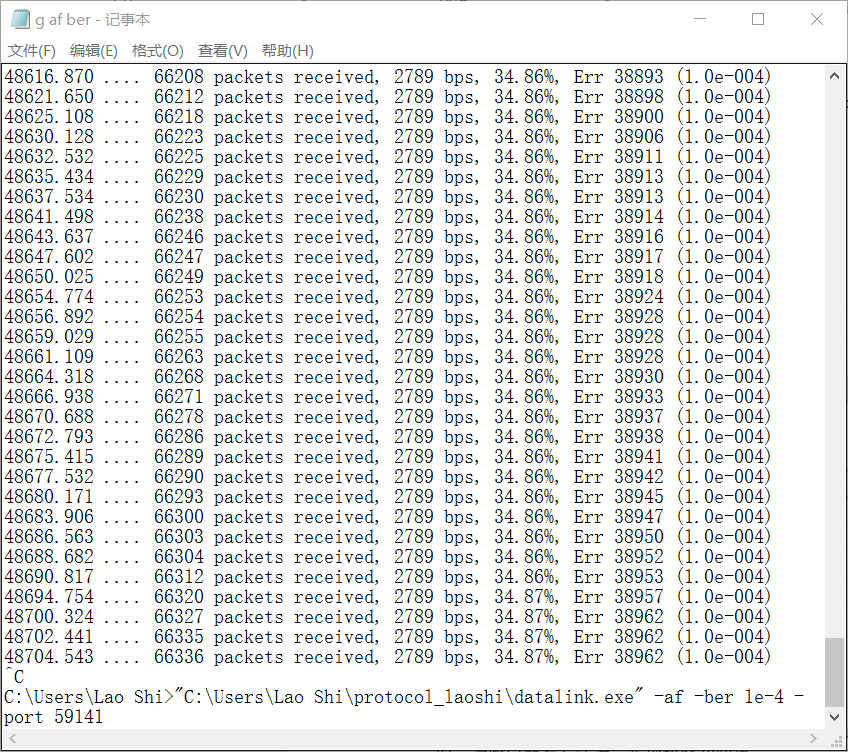


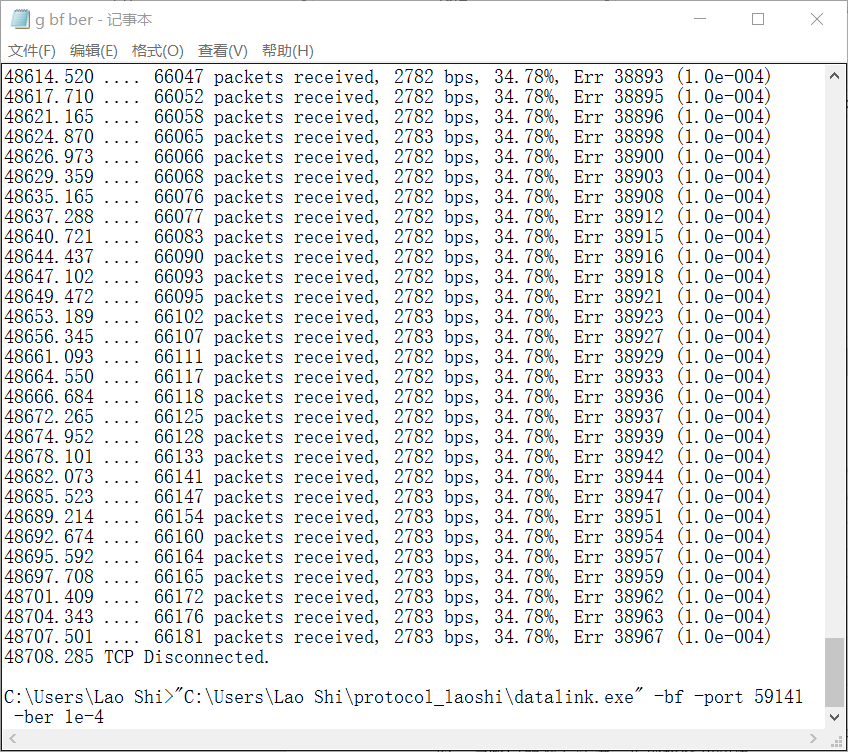
1. **af模式**





1. **af ber 1e-4模式**





由表格可以看出，随着误码率的增大，效率开始明显下降，与参考的数据还有一定差距，很可能是“回退”的实现过程还有待改进，这个程序参考了《计算机网络》教材的连续发送重发帧的方式，而没有采用示例程序的方法，所以可能导致效率并不理想。

对于参数的选择，是否存在一种更细致的分析方法，可以精确到毫秒级地分析具体的timer应该设置多少，我想可以利用统计学的方法，将程序复用后，选取以毫秒为单位的步长，得到若干个结果，再作图分析，得到统计意义上的最佳答案，这个答案势必更接近计算的理论极限。

**（五）、存在的问题**

协议未失败，但在效率上仍有几个百分点的差距。问题可能存在于定时器参数设置并不是最合理的，与定时器的理论计算偏差较大。

在（四）中提到了两个问题，一个是“回退”的方式，另一个是基于统计的参数选取，这些问题都有待进一步地探究和解决。

**四、研究和探索的问题**

**（一）、CRC校验能力**

在CRC32中出现了误码却没发现的概率是为1/2^31，每天发送12h，数据率是8K，即一秒钟可发送3.9帧，即一天发送1.7\*1e5帧。平均一天产生分组错误期望为

1.7\*1e5 \* (1/2^32) = 8 \* 1e-5

要使这个期望值达到1 ，求倒数有1.25 \* 1e4，即12500天。

平均34年才能产生这样的一个错误，因此这个错误基本上不会被感知。

**（二）、CRC校验和的计算方法**

tryfcs32函数给出了fcs的使用方法：传入函数的参数是缓冲区首地址和长度来判断该帧是否正确，注意到只使用了该帧帧头4位。而fcs是一个对于fcstab\_32的映射，需要不断更新，取哪一个fcs？pppfcs32函数完成了这个功能，它指出，一个新帧的fcs构造过程与前一帧的fcs选取有关，即前一帧的fcs用来构造新一帧的fcs。因此这个函数需要三个参数：前一帧的fcs（用来构造当前帧的fcs），缓冲区首地址（定位数据帧）和数据帧长度（用作寻址边界）。

**（三）、软件测试方面的问题**

设置五种测试方案的目的是在不同的情况下对协议的正确性进行测试。例如分为有误码信道和理想信道、平缓发送方式和交替发送方式、高误码率信道和低误码率信道等。

au/bu测试方式为程序最基本的逻辑提供保障，保证数据帧的正确传输。

a/b测试方式同时加入了两个机制：交替发送和一定的误码率。在A站停发时，可以检测独立的ack机制是否正确。在有一定误码率的情况下，可以测试出CRC、NAK、超时重传等机制的正确性。

afu/bfu是洪水方式发送数据帧，撤消了分组成给出的限制，这是在测试协议是否能承受大量帧的到来而不发生拥塞，并保持帧同步。注意到这里误码率为0，不存在重传、nak等问题。

af/bf测试了有误码率的上一种情况、即同时测试流量控制和差错处理。

af -ber 1e-4/bf -ber 1e-4 测试了高误码率的上一种情况，即同时测试流量控制和高误码率的差错处理。

对于测试方法，可以采用白盒测试，设置丰富的debug信息，通过模拟的方式对特定的情况（如重传的帧仍出现错误）进行测试。

在程序库中的定时器管理问题，在ack超时时，arg会返回128，这个值是不可变的。但如果希望序号超过127，这样返回的arg参数可能会出现问题。希望ack超时时，arg返回一个更不可能的值（如-1），而不是一个过大的值，不能保证这个过大的值将来不会被用到。

**（四）、程序设计方面的问题**

printf库函数使用了变长参数表，这些函数至少有一个参数是确认的，且已知的参数必须在最左端，后跟若干个其他参数。寻找参数的时候涉及到栈寻址，为了得到栈地址的偏移量，定义在stdarg.h的三个标准宏va\_start,va\_arg,va\_end解决了这个问题。

**（五）、流量控制问题**

本协议的流量控制主要依靠nbuffer参数和enable\_net\_work\_layer()等函数开关实现。它们完成了数据链路层与网络层和物理层之间的流量控制：与网络层，在数据链路层的缓冲区nbuffer满时，通过开关函数告知网络层，让其停止发送分组；而与物理层，若在事件处理周期内，至少调用过一次send\_frame()函数那么事件等待wait\_for\_event()会在物理层发送队列低于50 字节水平时，产生PHYSICAL\_LAYER\_READY 事件。通过监视物理层队列的长度，可以有效地防止拥塞。

**（六）、与标准协议的对比**

首先，与标准协议的效率还有一定的差异。其次，标准协议广泛适用于各类信道，这些信道情况十分复杂，参数也不尽相同，但是一些被设计得十分精巧的协议总是可以正常工作的，其参数的设置是“动态”的，并不是一成不变的。

在链路管理方面，成熟的CCITT协议使得当服务面向连接时，链路两端的[节点](http://baike.baidu.com/view/47398.htm)要进行通信前，必须首先确认对方已处于就绪[状态](http://baike.baidu.com/view/705553.htm)，并交换一些必要的信息以对帧序号[初始化](http://baike.baidu.com/view/872.htm)，然后才能建立连接，在传输过程中则要能维持该连接。如果出现差错，需要重新初始化，重新自动建立连接。这些动态的初始化、链路管理等内容在实验性协议中是不具备的，因为我们假定出现错误后打印错误报文，程序直接终止。

**五、实验总结和心得体会**

本次实验的编程时间为6个小时，时间主要花在理解库函数，配置环境和debug上（这使得开发选择另一个协议——选择重传协议的时间大大减少）。

下面列出几个bug示例：

1. 回退时没有考虑nbuffer的恢复过程，导致给了网络层错误的“缓冲区满”信息。
2. 设计帧结构时，第一次没有给CRC留出位置，导致缓冲区溢出。

在库文档方面，开始并没有注意到arg参数的价值和意义，花费了近1小时的时间自己重新开发了一个判断哪个帧超时的函数（这些函数在选择重传协议中得到应用），这个问题在gobackN中是简单的，因为引发超时的帧总是期待的那个帧。

本次协议的书写与开发令我受益匪浅。从广泛的角度来讲，它让我第一次完整、细致地看到了协议的实际运行过程。更重要的是，它让我清晰的认识到协议正确性和规范性的重要性（要不然总是abort掉，你会很不开心）。这次实验给出的库函数是出色的，可以节省协议很多的开发时间，让我能专注于数据链路层的开发。本次实验也是第一次主动去了解库，包括静态、动态链接库等一系列的机制和编写方法。遗憾的是，未能见到这些库函数的源代码，如果出现对库函数不甚理解的情况，也许给出源代码可以加深对程序的理解。对于debug的方法和机制，本协议也提供了重要的参考价值，对以后的软件测试，我们可以引入mask等方法来获取我们想要的debug信息。

这次开发也为层次化开发打下基础：先开发简单的停等协议，其次是gobackN（此实验），最后是灵活性最大的选择重传协议（在另一份实验报告中体现）。循序渐进的方式让我加快了开发的速度，并从本质上理解了程序的要义和方法。