**计算机网络实验报告**

**选择重传协议**

**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，1e-5和1e-4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选择重传协议，采用了ACK搭载技术，实现了NAK否定确认(优化了nak机制，使得高误码率情况下效率优于示例数据6%-7%)、计时器逻辑更新、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（请注意，nak也可以携带ack字段，但由于采用了优化的nak技术，这里不再对nak帧进行piggybacking过程）。

seq为帧需要字段，占用一个字节，它的范围是从0至MAX\_SEQ，请注意，对于数据帧，seq的作用是其自身的标号，对于nak帧，seq标示了需要重传帧的seq（与优化的nak相配合），对于ack帧，它是无用的字段，默认置为0。

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 out\_buffer[MAX\_BUF][PKT\_LEN]; // buffer to send

static unsigned char in\_buffer[MAX\_BUF][PKT\_LEN]; // buffer to receive

static unsigned char nbuffered; // the use of buffers

static unsigned char frame\_expected = 0; // lower bound of receive window

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

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

static unsigned char too\_far = MAX\_BUF; // upper bound of receive window

int no\_nak[MAX\_ENABLE\_SEQ + 1]; // nak for every possible seq of frame

int arrv[MAX\_BUF]; // if the in\_buffer is in use , make sure put\_packet in order

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。

这里着重讲一下优化的nak机制，这是在高误码率（1e-4）使效率提升至78%-79%（优于示例数据6%-7%）的关键。首先我们考虑这样的一个问题，在单nak机制下，nak只为接收方的frame\_expected服务，即只为接收方最期待的那个帧服务，如果出错且没有该帧的nak流通，那么传送该帧的nak。

但请考虑一下如下情况：最期待帧序号为7，此时接收方连续收到了7/8/9/10四个数据帧，不幸的是，这四个帧全都是坏帧（这在1e-4误码率的信道上是常见的，这也是为什么采用这种优化在1e-5误码率信道上效率提升并不明显），请求重传7号帧的nak被激活，此时8/9/10无法请求重传，这时，重传的7号帧到来，但倒霉的是这个7号帧还是错的（高误码率很常见），发送方忍无可忍，终于7号帧超时。同时由于8/9/10的nak被7号帧屏蔽，它们只能干等，它们没办法发送属于自己的nak，是7号帧耽搁了它们的大好前程。最后的结果是一连串不必要的超时，降低了信道效率。这里问题的本质在于：虽然最期待帧是7，但是8/9/10却只能等到7接收正确后才能发送自己的错误报告。

最直观的解决方法是：为每一个可能的帧设置属于自己的nak。nak数组是一个好的方法，请注意，此时我们要“有针对性”地发送nak。简单地想，能不能接收到错误的数据帧后，提取这个数据帧的f.seq，然后发送该f.seq的nak就可以了，这也符合数组的直接索引特点。

但是这里有一个致命的问题：既然此数据帧是坏帧，f.seq很可能已被损坏。在原来的版本中这个问题不存在，因为只要收到坏帧，就发送最期待帧的nak，此时没有从坏帧中提取任何信息。请注意：直接从坏帧中提取数据是危险的。

但是这不代表不可行，理论上可以证明，我们最需要的信息f.seq如果损坏（几率很小，但在8小时连续运作下，基本上一定会发生），有两种可能：其一，这个序号被变为不可能序号，即超出MAX\_SEQ范围的序号，此时，这个帧可能引发between机制（注意，between机制没有超限检查，一个大序号是可能落在范围内的），因此我要做的只是简单的排除，在接受帧时添加如下过滤信息

if (f.seq > MAX\_SEQ)

break；

其二，f.seq被篡改为另一个不正确的但是在序号域内的序号。此时接收方向发送方发送一个不正确的nak请求，请求了一个自己根本不需要的帧。可以证明这个自己不需要的帧是无害的，窗口机制保证：如果这个帧可以被发送，那么它会被存入缓存，或者会因不在接收窗口中而被丢弃，不会出现新旧重叠的情况。而本来自己请求重传的帧最终会超时。

由于seq的筛选机制，ack的seq不可再设置为默认，虽然它本身的无用的，但如果随机成一个大数字，那么此ack会被过滤掉。因此，我们人为地设置任何ack的seq字段为0。

**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**

主程序分为五个部分，packet处理、帧的发送、定时器管理、判定窗口和校验。分别对应put\_packet(get\_packet),send\_frame, timer、between和crc32。

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，负责了定时器的管理。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，这个帧分为五大类：正常数据帧，单独ack帧，nak帧，因nak而重传的数据帧和因超时而重传的数据帧。

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。

**（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。首先按照先前阐述的优化nak机制，先判断序号是否合理，若不合理直接将该帧丢弃。如果该帧是数据帧，我们提供发送关于该帧的nak的可能，尽管它不是最期待帧。

如果它通过了CRC校验，它只可能是数据帧、ACK帧或NAK帧。若是数据帧，先判别它是否是我们最期待的帧，如果不是，它将可能被缓冲，并引发nak机制，接着检查是否有按序上交的可能，处理入境缓冲，直到最低位的缓冲区没有被占用，最后激活捎带应答过程。如果是ACK帧，处理捎带应答过程。如果是NAK帧，需判断对方请求重传的帧是否落在发送窗口内，如果是，则重发该帧。

这里为了保证程序的可读性，分别在data帧和ack帧中捎带应答。

1. **DATA\_TIMEOUT**

此时某数据帧超时。程序中的arg参数同时用于ack超时和数据帧超时，因此直接使用arg参数会造成误解。这里我们重写了定时器管理模块，用模拟队列的方式判断哪个数据帧最先超时，即这个帧是最早的未被确认的帧。我们要重传这个帧。

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。由于选择重传不像gobackN,并没有连续回传的过程，因此大一些的窗口几乎不会带来坏处。当误码率很高时，帧频繁出错且有超时时间的约束，nbuffer迅速被填满，此时发送方被阻塞，如果窗口大小调大一些，可以增加信道误码率较高时的效率。我们在这里选择窗口大小为32。

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。请注意，在选择重传协议中，超时是不利的因素：一个帧的超时重传有可能引发一连串的超时重传，因此缩小DATA\_TIMER往往在高误码率信道上是不明智的。但超时重传机制又是必须的，以防止nak出错、帧丢失等极端情况。

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。在本次实验模型下，有使用了改进的nak机制，使得每一帧都可以独立、多次（连续出错的情况）发送nak，DATA\_TIMER一定程度上限制了给定某个帧传nak次数的“机会”，这个机会应该尽可能多，以减少超时带来的影响。经试验，设置为5000ms获得了较好的效果。显然地，这个参数的选择与误码率有关，如果误码率增大，那么这个“机会”应该给予地更多。

值得注意的是，如果增加了给予nak的“机会”（即增大DATA\_TIMER），将使nak变得活跃起来，大部分帧选择错误时传送nak，单独的nak帧也是浪费信道资源的一部分，但实际证明，这种方法的自适应性良好，即：比起传送独立nak，要比大量的超时重传要好得多；同样地，测试中也体现了这种方法的低误码率兼容性：会显著提升高误码率的效率，而同时几乎不影响地误码率的效率。

1. **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。但过小的ACK\_TIMER是不明智的，从稍后的极限效率分析中可以看出，单独的ack是浪费信道的因素。经试验，设置为300ms获得了较好的效果。

**（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)

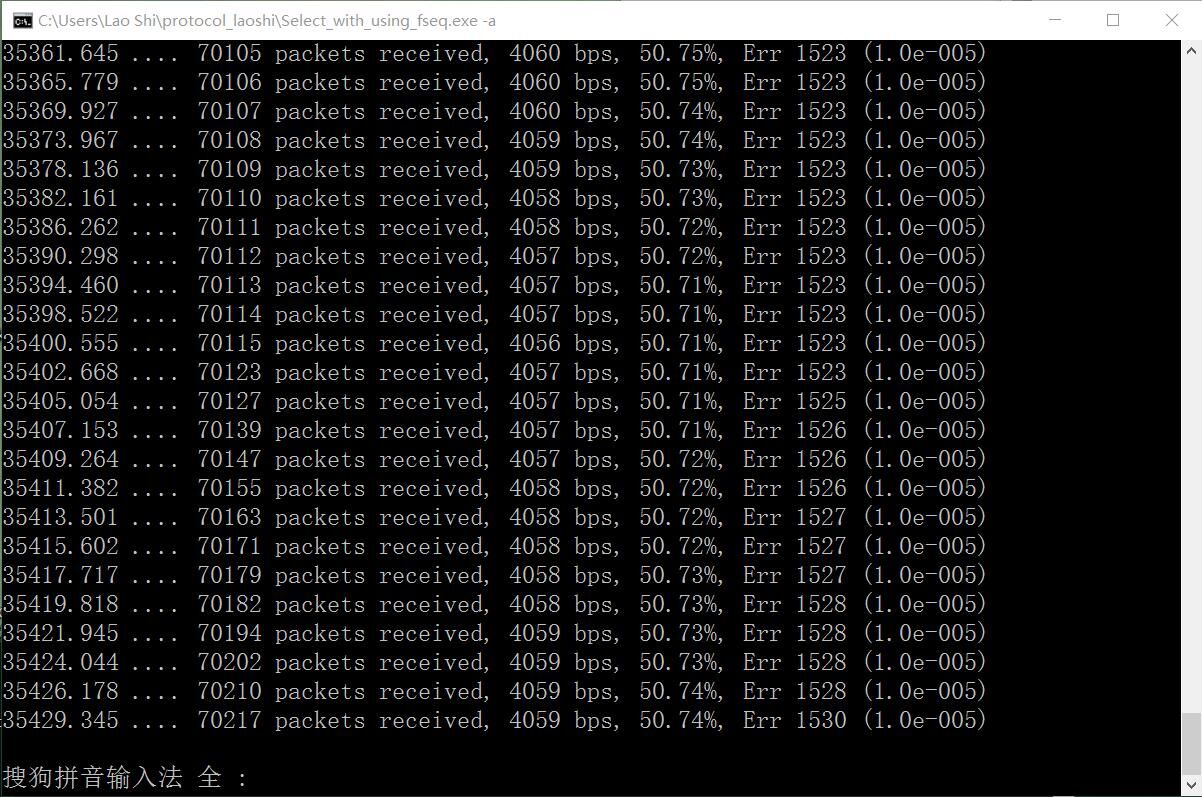
带入数据可得，在误码率为1e-5的情况下，η≈95.34%

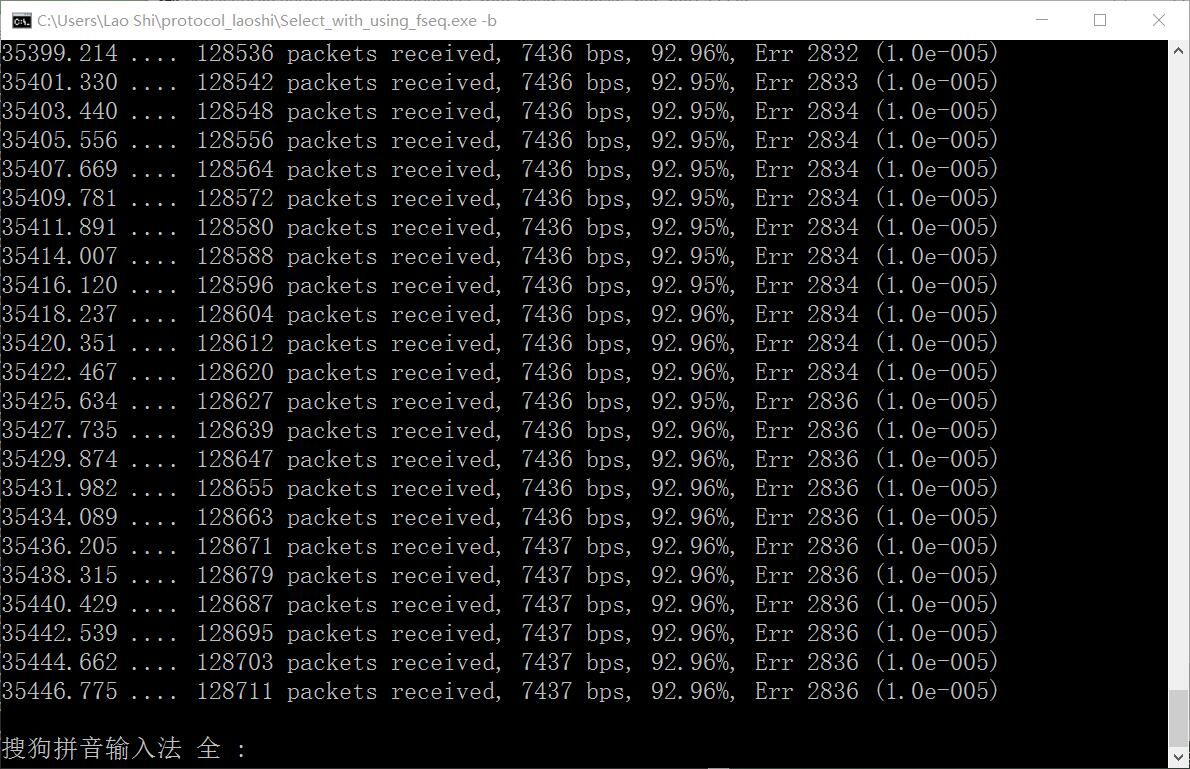
**（四）、实验结果分析**

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。

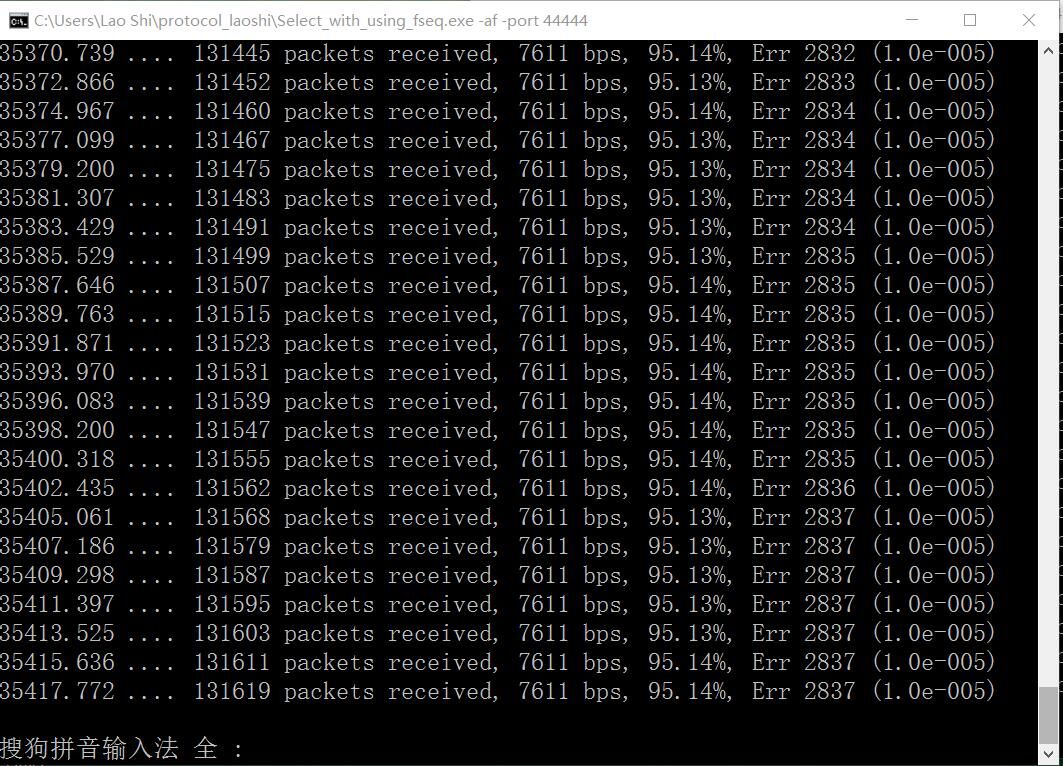
注：命令行命令显示在对话框的标题位置。

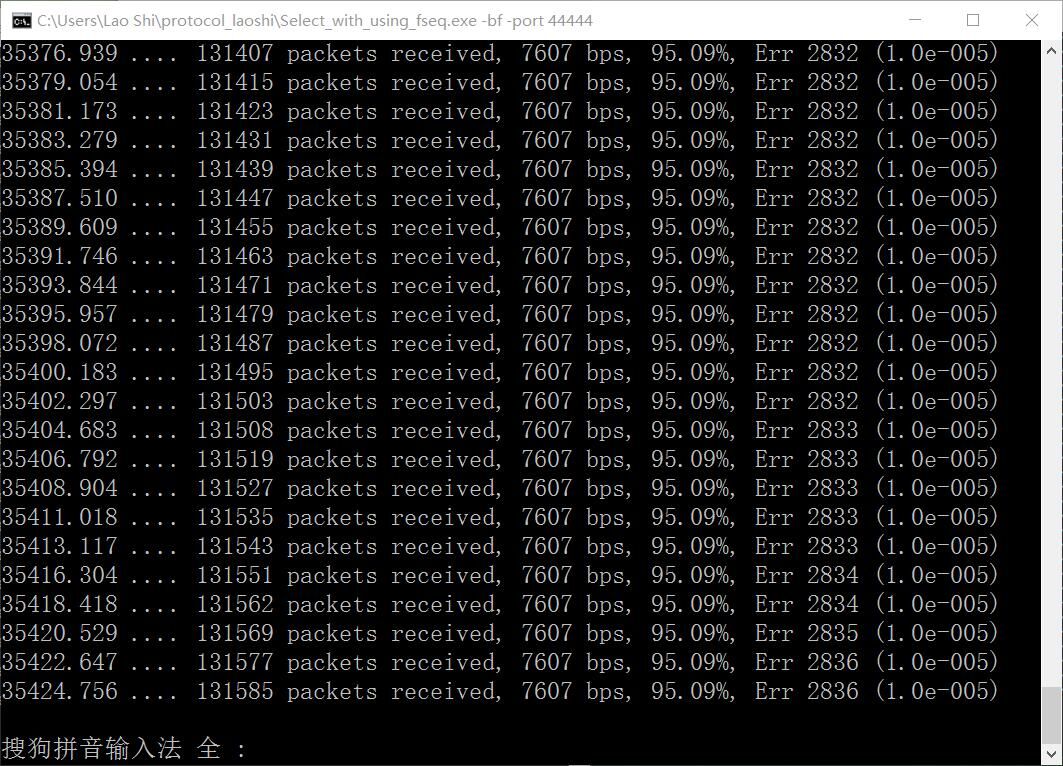
1. **a模式**



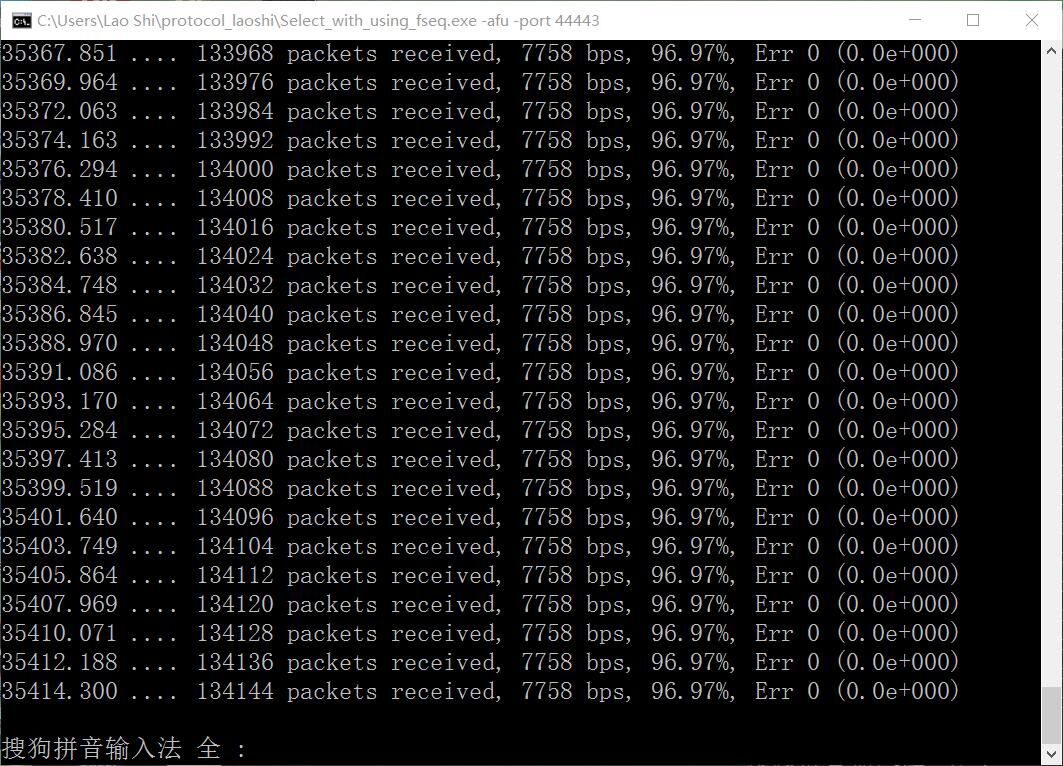


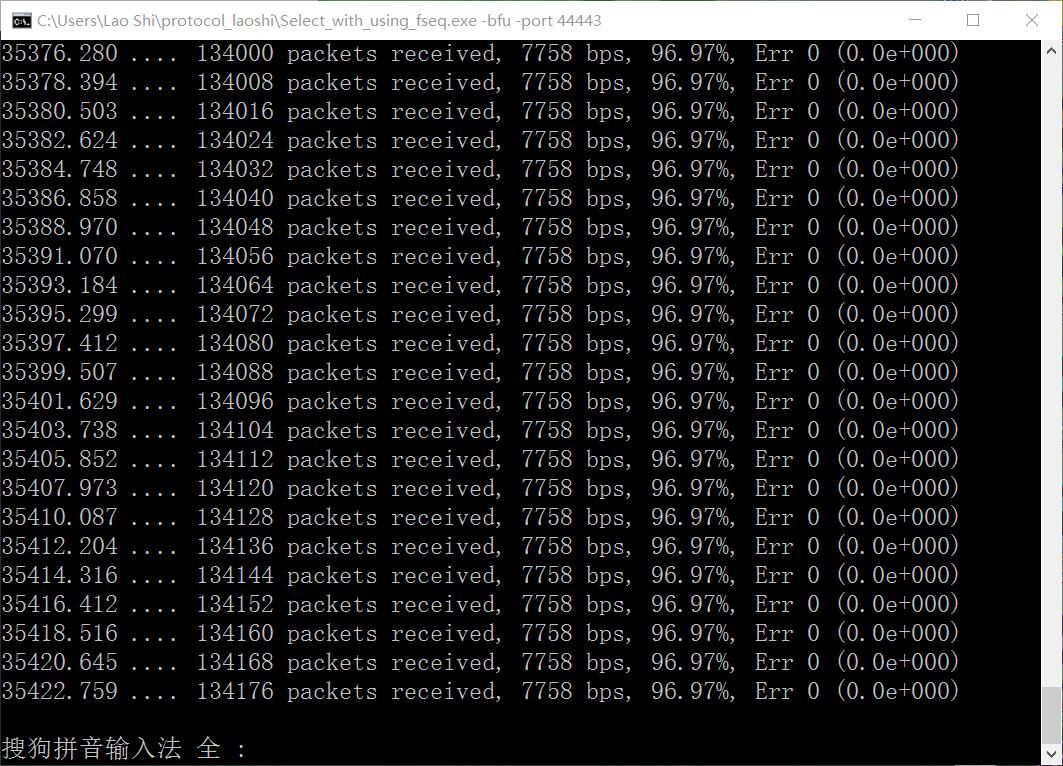
1. **af模式**





1. **afu模式**

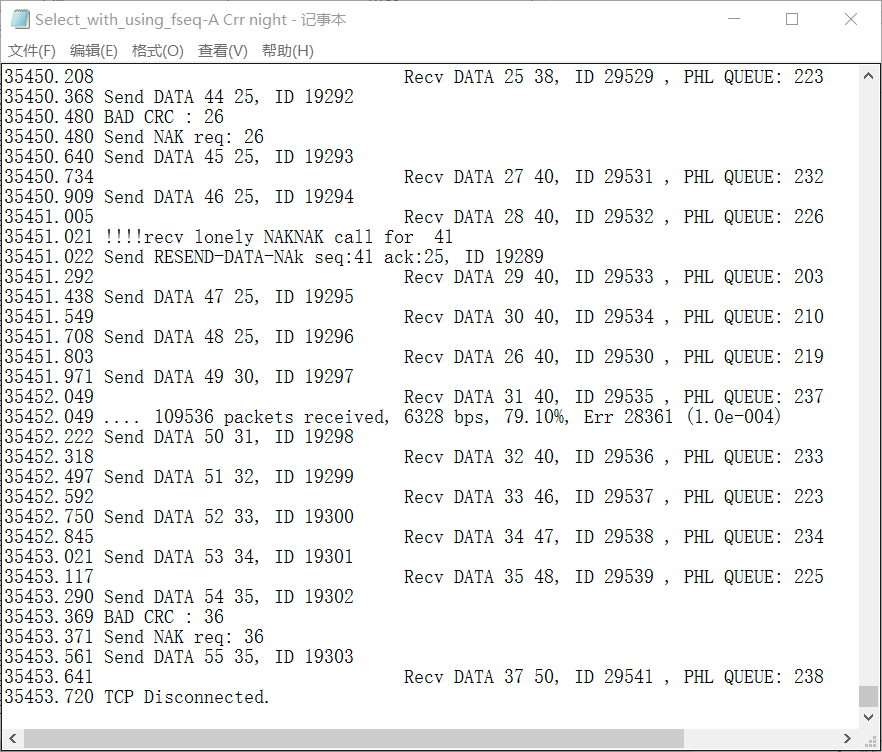




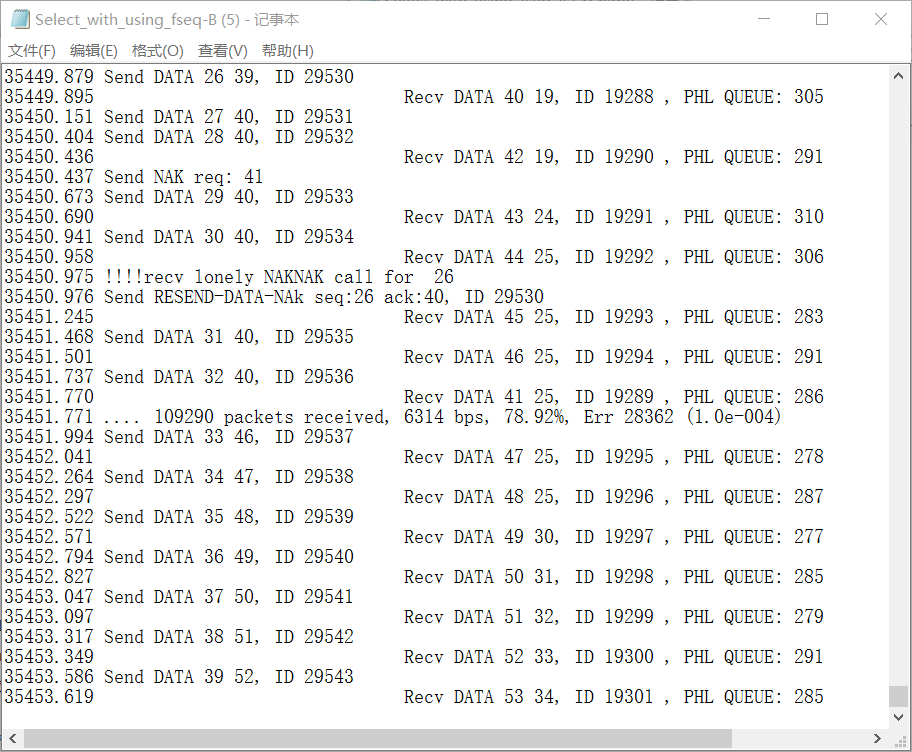
1. **af ber 1e-4模式**

用文档截图的形式给出。

**A方：**



**B方：**



由于采用了改进的nak机制，使得在高误码率情况下（1e-4）效率与示例数据相比有较为显著的提升，基本可以稳定在78-79%。其他方面的数据与示例差别也不大。

改进的nak机制主要体现在高误码率模式下，可以从局部看出，这种方法大大减少了DATA\_TIMER对协议效率的影响，使得传输过程更加平滑、高效。

1. **、存在的问题**

协议未失败，由于采用了改进的nak机制，效率可以超过示例数据。

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。

**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3个小时（因为在此之前开发了gobackN，因此选择重传开发过程比较顺利）。

下面列出几个bug示例：

1. 设计帧结构时，第一次没有给CRC留出位置，导致缓冲区溢出。
2. 在设计out\_buffer时，地址引用设置成了从0-MAX\_SEQ均可用，但实际上只用设计一半的容量就可以满足要求。
3. 对于改进的nak机制，虽然可以提升效率，但开发过程中还是遇到了不小的问题，从坏帧中提取信息本来就是一件有风险的事，除非做了足够的容错。具体方法在介绍nak数组时已经详述。

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。但在选择重传中，由于不是按序接收帧，那么任何一个帧都有可能引发超时，这导致判断哪个帧超时更加复杂。但幸运的是，自己编写模拟队列函数可以正确运行。

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。

最重要的是，示例数据给了我击败它的勇气和决心。当我发现第一次开发出来的选择重传协议版本（思路与书上类似）的高误码率效率仅有62%左右时，我第一个反应是先询问周边同学，得知它们也差不太多，但是这个数据离示例数据仍有差距。于是我开始思考改进的方式。有趣的是我的室友和我一起思考这个问题，最终提出了一个新的改进方案：nak不应该受到制约，开放的nak制度会减少不必要的重传，有必要将nak泛化，为每一个错误的数据帧服务。这个思想先是在他的方案下得到实现，使效率提升至74%，已略微超过示例数据，随后又被我改进至78%。

再者，它确实锻炼了编码者的独立思考能力，如：nak的引入一定是合适的吗（对于gobackN来讲）？窗口大小一定是一个2的指数形式吗？Ws（在选择重传中）理论上有上限，在未达到上限的时候，加大窗口宽度的同时也引入了新的bit占用，这会导致data字段占的比重下降，但是这种效率的降低比起窗口大小不足来说是微不足道的。之类的一些列的问题，或多或少的引起了我对于书本之外、课堂之外的知识的研究和思考。

其次，我意识到了开发层次的重要性，由于之前有gobackN的铺垫，选择重传的开发过程变得更加高效，仅需要考虑按序上交、定时器管理等新多出来的问题，协议的主体结构未曾改变。同样地，对于gobackN，也有停-等的协议为之铺垫，学习理应由易到难，这样的开发过程如顺水推舟，让人们从最简单、本质的东西做起。