**计算机网络实验一**

**——数据链路层滑动窗口协议的设计与实现**

**班级：2014211302**

**学号：2014211168**

**班内序号：14**

**姓名：周尧棋**

**目录**

一、实验内容和实验环境描述

1.实验内容和目的

2.实验环境

二、软件设计

1.数据结构

2.模块结构

3.算法流程

三、实验结果分析

1.性能测试记录表

2.截图

四、研究和探索的问题

五、实验总结和心得体会

六、源程序清单

**一、实验内容和实验环境描述**

1.实验内容和目的

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。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机理。滑动窗口机制的两个主要目标：(1) 实现有噪音信道环境下的无差错传输; (2)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，计算出最优取值，并通过程序的运行进行验证。

通过该实验提高同学的编程能力和实践动手能力，体验协议软件在设计上各种问题和调试难度，设计在运行期可跟踪分析协议工作过程的协议软件，巩固和深刻理解理论知识并利用这些知识对系统进行优化，对实际系统中的协议分层和协议软件的设计与实现有基本的认识。

2.实验环境

WindowsXP环境PC机，Microsoft Visual C++ 6.0 集成化开发环境。

**二、软件设计**

1.数据结构

(1)选择重传协议

#define MAX\_SEQ 31 //定义最大窗口序列

#define NR\_BUFS ((MAX\_SEQ+1)/2) //定义窗口尺寸

#define DATA\_TIMER 5000 //定义超时重发定时器时间

#define ACK\_TIMER 280 //定义ACK定时器超时重发时间

#define DATA 1

#define ACK 2

#define NAK 3

struct frame{

unsigned char kind ; //数据的类型

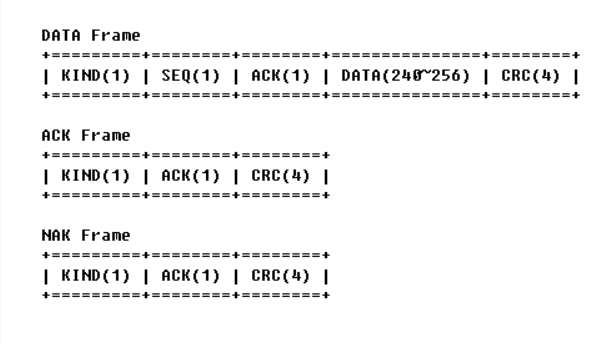
unsigned char ack ; //确认帧

unsigned char seq ; //帧的序列号

unsigned char info[PKT\_LEN] ; //数据内容

unsigned int padding;

} ;



KIND：表示帧的类别 ACK：表示ACK序列号 SEQ：表示帧序列号 CRC：校验和

static int no\_nak = 1 ; //标志是否已经发送过nak

static int phl\_ready = 0 ; //物理层就绪标志

int event, arg ; //event表示接受方等待的发生事件

int len = 0 ; //len表示帧的长度

int i ; //i为循环中使用的表量

struct frame f ; //表示变量帧

static unsigned char ack\_expected = 0, next\_frame\_to\_send = 0 ;

//ack\_expected表示该站期望收到的确认帧，next\_frame\_to\_send表示将要发送的帧的序号

static unsigned char frame\_expected = 0, too\_far = NR\_BUFS ;

//frame\_expected表示该站期望收到的下一帧

static unsigned char nbuffered = 0 ;

//表示已发送但未收到确认的帧的数量

int arrived[NR\_BUFS] ;

//缓存区标记

unsigned char in\_buf[NR\_BUFS][PKT\_LEN], out\_buf[NR\_BUFS][PKT\_LEN] ;

//缓冲区，用来存放网络层分组来的分组或接受到的分组

(2)GobackN协议

#define MAX\_SEQ 7

#define DATA\_TIMER 3000

#define ACK\_TIMER 270

#define DATA 1

#define ACK 2

struct frame{

unsigned char kind ;

unsigned char ack ;

unsigned char seq ;

unsigned char info[PKT\_LEN] ;

unsigned int padding;

} ;

static int phl\_ready = 0 ;

int event, arg ;

int len = 0 ;

static unsigned char i ;

struct frame f ;

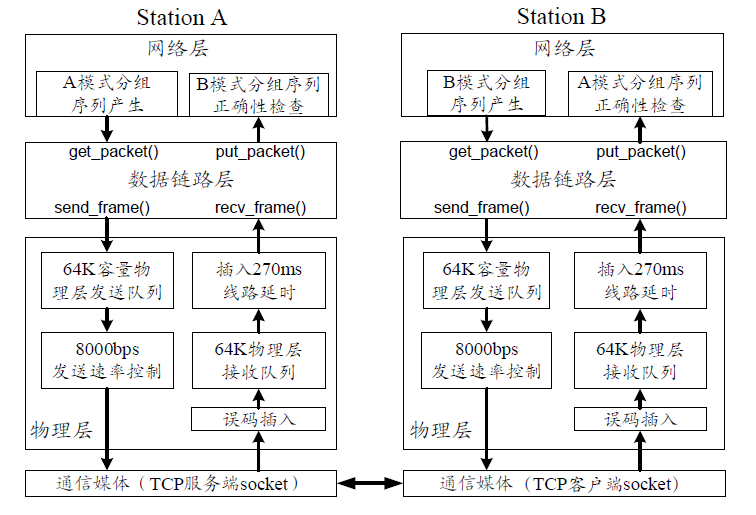
static unsigned char ack\_expected = 0, next\_frame\_to\_send = 0 ;

static unsigned char frame\_expected = 0 ;

static unsigned char nbuffered = 0 ;

unsigned char buffer[MAX\_SEQ + 1][PKT\_LEN] ;

2.模块结构



datalink

get\_packet

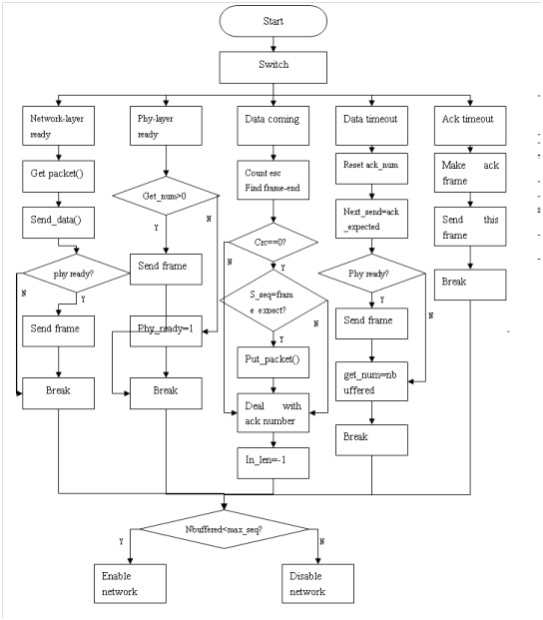
send\_data\_frame

send\_frame

between

crc32

put\_packet



static int between(unsigned char a, unsigned char b, unsigned char c)

作用：窗口函数，判断当前帧是否落在发送/接收窗口内。

参数：a为窗口下限，b为帧号，c为窗口上限。发送窗口的上界和下界分别为next\_frame\_to\_send和ack\_expected，接收窗口的上界和下界分别为too\_far和frame\_expected。

static void put\_frame(unsigned char \*frame, int len)

作用：为一个帧做CRC校验，填充至帧的尾部并将其递交给网络层发送

参数：frame，字符数组，由除padding域之外的帧内容转换而来。len，整型，为帧的当前长度

static void send\_data\_frame(unsigned char fk, unsigned char frame\_nr, unsigned char frame\_expected, unsigned char buffer[MAX\_SEQ + 1][PKT\_LEN] )

作用：构造一个帧，并将其发送

参数：fk，表示帧的类型；next\_frame\_to\_send，为帧的序号；frame\_expected，为希望收到的帧的序号；buffer[][PKT\_LEN]，二维数组，为缓冲区内容。

int main(int argc, char \*\*argv)

作用：主函数，包含选择重传/GobackN的算法流程

参数：argc，整型，表示命令行参数的个数；argv，二维字符数组，表示参数的内容。

**三、实验结果分析**

1.软件协议能实现有误码信道环境中无差错传输功能，采用了CRC校验和重传技术使得错误能被发现并纠正。

2.程序的健壮性较好，在高负荷和高误码率等条件下能够长时间工作。

3.协议的参数选取

(1)滑动窗口的大小

滑动窗口的选择涉及信道利用率和数据拥塞的问题，若窗口太小，会导致信道利用率很低，信道中长时间没有数据传送；若窗口太大，数据发送过快，会造成接收方数据链路层来不及处理，数据物理层及信道发生拥塞现象导致数据丢失，出错率增大。

假设接收方一直有数据发送，即无ack定时器超时现象。设滑动窗口的大小为M，信道传输时延为Tf，发送速率为C，帧大小为L，信道利用率为M\*(L/C)/[2Tf+2(L/C)]接近100%。则M>=[2Tf+2(L/C)]/ (L/C)。由于实际信道利用率不可能达到100%，但是M的选择要满足上式。

a.GobackN协议

在GobackN协议中，因为信道速率为8000bps，信道传输时延为270ms，网络层分组大小为256字节由上述一个数据帧加上帧头3字节以及校验和4字节后为263字节，数据发送时间Td=L/C=263\*8/8000=263ms，由上述分析得，当M=7时即可保证在理想情况下信道一直被使用且不会出现拥塞。

b.选择重传协议

窗口大小的分析同GobackN协议，根据实际测试MAX\_SEQ的值最佳设为31。

(2) ACK搭载定时器的时限

若ACK搭载定时器的时限太小，会导致确认一直使用ack帧，会降低信道的利用率；若时限太大，则会使接收方长时间得不到确认而重传数据帧，也会降低信道利用率。

首先ACK\_TIMER的值需要大于数据帧发送时间263ms，若接收方在接收到一帧后马上发送一帧，则ACK\_TIMER的值要大于发送时间，否则一帧还未发完便进入了超时状态。其次还要满足 DATA\_TIMER>2(Td+Tf)+ACK\_TIMER。

经过实际测试，GobackN的ACK\_TIMER设为270ms时比较好；选择重传的ACK\_TIMER设为280ms时效果较好。

(3) 重传定时器的时限

若重传时间选择过大，会导致重传等待的时间过长；若时间选择得较小，则会导致较为频繁地重传，这两种情况都会导致信道利用率的下降。

当发送方送完一个数据帧后，若接收方一直没有分组下来，而是利用单独的ack给发送方确定信号，则一个数据帧被确认的时间为270\*2+263+300+6=1109ms，因此一个帧最长经确认时间为1109ms，重传定时要大于此值。经过实际测试，将GobackN的DATA\_TIMER设为3000ms时比较好；选择重传的DATA\_TIMER设为5000ms时效果较好。

4.理论分析

根据所设计的滑动窗口工作机制(Go-Back-N 或者选择重传)，推导出在无差错信道环境下分组层能获得的最大信道利用率；推导出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。给出理论推导过程。理论推导的目的是得到信道利用率的极限数据。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，比如：假定超时重传的数据帧的回馈ACK 帧可以100%正确传输，但是简化问题分析的这些假设必须不会对整个结论产生较大的误差。

由于需要携带帧讯息，所以最大的信息利用率约为256/(256+3+4)\*100%=97.3%。又因为信道的最大比特率为8000bps，所以每传输一个字节耗时1ms，每帧的附加讯息固定为7，耗时7ms，如果出现转义字符，可能增加时间。假设信道上始终有数据需要传送，在10^(-5)错误率的信道上，在100000个比特中可以传送100000/(263\*8)=47个数据包，则每47个数据包将有一个出错，假设每出错一次，在限定时间内可以重传该帧为正确帧，则每传送47个数据包需要传输47+1=48次，此时信道利用率为47\*256/(48\*263+7)\*100%=95.3%

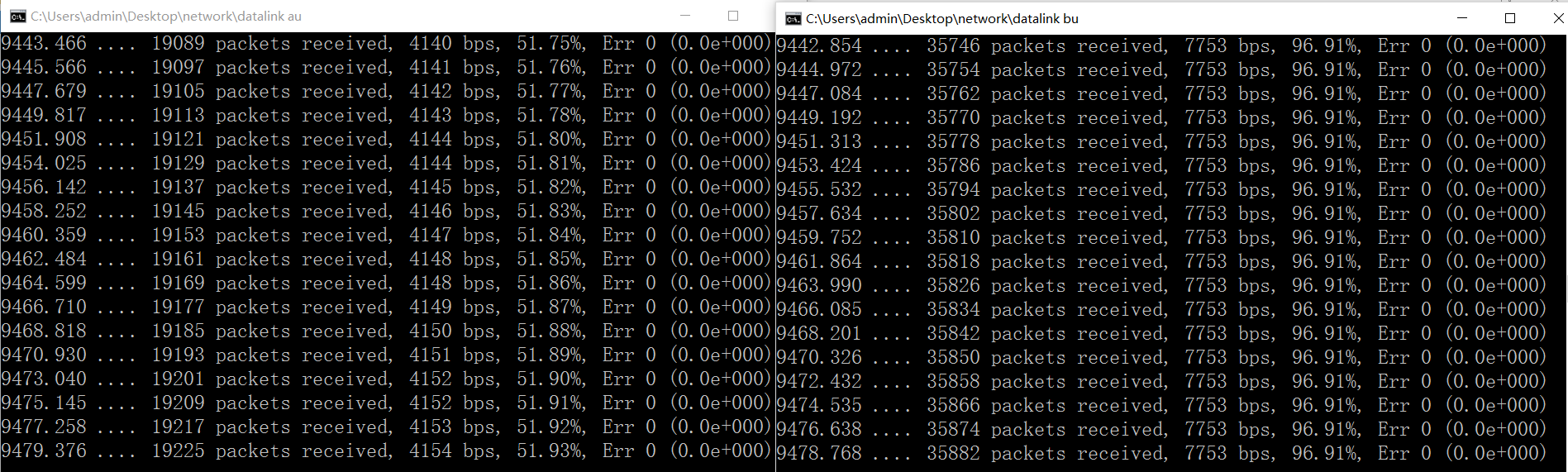
5.实验结果分析

(1) GobackN协议

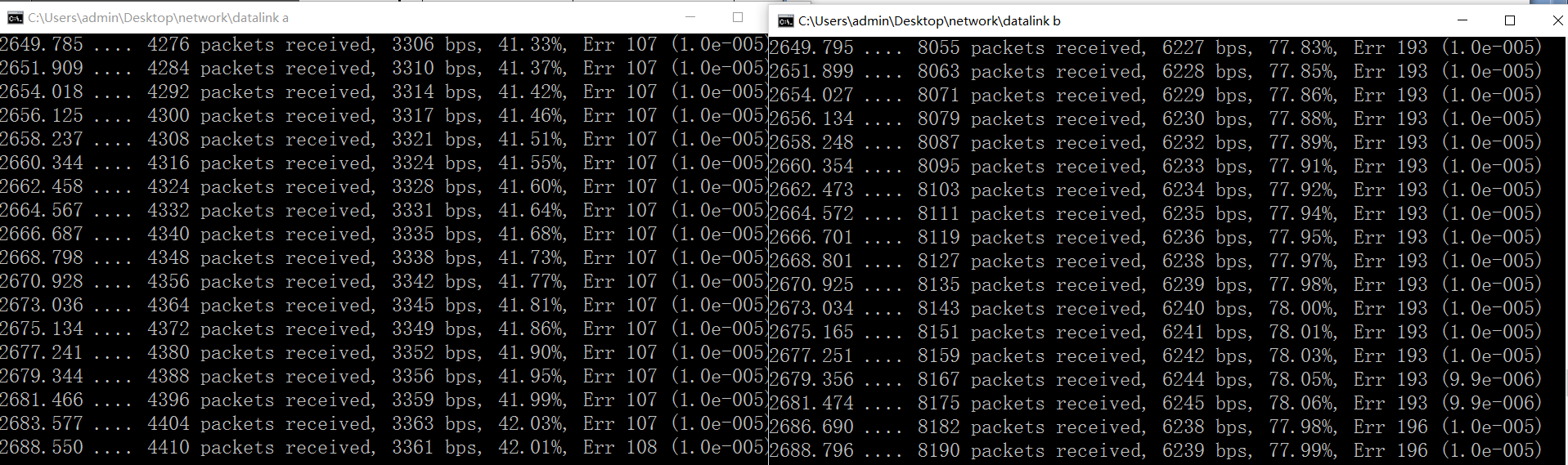
#### 性能测试记录表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 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 | 9447 | 51.93 | 96.91 |  |
| 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” | 2688 | 42.03 | 78.06 |  |
| 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 | 2195 | 96.77 | 96.77 |  |
| 4 | datalink af  datalink bf | 站点A/B的分组层都洪水式产生分组 | 1917 | 76.10 | 76.14 |  |
| 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 | 3092 | 24.86 | 26.40 |  |

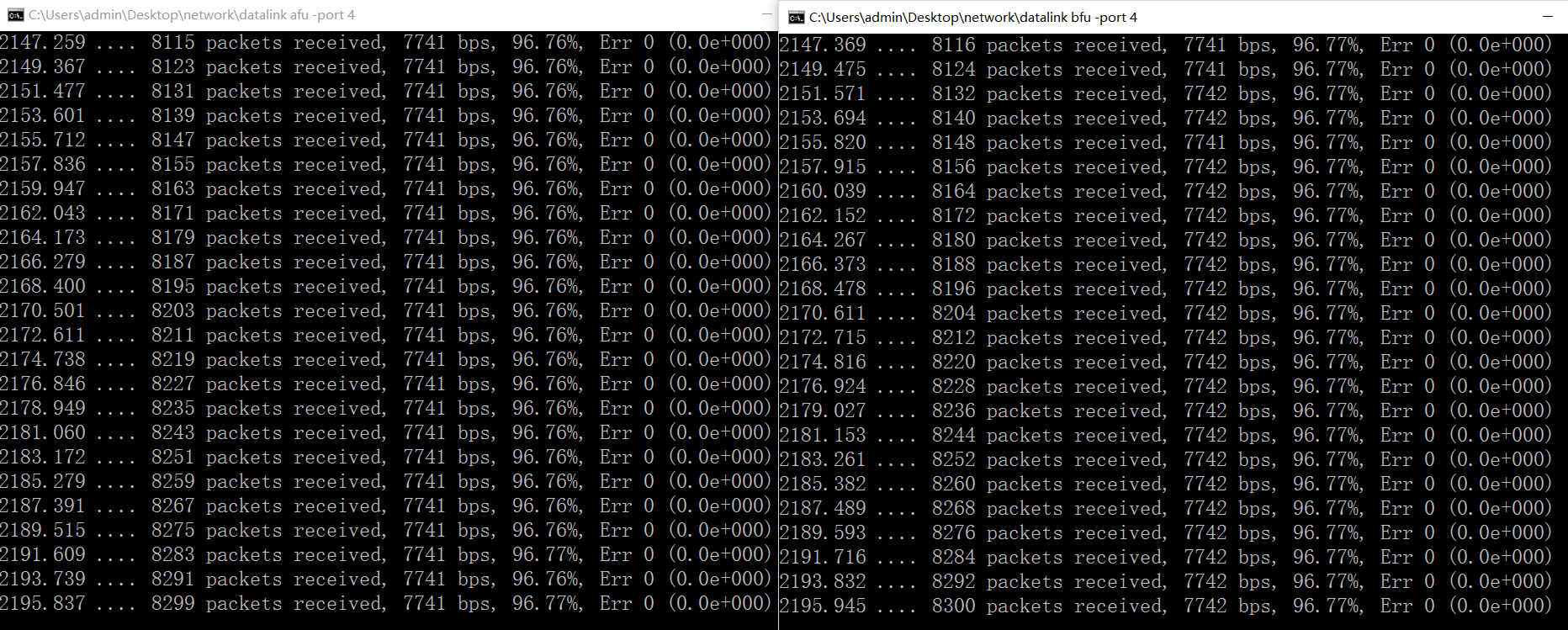
**au、bu**



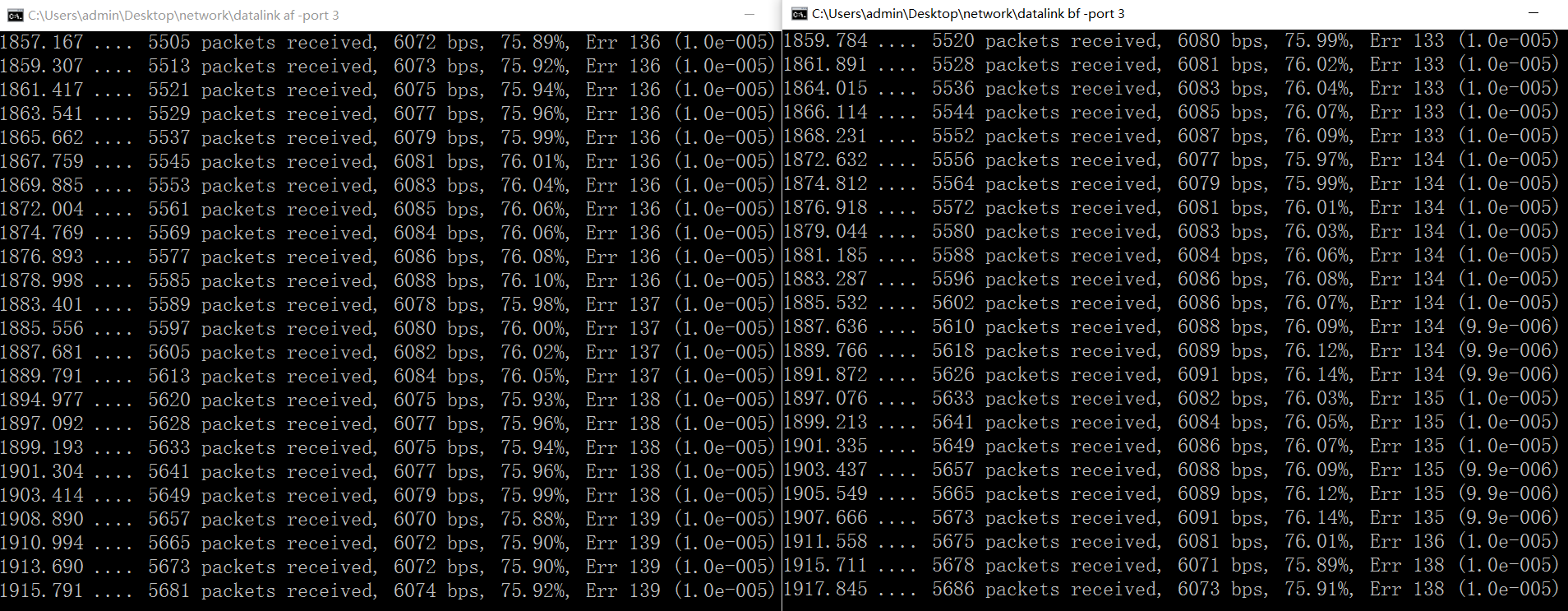
**a、b**



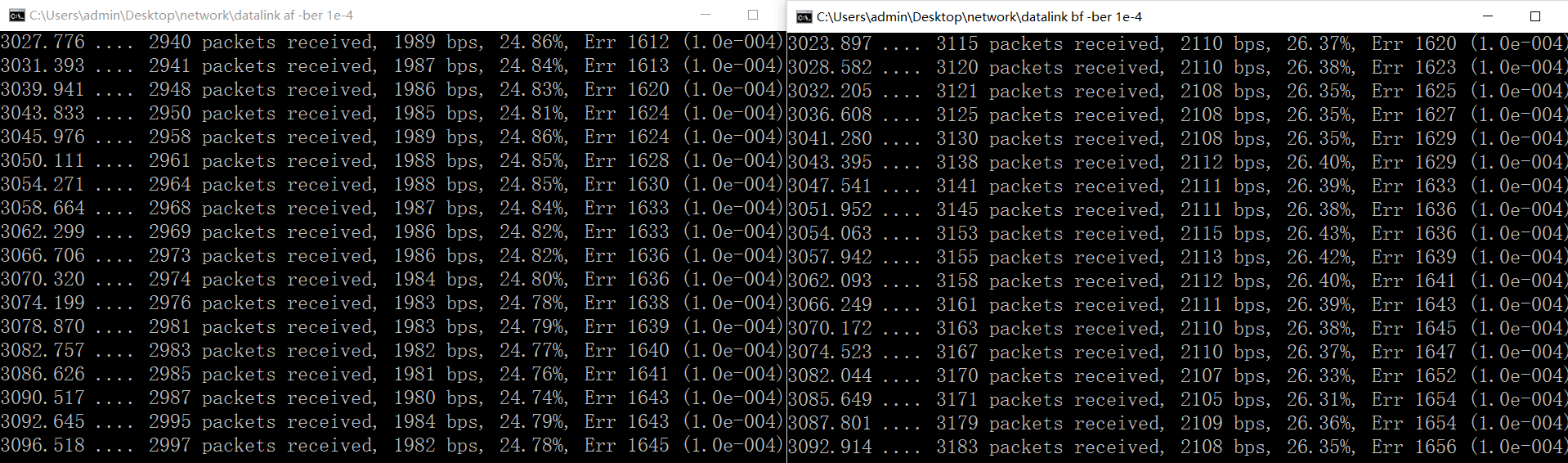
**afu、bfu**

****

**af、bf**

****

**af -ber 1e-4 bf -ber 1e-4**



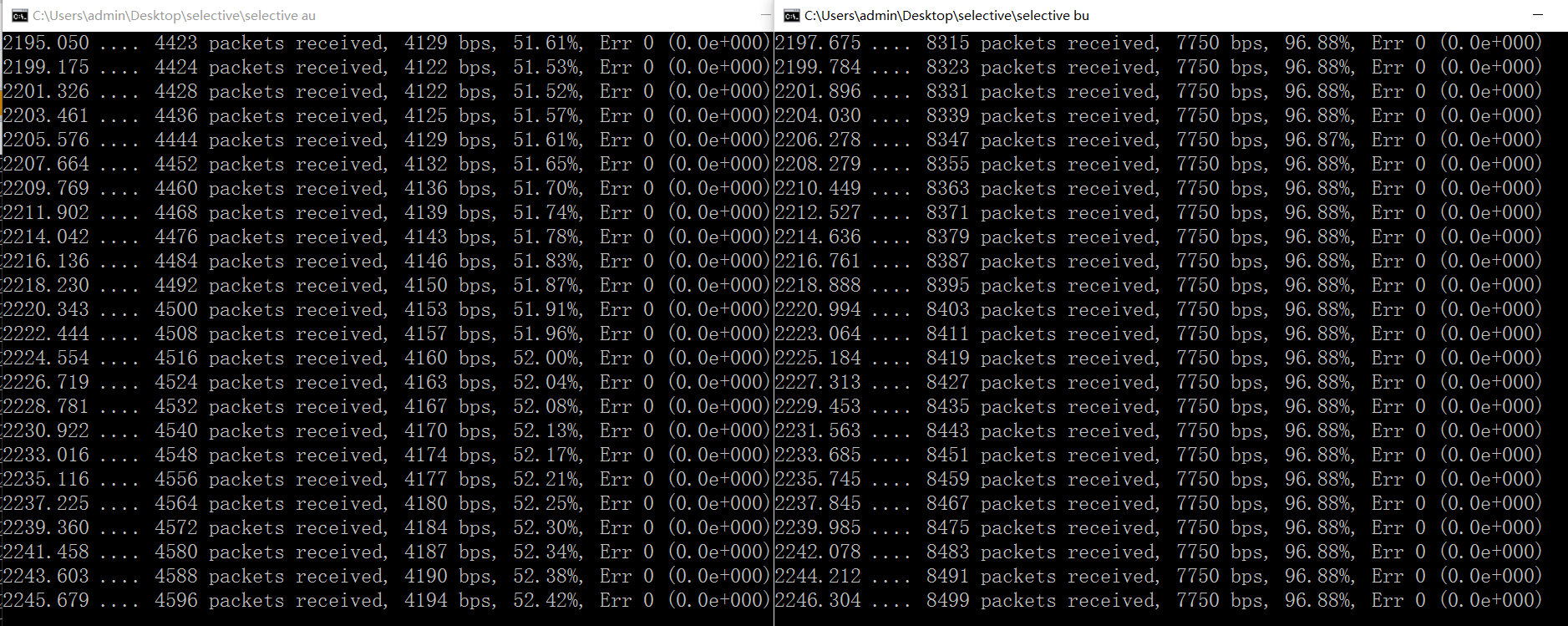
(2)选择重传协议

#### 性能测试记录表

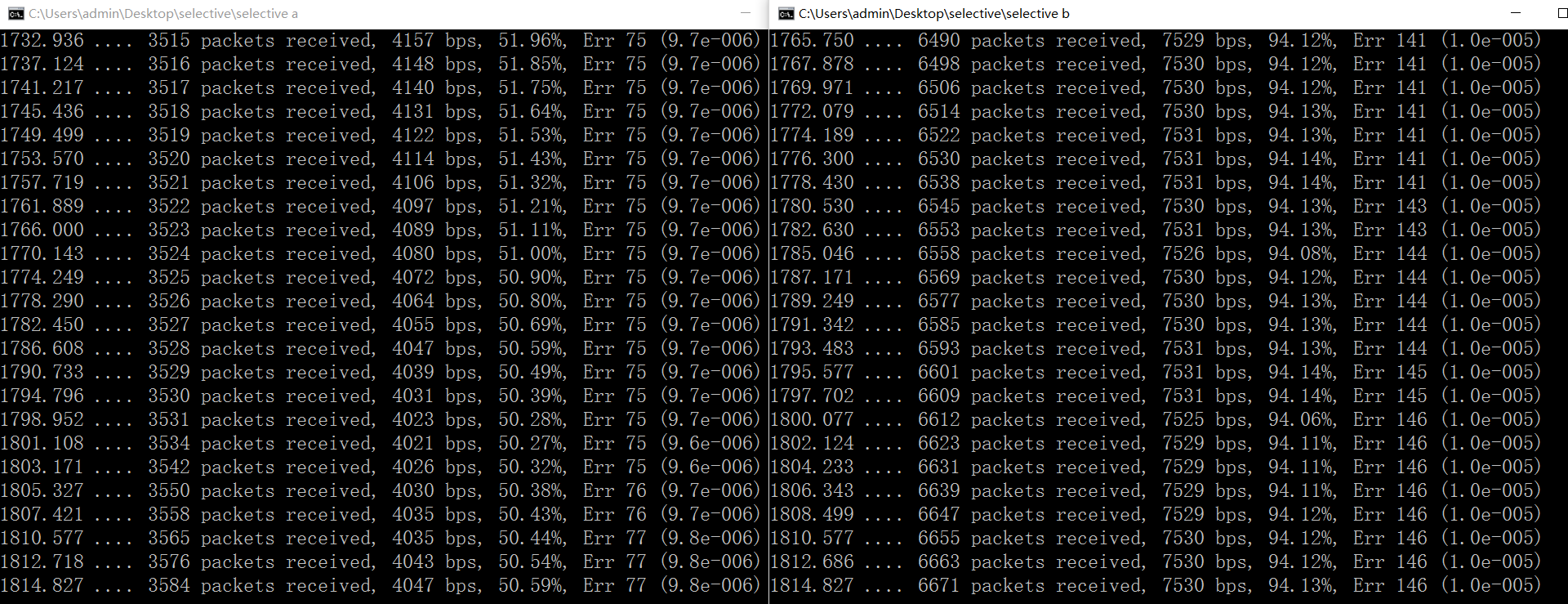
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 序号 | 命令 | 说明 | 运行时间(秒） | 效率(%) | | 备注 |
| A | B |
| 1 | datalink au  datalink bu | 无误码信道数据传输 | 2243 | 52.42 | 96.88 |  |
| 2 | datalink a  datalink b | 站点A分组层平缓方式发出数据，站点B周期性交替“发送100秒，停发100秒” | 1814 | 51.96 | 94.14 |  |
| 3 | datalink 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 | 1851 | 96.89 | 96.89 |  |
| 4 | datalink af  datalink bf | 站点A/B的分组层都洪水式产生分组 | 2288 | 93.01 | 93.26 |  |
| 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 | 2115 | 58.83 | 58.07 |  |

**选择重传协议**

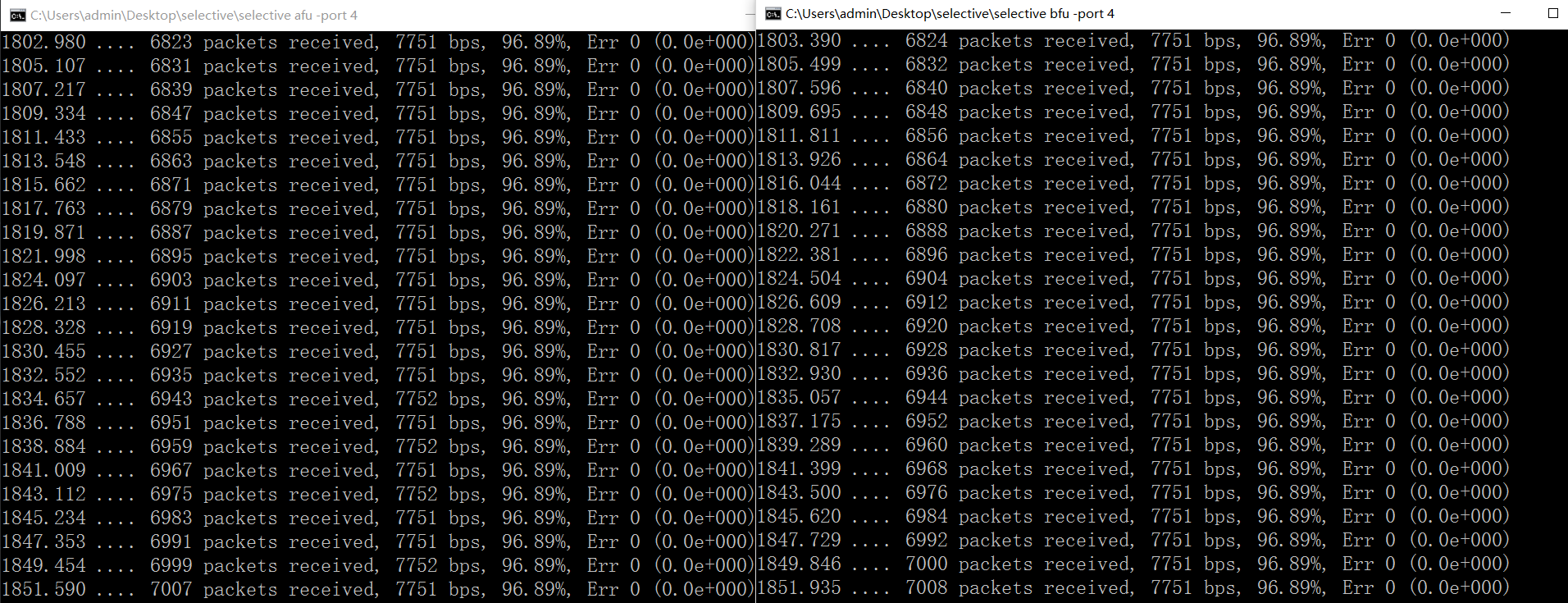
**au、bu**

****

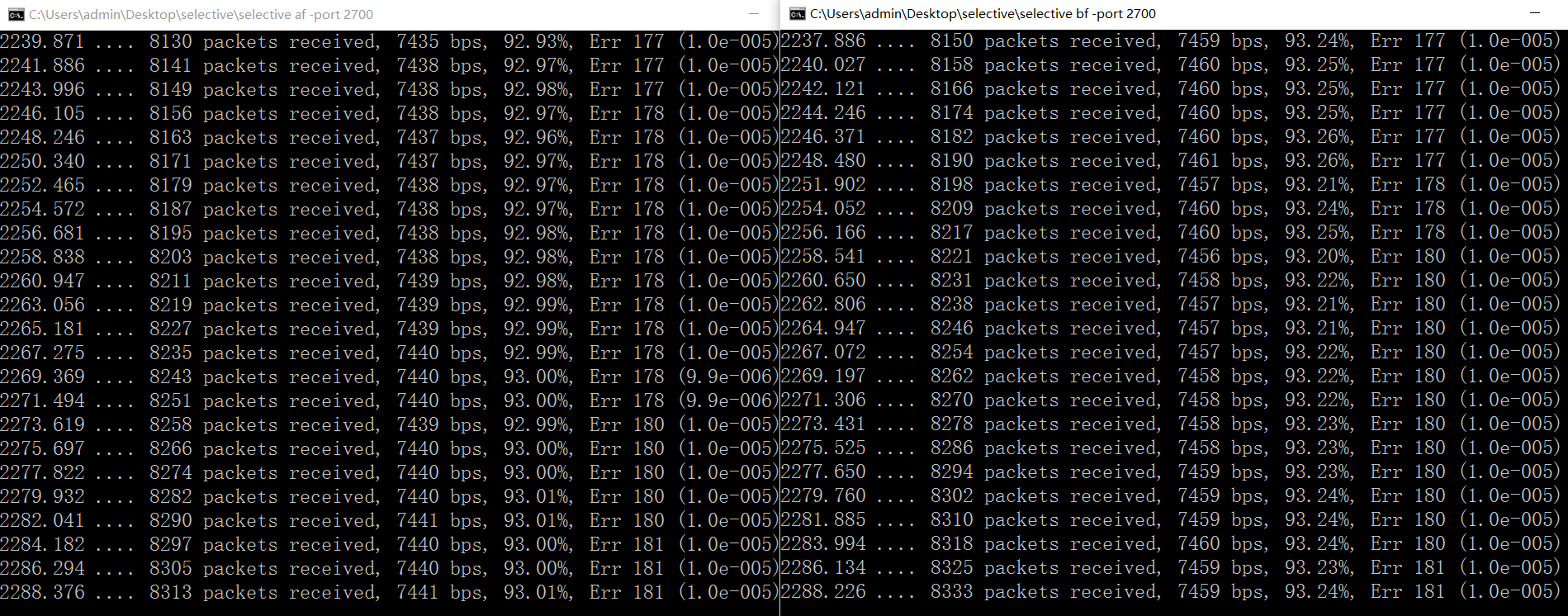
**a、b**

****

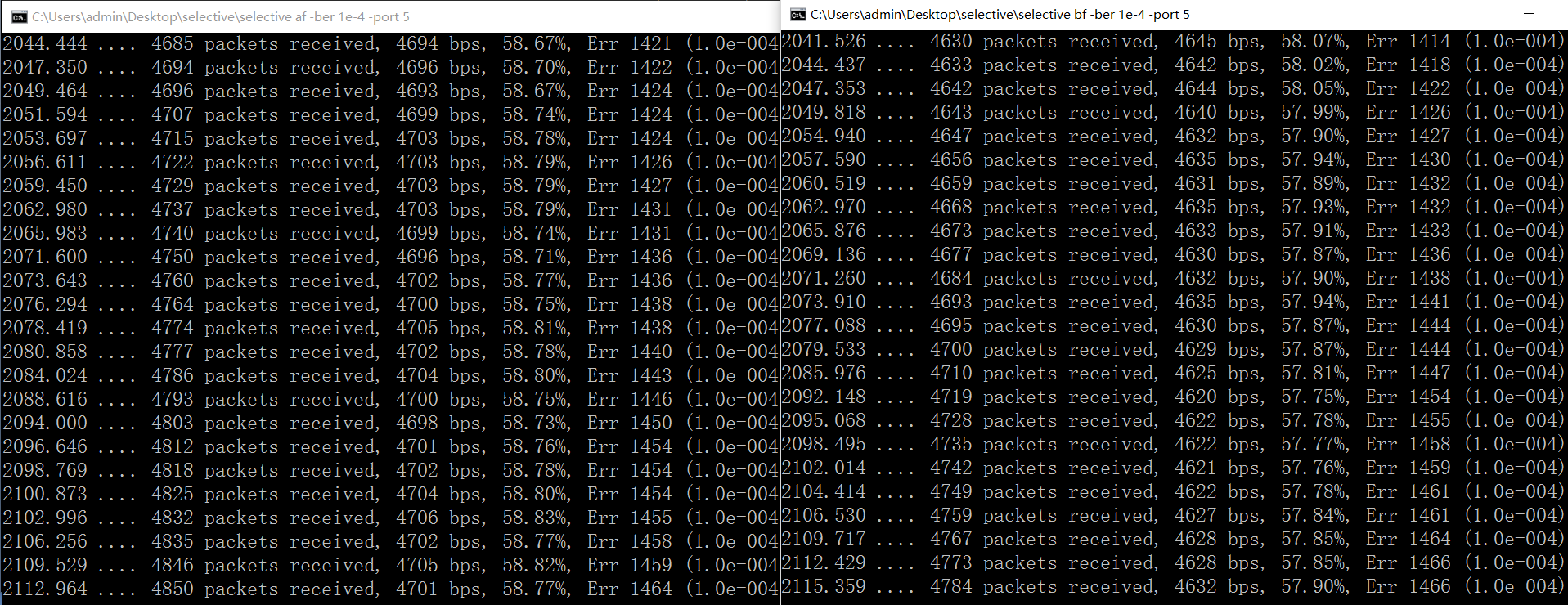
**afu、bfu**

****

**af、bf**

****

**af -ber 1e-4 、bf -ber 1e-4**

****

6.

选择重传协议在无误码下效率为97%左右，在误码率为1e-5的情况下效率为94%左右，与理想值还是有所差距但是相差不大，基本达到目标；GobackN协议在无误码下效率为97%左右，在误码率为1e-5的情况下效率为78%左右，与理想值差距较大，在误码率为1e-4的情况下差距更大。由于GobackN协议未使用NAK确认帧且做了捎带确认的假设，因此需要大量时间等待，所以应该在窗口大小和ACK\_TIMER的设置上加以改进并加上NAK确认帧的使用，以提高效率。

**四、研究和探索的问题**

**1.CRC校验能力**

Q：假设本次实验中所设计的协议用于建设一个通信系统。这种“在有误码的信道上实现无差错传输”的功能听起来很不错，但是后来该客户听说CRC 校验理论上不可能100%检出所有错误。这的确是事实。你怎样说服他相信你的系统能够实现无差错传输？如果传输一个分组途中出错却不能被接收端发现，算作一次分组层误码。该客户使用本次实验描述的信道，客户的通信系统每天的使用率50%，即：每天只有一半的时间在传输数据，那么，根据你对CRC32 检错能力的理解，发生一次分组层误码事件，平均需要多少年？从因特网或其他参考书查找相关材料，看看CRC32 有没有充分考虑线路误码的概率模型，实际校验能力到底怎样。你的推算是过于保守了还是夸大了实际性能？如果你给客户的回答不能让他满意这种分组层误码率，你还有什么措施降低发生分组层误码事件的概率，这些措施需要什么代价？

A：CRC校验码的检错能力很强，它除了能检查出离散错误外，还能检查出突发错误，CRC校验码具有以下检错能力：CRC校验码能检查出全部单个错误；CRC校验码能检查出全部离散的二位错误；CRC校验码能检查出全部奇数个错误；CRC校验码能检查出全部长度小于或等于k位的突发错误；CRC校验码能以[1-(1/2)^(k-1)]的概率检查出长度为(k+1)位的突发错误。

理论上，CRC32出错的概率最大为1/2^32，校验出错的概率约为0.0047%，又信道误码率为10^(-5)，则产生一个分组层错误的概率为p=4.7\*10^(-10)，由几何概率期望公式，需要1/p=2127659574次传输出现一次分组层误码事件。假设客户每天传输n次，则需要2127659574/n\*365=5829204/n年才会出现一次分组层误码事件。

如果客户不满意这种分组层误码率，则可以通过增加校验位以增加检错性能，但是这样降低了信息传输的效率并且对硬件的要求增高，需要更多的寄存器。

**3.程序设计方面的问题**

**Q：**

(1)8.10 节提出的协议软件的跟踪功能有什么意义？你的程序实现这样的功能了吗？程序库中获取时间坐标的函数get\_ms()不是C语言标准库中的函数，你能自己实现一个这样的函数吗？在“C 语言程序设计”课程中学习过printf 函数，但是教科书中可能没有介绍怎样设计这样的函数。printf 风格的函数特点是参数的数目是不确定的，类似的标准库函数还有fprintf，sprintf，scanf等。为了便于同学调试程序，程序库中提供了日志文件和屏幕窗口同步输出的log\_printf 和lprintf 函数(参见8.3)，这些函数都不是标准的C 语言库函数，调用风格与printf 类似。如果本次实验提供的程序库中不包含这两个函数，你能自己实现吗？建议从因特网中检索相关资料，或者，直接查阅Linux 的源代码，看看printf风格的函数是怎样实现的。

(2)8.9 节中给出了两个函数start\_timer()和start\_ack\_timer()，它们都是定时器函数，两个函数启动定时器的时机不同，而且在定时器到时之前重新调用函数对原残留时间的处理方式也不同，为什么要这样设计？

**A：**

(1) 实现了，当CRC 校验错误，帧超长（可能是因为误码毁掉了帧边界字符），定时器超时重传等情况或者每发送和接收一帧，都打印出相关调试信息，便于协议分析。

(2)start\_ack\_timer()和stop\_ack\_timer()两个定时器函数为搭载ACK机制设置。start\_ack\_timer()与start\_timer()有两点不同。首先，系统在把截止到调用函数start\_timer()时刻为止已经放入物理层发送队列的数据发送完毕后才开始启动计时（不是从当前时间开始计时），而start\_ack\_timer()启动时刻为当前时刻。其次，在先前启动的定时器未超时之前重新执行start\_ack\_timer()调用，定时器将依然按照先前的时间设置产生超时事件ACK\_TIMEOUT。两种定时器不同的定时处理方式是为了适应数据链路层协议软件开发的需要。

**4.软件测试方面的问题**

**Q：**验证所完成的程序能否在各种情况下都能够正确工作，是软件测试环节的主要目的。表3 中列出了七种测试方案，设计这么多种测试方案的目的是什么？分析每种测试方案，每种方案主要是为了瞄准你的协议软件中可能出问题的哪些环节？或者说，你的协议软件存在什么问题时，测试会失败？你觉得还存在有哪些问题是这些测试尚未覆盖的？这些测试方案和验证协议正确性的手段由指导教师给出，如果是由你自己独立完成整个协议的设计和测试，你会采用哪些手段来验证你的程序能正确工作？针对本次实验的具体问题，你能不能提出一种更高效的软件测试方案？本次实验所提供的程序库还有哪些不足，怎样才能对协议开发提供更方便的支撑，这关系到在整体软件开发过程中的不同模块间的功能划分问题，给出你的建议。

**A：**由于网络环境是多变的，而且网络协议的功能可能不是单一的，因此只有通过多种测试方案才能测得一个协议的性能与可靠程度。在此次实验中，在合理设置参数（窗口大小、ACK\_TIMER、DATA\_TIMER）的情况下可以长时间稳定运行而不会崩溃，而当某些参数设计得不合理比如 DATA\_TIMER等，会导致效率很低。

不同测试方案的目的在于：

datalink au(bu): 测试在无误码情况下的双工传输的线路利用率，可以测得协议在双工工作的模式下的实际最大线路率

datalink a(b): 测试在一定误码率(1e-5)情况下的双工传输利用率，可以测得协议在有一定误码率的信道以及双工工作模式的实际线路利用率。

datalink afu(bfu): 测试在无误码信道上以泛洪方式工作的线路利用率。

datalink af(bf): 测试在有误码信道(1e-5)上以泛洪方式工作的线路利用率。

datalink af(bf) -ber 1e-4: 测试在有误码信道(1e-4)上以泛洪方式工作的线路利用率。

**5.对等协议实体之间的流量控制**

**Q：**在教科书中多次提及“流量控制”问题，这个问题的确很重要。在本次实验所设计的程序中，多多少少也考虑了上下层软件实体之间的数据流量控制问题。你认为你所设计的滑动窗口协议软件有没有解决两个站点的数据链路层对等实体之间的流量控制问题？如果是已经解决了，那么是怎样解决的？如果尚未解决，那么还需要对协议进行哪方面的改进？

**A：**我所设计的滑动窗口协议，只是静态解决了流量控制问题，不能根据网络的具体情况作出动态调整。为了做到此要求，需要在协议中增加监控模块，根据当前网络的情况做出传输的调整，更好地解决流量控制问题

**6.与标准协议的对比**

**Q：**如果现实中有两个相距5000 公里的站点要利用你所设计的协议通过卫星信道进行通信，还有哪些问题需要解决？实验协议离实用还有哪些差距？你觉得还需要增加哪方面的功能？从因特网或其他资料查阅LAPB 相关的协议介绍和对该协议的评论，用成熟的CCITT 链路层协议标准对比你所实现的实验性协议，实验性协议的设计还遗漏了哪些重要问题？

**A：**如果是相距很远的卫星通信，还要考虑到的因素是信道的因素，包括信道的连通性，是否有可能会出现卫星正在偏离通信范围；信道的稳定性，误码率的波动，延时的波动等等情况，根据这些因素可能要考虑动态调整协议的参数（包括ACK\_TIMER和DATA\_TIMER等）。

**五、实验总结和心得体会**

1.完成本次实验的实际上机调试时间是多少？

实际上机调试的时间大概有一天，测试参数时又用了两三天。

2.编程工具方面遇到了哪些问题？包括Windows环境和VC软件的安装问题。

在安装VC++6.0时曾一度崩溃，电脑无法打开，后经过长时间的询问还是解决了问题。

3.编程语言方面遇到了哪些问题？包括C语言使用和对C语言操控能力上的问题。

在自己建工程的时候忘记将protocol.lib和protocol.dll添加进去导致一直报错，后面认真仔细地看了看样例工程后解决了问题。还有出现了定义变量名时与之前的宏定义重名，编译器报错又不详细导致一直没有发现错误，这些编程中的细节都是很重要的。

4.协议方面遇到了哪些问题？包括协议机制的设计错误，发现协议死锁，或者不能正确工作，协议参数的调整等问题。

大概按照书上的架构来的，但是在CRC校验部分数据帧与ack帧长度问题上一开始没有弄清楚，没有理解padding域的意义以及CRC转换的机制，还有提供的库函数调用时参数输入起初按照书上来结果出现了一系列编译错误。在参数设置上也出现了问题，信道利用率下降得很快最后稳定的值也很低，后来经过调整后信道利用率有所提高。

6. 总结本次实验，你在C 语言方面，协议软件方面，理论学习方面，软件工程方面等哪些方面上有所提高？

这次实验编写程序的时间并不是很长，但是在调试时出现了一些列问题，虽然跟编译器古老报错能力不佳有关，但是还是自己忽略了很多细节。在阅读库函数时起初没有认真观察参数的定义导致数据帧的长度出错，耗费了一定的时间，但最终还是解决了问题。

此次实验最艰苦的部分是ACK\_TIMER、DATA\_TIMER和窗口大小等参数的设置，最初没有进行思考和估算，运行出来的效率惨不忍睹，后面进行了公式的推算后开始进行了漫长的测试阶段，为了效率的提高不断的修改参数，但最后还是与参考最佳的效率有差距。

通过这次实验我还是加深了对协议的理解，并且认识到理论与实际巨大差别，为了实际的使用，开发者必须进行长时间的测试和实践方能使产品的性能达到最佳。

**五、源程序清单**

GobackN协议

#include <stdio.h>

#include <string.h>

#include "protocol.h"

#define MAX\_SEQ 7

#define DATA\_TIMER 3000

#define ACK\_TIMER 270

#define DATA 1

#define ACK 2

struct frame

{

unsigned char kind ; //帧的类型

unsigned char ack ; //帧的ack号

unsigned char seq ; //帧的序列号

unsigned char info[PKT\_LEN] ; //网络层分组

unsigned int padding ;

} ;

static int phl\_ready = 0 ;

static int between(unsigned char a, unsigned char b, unsigned char c)

//窗口函数，判断是否在窗口内

{

if (((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c )&& (c < a)))

return 1 ;

else

return 0 ;

}

static void put\_frame(unsigned char \*frame, int len)

//加入校验和CRC

{

\*(unsigned int \*)(frame + len) = crc32(frame, len) ;

send\_frame(frame, len + 4) ;

phl\_ready = 0 ;

}

static void send\_data\_frame(unsigned char fk, unsigned char frame\_nr, unsigned char frame\_expected, unsigned char buffer[MAX\_SEQ + 1][PKT\_LEN])

//发送数据帧、ack、nak

{

struct frame s ;

s.kind = fk ;

s.seq = frame\_nr ;

s.ack = (frame\_expected + MAX\_SEQ) % (MAX\_SEQ + 1) ;

if (fk == DATA)

{

memcpy(s.info, buffer[frame\_nr], PKT\_LEN) ;

//复制分组到帧中

dbg\_frame("Send DATA %d %d, ID %d\n", s.seq, s.ack, \*(short \*)s.info) ;

put\_frame((unsigned char \*)&s, 3 + PKT\_LEN) ;

//加入校验和

start\_timer(frame\_nr, DATA\_TIMER) ;

}

if (fk == ACK)

{

dbg\_frame("Send ACK %d\n", s.ack) ;

put\_frame((unsigned char \*)&s, 2) ;

}

phl\_ready = 0 ;

stop\_ack\_timer() ;

}

int main(int argc, char \*\*argv)

{

int event, arg ;

struct frame f ;

int len = 0 ;

static unsigned char i ;

static unsigned char ack\_expected = 0, next\_frame\_to\_send = 0 ;

static unsigned char frame\_expected = 0 ;

static unsigned char nbuffered = 0 ;

unsigned char buffer[MAX\_SEQ + 1][PKT\_LEN] ;

protocol\_init(argc, argv) ;

lprintf("Designed by dao, build: " \_\_DATE\_\_" "\_\_TIME\_\_"\n") ;

//enable\_network\_layer();

for (;;)

{

event = wait\_for\_event(&arg) ;

switch (event)

{

case NETWORK\_LAYER\_READY :

get\_packet(buffer[next\_frame\_to\_send]) ;

nbuffered ++ ;

send\_data\_frame(DATA, next\_frame\_to\_send, frame\_expected, buffer) ;

next\_frame\_to\_send = (next\_frame\_to\_send + 1) % (MAX\_SEQ + 1) ;

break;

case PHYSICAL\_LAYER\_READY :

phl\_ready = 1 ;

break ;

case FRAME\_RECEIVED :

//一个数据帧到达

len = recv\_frame((unsigned char \*)&f, sizeof f) ;

if (len < 5 || crc32((unsigned char \*)&f, len) != 0)

//校验和出错

{

dbg\_event("\*\*\*\* Receiver Error, Bad CRC Checksum\n") ;

break;

}

if (f.kind == DATA)

{

dbg\_frame("Recv DATA %d %d, ID %d\n", f.seq, f.ack, \*(short \*)f.info) ;

if (f.seq == frame\_expected)

{

put\_packet(f.info, len-7) ;

frame\_expected = (frame\_expected + 1) % (MAX\_SEQ + 1) ;

start\_ack\_timer(ACK\_TIMER) ;

}

}

if (f.kind == ACK)

{

dbg\_frame("Recv ACK %d %d, ID %d\n", f.seq, f.ack, \*(short \*)f.info) ;

}

while (between(ack\_expected, f.ack, next\_frame\_to\_send))

{

nbuffered -- ;

stop\_timer(ack\_expected) ;

ack\_expected = (ack\_expected + 1) % (MAX\_SEQ + 1) ;

}

break ;

case DATA\_TIMEOUT :

//超时，重传发送窗口中所有的帧

dbg\_event("---- DATA %d timeout\n", arg) ;

next\_frame\_to\_send = ack\_expected ;

for (i = 1 ; i <= nbuffered ; i ++)

{

send\_data\_frame(DATA, next\_frame\_to\_send, frame\_expected, buffer) ;

next\_frame\_to\_send = (next\_frame\_to\_send + 1) % (MAX\_SEQ + 1) ;

}

break ;

case ACK\_TIMEOUT :

dbg\_event("---- ACK %d timeout\n", arg) ;

send\_data\_frame(ACK, 0, frame\_expected, buffer) ;

break ;

}

if ((nbuffered < MAX\_SEQ) && phl\_ready)

enable\_network\_layer() ;

else

disable\_network\_layer() ;

}

}

选择重传协议

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include "protocol.h"

#define MAX\_SEQ 31

#define NR\_BUFS ((MAX\_SEQ+1)/2)

#define DATA\_TIMER 5000

#define ACK\_TIMER 280

#define DATA 1

#define ACK 2

#define NAK 3

struct frame{

unsigned char kind ;

unsigned char ack ;

unsigned char seq ;

unsigned char info[PKT\_LEN] ;

unsigned int padding ;

} ;

static int no\_nak = 1 ; //还没有nak发送过

static int phl\_ready = 0 ;

static int between(unsigned char a, unsigned char b, unsigned char c)

//窗口函数，判断帧号（ack号）是否在窗口内

{

if (((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c ) && (c < a)))

return 1 ;

else

return 0 ;

}

static void put\_frame(unsigned char \*frame, int len)

//加入校验和CRC

{

\*(unsigned int \*)(frame + len) = crc32(frame, len) ;

send\_frame(frame, len + 4) ;

phl\_ready = 0 ;

}

static void send\_data\_frame(unsigned char fk, unsigned char frame\_nr, unsigned char frame\_expected, unsigned char buffer[NR\_BUFS][PKT\_LEN])

//发送数据帧、ack、nak

{

struct frame s ;

s.kind = fk ;

s.seq = frame\_nr ;

s.ack = (frame\_expected + MAX\_SEQ) % (MAX\_SEQ + 1) ;

if ( fk == DATA )

{

memcpy(s.info, buffer[frame\_nr % NR\_BUFS], PKT\_LEN) ;

//复制分组到帧中

dbg\_frame("Send DATA %d %d, ID %d\n", s.seq, s.ack, \*(short \*)s.info) ;

put\_frame((unsigned char \*)&s, 3 + PKT\_LEN) ;

start\_timer(frame\_nr % NR\_BUFS, DATA\_TIMER) ;

}

else if ( fk == NAK )

{

no\_nak = 0 ;

put\_frame((unsigned char \*)&s, 2) ;

}

else if (fk == ACK)

{

dbg\_frame( "Send ACK %d\n", s.ack ) ;

put\_frame((unsigned char \*)&s, 2) ;

}

phl\_ready = 0 ;

stop\_ack\_timer() ;

}

int main(int argc, char \*\*argv)

{

int event, arg ;

struct frame f ;

int len = 0 ;

int i ;

static unsigned char ack\_expected = 0, next\_frame\_to\_send = 0 ;

static unsigned char frame\_expected = 0, too\_far = NR\_BUFS ;

static unsigned char nbuffered = 0 ;

int arrived[NR\_BUFS] ;

unsigned char in\_buf[NR\_BUFS][PKT\_LEN], out\_buf[NR\_BUFS][PKT\_LEN] ;

protocol\_init(argc, argv) ;

lprintf("Designed by dao, build: " \_\_DATE\_\_" "\_\_TIME\_\_"\n") ;

for (i = 0 ; i < NR\_BUFS ; i ++)

{

arrived[i] = 0 ;

}

//enable\_network\_layer();

for (;;)

{

event = wait\_for\_event(&arg) ;

switch (event)

{

case NETWORK\_LAYER\_READY :

get\_packet(out\_buf[next\_frame\_to\_send % NR\_BUFS]) ;

nbuffered ++ ;

send\_data\_frame(DATA, next\_frame\_to\_send, frame\_expected, out\_buf) ;

next\_frame\_to\_send = (next\_frame\_to\_send + 1) % (MAX\_SEQ + 1) ;

break ;

case PHYSICAL\_LAYER\_READY :

phl\_ready = 1 ;

break ;

case FRAME\_RECEIVED :

len = recv\_frame((unsigned char \*)&f, sizeof f) ;

if (len < 5 || crc32((unsigned char \*)&f, len) != 0)

//校验和出错，发送nak请求重传

{

if (no\_nak == 1)

{

send\_data\_frame(NAK, 0, frame\_expected, out\_buf) ;

}

dbg\_event("\*\*\*\* Receiver Error, Bad CRC Checksum\n") ;

break ;

}

if (f.kind == DATA)

{

if ((f.seq != frame\_expected ) && no\_nak == 1)

//序列号错误

{

send\_data\_frame(NAK, 0, frame\_expected, out\_buf) ;

}

else

{

start\_ack\_timer(ACK\_TIMER) ;

}

if (between( frame\_expected, f.seq, too\_far ) && arrived[f.seq % NR\_BUFS] == 0)

{

//log\_printf("Received a frame, "); log\_printf("%d bytes\n", len);

dbg\_frame("Recv DATA %d %d, ID %d\n", f.seq, f.ack, \*(short \*)f.info) ;

arrived[f.seq % NR\_BUFS] = 1 ;

memcpy(in\_buf[f.seq % NR\_BUFS], f.info, len-7) ;

while (arrived[frame\_expected % NR\_BUFS])

{

put\_packet(in\_buf[ frame\_expected % NR\_BUFS ], len-7) ;

no\_nak = 1 ;

arrived[frame\_expected % NR\_BUFS] = 0 ;

frame\_expected = (frame\_expected + 1) % (MAX\_SEQ + 1) ;

too\_far = (too\_far + 1) % (MAX\_SEQ + 1) ;

start\_ack\_timer(ACK\_TIMER) ;

}

}

}

if ((f.kind == NAK) && between(ack\_expected, (f.ack + 1) % (MAX\_SEQ + 1), next\_frame\_to\_send))

//发送方收到nak，重传

{

dbg\_frame("Recv NAK %d\n", f.ack) ;

send\_data\_frame(DATA, (f.ack + 1) % (MAX\_SEQ + 1), frame\_expected, out\_buf) ;

}

while (between(ack\_expected, f.ack, next\_frame\_to\_send))

{

nbuffered -- ;

stop\_timer(ack\_expected % NR\_BUFS) ;

ack\_expected = (ack\_expected + 1) % (MAX\_SEQ + 1) ;

}

break ;

case DATA\_TIMEOUT :

//超时重传

dbg\_event("---- DATA %d timeout\n", arg) ;

send\_data\_frame(DATA, ack\_expected, frame\_expected, out\_buf) ;

break ;

case ACK\_TIMEOUT :

dbg\_event("---- ACK %d timeout\n", arg) ;

send\_data\_frame(ACK, 0, frame\_expected, out\_buf) ;

break ;

}

if ((nbuffered < NR\_BUFS) && phl\_ready)

enable\_network\_layer() ;

else

disable\_network\_layer() ;

}

}