北京邮电大学《计算机网络》课程实验报告

<table>
<tr><td>实验名称</td><td colspan="2">数据链路层滑动窗口协议的设计与实现</td><td>学院</td><td>计算机</td><td>指导教师</td><td>卞佳丽</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>2018211301</td><td>1</td><td colspan="2">2018210715</td><td>石宇辉</td><td colspan="2"></td></tr>
<tr><td>2018211301</td><td>2</td><td colspan="2">2018210909</td><td>张昕旸</td><td colspan="2"></td></tr>
<tr><td>2018211301</td><td>28</td><td colspan="2">2018213664</td><td>孙昊</td><td colspan="2"></td></tr>
<tr><td>实验内容</td><td colspan="6">本次实验选用的滑动窗口协议为选择重传/回退 N 步协议，利用所学数据链路层原理，自行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}，信道提供帧传输服务，网络层分组长度固定为 256 字节。
本次实验选用的滑动窗口协议为选择重传与回退 N 步协议，其中选择重传协议使用了 NAK 通知机制，两个协议都采用捎带 ACK 的确认机制。</td></tr>
<tr><td>学生实验报告</td><td colspan="6">（详见“实验报告和源程序”册）</td></tr>
<tr><td>课程设计成绩评定</td><td colspan="6">评语:

成绩:

指导教师签名:

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

一、实验内容和实验环境描述

1.1 实验内容

利用所学数据链路层相关知识，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。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。

1.2 实验环境

操作系统：Microsoft Windows 10（开发）、CentOS（测试）

编辑器：Microsoft Visual Studio Code、Sublime Text、Vim

编译环境：Microsoft Visual Studio 2019、MinGW

二、软件设计

2.1 数据结构

2.1.1 帧格式示意图

数据帧格式

```
+ ======+ =====+======+ ============+======+
| KIND(1) | SEQ(1) | ACK(1) | DATA(240~256) | CRC(4) |
+ ======+ =====+======+ ============+======+
```

ACK 帧格式

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

NAK 帧格式

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

2.1.2 帧结构体

```
struct FRAME {
    unsigned char kind;             // 帧类型
    unsigned char ack;              // 捎带确认帧序列号
    unsigned char seq;              // 帧序列号
    unsigned char data[PKT_LEN];    // 帧数据信息
    unsigned int  padding;          // 帧间隔
};
```

2.1.3 宏定义常量

```
#define MAX_SEQ    7                    // 最大帧序号
#define NR_BUFS    ((MAX_SEQ + 1) / 2)  // 窗口大小
#define DATA_TIMER 2000                 // 数据定时器超时时间（单位：ms）
#define ACK_TIMER  1000                 // ACK 定时器超时时间（单位：ms）
```

2.1.4 全局变量说明

回退 N 步协议：

```
static unsigned char frame_nr = 0;                  // 准备发送的帧序号
static unsigned char nbuffered = 0;                 // 缓冲区已使用的数量
static unsigned char buffer[MAX_SEQ + 1][PKT_LEN];  // 发送缓冲区
static unsigned char frame_expected = 0;            // 期待接收的帧序号
static unsigned char ack_expected = 0;              // 期待的 ACK 序号
static int phl_ready = 0;                           // 物理层状态
```

选择重传协议：

```
static unsigned char out_buf[NR_BUFS][PKT_LEN]; // 发送缓冲区
static unsigned char nbuffered = 0;             // 发送缓冲区用量
static unsigned char in_buf[NR_BUFS][PKT_LEN];  // 接收缓冲区
static int in_buf_len[NR_BUFS];                 // 接收缓冲区里各分组的长度
static bool arrived[NR_BUFS] = {false};         // 接收缓冲区里是否存在该分组

static unsigned char ack_expected = 0;          // 发送窗口下界
static unsigned char next_frame_to_send = 0;    // 发送窗口上界加一
static unsigned char frame_expected = 0;        // 接收窗口下界
static unsigned char too_far = NR_BUFS;         // 接收窗口上界加一

static int phl_ready = 0;                       // 物理层状态
```

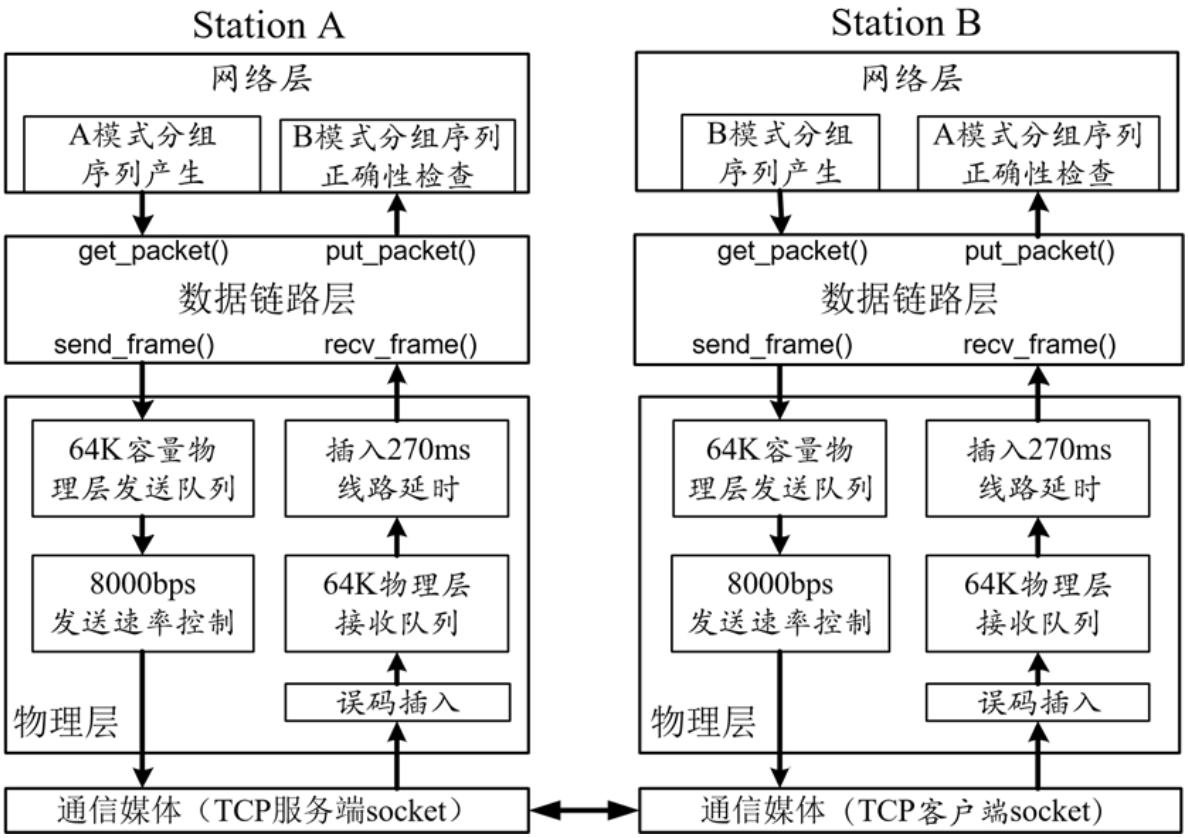
2.2 模块结构

程序中用到如下函数（两种协议下，函数的具体实现不同，但使用统一的接口以保证程序的可维护性）：

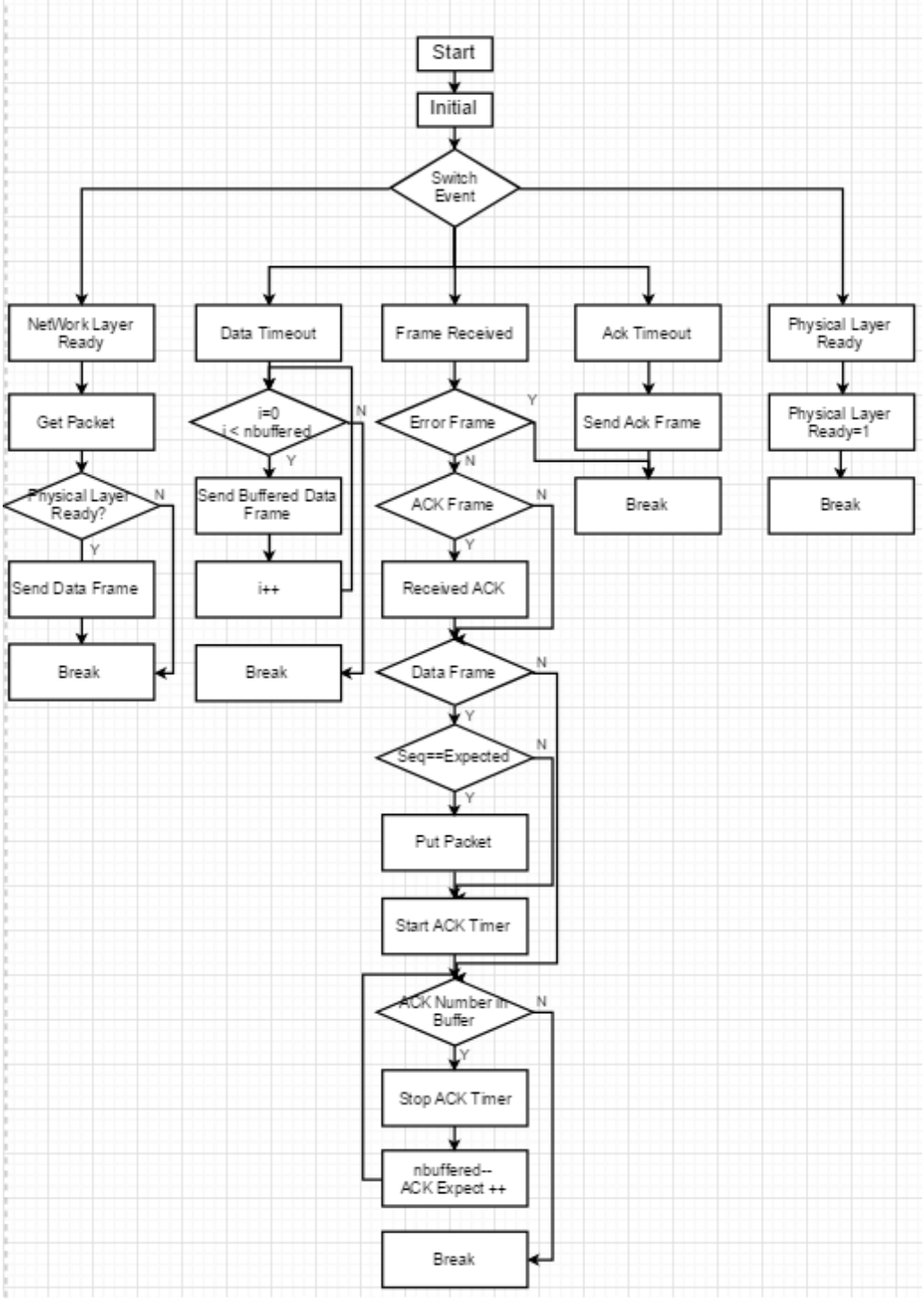
```
1.  /* 判断所接收到的序号是否在窗口范围内，即返回是否满足 a <= b < c */
2.  static bool between(unsigned char a, unsigned char b, unsigned char c);
3.
4.  /* 在帧末尾添加校验和，并递交给物理层发送 */
5.  static void put_frame(unsigned char *frame, int len);
6.
7.  /* 准备向物理层发送数据帧 */
8.  static void send_data_frame(void);
9.
10. /* 准备向物理层发送 ACK 帧 */
11. static void send_ack_frame(void);
12.
13. /* 准备向物理层发送 NAK 帧 */
14. static void send_nak_frame(void);
```

2.3 算法流程

2.3.1 程序库通信流程



2.3.2 回退 N 步协议流程



- 首先初始化协议，然后不断判断所到事件：

a) 若为网络层就绪事件：则从网络层接收分组到发送缓存窗口中，如果物理层就绪则包装并且发送帧，同时更新窗口和帧发送信息；

b) 若为物理层就绪事件：则将物理层就绪标志置为 1；

c) 若为数据帧超时事件：则重发在发送窗口中的所有数据帧；

d) 若为 ACK 超时事件：则单独发送 ACK；

e) 若为帧接收事件：则先判断是否为错误帧，如果是错误帧则直接结束，如果不是错误帧则判断帧的类型：

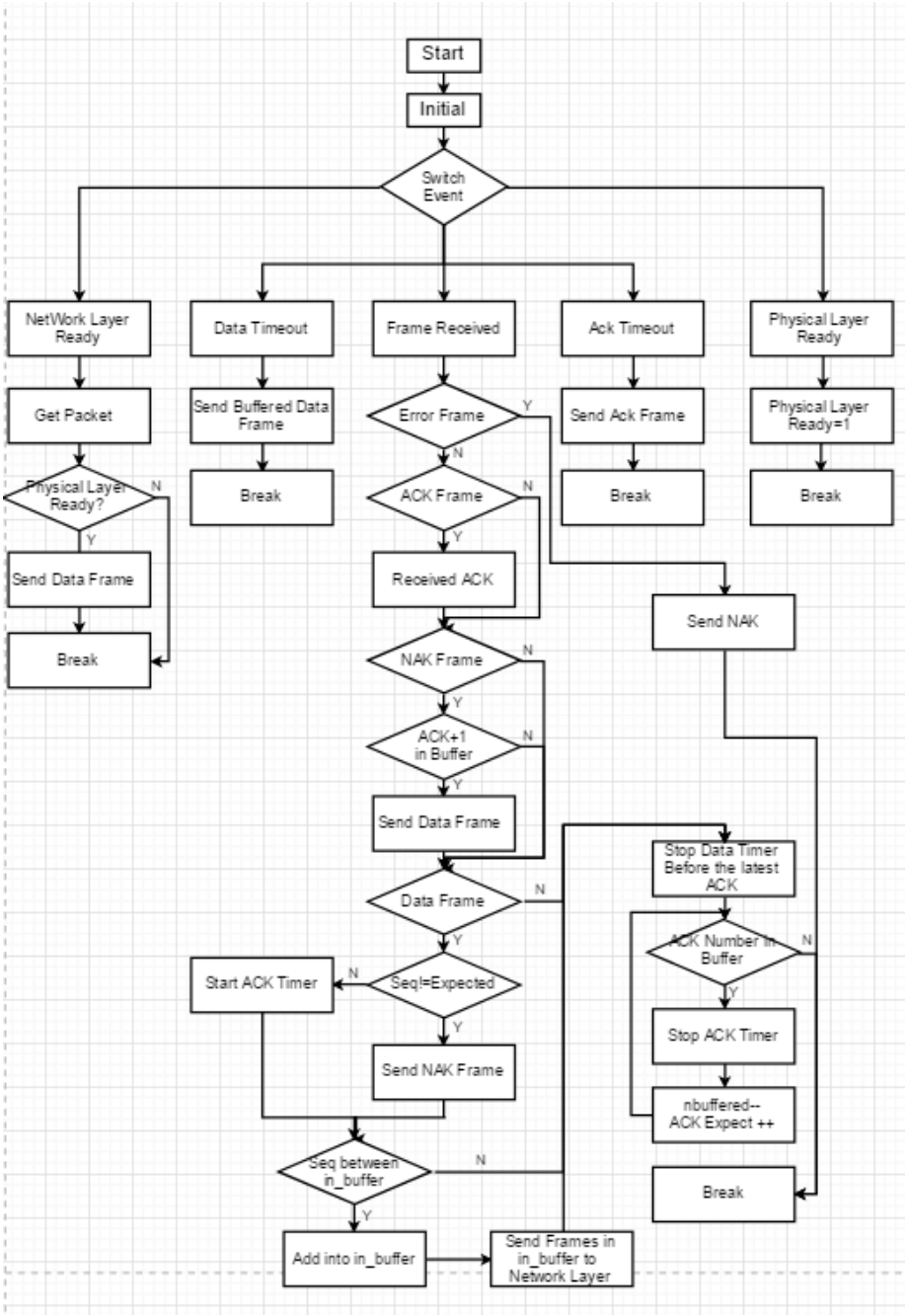
i) 对于 ACK 帧，则接收 ACK；

ii) 对于数据帧，如果序列号正确，则将该帧上传至网络层。启动 ACK 计时器；

对于所有类型的帧，对包含的 ACK 序号进行累计确认并释放对应发送缓存；

- 如果物理层就绪且窗口不满，则打开网络层接口，否则关闭。

2.3.3 选择重传协议流程



- 首先初始化协议，然后不断判断所到事件：

a) 若为网络层就绪事件：则从网络层接收分组到发送缓存窗口中，如果物理层就绪则包装并且发送帧，同时更新窗口和帧发送信息；

b) 若为物理层就绪事件：则将物理层就绪标志置为 1；

c) 若为数据帧超时事件：则重发在发送窗口中的该数据帧；

d) 若为 ACK 超时事件：则单独发送 ACK；

e) 若为帧接收事件：则先判断是否为错误帧，如果是错误帧则发送 NAK 帧并且结束，如果不是错误帧则判断帧的类型：

i) 对于 ACK 帧，则接收 ACK；

ii) 对于 NAK 帧，所携带 ACK 序号的下一帧在发送窗口中则重发该帧；

iii) 对于数据帧，如果序列号正确，则开始 ACK 计时器，否则发送 NAK。如果该帧序号在接收窗口范围内且对应接收窗口缓存为空，则将该帧存入接收窗口，并将接收窗口中缓存的信息上传至网络层；

对于所有类型的帧，对包含的 ACK 序号进行累计确认并释放对应发送缓存；

- 如果物理层就绪且窗口不满，则打开网络层接口，否则关闭。

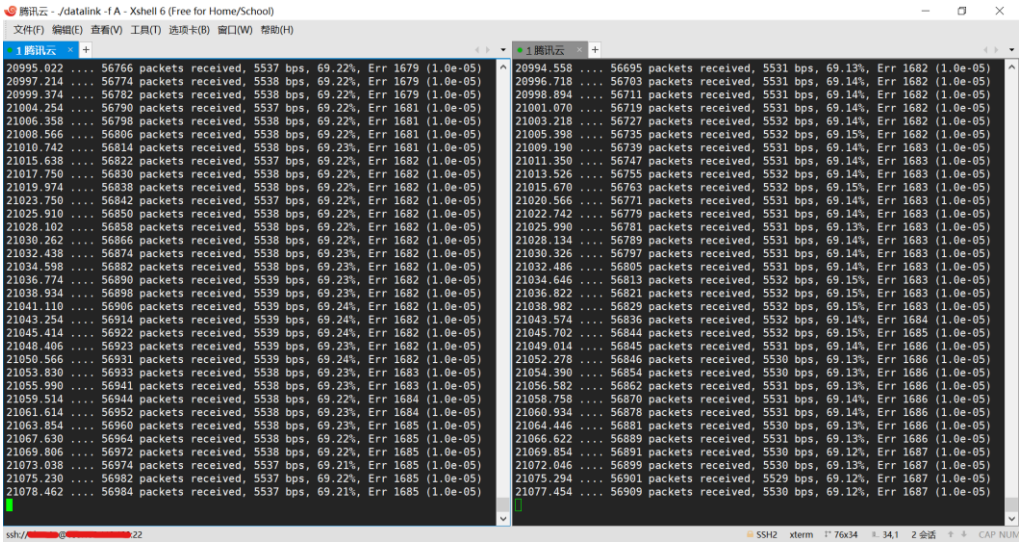
三、实验结果分析

3.1 实现情况

根据性能测试记录表（见附录）中第 2、4、5 项测试（即有差错环境下的测试）的结果，我们小组所实现的协议实现了有误码信道环境中无差错传输的功能。

3.2 健壮性

根据性能测试记录表（附录）所记录的结果，程序可以满足可靠运行 20 分钟以上的要求。实际上，在测试过程中，程序可以稳定可靠地运行 24 小时以上，但由于时间关系，我们无法对每项测试进行如此长时间的测试。



协议长时间运行情况

3.3 参数选取

3.3.1 滑动窗口大小

$$w = \frac{T_f + R}{T_f} = \frac{256 + 270 \times 2}{256} \approx 4$$

如果取值过小会导致频繁传输、等待 ACK 信号，降低信道使用率；如果取值过高则容易产生丢包现象，如果网络丢包率高，则会导致多次发送重复数据，同样耗费了网络带宽。结合老师所给的示例程序中的滑动窗口取值与理论计算值，并且由于一般取输入窗口=输出窗口，所以综合下来我们选择 w=8 作为协议的滑动窗口总数，其中 $W_T = W_R = 4$。

3.3.2 ACK 搭载定时器的时限

ACK_TIMER 如果取值过小则会导致频繁发送 ACK 帧，从而降低信道传送数据的有效利用率；如果取值过高则会在数据量小的时候导致发送窗口长时间等待，从而降低信道利用率。

经过计算:

$$2 \times 270\text{ms} \leq \text{ACK\_TIMER} \leq 270\text{ms} + 7 \times 256\text{ms}$$

即：

$$540\text{ms} \leq \text{ACK\_TIMER} \leq 2062\text{ms}$$

经过若干次调试并对性能做分析，我们取靠近中间值的 ACK_TIMER = 1000ms。

3.3.3 重传定时器的时限

DATA_TIMER 如果取值过小则会导致频繁重发数据帧，从而降低信道的有效利用率；如果取值过高则同样会导致发送窗口长时间等待，从而降低信道利用率。

经过计算:

$$\text{ACK\_TIMER} + 270\text{ms} \leq \text{DATA\_TIMER} \leq 270\text{ms} \times 2 + 7 \times 256\text{ms}$$

即：

$$1270\text{ms} \leq \text{DATA\_TIMER} \leq 2332\text{ms}$$

经过若干次调试并对性能做分析，我们取 DATA_TIMER = 2000ms。

3.4 理论分析

3.4.1 理论最大信道利用率

w = 滑动窗口数量　T_f = 发送数据帧时间　R = 传输延迟

利用率= $\frac{w \times T_f}{T_f + R} = \frac{4 \times 263}{263 + 270 \times 2} \approx 100\%$

3.4.2 理论最大有效利用率

无误码：

理论最大有效利用率=理论最大信道利用率×数据占比=$100\% \times \frac{256}{263} = 97.3\%$

有误码：

假设信道始终在传输数据帧，ACK 与 NAK 不会出错。

(1) 若误码率为10^{-5}，平均发送 100000bit 信息出错 1bit，即发送 48 个帧出错 1 帧。

对于选择重传协议，重传此帧信道有效利用率为$\frac{48 \times 256}{263 \times 49} = 95.4\%$；

对于回退 N 步协议，重传此帧平均需要重新传送 4 帧，所以重传此帧信道有效利用率为$\frac{48 \times 256}{263 \times 52} = 89.9\%$。

(2) 若误码率为10^{-4}，平均发送 10000bit 信息出错 1bit，即发送 5 个帧出错 1 帧。

对于选择重传协议，重传此帧信道有效利用率为$\frac{5 \times 256}{263 \times 6} = 81.1\%$；

对于回退 N 步协议，重传此帧平均需要重新传送 4 帧，所以重传此帧信道有效利用率为$\frac{5 \times 256}{263 \times 9} = 54.1\%$。

3.5 实验结果分析

我们在实际测试中发现，实际线路利用率与理论值有一定差距。经过分析，我们认为造成这种情况的原因有：

(1) 可能由于模拟的网络层发送分组到数据链路层的速率未达到信道传输能力的上限，所以由于数据量的不足，势必会导致信道出现空闲的情况，无法达到理论推导的最大利用率。

可以增加模拟网络层的数据密度。

(2) 在实际情况下，成帧过程中 CRC 的计算等数据处理过程需要消耗一定的时间，在处理时间内信道空闲，导致无法达到理论推导的最大利用率。

(3) 实际情况下，不可能做到每一帧都捎带确认，若出现单独确认情况，则也会降低线路的有效利用率。

3.6 存在的问题

虽然没有出现测试失败的情况，但是在线路利用率方面与样例程序还存在一定的差距，分析原因可能是数据定时器和 ACK 定时器的超时时间的取值可以进一步优化。

同时在计算参数的时候没有考虑帧头帧尾的长度，可能存在一些误差。

四、研究和探索的问题

4.1 CRC 校验能力

问：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？

答：CRC 校验码的检错能力很强，CRC 校验码除了能检查出全部单个错、奇数个错和离散的二位错外，CRC 校验码还能检查出全部长度小于或等于 r 位的突发错。尽管 CRC 校验码检错能力是有限的，但从本次实验的长时间测试过程中可以看出，它的表现是非常稳定的，而且实际的信道误码率一般也不会比本次实验更高，所以 CRC 校验码在这种误码率的情况下作用足够有效。另外，如果 CRC 校验错误给网络层传输了错误数据，网络层的校验机制也可以作为第二层保障，进一步保证传输的正确性。

对于 CRC32 来说，如果想要一个出错的分组无法检测到，那么至少要发生了 33 位（即 r+1 位）的突发差错，当且仅当错误多项式和 G（x）一致时，余数才为 0，根据突发错误的定义，第一位和最后一位必须为 1，它能否与 G（x）匹配取决于中间的 31 位（即 r-1 位）故这样一个不正确的帧被当做有效帧接受的概率为$\frac{1}{2^{31}}$（更多位数的突发错误概率更小，在此不考虑），本次实验所用的信道，每天有一半时间在传输数据，那么每天传送的数据为 8000kbps × 43200s=3.456 × 1011bit，即 1.35*109 个分组，误码率为 1e-5，对每个长度为 256 字节的来说，发生 33 位错误的概率为 $p=\binom{33}{2048}\times 10^{-165} \ll 1^{4\times 33}\times 10^{-165}=10^{-33}$，那么平均每年会出现分组层误码的概率为 $P=365\times 1.35\times 10^{9}\times p\times \frac{1}{2^{31}}\approx 4.36\times 10^{-30}$，那么平均需要$2.29\times 10^{31}$年会发生一次分组层误码事件。

查阅文献后可知，CRC32 大概 1820 万个数据会发生约 4 万个冲突，按这个碰撞概率计算，分组层的误码率会高很多，可见我们的估计是夸大了实际性能的。

如果想要增加检测性能可以增加更多的校验位，这样，对硬件的要求会更高，需要网络相关设备拥有更强的算力，并且会占用更多的存储资源。

4.2 程序设计方面的问题

问：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？

答：两类定时器启动时机的不同体现在：数据定时器在当前物理层发送队列的数据发送完毕后开始启动计时，而 ACK 定时器在调用后直接启动。这样设计的原因是：数据链路层将帧交给物理层后不会直接发送到对端，而是暂存在队列中等待发送，如果在调用定时器函数的同时就启动定时器，那么会导致帧在队列中等待的时间也被计入定时器的用时中，而非等待的净时间，使得真正可用的传输时间缩短，容易发生超时；而 ACK 定时器不存在这个问题是因为 ACK 定时器在收到帧之后启动，不存在在队列中等待的时间。

两类定时器对残留时间的处理方式不同体现在：重复设置同一个编号的数据计时器会导致重新按新调用计时，而重复调用 ACK 定时器后定时器仍旧以先前的启动时间产

生超时信号。这样设计的原因是：重复设置同一个编号的数据计时器是因为重发或窗口已经移动，需要开始一个新的传递过程，因此需要重新开始计时；而重复调用 ACK 定时器是因为收到了新的帧，但此前接收的帧仍没有进行确认，所以定时器应当以最早收到的没有确认的帧为准产生超时信号，不能重置定时器。

4.3 软件测试方面的问题

问：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 中列出了五种测试方案，设计这么多种测试方案的目的是什么？

答：因为这两种协议的设计目标是能够适应复杂多变的网络环境的，因此，需要通过多项测试来模拟各种网络环境下的工作情况，这样才能验证我们实现的协议是否满足设计目标，是否在各种环境下都能有可接受的性能、健壮性和可靠程度。

问：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？

答：这五种测试方案各自的测试目标如下表所示：

命令选项	说明	瞄准的环节	什么情况会失败
--utopia	无误码信道数据传输	理想信道下协议的最基本工作情况	协议的收发基本逻辑错误
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”	因为会停发 100 秒，所以会测试到无法捎带确认时协议能否单独确认	错误处理失效、ACK 定时器逻辑错误，没能在超时后单独确认
--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	理想信道下需要大量吞吐时协议会不会产生拥塞	分组没有及时上传网络层，或帧没有及时传至物理层，导致拥塞
--flood	站点 A/B 的分组层都洪水式产生分组	大吞吐量下，能否实现对错误的处理	错误处理失效
--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	大吞吐量、高错误率下，能否实现正确通信	错误处理失效

问：你觉得还存在有哪些问题是这些测试尚未覆盖的？

答：单工模式下的通信，测试无法捎带确认的情况下协议是否正常工作。

问：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。

答：程序库尽能完成本地测试，无法实现跨主机通信。可以增加参数以指定目标主机，以完成真实网络环境下的测试。

4.4 对等协议实体之间的流量控制

问：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？

答：在我们设计的协议当中，流量的控制主要通过接收窗口，发送窗口还有确认机制来实现。因为有窗口大小的限制，发送方不会一次性发送过多信息导致接收方被信息洪流

所淹没，导致信息丢失。这样可能会导致信道的利用率降低，但是如果合理的设计窗口大小，依然可以达到较高的信号利用率。这样实现的是静态的流量控制问题，还不能根据网络的具体情况做出动态调整，如果要投入实际应用，加入动态的流量控制是有必要的。

4.5 与标准协议的对比

问：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？

答：对于相距很远的卫星，我们主要还需要考虑信道的因素。主要有信道的连通性问题，因为卫星可能会存在偏离通信范围，甚至出现通信盲区的情况；还有信道的稳定性，如何控制误码率和延时的波动的问题；对于流量控制方面，静态的流量控制投入实际应用也还远远不够。需要根据这些因素动态调整协议中的参数，包括 MAX_SEQ，DATA_TIMER，ACK_TIMER 等。

问：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？

答：与成熟的 CCITT 链路层协议标准对比，其为了进行一次通信，通信的一端必须先呼叫另一端，请求建立连接，并且被呼叫的一端可以根据自己的情况接受或拒绝这个连接请求，连接建立后，任意一端在任何时候均有权利断开这个连接。

五、实验总结和心得体会

5.1 完成本次实验的实际上机调试时间

数据链路层两种协议的规划、细节确定大约用时 0.5 小时，程序的编写与调试大约用时 3 小时，完成各项测试大约用时 3 小时。

5.2 编程工具方面遇到的问题

尽管小组各成员使用的开发环境不尽相同，但我们编写的代码具有较强的兼容性，并未因编程工具的不同造成任何致命的问题。

5.3 编程语言方面遇到的问题

小组各成员均具备良好的 C 语言编程基础，能够在实验中熟练的运用 C 语言实现协议的各个部分。由于各成员习惯的编码风格略有出入，因此使用统一的编码风格后，在编码过程中偶有不适应，但均能较为轻松地克服。

5.4 协议方面遇到的问题

在实现选择重传协议时，我们最初参考了教材中的模型，但经过理论分析与实践验证后发现一处实现细节可能导致协议的致命错误：书中的算法为了避免重复发送相同的 NAK 帧而引入了 no_nak 变量以记录 NAK 帧是否发送过，但这种算法在特定情况下会导致应该发送的 NAK 帧漏发，进而导致窗口提早滑动，新老帧发生碰撞，最终导致向网络层发送错误的分组。经过反复讨论与修改我们最终解决了该问题，使得协议能够无差错地稳定运行。

5.5 小组分工情况概述

石宇辉：程序编写、调试，实验报告内容补充、排版
张昕旸：理论分析及实验结果分析，实验报告撰写

孙昊：测试数据整理，对探究性问题做出解答

5.6 实验总结与心得

石宇辉：通过此次实验，我进一步熟悉了数据链路层的两种重要协议——回退 N 步协议和选择重传协议。对协议的实现不仅需要编码的能力，更重要的是对协议中各个细节的理解和把握。在程序的调试过程中，需要做到对协议的流程了如指掌，并具备一定的分析问题解决问题的能力，要善用断点、日志等工具帮助编码者进行调试。此外，实验的完成离不开小组成员间的密切沟通、协调配合。因此，这次实验是对我全方位的考验，也使得我在诸多方面取得了长足进步。

张昕旸：通过对参数的理论分析与理论推导，对于回退 N 步协议和选择重传协议的参数定义和工作原理有了更清晰的认识；在动手实践操作和测试的过程中，体验到了实践与理论的差别，并且学会了分析查找导致差别的原因，对于我进一步掌握和了解这两个数据链路层的协议有着巨大的帮助。同时在理解同组成员的代码的时候也发现了自己在编程习惯上的不足，如不注重代码可读性，参数设置随意等，在以后的学习生活中会加以注意和改进。

孙昊：经过本次实验，我对完整的通信过程、数据链路层协议有了进一步的理解，之前课上有些模糊的代码和参数也了解的更透彻。初步了解了在现有库的基础上进行程序开发的方法，懂得了软件跟踪功能对于软件调试的重要性，学习到了软件测试方面的方法，我也认识到理论分析和实际投入应用之间是有着很大差距的，而且对网络协议的调试也比本地程序的调试更加复杂，同时也加强了小组合作的意识。

六、源程序文件

6.1 回退 N 步协议实现源码

```c
/*
FileName: datalink.c
Function: 数据链路层 回退 N 步协议
Author:   石宇辉 张昕旸 孙昊
Data:     2020-05
*/

#include <stdio.h>
#include <string.h>
#include <stdbool.h>

#include "protocol.h"
#include "datalink.h"

#define MAX_SEQ    7        // 最大帧序号
#define DATA_TIMER 2000     // 数据定时器超时时间
#define ACK_TIMER  1000     // ACK 定时器超时时间

struct FRAME {
    unsigned char kind;
    unsigned char ack;
    unsigned char seq;
```

```
23.     unsigned char data[PKT_LEN];
24.     unsigned int  padding;
25. };
26.
27. static unsigned char frame_nr = 0;                    // 准备发送的帧序号
28. static unsigned char nbuffered = 0;                   // 缓冲区已使用的数量
29. static unsigned char buffer[MAX_SEQ + 1][PKT_LEN];   // 发送缓冲区
30. static unsigned char frame_expected = 0;             // 期待接收的帧序号
31. static unsigned char ack_expected = 0;               // 期待的 ACK 序号
32. static int phl_ready = 0;                            // 物理层是否就位
33.
34. // 判断所接收到的序号是否在窗口范围内，即返回是否满足 a <= b < c
35. static bool between(unsigned char a, unsigned char b, unsigned char c)
36. {
37.     if (((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a)))
38.         return true;
39.     else
40.         return false;
41. }
42.
43. // 在帧末尾添加校验和，并递交给物理层发送
44. static void put_frame(unsigned char *frame, int len)
45. {
46.     *(unsigned int *)(frame + len) = crc32(frame, len); // 加入校验和
47.     send_frame(frame, len + 4);                         // 向物理层缓冲区发送帧
48.     phl_ready = 0;
49. }
50.
51. // 准备向物理层发送数据帧
52. static void send_data_frame(void)
53. {
54.     struct FRAME s;
55.
56.     s.kind = FRAME_DATA;
57.     s.seq = frame_nr;
58.     s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
59.     memcpy(s.data, buffer[frame_nr], PKT_LEN);
60.
61.     dbg_frame("Send DATA %d %d, ID %d\n", s.seq, s.ack, *(short *)s.data);
62.
63.     put_frame((unsigned char *)&s, 3 + PKT_LEN);
64.     start_timer(frame_nr, DATA_TIMER);      // 启动数据定时器
65. }
66.
67. // 准备向物理层发送 ACK 帧
68. static void send_ack_frame(void)
69. {
70.     struct FRAME s;
```

```
71.
72.     s.kind = FRAME_ACK;
73.     s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
74.
75.     dbg_frame("Send ACK  %d\n", s.ack);
76.
77.     put_frame((unsigned char *)&s, 2);
78.     stop_ack_timer();       // 停止 ACK 定时器
79. }
80.
81. int main(int argc, char **argv)
82. {
83.     int event, arg;
84.     struct FRAME f;
85.     int len = 0;
86.
87.     protocol_init(argc, argv);
88.     lprintf("Designed by Shi Yuhui, Zhang Xinyang, Sun Hao, build: " __DATE__"  "__T
    IME__"\n");
89.
90.     disable_network_layer();
91.
92.     for (;;) {
93.         event = wait_for_event(&arg);
94.
95.         switch (event) {
96.         case NETWORK_LAYER_READY:
97.             get_packet(buffer[frame_nr]);    // 从网络层获取分组
98.             nbuffered++;
99.             send_data_frame();
100.             frame_nr = (frame_nr + 1) % (MAX_SEQ + 1);  // 准备发送的帧序号右移
101.             break;
102.
103.         case PHYSICAL_LAYER_READY:
104.             phl_ready = 1;
105.             break;
106.
107.         case FRAME_RECEIVED:
108.             len = recv_frame((unsigned char *)&f, sizeof f);
109.             if (len < 5 || crc32((unsigned char *)&f, len) != 0) {  // 校验和错误
110.                 dbg_event("**** Receiver Error, Bad CRC Checksum\n");
111.                 break;
112.             }
113.             if (f.kind == FRAME_ACK)
114.                 dbg_frame("Recv ACK  %d\n", f.ack);
115.             if (f.kind == FRAME_DATA) {
116.                 dbg_frame("Recv DATA %d %d, ID %d\n", f.seq, f.ack, *(short *)f.dat
    a);
```

```
117.                 if (f.seq == frame_expected) {      // 收到了期待的帧
118.                     put_packet(f.data, len - 7);    // 向网络层发送分组
119.                     frame_expected = (frame_expected + 1) % (MAX_SEQ + 1);  // 期待
    接收下一帧
120.                 }
121.                 start_ack_timer(ACK_TIMER);
122.             }
123.
124.             while (between(ack_expected, f.ack, frame_nr)) {
125.                 stop_timer(ack_expected);       // 停止数据定时器
126.                 nbuffered--;
127.                 ack_expected = (ack_expected + 1) % (MAX_SEQ + 1);  // 发送窗口下界
    右移
128.             }
129.             break;
130.
131.         case DATA_TIMEOUT:
132.             dbg_event("---- DATA %d timeout\n", arg);
133.             frame_nr = ack_expected;
134.             for (int i = 0; i < nbuffered; ++i) {       // 从头开始重新发送
135.                 send_data_frame();
136.                 frame_nr = (frame_nr + 1) % (MAX_SEQ + 1);
137.             }
138.             break;
139.
140.         case ACK_TIMEOUT:
141.             dbg_event("---- ACK timeout\n");
142.             send_ack_frame();       // 发送单独的 ACK 确认帧
143.             break;
144.         }
145.
146.         if (nbuffered < MAX_SEQ && phl_ready)
147.             enable_network_layer();
148.         else
149.             disable_network_layer();
150.     }
151. }
```

6.2 选择重传协议实现源码

```
1.  /*
2.  FileName: datalink.c
3.  Function: 数据链路层 选择重传协议
4.  Author:   石宇辉 张昕旸 孙昊
5.  Data:     2020-05
6.  */
7.
```

```c
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

#include "protocol.h"
#include "datalink.h"

#define MAX_SEQ    7                    // 最大帧序号
#define NR_BUFS    ((MAX_SEQ + 1) / 2)  // 窗口大小
#define DATA_TIMER 2000                 // 数据定时器超时时间
#define ACK_TIMER  1000                 // ACK 定时器超时时间

struct FRAME {
    unsigned char kind;
    unsigned char ack;
    unsigned char seq;
    unsigned char data[PKT_LEN];
    unsigned int  padding;
};

static unsigned char out_buf[NR_BUFS][PKT_LEN]; // 发送缓冲区
static unsigned char nbuffered = 0;             // 发送缓冲区用量
static unsigned char in_buf[NR_BUFS][PKT_LEN];  // 接收缓冲区
static int in_buf_len[NR_BUFS];                 // 接收缓冲区里各分组的长度
static bool arrived[NR_BUFS] = {false};         // 接收缓冲区里是否存在该分组

static unsigned char ack_expected = 0;          // 发送窗口下界
static unsigned char next_frame_to_send = 0;    // 发送窗口上界加一
static unsigned char frame_expected = 0;        // 接收窗口下界
static unsigned char too_far = NR_BUFS;         // 接收窗口上界加一

static int phl_ready = 0;                       // 物理层是否就位

// 判断所接收到的序号是否在窗口范围内，即返回是否满足 a <= b < c
static bool between(unsigned char a, unsigned char b, unsigned char c)
{
    if (((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a)))
        return true;
    else
        return false;
}

// 在帧末尾添加校验和，并递交给物理层发送
static void put_frame(unsigned char *frame, int len)
{
    *(unsigned int *)(frame + len) = crc32(frame, len); // 加入校验和
    send_frame(frame, len + 4);                         // 向物理层缓冲区发送帧
    phl_ready = 0;
```

```
}

// 准备向物理层发送数据帧
static void send_data_frame(unsigned char frame_nr)
{
    struct FRAME s;

    s.kind = FRAME_DATA;
    s.seq = frame_nr;
    s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
    memcpy(s.data, out_buf[frame_nr % NR_BUFS], PKT_LEN);

    dbg_frame("Send DATA %d %d, ID %d\n", s.seq, s.ack, *(short *)s.data);

    put_frame((unsigned char *)&s, 3 + PKT_LEN);
    start_timer(frame_nr % NR_BUFS, DATA_TIMER);        // 启动数据定时器
    stop_ack_timer();                                   // 停止 ACK 定时器
}

// 准备向物理层发送 ACK 帧
static void send_ack_frame(void)
{
    struct FRAME s;

    s.kind = FRAME_ACK;
    s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);

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

    put_frame((unsigned char *)&s, 2);
    stop_ack_timer();       // 停止 ACK 定时器
}

// 准备向物理层发送 NAK 帧
static void send_nak_frame(void)
{
    struct FRAME s;

    s.kind = FRAME_NAK;
    s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);

    dbg_frame("Send NAK  %d\n", s.ack);

    put_frame((unsigned char *)&s, 2);
     stop_ack_timer();       // 停止 ACK 定时器
}

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

```
104. {
105.     int event, arg;
106.     struct FRAME f;
107.     int len = 0;
108. 
109.     protocol_init(argc, argv);
110.     lprintf("Designed by Shi Yuhui, Zhang Xinyang, Sun Hao, build: " __DATE__"  "__
    TIME__"\n");
111. 
112.     disable_network_layer();
113. 
114.     for (;;) {
115.         event = wait_for_event(&arg);
116. 
117.         switch (event) {
118.         case NETWORK_LAYER_READY:
119.             get_packet(out_buf[next_frame_to_send % NR_BUFS]);   // 从网络层获取分
    组，取至发送缓冲区
120.             nbuffered++;
121.             send_data_frame(next_frame_to_send);
122.             next_frame_to_send = (next_frame_to_send + 1) % (MAX_SEQ + 1);    // 发
    送窗口上界右移
123.             break;
124. 
125.         case PHYSICAL_LAYER_READY:
126.             phl_ready = 1;
127.             break;
128. 
129.         case FRAME_RECEIVED:
130.             len = recv_frame((unsigned char *)&f, sizeof f);
131.             if (len < 5 || crc32((unsigned char *)&f, len) != 0) {  // 校验和错误
132.                 dbg_event("**** Receiver Error, Bad CRC Checksum\n");
133.                 send_nak_frame();   // 发送 NAK 帧
134.                 break;
135.             }
136.             if (f.kind == FRAME_ACK)
137.                 dbg_frame("Recv ACK  %d\n", f.ack);
138.             if (f.kind == FRAME_NAK) {
139.                 dbg_frame("Recv NAK  %d\n", f.ack);
140.                 if (between(ack_expected, (f.ack + 1) % (MAX_SEQ + 1), next_frame_t
    o_send))
141.                     send_data_frame((f.ack + 1) % (MAX_SEQ + 1));   // 重发数据帧
142.             }
143.             if (f.kind == FRAME_DATA) {
144.                 dbg_frame("Recv DATA %d %d, ID %d\n", f.seq, f.ack, *(short *)f.dat
    a);
145.                 if (f.seq != frame_expected)
146.                     send_nak_frame();   // 不是期待的帧，发送 NAK
```

```
147.                 else
148.                     start_ack_timer(ACK_TIMER);
149.
150.                 if (between(frame_expected, f.seq, too_far) && !arrived[f.seq % NR_
    BUFS]) {
151.                     arrived[f.seq % NR_BUFS] = true;
152.                     memcpy(in_buf[f.seq % NR_BUFS], f.data, PKT_LEN);   // 将帧内容
    拷贝至接收缓冲区
153.                     in_buf_len[f.seq % NR_BUFS] = len - 7;              // 记录帧长
    度
154.
155.                     // 上传至网络层
156.                     while (arrived[frame_expected % NR_BUFS]) {
157.                         put_packet(in_buf[frame_expected % NR_BUFS], in_buf_len[fra
    me_expected % NR_BUFS]);
158.                         lprintf("put %d to network layer\n", frame_expected % NR_BU
    FS);
159.                         arrived[frame_expected % NR_BUFS] = false;
160.                         // 接收窗口右移
161.                         frame_expected = (frame_expected + 1) % (MAX_SEQ + 1);
162.                         too_far = (too_far + 1) % (MAX_SEQ + 1);
163.                         start_ack_timer(ACK_TIMER);
164.                     }
165.                 }
166.             }
167.
168.             while (between(ack_expected, f.ack, next_frame_to_send)) {
169.                 stop_timer(ack_expected % NR_BUFS);
170.                 nbuffered--;
171.                 ack_expected = (ack_expected + 1) % (MAX_SEQ + 1);  // 发送窗口下界
    右移
172.             }
173.             break;
174.
175.         case DATA_TIMEOUT:
176.             dbg_event("---- DATA %d timeout\n", arg);
177.             send_data_frame(arg);   // 重发超时的数据帧
178.             break;
179.
180.         case ACK_TIMEOUT:
181.             dbg_event("---- ACK timeout\n");
182.             send_ack_frame();       // 发送单独的 ACK 确认帧
183.             break;
184.         }
185.
186.         if (nbuffered < NR_BUFS && phl_ready)
187.             enable_network_layer();
188.         else
```

```
189.             disable_network_layer();
190.     }
191. }
```

附录

性能测试记录表

序号	命令选项	说明	运行时间（秒）	GoBackN 算法线路利用率(%)		Selective 算法线路利用率(%)	
				A	B	A	B
1	--utopia	无误码信道数据传输	3516	59.04	95.63	44.35	68.16
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”	3890	44.06	74.59	40.01	65.57
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	3629	94.47	94.47	80.90	80.91
4	--flood	站点 A/B 的分组层都洪水式产生分组	3660	68.73	68.57	71.59	71.03
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}	3545	22.85	20.70	28.44	24.46