# 实验内容

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。

# 实验环境

Windows Subsystem for Linux : Legacy, kernel version 4.4.0-17763-Microsoft

CLion 2019.1.2

CMake 3.5.1

GNU Make 4.1

gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.10)

# 软件设计

# 数据结构

### 结构体定义

1. **struct** FRAME {
2. unsigned **char** kind; /\* 帧种类 \*/
3. unsigned **char** ack;  /\* ACK号 \*/
4. unsigned **char** seq;  /\* 数据编号 \*/
5. unsigned **char** data[PKT\_LEN];  /\* 数据内容 \*/
6. unsigned **int**  padding;  /\* 内部边界，防止贴CRC32时内存访问越界 \*/
7. };
8. **struct** packet {
9. unsigned **char** data[PKT\_LEN];  /\* 数据字段 \*/
10. };

长帧（捎带返回帧）



短帧（不带数据）



数据包（用于缓存）



### 类型定义

1. **typedef** **struct** packet packet;
2. **typedef** **struct** FRAME FRAME;
3. **typedef** unsigned **char** seq\_nr;
4. **typedef** **int** event\_type;
5. **typedef** unsigned **char** frame\_kind;

### 宏定义

1. #ifndef DATA\_TIMER
2. #define DATA\_TIMER  2000
3. #endif
4. #ifndef ACK\_TIMER
5. #define ACK\_TIMER   240
6. #endif                          // 可以用编译参数覆盖掉这些设置
7. #define MAX\_SEQ 255
8. #define NR\_BUFS 128              // 限于定时器数量，窗口最大只能开到128
9. #define inc(x) ((x) = ((x) + 1) % (MAX\_SEQ + 1)) // 模意义下自加
10. #define DATA\_MASK 1             // 判断帧是否携带数据的MASK
11. #define ACK\_MASK 2              // 判断帧是否携带ACK的MASK
12. #define NAK\_MASK 4              // 判断帧是否携带NAK的MASK
13. #define SHORTFRM\_MASK 6         // 判断是否是短帧的MASK
14. #define DATA 3                  // 数据帧
15. #define ACK 2                   // ACK帧
16. #define NAK 6                   // NAK帧

### 全局变量

1. **static** **bool** no\_nak = **true**;         // 未发送过nak标记
2. **static** **int** phl\_ready = 0;          // 物理层就绪标志
3. seq\_nr next\_frame\_to\_send;         // 发送窗口上界
4. seq\_nr ack\_expected;               // 发送窗口下界
5. seq\_nr frame\_expected;             // 接收窗口下界-1(即被接受的最后一个帧的编号)
6. packet  out\_buf[NR\_BUFS],          // 发送缓存
7. in\_buf[NR\_BUFS];           // 接收缓存
8. seq\_nr nbuffered;                  // 发送缓存中的有效包数目
9. seq\_nr too\_far;                    // 接收窗口上界
10. **bool** arrived[NR\_BUFS];             // 接收缓存有效标记

### 主函数变量

1. **int** arg;             // 用于接收wait\_for\_event的第二返回参数
2. event\_type event;    // 接收event
3. FRAME r;             // 存储frame的临时变量
4. **int** len;             // 保存从物理层收到的数据的长度的临时变量

## 模块结构

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

参数：

|  |  |
| --- | --- |
| frame | 帧起始字节地址 |
| len | 帧长度 |

功能：为len字节的数据计算CRC32，并将贴上了CRC32的共len+4字节数据送到物理层

1. **static** **void** send\_data\_frame(frame\_kind kind, seq\_nr frame\_nr, seq\_nr frame\_expected, packet buffer[]);

参数：

|  |  |
| --- | --- |
| kind | 帧种类 |
| frame\_nr | 欲发送的数据包的编号 |
| frame\_expected | 欲捎带ACK的编号 |
| buffer | 缓存 |

功能：将缓存buffer中编号为frame\_nr的数据包作为kind帧发送出去，同时捎带返回frame\_expected

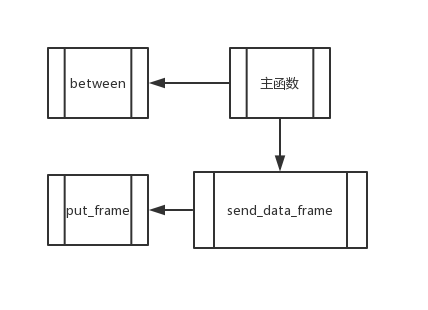
1. **static** **bool** between(seq\_nr a, seq\_nr b, seq\_nr c);

参数：

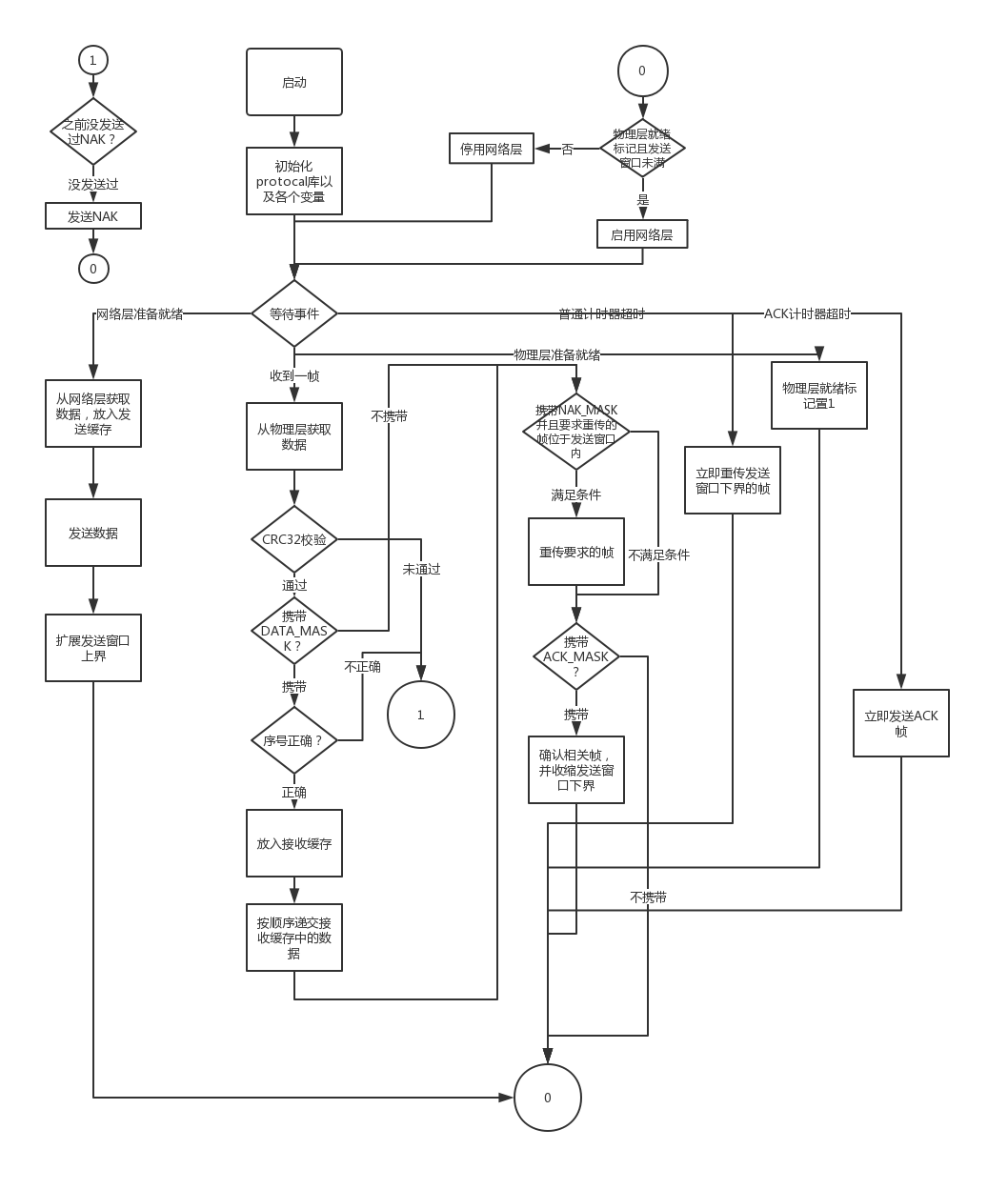
|  |  |
| --- | --- |
| a | 窗口下界编号 |
| b | 欲判断的编号 |
| c | 窗口上界编号 |

功能：判断b是否在a、c构成的窗口内

**调用关系图**



## 算法流程



# 实验结果分析

1. 描述你所实现的协议软件是否实现了有误码信道环境中无差错传输功能。

在有误码信道的环境下，接收到的数据帧计算出校验和错误，不会得到确认，接收方会立刻要求重传。缓存中的数据将被按顺序递交至网络层。因此所实现的选择重传协议实现了有误码信道环境中的无差错传输功能。

1. 程序的健壮性如何，能否可靠地长时间运行。

经过测试，程序能够在各种信道环境下长期无差错运行。换句话说，程序在长时间运行过程中并没有报错。

1. 协议参数的选取：滑动窗口的大小，重传定时器的时限，ACK 搭载定时器的时限，这些参数是怎样确定的？根据信道特性数据，分组层分组的大小，以及你的滑动窗口机制，给出定量分析，详细列举出选择这些参数值的具体原因。

**窗口大小256**

稍待确认

为避免窗口大小限制线路利用率，我们计算一下窗口最小需要多少。

信道速率8000bps，网络层数据包大小为256Bytes，信道延时270ms。一个完整数据帧共263Bytes，发送时间263ms，与信道延时相近。但是通过阅读protocol库的源码发现，共有129个定时器可供使用，也就是NR\_BUFS最大可为128。本着充分利用一切资源的原则，我将窗口开到了最大值也就是256，即MAX\_SEQ设置为255。注意到在回退N当中不能盲目的设置巨大的窗口，当一个数据包出错等待自然超时的时候，大窗口将导致发送方持续发送大量的包，这些包都会被丢弃，之后还会被重传，极大的降低线路利用率。但是选择重传并不会出现这个问题，因为之后发的包只要没出错，是不会被接收方忽视的。没有了小窗口的限制，当多个包出错时，后续的包的发送不受影响。所以在选择重传当中大的窗口大小有相当的好处。

**ACK计时器240ms**

如果ACK计时器设定太小，导致ACK帧的大量使用，则降低信道利用率，如果太大则导致重传，也会降低信道利用率。经过测试，240ms是一个不错的值

**重传定时器2000ms**

考虑一帧安全的被ACK所需要的最长时间：270\*2（来回时延）+263（去程传输时间）+240（ACK计时器）+6（ACK帧传输时间）=1049。这个是理论计算出来的值，但是实际应用起来并不是很好，接收方接收到后不一定立刻有数据帧回传，再加上回传数据可能在物理层排队队列中的等待时间，经过多次实验测试，发现3000ms是一个能带来良好结果的参数。

1. 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。

无差错环境下，a、b站都以洪水式产生分组时，信道利用率应趋近于256/163=97.34%

在差错率为的情况下，个bit可发送约48个数据包，其中有一个出错，需要重传。假定重传一定正确，则重传48个包需要约50个包的事件（多了一个rtt），则利用率约为

1. 实验结果分析

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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 | 700 | 57.74% | 96.60% | |
| 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” | 700 | 56.70% | 92.14% | |
| 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 | 700 | 96.59% | 96.58% | |
| 4 | –-flood | 站点A/B的分组层都洪水式产生分组 | 700 | 93.13% | 93.26% | |
| 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 | 700 | 64.73% | 63.95% | |

我的协议信道利用率与理论推算相差不大。

1. 存在的问题：在“表 3 性能测试记录表”中给出了几种测试方案，在测试中你的程序有没有失败，或者，虽未失败，但表现出来的性能仍有差距，你的程序中还存在哪些问题?这几种方案都成功了，但性能仍有差距。

我的程序在表3中给出的几种测试情况中表现良好，全部成功。性能与给定的参考数据相比各有千秋，除了高误码率洪水环境下与给定数据的42.0，73.6相比有较大差距外（我两个站点的数据比较平衡，算起线路利用率平均值的话，还是我高一些的），其他测试条件下与参考数据相差不大，可以说我的程序还不错

# 研究和探索的问题

1. 程序设计方面的问题

get\_ms()函数。通过阅读源码发现，get\_ms函数的实现原理是利用系统调用，获取精确到毫秒的一天中的事件，然后乘上相应系数，减去启动时的参数，从而获得毫秒级别的时间戳。

启动ack\_timer与普通timer不同之处在于普通timer额外考虑了物理层队列当中的内容，将物理层因素考虑在内更符合我们的需求（不然每个包从出发到超时的间隔将很不一样）

crc32中的DO宏。DO8将展开成8条命令，这会被编译器优化，达到空间并行同时计算的目的，提高程序效率。

1. CRC32校验能力

CRC32校验能力确实有限，一亿的随机数据即可产生数量可观的碰撞，但是实际信道的误码率不会更高。在本次实验中累计若干小时的测试中表现稳定，可见CRC32在这个信道误码率水平下足够有效。另外，如果实在不放心CRC32的精准度，可以在更高层实现其他的校验方式。数据链路层应保证足够的性能，而不能一味的追求完美无误。

To be effective, error control must be done in the highest layer, so that repeating it over and over in each of the lower layers is often unnecessary and infficient.

1. 软件测试方面的问题

网络情况复杂多变，协议要能适应各种情况。

-uf 测试最大线路利用率

-u 测试无误码情况下的最好性能

-f 测试大量数据时的协议性能

-b 1e-4 测试更糟糕的信道环境下的协议性能

为了同时对表中5种情况进行测试，我编写了一个test.sh脚本，同时启动测试。编写了一个show.sh脚本，用于分析日志文件并输出协议性能。这大大提高了测试效率。

1. 对等协议实体之间的流量控制

我的协议未解决对等协议实体之间的流量控制，但考虑到了上下层软件控制实体之间的数据流量控制问题。为了添加对等协议实体之间的流量控制功能，我们参照HDLC的方法，引入额外的帧种类，从而或显式或隐式的通知对等实体停发或继续发送数据。

# 实验总结和心得体会

1. 本次实验从编码到得到可以完全无误工作良好的协议程序共用了约4个小时。实际上，这四个小时我不光写了选择重传，还写了**回退N**的代码。两个滑动窗口协议都工作正常，但限于实验报告的篇幅，我只体现了选择重传的实验过程。
2. 编程工具方面，由于VS实在是过于臃肿，一开始我就决定在WSL上进行编码，可以说没有碰到任何问题。但由于我想使用Clion作为IDE,所以我又学习了CMake的用法，使得Clion可以解析实验源码。继而我还想用Clion进行构建，由于机器是Windows，而我是在Linux版本 的代码上进行编码的，所以我又对比了Linux实验环境和Windows实验环境，发现两者文件上仅仅差了一个getopt库。通过查阅资料了解到，getopt是Linux的一个内建库，Windows上没有。我把这个愿文件和头文件从VS版本拷贝到Linux版本文件夹中，并将其添加到CMakeLists中。除此之外，Windows上还需要链接wsock32库，两者都需要链接math库，我也悉数将链接命令添加到CMakeLists中。大功告成，可以直接利用Clion来编译了。为了灵活切换，我又建立了一个Release文件夹，在其中用WSL上用CMake生成了makefile，相当于脱离了实验材料中提供的makefile。除此之外，为了造福同学，我还用类似的方法弄了一个dev c++版本的kit，使得一些不想装VS和WSL的Windows同学能够愉快的做实验。
3. 在C语言的使用上面，最让我难受的就是自定义结构体的声明了：每次前面都需要加struct。于是我小小的typedef了一下。
4. 在协议方面，我曾遇到一个在设置ACK定时器的时候，错误的将ACK\_TIMEOUT而不是ACK\_TIMER（4）作为函数参数。前者是事件编号，是一个10以内的数字，这导致我的ACK定时器疯狂超时。还有一个是短帧的实现上，之前是3个字节，只是去掉了数据字段，后来发现seq也是不需要的，可以节省一个字节。
5. 在使用库的时候发现定时器实际上编号范围是0~128，而非实验指导书上所说的0~63。
6. 这次实验提升了我查阅文档，使用编程工具的能力，加深了我对两个滑动窗口协议的理解。

# 追加内容

我在实现选择重传协议之前先实现了回退N协议，在对两种协议实现和调参的过程中，我总结了两种协议之间的一些区别。

1. 回退N协议的窗口不可以随意开大，原因是当出错帧到达接收方时，接收方什么都不做，等待发送方超时重传。但发送方不知道这件事，只要发送窗口没达到上界，他就会一直发送新的帧，新的帧到达接收方后即被丢弃，大量的帧被丢弃，降低信道利用率。
2. 回退N协议相比选择重传协议更节省内存，因为他并不需要维护一个接收缓存。

## 回退N协议的参数选择

**窗口大小 7**

不同于选择重传，出错帧之后的正确帧会被直接丢弃，所以窗口开大会导致大量正确传输帧被多次传送，降低线路利用率。所以，窗口大小选择7，即取大于理论最优值的第一个符合2的次幂减1的数值。

**ACK计时器300ms**

选择理由与选择重传相同。

**重传定时器2000ms**

选择理由与选择重传相同。

## 实验结果分析

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 序号 | 命令选项 | 说明 | 运行时间(秒） | GoBackN算法  线路利用率(%) | |
| A | B | |
| 1 | -–utopia | 无误码信道数据传输 | 700 | 57.75% | 96.60% | |
| 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” | 700 | 45.80% | 77.80% | |
| 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 | 700 | 96.60% | 96.60% | |
| 4 | –-flood | 站点A/B的分组层都洪水式产生分组 | 700 | 75.65% | 75.68% | |
| 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 | 700 | 29.17% | 27.11% | |

与选择重传相比，回退N协议在信道质量较高时工作的几乎和选择重传协议一样好，但在误码率较高的情况下工作信道利用率很差。可以说，在RAM越来越便宜的当今，选择重传要明显优于回退N协议。