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>蒋砚军</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>2021211301</td><td></td><td colspan="2">2021210724</td><td>任飞</td><td colspan="2"></td></tr>
<tr><td></td><td></td><td colspan="2"></td><td></td><td colspan="2"></td></tr>
<tr><td></td><td></td><td colspan="2"></td><td></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协议，并且使用了 NAK 通知机制。</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

1 实验内容和实验环境描述

1.1 实验内容

本实验的任务是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，并对程序进行分析、调试和测试，实现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

1.2 实验环境

为了练习使用 Linux 的能力，我选择了在 WSL2(Windows Subsystem for Linux 2) 上运行本实验的 Linux 版本。虽然 WSL2 实质上是虚拟机，但基本没有性能损失（除了与 Windows 的文件系统中的文件交互以外）。因此能够胜任本实验。

2 实验历程与结果分析

2.1 简单选择重传协议

首先，我参考教材中的 Protocol 6 以及实验环境中提供的示例程序编写了一个简单的带有 NAK 的选择重传协议，使用了下面的的帧格式：

```
DATA Frame
+=========+========+========+===========+========+
| KIND(1) | ACK(1) | SEQ(1) | DATA(256) | CRC(4) |
+=========+========+========+===========+========+
ACK Frame
+=========+========+========+
| KIND(1) | ACK(1) | CRC(4) |
+=========+========+========+
NAK Frame
+=========+========+========+
| KIND(1) | ACK(1) | CRC(4) |
+=========+========+========+
```

双方各维护一个发送窗口、发送缓冲区以及接收窗口、接收缓冲区。每当发送一个消息帧时启动一个重发定时器，若在重发定时器超时后仍然没有收到对方的 ACK，则从发送缓冲区中重传消息帧并重启重发定时器。本协议采用累计重传机制，收到某一帧的 ACK 默认此前所有帧都已成功收到。当收到一个编号为接收窗口首位置的数据帧时启动 ACK 定时器，若 ACK 定时器超时前有数据帧发出则让该数据帧搭载 ACK，否则在 ACK 定时器超时后发送单独的 ACK。当收到的数据帧编号不是接收窗口首位置时，可以推测有帧丢失，对第一个没有收到的帧编号（也就是接收窗口首位置）发送 NAK，并检查收到的帧是否在接收窗口中，如果在接收窗口中则将收到的数据存入接收缓冲区。如果收到校验和错误的帧，也发送 NAK。此外，为避免一帧出错后多次请求重发同一帧，对于每一帧只允许重发一次 NAK，如果对某一帧已经发过了 NAK，则不再对该帧发 NAK。

一开始编写的代码总是在运行十几秒后出现错误，经过检查发现问题在于**没有在收到 NAK 时检查重传帧是否在发送窗口里**，如果不做这样的检查，就有可能重传一个不在发送缓冲区里的帧，从而引发错误。这是一个易错点。

解决这一问题后，协议可以在长时间的运行中保持稳定运行。现在协议的可靠性已经得到了保障，下一步是调整相关参数来提高协议效率。

首先计算一下理论最高信道利用率，一个数据帧为 263 字节，即 2104 位。默认配置下误码率为10^{-5}，则一个数据帧没有任何错误成功发送的概率为$(1-10^{-5})^{2104}=0.9792$，若假设捎带 ACK 以及单独 ACK、NAK 不会出错，则每个数据帧的重传次数为$\frac{1}{0.9792}=1.0213$，每个 263 字节的数据帧实际携带了 256 字节数据，因此理论最高信道利用率为$0.9792\times\frac{256}{263}=0.9531$，即 95.31%。实际由于 ACK、NAK 帧也会占用带宽，并且捎带 ACK 以及单独 ACK、NAK 也有可能出错，实际利用率必然低于该理论值。

下面开始介绍参数的选取：

2.1.1 ACK 定时器时限的选取

滑动窗口大小和重传定时器时限的取值都依赖于 ACK 定时器时限，因此 ACK 定时器的时限应该最先确定。

ACK 定时器的时限应该实现这样的目标：在有数据帧不断发出的情况下将 ACK 通过数据帧捎带传输，而在链路空闲时尽快地发送单独的 ACK 帧。在链路上有数据帧不断发出的情况下，由于每个数据帧大小为 263 字节，每隔约 263ms 就有一个数据帧发出（更精确地说，被放入物理层缓冲）。这样，如果我们将 ACK 定时器的时限设置为略大于 263ms，就可以保证在有数据帧不断发出的情况下 ACK 定时器不会超时，而一旦链路空闲就能尽早超时来触发单独的 ACK 帧。

综上，暂时设置 ACK 定时器时限为 300ms。

2.1.2 重传定时器时限的选取

理论上，重传定时器时限应当略大于数据未出错时的最大延迟。

假设信道不会出错，考虑从一个数据帧从开始发出到完整接收到该帧的 ACK 的最大延迟：由于链路的速率为 8000bps，一个数据帧为 263 个字节，发送方开始发送后 263ms 整个数据帧被完全发出，再加上链路的延迟 270ms，需要 533ms 来让接收方完全接收到这一数据帧。收到后接收方可能在 ACK 定时器时限的 300ms 内将 ACK 通过数据帧捎带传输，也可能在 300ms 后发送单独的 ACK 帧。由于 ACK 帧远短于数据帧，延迟最大的情况是接收方在即将达到 300ms 的 ACK 定时器时限时发出了一个捎带 ACK 的数据帧，这一帧同样需要 533ms 才能完全到达发送方，因此理论最大延迟为 533+300+533=1366ms。

综上，暂时设置重传定时器时限为 1500ms。

2.1.3 滑动窗口大小的选取

根据上一节的分析，从发送数据到收到 ACK 的最大延迟为 1366ms。链路的速率为 8000bps，延迟为 270ms，一个数据帧为 263 个字节，故整个链路往返的带宽-延迟乘积（用帧的数量来表示）为$\frac{1.366\times8000}{207\times8}=5.19$。因此滑动窗口的大小必须大于这个值才能充分利用信道。

综上，暂时设置滑动窗口大小为 8，这样需要 16 个帧编码，正好占用四位。

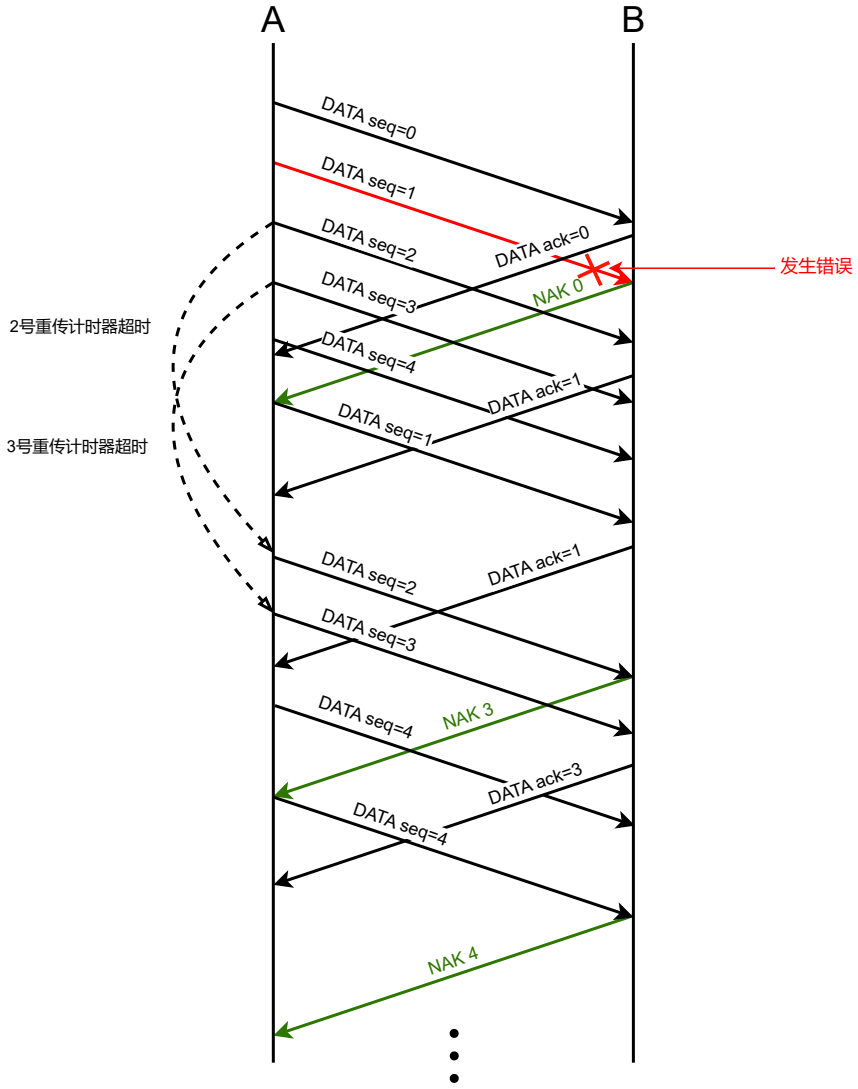
然而，上面设计的协议和参数虽然看似合理，但是实际运行的效果非常差。在10^{-5} 默认误码率的 flood 模式下，一方只有 70% 的信道利用率，而另一方只有 80% 的星岛利用率，可见本协议还有很大的改进空间。

2.2 对简单选择重传协议的思考与改进

通过对日志文件进行分析，我发现问题在于**偶然的一次错误会导致在这之后信道上持续出现冗余数据**。

下面举一个例子予以说明：A 连续向 B 发送了 0 号帧、1 号帧、2 号帧……，B 收到了 0 号帧，A 也收到了 B 发来的 0 号帧的 ACK，但是 A 发送的 1 号帧发生了错误。B 接收到错误的 1 号帧后会向 A 发送 NAK 请求重传 1 号帧。在 B 收到 A 重传的正确的 1 号帧之前，B 发给 A 的数据帧的捎带 ACK 都只能是 0，A 也就无从得知 2 号帧、3 号帧是否被 B 正常接收了。由于定时器的时限很短，A 的 2 号帧的定时器很快就会超时（甚至 3 号帧的定时器也会超时），于是 A 重发 2 号帧，但事实上在这个时候，B 不仅已经正确地收到过了 2 号帧，而且已经收到了重传的 1 号帧，因此 B 的接收窗口已经向前滑动了，2 号不再处于 B 的接收窗口内。根据协议的处理逻辑，B 会发送一个 NAK 帧来报告错误。然而，此时 B 其实没有任何帧是需要重传的，这个多余的 NAK 帧会导致 A 发来一个多余的重传帧，这个多余的重传帧到达 B 时又已经不在 B 的接收窗口内了，于是 B 又发送了一个多余的 NAK 帧……如此往复，可以看到信道上偶然的一次错误会导致此后信道上持续出现冗余数据。

为了简洁起见，下面的示意图省略了 A 发出的数据帧的 ACK 字段，B 发出的数据帧的 SEQ 字段，以及部分不重要的帧。



上面的分析已经指出了一个改进措施——如果收到的数据帧编号不在接收窗口范围内，那么应该直接忽略这一帧，而不要发送 NAK。教材中的 Protocol 6 对这一细节的处理方法是和我原先的做法一致的，我参考的时候也没有意识到这里需要改进，可见教材中的代码确实也有不足之处！

修正这一细节后，在10^{-5} 默认误码率的 flood 模式下协议在多数时候双方都能达到 90% 以上的信道利用率，个别时候甚至达到理论值，但偶尔也会有一方跌到 90% 以下，在长时间运行后双方的信道利用率稳定在约 91.8%。

现在的改进余地也是比较明显的，上面的改进虽然避免了无限的冗余 NAK 和重传，但每次出错时的一个或两个冗余重传帧是无法避免的（即上面例子中的 2 号帧和 3 号帧）。要想避免这一问题，就要延长重传定时器的时限，使得一个错误帧的重传帧到达时其后面的帧的重传定时器还没有超时。我们可以粗略地假设 2 号帧被 B 完全接收和 1 号帧的重传帧被 A 发出这两个事件同时发生（这个假设是合理的，但为了行文简洁不再详述），那么再过 263+270=533ms，1 号帧的重传帧才会完全到达 B，此时 B 的接收窗口滑动，ACK 定时器启动。也就是说重传定时器的时限应该在理论最大延迟 1366ms 的基础上再加 533ms，即 1899ms，在此我选择将重传定时器的时限设为 2000ms。同时滑动窗口的大小也要增大，以避免在出错重传时滑动窗口过小阻碍后面的帧继续发送。简单起见直接将滑动窗口大小翻倍，即改为 16，现在帧需要 32 个编码，即 5 位。

现在这一版本在10^{-5} 默认误码率的 flood 模式下长时间运行后双方的信道利用率稳定在约 95.0%，与理论值 95.31% 对比已经非常令人满意了。现在这一版本已经较好地完成了本实验的要求。

接下来尝试高误码率信道。通过类似的计算，可以得到在10^{-4} 误码率的信道下理论最大信道利用率为 78.87%。但实际运行发现这一版本在10^{-4} 误码率的 flood 模式下长时间运行后双方的信道利用率稳定在约 58%，与理论值差距较大，可见协议仍然存在改进的空间。

2.3 一个收效甚微的尝试——手工缓冲区

查看高误码率下的日志，发现经常会出现一连串的冗余重传帧。对于这一情况我提出了一个猜想：在向物理层发送重传帧的时候没有检查物理层是否空闲，这样在误码率高需要很多重传的情况下就会导致物理层缓冲区堆积了多个帧。问题在于一个帧的 ACK 信息是在其被放入物理层缓冲区时确定的，但到其真正被发出时这一端的 ACK 信息可能已经被更新了，而我们无法到物理层缓冲区里去修改信息，导致实际发出的 ACK 总是滞后的，这就引起了多余的重传帧。

将物理层缓冲区长度输出到日志，发现很多时候物理层缓冲区长度都在 400 字节以上，**似乎**也就是说也就是至少有一个完整的帧在物理层缓冲区里等待，我的猜想是有道理的。

既然物理层缓冲区无法修改，那么就自己实现一个可以修改的缓冲区。照着这个思路我开始修改程序，在发送帧时检查物理层是否空闲，如果空闲则直接发送给物理层；如果不空闲则加入发送队列。当物理层报告空闲时，就检查发送队列里是否有帧，如有则将其取出，更新其 ACK 值，并发送给物理层。这样每个帧在发出时携带的都是最新的 ACK。

完成了上述修改后，测试发现信道利用率与原先相比几乎没有变化。我又尝试了一系列的策略，包括：

- 如果发送队列不为空则不发送单独 ACK 帧和 NAK 帧。因为发送队列里的帧已经可以携带最新的 ACK 信息了；而 NAK 帧在需要在发送队列里等待较长时间的情况下还不如直接等待发送方超时重传。
- 将发送队列修改为双端队列，并将 NAK 帧设置为高优先级帧。将高优先级帧插入到发送队列的末尾，而将低优先级帧插入到发送队列的开头。这样相当于让 NAK 帧“插队”提前发送。

- 进一步延长超时定时器时限和滑动窗口大小。

在做了大量的修改和实验后，信道利用率的提升仍然非常有限，相比原来改进的简单选择重传协议只提升了约 3%。而且在调试过程中还遇到了大量细节问题，通信偶尔会进入死锁状态，甚至会在正常运行了 300s 后突然出现错误。

由于手工缓冲区的方案效率提升甚微，实现细节复杂，且稳定性差，最终的协议里并没有采用手工缓冲区。

实际上，查看 `protocol.c` 文件后发现，`phl_sq_len` 函数返回的实际上是物理层缓冲区长度的两倍，也就是说实际上物理层缓冲区通常只有 200 多字节，没有出现帧堆积的情况，也难怪手工缓冲区的方案收效甚微了。

2.4 最终的协议——累计确认与逐一确认结合

重新思考出现冗余重传帧的根本原因。回到最开始的例子。由于我们采用累计确认机制，在 B 收到 A 重传的正确的 1 号帧之前，B 发给 A 的数据帧的捎带 ACK 都只能是 0。即使 B 已经正确地收到了 2 号帧、3 号帧，也不能及时地反馈给 A。在低误码率情况下可以通过延长超时定时器时限来很大程度上缓解这个问题。而在高误码率情况下由于很有可能在短时间内发生多次错误，使双方都忙于错误处理，导致延迟大大增加。实验发现在10^{-4} 误码率信道下将超时定时器时限设为 5000ms，滑动窗口大小设为 32 时，尽管出现问题的频率下降了，但一旦出现问题，就会连续超时很多个帧，在日志中可以观察到一次连续重发了多达 15 个冗余帧。可见继续延长超时定时器时限无法解决问题。

根本的解决方案在于让 A 知道 B 已经正确地收到了某个出错帧后面的帧，这样 A 就再也不会重传这些帧了。也就是说在这里使用逐一确认是更加合适的做法。不过考虑到在没有出错的情况下逐一确认的大量 ACK 帧会占用较大带宽，我最终采取了累计确认与逐一确认结合的策略。在正常情况下采用累计确认，一旦收到错误帧，则在错误帧后面收到的正确帧使用逐一确认的方式反馈给 A。

新的帧格式如下，逐一确认 ACK（Sep ACK）的 KIND 字段值为 4。Sep ACK 帧格式上与 ACK 帧相同，只是一个帧的确认并不暗示前面的帧都已确认。

```
DATA Frame
+=========+========+========+===========+========+
| KIND(1) | ACK(1) | SEQ(1) | DATA(256) | CRC(4) |
+=========+========+========+===========+========+
ACK Frame
+=========+========+========+
| KIND(1) | ACK(1) | CRC(4) |
+=========+========+========+
NAK Frame
+=========+========+========+
| KIND(1) | ACK(1) | CRC(4) |
+=========+========+========+
Sep ACK Frame
+=========+========+========+
```

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

超时定时器和 ACK 定时器仍然采用 2000ms 和 300ms 的设置。根据实验结果滑动窗口大小设为 16 时有时会成为瓶颈，故设置滑动窗口大小为 32，即帧编号共有 64 个，占 6 位。

现在，在10^{-4} 误码率的 flood 模式下长时间运行后双方的信道利用率稳定在约 76.5%，与理论值 78.87% 相比已经足够令人满意。

那么10^{-3} 的误码率又如何呢？使用与之前类似的计算，可以得到10^{-3} 误码率下的理论值是 11.86%，而本协议的利用率约为 8%。实际上，在如此高的误码率下，几乎每一帧都要重传多次，而且捎带 ACK 以及单独 ACK、NAK 不会出错的假设也完全不再成立。在这样的信道传输信息更应该使用里德-所罗门编码等高强度的**纠错码**，而不是依赖重传，因此不再研究。

本协议即为最终使用的协议。下面的软件设计章节均针对本协议的程序描述。

3 软件设计

3.1 数据结构

3.1.1 数据类型

本程序使用 `typedef` 定义了两个数据类型：

```
1 typedef unsigned char buffer_t[PKT_LEN];
2 typedef unsigned char seq_nr;
```

`buffer_t` 为一个数据包的缓存（实质为 `PKT_LEN` 字节长的数组）。`seq_nr` 为帧编号的类型。

此外还定义了结构体 `FRAME`，表示一帧的格式，此结构体的定义修改自示例程序。具体的格式定义已在上文说明。

```
1 struct FRAME {
2     unsigned char kind;
3     seq_nr ack;
4     seq_nr seq;
5     buffer_t data;
6     unsigned int padding;
7 };
```

3.1.2 宏定义常量

首先是关于帧的 KIND 字段的常量。

```
1 #define FRAME_DATA 1
2 #define FRAME_ACK 2
3 #define FRAME_NAK 3
4 #define FRAME_SEP_ACK 4
```

以下常量都使用了教材中的命名，不再赘述。

```
#define MAX_SEQ 63
#define NR_BUFS ((MAX_SEQ + 1) / 2)
#define DATA_TIMER 2000
#define ACK_TIMER 300
```

3.1.3 全局变量

为了更好地体现“滑动窗口”的思想，本程序的变量围绕滑动窗口的概念命名。

发送窗口相关变量以 sw(sending window) 开头，用左闭右开区间 `[sw_begin, sw_end)` 表示发送窗口的区间，`sw_size` 表示发送窗口的大小，`sw_buffer` 表示发送窗口的缓冲区。

相关变量以 rw(receiving window) 开头，用左闭右开区间 `[rw_begin, rw_end)` 表示接收窗口的区间，`rw_buffer` 表示接收窗口的缓冲区，`arrived` 表示缓冲区是否已经收到数据。

`phl_ready` 表示物理层是否空闲。`sended_nak` 表示是否已经针对当前接收窗口的第一帧发送了 NAK。

3.2 模块结构

`next`,`prev` 和 `between` 三个函数用于处理帧编号。函数 `next` 和 `prev` 分别计算在模 `MAX_SEQ` 意义下 x 加 1 的值和 x 减 1 的值，相比于教材程序中的宏 `inc`，这两个函数只求值而不会自动赋值，使得这两个函数使用起来更加灵活。`between` 则用于在模 `MAX_SEQ + 1` 意义下检查一个值是否在给定区间内。为了提高效率，这三个函数都是内联函数。

`put_frame` 函数给一帧加上 CRC-32 校验和并发送给物理层。参数 `frame` 为帧的起始地址，参数 `len` 为不含校验和的帧长度。

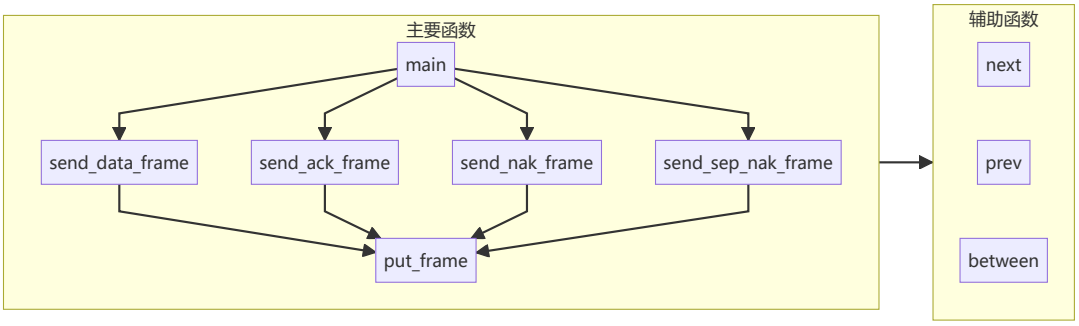
`send_data_frame` 函数发送数据帧，参数 `seq` 为帧编号，函数会根据帧编号从发送窗口缓冲区中找到对应的数据。

`send_ack_frame` 函数发送 ACK 帧。

`send_nak_frame` 函数发送 NAK 帧。

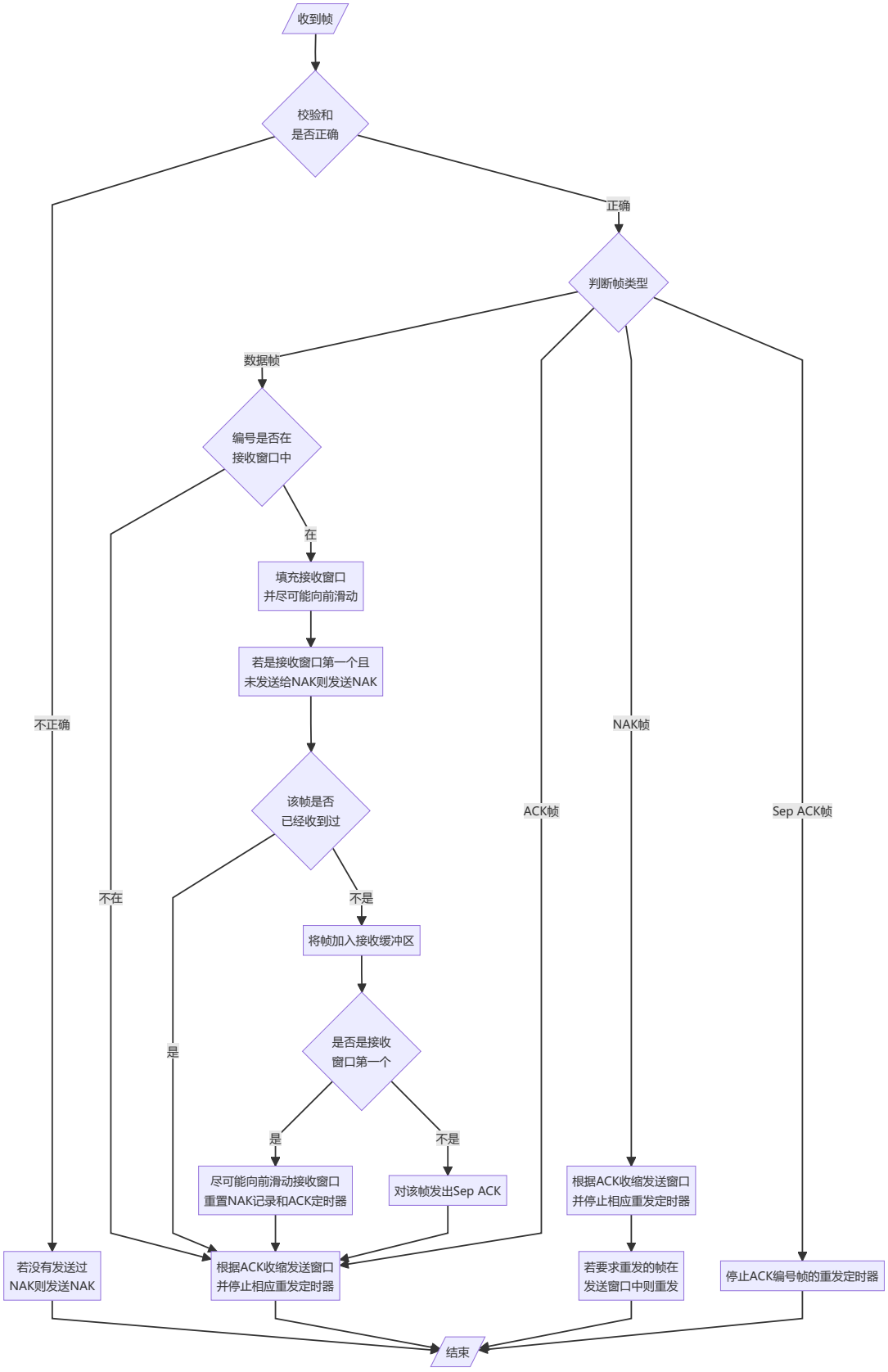
`send_sep_ack_frame` 函数发送 Sep ACK 帧，参数 `ack` 为要逐一确认的帧编号。

程序调用关系图：



3.3 算法流程

程序使用事件循环的结构。大多数事件的处理逻辑都比较简单，下面仅展示收到帧事件的处理。


收到帧
校验和
是否正确
不正确
正确
判断帧类型
数据帧
ACK帧
NAK帧
Sep ACK帧
编号是否在
接收窗口中
在
不在
填充接收窗口
并尽可能向前滑动
若是接收窗口第一个且
未发送给NAK则发送NAK
该帧是否
已经收到过
是
不是
将帧加入接收缓冲区
是否是接收
窗口第一个
是
不是
尽可能向前滑动接收窗口
重置NAK记录和ACK定时器
对该帧发出Sep ACK
若没有发送过
NAK则发送NAK
根据ACK收缩发送窗口
并停止相应重发定时器
根据ACK收缩发送窗口
并停止相应重发定时器
若要求重发的帧在
发送窗口中则重发
停止ACK编号帧的重发定时器
结束

4 实验结果分析

经过多个版本的迭代，最终版本的程序实现了在有误码信道环境中的无差错传输功能。程序的健壮性优秀，能够可靠地长时间运行。协议效率高，信道利用率接近理论最大值。较好地完成了实验的两个要求。

程序的实现细节、协议参数的选取、相关的理论分析以及遇到的问题已在前文详细描述过。

下面给出最终程序的性能测试表：

序号	命令选项	说明	运行时间 / 秒	线路利用率/ %	
				A	B
1	`--utopia`	无误码信道数据传输	850	54.17	96.97
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”	850	53.18	94.96
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	850	96.97	96.97
4	`--flood`	站点 A/B 的分组层都洪水式产生分组	850	94.84	94.88
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}	850	77.08	76.60

如果要进一步提高信道效率，可采用下面的方案：

- 减小帧头的大小，如本协议中的 KIND、ACK、SEQ 字段实际上只需要两个字节即可存储
- 使用纠错码代替校验码，这样绝大多数的重传都可以避免
- 对发送的数据进行压缩

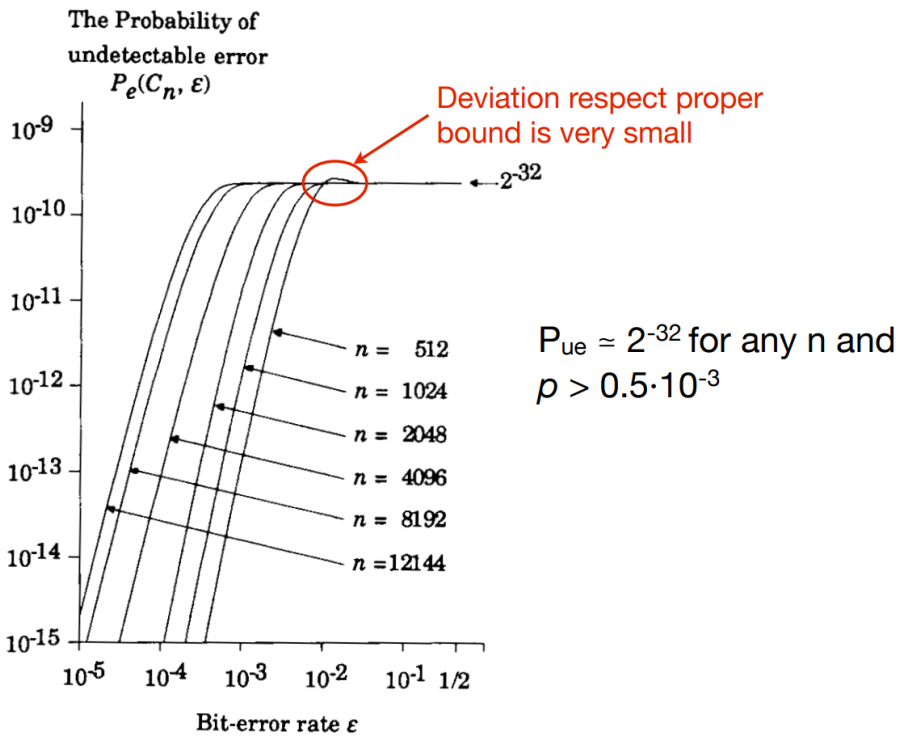
上面这些方案的具体实现偏离了本实验的重点，故不再展开说明。

5 研究和探索的问题

5.1 CRC 校验能力

CRC 校验的确不能 100% 检出所有错误，但遗漏错误的概率非常低。本协议采用的 CRC-32 能够检测所有一位错误、两位错误、奇数个数位错误和长度小于等于 32 位的突发错误。根据相关研究[1]（见下图），263 字节，即 2104 位的消息在误码率为10^{-5} 的信道上出现遗漏错误的概率远低于10^{-15}，估计可达10^{-18}。

[1] https://www.ieee802.org/3/bv/public/Jan_2015/perezaranda_3bv_2_0115.pdf



若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为 50%，则有

$$\frac{10^{18}}{0.5 \times \frac{1000}{263} \times 60 \times 60 \times 24 \times 365} = 1.668 \times 10^{10}$$

即平均1.668×10^{10} 年才会发生一次分组层错误。实际上现实中的信道更多地是发生突发错误而不是零散错误，而 CRC-32 保证能够检测出长度小于等于 32 的突发错误，上面的结果仍然显得保守，因此更加不必担心了。

如果还需要降低发生分组层误码事件的概率，可以使用更长的纠错码，代价是会降低信道利用率。

5.2 CRC 校验和的计算方法

程序使用的查表法和手工进行二进制“模 2”除法求余数的算法是等效的。查表法的原理是先预处理出从 0x00 到 0xFF 的每个字节后面填充 0 后的 CRC-32 余数，并存储到查找表中。计算时以字节为单位进行“模 2”除法，利用查找表快速“试商”，从而加快效率。

一个值得注意的点是 crc32.c 中查找表和函数中的数值都是**多项式次数高的在低位，多项式次数低的在高位**。

可使用下面的 C 语言代码生成 crc32.c 中的查找表。

```c
#include <stdio.h>
#define CRC 0xEDB88320
int main() {
    printf("static const unsigned int crc_table[256] = {");
    for (int i = 0; i < 256; i++) {
        if (i % 5 == 0) printf("\n ");
        unsigned int v = i;
        for (int j = 0; j < 8; j++) {
```

```
9             v = v & 1 ? (v >> 1) ^ CRC : v >> 1;
10        }
11        printf("0x%08xL", v);
12        if (i < 255) printf(", ");
13    }
14    printf("\n};\n");
15 }
```

在 x86 计算机上计算 16 位整数和 32 位整数的速度相同，因此使用查找表法为某一帧计算 32 位 CRC-32 校验和和计算 16 位 CRC-16 校验和所花费的 CPU 事件是一样的。

RFC1662 中的函数 `pppfcs32` 中的参数 `fcs` 是用于给校验和设置一个初值，以便进行增量计算。

5.3 程序设计方面的问题

5.3.1 协议软件的跟踪功能方面

协议软件的跟踪功能有助于获取详略得当的日志文件用以辅助调试和分析。我的程序合理地使用了 `lprintf`, `dbg_frame`, `dbg_event`, `dbg_warning` 系列函数，对调试分析有很大的帮助。

5.3.2 `get_ms` 函数方面

`get_ms` 函数可以通过 C 语言标准库中的 `clock` 函数实现，但 `protocol.c` 中实际直接调用了操作系统的函数，也许是可以获得更高的精度。

5.3.3 变长参数函数方面

`lprintf`, `dbg_frame`, `dbg_event`, `dbg_warning` 这些变长参数函数是通过 C 语言的 `va_start`, `va_end`, `va_arg`, `va_list` 等宏实现的。

5.3.4 定时器函数方面

`start_timer` 函数把截止到调用函数时刻为止已经放入物理层发送队列的数据发送完后才启动计时，这是因为 `start_timer` 函数用于重传定时器，应该在一个数据帧真正开始发出去的时候才开始定时。定时器超时前调用 `start_timer` 函数的效果是重新开始计时，这是因为在定时器到时前收到 NAK 帧的情况下需要重新开始重传定时器，方便代码编写。

`start_ack_timer` 函数在调用后立即开始计时，这是因为 `start_ack_timer` 函数用于 ACK 定时器，需要的就是如果在函数调用后一段时间内没有数据帧发出则发送单独 ACK 帧。定时器超时前调用 `start_ack_timer` 函数的效果是维持定时器当前的剩余时间，这是因为我们离最早的那个未发送 ACK 的帧到达的时间超过限制就应该发送单独 ACK 帧，不应该因为后续又有数据帧传来而延长等待时间。

可见 `start_timer` 和 `start_ack_timer` 各自的特点都是为了适应它们的使用场景而设计的，很大程度上方便了实验代码的编写。

5.4 对等协议实体之间的流量控制

本程序实现了面向物理层的流量控制。在物理层繁忙时通知网络层暂停发送数据包，在物理层空闲时通知网络层恢复发送数据包。

本程序没有实现面向网络层的流量控制。本实验假设网络层随时能够不限量地接收数据链路层提交的数据包。但在现实中网络层也有可能拥堵。如果要实现面向网络层的流量控制，可以使用如下两种方案之一：

- 在网络层拥堵时直接丢弃收到的帧。
- 增加一些控制帧，用于在某一方的网络层拥堵时通知另一方停止发送数据或降低发送数据的速率；在某一方的网络层恢复时通知另一方恢复发送数据或提高发送数据的速率。

5.5 与标准协议的对比

实验协议离实用的差距和遗漏的重要问题：

1. 本实验隐藏了成帧这一非常重要的过程。
2. 一个实用协议必须实现面向网络层的流量控制这一非常重要的功能。
3. 一个实用协议应该能够根据链路的带宽、延迟和误码率动态地协商各项参数而不是针对特定情景手动设置。
4. 一个实用协议除了提供可靠服务外也应该提供不可靠服务，即支持发送不需要加以确认的帧。
5. 一个实用协议还应该能够传递各种控制信息，控制链路的连接与断开等。
6. 本协议只能在点对点链路上使用，没有实现 mac 子层的功能，无法在共用介质上使用。

6 实验总结和心得体会

6.1 实验总结

完成本实验总共花费了约 18 小时的时间。这其中我把较多的时间花费在了调试和分析手工缓冲区和相关策略上，尝试了很久不同的方案和参数选取，浏览了很久输出日志，还花了很长的时间寻找 bug，但最终也没有取得较好的效果。后来想到了累计确认和逐一确认结合的方法，进展就相当顺利了。

由于我的编程经验还是比较丰富的，在编程工具和编程语言方面没有遇到大的问题，还找到了一些开发库的问题。

6.2 开发库存在的问题

本实验的开发库存在一些问题，包括：

1. 在 Linux 版的开发库中，`lprintf.c` 中 `lprintf` 和 `__v_lprintf` 这两个函数的返回值类型为 `size_t`，但在 `lprintf.h` 中 `lprintf` 和 `__v_lprintf` 的返回值类型为 `int`，两者不匹配。
2. `protocol.c` 的 689 行中的 `DBG_FRAME` 应改为 `DBG_EVENT`。
3. 由于逻辑上的一个字节在物理层中被处理为两个字节，`phl_sq_len` 函数的返回值是实际值的两倍，容易造成误解。
4. 给出的示例实现中 `datalink.h` 中的示意图的 DATA Frame 中 SEQ 和 ACK 字段反了，DATA 字段应固定为 256 字节。

6.3 心得体会

完成本实验的过程带给了我很多收获。

首先就是在反复的修改、调试、优化中对滑动窗口协议有了更加深刻全面的认识，搞清楚了滑动窗口协议的每一个细节。我在实验中还遇到了很多没有写入正文的细节问题，例如：超时定时器的编号不应该对 NR_BUFS 取模，因为在超时重传时需要发送重传帧的实际编号而不是对 NR_BUFS 取模后的编号；收到校验和错误的帧后究竟是否应该先检查帧长度再决定是否发生 NAK 帧；课本代码中启动 ACK 定时器的条件问题；ACK 定时器时长的设置与空闲状态下的通信效率问题；还有在调试时在日志中观察到的复杂时序问题。这些细节问题如果只学理论不上手写是很难体会到的。看似简单的内容背后也隐藏着大量值得挖掘的东西。

其次我也认识到了教材中的代码也并不完美，仔细思考便可做出改进。这次实验破除了我原先“累计确认一定优于逐一确认”的想法，实际上在发生错误时使用逐一确认更能发挥选择重传协议的优势。

再者就是体验了网络协议的设计，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，锻炼了调试程序和分析与网络通信相关的问题的能力。此外在回答本实验“研究和探索的问题”中的问题时也学到了一些关于 CRC 校验和实用数据链路层协议方面的知识。

7 源程序文件

见 datalink.c 和 datalink.h。