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 实验报告

毛子恒 庞仕泽
2019211397 2019211509
北京邮电大学 计算机学院

日期：2021 年 6 月 1 日

1 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 8000 bps 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。

1.2 实验环境

- macOS Big Sur 11.2.3
- Visual Studio Code 1.55.0
- Apple clang version 12.0.0
- GNU make 3.81

2 软件设计

2.1 数据结构

```c
typedef unsigned int seq_nr;
typedef unsigned char frame_kind;

typedef struct
{
    frame_kind kind;              // 帧的种类
    unsigned char ack;            // ACK 序号：最后一个被成功接收的帧序号
    unsigned char seq;            // 帧序号
    unsigned char data[MAX_PKT]; // 数据
    unsigned char padding[4];     // CRC 校验段
} Frame;                          // 帧，与物理层通信；seq,data,padding 只在 DATA 类型帧中有效

typedef struct
{
    unsigned char data[MAX_PKT];
    size_t len;
} Packet; // 包，与网络层通信

static bool phl_ready = false;  // 物理层是否就绪
static Packet out_buf[NR_BUFS]; // 接收窗口
static Packet in_buf[NR_BUFS];  // 发送窗口
static bool arrived[NR_BUFS];   // 标记帧是否到达的 bit map

// 以下是主程序中的变量

seq_nr ack_expected = 0;        // 期望收到 ACK 的帧序号，发送窗口的下界
seq_nr next_frame_to_send = 0; // 下一个发出的帧序号，发送窗口的上界 +1
seq_nr frame_expected = 0;      // 期望收到的帧序号，接收窗口的下界
seq_nr too_far = NR_BUFS;       // 接收窗口的上界 +1
int i;
Frame r;                // 接收到的帧
int len;                // 接收到的帧的长度
seq_nr nbuffered = 0; // 接收到的帧的数量
int event;
int arg; // 事件接收到的参数
```

2.2 模块结构

```c
// 判断在窗口中是否有 a<=b<c
static bool between(seq_nr a, seq_nr b, seq_nr c);

// 添加校验段，发送 DATA 帧到物理层
static void put_frame(unsigned char *frame, int len);

```

```
// 根据不同类型构建帧、发送帧、处理计时器
static void send_dllayer_frame(frame_kind fk, seq_nr frame_nr, seq_nr
    ↪ frame_expected);
```

模块间调用关系如图 1。

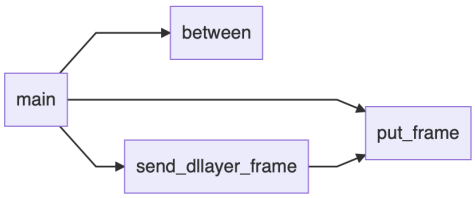

图 1: 模块调用关系图

2.3 算法流程

选择重传协议的算法流程如图 2。

3 实验结果分析

3.1 程序正确性和健壮性分析

程序实现了在有误码信道环境中的无差错传输功能。在误码率较高的情况下仍然能够稳定运行超过 1 小时并且保持可观的信道利用率，可见程序健壮性较好。

3.2 参数选择

3.2.1 窗口大小选择

在无差错信道的情况下，窗口大小 W 应当满足以下表达式：

$$\begin{cases} t_p = 270\ ms \\ t_f = \dfrac{263 * 8}{8000} \times 1000\ ms \\ \dfrac{W \cdot t_f}{2t_f + 2t_p} \geq 1 \end{cases}$$

计算出 $W \geq 4.05$。

当误码率较高时，取帧错误的概率 $P = 0.2$：

$$\begin{cases} t_p = 270\ ms \\ t_f = \dfrac{263 * 8}{8000} \times 1000\ ms \\ \dfrac{W(1-P) \cdot t_f}{2t_f + 2t_p} \geq 1 \end{cases}$$

计算出 $W \geq 5.06$。因此，当误码率较高时，至少会用到 6 个发送窗口。

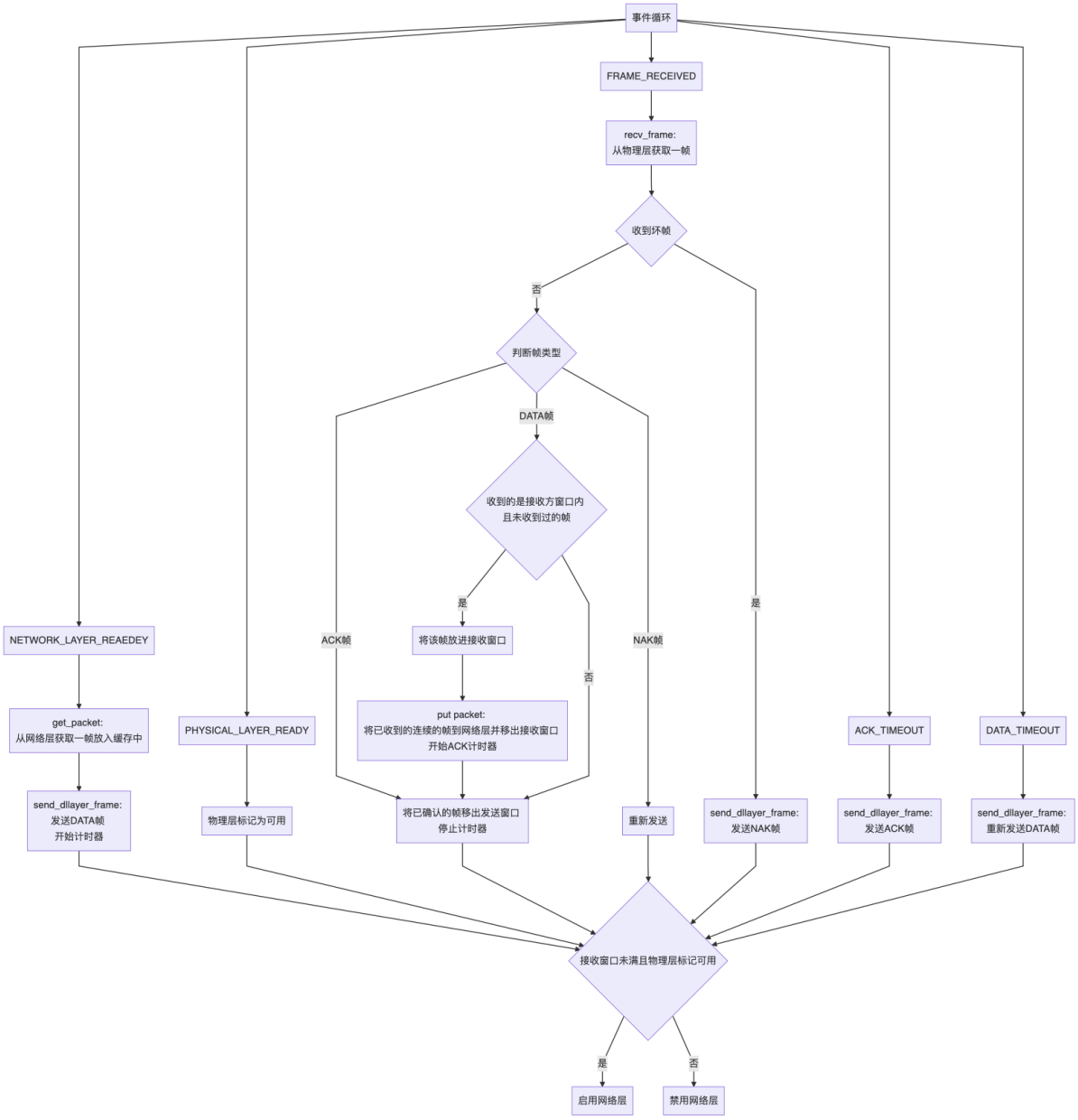
事件循环
FRAME_RECEIVED
recv_frame:
从物理层获取一帧
收到坏帧
否
判断帧类型
DATA帧
收到的是接收方窗口内
且未收到过的帧
是
否
ACK帧
NAK帧
是
将该帧放进接收窗口
put packet:
将已收到的连续的帧到网络层并移出接收窗口
开始ACK计时器
NETWORK_LAYER_REAEDEY
get_packet:
从网络层获取一帧放入缓存中
send_dllayer_frame:
发送DATA帧
开始计时器
PHYSICAL_LAYER_READY
物理层标记为可用
将已确认的帧移出发送窗口
停止计时器
重新发送
send_dllayer_frame:
发送NAK帧
ACK_TIMEOUT
send_dllayer_frame:
发送ACK帧
DATA_TIMEOUT
send_dllayer_frame:
重新发送DATA帧
接收窗口未满且物理层标记可用
是
否
启用网络层
禁用网络层

图 2: 算法流程图

通过在程序运行时打印发送方窗口大小，我们发现在一定概率下会用到 16 个以上的发送窗口，因此，为了最大化信道利用率，把窗口大小设为 32，MAX_SEQ 为 63。

3.2.2 超时参数选择

数据帧超时参数 T_{data} 和 ACK 帧超时参数 T_{ack} 应当满足以下条件：

$$\begin{cases} T_{data} \geq T_{ack} + 2t_p + 2t_f \\ W \cdot t_f \geq T_{ack} \end{cases}$$

由于协议设计时不考虑整个包丢失的情况，所以理论上选择重传协议的 T_{data} 可以设计的很大，而 T_{ack} 需要有下限，否则协议效率会变低，T_{ack} 的上限需要满足如上两个不等式，保证不会因为 T_{ack} 过长而导致发送方 DATA 计时器超时，同时保证不会因为 T_{ack} 过长而使得发送窗口满。

经实际测试，取回退 N 协议的 $T_{ack} = 280\ ms, T_{data} = 1000$ ms，选择重传协议的 $T_{ack} = 280\ ms, T_{data} = 5000$ ms。

3.3 理论分析

由于 ACK 帧和 NAK 帧长度很短，占比极小，所以假设这些帧能 100% 传输。设当前信道误码率为 p，每个帧长为 263 字节，一次发送成功的概率为：

$$P = (1-p)^{263\times 8} = (1-p)^{2102}$$

假设一帧传输需要的次数为随机变量 X，则其期望为：

$$E(X) = \sum_{i=1}^{\infty} i \cdot (1-P)^{i-1} \cdot P = \frac{1}{P}$$

因此理论最大信道利用率为：

$$\eta = \frac{256}{263 \cdot E(x)}$$

代入 $p = 0$，得：

$$\eta \approx 97.33\%$$

代入 $p = 10^{-5}$，得：

$$\eta \approx 95.31\%$$

代入 $p = 10^{-4}$，得：

$$\eta \approx 78.88\%$$

3.4 测试结果分析

回退 N 协议的测试结果见**表 ??**。

表 1: 回退 N 协议 性能测试记录表

序号	命令	说明	运行时间（秒）	接收方线路利用率（%）		存在的问题
				A	B	
1	`datalink -u A` `datalink -u B`	无误码信道数据传输	1247	53.26	96.97	
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，慢发 100 秒”	1219	40.27	75.15	与理论性能差距较大
3	`datalink -u -f A` `datalink -u -f B`	无误码信道，站点 A 和站点 B 的分组层都洪水式产生分组	1200	96.97	96.97	
4	`datalink -f A` `datalink -f B`	站点 A/B 的分组层都洪水式产生分组	1201	77.40	77.80	与理论性能差距较大
5	`datalink -f --ber 1e-4 A` `datalink -f --ber 1e-4 B`	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}	1311	31.86	32.48	与理论性能差距较大

选择重传协议的测试结果见**表 ??**。

表 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）	接收方线路利用率（%）		存在的问题
				A	B	
1	`datalink -u A` `datalink -u B`	无误码信道数据传输	1268	54.00	96.97	
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，慢发 100 秒”	1201	52.74	95.37	
3	`datalink -u -f A` `datalink -u -f B`	无误码信道，站点 A 和站点 B 的分组层都洪水式产生分组	1234	96.97	96.97	
4	`datalink -f A` `datalink -f B`	站点 A/B 的分组层都洪水式产生分组	1600	94.92	94.52	
5	`datalink -f --ber 1e-4 A` `datalink -f --ber 1e-4 B`	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}	1225	56.67	59.63	与理论性能差距较大

3.5 实验结果分析

回退 N 协议与理论性能差距较大，仅与选择重传协议的性能作对比参考。

观察选择重传协议第五组测试的日志，发现其中经常出现类似**图 ??**的情况。

```
007.727 ---- DATA 5 timeout ----
007.727 Send DATA 5 11, ID 10005
007.759 Recv DATA 22 3, ID 20022
008.000 ---- DATA 6 timeout ----
008.001 Send DATA 6 11, ID 10006
008.020 Recv DATA 23 3, ID 20023
008.265 Send DATA 25 11, ID 10025
008.284 Recv DATA 24 3, ID 20024
008.284 ---- DATA 7 timeout ----
008.284 Send DATA 7 11, ID 10007
008.543 ---- DATA 8 timeout ----
008.543 Send DATA 8 11, ID 10008
008.559 **** Receiver Error, Bad CRC Checksum ****
008.797 ---- DATA 9 timeout ----
008.797 Send DATA 9 11, ID 10009
008.831 Recv DATA 25 3, ID 20025
009.086 **** Receiver Error, Bad CRC Checksum ****
009.120 ---- ACK 12 timeout ----
009.121 Send ACK 11
009.306 ---- DATA 4 timeout ----
009.306 Send DATA 4 11, ID 10004
009.323 ---- DATA 10 timeout ----
009.323 Send DATA 10 11, ID 10010
009.360 Recv DATA 27 3, ID 20027
009.589 ---- DATA 11 timeout ----
009.589 Send DATA 11 11, ID 10011
009.589 Recv DATA 28 3, ID 20028
009.854 ---- DATA 12 timeout ----
009.854 Send DATA 12 11, ID 10012
009.870 Recv DATA 29 3, ID 20029
010.120 ---- DATA 13 timeout ----
010.120 Send DATA 13 11, ID 10013
010.139 Recv DATA 30 3, ID 20030
010.388 ---- DATA 14 timeout ----
010.388 Send DATA 14 11, ID 10014
010.407 Recv DATA 31 3, ID 20031
010.659 Recv DATA 13 3, ID 20013
010.659 ---- DATA 15 timeout ----
010.659 Send DATA 15 11, ID 10015
010.915 ---- DATA 16 timeout ----
010.915 Send DATA 16 11, ID 10016
010.931 Recv DATA 14 13, ID 20014
```

图 3: 常见的导致信道利用率低的情况

在第五组测试中，误码率为 $p = 10^{-4}$，每个帧发送失败的概率为 $1 - (1-p)^{2102} = 0.19$，在一个窗口内平均有 6 个发送失败的帧。

在发生多次错误时，选择重传协议只会对第一个发送失败的帧发送 NAK，当发送方重发该帧时，再发第二个失败的帧的 NAK，此时会有大量收到但未确认的帧积压在接收方的缓冲区内，导致这些帧在发送方超时，因而产生大量不必要的重传。

我认为解决方法有两个：其一是弃用累计确认，对每个帧单独确认；其二是对每个错帧都单独发送 NAK，取消每次只有一个 NAK 帧的限制。

我实现了较容易的第二种方案。具体来说，将 NAK 帧的 `ack` 字段含义改变为：否认某一个编号的帧并要求重发，同时取消 `no_nak` 变量。

改进后的选择重传协议的性能测试结果记录如**表 ??**。可见第五组测试的性能得到大幅度提高。

表 3: 改进后的选择重传协议 性能测试记录表

序号	命令	说明	运行时间（秒）	接收方线路利用率（%）		存在的问题
				A	B	
1	`datalink -u A` `datalink -u B`	无误码信道数据传输	1282	54.43	96.97	
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，慢发 100 秒”	1296	53.91	95.04	
3	`datalink -u -f A` `datalink -u -f B`	无误码信道，站点 A 和站点 B 的分组层都洪水式产生分组	1261	96.97	96.97	
4	`datalink -f A` `datalink -f B`	站点 A/B 的分组层都洪水式产生分组	1603	94.99	95.15	
5	`datalink -f --ber 1e-4 A` `datalink -f --ber 1e-4 B`	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}	3603	79.11	79.40	

3.6 存在的问题

经过改进后的选择重传协议在部分情况下仍然与理论情况存在细微差距，我推测和程序本身的延迟，以及我们在计算时没有考虑 ACK 和 NAK 帧有关，此外计时器参数仍然可以进行细微的调整，以便适应不同误码率的情况。

4 研究与探索的问题

4.1 CRC 校验能力

本次实验采用 32 位 CRC 校验码，理论上可以检测出所有影响到奇数位的错误和所有长度小于等于 32 的突发错误。

因此，只有发生 33 位以上的突发错误，坏帧才有可能通过检测。设一帧中出现突发错误的位数为随机变量 X，则可以认为 X 满足泊松分布即 $X \sim \pi(\lambda)$，其中 $\lambda = np_n = 2104 \times 10^{-5} = 2.104 \times 10^{-2}$，此时：

$$P\{33 \leq X \leq 2104\} = \sum_{k=33}^{2104} \frac{\lambda^k e^{-\lambda}}{k!} \approx 5.163 \times 10^{-93}$$

而这样的坏帧通过检测的概率为 2^{-32}，因此检测不出坏帧的概率为：

$$P = 2^{-32} \cdot P\{33 \leq X \leq 2104\} \approx 1.202 \times 10^{-102}$$

理想情况下客户每天发送帧的个数为：

$$k = \frac{95.31\% \times 8000 \times 60 \times 60 \times 24}{263 \times 8} \times 50\% \approx 156554$$

因此理论上发生一次误码需要 $\frac{1}{365 \cdot k \cdot P} \approx 1.455 \times 10^{94}$ 年。

此外，即使在本层出现没有检测到的错误，理论上上层协议也会有检错和纠错措施，能对本层漏检的错误进行补偿。

4.2 CRC 校验和的计算方法

查表法和手算 CRC 校验和是等效的，手算 CRC 校验和时是根据当前被除数最高位 1 或 0 判断是否异或 $G(x)$，然而查表法先预处理 8 位除以 $G(x)$ 的余数，之后每次除法从从表中取最高字节的余数，和原来余数的低 24 位异或得到新的余数。

值得注意的是这里的查表法默认是最低有效位在前。我实现的构造查表数组算法如下：

```
1  void make_crc32_table(unsigned int crc_table[])
2  {
3      for(int i=0; i<256; i++)
4      {
5          unsigned int value = i;
6          for(int j=0; j<8; j++)
7          {
8              if(value&1)
9                  value = (value >> 1) ^ 0xedb88320;
10             else
11                 value = (value >> 1);
12         }
13         crc_table[i] = value;
14     }
15 }
```

4.3 程序设计方面的问题

跟踪功能主要是便于调试，在程序出错时能快速定位出错位置，或者实时观察某些变量的取值以便针对性地进行性能优化。我的程序中多次调用了跟踪函数，并且将其应用到调试中。

`get_ms()` 函数可以调用 `time.h` 中的 `clock()` 函数实现，具体来说，在进入主函数的`while`循环前调用该函数获取初始时间，之后每次调用 `get_ms()` 函数时获取当前时间并且减去初始时间。

`lprintf` 函数使用可变参数实现，原型为`int lprintf(const char *format,...)`，通过 `format` 字符串获取输出格式，之后用`va_list`获取之后的参数，传入 `__v_lprintf` 函数，根据格式输出。

DATA 帧计时器需要和帧一一对应，当计时器超时时需要重发该计时器对应的帧，此后计时器需要清零。而 ACK 计时器关注的是第一个被成功接收但是没有发出 ACK 的帧，因此调用 `start_ack_timer` 后不会被清零，当发送 ACK 进行累计确认后才清零。

4.4 软件测试方面的问题

- 无误码信道，主要考察协议数据包的设计，如果冗余信息较少则可以取得较高的信道利用率。
- 交替发送、停发，主要考察计时器参数的设计，较好的参数能够保证在各种情况下都能防止窗口满并且避免冗余重传。
- 洪水模式，考察协议的 ACK、NAK 帧的设计和流量控制能力，如果协议的流量控制系统较弱，或者 ACK、NAK 帧的发送时机不恰当，协议容易发生死锁。
- 高误码率模式，考察协议重传效率，如果协议在高误码率时有多次冗余的重传，则信道利用率会降低。

4.5 对等协议实体之间的流量控制

通过有限的发送方和接收方窗口实现流量控制。当发送方窗口满时，将调用 `disable_network_layer()` 来禁止网络层发来数据包。而当接收方窗口满时将会拒绝发送来的帧，引发发送方超时重传。

4.6 与标准协议的对比

本协议设计时只考虑到了延迟固定、包大小固定、误码率固定、带宽固定的情况，而在实际应用中这些量都有可能发生变化，协议需要一个自适应机制来应对不同的情况，例如通过拆分数据包使其变成小帧从而降低重发的频率、在不同误码率和延迟时自动调整计时器参数等。

5 实验总结和心得体会

本次程序的编码和调试大约花费我 5 个小时的时间，期间代码实现大部分参考了教材中的程序和实验材料中的示例。

在实现期间，由于对 C 语言和协议本身理解不够深刻，将`#define inc(k) k = (k + 1) & MAX_SEQ`误写作`#define inc(k) k = (k + 1) % MAX_SEQ`，导致程序运行时会出现窗口过长而出现错误，经过几十分钟的调试才发现帧的编号出现问题。

由于事先配置好环境，对 C 语言语法本身也较为熟悉，并且实现之前认真阅读了课本内容和库文档，在这些部分没有遇到太多问题。

在测试时我发现程序在误码率较高时效率低下，通过查看日志分析原因，进而提出了改进的方法，使得程序在高误码率的情况下得到了可观的效率。

经过本次实验，我对选择重传协议有了更深刻的理解，另外对于协议的不足之处也有了认识，这是只看课本学习不到的知识。此外，通过阅读库代码和上手实践，我对事件驱动的网络编程有了大体的观念，这对于实践课实现 DNS 中继服务器有很大的帮助。在阅读代码的过程中，我也留意了编码规范和格式等一些细节，对我以后的 C 语言编程也会有很大帮助。

A 源程序

源代码 1: datalink.c

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

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

#define MAX_PKT 256
#define MAX_SEQ 63
#define NR_BUFS ((MAX_SEQ + 1) / 2)
#define DATA_MAX_TIME 5000
#define ACK_MAX_TIME 280

#define inc(k) k = (k + 1) & MAX_SEQ

typedef unsigned int seq_nr;
typedef unsigned char frame_kind;

typedef struct
{
    frame_kind kind;              // 帧的种类
    unsigned char ack;            // ACK 序号：最后一个被成功接收的帧序号
    unsigned char seq;            // 帧序号
    unsigned char data[MAX_PKT]; // 数据
    unsigned char padding[4];     // CRC 校验段
} Frame;                          // 帧，与物理层通信；seq,data,padding 只在 DATA 类型帧中有效

typedef struct
{
    unsigned char data[MAX_PKT];
    size_t len;
} Packet; // 包，与网络层通信

static bool phl_ready = false;  // 物理层是否就绪
static Packet out_buf[NR_BUFS]; // 接收窗口
static Packet in_buf[NR_BUFS];  // 发送窗口
static bool arrived[NR_BUFS];   // 标记帧是否到达的 bit map

// 判断在窗口中是否有 a<=b<c
static bool between(seq_nr a, seq_nr b, seq_nr c)
{
    return a <= b && b < c || c < a && a <= b || b < c && c < a;
}

// 添加校验段，发送 DATA 帧到物理层
static void put_frame(unsigned char *frame, int len)
```

```
47  {
48      *(unsigned int *)(frame + len) = crc32(frame, len);
49      send_frame(frame, len + 4);
50      phl_ready = false;
51  }
52
53  // 根据不同类型构建帧、发送帧、处理计时器
54  static void send_dllayer_frame(frame_kind fk, seq_nr frame_nr, seq_nr
    ↪  frame_expected)
55  {
56      Frame s;
57      s.kind = fk;
58      s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
59      if (fk == FRAME_DATA)
60      {
61          s.seq = frame_nr;
62          memcpy(s.data, out_buf[frame_nr % NR_BUFS].data, out_buf[frame_nr %
            ↪  NR_BUFS].len);
63          dbg_frame("Send DATA %d %d, ID %d\n", s.seq, s.ack, *(short *)s.data);
64          put_frame((unsigned char *)&s, 3 + out_buf[frame_nr % NR_BUFS].len);
65          start_timer(frame_nr % NR_BUFS, DATA_MAX_TIME);
66      }
67      else
68      {
69          if (fk == FRAME_NAK)
70          {
71              s.ack = frame_expected;
72              dbg_frame("Send NAK %d\n", frame_expected);
73          }
74          else
75              dbg_frame("Send ACK %d\n", s.ack);
76          send_frame((unsigned char *)&s, 3);
77      }
78      stop_ack_timer();
79  }
80
81  int main(int argc, char **argv)
82  {
83      seq_nr ack_expected = 0;        // 期望收到 ACK 的帧序号，发送窗口的下界
84      seq_nr next_frame_to_send = 0; // 下一个发出的帧序号，发送窗口的上界 +1
85      seq_nr frame_expected = 0;      // 期望收到的帧序号，接收窗口的下界
86      seq_nr too_far = NR_BUFS;       // 接收窗口的上界 +1
87      int i;
88      Frame r;                // 接收到的帧
89      int len;                // 接收到的帧的长度
90      seq_nr nbuffered = 0; // 接收到的帧的数量
91      int event;
92      int arg; // 事件接收到的参数
93
94      protocol_init(argc, argv);
95      lprintf("Designed by xqmmcqs, build: " __DATE__ "  "__TIME__
```

```
96              "\n");
97
98      disable_network_layer();
99
100     while (true)
101     {
102         event = wait_for_event(&arg);
103         // lprintf("NBUFFERED %d ACK_EXPECTED %d NEXT_FRAME_TO_SEND %d\n",
            ↪ nbuffered, ack_expected, next_frame_to_send);
104
105         switch (event)
106         {
107         case NETWORK_LAYER_READY: // 网络层就绪
108             nbuffered++;
109             out_buf[next_frame_to_send % NR_BUFS].len =
                ↪ get_packet(out_buf[next_frame_to_send % NR_BUFS].data);
110             send_dllayer_frame(FRAME_DATA, next_frame_to_send,
                ↪ frame_expected); // 发送 DATA 帧
111             inc(next_frame_to_send);
112             break;
113
114         case PHYSICAL_LAYER_READY:
115             phl_ready = 1;
116             break;
117
118         case FRAME_RECEIVED:
119             len = recv_frame((unsigned char *)&r, sizeof r);
120             if (len < 5 && len != 3)
121                 break;
122             if (len >= 5 && crc32((unsigned char *)&r, len) != 0) // 校验错误
123             {
124                 dbg_event("**** Receiver Error, Bad CRC Checksum ****\n");
125                 send_dllayer_frame(FRAME_NAK, 0, r.seq);
126                 break;
127             }
128
129             if (r.kind == FRAME_ACK)
130                 dbg_frame("Recv ACK %d\n", r.ack);
131
132             if (r.kind == FRAME_DATA)
133             {
134                 dbg_frame("Recv DATA %d %d, ID %d\n", r.seq, r.ack, *(short
                    ↪ *)r.data);
135                 start_ack_timer(ACK_MAX_TIME);
136                 if (between(frame_expected, r.seq, too_far) && !arrived[r.seq
                    ↪ % NR_BUFS])
137                 {
138                     arrived[r.seq % NR_BUFS] = true;
139                     memcpy(in_buf[r.seq % NR_BUFS].data, r.data, len - 7);
140                     in_buf[r.seq % NR_BUFS].len = len - 7;
```

```
141                     while (arrived[frame_expected % NR_BUFS]) // 将接收缓冲区中
                        ↪ 连续的已收到的帧发送到网络层
142                     {
143                         put_packet(in_buf[frame_expected % NR_BUFS].data,
                            ↪  in_buf[frame_expected % NR_BUFS].len);
144                         arrived[frame_expected % NR_BUFS] = false;
145                         inc(frame_expected);
146                         inc(too_far);
147                         start_ack_timer(ACK_MAX_TIME);
148                     }
149                 }
150             }
151 
152             if (r.kind == FRAME_NAK && between(ack_expected, r.ack,
                ↪ next_frame_to_send)) // 发送特定的 NAK 帧
153             {
154                 dbg_frame("Recv NAK %d\n", r.ack);
155                 send_dllayer_frame(FRAME_DATA, r.ack, frame_expected);
156                 break;
157             }
158 
159             while (between(ack_expected, r.ack, next_frame_to_send)) // 累计确
                ↪ 认
160             {
161                 nbuffered--;
162                 stop_timer(ack_expected % NR_BUFS);
163                 inc(ack_expected);
164             }
165             break;
166 
167         case DATA_TIMEOUT:
168             dbg_event("---- DATA %d timeout ----\n", arg);
169             if (!between(ack_expected, arg, next_frame_to_send)) // 计时器中的
                ↪ 序号只有一半，要将序号转化到窗口区间内
170                 arg += NR_BUFS;
171             send_dllayer_frame(FRAME_DATA, arg, frame_expected);
172             break;
173 
174         case ACK_TIMEOUT:
175             dbg_event("---- ACK %d timeout ----\n", frame_expected);
176             send_dllayer_frame(FRAME_ACK, 0, frame_expected);
177             break;
178         }
179 
180         if (nbuffered < NR_BUFS && phl_ready)
181             enable_network_layer();
182         else
183             disable_network_layer();
184     }
185 }
```