**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**

姓名：张梓良 杨晨

学号：2021212484 2021212171

专业：计算机科学与技术

日期：2023年5月3日

1. **实验内容和实验环境描述**
   1. **实验内容**

利用所学数据链路层原理，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。

此处，我们小组选择设计的协议是选择重传协议。

**1.2 实验环境**

* windowsSdkVersion: 10.0.22000.0
* Visual Studio 2022 (v143)
* cStandard: c17

1. **协议设计**
   1. **帧中各个字段的定义和编码**
2. // 帧的数据结构
3. typedef struct
4. {
5. frame\_kind kind; // 帧的类型
6. unsigned char ack; // 确认的帧的序号
7. unsigned char seq; // 帧序号
8. unsigned char data[MAX\_PKT]; // 帧数据
9. unsigned int padding; // CRC校验
10. } FRAME;
    1. **站点信息交换的过程控制**

站点A给站点B传输一帧，如果该帧无差错，则站点B回传给站点A一帧，该帧中携带了对A传输的上一帧的确认。如果B在一定时间内没有要给A发送的数据帧，则B单独回传一个ACK确认帧。

若该帧有差错，则站点B回传给站点A一个NAK帧，站点A收到NAK帧，选择重传出错的帧。

具体过程详见算法流程图。

1. **软件设计**

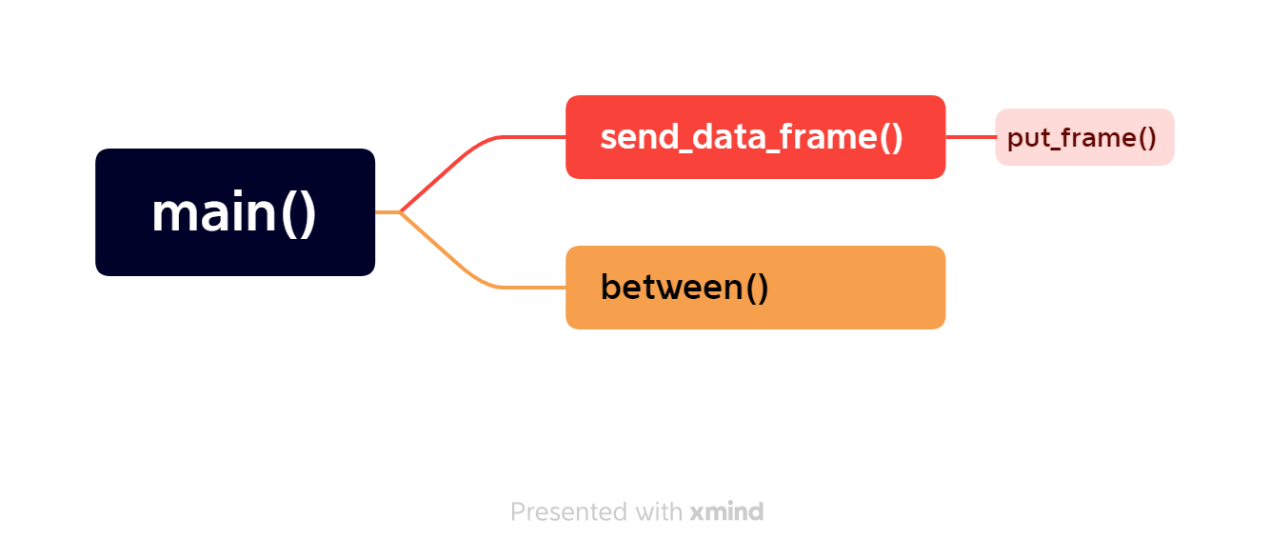
**3.1数据结构**

1. // 自定义结构体和全局变量
2. typedef unsigned char frame\_kind;
3. typedef unsigned int seq\_nr;
4. // 帧的数据结构
5. typedef struct
6. {
7. frame\_kind kind; // 帧的类型
8. unsigned char ack; // 确认的帧的序号
9. unsigned char seq; // 帧序号
10. unsigned char data[MAX\_PKT]; // 帧数据
11. unsigned int padding; // CRC校验
12. } FRAME;
13. // 包的数据结构
14. typedef struct
15. {
16. unsigned char data[MAX\_PKT]; // 包数据
17. } PACKET;
18. static bool no\_nak = true; // 是否已经发送过NAK
19. static int phl\_ready = 0; // 物理层是否就绪
20. // 主函数中的变量
21. seq\_nr ack\_expected; // 发送窗口的下界
22. seq\_nr next\_frame\_to\_send; // 发送窗口的上界+1
23. seq\_nr frame\_expected; // 接收窗口的下界
24. seq\_nr too\_far; // 接收窗口的上界+1
25. int i; // 缓冲区的index
26. FRAME r; // 接收到的帧
27. PACKET out\_buf[NR\_BUFS]; // 发送窗口
28. PACKET in\_buf[NR\_BUFS]; // 接受窗口
29. bool arrived[NR\_BUFS]; // 接收窗口的bit map
30. seq\_nr nbuffered; // 发送窗口已经用了多少
31. int event; // 事件类型
32. int arg; // 超时计时器的编号
33. int len; // 帧长度

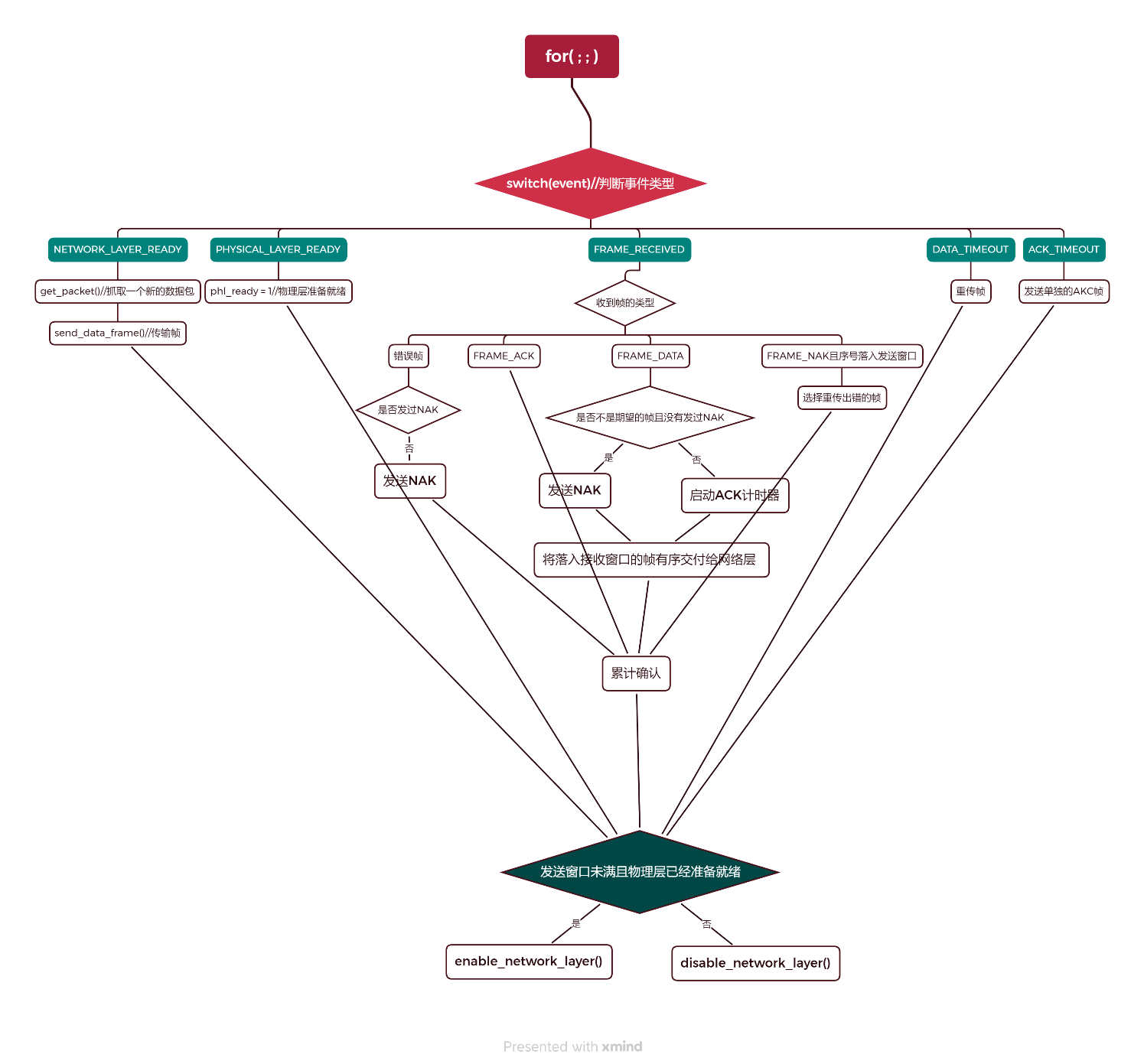
**3.2模块结构**

**3.2.1 子程序**

1. // 判断序号是否满足a<=b<=c(循环)
2. static bool between(seq\_nr a, seq\_nr b, seq\_nr c);
3. // 给帧加入校验字段，并将帧发送给物理层
4. static void put\_frame(unsigned char\* frame, int len);
5. // 构建并发送一个data,ack,或者nak帧
6. static void send\_data\_frame(frame\_kind fk, seq\_nr frame\_nr, seq\_nr frame\_expected, PACKET buffer[]);
7. // 主函数
8. int main(int argc, char\*\* argv);
   * 1. **子程序调用关系图**

****

**3.3 算法流程**

****

**4 实验结果分析**

* 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。程序的健壮性良好，能够可靠地长时间运行。

* 1. **协议参数的选取**
     1. **滑动窗口的大小**

首先根据计算出：

在无差错的情况下，根据选择重传协议的性能分析公式有(由于是piggybacking，所以是 )：

推导出 ，取整得 。

在有误码的情况下，假定误码率较高为 ，根据选择重传协议的性能分析公式有：

推导出 ，取整得 。

又由于 ，所以取 。

通过调试程序发现，有一定的几率可能会用到 个以上的发送窗口，由于这点内存开销是十分小的，因此为了达到最大性能，取 。

* + 1. **重传定时器的时限和ACK搭载定时器的时限**

重传定时器的时限应大于一帧发出并收到它的ACK所需的时间，保证不会因为长时间收不到ACK而重发帧。同时ACK搭载定时器的时限应该小于发送窗口被装载满的时间，保证不会因为发送窗口长时间不更新，而导致协议性能降低。因此可以得到以下不等关系：

同时不能设定的太小，否则会使得接收方一直发送单独的ACK帧，而不采用piggyback机制，导致协议的性能大大降低。

经过调试程序发现，， 时，协议的性能较为良好。

* 1. **理论分析**

无差错信道环境下分组层能获得的最大信道利用率：

下面分析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。

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。由于 ACK 帧和 NAK 帧长度很短，出现机率较小，所以假设这些帧能 正确传输。

假定信道误码率为 ，由于误码率代表每个bit出错的几率，因此对于长度为263字节的一帧，发送成功的概率为：

因此成功传输一帧所需要的次数为：

因此有误码条件下的最大信道利用率为：

当信道误码率为时：

当信道误码率为时：

* 1. **实验结果分析**

性能测试记录表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 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 | 716.748 | 55.41% | 96.96% |  |
| 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” | 1011.343 | 51.02% | 94.58% |  |
| 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 | 1055.153 | 96.96% | 96.96% |  |
| 4 | --flood | 站点A/B的分组层都洪水式产生分组 | 1127.253 | 94.79% | 94.15% |  |
| 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 | 1150.787 | 56.48% | 58.65% | B站点的线路利用率与理论性能差距较大 |

运行时间要求超过10分钟。

* 1. **存在的问题**

在序号5的命令下，协议的线路利用率与理论性能差距较大。观察datalink-B.log发现存在以下情况：

006.387 Send NAK 12

006.447 Recv NAK 11

006.448 Send DATA 11 11, ID 20011

006.678 Recv DATA 19 10, ID 10019

006.876 Send DATA 20 11, ID 20020

006.938 Recv DATA 20 10, ID 10020

007.137 Send DATA 21 11, ID 20021

007.197 Recv DATA 21 10, ID 10021

007.399 Send DATA 22 11, ID 20022

007.430 Recv DATA 22 10, ID 10022

007.661 Send DATA 23 11, ID 20023

007.723 Recv DATA 12 10, ID 10012

007.953 Send DATA 24 14, ID 20024

008.000 \*\*\*\* Receiver Error, Bad CRC Checksum \*\*\*\*

008.001 Send NAK 15

008.031 Recv NAK 13

由于信道的误码率较大，出现坏帧的概率比较大，而同一时间段只能发送一个NAK，因此只有等待出错的一帧重传后，才能对其后出错的帧回传NAK。因此会导致大量的帧滞留在接受窗口，而接收方没有回传对应的ACK，这些帧的计时器会超时导致不必要的帧重发。

修改代码第143行，去掉条件判断if (no\_nak)，允许同一时间段能发送多个NAK帧后，运行1065.825s后，A站点的线路利用率为：71.05%，B站点的线路利用率为：70.62%，线路利用率得到显著提高。(详情见下图)





1. **研究和探索的问题**
   1. **CRC校验能力**

CRC校验和的计算方法本次实验采用32位CRC校验码，理论上可以检测出所有长度小于等于32位的突发错误。因此只有长度为32位以上的突发错误才有可能不被检测到。

对于长度为33的突发错误，不被检测到的概率为：

对于长度为k的突发错误，k > 33，不被检测到的概率为：

帧的长度为263字节，共2104位。假设信道的误码率为 ，且每个bit出错与否都相互独立，则出错的位数服从二项分布。由于误码率足够小，此处可以用的泊松分布来近似代替二项分布以简化计算复杂度。因此每个帧中发生长度为33-2104的突发错误的概率为：

为了简便计算，我们假定长度为33-2104的突发错误通过检测的概率都为，则坏帧通过检测的概率为：

客户使用本次实验描述的信道，如果客户的通信系统每天的使用率 50%，则客户一天发送的帧数为：

因此理论上产生一次分组层误码的时间为：

因此可以认为CRC32能够实现在有误码的信道上无差错传输。

* 1. **CRC校验和的计算方法**

从理论上来讲，通过字节值查表并叠加的 CRC 算法与手工进行二进制“模 2”除法求余数的算法是等效的。这是因为在实际计算时，每一位（0 或 1）都会被考虑到，而且采用查表法可以减少计算量和时间复杂度。

设置查表数组 crc\_table[256] 数组的构造方式是先用一个循环遍历所有可能的字节值，然后对于每个字节值，执行一次手工计算 CRC 校验和的过程，把结果存储在 crc\_table 数组中相应位置的元素中。最终得到一个包含了所有可能字节值的 CRC 校验和表格。

我实现的构造查表数组算法如下：

1. #include <stdio.h>
2. #include <stdint.h>
3. uint32\_t crc\_table[256];
4. void generate\_crc\_table(void)
5. {
6. uint32\_t i, j;
7. uint32\_t crc;
8. for (i = 0; i < 256; ++i)
9. {
10. crc = i;
11. for (j = 0; j < 8; ++j)
12. {
13. if (crc & 1)
14. crc = (crc >> 1) ^ 0xEDB88320UL;
15. else
16. crc >>= 1;
17. }
18. crc\_table[i] = crc;
19. }
20. }
21. int main()
22. {
23. generate\_crc\_table();
24. for (int i = 0; i < 256; ++i)
25. {
26. printf("crc\_table[%d]: %08X\n", i, crc\_table[i]);
27. }
28. return 0;
29. }
    1. **程序设计方面的问题**
       1. **协议软件的跟踪功能**

协议软件的跟踪功能，能够在程序运行期间对协议动作进行跟踪和观察分析以确定协议逻辑是否正确。默认情况下，不产生任何调试信息的输出；通过某种命令可以打开调试开关，边运行边产生协议动作信息的输出。在我的程序调试中，大大依靠了跟踪功能来检测协议的运行。

* + 1. **get\_ms()函数**

对于程序库中获取时间坐标的函数get\_ms()，还能在Unix系统下使用C标准库中的time.h头文件和sys/time.h头文件通过以下函数实现：

1. #include <stdio.h>
2. #include <sys/time.h>
3. unsigned long get\_ms()
4. {
5. struct timeval tv;
6. gettimeofday(&tv, NULL);
7. return (unsigned long)((tv.tv\_sec \* 1000ul) + (tv.tv\_usec / 1000ul));
8. }
   * 1. **定时器函数**

start\_timer()函数的启动时刻不是当前时刻，而是将当前物理层发送队列的数据发送完毕后开始启动计时，这是为了保证从帧真正发出去的那一刻才开始计时。重复设置同一个编号的计时器会导致重新按新调用计时，这是为了保证不会产生不必要的重传。

start\_ack\_timer()函数启动时刻为当前时刻，这是为了保证能够在接受到帧后立刻启动计时。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事件ACK\_TIMEOUT，这是为了保证对每一帧都能及时发回一个ACK，避免产生大量不必要的重传。

* 1. **对等体协议实体之间的流量控制**

我认为我所设计的滑动窗口协议软件已经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。通过限制发送方和接受方的窗口大小以实现流量控制。

* 1. **与标准协议的对比**

如果要通过卫星信道进行通信，需要解决以下问题：

* 高延迟：卫星信道的传输延迟一般在数百毫秒到数秒之间，这会对通信造成较大的影响，需要设计合适的协议来降低延迟。
* 信号衰减和噪声干扰：卫星信号在传输过程中会受到衰减和噪声干扰，这会导致数据错误或丢失，需要设计合适的纠错机制和重传机制来保证数据的可靠性。
* 安全性：卫星信道的通信容易受到窃听和篡改攻击，需要设计合适的加密和身份认证机制来保护数据的安全性。

实验协议与实用协议之间存在一定的差距。实验协议只是在模拟场景下测试协议的正确性和性能，而现实场景中会涉及到更多的复杂因素，如网络拥塞、动态路由、故障恢复等问题。实用协议需要经过长期的实践检验和优化，才能具备稳定性和可靠性。

我觉得还需要增加以下功能：

* 拥塞控制机制：卫星信道的带宽较窄，容易发生网络拥塞。需要设计合适的拥塞控制机制来避免网络崩溃。
* 动态路由机制：卫星信道的连接不稳定，需要设计合适的动态路由机制来保证数据能够传输到目标站点。

对于LAPB协议，它是CCITT（现在称为ITU-T）推荐的链路层协议之一，主要用于X.25协议中。LAPB协议具有以下特点：

* 可靠性高：LAPB协议支持可靠的传输和重传机制，能够保证数据的正确性和完整性。
* 连接管理：LAPB协议可以管理多个连接，支持多种连接模式，如无连接、面向连接等。
* 流量控制：LAPB协议支持流量控制机制，可以根据接收方的处理能力调整发送速率，避免因过快发送导致的网络拥塞。

与我所设计的实验性协议相比，LAPB协议具备更为丰富的功能和更高的可靠性。

1. **实验总结和心得体会**
   1. **实验总时长**

本次实验在实验指导书的解读上花费了1.5h，在协议程序的编写和调试上花费了3h，在协议性能的分析和提升上花费了2.5h，在实验报告的撰写和回顾分析整个实验上花费5h。总计花费12h。

* 1. **实验中所遇到的问题**
     1. **编程工具**

在安装配置visual studio上花费了一定的时间，最终在网络上的教程指导下成功安装配了visual studio 2022 community。

* + 1. **编程语言**

对C语言的多文件编程和编译不太熟悉，在多文件的链接编译上遇到了一定的问题。最初使用visual studio code无法实现多文件链接编译，最终采取使用visual studio 2022 community的办法成功解决该问题。

* + 1. **协议**

对选择重传协议的一些细节不太熟悉。在编写协议程序的过程中，先阅读并理解了一遍教材中所给的伪代码，了解了SR协议的大致思路和原理。

* 1. **总结**

通过该实验，让我更好的理解并掌握了选择重传协议。同时通过自身动手实践完成一个协议，并分析协议的理论性能和实际性能之间的差异，调试协议程序以提高协议的性能的过程让我更加深刻的认识到了网络协议的精细以及不断追求更高性能的特点。在这个过程中，我的C语言编程能力也得到了一定的提升，代码书写的规范程度也得到了加强。

1. **源程序文件 datalink.c**
2. #include <stdio.h>
3. #include <string.h>
4. #include <stdbool.h>
5. #include "protocol.h"
6. #include "datalink.h"
7. #define MAX\_PKT 256 // 包的大小
8. #define MAX\_SEQ 63 // 最大序号
9. #define NR\_BUFS ((MAX\_SEQ + 1) / 2) // 缓存空间大小
10. #define DATA\_TIMER 5000 // 帧超时时间
11. #define ACK\_TIMER 280 // ack超时时间
12. #define inc(k) \
13. if (k < MAX\_SEQ) \
14. k = k + 1; \
15. else \
16. k = 0
17. typedef unsigned char frame\_kind;
18. typedef unsigned int seq\_nr;
19. // 帧的数据结构
20. typedef struct
21. {
22. frame\_kind kind; // 帧的类型
23. unsigned char ack; // 确认的帧的序号
24. unsigned char seq; // 帧序号
25. unsigned char data[MAX\_PKT]; // 帧数据
26. unsigned int padding; // CRC校验
27. } FRAME;
28. // 包的数据结构
29. typedef struct
30. {
31. unsigned char data[MAX\_PKT]; // 包数据
32. } PACKET;
33. static bool no\_nak = true; // 是否已经发送过NAK
34. static int phl\_ready = 0; // 物理层是否就绪
35. // 判断序号是否满足a<=b<=c(循环)
36. static bool between(seq\_nr a, seq\_nr b, seq\_nr c)
37. {
38. return ((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a));
39. }
40. // 给帧加入校验字段，并将帧发送给物理层
41. static void put\_frame(unsigned char\* frame, int len)
42. {
43. \*(unsigned int\*)(frame + len) = crc32(frame, len);
44. send\_frame(frame, len + 4);
45. phl\_ready = 0;
46. }
47. // 构建并发送一个data,ack,或者nak帧
48. static void send\_data\_frame(frame\_kind fk, seq\_nr frame\_nr, seq\_nr frame\_expected, PACKET buffer[])
49. {
50. FRAME s;
51. s.kind = fk;
52. s.seq = frame\_nr;
53. s.ack = (frame\_expected + MAX\_SEQ) % (MAX\_SEQ + 1);
54. if (fk == FRAME\_DATA)
55. {
56. memcpy(s.data, buffer[frame\_nr % NR\_BUFS].data, PKT\_LEN);
57. dbg\_frame("Send DATA %d %d, ID %d\n", s.seq, s.ack, \*(short\*)s.data);
58. put\_frame((unsigned char\*)&s, 3 + PKT\_LEN);
59. start\_timer(frame\_nr % NR\_BUFS, DATA\_TIMER);
60. }
61. else
62. {
63. if (fk == FRAME\_NAK)
64. {
65. no\_nak = false; // 已经发送过NAK了
66. dbg\_frame("Send NAK %d\n", frame\_expected);
67. }
68. else // ack
69. {
70. dbg\_frame("Send ACK %d\n", s.ack);
71. }
72. put\_frame((unsigned char\*)&s, 2);
73. }
74. // 有帧发送无需单独的ack帧
75. stop\_ack\_timer();
76. }
77. int main(int argc, char\*\* argv)
78. {
79. seq\_nr ack\_expected; // 发送窗口的下界
80. seq\_nr next\_frame\_to\_send; // 发送窗口的上界+1
81. seq\_nr frame\_expected; // 接收窗口的下界
82. seq\_nr too\_far; // 接收窗口的上界+1
83. int i; // 缓冲区的index
84. FRAME r; // 接收到的帧
85. PACKET out\_buf[NR\_BUFS]; // 发送窗口
86. PACKET in\_buf[NR\_BUFS]; // 接受窗口
87. bool arrived[NR\_BUFS]; // 接收窗口的bit map
88. seq\_nr nbuffered; // 发送窗口已经用了多少
89. int event; // 事件类型
90. int arg; // 超时计时器的编号
91. int len; // 帧长度
92. len = 0;
93. ack\_expected = 0; // 期望收到的ACK
94. next\_frame\_to\_send = 0; // 下一个发送的帧的序号
95. frame\_expected = 0;
96. too\_far = NR\_BUFS;
97. nbuffered = 0; // 初始没有包被缓存
98. for (i = 0; i < NR\_BUFS; i++)
99. arrived[i] = false;
100. protocol\_init(argc, argv);
101. lprintf("Designed by ziliangzhang, build: " \_\_DATE\_\_ " "\_\_TIME\_\_
102. "\n");
103. disable\_network\_layer();
104. for (;;)
105. {
106. event = wait\_for\_event(&arg);
107. switch (event)
108. {
109. case NETWORK\_LAYER\_READY:
110. nbuffered = nbuffered + 1;
111. get\_packet(out\_buf[next\_frame\_to\_send % NR\_BUFS].data); // 抓取一个新的数据包
112. send\_data\_frame(FRAME\_DATA, next\_frame\_to\_send, frame\_expected, out\_buf); // 传输帧
113. inc(next\_frame\_to\_send);
114. break;
115. case PHYSICAL\_LAYER\_READY:
116. phl\_ready = 1;
117. break;
118. case FRAME\_RECEIVED:
119. len = recv\_frame((unsigned char\*)&r, sizeof r);
120. // 收到错误帧
121. if (len < 5 || crc32((unsigned char\*)&r, len) != 0)
122. {
123. dbg\_event("\*\*\*\* Receiver Error, Bad CRC Checksum \*\*\*\*\n");
124. // 收到错误帧，且没有发过NAK，则发送NAK
125. if (no\_nak)
126. send\_data\_frame(FRAME\_NAK, 0, frame\_expected, out\_buf);
127. break;
128. }
129. // 收到单独的确认帧ACK
130. if (r.kind == FRAME\_ACK)
131. dbg\_frame("Recv ACK %d\n", r.ack);
132. // 收到数据帧
133. if (r.kind == FRAME\_DATA)
134. {
135. dbg\_frame("Recv DATA %d %d, ID %d\n", r.seq, r.ack, \*(short\*)r.data);
136. // 收到的帧不是期望收到的帧且没有发过NAK
137. if (r.seq != frame\_expected && no\_nak)
138. send\_data\_frame(FRAME\_NAK, 0, frame\_expected, out\_buf);
139. else
140. start\_ack\_timer(ACK\_TIMER);
141. if (between(frame\_expected, r.seq, too\_far) && arrived[r.seq % NR\_BUFS] == false)
142. {
143. arrived[r.seq % NR\_BUFS] = true;
144. // len=PKT\_LEN+3, len=len+4, len在PKT\_LEN的基础上+7, 所以包长=len-7
145. memcpy(in\_buf[r.seq % NR\_BUFS].data, r.data, len - 7);
146. while (arrived[frame\_expected % NR\_BUFS])
147. {
148. put\_packet(in\_buf[frame\_expected % NR\_BUFS].data, len - 7);
149. no\_nak = true;
150. arrived[frame\_expected % NR\_BUFS] = false;
151. inc(frame\_expected);
152. inc(too\_far);
153. start\_ack\_timer(ACK\_TIMER);
154. }
155. }
156. }
157. // 收到NAK
158. if ((r.kind == FRAME\_NAK) && between(ack\_expected, (r.ack + 1) % (MAX\_SEQ + 1), next\_frame\_to\_send))
159. {
160. dbg\_frame("Recv NAK %d\n", (r.ack + 1) % (MAX\_SEQ + 1));
161. send\_data\_frame(FRAME\_DATA, (r.ack + 1) % (MAX\_SEQ + 1), frame\_expected, out\_buf);
162. }
163. // 累计确认
164. while (between(ack\_expected, r.ack, next\_frame\_to\_send))
165. {
166. nbuffered = nbuffered - 1;
167. stop\_timer(ack\_expected % NR\_BUFS);
168. inc(ack\_expected);
169. }
170. break;
171. case DATA\_TIMEOUT:
172. dbg\_event("---- DATA %d timeout ----\n", arg);
173. if (!between(ack\_expected, arg, next\_frame\_to\_send))
174. arg += NR\_BUFS;
175. send\_data\_frame(FRAME\_DATA, arg, frame\_expected, out\_buf);
176. break;
177. case ACK\_TIMEOUT:
178. dbg\_event("---- ACK %d timeout ----\n", (frame\_expected + MAX\_SEQ) % (MAX\_SEQ + 1));
179. send\_data\_frame(FRAME\_ACK, 0, frame\_expected, out\_buf);
180. break;
181. }
182. if (nbuffered < NR\_BUFS && phl\_ready)
183. enable\_network\_layer();
184. else
185. disable\_network\_layer();
186. }
187. }