实验一：数据链路层滑动窗口协议的设计与实现

一.实验内容与实验环境

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,本小组决定实现的滑动窗口协议为选择重传,并且未使用 NAK 通知机制.

我们之后也实现了 GBN 协议作为对比,由于 SR 协议的整体效果优于 GBN,报告主体介绍 SR 协议,GBN 协议的实现仅附代码,不再说明.

小组成员采用不同的实验环境-Linu x86_64 和 Windows10,代码在两种环境下可以无差异地运行.

二.软件设计(SR)

(1)数据结构

```
#define MAX_SEQ 31                          //  帧的最大序号
#define NR_BUFS ((MAX_SEQ + 1) / 2)         //  发送方和接收方的窗口大小
#define DATA_TIMER 1000                         //  数据帧的超时时间

//  帧的结构
struct FRAME {
    unsigned char kind;                 //  帧类型,指示该帧为数据帧或 ACK 帧
    unsigned char ack_seq;              //  若帧类型为数据帧,则含义为数据包的序
        //  号(seq);若帧类型为 ACK 帧,则含义为确认的数据包的序号(ack)
    unsigned char data[PKT_LEN];        //  数据帧的数据字段
    unsigned int padding;               //  填充字段
};

typedef unsigned char BUFFER[PKT_LEN];

//  缓存的结构
struct BUFFER {
    unsigned char buf[PKT_LEN];
    int len;
};
struct BUFFER in_buf[NR_BUFS], out_buf[NR_BUFS];     //  接收缓存与发送缓存
//  乱序到达的数据包需要缓存,直到缺失的数据包达到时才能与之一起传递给上层
```

```
// 已发送的数据包可能丢失(超时),此时需要读取发送缓存以重传此数据包
static int phl_ready = 0;    // 物理层是否就绪
static int recv_base = 0;    // 接收基准序号
static int send_base = 0, next_seq_nr = 0;   // 发送基准序号,和下一个可用序号
int acked[MAX_SEQ + 1]; // 已发送并被确认的序号
int cached[MAX_SEQ + 1];      // 已接收并被缓存的序号

// 事件处理函数表,仅用于结构化代码
void (*event_handler[])(int *) = {
        [NETWORK_LAYER_READY] = network_layer_ready_handler,
        [PHYSICAL_LAYER_READY] = physical_layer_ready_handler,
        [FRAME_RECEIVED] = frame_received_handler,
        [DATA_TIMEOUT] = data_timeout_handler,
};
```

(2)模块结构

static void put_frame(unsigned char *frame, int len)

frame-需要发送的帧的指针　　len-帧的长度

为已设置好前半部分字段的帧在尾部添加 4 字节的校验字段并发送到物理层.

static int between(int a, int b, int c)

a-窗口左端序号　　b-待判断序号　c-窗口右端序号

判断在滑动窗口意义上 b 是否在[a,c]范围内.若是,则返回 1,否则返回 0.

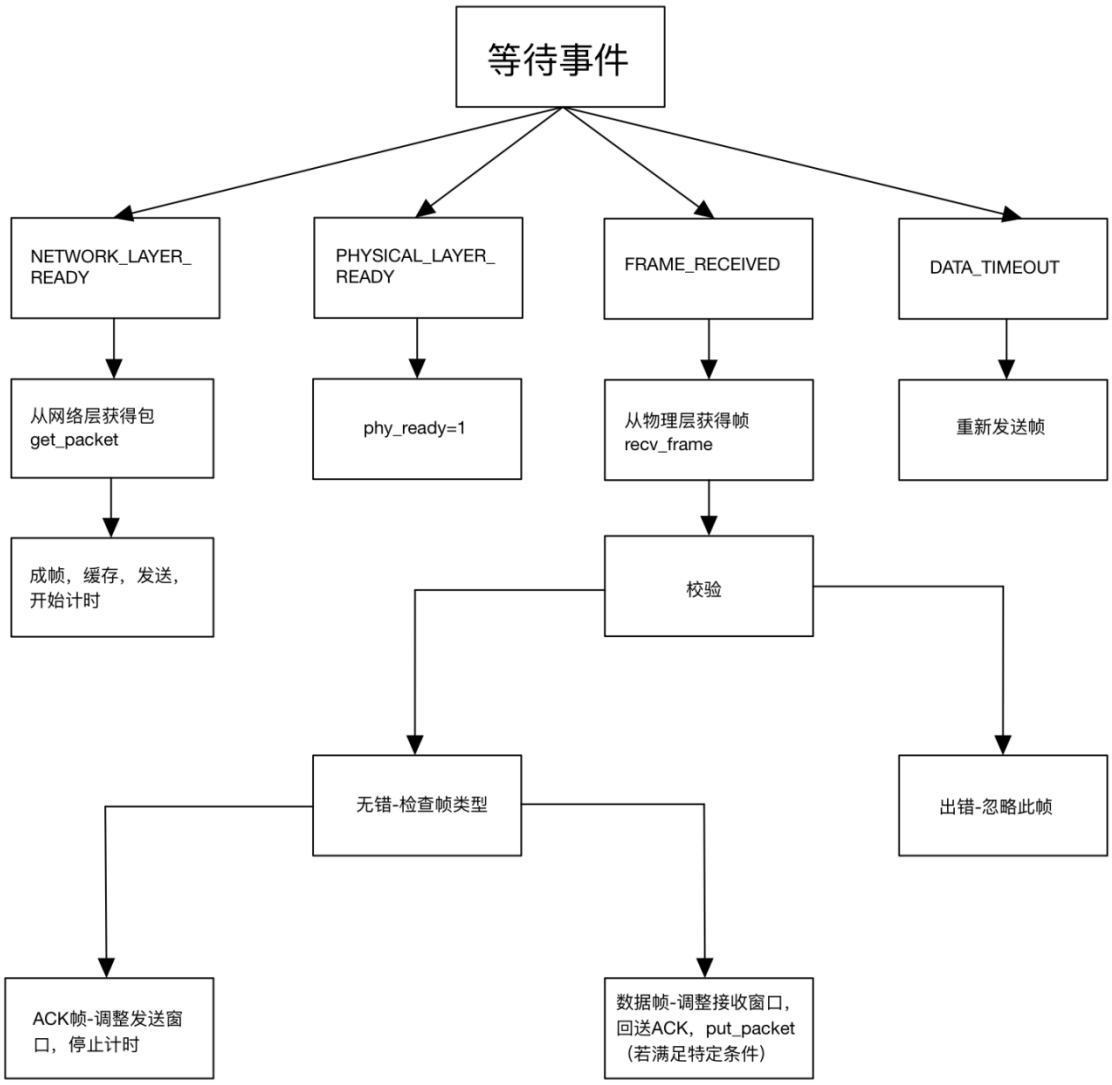
..._handler(int *arg)

arg-由 wait_for_event(&arg)设置的事件参数

事件处理函数,详见(3)部分.

(3) 算法流程

我们实现了一种较为简单的搭载 ACK 的选择重传协议.不使用 NAK,接收方接收到损坏帧只是忽略它,只有超时才会引起发送方重传.



三. 实验结果分析

(1)正确性与健壮性

在文档中给出的所有性能测试命令下,程序均可正常运行超过 20 分钟,且性能接近参考数据,这证明了程序的正确性与健壮性,实现了有误码信道环境中无差错传输功能.

(2)协议参数的选取

滑动窗口大小:信道延迟 t0=270ms,发送一个数据帧用时 t1=(1+1+256+4)B/8000bps=262ms.

则信道利用率为 NR_BUFS*t1/(t1+2*t0),欲使利用率达到理论值,可得 NR_BUFS 不小于 4.考虑到误码影响,决定 NR_BUFS 的值为 16.据此可得 MAX_SEQ 为 31.

超时时间:从发送一个数据包到接收 ACK 的时间为 t1+2*t0=802ms.考虑到一定的宽容时间,决定 DATA_TIMER 的值为 1000.

(3)理论分析

在下面的计算中,考虑的情况为站点 A/B 的分组层都洪水式产生分组.并且假定 ACK 帧一定正确传输.

无差错信道环境:

每当一个数据帧进入信道,必将对应地产生一个 ACK 帧作为答复.这一对帧总大小为(1+1+256+4)+(1+1+4)=268 字节,其中包含的有效数据段为 256 字节,故有理论信道利用率:

256/268*100%=95.52%

有误码信道环境:

若误码率为 p,则易得平均每发送 1/p 个字节会产生一次错误,即平均每发送 N=1/(268*8*p)个帧会出现一个错误帧,可得理论信道利用率:

N/(1+N)*95.52%

对于 p=1e-5,这个值为 93.52%.

对于 p=1e-4,这个值为 78.64%.

(4)实验结果分析

序号	说明	运行时间(秒)	GoBackN 算法线路利用率(%)		Selective 算法线路利用率(%)		参考数据
			A	B	A	B	
1	无误码信道数据传输	1210.578	56.53	96.97	52.23	95.97	53.9 97.0
2	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”	1222.931	41.47	76.35	50.37	94.25	52.9 95.1
3	无误码信道,站点 A 和站点 B 的分组层都洪水式产生分组	1845.423	96.96	96.96	94.82	94.83	97.0 97.0
4	站点 A/B 的分组层都洪水式产生分组	1351.592	88.1	87.7	93.16	93.15	95.0 95.1
5	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}	1292.258	23.1	46.8	78.63	78.01	42.0 73.6

对于 SR 算法:

在--flood 模式下,实际利用率与理论值相差已不到 1%,认为这是可以接受的误差.

我们的理论值与实际数据在低误码率情况下总是小于参考数据,猜测这是因为参考程序所采取的实现方式依赖较少的 ACK/NAK 报文.

而在高误码率情况下,理论值与实际数据均优于参考数据.我们认为这是由一报文一 ACK 的实现带来的健壮性造成的,这一实现的缺陷就是低误码率情况下的低效性.

与 GBN 算法相比:

显然,在无误码信道中,GBN 算法产生的 NAK 或 ACK 数量是极少的,因此相比于 SR 算法有明显优势.而一旦信道存在误码,那么不论是平缓式还是洪水式情况下效率均明显低于 SR 算法,这是因为重传的代价太大了.

这是符合理论认知的,SR 算法本身便是 GBN 算法在此方面进行优化产生的算法.

(5)存在的问题与改进方案

程序虽然经过了所有测试方案,但在低误码率的情况下理论值与实际数据均有改进空间.

我们提出了以下几点可能的改进方案:

1. 压缩字段

原先的帧格式中,类型字段(kind)和 ACK 码/序号字段(ack_seq)各自占用了 1 字节.

考虑后发现,既然帧类型仅有两种,序号范围是 0~31,那么就可以将 KIND 字段压缩为 1bit,将 ACK/SEQ 字段压缩为 5bit,这样两者就可以置入帧的第一个字节之内.

这使帧的长度减小了 1 字节.

2.改变 ACK 帧的校验方式

根据 1 进行改进后的 ACK 帧除 CRC 字段仅有 1 字节,使用占用四字节的 CRC 校验显得有些夸张,而如果采用重复码将这一字节重复为三字节并发送,粗略估计出现无法检出的错误的概率低于 1e-15(误码率 1e-5).

这使 ACK 帧的长度减小了 2 字节.

3.数据帧捎带 ACK

根据 1 修改后的数据帧的第一个字节中,前 7 位已被占用,可以利用剩余的一位表示该数据帧是否捎带 ACK 并且添加一个字节表示 ACK 码.

但具体实现时还需要额外考量:何时用数据帧捎带 ACK,何时单独 ACK 帧?一种想法是对 ACK 设置计时器,每当需要发送 ACK 时开始计时,若超时前出现了需要发送的数据帧,则捎带之;若超时,则单独发送之.

这样一来,还需要考量和测试出最佳的超时时间.

四. 研究与探索的问题

(1) CRC 校验能力

CRC32 可以检测出所有不超过 32 位的突发错误,对于 268 字节的数据帧而言,出现 33 位突发错误的概率为$P_{33} = C_{268}^{33}(10^{-5})^{33}(1-10^{-5})^{235}$<10^-100,用此概率估算出超过 32 位突发错误的概率 $P=236*P_{33}$.

该客户一天之中传输的帧数为

(8000*0.5*24*60*60*93.52%)/(256*8)=157815 帧,

则发送一次误码平均需要的时间为

(1/(P*157815))/365=1.7360*10^92 年.

根据估算时放缩的方向,这是一个保守的值.

(2) 软件测试方面的问题

不同的测试方案可以检测软件在不同状况下的稳定性与性能,以模拟真实环境.

无误码信道数据传输可以用于验证程序基础功能的正确性,这样可以先摆脱校验模块进行测试,提高开发效率.

平缓方式和洪水式产生分组则是模拟真实网络环境中的空闲期与高峰期.

高误码率情况则可检测程序健壮性,就网络而言,一般来说人们愿意牺牲一定的性能来换来更高的稳定性.

此外,认为测试还应该覆盖信道完全空闲,无数据传输时的情况,以检测程序是否表现正常.

(3) 对等协议实体之间的流量控制

发送方只有在确认接收方已收到时才会移动窗口,这便限制了其发送数据的速度,使得接收方不会被流量淹没.也就是说接收方通过反馈 ACK 限制了发送方流量.

五. 实验总结与心得体会

我们实现了 SR 协议和 GBN 协议,前者耗费约 6 小时,后者耗费约 5 小时.

小组成员中部分是在 Linux 环境下编程,部分是在 Windows 下.所幸程序所调用的库函数均为跨平台可移植的,不同环境下的同一份代码除了使用不同的编译命令之外并无区别.

由于(想象中)较为棘手的模块都已经封装为函数可以直接使用,剩下的都是一些简单的逻辑控制代码,不需要用到复杂的 C 语法,在语言方面没有遇到什么问题.

协议方面,在实现 SR 协议时过于自信地开始编码,没有仔细将各个变量的意义研究清楚,以至于后期发现了大量细节性错误(例如把序号空间和窗口大小混淆,导致仅使用了一半的序号空间),程序运行不到几分钟就因传递了错误的数据被中止.协议参数起初使用了参考程序中的值(DATA_TIMER=2000),后来经过理论分析发现了调整方向(DATA_TIMER 减小为 1000),实践后确实得到了显著的性能提升.

除了编码与调试之外,本次实验中耗时最多的事情就是阅读与理解文档内容了.长达 22 页的高密度信息初读起来令人十分头疼,完成实验后回头再看这份文档,认为大部分内容有必要存在,如果能够把函数与功能说明列入一个表格中,可读性也许会更好.

心得体会:

这次实验与我们至今经历过的其他编程实验最大的区别在于:这次实验的核心是算法.

虽然简易滑动窗口算法相比图论和字符串匹配等领域的算法并不算复杂,初次实现起来也并不轻松,充满细节.最大的感受是理解算法与实现算法之间还是有一段距离的,

今后在类似的工作中应当注重做好写代码前的准备工作,否则很可能会陷入不断调试的时间陷阱.

GBN 协议是在 SR 协议之后完成的,我们没有选择继续在 Linux 环境下编程,而是选择在之前没有尝试过的 Windows 环境下编写,本想体验一下两平台的不同之处,但是在该实验中二者的区别并不明显.在完成 SR 协议之后,GBN 协议的编写便顺畅了许多,代码的框架也与 SR 协议基本相同.GBN 的核心算法虽然不算复杂,但是要注意的细节也挺多,特别是 ACK 与 NAK 的发送与接收.有了编写 SR 协议时的教训,我们在编写 GNB 协议时更加注重细节,所以程序没有出现什么大问题,编写完第一次运行便成功了.然而我们的 GBN 协议的性能与参考的 GBN 性能有较大的差距,特别是在有误码并洪水式产生分组的情形之下,A 站点性能与参考值相近,但是 B 站点性能相差百分之二十几.经过理论推算,我们得出 MAX_SEQ 的值为 7,修改后协议性能得到些许提升,差距缩小到百分之十五左右.我们后续修改了其他参数但效果都不太明显,所以程序性能与参考值仍然有差距.虽然性能上略有遗憾,但是我们通过此次实验对 GBN 协议的原理有了更深入的理解,也看到了 GBN 协议的缺点,我们从数据上直观的感受到了 SR 协议与 GBN 协议的性能差距,SR 协议明显性能更好.

这是一次少有的通过调整参数提高性能的编程体验,是网络编程的一个特色.

六. 源程序文件

(1)SR 协议

```
// SR.c
#include "SR.h"
#include <stdio.h>
#include <string.h>
#include "protocol.h"

#define MAX_SEQ 31
#define NR_BUFS ((MAX_SEQ + 1) / 2)

#define DATA_TIMER 1000

struct FRAME {
    unsigned char kind; /* FRAME_DATA */
    unsigned char ack_seq;
    unsigned char data[PKT_LEN];
    unsigned int padding;
};

typedef unsigned char BUFFER[PKT_LEN];
```

```
struct BUFFER {
    unsigned char buf[PKT_LEN];
    int len;
};
struct BUFFER in_buf[NR_BUFS], out_buf[NR_BUFS];
static int phl_ready = 0;
static int recv_base = 0;
static int send_base = 0, next_seq_nr = 0;
int acked[MAX_SEQ + 1];
int cached[MAX_SEQ + 1];

static void put_frame(unsigned char *frame, int len) {
    *(unsigned int *)(frame + len) = crc32(frame, len);
    send_frame(frame, len + 4);
    phl_ready = 0;
}

// is b in [a, c]?
static int between(int a, int b, int c) {
    a %= MAX_SEQ + 1;
    b %= MAX_SEQ + 1;
    c %= MAX_SEQ + 1;
    int ret = (((a <= b) && (b <= c)) || ((c <= a) && (a <=
b)) ||
               ((b <= c) && (c <= a)));
    return ret;
}

// note: only called when between(send_base, next_seq_nr,
send_base + NR_BUFS -
// 1) && phl_ready
void network_layer_ready_handler(int *arg) {
    int len = get_packet(out_buf[next_seq_nr % NR_BUFS].buf);
    out_buf[next_seq_nr % NR_BUFS].len = len;

    // send out immediately and cache it(for possible re-send)
    struct FRAME s;
    s.kind = FRAME_DATA;
    s.ack_seq = next_seq_nr;  // seq
    memcpy(s.data, out_buf[next_seq_nr % NR_BUFS].buf,
PKT_LEN);
```

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

    put_frame((unsigned char *)&s, 2 + len);

    start_timer(s.ack_seq, DATA_TIMER);

    next_seq_nr++;
    next_seq_nr %= MAX_SEQ + 1;
}

void physical_layer_ready_handler(int *arg) { phl_ready = 1; }

void frame_received_handler(int *arg) {
    struct FRAME f;

    int len = recv_frame((unsigned char *)&f, sizeof(f));
    if (len > 6 && crc32((unsigned char *)&f, len) != 0) {
        dbg_event("**** Receiver Error, Bad CRC Checksum\n");
        // ignore the frame
        return;
    }

    // FRAME_ACK:
    if (f.kind == FRAME_ACK &&
        between(send_base, f.ack_seq, send_base + NR_BUFS - 1))
{
        acked[f.ack_seq] = 1;  // ack

        dbg_frame("Recv ACK  %d\n", f.ack_seq);
        stop_timer(f.ack_seq);

        while (acked[send_base]) {
            acked[send_base] = 0;
            send_base++;
            send_base %= MAX_SEQ + 1;
         if(send_base == next_seq_nr) break;
        }
    }

    // FRAME_DATA:
```

```
    if (f.kind == FRAME_DATA) {
        dbg_frame("Recv DATA %d, ID %d\n", f.ack_seq, *(short
*)f.data);

        // send ack
        struct FRAME s;
        s.kind = FRAME_ACK;
        s.ack_seq = f.ack_seq;  // ack, seq

        dbg_frame("Send ACK  %d\n", s.ack_seq);  // ack
        put_frame((unsigned char *)&s, 2);

        if (between(recv_base, f.ack_seq, recv_base + NR_BUFS
- 1)) {
            // cache it, later send to network layer
            if (!cached[f.ack_seq]) {
                memcpy(in_buf[f.ack_seq % NR_BUFS].buf,
f.data, len - 6);
                in_buf[f.ack_seq % NR_BUFS].len = len - 6;
                cached[f.ack_seq] = 1;
            }
            while (cached[recv_base]) {
                cached[recv_base] = 0;
                put_packet(in_buf[recv_base % NR_BUFS].buf,
                           in_buf[recv_base % NR_BUFS].len);
                recv_base++;
                recv_base %= MAX_SEQ + 1;
            }
        }
    }
}

void data_timeout_handler(int *arg) {
    int num = *arg;

    struct FRAME s;
    s.kind = FRAME_DATA;
    s.ack_seq = num;
    memcpy(s.data, out_buf[num % NR_BUFS].buf, PKT_LEN);

    put_frame((unsigned char *)&s, 2 + out_buf[num %
NR_BUFS].len);
```

```
    start_timer(num, DATA_TIMER);

    dbg_frame("Timeout, ReSend DATA %d, ID %d\n", s.ack_seq,
*(short *)s.data);
}

void (*event_handler[])(int *) = {
    [NETWORK_LAYER_READY] = network_layer_ready_handler,
    [PHYSICAL_LAYER_READY] = physical_layer_ready_handler,
    [FRAME_RECEIVED] = frame_received_handler,
    [DATA_TIMEOUT] = data_timeout_handler,
};

int main(int argc, char **argv) {
    int event, arg;

    protocol_init(argc, argv);
    lprintf("Designed by Jiang Yanjun, build: " __DATE__
            "  "__TIME__
            "\n");

    disable_network_layer();

    for (;;) {
        event = wait_for_event(&arg);

        event_handler[event](&arg);

        if (between(send_base, next_seq_nr, send_base +
NR_BUFS - 1) &&
            phl_ready)
            enable_network_layer();
        else
            disable_network_layer();
    }

    return 0;
}
```

```
// SR.h
/* FRAME kind */
#define FRAME_DATA 1
#define FRAME_ACK 2

/*

struct FRAME {
    unsigned char kind;
    unsigned char ack_seq;  // SEQ(1) or ACK(1)
    unsigned char data[PKT_LEN];
    unsigned int padding;
};

    DATA Frame
    +=========+========+=============+========+
    | KIND(1) | SEQ(1) | DATA(?~256) | CRC(4) |
    +=========+========+=============+========+

    ACK Frame
    +=========+========+========+
    | KIND(1) | ACK(1) | CRC(4) |
    +=========+========+========+
*/
```

(2)GBN 协议

```
// GBN.c
#include "GBN.h"

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

#include "protocol.h"

#define DATA_TIMER 2000
#define ACK_TIMER 300
#define MAX_SEQ 7
#define inc(num) num = ((num + 1) & MAX_SEQ)

typedef unsigned char seq_nr;
```

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

static seq_nr frame_to_send = 0;
static seq_nr ack_expected = 0;
static seq_nr frame_expected = 0;

static bool no_nak = true;
static bool phl_ready = false;

static int frame_length = 0;

static unsigned char buffer[MAX_SEQ + 1][PKT_LEN];
static int packet_length[MAX_SEQ + 1];

static seq_nr nbuffered = 0;

static void put_frame(unsigned char* frame, int len) {
    *(unsigned int*)(frame + len) = crc32(frame, len);

    send_frame(frame, len + 4);
}

static void send_data(seq_nr frame_nr, seq_nr frame_expected,
                      unsigned char* packet, size_t len) {
    struct FRAME s;

    s.kind = FRAME_DATA;
    s.seq = frame_nr;
    s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);

    if (len > PKT_LEN) {
        dbg_frame(
            "Error while sending packet %d with ack %d: Length
too large.\n",
            s.seq, s.ack, len);
```

```
        return;
    }
    memcpy(s.data, packet, PKT_LEN);

    dbg_frame("Packet sent: seq = %d, ack = %d, data id = %d\n",
s.seq, s.ack,
              *(short*)s.data);

    // kind + ack + seq + original data
    put_frame((unsigned char*)&s, 3 + PKT_LEN);
}
static void send_ACK(unsigned char frame_expected) {
    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);
}
static void send_NAK(unsigned char frame_expected) {
    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);
}
static inline bool bewteen(unsigned char a, unsigned char b,
unsigned char c) {
    return ((a <= b) && (b < c)) || ((c < a) && (a <= b)) ||
```

```
          ((b < c) && (c < a));
}

static void network_ready_handler() {
    // from_network_layer
    packet_length[frame_to_send] =
get_packet(buffer[frame_to_send]);
    // expand the sender's window
    nbuffered++;
    // send_data
    send_data(frame_to_send, frame_expected,
buffer[frame_to_send],
              packet_length[frame_to_send]);
    start_timer(frame_to_send, DATA_TIMER);
    stop_ack_timer();

    inc(frame_to_send);
    phl_ready = false;
}
static void physical_ready_handler() { phl_ready = true; }
static void frame_received_handler() {
    // from_physical_layer
    struct FRAME f;
    frame_length = recv_frame((unsigned char*)(&f),
sizeof(f));
    // bad_frame, should return nak
    if (frame_length < 5 || crc32((unsigned char*)&f,
frame_length) != 0) {
        dbg_event("Bad CRC Checksum,Receieve Error!!!\n");
        if (no_nak) {
            send_NAK(frame_expected);
            no_nak = true;
            stop_ack_timer();
        }

        return;
    }
    if (f.kind == FRAME_ACK) dbg_frame("Recv ACK  %d\n",
f.ack);
    if (f.kind == FRAME_NAK) dbg_frame("Recv NAK  %d\n",
f.ack);
    if (f.kind == FRAME_DATA) {
```

```
        dbg_frame("Recv DATA %d %d, ID %d\n", f.seq, f.ack,
*(short*)f.data);
        if (f.seq == frame_expected) {
            put_packet(f.data, frame_length - 7);
            no_nak = true;
            inc(frame_expected);
            start_ack_timer(ACK_TIMER);
        } else if (no_nak) {
            send_NAK(frame_expected);
            no_nak = false;
            stop_ack_timer();
        }
    }

    // stop all the timers before
    while (bewteen(ack_expected, f.ack, frame_to_send)) {
        nbuffered--;
        stop_timer(ack_expected);
        inc(ack_expected);
    }

    if (f.kind == FRAME_NAK) {
        stop_timer(ack_expected + 1);
        frame_to_send = ack_expected;
        // resend those expected packet
        for (seq_nr i = 0; i < nbuffered; i++) {
            send_data(frame_to_send, frame_expected,
buffer[frame_to_send],
                      packet_length[frame_to_send]);
            start_timer(frame_to_send, DATA_TIMER);
            stop_ack_timer();

            inc(frame_to_send);
        }

        phl_ready = false;
    }
    return;
}
static void data_timeout_handler(int* arg) {
    dbg_event("---- DATA %d timeout\n", arg);
    frame_to_send = ack_expected;
```

```
    for (seq_nr i = 0; i < nbuffered; i++) {
        send_data(frame_to_send, frame_expected,
buffer[frame_to_send],
                  packet_length[frame_to_send]);
        start_timer(frame_to_send, DATA_TIMER);
        stop_ack_timer();

        inc(frame_to_send);
    }

    phl_ready = false;
}
static void ACK_timeout_handler() {
    send_ACK(frame_expected);
    stop_ack_timer();
}

void (*event_handler[])(int*) = {
    [NETWORK_LAYER_READY] = network_ready_handler,
    [PHYSICAL_LAYER_READY] = physical_ready_handler,
    [FRAME_RECEIVED] = frame_received_handler,
    [DATA_TIMEOUT] = data_timeout_handler,
    [ACK_TIMEOUT] = ACK_timeout_handler,
};

int main(int argc, char** argv) {
    int event, arg;

    protocol_init(argc, argv);
    disable_network_layer();
    for (;;) {
        event = wait_for_event(&arg);

        event_handler[event](&arg);

        if (nbuffered < MAX_SEQ && phl_ready)
            enable_network_layer();
        else
            disable_network_layer();
    }

    return 0;
```

```
}


// GBN.h
// not modified

/* FRAME kind */
#define FRAME_DATA 1
#define FRAME_ACK 2
#define FRAME_NAK 3

/*
    DATA Frame
    +=========+========+========+===============+========+
    | KIND(1) | SEQ(1) | ACK(1) | DATA(240~256) | CRC(4) |
    +=========+========+========+===============+========+

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

    NAK Frame
    +=========+========+========+
    | KIND(1) | ACK(1) | CRC(4) |
    +=========+========+========+
*/
```