滑动窗口实验报告

1800017781 程芷怡 元培学院

一、实验要求和接口

根据实验要求，停等协议函数和回退n帧重传协议函数的测试函数包括停等协议测试函数 `stud_slide_window_stop_and_wait` 和回退 N 帧协议测试函数 `stud_slide_window_back_n_frame`，在下列情况系统会用学生的测试函数。

当发送端需要发送帧时，会调用学生测试函数，并置参数 `messageType` 为 `MSG_TYPE_SEND`，测试函数应该将该帧缓存，存入发送队列中。若发送窗口还未打开到规定限度，则打开一个窗口，并将调用 `SendFRAMEPacket` 函数将该帧发送。若发送窗口已开到限度，则直接返回，相当于直接进入等待状态。

当发送端收到接收端的 ACK 后，会调用学生测试函数，并置参数 `messageType` 为 `MSG_TYPE_RECEIVE`，测试函数应该检查 ACK 值后，将该 ACK 对应的窗口关闭。由于关闭了窗口，等待发送的帧可以进入窗口并发送，因此，此时若发送队列中存在等待发送的帧应该将一个等待发送的帧发送并打开一个新的窗口。

发送每发送一个帧，系统都会为他创建一个定时器，当被成功 ACK 后，定时器会被取消，若某个帧在定时器超时时间仍未被 ACK，系统则会调用测试函数，并置参数 `messageType` 为 `MSG_TYPE_TIMEOUT`，告知测试函数某帧超时，测试函数应该将根据帧序号将该帧以及后面发送过的帧重新发送。

选择重传协议函数的测试函数包括 `stud_slide_window_choice_frame_resend`，在下列情况下系统会调用学生的测试函数。

当发送端需要发送帧时，会调用学生测试函数，并置参数 `messageType` 为 `MSG_TYPE_SEND`，测试函数应该将该帧缓存，存入发送队列中。若发送窗口还未打开到规定限度，则打开一个窗口，并将调用 `SendFRAMEPacket` 函数将该帧发送。若发送窗口已开到限度，则直接返回，相当于直接进入等待状态。

当发送端收到接收端的 ACK 后，会调用学生测试函数，并置参数 `messageType` 为 `MSG_TYPE_RECEIVE`，测试函数应该检查 ACK 值后，将该 ACK 对应的窗口关闭。由于关闭了窗口，等待发送的帧可以进入窗口并发送，因此，此时若发送队列中存在等待发送的帧应该将一个等待发送的帧发送并打开一个新的窗口。

当发送端收到接收端的帧类型为 NAK(表示出错的帧号)后，会调用学生
测试函数，并置参数 `messageType` 为 `MSG_TYPE_RECEIVE`，测试函数应该检查 NAK 值后，系统则会调用测试函数，告知测试函数某帧错误，测试函数应该将根据帧序号将该帧重新发送。

需要实现的接口有：

```
int stud_slide_window_stop_and_wait(char *pBuffer, int bufferSize, UINT8 messageType)
```

其中， pBuffer 指向的结构定义在下有述。 pBuffer 指向系统要发送或接收到的帧内容的指针，或者指向超时消息中超时帧的序列号内容的指针。 bufferSize 是 pBuffer 表示内容的长度。
messageType 是传入的消息类型，可以为以下几种情况：
MSG_TYPE_TIMEOUT 某个帧超时
MSG_TYPE_SEND 系统要发送一个帧
MSG_TYPE_RECEIVE 系统接收到一个帧的 ACK
对于 MSG_TYPE_TIMEOUT 消息， pBuffer 指向数据的前四个字节为超时帧的序列号，以 UINT32 类型存储，在与帧中的序列号比较时，请注意字节序，并进行必要的转换。

```
int stud_slide_window_back_n_frame(char *pBuffer, int bufferSize, UINT8 messageType)
```

```
int stud_slide_window_choice_frame_resend(char *pBuffer, int bufferSize,UINT8 messageType)
```

系统提供的接口有：

```
extern void SendFRAMEPacket(unsigned char* pData, unsigned int len);
```

pData 是指向要发送的帧的内容的指针。
len 是要发送的帧的长度。

二、数据结构设计

实验所用到的数据结构有三个： frame_head 和 frame 是实验要求中提到的api接口。分别表示帧头和帧。帧头中包括帧的类型（ data 、 ack 、 nak ）。 seq 序号、 ack 序号和数据，数据缓冲区大小为100字节。帧中记录了帧头的大小和帧的大小。代码如下所示。

```
struct frame_head {
    frame_kind kind;
    unsigned int seq;
    unsigned int ack;
    unsigned char data[100];
};

struct frame {
    frame_head head;
    int size;
};
```

另外，由于每次函数提供发送帧时还有一个参数 bufferSize 记录了帧的大小。因此在将帧储存时需要记录这个参数，因此设置 store_frame 的数据结构，储存帧和发送帧的大小。

```
struct store_frame {
    frame f;
    unsigned int size;
    store_frame() {}
    store_frame(frame _frame, int _size): f(_frame), size(_size) {}
};
```

三、函数过程

1. 停等协议函数

根据实验要求，停等协议函数的主体是一个 switch 分支选择语句，用来判断信息类型并根据不同的类型做出不同的反应。在函数开头设置了两个static变量，记录发送未 ack 的帧号 expect 和储存帧的向量 frame_list 。

当信息类型为 MSG_TYPE_SEND 时，使用指针指向 pBuffer ，并将帧 push 进 frame_list 中。如果 expect 值为刚 push 进的帧号，说明此时没有在等待 ack 的帧，直接将该帧发送。否则等待并返回。经过试验发现帧中的信息已经是大端法存储，因此不需要 htonl 函数进行转换。

当信息类型是 MSG_TYPE_RECEIVE 时，从 frame_list 中取出expect的帧并逐一和收到的帧的 ack 值进行比较，更新 expect 值，并重传一个帧。此处假定进行了累计确认，即 ack 之前的帧都已经成功传输。成功通过测试。

当信息类型是 MSG_TYPE_TIMEOUT 时，提取 pBuffer 的前四个字节作为超时序号，将 frame_list 中对应的帧重传。此处有一个小问题，没有将 expect 对应的帧序号和 timeout_seq 进行比对，但仍然通过了测试。这是因为停等协议的帧一定是顺序传输的。事实上 MSG_TYPE_RECEIVE 分支 while 循环去除也可以通过测试。

```
int stud_slide_window_stop_and_wait(char *pBuffer, int bufferSize, uint_8 messageType) {
    static vector<store_frame> frame_list;
    static int expect = 0;
    switch (messageType) {
        case MSG_TYPE_TIMEOUT: {
            uint32_t timeout_seq = *(uint32_t *)(pBuffer);
            store_frame send_frame = frame_list[expect];
            printf("Timeout, resend %d\n", timeout_seq);
            SendFRAMEPacket((unsigned char *)(&(send_frame.f)), send_frame.size);
            break;
        }
        case MSG_TYPE_RECEIVE: {
            store_frame send_frame = frame_list[expect];
            frame ack_frame = *(frame *)(pBuffer);
            printf("Receive ack %d\n", ntohl(ack_frame.head.ack));
            while (send_frame.f.head.seq <= ack_frame.head.ack) {
                expect++;
                send_frame = frame_list[expect];
            }
            printf("Send a new frame %d\n", ntohl(send_frame.f.head.seq));
            SendFRAMEPacket((unsigned char *)&(send_frame.f), send_frame.size);
            break;
        }
        case MSG_TYPE_SEND: {
            //需要发送这一个帧
            frame *send_frame = (frame *)(pBuffer);
            printf("Frame info: %d,%d\n", ntohl(send_frame->head.seq), send_frame->head.data);
            frame_list.push_back(store_frame(*send_frame, bufferSize));
            if (expect == frame_list.size() - 1) {
                printf("Sending frame %d...\n", ntohl(send_frame->head.seq));
                SendFRAMEPacket((unsigned char *)send_frame, bufferSize);
            } else {
                printf("Waiting frame %d...\n", ntohl(send_frame->head.seq));
            }
            break;
        }
    }
    return 0;
}
```

2. 回退n帧重传协议函数

根据实验要求，回退n帧重传协议函数和停等协议函数的主体结构相似。不同的是在函数开头增加了 `window_size` 字段，以及将 `expect` 改成了 `window_begin` 和 `window_end` 字段。

当信息类型是 `MSG_TYPE_SEND` 时，如果 `frame_list` 的大小小于 `window_size` 时，说明窗口没有开满，发送该帧并开大窗口，否则将该帧储存并返回，进入等待状态。

当信息类型是 MSG_TYPE_RECEIVE 时，找到 seq 值为 ack 值的帧，将 window_begin 设置为其下一个帧，更改 window_end 值，并将新窗口内其他未发送的帧发送。注意 window_begin + window_size > frame_list.size() 时需要将 window_end 设置为 frame_list.size() 。这里的实现也有一个问题，就是在信息类型是 MSG_TYPE_SEND 时应该比较的不是 window_size 和 frame_list.size() ，而是 window_size 和 window_end - window_begin ，但仍然通过了测试。

当信息类型是 MSG_TYPE_TIMEOUT 时，根据实验要求，将窗口内所有的帧重传。

```
int stud_slide_window_stop_and_wait(char *pBuffer, int bufferSize, uint_8 messageType) {
    static vector<store_frame> frame_list;
    static int expect = 0;
    switch (messageType) {
        case MSG_TYPE_TIMEOUT: {
            uint32_t timeout_seq = *(uint32_t *)(pBuffer);
            store_frame send_frame = frame_list[expect];
            printf("Timeout, resend %d\n", timeout_seq);
            SendFRAMEPacket((unsigned char *)(&(send_frame.f)), send_frame.size);
            break;
        }
        case MSG_TYPE_RECEIVE: {
            store_frame send_frame = frame_list[expect];
            frame ack_frame = *(frame *)(pBuffer);
            printf("Receive ack %d\n", ntohl(ack_frame.head.ack));
            while (send_frame.f.head.seq <= ack_frame.head.ack) {
                expect++;
                send_frame = frame_list[expect];
            }
            printf("Send a new frame %d\n", ntohl(send_frame.f.head.seq));
            SendFRAMEPacket((unsigned char *)&(send_frame.f), send_frame.size);
            break;
        }
        case MSG_TYPE_SEND: {
            //需要发送这一个帧
            frame *send_frame = (frame *)(pBuffer);
            printf("Frame info: %d,%d\n", ntohl(send_frame->head.seq), send_frame->head.data);
            frame_list.push_back(store_frame(*send_frame, bufferSize));
            if (expect == frame_list.size() - 1) {
                printf("Sending frame %d...\n", ntohl(send_frame->head.seq));
                SendFRAMEPacket((unsigned char *)send_frame, bufferSize);
            } else {
                printf("Waiting frame %d...\n", ntohl(send_frame->head.seq));
            }
            break;
        }
    }
    return 0;
}
```

3. 选择重传协议函数

选择重传协议函数和回退n帧协议函数基本相同，但是数据类型多了 `nak` 而在收到 `MSG_TYPE_RECEIVE` 时需要检查数据类型，这里的坑是网络序（大端法）和主机序（小端法）的转换。当数据类型是 `ack` 时函数行为和回退n帧协议函数相同，而数据类型是 `nak` 时在窗口内检测序号等于接受帧帧头的 `ack` 字段的帧重新发送。至于是 `ack` 字段还是 `seq` 字段，实验要求并不明晰，经过尝试后得出。

```
int stud_slide_window_choice_frame_resend(char *pBuffer, int bufferSize, uint_8 messageType) {
    static const int window_size = 4;
    static vector<store_frame> frame_list;
    static int window_begin, window_end;

    switch (messageType) {
        case MSG_TYPE_TIMEOUT: {
            uint32_t timeout_seq = *(uint32_t *)(pBuffer);
            printf("Timeout seq: %d\n", timeout_seq);
            for (int i = window_begin; i < window_end; i++) {
                printf("Timeout, resend %d\n", ntohl(frame_list[i].f.head.seq));
                SendFRAMEPacket((unsigned char *)(&(frame_list[i].f)), frame_list[i].size);
            }
            break;
        }
        case MSG_TYPE_RECEIVE: {
            frame ack_frame = *(frame *)(pBuffer);
            if (ntohl(ack_frame.head.kind) == ack) {
                printf("Receive ack %d\n", ntohl(ack_frame.head.ack));
                for (int i = window_begin; i < window_end; i++) {
                    if (frame_list[i].f.head.seq == ack_frame.head.ack) {
                        for (int j = window_end; j < i + 1 + window_size, j < frame_list.size(); j++) {
                            printf("Opening window, sending %d\n", ntohl(frame_list[j].f.head.seq));
                            SendFRAMEPacket((unsigned char *)(&(frame_list[j].f)), frame_list[j].size);
                        }
                        window_begin = i + 1;
                        window_end = min((int)frame_list.size(), window_begin + window_size);
                        break;
                    }
                }
            }
            else if (ntohl(ack_frame.head.kind) == nak) {
                printf("Receive ack %d\n", ntohl(ack_frame.head.ack));
                for (int i = window_begin; i < window_end; i++) {
                    if (frame_list[i].f.head.seq == ack_frame.head.ack) {
                        printf("Error, resend %d\n", ntohl(ack_frame.head.ack));
                        SendFRAMEPacket((unsigned char *)(&(frame_list[i].f)), frame_list[i].size);
                        break;
                    }
                }
            }
            break;
        }
        case MSG_TYPE_SEND: {
            frame *send_frame = (frame *)(pBuffer);
            frame_list.push_back(store_frame(*send_frame, bufferSize));
            if (frame_list.size() <= window_size) {
                window_end++;
                printf("Sending frame %d...\n", ntohl(send_frame->head.seq));
                SendFRAMEPacket((unsigned char *)send_frame, bufferSize);
            } else {
```

```
                printf("Waiting frame %d...\n", ntohl(send_frame->head.seq));
            }
            break;
        }
    }
    return 0;
}
```