1 定义的结构和变量

1.1 typedef enum frame_kind, typedef struct frame_head, typedef struct frame

```
typedef enum { data, ack, nak } frame_kind;
typedef struct frame_head
{
  frame_kind kind; //帧类型
  unsigned int seq; //序列号
  unsigned int ack; //确认号
  unsigned char data[100];//数据
};
typedef struct frame
{
  frame_head head; //帧头
  unsigned int size; //数据的大小
};
```

对于模版中已经给出的结构和变量，我就不做重复解释了。

1.2 typedef struct Buffer

```
typedef struct Buffer{ // 用链表构建缓冲区表示等待发送队列
    frame *message;
    unsigned int size;
    Buffer *nextBuffer;
}
```

我们考虑用链表构建缓冲区表示等待发送队列，其中每个Buffer是链表中的一个结点。等待队列是一个单向链表，有一个指向头部结点的指针 Buffer *waitLine。Buffer中存放一个指向下一个链表节点的指针nextBuffer，一个指向frame的指针message，和一个表示frame大小的变量size。

1.3 typedef struct Window

```
typedef struct Window{ // 滑动窗口 用来缓存数据
    frame message;
    unsigned int size;
}
```

由于滑动窗口的大小是固定的，所以没有必要使用链表，直接用数组表示即可，我们这里定义滑动窗口数组的每个元素为Window。其中有一个存放frame的变量message和一个表示frame大小的变量size。

1.4 函数中的变量

将在每个函数中进行解释。

2 函数及实现逻辑

2.1 Buffer *AddFrame(Buffer *waitLine, char *pBuffer, int bufferSize)

```
Buffer *AddFrame(Buffer *waitLine, char *pBuffer, int bufferSize){ // 将buffer加到
waitLine中
    Buffer *addOne = new Buffer;

    // 构造addOne
    addOne->message = new frame;
    memcpy(addOne->message, pBuffer, bufferSize);
    addOne->size = bufferSize;
    addOne->nextBuffer = nullptr;

    // 添加addOne
    if(waitLine == nullptr)
        waitLine = addOne;
    else{
        Buffer *tmp = waitLine;
        while(tmp->nextBuffer != nullptr)
            tmp = tmp->nextBuffer;
        tmp->nextBuffer = addOne;
    }

    return waitLine;
}
```

- 函数作用：用于将等待的Buffer加入到等待队列waitLine中。
- 函数参数：
  - waitLine指向等待队列的头结点。
  - pBuffer指向等待的帧。
  - bufferSize表示等待的帧的大小。
- 函数的返回值：一个指向等待队列头结点的指针。（若waitLine为空则会修改waitLine指向一个新结点，所以需要AddFrame有一个返回值，否则waitLine不会改变自身值）
- 实现逻辑：创建一个Buffer指针 addOne 并为它 new 一个新空间，再为addOne->message new 一个新空间，将pBuffer的内容copy到addOne->message中。若waitLine是空，则等待队列为空，就让waitLine和addOne指向同一个Buffer作为等待队列的头结点；若waitLine非空，则遍历链表到最后一个结点，让最后一个结点的nextBuffer和addOne指向同一个Buffer，将Buffer加到链表的末尾。最后返回指向头结点的指针。

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

```
/*
* 停等协议测试函数
*/
int stud_slide_window_stop_and_wait(char* pBuffer, int bufferSize, UINT8 messageType)
// pBuffer已经是网络序了
{
    static Window window; // 窗口大小为1
    static Buffer* waitLine = NULL; // 等待队列
    static bool flag = false; // 只有一个窗口 用bool表示是否被使用就行

    int ackRemote;
    int seqLocal;
    switch (messageType) {
    case MSG_TYPE_SEND:
        if (!flag) { // 窗口未被使用 直接发送
            SendFRAMEPacket((unsigned char*)pBuffer, (unsigned int)bufferSize);
            window.message = *((frame*)pBuffer);
            window.size = (unsigned int)bufferSize;
            flag = true;
        }
        else { // 窗口已经被使用 将pBuffer存到waitLine中
            waitLine = AddFrame(waitLine, pBuffer, bufferSize);
        }
        break;


    case MSG_TYPE_RECEIVE:
        ackRemote = ntohl(((frame*)pBuffer)->head.ack);
        seqLocal = ntohl(window.message.head.seq);
        printf("Receive a message, seq is %d\n", ackRemote);
        if (ackRemote == seqLocal) {
            flag = false; // 窗口未被使用
            window.size = 0; // 缓存的不需要了
            if (waitLine != NULL) { // 若等待队列有buffer 就放到窗口里
                Buffer* firstOne = waitLine;
                waitLine = waitLine->nextBuffer;

                SendFRAMEPacket((unsigned char*)(firstOne->message), firstOne->size);
                window.message = *(firstOne->message);
                window.size = firstOne->size;
                flag = true;

                // 释放内存
                delete (firstOne->message);
                delete firstOne;
            }
        }
```

```
        break;

    case MSG_TYPE_TIMEOUT:
        SendFRAMEPacket((unsigned char*)&(window.message), window.size); // 重发帧

    }

    return 0;
}
```

函数作用、函数参数和函数返回值已经在指导书中说明，故这里只阐述实现逻辑。

- 由于窗口大小为1，所以只需要一个Window变量window，不需要创建一个数组。
- waitLine是指向等待队列头结点的指针，初值为nullptr。
- 由于只有一个窗口，所以只需要一个bool变量flag表示窗口是否被使用。
- 若messageType == MSG_TYPE_SEND，flag = false则直接发送帧并将帧的副本放入window中并设flag = true；flag = true则将帧存入等待队列中。
- 若messageType == MSG_TYPE_RECEIVE，比较发送的帧的ack和window中帧的seq是否相同，若相同则删去window中帧的副本，若等待队列中有帧等待发送则从等待队列中取一个帧发送并将副本存入window，否则flag = false；若不同则不做处理。
- 若messageType == MSG_TYPE_TIMEOUT，直接把window中的帧重新发送即可。

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

```
/*
* 回退n帧测试函数
*/
int stud_slide_window_back_n_frame(char* pBuffer, int bufferSize, UINT8 messageType)
{
    static Window window[WINDOW_SIZE_BACK_N_FRAME]; // 窗口大小为WINDOW_SIZE_BACK_N_FRAME
    static Buffer* waitLine = NULL; // 等待队列
    static int start = 0;
    static int end = 0; // 分别指向window的头和尾
    // end指向最后一个buffer之后的位置 start和end使用的时候一直增加
用%WINDOW_SIZE_BACK_N_FRAME表示在window中的index 表示一个环
    int ackRemote;
    int seqLocal;
    switch (messageType) {
    case MSG_TYPE_SEND:
        if (end - start < WINDOW_SIZE_BACK_N_FRAME) { // 窗口未被完全使用 可以发送
            SendFRAMEPacket((unsigned char*)pBuffer, (unsigned int)bufferSize);
            window[end % WINDOW_SIZE_BACK_N_FRAME].message = *((frame*)pBuffer);
            window[end % WINDOW_SIZE_BACK_N_FRAME].size = (unsigned int)bufferSize;
            end++;
        }
        else { // 窗口已经被使用 将pBuffer存到waitLine中
```

```
            waitLine = AddFrame(waitLine, pBuffer, bufferSize);
        }
        break;


    case MSG_TYPE_RECEIVE:
        ackRemote = ntohl(((frame*)pBuffer)->head.ack);
        // 和停等协议的区别是要和window中所有的frame的seq比较 删除匹配到的第i个和之前所有的frame
        for (int i = start; i < end; i++) {
            seqLocal = ntohl(window[i % WINDOW_SIZE_BACK_N_FRAME].message.head.seq);
            if (seqLocal == ackRemote) {
                for (int j = start; j <= i; j++) { // 删除前面所有缓存的frame
                    window[j % WINDOW_SIZE_BACK_N_FRAME].size = 0;
                }
                start = i + 1;

                // 如果等待队列有buffer，放到窗口里
                while (end - start < WINDOW_SIZE_BACK_N_FRAME && waitLine != NULL) {
                    Buffer* firstOne = waitLine;
                    waitLine = waitLine->nextBuffer;

                    SendFRAMEPacket((unsigned char*)(firstOne->message), firstOne-
>size);
                    window[end % WINDOW_SIZE_BACK_N_FRAME].message = *(firstOne-
>message);
                    window[end % WINDOW_SIZE_BACK_N_FRAME].size = firstOne->size;
                    end++;

                    // 释放内存
                    delete firstOne->message;
                    delete firstOne;
                }

                break;
            }
        }

        break;


    case MSG_TYPE_TIMEOUT:
        // 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
        int ackRemote = *(unsigned int*)pBuffer;
        // 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
        for (int i = start; i < end; i++) {
            int seqLocal = ntohl(window[i %
WINDOW_SIZE_BACK_N_FRAME].message.head.seq);
            if (seqLocal == ackRemote) {
                for (int j = start; j < end; j++) {
```

```
                    SendFRAMEPacket((unsigned char*)&(window[j %
WINDOW_SIZE_BACK_N_FRAME].message), window[j % WINDOW_SIZE_BACK_N_FRAME].size);
                }
                break;
            }
        }
    }

    return 0;
}
```

函数作用、函数参数和函数返回值已经在指导书中说明，故这里只阐述实现逻辑。

- 窗口大小可以>1，所以创建一个Window数组window。
- waitLine是指向等待队列头结点的指针，初值为nullptr。
- 两个int变量start和end表示滑动窗口的开头和结尾。
- 若messageType == MSG_TYPE_SEND，若end - start < WINDOW_SIZE_STOP_WAIT，则直接发送帧并将帧的副本放入window[end%WINDOW_SIZE_STOP_WAIT]中并让end++；否则则将帧存入等待队列中。
- 若messageType == MSG_TYPE_RECEIVE，从滑动窗口的开头逐一比较pBuffer的ack和window[i]中帧的seq是否相同，若相同则删去window[i]以及之前所有帧的副本，并设start = i+1。若等待队列中有帧等待发送且end - start < WINDOW_SIZE_STOP_WAIT，则从等待队列中取一个帧发送并将副本存入window[end]，令end++；否则不做处理。
- 若messageType == MSG_TYPE_TIMEOUT，取pBuffer的前四个字节作为ack，从滑动窗口的开头逐一比较pBuffer的ack和window[i]中帧的seq是否相同，若相同则重新发送window[i]以及之后所有帧。

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

```
/*
* 选择性重传测试函数
*/
int stud_slide_window_choice_frame_resend(char* pBuffer, int bufferSize, UINT8
messageType)
{

    static Window window[WINDOW_SIZE_BACK_N_FRAME]; // 窗口大小为WINDOW_SIZE_BACK_N_FRAME
    static Buffer* waitLine = NULL; // 等待队列
    static int start = 0;
    static int end = 0; // 分别指向window的头和尾
    // end指向最后一个buffer之后的位置 start和end使用的时候一直增加
用%WINDOW_SIZE_BACK_N_FRAME表示在window中的index 表示一个环

    int kindRemote;
    int ackRemote;
    switch (messageType) {
    case MSG_TYPE_SEND:
        if (end - start < WINDOW_SIZE_BACK_N_FRAME) { // 窗口未被完全使用 可以发送
```

```
            SendFRAMEPacket((unsigned char*)pBuffer, (unsigned int)bufferSize);
            window[end % WINDOW_SIZE_BACK_N_FRAME].message = *((frame*)pBuffer);
            window[end % WINDOW_SIZE_BACK_N_FRAME].size = (unsigned int)bufferSize;
            end++;
        }
        else { // 窗口已经被使用 将pBuffer存到waitLine中
            waitLine = AddFrame(waitLine, pBuffer, bufferSize);
        }
        break;


    case MSG_TYPE_RECEIVE:
        kindRemote = ntohl(((frame*)pBuffer)->head.kind);
        ackRemote = ntohl(((frame*)pBuffer)->head.ack);
        if (kindRemote == ack) {
            // 和停等协议的区别是要和window中所有的frame的seq比较 删除匹配到的第i个和之前所有的
frame
            for (int i = start; i < end; i++) {
                int seqLocal = ntohl(window[i %
WINDOW_SIZE_BACK_N_FRAME].message.head.seq);
                if (seqLocal == ackRemote) {
                    for (int j = start; j <= i; j++) { // 删除前面所有缓存的frame
                        window[j % WINDOW_SIZE_BACK_N_FRAME].size = 0;
                    }
                    start = i + 1;

                    // 如果等待队列有buffer，放到窗口里
                    while (end - start < WINDOW_SIZE_BACK_N_FRAME && waitLine != NULL)
{
                        Buffer* firstOne = waitLine;
                        waitLine = waitLine->nextBuffer;

                        SendFRAMEPacket((unsigned char*)(firstOne->message), firstOne-
>size);
                        window[end % WINDOW_SIZE_BACK_N_FRAME].message = *(firstOne-
>message);
                        window[end % WINDOW_SIZE_BACK_N_FRAME].size = firstOne->size;
                        end++;

                        // 释放内存
                        delete firstOne->message;
                        delete firstOne;
                    }

                    break;
                }
            }
        }
        else if (kindRemote == nak) {
```

```
            for (int i = start; i < end; i++) {
                int seqLocal = ntohl(window[i %
WINDOW_SIZE_BACK_N_FRAME].message.head.seq);
                if (seqLocal == ackRemote) {
                    SendFRAMEPacket((unsigned char*)&(window[i %
WINDOW_SIZE_BACK_N_FRAME].message), window[i % WINDOW_SIZE_BACK_N_FRAME].size);
                    break;
                }
            }
        }
        else {
            printf("something wrong happened\n");
        }
        break;



    case MSG_TYPE_TIMEOUT:
        // 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
        int ackRemote = *(unsigned int*)pBuffer;
        // 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
        for (int i = start; i < end; i++) {
            int seqLocal = ntohl(window[i %
WINDOW_SIZE_BACK_N_FRAME].message.head.seq);
            if (seqLocal == ackRemote) {
                for (int j = start; j < end; j++) {
                    SendFRAMEPacket((unsigned char*)&(window[j %
WINDOW_SIZE_BACK_N_FRAME].message), window[j % WINDOW_SIZE_BACK_N_FRAME].size);
                }
                break;
            }
        }
    }
    return 0;
}
```

函数作用、函数参数和函数返回值已经在指导书中说明，故这里只阐述实现逻辑。

基本和回退N帧函数完全相同，这里只说不同之处：

- 若messageType == MSG_TYPE_RECEIVE，判断pBuffer的kind，若kind == ack则之后的操作与回退N帧函数相同；若kind == nak，则从滑动窗口的开头逐一比较pBuffer的ack和window[i]中的seq是否相同，若相同则重新发送window[i]。

3 实验过程的重点难点及遇到的问题

- frame中存放的size大小是指frame_head中data的大小，不是整个frame的大小，所以在Buffer和Window中需要一个表示整个frame大小的变量size。
- 对于回退N帧和选择性重传实验中，滑动窗口的大小为WINDOW_SIZE_STOP_WAIT（咨询的同学）。
- 对于回退N帧和选择性重传实验，由于窗口大小不为一，若用一个bool数组flag表示每一个窗口是否被使用，在判断窗口是否占满时需要遍历数组，很麻烦，所以换一种方法。我们把线性数组想象成一个环，设置两个int变量start和end，start是被使用的滑动窗口开头的下标，end是被使用的滑动窗口最后一个元素之后的下标，所以end - start即为滑动窗口目前已被使用的大小。而将start和end转化成window中的下标即为start % WINDOW_SIZE_STOP_WAIT 和 start % WINDOW_SIZE_STOP_WAIT。
- 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，此时取ack与其他情况略有区别。（但是不知道这里是否需要将网络序转成主机序，没有进行实验）
- 注意所有求ack和seq的时候都要调用ntohl将网络序转成主机序。
- 注意！在做实验中发现错误：系统会多次调用你的函数，但在函数中创建的局部变量不同函数不会共享！所以在函数中创建的window，waitLine和begin，end等都要加static！确保所有函数共享这些变量。
- 当你将pBuffer的内容移入waitLine和将waitLine的内容移入pBuffer时，一定要注意用memcpy！而不要用=，=只是让他们指向了相同的内存空间，如果这时一个delete了，另一个的内容也没了。
- 在TIMEOUT或者NAK时，需要将窗口中的所有帧全部重传！

4 感想和建议

通过滑动窗口协议实验，我对停等协议、回退N帧协议和选择重传协议有了更深的理解。

但个人感觉完成代码编写后，自己只是协议中的很小的写了一部分函数，并没有参与设计完整的协议内容，比如接收端的行为和发送端计时器等行为的设计。

由于个人原因还没有去机房进行代码实验，代码内容若有小bug还请见谅，等4月1日返校后我会尽快去机房完成实验并更新正确的代码。