NetRiver 1–滑动窗口协议

张煌昭，1400017707，元培学院

摘要—本次作业对滑动窗口协议中停等协议（Stop-and-Wait），回退 N 协议（Go-back-N），选择重传（Selective-Repeat）的发端进行了实现，并于 NetRiver 平台进行提交和测试。本次报告使用 Overleaf L^AT_EX 在线平台编写[1]，作业源码赋于报告之后。

I. 介绍

本次作业，要求使用 C/C++ 语言，在 NetRiver 的链路层虚拟环境中，实现停等协议，回退 N 协议，和选择重传协议的发送端。要求可以响应发送请求并正确得发送或等待，接收帧确认（ACK）和帧错误（NACK）并进行正确的应对，响应超时（Timeout）并进行相应的处理。

停等协议的具体要求以及实现详情见第 II节，回退 N 协议的具体要求及实现详情见第 III节，选择重传协议的具体要求及实现详情见第 IV节。

II. 停等协议

A. 要求

停等协议下，1) 发送端每发送一个帧，都必须等待收到该帧的 ACK 后，才可以继续发送；2) 在等待期间，如果还有新的待发送的帧到来，则进入等待队列等待，直至可以继续发送时按队列顺序发送；3) 当 ACK 超过阈值时间未到，则发送端默认超时，重新发送之前发送的帧，并等待该 ACK。上述情况均满足时，停等协议可以正常运行。

本次作业下，发端系统通过以下函数实现停等协议。

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

当发端系统需要发送时，调用函数，并设置 messageType 参数为 MSG_TYPE_SEND。函数将帧（即 pBuffer 参数）存入发送队列中，若目前状态为可以发送，则进行发送，并在发送后将状态切换为等待 ACK；否则状态为等待 ACK，则该待发送帧在队列中等待，当等待状态结束（即接收到 ACK）后，等待队列中按队列顺序的第一个帧继续发送。

当发端系统收到 ACK 时，调用函数，并设置 messageType 参数为 MSG_TYPE_RECEIVE。函数接收到 ACK，将状态切换为可以发送；若此时等待队列中仍有待发送的帧，则按队列顺序取第一个帧发送。

当发端系统超时时，调用函数，并设置 messageType 参数为 MSG_TYPE_TIMEOUT。函数接收超时，将最近发送过的帧重新发送。

B. 实现

按照 NetRiver 手册[2]定义消息类型和帧结构如下。

```
typedef enum {data,ack,nak} frame_kind;
typedef struct frame_head
{
    frame_kind kind;
    unsigned int seq;
    unsigned int ack;
    unsigned char data[100];
};
typedef struct frame
{
    frame_head head;
    unsigned int size;
};
```

由于整个发端存在两个状态——可以发送和等待 ACK，因此在函数中使用静态的 bool 变量 send 对状态进行标记。当 send 为 true 时，代表可以发送；当 send 为 false 时，代表等待 ACK。

此外，由于发送过程中会需要等待，因此使用使用队列 sendList 作为等待队列。又由于停等协议本质上滑动窗口大小为 1，因此等待队列的第一个元素就是窗口中元素，不再需要额外的窗口。

根据调用参数 messageType 的不同分情况讨论：1) 为 MSG_TYPE_SEND 时，首先将 pBuffer 转换为帧

[1]本报告源码可通过以下 git 命令获得，
git clone https://git.overleaf.com/15853721gmnmbcjkdywj

[2]计算机网络实验系统实验指导书——NetRiver 系列

格式，存入队尾，若 send 为 false 则返回等待；否则可以发送，队首的帧发送后，将 send 置为 true。2) 为 MSG_TYPE_RECEIVE 时，接收到队首帧对应的 ACK，将队首帧弹出，若队列中非空则继续发送队首帧。3) 为 MSG_TYPE_TIMEOUT 时，出现超时，将队首元素重新发送并等待 ACK。

程序的 C/C++ 风格的伪代码如下所示。

```
int stud_slide_window_stop_and_wait(char *pBuffer,
                                    int bufferSize,
                                    UINT8 messageType)
{
    switch (messageType)
    {
    case MSG_TYPE_SEND:
        // 复制pBuffer
        // 将帧加入队尾
        if (可以发送)
        {
            // 发送队首的帧
            // 进入等待状态，不允许再发送
        }
        break;
    case MSG_TYPE_RECEIVE:
        // 弹出队首帧
        // 由等待状态进入可以发送状态
        if (队列非空)
        {
            // 发送队首的帧
            // 进入等待状态，不允许再发送
        }
        break;
    case MSG_TYPE_TIMEOUT:
        // 发送队首的帧
        // 进入等待状态，不允许再发送
        break;
    default:
        // 程序运行时，不会到达此处
        break;
    }
}
```

III. 回退 N 协议

A. 要求

回退 N 协议下，发送端具有等待队列和发送窗口，1) 发送端只有在发送窗口未满时才可以发送，若发送窗口已满则必须等待直至可以发送；2) 接收某帧的 ACK，说明该帧之前发送的所有帧均已被收端正确接收，则将所有正确接收的帧从窗口中移出，若发送队列非空，则继续发送直至队列空或窗口填满；3) 当 ACK 超过阈值时间未到，则发送端默认超时，重新发送窗口内的所有帧，并等待 ACK。上述情况均满足时，回退 N 协议可以正常运行。

本次作业下，发端系统通过以下函数实现回退 N 协议。

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

当发端系统需要发送时，调用函数，并设置 messageType 参数为 MSG_TYPE_SEND。函数将帧（即 pBuffer 参数）存入发送队列中，若窗口未满，则发送该帧，并将其移入窗口中；否则不允许发送，在队列中等待。

当发端系统收到 ACK 时，调用函数，并设置 messageType 参数为 MSG_TYPE_RECEIVE。函数接收到 ACK，将窗口中 ACK 对应的帧之前的帧全部移出；若此时等待队列中仍有待发送的帧，则按队列顺序逐个发送直至队列空或窗口满。

当发端系统超时时，调用函数，并设置 messageType 参数为 MSG_TYPE_TIMEOUT。函数接收超时，将最近发送过的帧重新发送。

B. 实现

帧结构和帧类型同第II-A节。

定义发端需要使用的等待队列 queue<frame> sendList 和发送窗口 deque<frame> sendWindow。状态可以通过队列和窗口的填充情况进行判断，不需要额外的标记。

根据调用参数 messageType 的不同分情况讨论：1) 为 MSG_TYPE_SEND 时，首先将 pBuffer 转换为帧格式，存入队尾，若窗口中元素个数大于等于窗口大小，则等待；否则可以发送，将队列中元素发送直至队列空或窗口满。2) 为 MSG_TYPE_RECEIVE 时，接收到窗口内某帧对应的 ACK，将窗口内帧逐个弹出，直至弹出帧号对应 ACK 的帧，若等待队列非空，则按队列顺序发送，并将其移入窗口中，直至队列空或窗口满。3) 为 MSG_TYPE_TIMEOUT 时，出现超时，将窗口中所有帧重新发送并等待 ACK。

考虑发送时的情况，由于在接收 ACK 时会自动地将窗口填充或将队列清空，因此在发送时只可能存在两种情况：a) 窗口未满，队列空；b) 窗口满，队列非空（或恰好为空）。只有 a) 情况下会需要在发送时将队列中的帧发送，此时队列中仅仅有刚刚存入的新帧，即只有一帧，因此只需要发送队列中的一帧即可。

程序的 C/C++ 风格的伪代码如下所示。

```
int stud_slide_window_back_n_frame(char *pBuffer,
                                   int bufferSize,
                                   UINT8 messageType)
{
    switch (messageType)
    {
    case MSG_TYPE_SEND:
        // 复制pBuffer
        // 将帧加入队尾
        if (窗口内帧数 < 窗口大小)
        {
            // 发送队首的帧
            // 将该帧从队首移至窗口尾
        }
        break;
    case MSG_TYPE_RECEIVE:
        // 复制pBuffer
        /* 将窗口中ACK帧号前的等待确认的帧清空 */
        while(窗口非空 && 窗口首帧号 != ACK帧号)
            // 弹出窗口首帧
        if (窗口非空)
            // 弹出窗口首帧
        while (窗口内帧数 < 窗口大小 && 队列非空)
        {
            // 发送队首的帧
            // 将该帧从队首移至窗口尾
        }
        break;
    case MSG_TYPE_TIMEOUT:
        /* 重新发送窗口中所有帧 */
        for (iter = 窗口首帧; iter != 窗口尾帧; iter++)
            // 发送iter帧
        break;
    default:
        // 程序运行时，不会到达此处
        break;
    }
}
```

IV. 选择重传协议

A. 要求

选择重传协议下，发送端具有等待队列和发送窗口，1) 发送端只有在发送窗口未满时才可以发送，若发送窗口已满则必须等待直至可以发送；2) 接收某帧的 ACK，说明该帧之前发送的所有帧均已被收端正确接收，则将所有正确接收的帧从窗口中移出，若发送队列非空，则继续发送直至队列空或窗口填满；3) 接收某帧的 NACK，说明接收端发现该帧出错，则在窗口中找到出错帧并重新发送；4) 当 ACK 超过阈值时间未到，则发送端默认超时，重新发送窗口内的所有帧，并等待 ACK。上述情况均满足时，选择重传协议可以正常运行。

本次作业下，发端系统通过以下函数实现选择重传协议。

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

当发端系统需要发送时，和接收到 ACK 时，采取的动作同第III-A节。

当发端系统接收到 NACK 时，调用函数，并设置 messageType 参数为 MSG_TYPE_RECEIVE。函数接收 NACK，从发送窗口中找到出错的帧并将其重新发送。

当发端系统超时时，调用函数，并设置 messageType 参数为 MSG_TYPE_TIMEOUT。函数接收超时，将最近发送过的帧重新发送。

B. 实现

帧结构和帧类型，以及等待队列和发送窗口同第IV-B节。

根据调用参数 messageType 的不同分情况讨论：1) 为 MSG_TYPE_SEND 时，首先将 pBuffer 转换为帧格式，存入队尾，若窗口中元素个数大于等于窗口大小，则等待；否则可以发送，将队首帧发送。2) 为 MSG_TYPE_RECEIVE，且帧类型为 ACK 时，将窗口内帧逐个弹出，直至弹出帧号对应 ACK 的帧，若等待队列非空，则按队列顺序发送，并将其移入窗口中，直至队列空或窗口满。3) 为 MSG_TYPE_RECEIVE，且帧类型为 NACK 时，遍历窗口寻找出错帧，并将其重新发送，若等待队列非空，则按队列顺序发送，并将其移入窗口中，直至队列空或窗口满。4) 为 MSG_TYPE_TIMEOUT 时，某一帧出现超时，遍历窗口寻找超时帧，并将其重新发送。

程序的 C/C++ 风格的伪代码如下所示。

```
int stud_slide_window_choice_frame_resend(char *pBuffer,
                                          int bufferSize,
                                          UINT8 messageType)
{
    switch (messageType)
    {
    case MSG_TYPE_SEND:
        // 复制pBuffer
        // 将帧加入队尾
        if (窗口内帧数 < 窗口大小)
```

```
        {
            // 发送队首的帧
            // 将该帧从队首移至窗口尾
        }
        break;
    case MSG_TYPE_RECEIVE:
        // 复制pBuffer
        if (ACK)
        {
            /* 将窗口中ACK帧号前的等待确认的帧清空 */
            while(窗口非空 && 窗口首帧号 != ACK帧号)
                // 弹出窗口首帧
            if (窗口非空)
                // 弹出窗口首帧
        }
        else if (NACK)
        {
            /* 寻找窗口中出错帧 */
            for (iter = 队首帧; iter != 队尾帧; iter++)
                if (iter帧号 == 出错帧号)
                {
                    // 重新发送iter帧
                    break;
                }
        }
        else
            // 程序运行时，不会到达此处
            pass;
        /* 填充窗口 */
        while (窗口内帧数 < 窗口大小 && 队列非空)
        {
            // 发送队首的帧
            // 将该帧从队首移至窗口尾
        }
        break;
    case MSG_TYPE_TIMEOUT:
        /* 寻找窗口中超时帧 */
        for (iter = 窗口首帧; iter != 窗口尾帧; iter++)
            if (iter帧号 == 超时帧号)
            {
                // 重新发送iter帧
                break;
            }
        break;
    default:
        // 程序运行时，不会到达此处
        break;
    }
}
```

V. 代码

本次作业详细代码请见附录部分。各部分均在注释中进行详细说明和解释。

附录

```
#include "sysinclude.h"
#include<queue>
using namespace std;
extern void SendFRAMEPacket(unsigned char* pData, unsigned int len);

#define WINDOW_SIZE_STOP_WAIT 1
#define WINDOW_SIZE_BACK_N_FRAME 4

// Suggested in the handbook of NetRiver
typedef enum {data,ack,nak} frame_kind;
typedef struct frame_head
{
    frame_kind kind;
    unsigned int seq;
    unsigned int ack;
    unsigned char data[100];
};
typedef struct frame
{
    frame_head head;
    unsigned int size;
};

// Queue and Window
queue<struct frame> sendList;
deque<struct frame> sendWindow;

/*
* Stop-and-Wait Protocol
*/
int stud_slide_window_stop_and_wait(char *pBuffer, int bufferSize, UINT8 messageType)
{
    static bool send = true;    // A flag to tag if it is available to send
    struct frame f;

    switch(messageType)
    {
        // To send a new frame
        case MSG_TYPE_SEND:
            // Push the frame into the queue
            memcpy(&f,pBuffer,sizeof(f));
            f.size = bufferSize;
            sendList.push(f);
            // If it is available to send
            if(send)
            {
                // Send the first frame in the queue.
                // (There is at least one frame in the queue.)
                f=sendList.front();
                SendFRAMEPacket((unsigned char*)(&f),f.size);
                // Because of the Stop&Wait protocol, after sending one frame,
                // the sender cannot send any new frames, until it receives an ACK.
                send = false;
            }
            break;
        // To receive an ACK
        case MSG_TYPE_RECEIVE:
```

```
            // The first frame is received. Pop it.
            sendList.pop();
            // Sending is available.
            send = true;
            // If there are still frames in the queue, continue to send.
            if(!sendList.empty())
            {
                f=sendList.front();
                SendFRAMEPacket((unsigned char*)(&f),f.size);
                send = false;
            }
            break;
        // To handle a timeout exception
        case MSG_TYPE_TIMEOUT:
            // Send the first frame in the queue again.
            f = sendList.front();
            SendFRAMEPacket((unsigned char*)(&f),f.size);
            send = false;
            break;
        // Shouldn't come to default!
        default: break;
    }
    return 0;
}

/*
* Back-N Protocol
*/
int stud_slide_window_back_n_frame(char *pBuffer, int bufferSize, UINT8 messageType)
{
    // Because in BackN protocol the sender sends all frames left
    // in the window, the send flag is not necessary anymore.
    struct frame f;

    switch(messageType)
    {
        // To send a new frame
        case MSG_TYPE_SEND:
            // Push the frame into the queue.
            memcpy(&f,pBuffer,sizeof(f));
            f.size = bufferSize;
            sendList.push(f);
            // If there are still spare places in the window,
            // send this frame and push it into the window.
            if(sendWindow.size() < WINDOW_SIZE_BACK_N_FRAME)
            {
                f = sendList.front();
                sendWindow.push_back(f);
                SendFRAMEPacket((unsigned char*)(&f),f.size);
                // The frame waits for its ACK in the window.
                sendList.pop();
            }
            break;
        // To receive an ACK
        case MSG_TYPE_RECEIVE:
            memcpy(&f,pBuffer,sizeof(f));
            // Pop all frames in the window, whose number is
            // lower than or equal to the ACK.
            while(!sendWindow.empty() && ntohl(sendWindow.begin()->head.seq) != ntohl(f.head.ack))
```

```
                // Big endien - Small endien
                sendWindow.pop_front();
            if (!sendWindow.empty())
                sendWindow.pop_front();
            // Fill the window with frames in the waiting list,
            // and send all these frames
            while(sendWindow.size()<WINDOW_SIZE_BACK_N_FRAME && !sendList.empty())
            {
                f=sendList.front();
                sendWindow.push_back(f);
                SendFRAMEPacket((unsigned char*)(&f),f.size);
                sendList.pop();
            }
            break;
        // To handle a timeout exception
        case MSG_TYPE_TIMEOUT:
            // Resend all frames
            for(deque<struct frame>::iterator iter = sendWindow.begin(); iter != sendWindow.end(); ++iter)
                SendFRAMEPacket((unsigned char*)&(*iter),iter->size);
            break;
        // Shouldn't come to default!
        default: break;
    }
    return 0;
}

/*
* Selective Repeat
*/
int stud_slide_window_choice_frame_resend(char *pBuffer, int bufferSize, UINT8 messageType)
{
    struct frame f;

    switch(messageType)
    {
        // To send a frame
        case MSG_TYPE_SEND:
            // Push the frame into the queue.
            memcpy(&f,pBuffer,sizeof(f));
            f.size=bufferSize;
            sendList.push(f);
            // If there are still spare places in the window,
            // send this frame and push it into the window.
            if(sendWindow.size() < WINDOW_SIZE_BACK_N_FRAME)
            {
                f=sendList.front();
                sendWindow.push_back(f);
                SendFRAMEPacket((unsigned char*)(&f),f.size);
                sendList.pop();
            }
            break;
        // To receive an ACK/NACK
        case MSG_TYPE_RECEIVE:
            memcpy(&f,pBuffer,sizeof(f));
            // If it is an ACK...
            if(ntohl(f.head.kind) == ack)
            {
                // Pop all frames in the window, whose number is
                // lower than or equal to the ACK.
```

```
                while(!sendWindow.empty() && ntohl(sendWindow.begin()->head.seq) != ntohl(f.head.ack))
                    // Big endien - Small endien
                    sendWindow.pop_front();
                if (!sendWindow.empty())
                    sendWindow.pop_front();
            }
            // Otherwise, it is an NACK
            else if(ntohl(f.head.kind) == nak)
                // Find the wrong frame
                for(deque<struct frame>::iterator iter = sendWindow.begin(); iter != sendWindow.end(); ++iter)
                    if(ntohl(f.head.ack)==ntohl(iter->head.seq))
                    {
                        SendFRAMEPacket((unsigned char*)&(*iter),iter->size);
                        break;
                    }
            // Fill the window with frames in the waiting list,
            // and send all these frames.
            while(sendWindow.size()<WINDOW_SIZE_BACK_N_FRAME && !sendList.empty())
            {
                    f=sendList.front();
                    sendWindow.push_back(f);
                    sendList.pop();
                    SendFRAMEPacket((unsigned char*)(&f),f.size);
            }
            break;
        // To handle a timeout exception
        case MSG_TYPE_TIMEOUT:
            // Wrong frame sequence number
            unsigned int seq;
            memcpy(&seq,pBuffer,sizeof(seq));
            // Find the wrong frame and resend it.
            for(deque<struct frame>::iterator iter = sendWindow.begin(); iter != sendWindow.end(); ++iter)
                if(ntohl(seq) == ntohl(iter->head.seq))
                {
                    SendFRAMEPacket((unsigned char*)(&(*iter)),iter->size);
                    break;
                }
            break;
        // Shouldn't come to default!
        default: break;
    }
    return 0;
}
```