滑动窗口协议实验报告

2013011371 计 34 沈哲言

一、 实验内容

（1） 熟悉 NetRiver 实验系统，学会在实验系统上进行编辑、编译、运行和调试

（2） 掌握滑动窗口协议，在理解协议的基础上利用 C/C++语言实现 1bit 滑动窗口协议和退后 n 帧协议的发送方部分，使得能够响应系统的发送请求、接收帧消息和超时消息

二、 实验原理

– 滑动窗口协议（Sliding Window Protocol）工作原理：

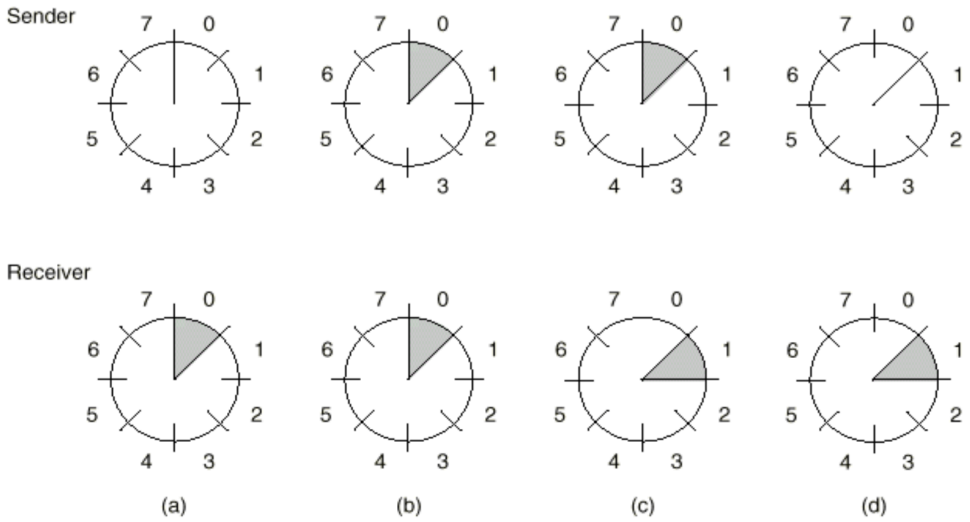
- 发送的信息帧都有一个序号，从0到某个最大值，$0 \sim 2^n - 1$，一般用n个二进制位表示；
- 发送端始终保持一个已发送但尚未确认的帧的序号表，称为发送窗口。发送窗口的上界表示要发送的下一个帧的序号，下界表示未得到确认的帧的最小编号。发送窗口 = 上界 - 下界，大小可变；
- 发送端每发送一个帧，序号取上界值，上界加1；每接收到一个正确响应帧，下界加1；
- 接收端有一个接收窗口，大小固定，但不一定与发送窗口相同。接收窗口的上界表示允许接收的序号最大的帧，下界表示希望接收的帧；
- 接收窗口表示允许接收的信息帧，落在窗口外的帧均被丢弃。序号等于下界的帧被正确接收，并产生一个响应帧，下界加1。接收窗口大小不变。

这是基础的滑动窗口协议的原理，本实验要求实现的 1bit 重传和退后 n 帧协议都是在基本的协议上的扩展，下面分别叙述

（1）1bit 滑动窗口协议

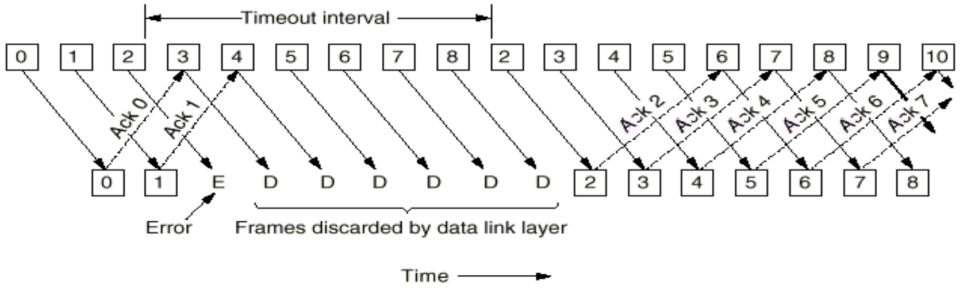
这个协议其实是单工停等协议的双工版本，允许双向发送，但发送方还是只能一次发一帧，等待确认之后再发下一帧，发送方窗口大小为

1，即没有接到当前发送帧的响应时不能发下一帧，接收方窗口大小也为 1，即只能接收期待的帧，其余一律丢弃，下图为原理图



（2）退后 n 帧协议

接收方的窗口大小依然是 1，区别在于发送方窗口大小调整为大于 1 的常数 n（本实验中为 4），也即发送端可以在未收到响应帧数目小于 n 的时候连续发送，并未这些帧单独设置计时器和缓冲区，一旦有一个帧发送失败，从这个帧起的之后所有帧都要重发，原理图如下



三、 实现思路

首先设置全局缓冲区和发送端上下界（即待发送帧编号和最早的未确认帧编号），然后分别响应不同的消息类型

（1）　发送请求

先将帧存入缓冲区，让后判断当前帧是否在发送窗口内，如果是就发送否则返回

（2）　接收帧消息

判断确认帧号是否和待确认帧好一致，如果一致更新待确认帧号，并且将缓存的帧在窗口大小允许的条件下发出

（3）　接收超时消息

从待确认帧起重发帧（1bit 重发 1 帧，退后 n 帧重发超时帧后所有帧）

四、 源代码

```
/*
* 1 bit protocol variable declaration
*/
unsigned int next_frame_to_send = 0;
unsigned int frame_expect_to_ack = 0;
char buffer[256][256];
unsigned int buffer_size[256];
unsigned int buffer_seq[256];
/*
* 停等协议测试函数
*/
int stud_slide_window_stop_and_wait(char *pBuffer, int bufferSize, UINT8 messageType)
{
    switch(messageType){
        case MSG_TYPE_TIMEOUT:{
            SendFRAMEPacket((unsigned char*)buffer[frame_expect_to_ack], buffer_size[frame_expect_to_ack]); //如果超
            return 0;
        }
        case MSG_TYPE_SEND:{
            memcpy(buffer[next_frame_to_send], pBuffer, bufferSize);
            buffer_size[next_frame_to_send] = bufferSize;
            buffer_seq[next_frame_to_send] = ((frame*)pBuffer)->head.seq;
            if (next_frame_to_send == frame_expect_to_ack) SendFRAMEPacket((unsigned char*)pBuffer, bufferSize);
            next_frame_to_send++;
            return 0;
        }
        case MSG_TYPE_RECEIVE:{
            if (((frame*)pBuffer)->head.ack == buffer_seq[frame_expect_to_ack]){
                frame_expect_to_ack++;
                if (frame_expect_to_ack < next_frame_to_send)
                    SendFRAMEPacket((unsigned char*)buffer[frame_expect_to_ack], buffer_size[frame_expect_to_ack]);
                return 0;
            }
        }
    }
    return -1;
}
```

```c
/*
* n bit protocol variable declaration
*/
unsigned int buffered_upperbound = 0;
unsigned int n_next_frame_to_send = 0;
unsigned int n_frame_expect_to_ack = 0;
char nBuffer[256][256];
unsigned int n_buffer_size[256];
unsigned int n_buffer_seq[256];
/*
* 回退n帧测试函数
*/
int stud_slide_window_back_n_frame(char *pBuffer, int bufferSize, UINT8 messageType)
{
    switch(messageType){
        case MSG_TYPE_TIMEOUT:{
            for (int i=n_frame_expect_to_ack; i<n_next_frame_to_send; i++)
                SendFRAMEPacket((unsigned char*)nBuffer[i], n_buffer_size[i]);
            return 0;
        }
        case MSG_TYPE_SEND:{
            memcpy(nBuffer[buffered_upperbound], pBuffer, bufferSize);
            n_buffer_size[buffered_upperbound] = bufferSize;
            n_buffer_seq[buffered_upperbound] = ntohl(((frame*)pBuffer)->head.seq);
            buffered_upperbound++;
            while (n_next_frame_to_send < n_frame_expect_to_ack + WINDOW_SIZE_BACK_N_FRAME && n_next_frame_to_send < buffered_upperbound){
                SendFRAMEPacket((unsigned char*)nBuffer[n_next_frame_to_send], n_buffer_size[n_next_frame_to_send]);
                n_next_frame_to_send++;
            }
            return 0;
        }
        case MSG_TYPE_RECEIVE:{
            while (ntohl(((frame*)pBuffer)->head.ack) >= n_buffer_seq[n_frame_expect_to_ack])
                n_frame_expect_to_ack++;
            while (n_next_frame_to_send < n_frame_expect_to_ack + WINDOW_SIZE_BACK_N_FRAME && n_next_frame_to_send < buffered_upperbound){
                    SendFRAMEPacket((unsigned char*)nBuffer[n_next_frame_to_send], n_buffer_size[n_next_frame_to_send]);
                    n_next_frame_to_send++;
            }
            return 0;
        }
    }
    return -1;
}
```

五、 思考题

（1）1bit 重传协议本质上基于停等的方式，双方同时开始的话有一半的重复帧，传输时间长，退后 n 帧协议大大提高的信道的利用率，一次可以发送 n 帧，可以充分利用高带宽的优势同时部分弥补延迟大的劣势，这个结论可以从信道利用率公式上看出

- 一般情况
  信道带宽b比特/秒，帧长度l比特，往返传输延迟R秒，则信道利用率为 $(l/b) / (l/b + R) = l / (l + Rb)$

这是 1bit 的情况，nbit 的话上下的 l 同时乘以 n 可以提高利用率

（2）退后 n 帧重传要设置比较大的发送端缓冲区，以便在重传的时候能够顺利进行，这要消耗一定的内存空间；其次，退后 n 帧重传可能会重发已经正确发送的帧，假设每一帧的出错概率为 p，n 帧中需要重发某些帧的概率为 1-（1-p）^n，在 n 很大时这个概率是很高

的，故如果信道误码率高的情况下，很大的 n 将导致不断重发，这显然是对带宽的浪费