滑动窗口协议实验报告

黄道吉

2020 年 3 月 16 日

滑动窗口协议是链路层数据传输的重要协议。这次实验要求在模拟的数据链路层环境中，实现下列三个协议中发送端的功能

- 停等协议
- 回退N滑动窗口协议
- 选择重传协议

1 数据结构

除了提供的帧和帧头之外，我们另外维护两个数据结构分别用来存储 1)未发送的数据和 2)已经发送(可能需要重发的数据)。其中未发送的数据只需要能够保证先进先出，用queue实现，已发送的数据在全部重发时需要一一迭代访问，最好实现成deque。两个数据结构的用法在协议实现部分有详细的说明。

2 协议实现

2.1 停等协议

按照停等协议的要求，每次只能发送一个帧，并保存刚刚发送的帧。因此

- 当遇到超时信号时，重传刚刚发送的帧
- 当遇到发送信号时，将需要发送的帧入队。如果已发送的队列为空，即是可以发送，则从待发送队列头取一帧发送。
- 当收到接受信号时，清空已发送队列。如果待发送队列不空，则可以再发送一帧。

2.2 回退N滑动窗口协议

按照回退N滑动窗口协议的要求，维护一个最大长度为N的已发送队列。因此

- 当遇到超时信号时，重传所有已发送的帧
- 当遇到发送信号时，将需要发送的帧入队。如果已发送的队列长度小于N，即是可以发送，则从待发送队列头取一帧发送，直到已发送N帧。
- 当收到接受信号时，清空序号不等于接受帧序号的帧。如果待发送队列不空，则可以再发送帧，直到已发送N帧。

2.3 选择重传协议

按照选择重传协议的要求，维护一个最大长度为N的已发送队列。因此

- 当遇到超时信号时，重传需要重传的帧
- 当遇到发送信号时，将需要发送的帧入队。如果已发送的队列长度小于N，即是可以发送，则从待发送队列头取一帧发送，直到已发送N帧。
- 当收到接受信号时，如果是成功接收，则清空序号不等于接受帧序号的帧。如果待发送队列不空，则可以再发送帧，直到已发送N帧；如果是未成功接受，按照上课说的解决方法，重传所有帧

```
#include <queue>

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
};

queue<frame> frames_to_send;
deque<frame> frames_sent;

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

    switch(messageType){
        case MSG_TYPE_TIMEOUT:{
            frame f = frames_sent.front();
            SendFRAMEPacket((unsigned char *)&f, f.size);
            break;
        }
        case MSG_TYPE_SEND:{
            frame f;
            memcpy(&f, pBuffer, sizeof(frame));
            f.size = bufferSize;
            frames_to_send.push(f);
            if(frames_sent.empty()){ // can send more frames!
                f = frames_to_send.front();
                frames_to_send.pop();
                frames_sent.push_back(f);
                SendFRAMEPacket((unsigned char *)&f, f.size);
            }
            break;
        }
        case MSG_TYPE_RECEIVE:{
            frames_sent.pop_front();
            if(frames_to_send.size()){
                frame f = frames_to_send.front();
```

```
            frames_to_send.pop();
            frames_sent.push_back(f);
            SendFRAMEPacket((unsigned char *)&f, f.size);
        }
        break;
    }
    default:{
        return -1;
    }
  }
  return 0;
}

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

  switch(messageType){
    case MSG_TYPE_TIMEOUT:{
        for(deque<struct frame>::iterator it = frames_sent.begin();
            it != frames_sent.end();
            it++){
                SendFRAMEPacket((unsigned char *)&(*it), (*it).size);
        }
        break;
    }
    case MSG_TYPE_SEND:{
        frame f;
        memcpy(&f, pBuffer, sizeof(frame));
        f.size = bufferSize;
        frames_to_send.push(f);
        while(frames_to_send.size() && frames_sent.size() < n){ // can send more
frames...but what is n?
            f = frames_to_send.front();
            frames_to_send.pop();
            frames_sent.push_back(f);
            SendFRAMEPacket((unsigned char *)&f, f.size);
        }
        break;
    }
    case MSG_TYPE_RECEIVE:{
        frame f;
        memcpy(&f, pBuffer, sizeof(frame));
        while(frames_sent.front().head.seq != f.head.ack && frames_sent.size()){
            frames_sent.pop_front();
        }
```

```
            frames_sent.pop_front();
            while(frames_to_send.size() && frames_sent.size() < n){
                f = frames_to_send.front();
                frames_to_send.pop();
                frames_sent.push_back(f);
                SendFRAMEPacket((unsigned char *)&f, f.size);
            }
        }
        default:{
            return -1;
        }
    }
    return 0;
}

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

    switch(messageType){
        case MSG_TYPE_TIMEOUT:{
            unsigned int seq; //序列号
            memcpy(&seq, pBuffer, sizeof(seq));
            for(deque<struct frame>::iterator it = frames_sent.begin();
                it != frames_sent.end();
                it++){
                if(seq == (*it).head.seq){
                    SendFRAMEPacket((unsigned char *)&(*it), (*it).size);
                    break;
                }
            }
            break;
        }
        case MSG_TYPE_SEND:{
            frame f;
            memcpy(&f, pBuffer, sizeof(frame));
            f.size = bufferSize;
            frames_to_send.push(f);
            while(frames_to_send.size() && frames_sent.size() < n){ // can send more
frames...but what is n?
                f = frames_to_send.front();
                frames_to_send.pop();
                frames_sent.push_back(f);
                SendFRAMEPacket((unsigned char *)&f, f.size);
            }
            break;
```

```
        }
        case MSG_TYPE_RECEIVE:{
            frame f;
            memcpy(&f, pBuffer, sizeof(frame));

            if(ntohl(f.head.kind) == ack){
                // delete acknowledged frames
                while(frames_sent.front().head.seq != f.head.ack && frames_sent.size()){
                    frames_sent.pop_front();
                }
                frames_sent.pop_front();
                // send new ones
                while(frames_to_send.size() && frames_sent.size() < n){
                    f = frames_to_send.front();
                    frames_to_send.pop();
                    frames_sent.push_back(f);
                    SendFRAMEPacket((unsigned char *)&f, f.size);
                }
            }
            if(ntohl(f.head.kind) == nak){
                for(deque<struct frame>::iterator it = frames_sent.begin();
                    it != frames_sent.end();
                    it++){
                    if(f.head.ack == (*it).head.seq){
                        SendFRAMEPacket((unsigned char *)&(*it), (*it).size);
                        break;
                    }
                }
            }

        }
        default:{
            return -1;
        }
    }
    return 0;
}
```