Zixiao Zhao <ziz093@eng.ucsd.edu> A53067425

# Sliding Window Protocol Project

Project 1 report for CSE 123 Computer Networks

## Frame Format

### The frame type defined in comm.h:

```
struct Frame_t
{
    SwpSeqNo swpSeqNo; // 1 byte
    uint16_t send_id; // 2 bytes
    uint16_t recv_id; // 2 bytes
    unsigned char flag[FRAME_FLAG_SIZE]; // 3 bytes
    char data[FRAME_PAYLOAD_SIZE]; // 52 bytes
    Parity parity; // 4 bytes
};
```

A frame contains 64 bytes data in total and 12 bytes are for header and parity and 52 bytes are for payload.
swpSeqNo is a 8-bit unsigned int that denotes the sequence number in sliding window protocol.
send_id and recv_id denote the id of the sender and the target receiver.
The first bit is used to indicates the massage has subsequent massages. (for long message transimission)
52-byte data is to store the payload.
4-byte parity is to store CRC32 code.

## Sliding Windows Protocol Implement

### Sender

### Related Variables:

```
SwpSeqNo LastAckReceived;
SwpSeqNo LastFrameSent;
uint8_t SwpWindow;
Frame framesInWindow[SWP_WINDOW_SIZE];
SwpSeqNo lastAckNo;
struct timeval framesInWindowTimestamp[SWP_WINDOW_SIZE];
```

SwpWindow is 8-bit and each bit denotes if a frame in the SWP window in sent. (1 for sent and 0 for unsent)
framesInWindow array stores each frame sent in the SWP window for retransmission.
framesInWindowTimestamp array stores the time value of each frame sent in the SWP for calculation of the wait time for the next retransmission.

### Handling messages sending:

Construct frame for each message, then LastFrameSent(LFS) plus one and update the SWP windows status. Continue sending messages until LastAckReceived(LAR) – LFS > 8.

### Handling incoming acks:

If the sequence number of the incoming ack equals LAR + 1, then SWP windows move forward to the new LAR. If not equals to LAR + 1, but still in the SWP window, update the SWP status. If the sequence > LAR + 8, drop the ack because it beyond the window.

## Receiver

### Related Variables:

```
SwpSeqNo LastFrameReceived;
uint8_t SwpWindow;
Frame framesInWindow[SWP_WINDOW_SIZE - 1];
```

SwpWindow is 8-bit and each bit denotes if a frame in the SWP window in received. (1 for sent and 0 for unsent), only the first 7 bits is needed because it is no need to store the first following message of the last message received. (When it comes, the window moves immediately) framesInWindow array stores each frame sent in the SWP window for retransmission. Also, the first following message is excluded.

### Handling incoming messages:

If the sequence number of the incoming ack equals LastFrameReceived(LFR) + 1, then SWP windows move forward to the new LAR. If not equals to LAR + 1, but still in the SWP window, update the SWP status and store the frame ahead of time in the framesInWindow array. If the sequence > LAR + 7, drop the frame because it beyond the window.

# Retransmission

### Calculate the wait time of next retransmission:

Go over the SWP window and find the earliest timestamp of sent frames, so the next timeout should be triggered at the earliest timestamp + 100000 usec.

### Retransmit:

When time out, go over the SWP window again and check if the timestamp + 100000 usec equal or less than the current time value, if yes, retransmit the frame and update the timestamp.

# Sequence Number Warp Around

I use the following function to compare the SeqSwpNo(LAR, LFS, LFR), it works correctly when the 8-bit SeqSwpNo overflows.

```
uint8_t SwpSeqNo_minus(SwpSeqNo a, SwpSeqNo b)
{
    if(a >= b)
        return a - b;
    else
        return 255 - b + a + 1;
}
```

## Long Messages Handling

When the sender handling incoming commands, if the message to send it longer than the size of payload of frame, truncate the first 52 bytes (my payload size) and put the remaining message to the head of incoming commands using the following function.

```
void ll_append_node_toFirst(LLnode ** head_ptr,
                    void * value)
{
    ll_append_node(head_ptr, value);
    *head_ptr = ((*head_ptr)->prev);
}
```

## Many-to-Many Transmission

### Data Structures:

```
struct SenderSwpState_t
{
    SwpSeqNo LastAckReceived;
    SwpSeqNo LastFrameSent;
    uint8_t SwpWindow;
    Frame framesInWindow[SWP_WINDOW_SIZE];
    SwpSeqNo lastAckNo;
    int lastAckNoDuplicateTimes;
    struct timeval framesInWindowTimestamp[SWP_WINDOW_SIZE];
};

struct ReceiverSwpState_t
{
    SwpSeqNo LastFrameReceived;
    uint8_t SwpWindow;
    Frame framesInWindow[SWP_WINDOW_SIZE - 1];
    uint8_t preMsgHasSubsequent;
    int longMsgBufferSize;
    char *longMsgBuffer;
};
```

### Switch states:

When the current target sender or target receiver changes, the sender/receiver switch the states to ensure SWP protocol works properly.

### Retransmission:

To support many-to-many communication, the new retransmit function and wait time calculation need to go over all the the current SWP status as well SWP status in all saved states to retransmit frame for all the receivers

## Optimization

### CRC of ACK Messages

As the ack messages contains no payload, it's no need to go over the whole frame to calculate the CRC value. In this way, the corruption rate of the ack message decreased. (The corruption may occur at the payload part, it would not affect the ack message, do not drop them)

```
void ackFrameAddCRC32(Frame *frame)
{
        char* char_buf = convert_frame_to_char(frame);
        frame->parity = crc32(char_buf, sizeof(SwpSeqNo) + sizeof(uint16_t) *
2);
        free(char_buf);
}

uint8_t ackFrameIsCorrupted(Frame *frame)
{
        char* char_buf = convert_frame_to_char(frame);
        if(frame->parity == crc32(char_buf, sizeof(SwpSeqNo) +
sizeof(uint16_t) * 2))
        {
                return 0;
        }
        else
        {
                return 1;
        }
}
```

### Fast Retransmission

If the sender receive two continues same ack messages, receiver definitely missed some message. Retransmit immediately.