Ryan Bridges, A09393354
10/25/2015
CSE 123, Snoeren

PA1 Design Document

Frame struct

```
#define MAX_FRAME_SIZE 64
//TODO: You should change this!
//Remember, your frame can be AT MOST 64 bytes!
#define FRAME_PAYLOAD_SIZE 55
struct Frame_t {
    uchar_t SeqNum;              // 1 byte
    uint16_t recv_id;            // 2 bytes
    uint16_t send_id;            // 2 bytes
    char data[FRAME_PAYLOAD_SIZE]; // 58 bytes
    uint32_t crc;                // 4 bytes
};
typedef struct Frame_t Frame;
```

All that was added to the frame struct was a sequence number denoting the sequence number associated with the frame, and a 4 byte field which will hold the crc remainder.

receiver.c

receiver members:

```
//Receiver and sender data structures
struct Receiver_t {
    //DO NOT CHANGE:
    // 1) buffer_mutex
    // 2) buffer_cv
    // 3) input_framelist_head
    // 4) recv_id
    pthread_mutex_t buffer_mutex;
    pthread_cond_t buffer_cv;
    LLnode *input_framelist_head;
    uchar_t last_frame_received; // last ack sent out by the receiver
    uchar_t largest_acceptable_frame; // The highest SeqNum we can receive while staying in the window
    Frame **frame_buffer;
    uchar_t sliding_window;
    int recv_id;
};
```

I added 4 members to the Receiver struct. last_frame_received and largest_acceptable_frame hold the last sequence number that was received and the sequence number of the largest acceptable frame respectively. frame_buffer acts as a cache for all frames that are currently in the window that have been received. sliding_window is a set of 8 bits that act as markers for the frames that have been received and are currently in the window. So for example, last_frame_received = 255 largest_acceptable_frame = 7 and sliding_window = 00110101, then we know that we have received frames 2, 3, 5, and 7 and that those frames would be in their respective places in frame_buffer. recv_id denotes the id of the receiver.

void handle_incoming_msgs walkthrough:

This method begins by making sure that the incoming frame is for this receiver. It then calls a method defined in util.c to recalculate the crc and ensure that the frame was not corrupted. Then, it checks to make sure that the incoming frame is within LFR and LAF. If it is not, that means that the frame timed out at the sender, and the receiver is now receiving the frame again. In this case, the receiver simply sends back an ack to the sender.

Next, we check to see if the incoming frame's sequence number is equal to LFR+1. If this is the case, then we have received a frame in order, so we print it out and shift our window. The method then checks to see if any there are frames that arrived out of order and were waiting for the frame that just came in before printing their messages. It does this by shifting the sliding

window to the left and checking if the leftmost frame has been received. If this is the case, then we print the message of the frame. We continue doing this until the leftmost position in the window is empty.

If the frame that was just received came in out of order, then we insert it into the frame buffer in the correct place but we do not print the message yet because the frame was out of order. We then also send an ack for the frame.

sender.c

sender members:

```
struct Sender_t {
    //DO NOT CHANGE:
    // 1) buffer_mutex
    // 2) buffer_cv
    // 3) input_cmdlist_head
    // 4) input_framelist_head
    // 5) send_id
    pthread_mutex_t buffer_mutex;
    pthread_cond_t buffer_cv;
    LLnode *input_cmdlist_head;
    LLnode *input_framelist_head;
    int send_id;
    Frame **frame_buffer;
    struct timeval **timeout_times;
    uchar_t last_frame_sent;
    uchar_t last_ack_received;
    uchar_t acks_in_window;
};
```

I added 5 members to the sender struct. last_frame_sent and last_ack_received hold the sequence number of the last frame that this receiver sent out and the last in order acknowledgement that this sender received respectively. The frame_buffer is similar to that of the Receiver. It holds the frames that are currently inside of the window. timeout_times holds the timeout times of each frame that is currently inside of the window. acks_in_window is similar to the receiver's sliding_window. It is a set of 8 bits which denote which of the frames in the window have been successfully acked by the receiver. So for example, if last_frame_sent = 10, last_ack_received = 3 and acks_in_window = 10110000, then we would know that this sender has received acks for frames 4, 6 and 7. We would also know that frames 4 thru 10 are stored in the frame buffer, with frame 4 in index 0 and frame 10 in index 6. And we would know that the timeouts corresponding to each frame in the buffer are stored in timeout_times at indices which match the frame_buffer.

*struct timeval *sender_get_next_expiring_timeval walkthrough:* This method simply iterates through timeout_times and returns the smallest timeout time in the list for a frame that has not been acked yet.

void handle_incoming_acks walkthrough:

The function begins by checking if the incoming ack was corrupted. It then checks to make sure that the ack is for this sender, and that the ack is within for a frame that is currently in the sliding window.

It will then check to see if the incoming ack is for the leftmost frame in our window. If this is the case, then we can increment LAR and shift the window over. Shifting the window over

includes shifting last_ack_received, the frame_buffer, and the timeout_buffer. Similar to the receiver, after receiving an ack on the left edge of the window, the sender will check if additional acks were received out of order that are now on the edge of the window as a result of the shift. If acks are found on the edge of the window, the sliding window is shifted over until the frame on the leftmost edge of the window has not received an ack.

If the incoming ack was not on the leftmost side of the window, then that means we are receiving an ack out of order. In this case, the ack is marked at the correct location inside of acks_in_window.

void handle_incoming_acks walkthrough:

First, this function checks to see if the sender already has 8 outstanding frames with no acks. If this is the case, we break out until an ack is received. We then ensure that the cmd is for this receiver.

Next, we check to see if the incoming message is bigger than the FRAME_PAYLOAD_SIZE. If this is the case, the method creates a new char* and allocates enough memory to fit a full payload. It copies FRAME_PAYLOAD_SIZE-1 chars from the incoming cmd into this new char*, and then appends the null character to the last spot. It then allocates memory for a new cmd, and gives it the same src_id and dst_id. It then gives this new cmd the remainder of the message from the original cmd that was not copied into the newly allocated char*, and it puts this new cmd onto the front of the input_cmdlist_head. So essentially, the function takes just enough chars to fill the buffer completely, and then puts the rest back onto the queue at the front so that it will be popped off on the next iteration and order will be maintained.

Next, memory for a new frame is allocated and the appropriate send_id, recv_id, and SeqNum are assigned. The message is also assigned. If the original message was too large, then just the portion of the message that fills the buffer will be copied into the frame. Then the crc is calculated. Then the timeout for this frame is calculated, and it is converted into a char* via convert_frame_to_char. Then the message is appended to the outgoing_frames_head_ptr.

void handle_timedout_frames walkthrough:

This function iterates through the timeout_times of frames who that are in the window and have not received an ack. If it encounters a frame that has timed out, it copies the frame out of the frame_buffer and appends it to the outgoing_frames_head_ptr.

util.c

void uchar_t wrapped_subtract walkthrough

This function is used throughout the program to calculate the difference between two 8 bit numbers with the wrap-around taken into account. For example, normally, 3-255=-252. But if we invoke wrapped_subtract(3,255), we get 4 which is what we want.

void send_ack walkthrough

This function is used by the receiver to send acks. It allocates memory for a new frame and sets all of the fields associated with the frame based on the frame that we want to send an ack for. It calls convert_frame_to_char to convert the new frame into char*, then appends it to outgoing_frames_head_ptr.

void ll_insert_node_at_front walkthrough

This function simply puts an LLnode at the **front** of a list rather than appending it to the end. This is used when a frame's message is too large. The remainder of the message is continuously put back at the front of the list until the entire message has been processed.

uint32_t crc walkthrough

This function is used to calculate the 32 bit crc remainder. It is quite small and fairly self-explanatory. One thing to note is the line:

crc = (crc >> 1) ^ (crc & 1) * CRC_POLY;

This will evaluate to (crc >> 1) if the LSB of crc is 0, and it will evaluate to (crc >> 1) ^ CRC_POLY if the LSB of the crc is 1. This eliminates the need for a conditional statement which leads to a small speed up in calculation time. I selected my crc polynomial arbitrarily from this list http://users.ece.cmu.edu/~koopman/crc/crc32.html

uint32_t append_crc walkthrough

This function takes in a frame, and converts it to char* using convert_frame_to_char. It then passes the char* **except for** the crc field of the input frame to the crc function. The output of the crc function is then stored into the incoming frame's crc field and is also returned.