Introduction:

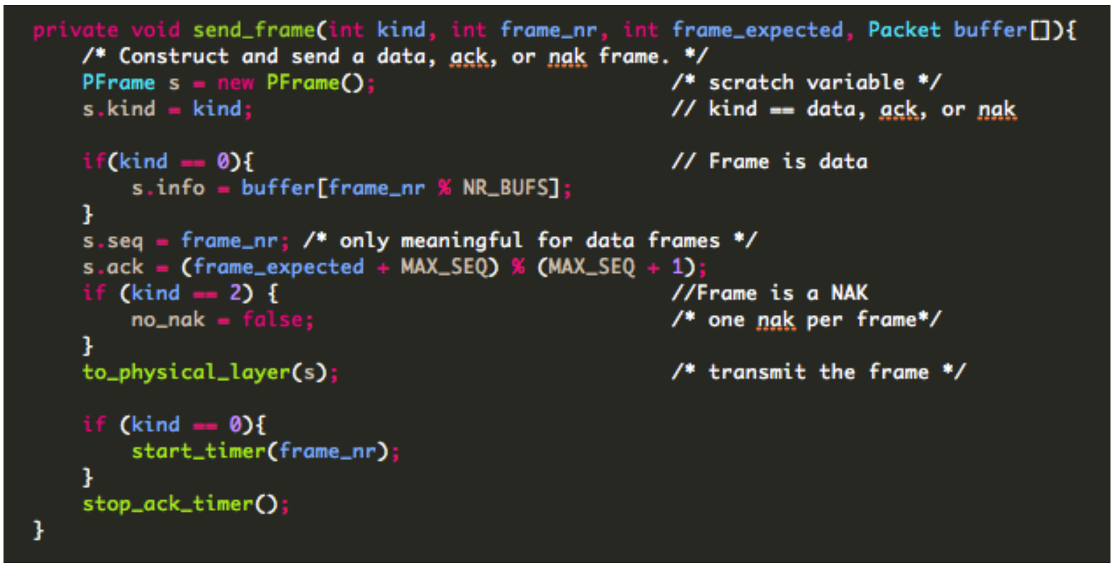
The Sliding Window Protocol is a feature of packet-based data transmission protocols. The data transmission takes place between a sender and receiver via full-duplex transmission over a single channel. The differentiating factor among the frames is using the field in the frame header to distinguish a data packet from an acknowledgement. The idea behind the sliding window protocol is that any instance of time the sender can transmit a batch of packets without waiting for acknowledgements. This batch depends on the window size of the protocol.

This report discusses the implementation of the sliding window protocol using two virtual machines that communicate with one another over a network simulator of different quality levels of service. The virtual machines act as both sender and receiver as they pass messages to each other over the network and this is used to verify the implementation. The error recovery technique used here is the selective repeat retransmission strategy.

Implementation:

All the parts of the assignment have been completed and verified by me. The implementation was tested at all four quality levels of service.

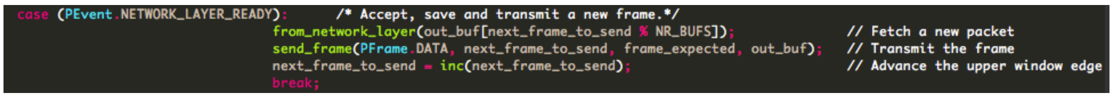
The first function implemented was the `send_frame()` function that constructs a frame so that the control data such as frame headers are abstracted from the Network layer.



```
private void send_frame(int kind, int frame_nr, int frame_expected, Packet buffer[]){
    /* Construct and send a data, ack, or nak frame. */
    PFrame s = new PFrame();                        /* scratch variable */
    s.kind = kind;                                  // kind == data, ack, or nak

    if(kind == 0){                                  // Frame is data
        s.info = buffer[frame_nr % NR_BUFS];
    }
    s.seq = frame_nr; /* only meaningful for data frames */
    s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
    if (kind == 2) {                                //Frame is a NAK
        no_nak = false;                             /* one nak per frame*/
    }
    to_physical_layer(s);                           /* transmit the frame */

    if (kind == 0){
        start_timer(frame_nr);
    }
    stop_ack_timer();
}
```

Figure 1

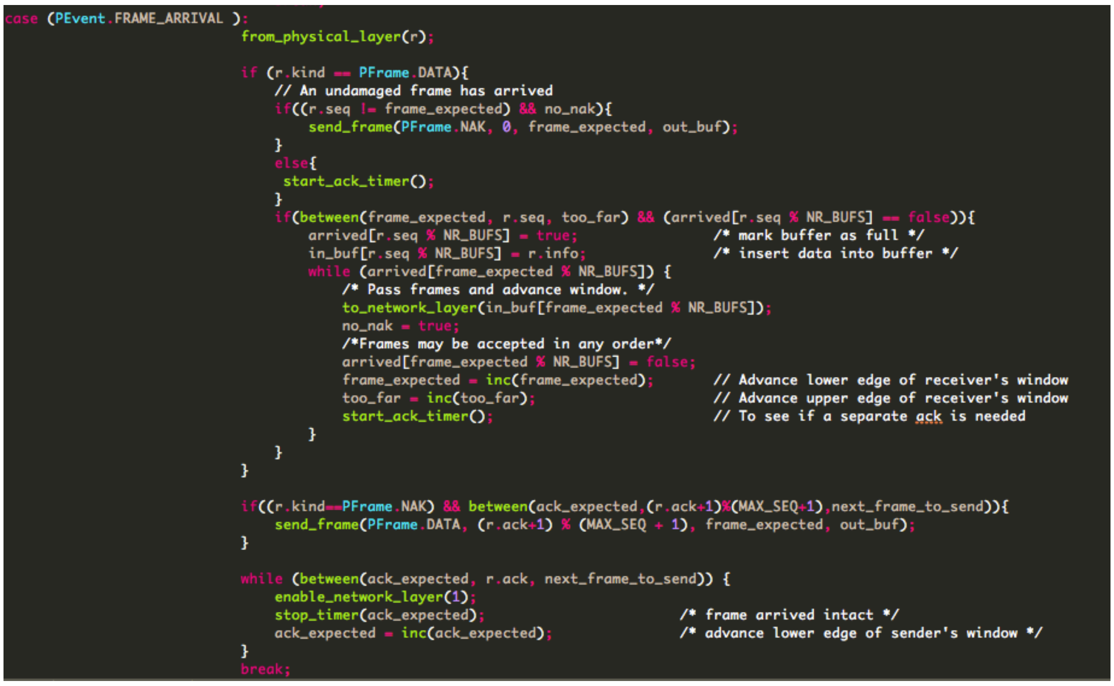
The transmission consists of a few event-handling cases. The NETWORK_LAYER_READY and the FRAME ARRIVAL blocks were straightforward implementations and the code was simply ported over from C to Java. These blocks were tested at quality level 0 for communication between the two virtual machines. The NETWORK_LAYER_READY signifies that the Network layer is ready with a packet for the Data-Link layer. This packet needs to be encapsulated in a frame with the appropriate control headers before it is sent out.



```
case (PEvent.NETWORK_LAYER_READY):      /* Accept, save and transmit a new frame.*/
                            from_network_layer(out_buf[next_frame_to_send % NR_BUFS]);          // Fetch a new packet
                            send_frame(PFrame.DATA, next_frame_to_send, frame_expected, out_buf);  // Transmit the frame
                            next_frame_to_send = inc(next_frame_to_send);                       // Advance the upper window edge
                            break;
```

Figure 2 - NETWORK_LAYER_READY

The FRAME_ARRIVAL event handles the arrival of a frame from the receiver's side. It checks an incoming frame for discrepancies and handles them appropriately. If the frame header is neither Data nor NAK, then it requests for a resend. If Data is expected and it doesn't receive it or if the received Data is duplicated, it sends out a NAK. If the Data arrives without any error in the correct form, it passes it to the higher Network layer.



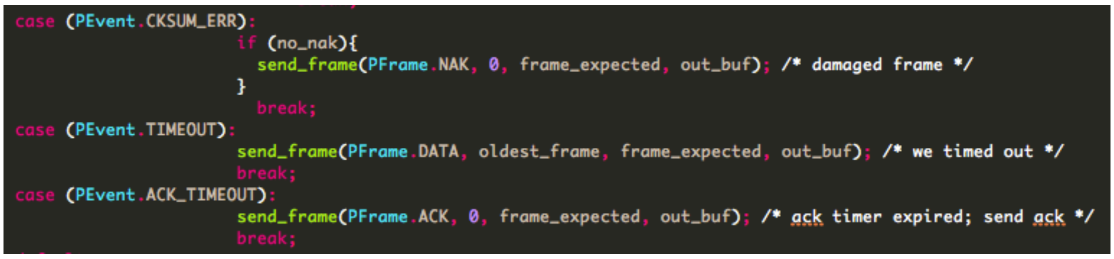
```
case (PEvent.FRAME_ARRIVAL ):
                          from_physical_layer(r);

                          if (r.kind == PFrame.DATA){
                              // An undamaged frame has arrived
                              if((r.seq != frame_expected) && no_nak){
                                  send_frame(PFrame.NAK, 0, frame_expected, out_buf);
                              }
                              else{
                               start_ack_timer();
                              }
                              if(between(frame_expected, r.seq, too_far) && (arrived[r.seq % NR_BUFS] == false)){
                                  arrived[r.seq % NR_BUFS] = true;                 /* mark buffer as full */
                                  in_buf[r.seq % NR_BUFS] = r.info;                /* insert data into buffer */
                                  while (arrived[frame_expected % NR_BUFS]) {
                                      /* Pass frames and advance window. */
                                      to_network_layer(in_buf[frame_expected % NR_BUFS]);
                                      no_nak = true;
                                      /*Frames may be accepted in any order*/
                                      arrived[frame_expected % NR_BUFS] = false;
                                      frame_expected = inc(frame_expected);        // Advance lower edge of receiver's window
                                      too_far = inc(too_far);                      // Advance upper edge of receiver's window
                                      start_ack_timer();                           // To see if a separate ack is needed
                                  }
                              }
                          }

                          if((r.kind==PFrame.NAK) && between(ack_expected,(r.ack+1)%(MAX_SEQ+1),next_frame_to_send)){
                              send_frame(PFrame.DATA, (r.ack+1) % (MAX_SEQ + 1), frame_expected, out_buf);
                          }

                          while (between(ack_expected, r.ack, next_frame_to_send)) {
                              enable_network_layer(1);
                              stop_timer(ack_expected);                       /* frame arrived intact */
                              ack_expected = inc(ack_expected);               /* advance lower edge of sender's window */
                          }
                          break;
```

Figure 3 – FRAME_ARRIVAL

The next three events implemented are used to handle abnormal events. They are CKSUM_ERR which is an event triggered by the hardware indicating that the packet has failed the checksum. In such a case, the entire frame is discarded, an NAK is out on the output buffer (as per the selective-repeat strategy) and the sender is informed to resend the frame.

An event where the timer runs out is handled by the TIMEOUT event. The timer runs while waiting for an acknowledgement and when it expires, this event is triggered. The timeout usually indicates that the packet or the acknowledgement is lost somewhere in the communication channel. In the case of such an event, the sender places the frame on the output buffer in order to resend it.

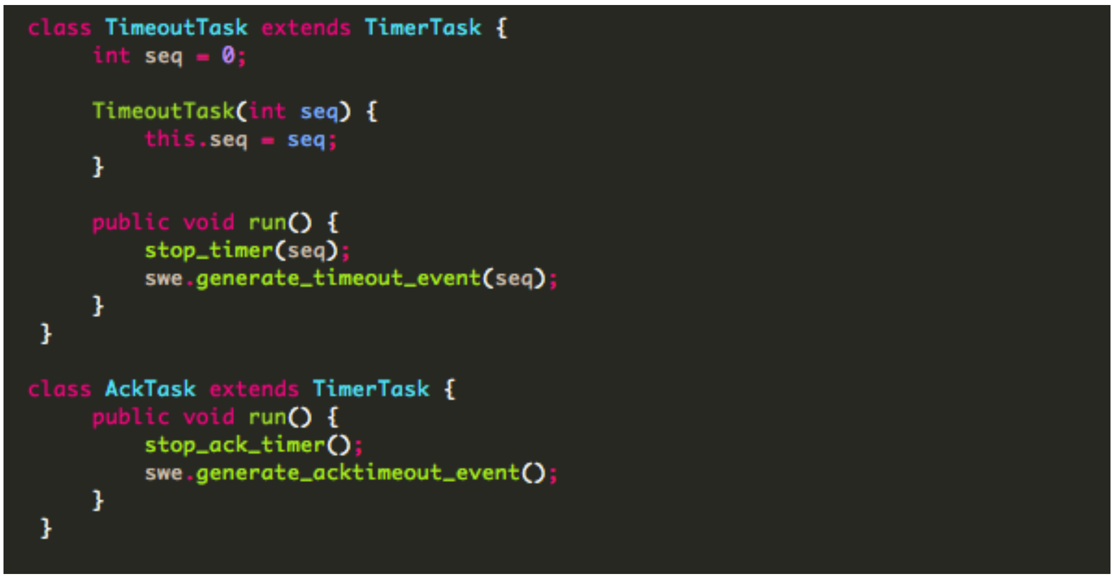
The last type of timeout is the ACK_TIMEOUT event that occurs when it takes too long for an acknowledgement to piggyback on an outgoing frame. In the case of such an event, the acknowledgement frame is sent out on its own immediately without waiting for an opportunity to piggyback.



```
case (PEvent.CKSUM_ERR):
                    if (no_nak){
                      send_frame(PFrame.NAK, 0, frame_expected, out_buf); /* damaged frame */
                    }
                      break;
case (PEvent.TIMEOUT):
                    send_frame(PFrame.DATA, oldest_frame, frame_expected, out_buf); /* we timed out */
                    break;
case (PEvent.ACK_TIMEOUT):
                    send_frame(PFrame.ACK, 0, frame_expected, out_buf); /* ack timer expired; send ack */
                    break;
```

Figure 4 - Abnormal Event Handling

To implement the timers, two classes had to devised separately using the built-in Java timer library. The function `run()` in both the cases, get executed when the timers expire. Therefore, for TimeoutTask, when the acknowledgement fails to be received, the timer stops and an event (Timeout) is generated. The AckTask is a separate timer implemented to keep track of the timeouts that arise due to failed acknowledgement receipts. By separating the timers and keeping them independent, it simplifies the process and reduces the risk for errors greatly.



```
class TimeoutTask extends TimerTask {
     int seq = 0;

     TimeoutTask(int seq) {
         this.seq = seq;
     }

     public void run() {
         stop_timer(seq);
         swe.generate_timeout_event(seq);
     }
 }

class AckTask extends TimerTask {
     public void run() {
         stop_ack_timer();
         swe.generate_acktimeout_event();
     }
 }
```

Figure 5 - Timer Class Implementation

Conclusion:

In this experiment, a sliding window protocol using selective repeat strategy was implemented. The Sliding window makes efficient use of a communication channel by permitting multiple frames to be transmitted before receiving the acknowledgement for their receipt. The Data in this experiment was read from one file and written into another. Towards the end of the experiment, the received files correctly matched the sent files.