CS7344 Homework 3

Name: Bingying Liang
ID: 48999397

Oct 5 2023

1. The following data fragment occurs in the middle of a data stream for which the byte-stuffing algorithm described in the text is used: A B ESC C ESC FLAG FLAG D. What is the output after stuffing?

   Solution: A B ESC ESC C ESC ESC ESC FLAG ESC FLAG D

2. In protocol 3 (See Figure 3-14 from the book), is it possible for the sender to start the timer when it is already running? If so, how might this occur? If not, why is it impossible?

```
/* Protocol 3 (PAR) allows unidirectional data flow over an unreliable channel. */

#define MAX_SEQ 1                       /* must be 1 for protocol 3 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"

void sender3(void)
{
  seq_nr next_frame_to_send;            /* seq number of next outgoing frame */
  frame s;                              /* scratch variable */
  packet buffer;                        /* buffer for an outbound packet */
  event_type event;

  next_frame_to_send = 0;               /* initialize outbound sequence numbers */
  from_network_layer(&buffer);          /* fetch first packet */
  while (true) {
      s.info = buffer;                  /* construct a frame for transmission */
      s.seq = next_frame_to_send;       /* insert sequence number in frame */
      to_physical_layer(&s);            /* send it on its way */
      start_timer(s.seq);               /* if answer takes too long, time out */
      wait_for_event(&event);           /* frame_arrival, cksum_err, timeout */
      if (event == frame_arrival) {
          from_physical_layer(&s);      /* get the acknowledgement */
          if (s.ack == next_frame_to_send) {
              stop_timer(s.ack);        /* turn the timer off */
              from_network_layer(&buffer); /* get the next one to send */
              inc(next_frame_to_send);  /* invert next_frame_to_send */
          }
      }
  }
}
void receiver3(void)
{
  seq_nr frame_expected;
  frame r, s;
  event_type event;

  frame_expected = 0;
  while (true) {
      wait_for_event(&event);           /* possibilities: frame_arrival, cksum_err */
      if (event == frame_arrival) {     /* a valid frame has arrived */
          from_physical_layer(&r);      /* go get the newly arrived frame */
          if (r.seq == frame_expected) { /* this is what we have been waiting for */
              to_network_layer(&r.info); /* pass the data to the network layer */
              inc(frame_expected);      /* next time expect the other sequence nr */
          }
          s.ack = 1 - frame_expected;   /* tell which frame is being acked */
          to_physical_layer(&s);        /* send acknowledgement */
      }
  }
}
```

Figure 3-14. A positive acknowledgement with retransmission protocol.

Solution: It is possible for the sender to start the timer when it is already running. It might occur in the following situation: After transmitting a frame, the sender starts the timer running, the sender waits for some thing exciting to happen. If a damaged acknowledgement frame straggers in, or the timer expires, neither the buffer nor the sequence number is changed so that a duplicate can be sent. The timer will be reset to allow another full timer interval.

3. In protocol 6 (See Figure 3-21 from the book), when a data frame arrives, a check is made to see if the sequence number differs from the one expected and no_nak is true. If both conditions hold, a NAK is sent. Otherwise, the auxiliary timer is started. Suppose that the else clause were omitted. Would this change affect the protocol's correctness?

```
/* Protocol 6 (Selective repeat) accepts frames out of order but passes packets to the
   network layer in order. Associated with each outstanding frame is a timer. When the timer
   expires, only that frame is retransmitted, not all the outstanding frames, as in protocol 5. */
#define MAX_SEQ 7                                   /* should be 2^n – 1 */
#define NR_BUFS ((MAX_SEQ + 1)/2)
typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready, ack_timeout} event_type;
#include "protocol.h"
boolean no_nak = true;                              /* no nak has been sent yet */
seq_nr oldest_frame = MAX_SEQ + 1;                  /* initial value is only for the simulator */

static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
/* Same as between in protocol 5, but shorter and more obscure. */
  return ((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a));
}

static void send_frame(frame_kind fk, seq_nr frame_nr, seq_nr frame_expected, packet buffer[])
{
/* Construct and send a data, ack, or nak frame. */
  frame s;                                          /* scratch variable */
  s.kind = fk;                                      /* kind == data, ack, or nak */
  if (fk == 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 (fk == nak) no_nak = false;                    /* one nak per frame, please */
  to_physical_layer(&s);                            /* transmit the frame */
  if (fk == data) start_timer(frame_nr % NR_BUFS);
  stop_ack_timer();                                 /* no need for separate ack frame */
}
void protocol6(void)
{
  seq_nr ack_expected;                              /* lower edge of sender's window */
  seq_nr next_frame_to_send;                        /* upper edge of sender's window + 1 */
  seq_nr frame_expected;                            /* lower edge of receiver's window */
  seq_nr too_far;                                   /* upper edge of receiver's window + 1 */
  int i;                                            /* index into buffer pool */
  frame r;                                          /* scratch variable */
  packet out_buf[NR_BUFS];                          /* buffers for the outbound stream */
  packet in_buf[NR_BUFS];                           /* buffers for the inbound stream */
  boolean arrived[NR_BUFS];                         /* inbound bit map */
  seq_nr nbuffered;                                 /* how many output buffers currently used */
  event_type event;
  enable_network_layer();                           /* initialize */
  ack_expected = 0;                                 /* next ack expected on the inbound stream */
  next_frame_to_send = 0;                           /* number of next outgoing frame */
  frame_expected = 0;
  too_far = NR_BUFS;
  nbuffered = 0;                                    /* initially no packets are buffered */
  for (i = 0; i < NR_BUFS; i++) arrived[i] = false;
  while (true) {
    wait_for_event(&event);                         /* five possibilities: see event_type above */
```

```
switch(event) {
  case network_layer_ready:                 /* accept, save, and transmit a new frame */
       nbuffered = nbuffered + 1;           /* expand the window */
       from_network_layer(&out_buf[next_frame_to_send % NR_BUFS]); /* fetch new packet */
       send_frame(data, next_frame_to_send, frame_expected, out_buf);/* transmit the frame */
       inc(next_frame_to_send);             /* advance upper window edge */
       break;

  case frame_arrival:                       /* a data or control frame has arrived */
       from_physical_layer(&r);             /* fetch incoming frame from physical layer */
       if (r.kind == data) {
            /* An undamaged frame has arrived. */
            if ((r.seq != frame_expected) && no_nak)
               send_frame(nak, 0, frame_expected, out_buf); else start_ack_timer();
            if (between(frame_expected,r.seq,too_far) && (arrived[r.seq%NR_BUFS]==false)) {
                 /* Frames may be accepted in any order. */
                 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;
                      arrived[frame_expected % NR_BUFS] = false;
                      inc(frame_expected);  /* advance lower edge of receiver's window */
                      inc(too_far);         /* advance upper edge of receiver's window */
                      start_ack_timer();    /* to see if a separate ack is needed */
                 }
            }
       }
       if((r.kind==nak) && between(ack_expected,(r.ack+1)%(MAX_SEQ+1),next_frame_to_send))
            send_frame(data, (r.ack+1) % (MAX_SEQ + 1), frame_expected, out_buf);
       while (between(ack_expected, r.ack, next_frame_to_send)) {
            nbuffered = nbuffered - 1;      /* handle piggybacked ack */
            stop_timer(ack_expected % NR_BUFS);     /* frame arrived intact */
            inc(ack_expected);              /* advance lower edge of sender's window */
       }
       break;

  case cksum_err:
       if (no_nak) send_frame(nak, 0, frame_expected, out_buf); /* damaged frame */
       break;

  case timeout:
       send_frame(data, oldest_frame, frame_expected, out_buf); /* we timed out */
       break;

  case ack_timeout:
       send_frame(ack,0,frame_expected, out_buf);      /* ack timer expired; send ack */
  }

  if (nbuffered < NR_BUFS) enable_network_layer(); else disable_network_layer();
 }
}
```

Figure 3-21. A sliding window protocol using selective repeat.

Solution: If the else clause were omitted, this change will affect the protocol's correctness. Explain: If the else clause were omitted, the code will change in the following:

```
case frame_arrival: /*a data or control frame has arrived*/
    from_physical_layer(&r); /*fetch incoming frame from physical layer*/
    if (r.kind == data) {
    /*An undamaged frame has arrived.*/
        if ((r.seq != frame_expected) && no_nak)
            send_frame(nak, 0, frame_expected, out_buf);
        // else start_ack_timer();
        if (between(frame_expected,r.seq,too_far)
        && (arrived[r.seq%NR_BUFS]==false)) {
```

```
10          /*Frames may be accepted in any order.*/
11              arrived[r.seq % NR_BUFS] = true; /*mar k buffer as full*/
12              in_buf[r.seq % NR_BUFS] = r.info; /*inser t data into buffer*/
13              while (arrived[frame_expected % NR_BUFS]) {
14              /*Pass frames and advance window.*/
15                  to_network_layer(&in_buf[frame_expected % NR_BUFS]);
16                  no_nak = true;
17                  arrived[frame_expected % NR_BUFS] = false;
18                  inc(frame_expected); /*advance lower edge of receiver's window*/
19                  inc(too_far); /*advance upper edge of receiver's window*/
20                  start_ack_timer(); /*to see if a separate ack is needed*/
21              }
22          }
23      }
24
25      if((r.kind==nak)
26      && between(ack_expected,(r.ack+1)%(MAX SEQ+1),next_frame_to_send))
27          send_frame(data, (r.ack+1) % (MAX_SEQ + 1), frame_expected, out_buf);
28          while (between(ack_expected, r.ack, next_frame_to_send)) {
29              nbuffered = nbuffered - 1; /* handle piggybacked ack */
30              stop_timer(ack_expected % NR_BUFS); /*frame arrived intact*/
31              inc(ack_expected); /*advance lower edge of sender's window*/
32          }
33      break;
```

If the frame is not expected frame and `no_nak == true`, it will send the run `send_frame(nak, 0, frame_expected, out_buf);` else has three situations:

1 `r.seq != frame_expected && no_nak == false`
2 `r.seq == frame_expected && no_nak == false`
3 `r.seq == frame_expected && no_nak == true`

It might lead to deadlock. For these three situations, the receiver will send acknowledgements to the sender. If the acknowledgements were lost. The receiver will not realize that. But the sender realize it does not get the acknowledgement, it will resend the frame again. But at this time the receiver will run `r.seq != frame_expected && no_nak==true`. Then receiver will always run `send_frame(nak, 0, frame_expected, out_buf);`. And then the sender and will keep send the same frame again but the reciver will ignore it. But the sender don't know it will time out and then send again. The whole process will stuck here.
Therefore it is necessary to set the auxiliary timer which can result in a correct acknowledgement being sent back eventually instead, which resynchronizes.

4. Suppose that the three-statement while loop near the end of protocol 6 (See Figure 3-21 from the book) was removed from the code. Would this affect the correctness of the protocol or just the performance? Explain your answer.

Solution: If the three-statement while loop near the end of protocol 6 was removed from the code, this will affect the correctness of the protocol and performance.
Explain:
If removed the code like in the following:

```
case frame_arrival: /*a data or control frame has arrived*/
    from_physical_layer(&r); /*fetch incoming frame from physical layer*/
    if (r.kind == data) {
    /*An undamaged frame has arrived.*/
        if ((r.seq != frame_expected) && no_nak)
            send_frame(nak, 0, frame_expected, out_buf);
        else start_ack_timer();
        if (between(frame_expected,r.seq,too_far)
        && (arrived[r.seq%NR_BUFS]==false)) {
        /*Frames may be accepted in any order.*/
            arrived[r.seq % NR_BUFS] = true; /*mar k buffer as full*/
            in_buf[r.seq % NR_BUFS] = r.info; /*inser t 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;
                arrived[frame_expected % NR_BUFS] = false;
                inc(frame_expected); /*advance lower edge of receiver's window*/
                inc(too_far); /*advance upper edge of receiver's window*/
                start_ack_timer(); /*to see if a separate ack is needed*/
            }
        }
    }

    if((r.kind==nak)
    && between(ack_expected,(r.ack+1)%(MAX SEQ+1),next_frame_to_send))
        send_frame(data, (r.ack+1) % (MAX_SEQ + 1), frame_expected, out_buf);
        //while (between(ack_expected, r.ack, next_frame_to_send)) {
            //nbuffered = nbuffered - 1; /* handle piggybacked ack */
            //stop_timer(ack_expected % NR_BUFS); /*frame arrived intact*/
            //inc(ack_expected); /*advance lower edge of sender's window*/
        //}
    break;
```

The receiver can not create the acknowledgements successfully, because just delete the while loop code. There is no other place to create acknowledgements, which means the sender will never receive the acknowledgements, and then it will resend again and always time out and never make any progress. The deadlock happens. Therefore, the three-statement while loop near the end of protocol 6 (See Figure 3-21 from the book) cannot be removed from the code.

If these are removed, will affect the correctness of the protocol and the performance.