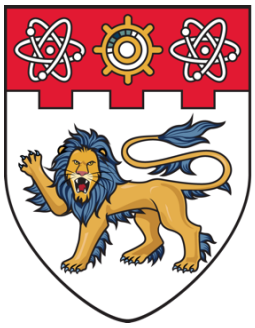

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ3006 Net Centric Computing

Assignment 1

See Jie Xun

U1522059A

Declaration: This assignment was completed individually, and no parts were left unfinished.

1. Introduction

The code implements the “Selective Repeat” strategy for transmitting data in the data link layer. This is a protocol that uses pipelining to ensure high bandwidth utilisation and minimises retransmissions by only retransmitting frames that were missed.

The sender sends a sequence of frames (pipelining). The receiver is capable of buffering arrived frames that are out of order. However, even though it can receive frames out of order, it delivers frames in order to higher layers.

2. Strategy

These are some key elements in the implementation of the “Selective Repeat” strategy.

2.1 Buffering

Both the sender and the receiver maintain a window of acceptable frames, and their windows are expanded, shrunk, and rotated independently. Both sender and receiver also maintain a buffer of frames. As a result, the receiver is able to receive frames non-sequentially.

When a frame arrives, if its sequence number falls within the receiver’s window and if it has not already been received, it is accepted and stored. This is regardless of whether it contains the next packet expected by the network layer. However, it is kept within the data link layer (in the receiver’s buffer) and not passed to the network layer until all preceding frames have already been delivered, so as to transmit the frames in order.

2.2 Preventing overlap in window

For non-sequential receive to function successfully, the maximum window size is set to at most half the range of the sequence numbers. Ensuring there is no overlap takes care of the edge case where acknowledgements are lost.

2.3 Negative acknowledgement

A negative acknowledgement (NAK) frame is introduced, to improve the performance of the protocol. When an error – checksum error or out-of-sequence frame arrival – is detected, the receiver can send an NAK, to request retransmission even before the corresponding timer expires.

If an NAK frame gets lost, no harm is done as the required frame will still be retransmitted when its corresponding timer expires. However, only one NAK is allowed to be sent at a time, to avoid making multiple retransmission requests for the same frame.

2.4 Acknowledgement timer

Acknowledgements are piggybacked onto outgoing frames. However, if the channel has a lot of traffic in one direction and little traffic in the other direction, acknowledgements will be slow to be sent, resulting in inefficiency. An auxiliary timer is introduced and is started after an in-sequence data frame arrives. If no reverse traffic has presented itself before the timer expires, a separate acknowledgement frame is sent. The amount of time before the auxiliary timer expires has to be significantly shorter than that of the timer used for timing out data frames.

3. Explanation of code

The code contains comments that shed light on why some parts are written as they are, including additional comments not found in the textbook. This section provides more detail on the workings of the code.

3.1 Helper functions

The protocol6 function is the main function that implements the logic behind the "Selective Repeat" strategy. These are some helper functions that are used by the protocol6 function.

Function name	Description
between	Checks if a sequence number is between the lower and upper bound of a window. The window is a rotating window, where the upper bound can be lower in value than the lower bound, so this function has to be robust to handle those cases.
inc	Increments a value, but in a rotating manner where the maximum value is MAX_SEQ
send_frame	Constructs and sends a data, acknowledgement, or negative acknowledgement frame, and sends it through the physical layer. It is invoked at multiple points in protocol6, so it is abstracted for convenience.

3.2 Implementation of timers

The modules java.util.Timer and java.util.TimerTask are used to implement the frame timeout timer and the auxiliary acknowledgement timer.

For frame timeouts, an array of timers is needed, of size equal to the size of the sending buffer. In other words, there is one timer for each buffer. When a data frame is sent, a timer corresponding to the sequence number of the frame sent is started. The TimerTask class is extended to create a FrameTimeoutTask, which has to be initialised with a sequence number. The frame timeout timer is scheduled with this FrameTimeoutTask, and when it activates, the generate_timeout_event(seqnr) method of the SWE class is called.

Only one auxiliary timer is necessary, and it is implemented similarly. When an undamaged data frame is received, this auxiliary acknowledgement timer is started, and if there are no frames that an acknowledgement can piggyback on before the timer is up, then a separate acknowledgement frame will have to be sent. As above, the TimerTask class is extended to create an AckTimeoutTask, with which the auxiliary timer is scheduled. When activated, the generate_acktimeout_event() method of the SWE class is called.

The amount of time before timeout for the auxiliary timer is set at 100 ms, and 300 ms for the timer for data frame retransmissions. These values were chosen empirically.

3.3 Walkthrough of protocol6 function code

The code for the protocol6 function is interjected with explanatory notes below each chunk.

```
1.  public void protocol6() {
2.      int ack_expected = 0; /* lower edge of sender's window */
3.      int next_frame_to_send = 0; /* upper edge of sender's window + 1 */
4.      int frame_expected = 0; /* lower edge of receiver's window */
5.      int too_far = NR_BUFS; /* upper edge of receiver's window + 1 */
6.      PFrame r = new PFrame(); /* scratch variable */
7.      boolean arrived[] = new boolean[NR_BUFS]; /* inbound bit map */
```

```
8.      for (int i = 0; i < NR_BUFS; i++) arrived[i] = false;
```

Lines 2-5 declare some variables that keep track of the receiver's and sender's windows. Line 7 declares an array that keeps track of which buffers in the receiver's buffer array are available.

```
9.      enable_network_layer(NR_BUFS); /* initialize */
10.     init();
11.     while (true) {
12.         wait_for_event(event);
13.         switch (event.type) {
```

In line 8, enable_network_layer(credit) is an important function call that informs the network layer of how many buffers there are available. When there is at least one buffer available, the network layer is then able to generate an event (i.e. a NETWORK_LAYER_READY event). In line 11, the function enters into a permanent loop where it starts by waiting for an event (line 12) and acting accordingly.

```
14.             case (PEvent.NETWORK_LAYER_READY):
15.                 from_network_layer(out_buf[next_frame_to_send % NR_BUFS]);
16.                 send_frame(PFrame.DATA, next_frame_to_send, frame_expected, out_buf);
17.                 next_frame_to_send = inc(next_frame_to_send);
18.                 break;
```

When the network layer is ready to send a frame. The sender buffer is populated (line 15) and then the frame is sent to the physical layer by calling send_frame (line 16). The frame is sent with the sequence number being next_frame_to_send. In line 17, the next_frame_to_send variable is incremented to advance the upper edge of the sender's window, indicating the next frame to be sent.

```
19.             case (PEvent.FRAME_ARRIVAL):
20.                 from_physical_layer(r);
21.                 if (r.kind == PFrame.DATA) { /* an undamaged frame has arrived */
22.                     if ((r.seq != frame_expected) && no_nak) {
23.                         send_frame(PFrame.NAK, 0, frame_expected, out_buf);
24.                     } else start_ack_timer();
```

When a frame arrives, line 20 populates the PFrame r with the attributes of the arriving frame. If the frame is a data frame (line 21), then it is checked if the frame number is expected. If it is not expected, and if no negative acknowledgement (NAK) has been sent for the frame, then a NAK is sent (line 23). The expected frame number is sent as the sequence number in the send_frame function. Otherwise, the auxiliary timer is started. This is to address the edge case where a deadlock can arise if all acknowledgements are lost for a window of frames. The sender would have no way of knowing that the receiver has received the frames and would keep attempting to retransmit frames that the receiver has already received.

```
25.                     if (between(frame_expected, r.seq, too_far) && arrived[r.seq % NR_BUFS]
    == false) { /* check if incoming frame is within receiver window */ /* Frames may be accep
    ted in any order */
26.                         arrived[r.seq % NR_BUFS] = true; /* mark buffer as full */
27.                         in_buf[r.seq % NR_BUFS] = r.info; /* insert data into buffer */
28.                         while (arrived[frame_expected % NR_BUFS]) {
29.                             to_network_layer(in_buf[frame_expected % NR_BUFS]);
30.                             no_nak = true;
31.                             arrived[frame_expected % NR_BUFS] = false;
32.                             frame_expected = inc(frame_expected);
33.                             too_far = inc(too_far);
34.                             start_ack_timer(); /* to see if a separate ack is needed */
35.                         }
36.                     }
37.                 }
```

Line 25 checks if the arriving frame is within the receiver window, and if it has not already been buffered. If the conditions are met, then the buffer for the frame is marked as full (line 26), and the data is inserted into the receiver buffer (line 27). The while loop in line 28 begins if the expected frame has arrived. Only then will the data be transmitted to the network layer (line 29). This is because the frames are to be transmitted in order. NAK is once again allowed (line 30), and buffer is marked as free (line 31). The lower and upper edge of the receiver window are incremented (line 32, 33). Line 34 starts the auxiliary timer to send an acknowledgement if there are no frames to piggyback on.

```
38.                 /* retransmit frame that was not transmitted successfully */
39.                 if ((r.kind == PFrame.NAK) && between(ack_expected, (r.ack + 1) % (MAX_SEQ
    + 1), next_frame_to_send)) /* check if r.ack+1 is within sending window */ send_frame(PFram
    e.DATA, (r.ack + 1) % (MAX_SEQ + 1), frame_expected, out_buf);
```

If a NAK is received, and (r.ack+1) is within the sending window, then the (r.ack+1) frame is retransmitted. The frame number (r.ack+1) corresponds to the frame that the NAK is being sent for.

```
40.                 while (between(ack_expected, r.ack, next_frame_to_send)) { /* when ack is r
    eceived for frame with a higher seq number, means all preceding frames have also been recei
    ved. */
41.                     stop_timer(ack_expected); /* frame arrived intact */
42.                     ack_expected = inc(ack_expected); /* advance lower edge of sender's win
    dow */
43.                     enable_network_layer(1);
44.                 }
45.                 break;
```

This while loop stops the frame timeout timers for frames that have been sent and are awaiting acknowledgement. When an ack is received for a frame with a higher sequence number, by this protocol, it means that all preceding frames have also been received successfully. With this loop, the lower edge of the sender's window is advanced until it increments past the acknowledged frame, and the timers for all the corresponding frames are stopped. This loop takes place as long as a FRAME_ARRIVAL event happens. This is because acknowledgements can be piggybacked onto data, NAK or ACK frames all the same.

```
46.             case (PEvent.CKSUM_ERR):
47.                 if (no_nak) send_frame(PFrame.NAK, 0, frame_expected, out_buf); /* damaged
    frame */
48.                 break;
```

In case of a checksum error event, a NAK for the expected frame is sent if it has not already been sent.

```
49.             case (PEvent.TIMEOUT):
50.                 send_frame(PFrame.DATA, oldest_frame, frame_expected, out_buf);
51.                 break;
```

In case of a frame timeout without an acknowledgement being received, the frame is retransmitted. The sequence number oldest_frame is used. This is the correct frame to be retransmitted as the timeout event is generated with the sequence number that needs to be retransmitted and oldest_frame is set to this number.

```
52.             case (PEvent.ACK_TIMEOUT):
53.                 send_frame(PFrame.ACK, 0, frame_expected, out_buf);
54.                 break;
55.             default:
56.                 System.out.println("SWP: undefined event type = " + event.type);
57.                 System.out.flush();
58.         }
59.     }
60. }
```

In case of an acknowledgement timeout, an acknowledgement is sent for the expected frame which has been received.

4. Appendix

4.1 Full code

```
1.  private static boolean no_nak = true;
2.  static boolean between(int a, int b, int c) {
3.      if (((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a))) return true
    ; /* checks for between relationship even if window edge is rotating */
4.      else return false;
5.  }
6.  static int inc(int f) {
7.      if (f < MAX_SEQ) {
8.          f++;
9.      } else f = 0;
10.     return f; // return (f + 1)%(MAX_SEQ + 1);
11. }
12. private void send_frame(int fk, int frame_nr, int frame_expected, Packet buffer[]) { /* Con
    struct and send a data, ack, or nak frame. */
13.     PFrame s = new PFrame(); /* scratch variable */
14.     s.kind = fk; /* kind == data, ack, or nak */
15.     if (fk == PFrame.DATA) s.info = buffer[frame_nr % NR_BUFS];
16.     s.seq = frame_nr; /* only meaningful for data frames */
17.     s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1); /* acknowledging the one frame befo
    re the frame_expected. MAX_SEQ+1 because indexing from zero. */
18.     if (fk == PFrame.NAK) no_nak = false; /* one nak per frame, please */
19.     to_physical_layer(s); /* transmit the frame */
20.     if (fk == PFrame.DATA) start_timer(frame_nr);
21.     stop_ack_timer(); /* no need for separate ack frame, the ack is piggybacked on this fra
    me */
22. }
23. public void protocol6() {
24.         int ack_expected = 0; /* lower edge of sender's window */
25.         int next_frame_to_send = 0; /* upper edge of sender's window + 1 */
26.         int frame_expected = 0; /* lower edge of receiver's window */
27.         int too_far = NR_BUFS; /* upper edge of receiver's window + 1 */
28.         PFrame r = new PFrame(); /* scratch variable */
29.         boolean arrived[] = new boolean[NR_BUFS]; /* inbound bit map */
30.         enable_network_layer(NR_BUFS); /* initialize */
31.         for (int i = 0; i < NR_BUFS; i++) arrived[i] = false;
32.         init();
33.         while (true) {
34.             wait_for_event(event);
35.             switch (event.type) {
36.                 case (PEvent.NETWORK_LAYER_READY):
37.                     /* will only happen when network layer is enabled, which is when out bu
    ffer has available space */ from_network_layer(out_buf[next_frame_to_send % NR_BUFS]);
38.                     send_frame(PFrame.DATA, next_frame_to_send, frame_expected, out_buf);
39.                     next_frame_to_send = inc(next_frame_to_send);
40.                     break;
41.                 case (PEvent.FRAME_ARRIVAL):
42.                     from_physical_layer(r);
43.                     if (r.kind == PFrame.DATA) { /* an undamaged frame has arrived */
44.                         if ((r.seq != frame_expected) && no_nak) {
45.                             send_frame(PFrame.NAK, 0, frame_expected, out_buf);
46.                         } else start_ack_timer(); /* only want to send one no_nak at a time
    . this timer is for ack to be sent for previous frame. */
47.                         if (between(frame_expected, r.seq, too_far) && arrived[r.seq % NR_B
    UFS] == false) { /* check if incoming frame is within receiving window */ /* Frames may be
    accepted in any order */
48.                             arrived[r.seq % NR_BUFS] = true; /* mark buffer as full */
49.                             in_buf[r.seq % NR_BUFS] = r.info; /* insert data into buffer */

50.                             while (arrived[frame_expected % NR_BUFS]) { /* Pass frames and
    advance window. */
51.                                 to_network_layer(in_buf[frame_expected % NR_BUFS]); /* deli
    ver frames in that order. */
```

```
                                no_nak = true;
                                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 */

                            }
                        }
                    } /* retransmit frame that was not transmitted successfully */
                    if ((r.kind == PFrame.NAK) && between(ack_expected, (r.ack + 1) % (MAX_SEQ + 1), next_frame_to_send)) /* check if r.ack+1 is within sending window */ send_frame(PFrame.DATA, (r.ack + 1) % (MAX_SEQ + 1), frame_expected, out_buf);
                    while (between(ack_expected, r.ack, next_frame_to_send)) { /* when ack is received for frame with a higher seq number, means all preceding frames have also been received. */
                        stop_timer(ack_expected); /* frame arrived intact */
                        ack_expected = inc(ack_expected); /* advance lower edge of sender's window */
                        enable_network_layer(1);
                    }
                    break;
                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); /* oldest_frame is set in wait_for_event(PEvent e), and this TIMEOUT event is generated with the seqnr that needs to be retransmitted */
                    break;
                case (PEvent.ACK_TIMEOUT):
                    send_frame(PFrame.ACK, 0, frame_expected, out_buf);
                    break;
                default:
                    System.out.println("SWP: undefined event type = " + event.type);
                    System.out.flush();
            }
        }
    }
    /* Note: when start_timer() and stop_timer() are called, the "seq" parameter must be the sequence number, rather than the index of the timer array, of the frame associated with this timer, */
public class FrameTimeoutTask extends TimerTask {
    private int seq;
    public FrameTimeoutTask(int seq) {
        this.seq = seq;
    }
    public void run() {
        swe.generate_timeout_event(seq);
    }
}
public class AckTimeoutTask extends TimerTask {
    public void run() {
        swe.generate_acktimeout_event();
    }
}
private Timer[] timers = new Timer[NR_BUFS];
private Timer ack_timer;
private void start_timer(int seq) {
            stop_timer(seq); /* may be starting a timer that has already been started, hence stop first */
            int i = seq % NR_BUFS;
            timers[i] = new Timer();
            timers[i].schedule(new FrameTimeoutTask(seq), 300);
```

```
104.        }
105.        private void stop_timer(int seq) {
106.            int i = seq % NR_BUFS;
107.            if (timers[i] != null) {
108.                timers[i].cancel();
109.                timers[i] = null;
110.            }
111.        }
112.        private void start_ack_timer() {
113.            stop_ack_timer();
114.            ack_timer = new Timer();
115.            ack_timer.schedule(new AckTimeoutTask(), 100);
116.        }
117.        private void stop_ack_timer() {
118.            if (ack_timer != null) {
119.                ack_timer.cancel();
120.                ack_timer = null;
121.            }
122.        }
```