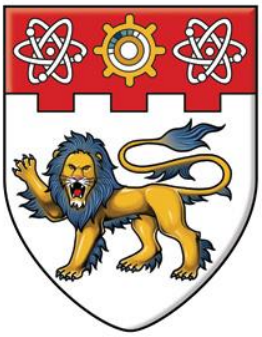

NANYANG TECHNOLOGICAL UNIVERSITY

Assignment 1 Report

CZ3006: Net Centric Computing

Academic Year 2018/2019

Semester 1

Name: Nathanael S Raj

Matriculation Number: U1521654C

Lab Group: SSP5

Summary of Parts Implemented

All parts of the assignment were fully completed and implemented and the Sliding Window Protocol (SWP) was able to achieve full accuracy even with level 3 noise. For the rest of the report I will summarise the approach in fulfilling the following features implemented in the SWP:

1. Full-duplex data communication
2. In-order delivery of packets to the network-layer
3. Negative acknowledgement
4. Selective repeat retransmission strategy
5. Synchronization with the network-layer by granting credits
6. Separate acknowledgment when the reverse traffic is light or none

Full-duplex data communication

The SWP facilitates full-duplex communication as the algorithm supports both the functionality of a sender and receiver. To reduce the traffic on the channel when acknowledging receipt of messages from a sender, a receiver waits for an outgoing message and piggybacks the acknowledgement onto this outgoing message as opposed to delivering a dedicated acknowledgement frame.

```
public void send_frame(int fk, 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 = fk; /* kind == data, ack, or nak */
        if (fk == PFrame.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 == PFrame.NAK)
                no_nak = false; /* one nak per frame, please */
        to_physical_layer(s); /* transmit the frame */
        if (fk == PFrame.DATA)
                start_timer(frame_nr);
        stop_ack_timer(); /* no need for separate ack frame */
}
```

The highlighted code shows that whenever a frame is sent, it also sends a acknowledgement as part of the meta data of the frame, this piggybacked acknowledgement would corresponds to the frame just received.

In-order delivery of packets to the network-layer

The SWP allows packets to be received out of order if it is within the window but ensures that the packets will be delivered in order to the network layer. The code snippet below shows how the SWP handles frame arrivals. During frame arrival, its sequence number (r.seq) is checked if it falls within the current window. This is performed by the line:

if (between(frame_expected, r.seq, too_far) && (arrived[r.seq%NR_BUFS] == false)).

If the arriving frame has a sequence number between the frame expected (lower window boundary) and too_far(upper window boundary) and it has is arriving for the first time, it will be accepted, even if out of order.

Despite this, the order in which arriving frames are sent to the network layer is preserved by the following while statement:

while (arrived[frame_expected % NR_BUFS])

As shown in the code below, by iterating through the buffer of frames arrived, starting at the lower window boundary (frame_expected), the SWP ensures that frames are sent to the network layer from the lower boundary to the upper boundary, preserving the order.

```
case (PEvent.FRAME_ARRIVAL ):
    from_physical_layer(r); /* fetch incoming frame from physical layer */
    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)) {
            /* 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;
                frame_expected = (frame_expected + 1) % (MAX_SEQ + 1); /* advance lower edge of receiver's window */
                too_far = (too_far + 1) % (MAX_SEQ + 1); /* 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)) {

        stop_timer(ack_expected % NR_BUFS); /* frame arrived intact */
        ack_expected = (ack_expected + 1) % (MAX_SEQ + 1); /* advance lower edge of sender's window */
        enable_network_layer(1);
    }
    break;
```

Negative acknowledgement

Negative acknowledgements are sent in two scenarios: Firstly, when a frame is sent out of order and secondly, when there is a checksum error in the frame. In these scenarios, the sender is informed by the receiver that a frame has been sent wrongly and it responds by selectively retransmitting the wrong frame.

1. Frame is sent out of order

```
case (PEvent.FRAME_ARRIVAL ):
    from_physical_layer(r); /* fetch incoming frame from physical layer */
    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)) {
            /* 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;
                frame_expected = (frame_expected + 1) % (MAX_SEQ + 1); /* advance lower edge of receiver's window */
                too_far = (too_far + 1) % (MAX_SEQ + 1); /* 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)) {

        stop_timer(ack_expected % NR_BUFS); /* frame arrived intact */
        ack_expected = (ack_expected + 1) % (MAX_SEQ + 1); /* advance lower edge of sender's window */
        enable_network_layer(1);
    }
    break;
```

The code snippet above shows what happens during frame arrival. The highlighted portion, shows that when a frame arrives out of order, a negative acknowledgement (PFrame.NAK) is sent as a reply to the sender with its expected frame number (frame_expected). The no_nak Boolean flag is used to ensure that the same negative acknowledgement is not sent twice.

2. Checksum error

```
case (PEvent.CKSUM_ERR):
        if (no_nak)
                send_frame(PFrame.NAK, 0, frame_expected, out_buf);
        break;
```

When a frame with a checksum error arrives, it is handled by the code snippet above. The checksum error flag (PEvent.CKSUM_ERR) is raised and negative acknowledgement is sent.

Selective repeat retransmission strategy

Selective repeat transmission strategy is made possible by negative acknowledgements. As mentioned above, when frames are sent out of order or contain checksum errors, the sender will receive a negative acknowledgment from the receiver and it responds by retransmitting the failed frames. The sender only retransmits the wrong frames and not the frames proceeding the wrong frame, as opposed to the Go-Back-N strategy.

```
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);
```

The code snippet shows how a negative acknowledgment is handled. When the frame kind flag is set to negative acknowledgement (PFrame.NAK) and it falls within the sender window, the frame indicated in the negative acknowledgment message (r.ack) will be retransmitted.

Sent frames also have a timer to receive acknowledgements and if it times out, the frames are also retransmitted. During timeout, swe.generate_timeout_event(seq) is called, in which the sliding window environment is asked to generate a time out event on the named sequence number (seq). It generates a PEvent.TIMEOUT event, which will trigger the retransmission of the oldest frame as shown by the code snippet below.

```
case (PEvent.TIMEOUT):
        send_frame(PFrame.DATA, oldest_frame, frame_expected, out_buf); /* we timed out */
        break;
```

Synchronization with the network-layer by granting credits

To prevent the network layer from flooding the SWP with frames, the network layer is only allowed to generate a network layer ready event (PEvent.NETWORK_LAYER_READY) when it is granted credits.

```
public void protocol6() {
    init();
 int ack_expected;  /* lower edge of sender's window */
 int next_frame_to_send; /* upper edge of sender's window + 1 */
 int frame_expected; /* lower edge of receiver's window */
 int too_far; /* upper edge of receiver's window + 1 */
 int i; /* index into buffer pool */
 PFrame r = new PFrame(); /* scratch variable */
 Packet in_buf[] = new Packet[NR_BUFS]; /* buffers for the inbound stream */
 boolean arrived[] = new boolean[NR_BUFS]; /* inbound bit map */
 int nbuffered; /* how many output buffers currently used */


 enable_network_layer(NR_BUFS); /* 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;
```

The following code above shows how the SWP is initialised. The network layer is initially given credits equal to the buffer size.

```
 while (between(ack_expected, r.ack, next_frame_to_send)) {

     stop_timer(ack_expected % NR_BUFS); /* frame arrived intact */
     ack_expected = (ack_expected + 1) % (MAX_SEQ + 1); /* advance lower edge of sender's window */
     enable_network_layer(1);
 }
 break;
```

During frame arrival, more credits are generated as the sender window is advanced as shown by the code above. When acknowledgements are received, the sender will move its window and give more credits to the network layer according to the distance the window moved.

In this way, the SWP ensures that the number of frames received from the network layer does not exceed the number of frames being currently processed.

```
case (PEvent.NETWORK_LAYER_READY):
    from_network_layer(out_buf[next_frame_to_send % NR_BUFS]); /* fetch new packet */
    send_frame(PFrame.DATA, next_frame_to_send, frame_expected, out_buf);/* transmit the frame */
    next_frame_to_send = (next_frame_to_send + 1) % (MAX_SEQ + 1);  /* advance upper window edge */
    break;
```

Finally, we see from the above code that the SWP only receives frame from the network layer when PEvent.NETWORK_LAYER_READY, which in turn can only be generated when the network layer has credits. Thus, the SWP is able to synchronise the number of frames processed between the receiver and the network layer.

Separate acknowledgment when the reverse traffic is light or none

```
case (PEvent.FRAME_ARRIVAL ):
    from_physical_layer(r); /* fetch incoming frame from physical layer */
    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();
```

From code above, we see that during frame arrival, if a data frame has arrived in order, the acknowledgement timer will be started (start_ack_timer()). This timer will be stopped when an acknowledgement is sent, piggybacked onto an outgoing data frame to the sender. We see this in the highlighted line below.

```
public void send_frame(int fk, 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 = fk; /* kind == data, ack, or nak */

        if (fk == PFrame.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 == PFrame.NAK)

            no_nak = false; /* one nak per frame, please */

        to_physical_layer(s); /* transmit the frame */

        if (fk == PFrame.DATA)

            start_timer(frame_nr);

        stop_ack_timer(); /* no need for separate ack frame */

    }
```

However, if there is no or light traffic coming from the network layer, the SWP will need to generate a dedicated acknowledgement frame to the sender since there are no outgoing data frames to be piggybacked on. Pass a predefined period, the acknowledgement timer will timeout and swe.generate_acktimeout_event(); will be called. This triggers the PEvent.ACK_TIMEOUT event which results in the dedicated acknowledgement message being sent as shown in the code below.

```
case (PEvent.ACK_TIMEOUT):
    send_frame(PFrame.ACK, 0, frame_expected, out_buf); /* ack timer expired; send ack */
    break;
```

Thus, acknowledgments are always sent, whether there are outgoing frames to be piggybacked on or not.

SWP Implementation

1. Send frame function

```
public void send_frame(int fk, 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 = fk; /* kind == data, ack, or nak */
    if (fk == PFrame.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 == PFrame.NAK)
        no_nak = false; /* one nak per frame, please */
    to_physical_layer(s); /* transmit the frame */
    if (fk == PFrame.DATA)
        start_timer(frame_nr);
    stop_ack_timer(); /* no need for separate ack frame */
}
```

2. Between function

```
public boolean between(int a, int b, int c){
    return ((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a));
}
```

3. Sliding window protocol

```
public void protocol6() {
    init();
int ack_expected;  /* lower edge of sender's window */
int next_frame_to_send; /* upper edge of sender's window + 1 */
int frame_expected; /* lower edge of receiver's window */
int too_far; /* upper edge of receiver's window + 1 */
int i; /* index into buffer pool */
PFrame r = new PFrame(); /* scratch variable */
Packet in_buf[] = new Packet[NR_BUFS]; /* buffers for the inbound stream */
boolean arrived[] = new boolean[NR_BUFS]; /* inbound bit map */
int nbuffered; /* how many output buffers currently used */


enable_network_layer(NR_BUFS); /* 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;
for (i = 0; i < NR_BUFS; i++){
    arrived[i] = false;
}
while(true) {
    wait_for_event(event);
    switch(event.type) {
        case (PEvent.NETWORK_LAYER_READY):
            from_network_layer(out_buf[next_frame_to_send % NR_BUFS]); /* fetch new packet */
            send_frame(PFrame.DATA, next_frame_to_send, frame_expected, out_buf);/* transmit the frame */
            next_frame_to_send = (next_frame_to_send + 1) % (MAX_SEQ + 1);  /* advance upper window edge */
            break;

        case (PEvent.FRAME_ARRIVAL ):
            from_physical_layer(r); /* fetch incoming frame from physical layer */
            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)) {
                /* 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;
                    frame_expected = (frame_expected + 1) % (MAX_SEQ + 1); /* advance lower edge of receiver's
                    too_far = (too_far + 1) % (MAX_SEQ + 1); /* 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)) {

            stop_timer(ack_expected % NR_BUFS); /* frame arrived intact */
            ack_expected = (ack_expected + 1) % (MAX_SEQ + 1); /* 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); /* we timed out */
        break;
    case (PEvent.ACK_TIMEOUT):
        send_frame(PFrame.ACK, 0, frame_expected, out_buf); /* ack timer expired; send ack */
        break;
    default:
        System.out.println("SWP: undefined event type = "
                + event.type);

        System.out.flush();
  }
 }
}
```

4. Sender timer Implementation

```
Timer[] sender_timer = new Timer[NR_BUFS];
Timer ack_timer ;
private void start_timer(int seq) {
    stop_timer(seq);
    sender_timer[seq % NR_BUFS] = new Timer();
    sender_timer[seq % NR_BUFS].schedule(new sender_task(seq), 300);
}

private void stop_timer(int seq) {
    if (sender_timer[seq % NR_BUFS] != null) {
        sender_timer[seq % NR_BUFS].cancel();
        sender_timer[seq % NR_BUFS] = null;
    }
}
class sender_task extends TimerTask {

    private int seq;

    public sender_task(int seq) {
        this.seq = seq;
    }

    public void run() {
        swe.generate_timeout_event(seq);
    }
}
private void start_ack_timer() {
    stop_ack_timer();
    ack_timer = new Timer();
    ack_timer.schedule(new ack_task(), 50);
}
```

5. Acknowledgement timer implementation

```
private void start_ack_timer() {
    stop_ack_timer();
    ack_timer = new Timer();
    ack_timer.schedule(new ack_task(), 50);
}

private void stop_ack_timer() {
    if (ack_timer != null) {
        ack_timer.cancel();
        ack_timer = null;
    }
}
class ack_task extends TimerTask {

    public void run() {

        swe.generate_acktimeout_event();
    }
}
```