

**CZ3006: Net Centric Computing  
AY 16/17 Semester 2**

**Assignment 2 Report**

**Student Name: Sim Long Siang**

**Matriculation Number: U1522053H**

**Lab Group: TS1**

**9 April 2017**

# Introduction

The objective of this report is to document the various features of the implemented Sliding Window Protocol based on the simulated communication system provided. The Sliding Window Protocol is commonly used in the data link layer for flow control and error control.  
  
In this report, several features of the sliding window protocol will be discussed:

* Full-duplex data communication
* In-order delivery of packets to the network-layer
* **Selective repeat** retransmission strategy
* Synchronization with the network-layer by grant credits
* Negative acknowledgement
* Separate acknowledgement when the reverse traffic is light or none

All features implemented will be based on sliding window **Protocol 6 – Selective Repeat** strategy. This protocol accepts frames in an out-of-order fashion, but passes the packets to network layer in-order. It only retransmits one outstanding frame if its timer expires, rather than all outstanding frames, as in Protocol 5.

# Features of Sliding Window Protocol

## 1. Full-duplex data communication

Given two machines, a working full-duplex data communication allows data to flow in both direction, forward and reverse. Protocol 6 achieved this full-duplex concept by making the machine get triggered by events. These events including receiving a frame from outside, sending frames out after fetching packets from network layer. These events allow the machine to be both a sender or a receiver.  
  
*while (true) {*

*wait\_for\_event(&event);*

*switch(event) {*

*case* ***network\_layer\_ready****: /\* the network layer has a packet to send \*/*

*from\_network\_layer(&buffer[next\_frame\_to\_send]); /\* fetch new packet \*/*

*…*

*send\_data(next\_frame\_to\_send, frame\_expected, buffer);/\* transmit the frame \*/*

*…*

*break;*

*case* ***frame\_arrival****: /\* a data or control frame has arrived \*/   
from\_physical\_layer(&r); /\* get incoming frame from physical layer \*/*

## 2. In-order delivery of packets to the network-layer

There is a possibility that frames might be damaged and lost throughout the transmission. A one-bit frame error can cause a packet to be corrupted. Therefore, it is critical that any packets delivered to the network layer must be in order first. Protocol 6 presents itself as one that accepts frames in out-of-order fashion but delivers packets in order. This allows flexibility for lost and damaged frames to be received again by the machine after retransmission.

The implementation is as follows:

1. If the frames arrival is not the expected frame, the receiver will check whether it falls between the expected frame of sliding window and if it has been received before.

*if (between(expected\_frame, temp\_frame.seq, upper\_limit) && (recieved[temp\_frame.seq % NR\_BUFS] == false)) {*

2. If both the condition is satisfied, it will accept the frame in any auxiliary array even if it is out of order.

*recieved[temp\_frame.seq % NR\_BUFS] = true;*

3. The received frame are inserted into the buffer at the correct position.

*in\_buffer[temp\_frame.seq % NR\_BUFS] = temp\_frame.info;*

4. Receiver will start to send the frames in-order to the network layer using the *received* array as a guide.

*while (received [expected\_frame % NR\_BUFS]) {*

*to\_network\_layer(in\_buffer[expected\_frame % NR\_BUFS]);*

*…*

*start\_ack\_timer ();*

*}*

## 3. Selective repeat retransmission strategy

The Main feature of Protocol 6 is its selective repeat strategy. Using a timer function, only that particular frame that are lost or damaged will be retransmitted by the sender. This greatly reduces the overall retransmission time as compared to Protocol 5.  
  
Selective Repeat strategy is implemented as follows:

1. For every data frame to be send over, a timer is created for that particular frame.

*private void send\_frame(int f, int next\_frame, int frame\_expected, Packet buffer[])*

*{*

*...*

*if (f == PFrame.DATA) start\_timer(next\_frame);*

*…*

*}*

*private void start\_timer(int seq) {*

*stop\_timer(seq);*

*frame\_timer[seq % NR\_BUFS] = new Timer(); //create new timer*

*frame\_timer[seq % NR\_BUFS].schedule(new ReTask(swe, seq), 200);// set timeout*

*}*

*2.* If the frame did not successfully transmit, a TIMEOUT event on the frame will be triggered. The variable *oldest\_frame* will be updated with the frame’s sequence number that caused the TIMEOUT event. This variable is then used to identify the frame required for retransmission.

*private void wait\_for\_event(PEvent e){*

*swe.wait\_for\_event(e); //may be blocked*

*oldest\_frame = e.seq; //set timeout frame seq*

*}*

*while(true) {*

*wait\_for\_event(event);*

*switch(event.type) {*

*…*

*case (PEvent.TIMEOUT):*

*//resend the frame for which the timer has expired*

*send\_frame(PFrame.DATA, oldest\_frame, expected\_frame, out\_buf);*

*break;*

*}*

**4. Synchronization with the network-layer by granting credits**

In order for the network layer to have sufficient time to receive packets from the physical layer, synchronization is done within the data link layer. This is achieved through granting credits to the network layer based on the number of available buffer. The buffer is determined by the window size declared below:

*public static final int NR\_BUFS = (MAX\_SEQ + 1)/2;*

When the protocol starts up, it will be initialised with the window size value to determine the available buffer.

*enable\_network\_layer(NR\_BUFS);*

*private void enable\_network\_layer(int nr\_of\_bufs) {*

*//network layer is permitted to send if credit is available*

*swe.grant\_credit(nr\_of\_bufs);*

*}*

After which, the network layer is granted credits based on the acknowledgement sent by the receiver and the amount of available buffers, allowing the network layer to be synchronized to the data link layer and both the sending and receiving window

*while (between(expected\_ack, temp\_frame.ack, next\_frame\_send)){*

*stop\_timer(expected\_ack % NR\_BUFS); //If a complete frame is received*

*expected\_ack = inc(expected\_ack); // increase the expected sliding window for the ack to be received*

*//for the sent data frames*

*enable\_network\_layer(1); //always free 1 buffer slot if ack has been done*

*}*

## 5. Negative acknowledgement

Negative acknowledgement is implemented in Protocol 6 to ensure that the sender knows information that a particular frame sent previously has not been successfully received by the receiver. Declaring a boolean variable *no\_nak* allows the protocol to consistently keep track whether a negative acknowledgement is sent for a particular damaged or lost frame. If no negative acknowledgement is to be sent, the value will be *true*.

*boolean no\_nak = true;*

Before sending a negative acknowledge frame to the physical layer, the protocol is required to keep track of this, thus changing the value to *false.*

*private void send\_frame(int f, int next\_frame, int frame\_expected, Packet buffer[])*

*{*

*…*

*if (f == PFrame.NAK) no\_nak = false; //one nak per frame*

*to\_physical\_layer(s); //transmit frame*

*…*

*}*

The *no\_nak* variable is also used to as a condition to determine if the frame is corrupted. In the code below, even though the frame might not be the expected frame, but if *no\_nak* is true, it means that this frame is completed, just out-of-order.

*from\_physical\_layer(temp\_frame);*

*if (temp\_frame.kind == PFrame.DATA){*

*if ((temp\_frame.seq != expected\_frame) && no\_nak)*

.

## 6. Separate acknowledgement when the reverse traffic is light or more

Sending separate acknowledgement allows the channel to be used/optimised to the fullest since the concept of piggybacking reduces bandwidth that are wasted. This is done by implementing another timer *start\_ack\_timer*. This timer ensures that a separate acknowledgement frame will be sent once the timer expires.

*private void start\_ack\_timer( ) {*

*stop\_ack\_timer();*

*ack\_timer = new Timer(); //starts another timer for sending separate ack*

*ack\_timer.schedule(new AckTask(swe), 100);*

*}*

There’s a need to have a function that stops the acknowledgement timer as acknowledgement frame is being piggybacked.

*private void send\_frame(int f, int next\_frame, int frame\_expected, Packet buffer[])*

*{*

*…*

*stop\_ack\_timer(); //no need for separate ack frame*

*}*

**Conclusion**

The overall implementation of Protocol 6 – Selective Repeat mentioned in this report has managed to fulfil all requirements of the sliding window protocol features. It is able to withstand all quality levels of service. The implementation program code can be found in the next section to verify the correctness of the assignment.

\*===============================================================\*

\* File: SWP.java

\*================================================================\*

import java.util.Timer;

import java.util.TimerTask;

public class SWP {

/\*========================================================================\*

the following are provided, do not change them!!

\*========================================================================\*/

//the following are protocol constants.

public static final int MAX\_SEQ = 7;

public static final int NR\_BUFS = (MAX\_SEQ + 1)/2;

// the following are protocol variables

private int oldest\_frame = 0;

private PEvent event = new PEvent();

private Packet out\_buf[] = new Packet[NR\_BUFS];

//declare in\_buffer which is a packet data type

private Packet in\_buffer[] = new Packet[NR\_BUFS];

//the following are used for simulation purpose only

private SWE swe = null;

private String sid = null;

//Constructor

**public SWP(SWE sw, String s)**{

swe = sw;

sid = s;

}

//the following methods are all protocol related

**private void init()**{

for (int i = 0; i < NR\_BUFS; i++){

out\_buf[i] = new Packet();

in\_buffer[i] = new Packet(); //initialize in buffer in arrays

}

}

**private void wait\_for\_event(PEvent e)**{

swe.wait\_for\_event(e); //may be blocked

oldest\_frame = e.seq; //set timeout frame seq

}

private void enable\_network\_layer(int nr\_of\_bufs) {

//network layer is permitted to send if credit is available

swe.grant\_credit(nr\_of\_bufs);

}

**private void from\_network\_layer(Packet p**) {

swe.from\_network\_layer(p);

}

**private void to\_network\_layer(Packet packet)** {

swe.to\_network\_layer(packet);

}

**private void to\_physical\_layer(PFrame fm)** {

System.out.println("SWP: Sending frame: seq = " + fm.seq + " ack = " + fm.ack + " kind = " +

PFrame.KIND[fm.kind] + " info = " + fm.info.data );

System.out.flush();

swe.to\_physical\_layer(fm);

}

**private void from\_physical\_layer(PFrame fm)** {

PFrame fm1 = swe.from\_physical\_layer();

fm.kind = fm1.kind;

fm.seq = fm1.seq;

fm.ack = fm1.ack;

fm.info = fm1.info;

}

\*===========================================================================\*

Code implementation \*==========================================================================\*/

//Protocol 6 – Selective Repeat

//queue to check circular condition

**public static boolean between(int x, int y, int z**) {

return ((x <= y) && (y < z)) || ((z < x) && (x <= y)) || ((y < z) && (z < x));

}

boolean no\_nak = true; //no nak has been sent yet

//A java method to send the frames from

**private void send\_frame(int f, int next\_frame, int frame\_expected, Packet buffer[])**

{

PFrame s = new PFrame(); //Decaring temporary variable for frame

s.kind = f; //There are 3 kinds of frames, namely data,ack or nak

if (f == PFrame.DATA) s.info = buffer[next\_frame % NR\_BUFS];

s.seq = next\_frame; //only meaningful for data frames

s.ack = (frame\_expected + MAX\_SEQ) % (MAX\_SEQ + 1);

if (f == PFrame.NAK) no\_nak = false; //one nak per frame

to\_physical\_layer(s); //transmit frame

if (f == PFrame.DATA) start\_timer(next\_frame);

stop\_ack\_timer(); //no need for separate ack frame

}

**public void protocol6()** {

init();

int expected\_ack; //expected frame acknowledgement

int next\_frame\_send; //Frame number of the next frame to be sent

int expected\_frame; //frame expected to be recieved by the reciever

int index; //index into buffer pool

int upper\_limit; //upper limit of the buffer

boolean recieved[] = new boolean[NR\_BUFS]; //keeping track of frames arrived

PFrame temp\_frame = new PFrame(); //declare the temporary frame

enable\_network\_layer(NR\_BUFS); //initialize network layer

//initialize the counter variables

expected\_ack = 0;

next\_frame\_send = 0;

expected\_frame = 0;

upper\_limit = NR\_BUFS;

index = 0;

for(int i = 0;i<NR\_BUFS;i++)

recieved[i] = false;

//Protocol rules for each of the parties involved

while(true) {

wait\_for\_event(event);

switch(event.type) {

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//when packets are ready to be fetched from network layer

**case (PEvent.NETWORK\_LAYER\_READY):**  
//put the fetched packets in the out buffer of the sender

from\_network\_layer(out\_buf[next\_frame\_send % NR\_BUFS]); //transmit the data fetched which is in the senders' buffer

send\_frame(PFrame.DATA, next\_frame\_send, expected\_frame, out\_buf);

//increment the next frame to be sent from the buffer

next\_frame\_send = inc(next\_frame\_send); break;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**case (PEvent.FRAME\_ARRIVAL):**

from\_physical\_layer(temp\_frame);

if (temp\_frame.kind == PFrame.DATA){ //if the received frame is a DATA  
 //A complete and undamaged frame is received

if ((temp\_frame.seq != expected\_frame) && no\_nak) send\_frame(PFrame.NAK, 0, expected\_frame, out\_buf);

else

start\_ack\_timer();

//If the received frame is not DATA

//Check if frame received is between the expected frames of the sliding window and received frame has not been previously received

if (between(expected\_frame, temp\_frame.seq, upper\_limit) && (recieved[temp\_frame.seq % NR\_BUFS] == false)) {  
 //allows frames to be accepted in any order of arrival in the receiver

recieved[temp\_frame.seq % NR\_BUFS] = true;   
 //add the frame into the input buffer

in\_buffer[temp\_frame.seq % NR\_BUFS] = temp\_frame.info;

while (recieved[expected\_frame % NR\_BUFS]){

//Pass frames to the network layer

to\_network\_layer(in\_buffer[expected\_frame % NR\_BUFS]);

no\_nak = true;  
 //mark the undamaged DATA frame received as received

recieved[expected\_frame % NR\_BUFS] = false;

//Increment the lower expected in the sliding window

expected\_frame = inc(expected\_frame);

//increment the upper edge of the sliding window

upper\_limit = inc(upper\_limit);

start\_ack\_timer(); //start the ack timer

}

}

}

//If the frame is NAK,check if the frame is in between the current sliding window

if ((temp\_frame.kind == PFrame.NAK) && between(expected\_ack,((temp\_frame.ack + 1) % (MAX\_SEQ +1)), next\_frame\_send)){

//send the data frame for which NAK has been received

send\_frame(PFrame.DATA, ((temp\_frame.ack + 1) % (MAX\_SEQ + 1)),expected\_frame, out\_buf);

}

while (between(expected\_ack, temp\_frame.ack, next\_frame\_send)){

stop\_timer(expected\_ack % NR\_BUFS); //If a complete frame is received

expected\_ack = inc(expected\_ack); // increase the expected sliding window //for the sent data frames

enable\_network\_layer(1); // free 1 buffer slot if ack has been done

}

break;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**case (PEvent.CKSUM\_ERR):**

if (no\_nak)

send\_frame(PFrame.NAK, 0, expected\_frame, out\_buf); //damaged frame

break;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**case (PEvent.TIMEOUT):**

//If the timer is expired for the oldest frame, resend the data for the frame

send\_frame(PFrame.DATA, oldest\_frame, expected\_frame, out\_buf);

break;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**case (PEvent.ACK\_TIMEOUT):**

//if ack timer expired for the frame resend the ACK again

send\_frame(PFrame.ACK, 0, expected\_frame, 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,

\*/

Timer frame\_timer[] = new Timer[NR\_BUFS];

Timer ack\_timer;

**public static int inc(int num)** {

num = ((num + 1) % (MAX\_SEQ + 1));

return num;

}

**private void start\_timer(int seq)** {

stop\_timer(seq);

frame\_timer[seq % NR\_BUFS] = new Timer(); //create new timer

frame\_timer[seq % NR\_BUFS].schedule(new ReTask(swe, seq), 200); //schedule the task for execution after 200ms

}

**private void stop\_timer(int seq)** {

if (frame\_timer[seq % NR\_BUFS] != null)

frame\_timer[seq % NR\_BUFS].cancel();

}

**private void start\_ack\_timer( )** {

stop\_ack\_timer();

ack\_timer = new Timer(); //starts another timer for sending separate ack

ack\_timer.schedule(new AckTask(swe), 100);

}

**private void stop\_ack\_timer()** {

if (ack\_timer != null)

ack\_timer.cancel();

}

//for retransmission timer

class ReTask extends TimerTask {

private SWE swe = null;

public int seqnr;

public ReTask(SWE sw, int seq) {

swe = sw;

seqnr = seq;

}

public void run() { //stops timer and discard any scheduled tasks for the current seqnr

stop\_timer(seqnr);

swe.generate\_timeout\_event(seqnr);

}

}

//for ack timer

class AckTask extends TimerTask {

private SWE swe = null;

public AckTask(SWE sw) {

swe = sw;

}

public void run() {

stop\_ack\_timer(); // stop the timer

swe.generate\_acktimeout\_event();

}

}

} //End of class