****

**LAB GROUP: SSP4**

**CZ3006:**

**NET-CENTRIC COMPUTING**

***Assignment 1:***

***Implementation of a Sliding Window Protocol***

by

Alvin Lee Yong Teck (U1620768F)

Professor Sun Chengzheng

N4-02b-56

[czsun@ntu.edu.sg](mailto:CZSUN@NTU.EDU.SG)

**Date of Submission: \_\_\_\_\_\_\_\_\_\_\_\_**

**SCHOOL OF COMPUTER SCIENCE AND ENGINEERING**

**NANYANG TECHNOLOGICAL UNIVERSITY**

# Table of Contents

[Summary Of Assignment Tasks Status 2](#_Toc493427008)

[I. Introduction 3](file:///C:\Users\Vin\Desktop\Shortcuts%20for%20Rocketdock\School%20Work\Y2S1\CZ3006%20NET%20CENTRIC%20COMPUTING\Assignment%201\Alvin%20Lee_U1620768F_CZ3006%20Assignment%201.docx#_Toc493427009)

[a. Network Simulator 3](#_Toc493427010)

[b. Virtual Machine 3](#_Toc493427011)

[b.1. Sliding Window Protocol 3](#_Toc493427012)

[b.2. Sliding Window Environment 3](#_Toc493427014)

[II. Tasks 4](file:///C:\Users\Vin\Desktop\Shortcuts%20for%20Rocketdock\School%20Work\Y2S1\CZ3006%20NET%20CENTRIC%20COMPUTING\Assignment%201\Alvin%20Lee_U1620768F_CZ3006%20Assignment%201.docx#_Toc493427016)

[a. Full-Duplex Data Communication 4](#_Toc493427010)

[b. In-Order Delivery of Packets to the Network-Layer 5](#_Toc493427011)

[c. Selective Repeat Retransmission Strategy 6](#_Toc493427010)

[d. Synchronisation with the Network-Layer by Granting Credits 7](#_Toc493427011)

[e. Negative Acknowledgement 8](#_Toc493427010)

[f. Separate Acknowledgement when the Reverse Traffic is light or none 8](#_Toc493427011)

[g. Ability to withstand quality Level 3 of the Network Simulator component 9](#_Toc493427011)

[III. Java Source Code – Sliding Window Protocol 13](file:///C:\Users\Vin\Desktop\Shortcuts%20for%20Rocketdock\School%20Work\Y2S1\CZ3006%20NET%20CENTRIC%20COMPUTING\Assignment%201\Alvin%20Lee_U1620768F_CZ3006%20Assignment%201.docx#_Toc493427016)

# SUMMARY OF ASSIGNMENT TASKS STATUS

|  |  |  |
| --- | --- | --- |
| S/N | Tasks | Status |
| 1 | **Full-Duplex Data Communication** | **Completed** |
| 2 | **In-Order Delivery of Packets to the Network-Layer** | **Completed** |
| 3 | **Selective Repeat Retransmission Strategy** | **Completed** |
| 4 | **Synchronisation with the Network-Layer by Granting Credits** | **Completed** |
| 5 | **Negative Acknowledgement** | **Completed** |
| 6 | **Separate Acknowledgement when the Reverse Traffic is light or none** | **Completed** |
| 7 | **Ability to withstand quality Level 3 of the Network Simulator component** | **Completed** |

# I. Introduction

This assignment aims to enhance understanding of the **network protocol hierarchy** **and flow control** and **error control techniques** by implementing a **sliding window protocol** in a simulated communication network system.

The simulated communication system consists of the following two major components:

## **a. Network Simulator**

This component simulates the physical transmission media which connects two communicating virtual machines. The component may be set to operate in one of the four different quality levels of service:

**- Level 0:** an error-free transmission media.   
**- Level 1:** a transmission media which may lose frames.  
**- Level 2:** a transmission media which may damage frames (i.e., generating checksum-errors).   
**- Level 3:** a transmission media which may lose and damage frames.

**b. Virtual Machine**

This component simulates a communicating virtual machine. Internally, it is divided into two sub-components:

**b.1. Sliding Window Protocol**

### This component implements the sliding window protocol (i.e., the data link layer). In this simulated system, this component cannot work alone, and must interact with the Sliding Window Environment component in order to fetch/deliver packets from/to the upper network layer, and to fetch/deliver frames from/to the lower physical layer.

### **b.2. Sliding Window Protocol**

This component provides the environment in which the sliding window protocol component is working. Basically, this component implements the following interfaces:

* The interface between the **data link** layer and the **network** layer.
* The interface between the **data link** layer and the **physical** layer.
* The interface between the **data link** layer and the **underlying event queue**.

It should be pointed out that the Network Simulator component is running in one process, and the Virtual Machine component (including both the Sliding Window Protocol and the Sliding Window Environment) is running in another process. To simulate the communication between two virtual machines, two Virtual Machine processes must be executed.

# 

# II. Tasks

To implement the Sliding Window Protocol component (i.e., the data link-layer) of the simulated communication system. This component must implement all the features in the sliding window protocol, including:

## **a. Full-Duplex Data Communication**

## Full-Duplex Data Communication means that it is a bi-directional communication whereby data can be transmitted in both directions simultaneously in a single circuit.

Sending and receiving data **at the same time**

**VMach 2**

**VMach 1**

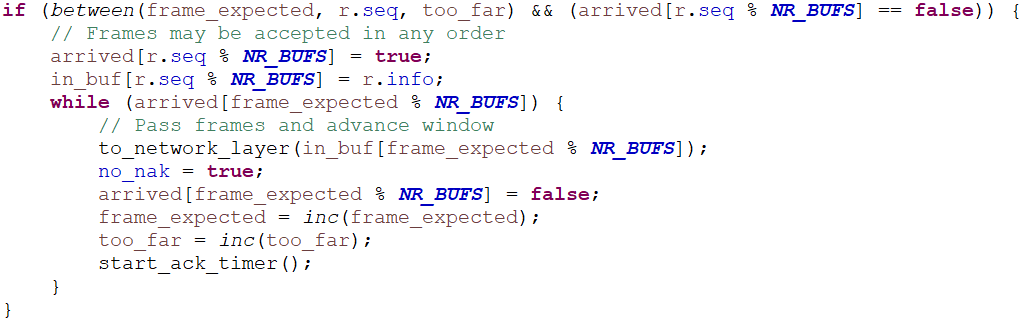
To implement bi-directional transmission, it can be achieve by placing the receiver and sender into a single **Protocol 6** function, instead of two separate functions of receiver and sender. In addition, the **send\_frame()** function is implemented separately so as to re-invoke in other parts of the code.

## **b. In-Order Delivery of Packets to the Network-Layer**

## A sequence number associated with each transmitted frame is introduced so that it can ensure that the packet delivered are in-order. The sequence numbers are **consecutive**, ranging from 0 to (2n -1) **circularly**. The sender will have to maintain the sending window opened, and the receiver will have to maintain the receiving windows with the expected sequence number of frames to be received. After the receiver received the sequence number (which is the **same** as the one of the frame), it will send an acknowledgement back to the sender, causing the current window to close, and hence open the next sending window.

## 

***Figure 1: If data frame arrived is out-of-order, send NAK***

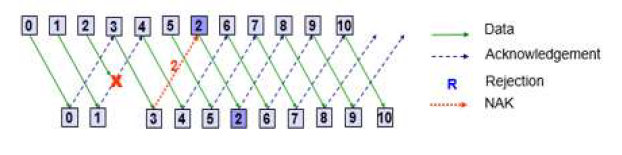
  
***Figure 2: If data frame arrived is in-order, transmit buffered frames to network layer***

## **c. Selective Repeat Retransmission Strategy**

## To prevent the sender from being blocked during the period of waiting for transmission and processing, pipelining solution was hence introduced. It can be done by having to set a large maximum sender’s window size, hence allowing the sender to transmit multiple frames until the acknowledgement for the first frame comes back, which will then make sure that the sender will always get the permission to transmit. However, the introduction of the pipelining solution raised concerns with regards to damaged/lost frame in the middle of the long stream. Thus, **selective repeat retransmission strategy** was introduced for error recovery of lost/damaged frames without the need to discard any frames.

## When a frame is lost/damaged, the receiver will ***buffer(in\_buf[])*** the correct subsequent frames. After receiving NAK (or timeout), the sender will retransmit the lost/damaged frame and the acknowledgement is done for the last frame buffered. The **selective repeat transmission strategy** is implemented in

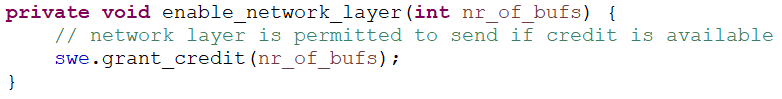
## Protocol 6 method.



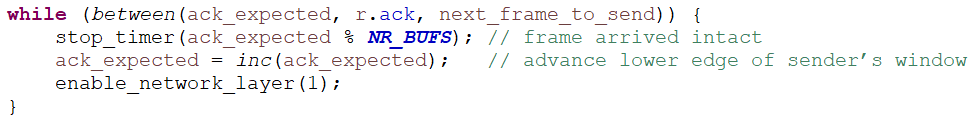
***Figure 3: Selective Repeat strategy as given in CZ3006 lecture slides***

## **d. Synchronisation with the Network-Layer by Granting Credits**

## The sender will have to synchronise with the network layer as it does not always have data to transmit. The sender will inform the data-link layer that it have to send frames, hence an event is generated. The data-link layer controls the network layer by preventing the network layer from sending more events until the network layer have the **credit** (the number of available buffer in the data-link layer). Additionally, the number of credits granted will be the same size as the size of the receiver’s window. To implement synchronization, we will use **enable\_network\_layer(NR\_BUFS)** to give credits to the network layer. Credit will be incremented and granted to the network layer upon receiving **complete** and **undamaged** frame.



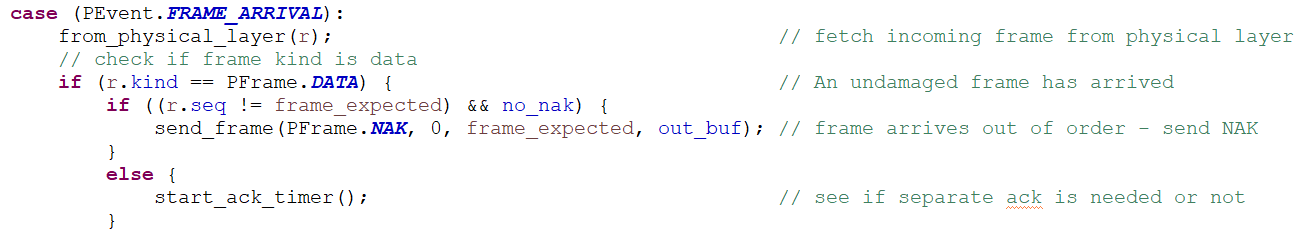
***Figure 4: Giving credits to network layer***



***Figure 5: Credit incremented upon receiving complete and undamaged frame***

## **e. Negative Acknowledgement**

## In cases where there are any lost/damaged frames that needs to be retransmitted, a **negative acknowledgement (NAK)** is sent by the receiver to the sender. Boolean variable **no\_nak** was initially set to ***true***, indicating that no NAK has been sent yet. Variable **no\_nak** will be set to ***false*** when a NAK has to be sent. NAK is sent when the frame arrives out-of-order or when the frame is lost/damaged.



***Figure 6: Send NAK if frame arrives lost/damaged (or out-of-order)***

## 

***Figure 7: Frame is lost/damaged***

## **f. Separate Acknowledgement when the Reverse Traffic is light or none**

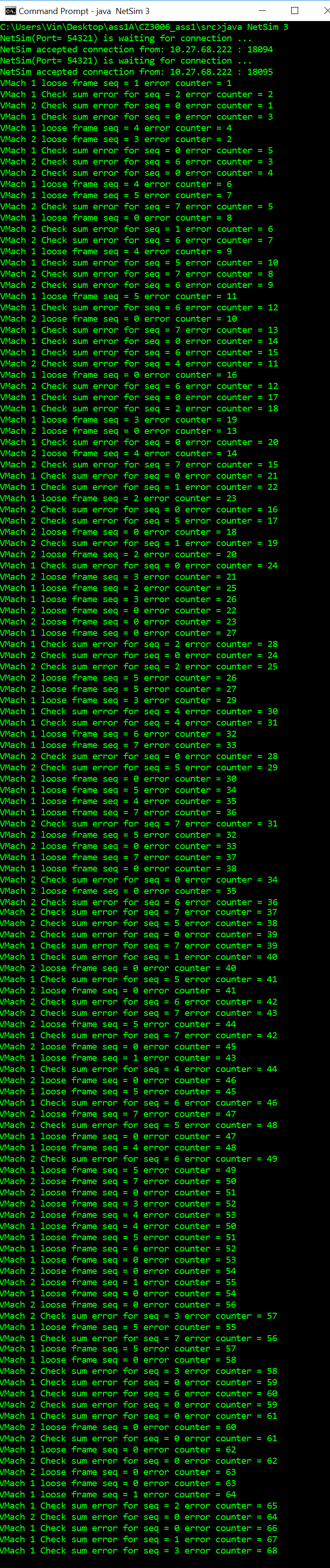
## When the last frame is received successfully by the receiver after a delay, a separate acknowledgement packet is sent. In order to implement this, an acknowledgement timer will be used to keep track of the waiting time for an acknowledgement to be sent. When acknowledgement timeout, it will send a separate acknowledgement packet.

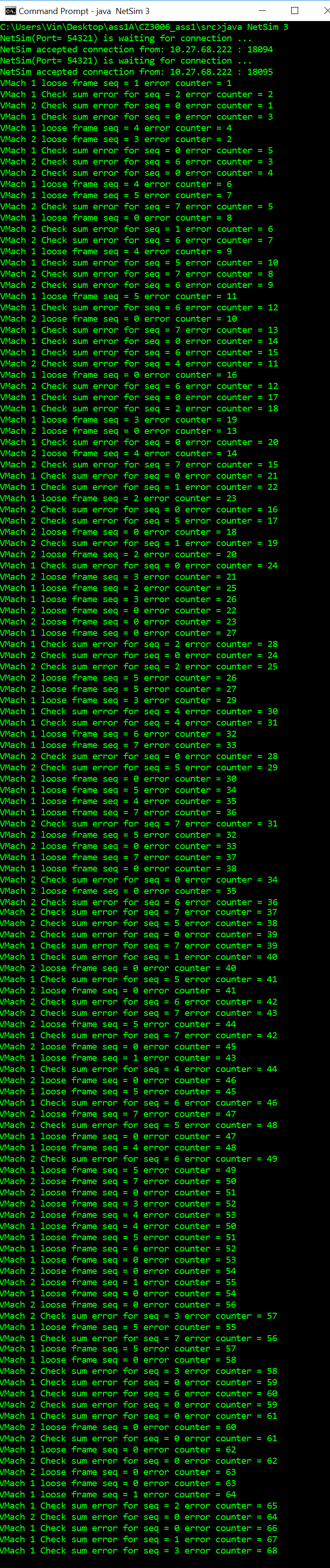
## 

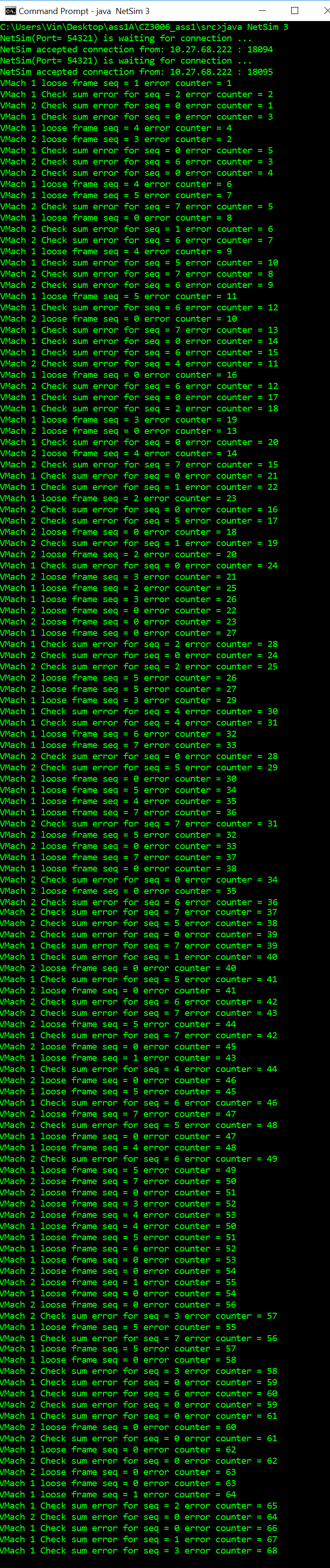
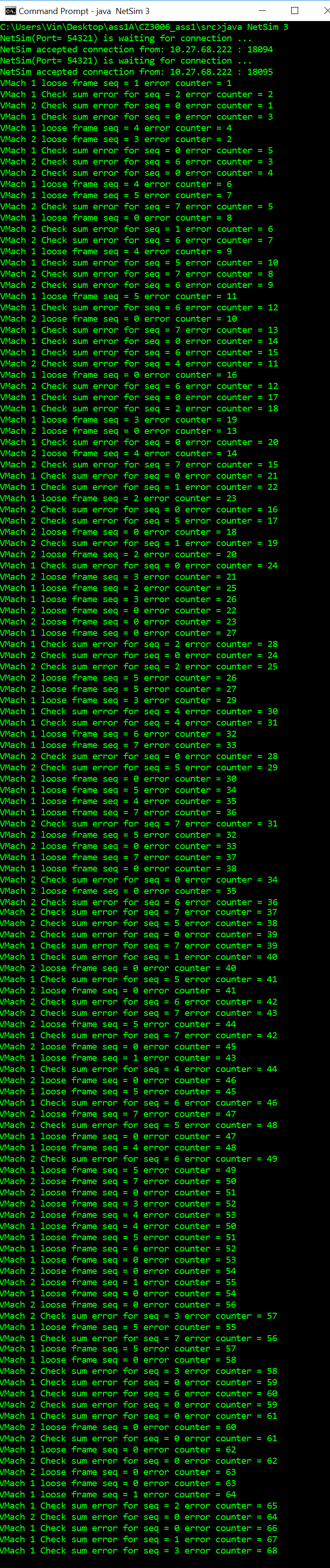
***Figure 8: It will send a separate acknowledgement packet when acknowledgement timeout***

## **g. Ability to withstand quality Level 3 of the Network Simulator component**

Example of **NetSim 3** cmd output:







# III. Java Source Code – Sliding Window Protocol

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

\* Author: ALVIN LEE YONG TECK \*

\* Matric No: U1620768F \*

\* Tutorial Group: SSP4 \*

\* Lab Assignment: Assignment 1 - Implementation of a Sliding Window Protocol \*

\* Course Code: CZ3006 NET CENTRIC COMPUTING \*

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

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

\* File: SWP.java \*

\* \*

\* This class implements the sliding window protocol \*

\* Used by VMach class \*

\* Uses the following classes: SWE, Packet, PFrame, PEvent, \*

\* \*

\* Author: Professor SUN Chengzheng \*

\* School of Computer Engineering \*

\* Nanyang Technological University \*

\* Singapore 639798 \*

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

**import** java.util.\*;

**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***];

// 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();

}

}

**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;

}

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

\* implement your Protocol Variables and Methods below:

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

// No nak has been sent yet

**private** **boolean** no\_nak = **true**;

// To check if the frame number is in the window (e.g. circular window range)

**private** **static** **boolean** between(**int** a, **int** b, **int** c)

{

// Same as between in protocol5, but shorter and more obscure

// the logic to check for Circular Window Range

**return** ((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a));

}

// Function is separated to re-invoke in other parts of the code

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

{

// Construct and send a data, ack, or nak frame

PFrame s = **new** PFrame(); // create a new frame for the sender to send

s.kind = frame\_kind; // frame\_kind can be equal to data, ack, or nak

// If the frame\_kind is data, take data from one of the buffer

**if** (frame\_kind == 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 the frame\_kind is nak, i.e. nak has been sent out (one nak per frame)

**if** (frame\_kind == PFrame.***NAK***) {

no\_nak = **false**; // one nak per frame, please

}

to\_physical\_layer(s); // sending the frame to receiver

**if** (frame\_kind == PFrame.***DATA***) {

start\_timer(frame\_nr); // start timer for sending of data

}

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

}

// To increase the frame number in a circular manner

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

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

**return** num;

}

**public** **void** protocol6() {

init();

**int** ack\_expected = 0; // lower edge of sender’s window

// next ack expected on the inbound stream

**int** next\_frame\_to\_send = 0; // upper edge of sender’s window + 1

// number of next outgoing frame

**int** frame\_expected = 0; // lower edge of receiver’s window

**int** too\_far = ***NR\_BUFS***; // upper edge of receiver’s window + 1

**int** i; // index into buffer pool

PFrame r = **new** PFrame(); // scratch variable

//Packet out\_buf[] = new Packet[NR\_BUFS]; // buffers for the outbound stream

Packet in\_buf[] = **new** Packet[***NR\_BUFS***]; // buffers for the inbound stream

**boolean** arrived[] = **new** **boolean**[***NR\_BUFS***]; // inbound bit map

//int nbuffered = 0; // how many output buffers currently used

// initially no packets are buffered

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

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

arrived[i] = **false**;

}

**while** (**true**) {

wait\_for\_event(event); // five possibilities: see event.type above

**switch** (event.type) {

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

\* PEvent.NETWORK\_LAYER\_READY

\* - Network layer have a packet to send

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

\*/

**case** (PEvent.***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(PFrame.***DATA***, next\_frame\_to\_send, frame\_expected, out\_buf); // transmit the frame

next\_frame\_to\_send = *inc*(next\_frame\_to\_send); // advance upper window edge

**break**;

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

\* PEvent.FRAME\_ARRIVAL

\* - A data or control frame has arrived

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

\*/

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

from\_physical\_layer(r); // fetch incoming frame from physical layer

// check if frame kind is data

**if** (r.kind == PFrame.***DATA***) { // An undamaged frame has arrived

**if** ((r.seq != frame\_expected) && no\_nak) {

// frame arrives out of order - send NAK  
 send\_frame(PFrame.***NAK***, 0, frame\_expected, out\_buf); }

**else** {

start\_ack\_timer(); // see if separate ack is needed or not

}

// buffer not occupied

**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

// check if buffered frames are in-order

**while** (arrived[frame\_expected % ***NR\_BUFS***]) {

// Pass frames and advance window

// pass to network layer if in-order

to\_network\_layer(in\_buf[frame\_expected % ***NR\_BUFS***]);

no\_nak = **true**; // no nak has been sent

arrived[frame\_expected % ***NR\_BUFS***] = **false**;

// advance lower edge of receiver’s window

frame\_expected = *inc*(frame\_expected);

too\_far = *inc*(too\_far); // advance upper edge of receiver’s window

start\_ack\_timer(); // see if a separate ack is needed or not

}

}

}

**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)) {

//nbuffered = nbuffered - 1; // handle piggybacked ack

stop\_timer(ack\_expected % ***NR\_BUFS***); // frame arrived intact

ack\_expected = *inc*(ack\_expected); // advance lower edge of sender’s window

enable\_network\_layer(1);

}

**break**;

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

\* PEvent.CKSUM\_ERR

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

\*/

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

**if** (no\_nak) {

// damaged frame (checking if NAK sent)

send\_frame(PFrame.***NAK***, 0, frame\_expected, out\_buf);

}

**break**;

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

\* PEvent.TIMEOUT

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

\*/

**case** (PEvent.***TIMEOUT***):

send\_frame(PFrame.***DATA***, oldest\_frame, frame\_expected, out\_buf); // timed out

**break**;

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

\* PEvent.ACK\_TIMEOUT

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

\*/

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

send\_frame(PFrame.***ACK***,0,frame\_expected, out\_buf); // ack timer expired, send a separate ack

**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,

\*/

**private** Timer[] clk\_timer = **new** Timer[***NR\_BUFS***];

**private** Timer ack\_timer = **new** Timer();

**private** **static** **final** **int** ***ack\_delay*** = 250; // delay for ACK waiting for outgoing frame

**private** **static** **final** **int** ***delay*** = 500; // delay for ACK

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

stop\_timer(seq);

clk\_timer[seq % ***NR\_BUFS***] = **new** Timer();

clk\_timer[seq % ***NR\_BUFS***].schedule(**new** TempTimerTask(seq), ***delay***);

}

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

**try**{

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

}

**catch** (Exception e){

}

}

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

stop\_ack\_timer();

ack\_timer = **new** Timer();

ack\_timer.schedule(**new** TimerTask() {

**public** **void** run() {

swe.generate\_acktimeout\_event();

}

}, ***ack\_delay***);

}

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

**try** {

ack\_timer.cancel();

}

**catch** (Exception e){

}

}

**private** **class** TempTimerTask **extends** TimerTask{

**public** **int** seq; // keeps track of sequence number

**public** TempTimerTask (**int** seq){

**super**();

**this**.seq = seq;

}

**public** **void** run() {

swe.generate\_timeout\_event(**this**.seq); // timeout event

}

}

}// End of class

/\*

\* Note: In class SWE, the following two public methods are available: .

\* generate\_acktimeout\_event() and . generate\_timeout\_event(seqnr).

\*

\* To call these two methods (for implementing timers), the "swe" object should

\* be referred as follows: swe.generate\_acktimeout\_event(), or

\* swe.generate\_timeout\_event(seqnr).

\*/