MINI PROJECT REPORT

Submitted to the faculty of Engineering and Technology
B.Tech CSE
VI Semester
(Autonomous Batch)

SLIDING WINDOW PROTOCOL
(SELECTIVE REPEAT)

By
A.Bhuvana Chandra - B14CS010

Under the Guidance of
S.Naga Raju, Assoc. Professor

Department of Computer Science & Engineering
Kakatiya Institute of Technology & Science,
WARANGAL (A.P)
2016-2017

CERTIFICATE

This is to certify that ……

of the B.Tech. VI Sem, Computer science and Engineering(Autonomous) has

Satisfactorily completed the dissertation work entitled " ……

……." in the partial

fulfillment of the requirements of B.Tech. VI Sem during this academic year

2016-17.

Prof S. Naga Raju
Supervisor

Dr. P. Niranjan Reddy
Head of the Department

ACKNOWLEDGEMENT

I extend our sincere and heartfelt thanks to our esteemed ***guide*** & **Co-ordinator**, ***Mr.Naga Raju.S*** and for his exemplary guidance, monitoring and constant encouragement throughout the course at crucial junctures and for showing us the right way.

I would like to extend thanks to our respected ***Head of the department,***

Dr. P.Niranjan Reddy for allowing us to use the facilities available. We would like to thank other faculty members also .Last but not the least,

I would like to thank our friends and family for the support and encouragement they have given us during the course of our work.

ATCHE BHUVANA CHANDRA

ABSTRACT

Selective Repeat is part of the automatic repeat-request (ARQ). With selective repeat, the sender sends a number of frames specified by a window size even without the need to wait for individual ACK from the receiver as in Go-Back-N ARQ. The receiver may selectively reject a single frame, which may be retransmitted alone; this contrasts with other forms of ARQ, which must send every frame from that point again. The receiver accepts out-of-order frames and buffers them. The sender individually retransmits frames that have timed out.

The receiver process keeps track of the sequence number of the earliest frame it has not received, and sends that number with every acknowledgement (ACK) it sends. If a frame from the sender does not reach the receiver, the sender continues to send subsequent frames until it has emptied its window. The receiver continues to fill its receiving window with the subsequent frames, replying each time with an ACK containing the sequence number of the earliest missing frame. Once the sender has sent all the frames in its window, it re-sends the frame number given by the ACKs, and then continues where it left off.

The size of the sending and receiving windows must be equal, and half the maximum sequence number (assuming that sequence numbers are numbered from 0 to n−1) to avoid miscommunication in all cases of packets being dropped. To understand this, consider the case when all ACKs are destroyed. If the receiving window is larger than half the maximum sequence number, some, possibly even all, of the packets that are resent after timeouts are duplicates that are not recognized as such. The sender moves its window for every packet that is acknowledged.

CONTENTS

ABBREVIATIONS

base - our sending base - the next expected Packet to be received

nextseqnum - the next sequence number that will be given to a newly created Packet

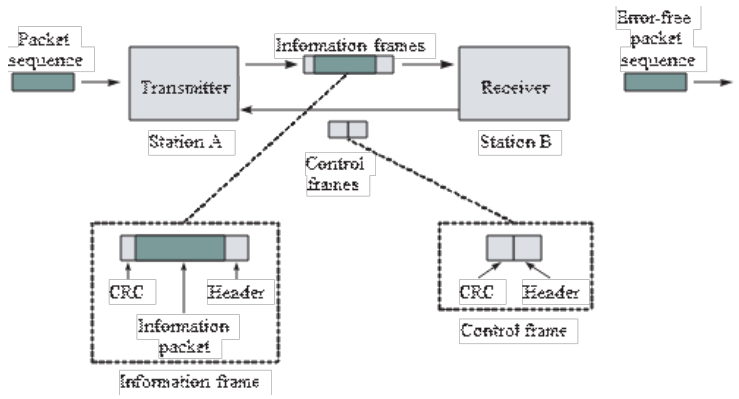
selected - the index of the currently selected Packet in transmission

lastKnownSucPacket - LAST KNOWN SUCcessful PACKET received by receiving node.

List of Figures

1. INTRODUCTION

Automatic Repeat reQuest (ARQ), also known as Automatic Repeat Query, is an error-control method for data transmission that uses acknowledgements (messages sent by the receiver indicating that it has correctly received a data frame or packet) and timeouts (specified periods of time allowed to elapse before an acknowledgment is to be received) to achieve reliable data transmission over an unreliable service. If the sender does not receive an acknowledgment before the timeout, it usually re-transmits the frame/packet until the sender receives an acknowledgment or exceeds a predefined number of re-transmissions.



(Figure 1) -arq architecture.

1.1. OBJECTIVE

When used as the protocol for the delivery of messages, the sending process continues to send a number of frames specified by a window size even after a frame loss. Unlike Go-Back-N ARQ, the receiving process will continue to accept and acknowledge frames sent after an initial error; this is the general case of the sliding window protocol with both transmit and receive window sizes greater than 1.

The receiver process keeps track of the sequence number of the earliest frame it has not received, and sends that number with every acknowledgement (ACK) it sends. If a frame from the sender does not reach

the receiver, the sender continues to send subsequent frames until it has emptied its window. The receiver continues to fill its receiving window with the subsequent frames, replying each time with an ACK containing the sequence number of the earliest missing frame. Once the sender has sent

all the frames in its window, it re-sends the frame number given by the ACKs, and then continues where it left off.

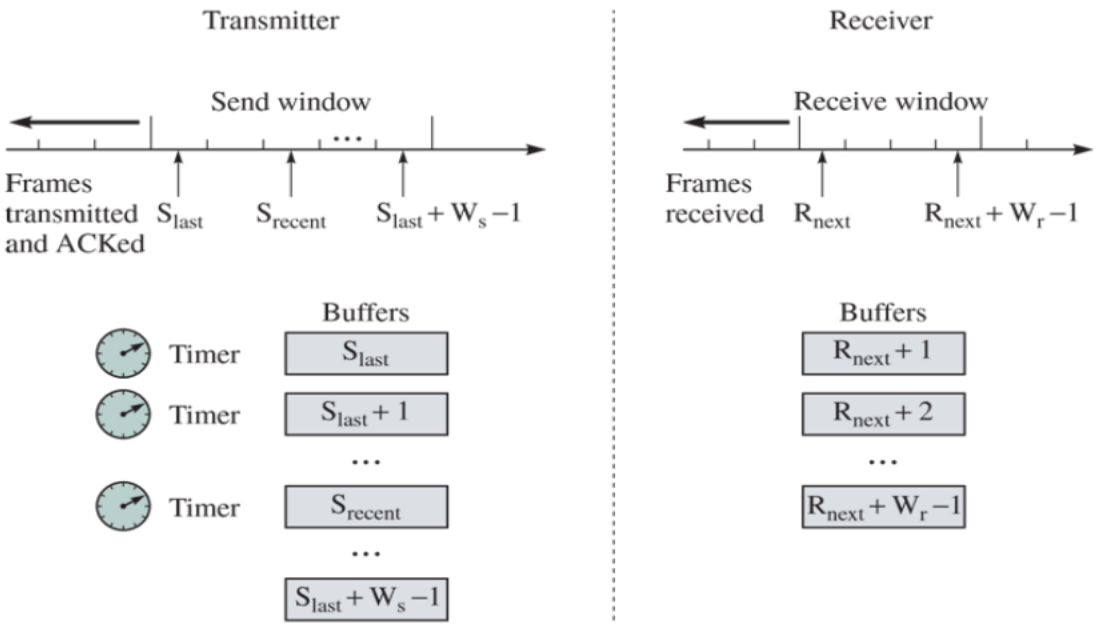
The size of the sending and receiving windows must be equal, and half the maximum sequence number (assuming that sequence numbers are numbered from 0 to n−1) to avoid miscommunication in all cases of packets being dropped. To understand this, consider the case when all ACKs are destroyed. If the receiving window is larger than half the maximum sequence number, some, possibly even all, of the packets that are resent after timeouts are duplicates that are not recognized as such. The sender moves its window for every packet that is acknowledged.[1]

When used as the protocol for the delivery of subdivided messages it works somewhat differently. In non-continuous channels where messages may be variable in length, standard ARQ or Hybrid ARQ protocols may treat the message as a single unit. Alternately selective retransmission may be employed in conjunction with the basic ARQ mechanism where the message is first subdivided into sub-blocks (typically of fixed length) in a process called packet segmentation. The original variable length message is thus represented as a concatenation of a variable number of sub-blocks. While in standard ARQ the message as a whole is either acknowledged (ACKed) or negatively acknowledged (NAKed), in ARQ with selective transmission the ACK response would additionally carry a bit flag indicating the identity of each sub-block successfully received. In ARQ with selective retransmission of sub-divided messages each retransmission diminishes in length, needing to only

contain the sub-blocks that were linked.

In most channel models with variable length messages, the probability of error -free reception diminishes in inverse proportion with increasing message length. In other words it's easier to receive a short message than a longer message. Therefore standard ARQ techniques involving variable length messages have increased difficulty delivering longer messages, as each repeat is the full length. Selective re-transmission

applied to variable length messages completely eliminates the difficulty in delivering longer messages, as successfully delivered sub-blocks are retained after each transmission, and the number of outstanding sub-blocks in following transmissions diminishes. Selective Repeat is implemented in UDP transmission.



(figure 2) – buffers and sender receiver timer variables

1.2. SYSTEM SCOPE

Scope of the software:-

Secure and sequential end to end transmission.

Shows the lost packets.

Socket transmission in lan network.

2. SYSTEM ANALYSIS

2.1 WORKING

Selective Repeat attempts to retransmit only those packets that are actually lost (due to errors)

– Receiver must be able to accept packets out of order

– Since receiver must release packets to higher layer in order, the receiver must be able to buffer some packets

• Retransmission requests

– Implicit The receiver acknowledges every good packet, packets that are not ACKed before a time-out are assumed lost or in error Notice that this approach must be used to be sure that every packet is eventually received

– Explicit An explicit NAK (selective reject) can request retransmission of just one packet This approach can expedite the retransmission but is not strictly needed – One or both approaches are used in practice.

2.2. RULES

• Window protocol just like GO Back N – Window size W.

• Packets are numbered Mod M where M >= 2W.

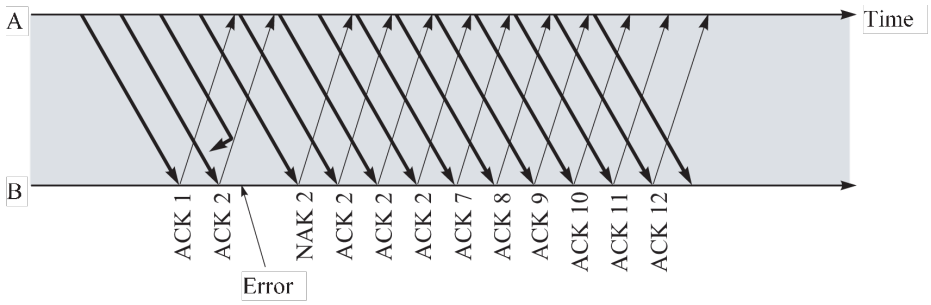
• Sender can transmit new packets as long as their number is with W of all unACKed packets.

• Sender retransmit un-ACKed packets after a timeout

– Or upon a NAK if NAK is employed.

• Receiver ACKs all correct packets.

• Receiver stores correct packets until they can be delivered in order to the higher layer.



(figure 3)-transmission diagram

2.3. NEED FOR BUFFERING

• Sender must buffer all packets until they are ACKed – Up to W un-ACKed packet are possible.

• Receiver must buffer packets until they can be delivered in order

– I.e., until all lower numbered packets have been received

– Needed for orderly delivery of packets to the higher layer

– Up to W packets may have to be buffered (in the event that the first packet of a window is lost).

• Implication of buffer size = W – Number of un-ACKed packets at sender ≤ W Buffer limit at sender.

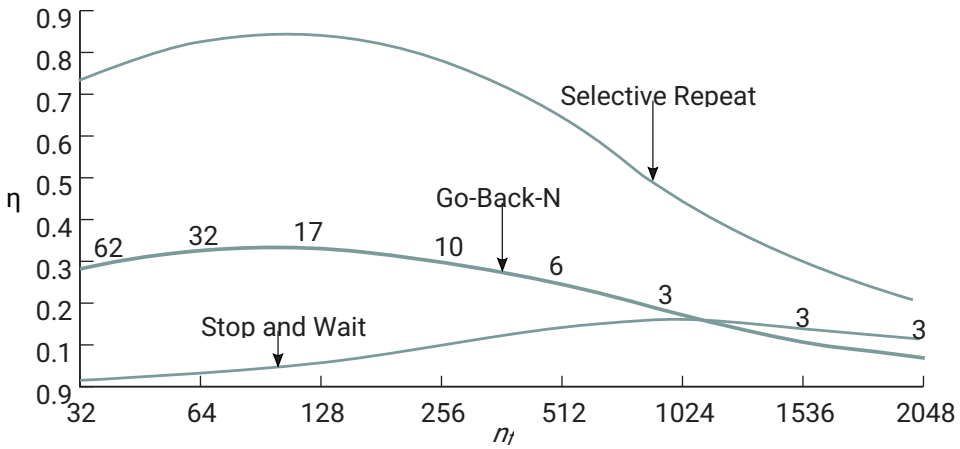
– Number of un-ACKed packets at sender cannot differ by more than W Buffer limit at the receiver (need to deliver packets in order).

– Packets must be numbered modulo M ≥ 2W (using log2(M) bits) .

2.4.EFFICIENCY

• For ideal SRP, only packets containing errors will be retransmitted – Ideal is not realistic because sometimes packets may have to be retransmitted because their window expired. However, if the window size is set to be much larger than the timeout value then this is unlikely.
• With ideal SRP, efficiency = 1 - P – P = probability of a packet error • Notice the difference with Go Back N where
efficiency (Go Back N) = 1/(1 + N*P/(1-P))
• When the window size is small performance is about the same, however with a large window SRP is much better – As transmission rates increase we need larger windows and hence the increased use of SRP.

(figure 4)- efficiency graph



2.5. HARDWARE & SOFTWARE REQUIREMENT

- Processor - Pentium iv and above
- Ram - 256 MB and above
- Hard disk - 1GB or above

Programming Language - Java

Operating System	-	Windows 8
IDE	-	Netbeans

3. SYSTEM DESIGN

3.1. FLOW OF PROCESS

MAIN PROCESS:

2 threads run for the applet gbnTread - runs to create our animation and process Packets timerThread - created and sleeps for a specified period of time. On wake up performs timeout processing A timeout causes all of the outstanding Packets to be re-transmitted. NOTE: The text(Computer Networking: A Top Down Approach) specified a per Packet timer, however this is rarely implemented as there is a significant overhead in using that many timers. Logically, the only Packet that would ever timeout is

the left most edge of the sending window as this has been in transmission the longest. Since a per Packet timer system is not implemented in practice we have simulated per Packet timers per the books description while using only a single timer.

SEQUENCE OF METHODS :

*actionPerformed
*check_upto_n
*deliverBuffer
*init
*mouseDown
*onTheWay
*paint
*reset_app
*retransmitOutstandingPackets
*run
*setupSimulationParams
*start
*update

PacketTimerTask:
*cancelTimer
*pause
*resume
*startTimer

Method init

Purpose: init method to set up applet for running - first method called on loading the code. Attempts to load parameters passed from HTML code contained in the website. If there is an error or no parameters are provided then the default values(declared above) are used. Global variables used: sender - array holding the Packets and the corresponding

acks for the Packets sent in the applet output - console window for applet activities & messages

Method mouseDown

Purpose: Determines when the mouse is pressed down and what object(Packet) is currently under the mouse. mouseDown is used to select a Packet in transmission to be killed(possibly) Global variables used: sender - array holding the Packets and the corresponding acks for the Packets sent in the applet output - console window to display information about the applet activities

Method Start

Purpose: Start method required for implementing multi-threading. Start is the first method called by a thread after creation. Procedures Calling: run Procedures Called: run Global Variables Used: gbnThread - creates new thread for first execution and starts thread(calling run method of thread)

Method run

Purpose: Run method required by runnable interface. Determines which thread is

calling and process accordingly. gbnThread produces the animation for the applet. The timerThread sleeps until timeout processing is needed to retransmit the sending window.

procedures/Functions Called: check_upto_n, paint/update(indirectly)

Procedures/Functions Calling: main, start Local variables: currentThread - holds the identifier for the currently executing thread i - temporary variable used for loop control Global variables used: sender - array holding the Packets and the corresponding acks for the Packets sent in the applet output - console window to display information about the applet activities.

lastKnownSucPacket - holds the number of the last successful Packet to arrive gbnThread - thread to advance animation.

Method deliverBuffer

Purpose: Handles the delivery of buffered packets at the receiver.

calling: run Procedures called: none Global variables used: sender[] - access packet information Local variables used: j, k, l - loop control variables PacketNumber - process up to this index in sender

Method retransmitOutstandingPacket

Purpose: handles transmission of a packet when a timeout occurs Procedures

calling: run(called by timerThread) Procedures called: none Global variables used: sender[] - to set up params for retransmission GBNThread - set animation thread for retransmission
output - output messages to user about retransmission base - number of left-most Packet in the sending window Local variables: n - used as loop control variable

Method setupSimulationParams

Purpose: Extract simulation parameters from the HTML page the applet is being executed from. If the parameter is supplied convert to value to integer and check for greater than 0(less than 0 will throw exceptions) if the value supplied is in range, assign that value to the simulation parameter Global variables used: window_len,pack_widt, pack_height, h_offset, v_offset, v_clearance, total_Packet, time_out_sec.

Method Update

Purpose: Handles the actual drawing for the applet. Draws the Packets, message boxes, ... Procedures/Functions Called:check_upto_n, paint/update(indirectly) Procedures/Functions Calling: paint Local variables: i - temporary variable used for loop control Global variables used: sender - array holding the Packets and the corresponding acks for the Packets sent in the applet offGraphics - used to create a secondary buffer to draw the necessary components before putting the completed

drawing to screen. This prevents "flashing" when viewing the applet on higher frame rates

Method reset_app

Purpose: resets the applet to its initial state to allow for a second run without reloading the webpage Local variables: i - temporary variable used for loop control Global variables used: sender - array holding the Packets and the corresponding acks for the Packets sent in the applet base - what number our sending window is set to nextseq - the next sequence number that can be used for a Packet selected - the Packet currently selected fps - how fast shoud the animation run timerFlag - gbnThread - used to process and display the animation timerThread - used to handle timeouts and retransmit the sending window

Method actionPerformed

Purpose: actionPerformed method required to be an action listener class. Determines which button in the animation is pressed (ie send new, stop animation, kill Packet/ack, ...) Procedures/Functions Called: paint/update i - temporary variable used for loop control Global variables used: sender - array holding the Packets and the corresponding acks for the Packets sent in the applet nextSeq - the next unused

sequence number for a Packet

Method onTheWay

Purpose: checks to see if all of the Packets in an array(in our case the sender array) have been created and are being processed Procedures/Functions Calling: run Local variables: i - temporary variable used for loop control

Method check_upto_n

Purpose: checks the sender array to see if all of the pacekts up to index packno have reached thier destination Procedures/Functions Calling: run Local variables: i - temporary variable used for loop control Global variables used: sender - array holding the Packets and the corresponding acks for the Packets sent in the applet.

Method cancelTimer

Purpose: cancels the current packet retransmission timeout task from Local variables: _cancel - bool run method uses to determine whether the retransmission of the given packet (located in the selectiverepeatpacket in the sender[] array) should occur

Gloabal variables - none

Method startTimer

Purpose: designates that the currently scheduled packet retransmission timeout method should be executed. This retransmission method will execute after the scheduled timeout period has elapsed.

Local variables: _cancel - bool run method uses to determine whether the retransmission of the given packet (located in the selectiverepeatpacket in the sender[] array) should occur

Gloabal variables – none.

ALGORITHM OF SENDING AND RECEIVING:

1)SENDER SIDE SELECTIVE REPEAT ALGORITHM

```
1  Sw = 2^(m-1) ;
2  Sf = 0;
3  Sn = 0;
4
5  while (true)                          //Repeat forever
6  {
7    WaitForEvent();
8    if(Event(RequestToSend))            //There is a packet to send
9    {
10      if(Sn-Sf >= Sw)                  //If window is full
11            Sleep();
12      GetData();
13      MakeFrame(Sn);
14      StoreFrame(Sn);
15      SendFrame(Sn);
16      Sn = Sn + 1;
17      StartTimer(Sn);
18   }
19
```

```
20   if(Event(ArrivalNotification)) //ACK arrives
21   {
22      Receive(frame);              //Receive ACK or NAK
23      if(corrupted(frame))
24           Sleep();
25      if (FrameType == NAK)
26         if (nakNo between Sf and Sn)
27         {
28          resend(nakNo);
29          StartTimer(nakNo);
30         }
31      if (FrameType == ACK)
32         if (ackNo between Sf and Sn)
33         {
34           while(sf < ackNo)
35           {
36            Purge(sf);
37            StopTimer(sf);
38            Sf = Sf + 1;
39           }
40         }
41   }
42
43   if(Event(TimeOut(t)))           //The timer expires
44   {
45    StartTimer(t);
46    SendFrame(t);
47   }
48 }
```

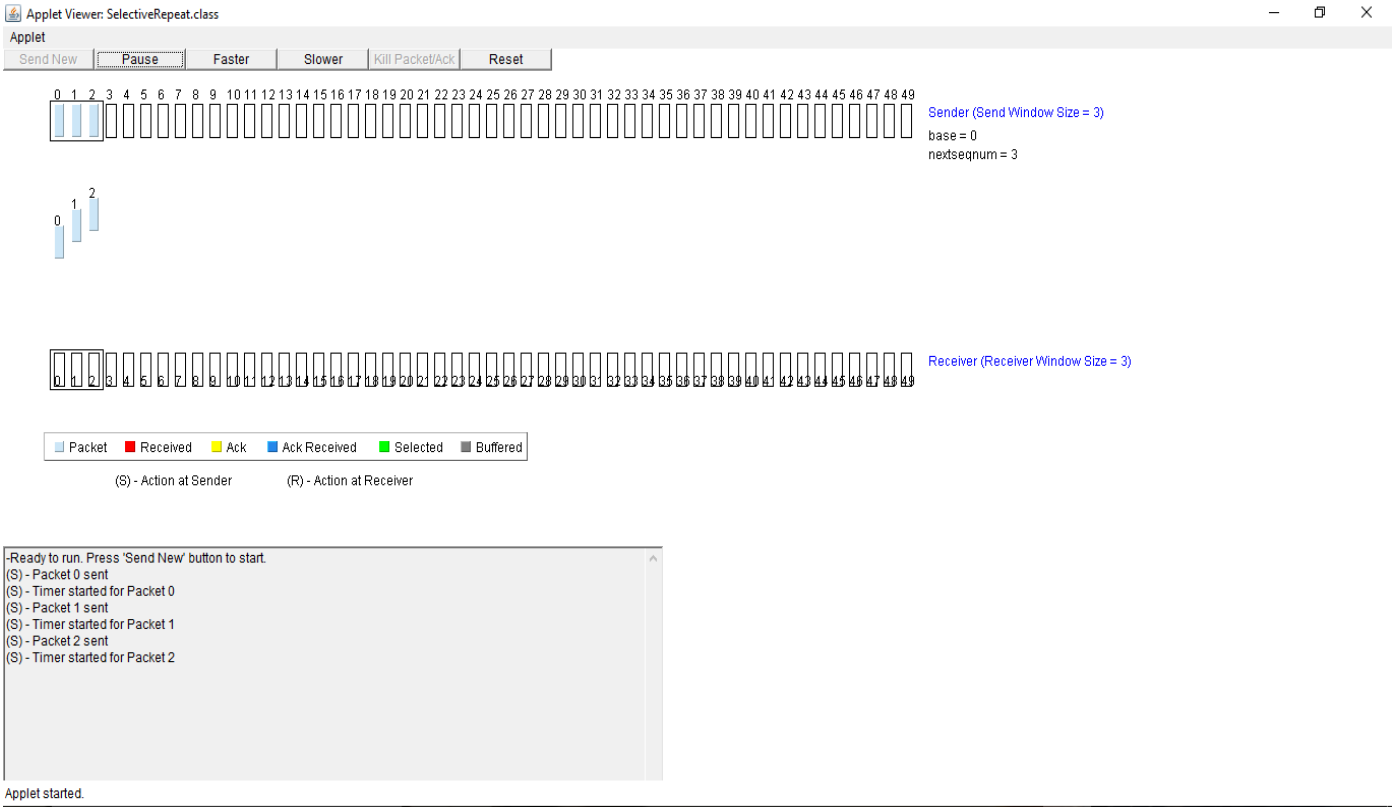
2) RECIEVER SIDE SELECTIVE REPEAT ALGORITHM

```
1  Rn = 0;
2  NakSent = false;
3  AckNeeded = false;
4  Repeat(for all slots)
5      Marked(slot) = false;
6
7  while (true)                              //Repeat forever
8  {
9    WaitForEvent();
10
11   if(Event(ArrivalNotification))          /Data frame arrives
12   {
13      Receive(Frame);
14      if(corrupted(Frame))&& (NOT NakSent)
15      {
16       SendNAK(Rn);
17       NakSent = true;
18       Sleep();
19      }
20      if(seqNo <> Rn)&& (NOT NakSent)
21      {
22       SendNAK(Rn);
```

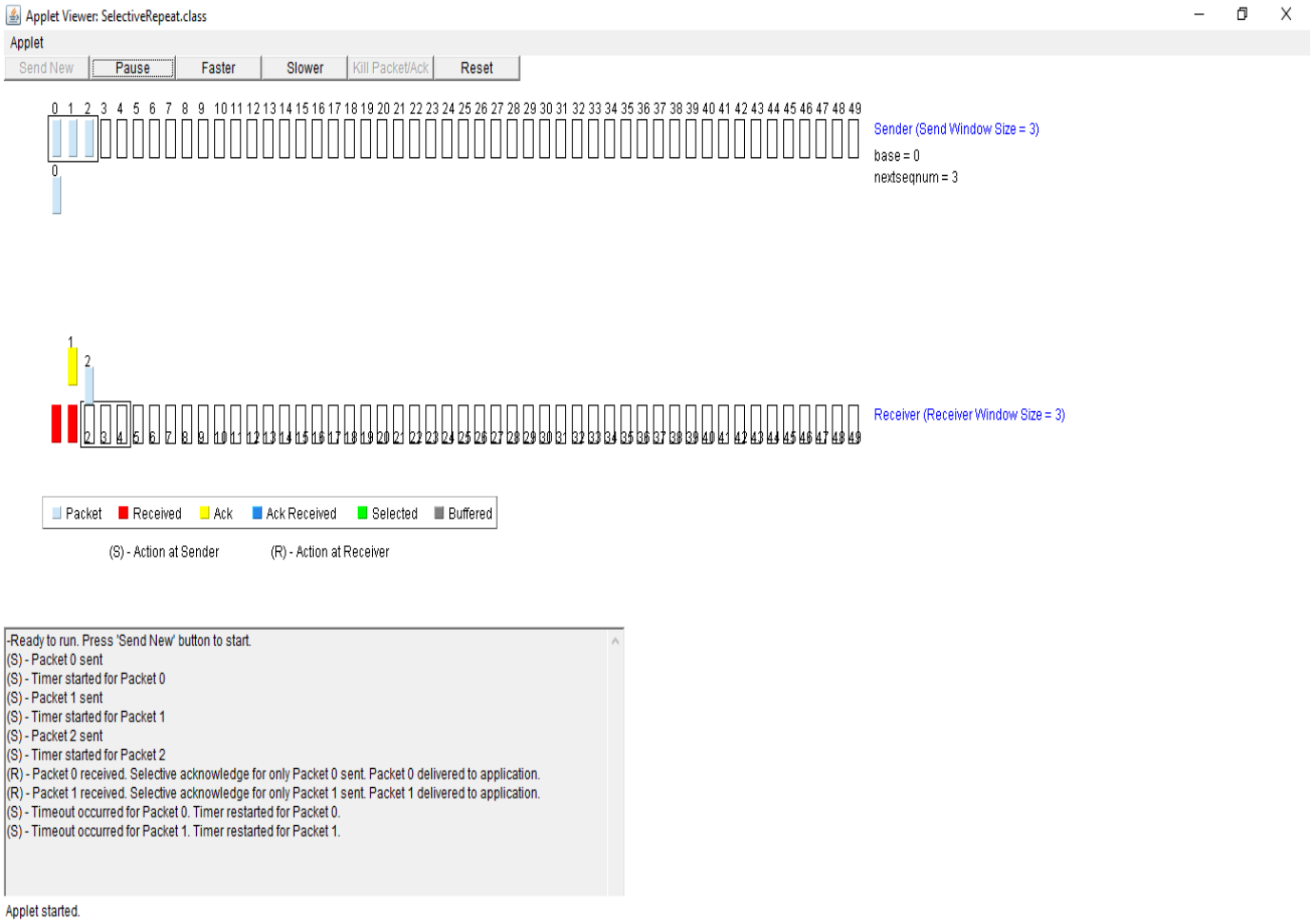
RESULT SCREEN SHOTS :

1) Applet at the initial stage.

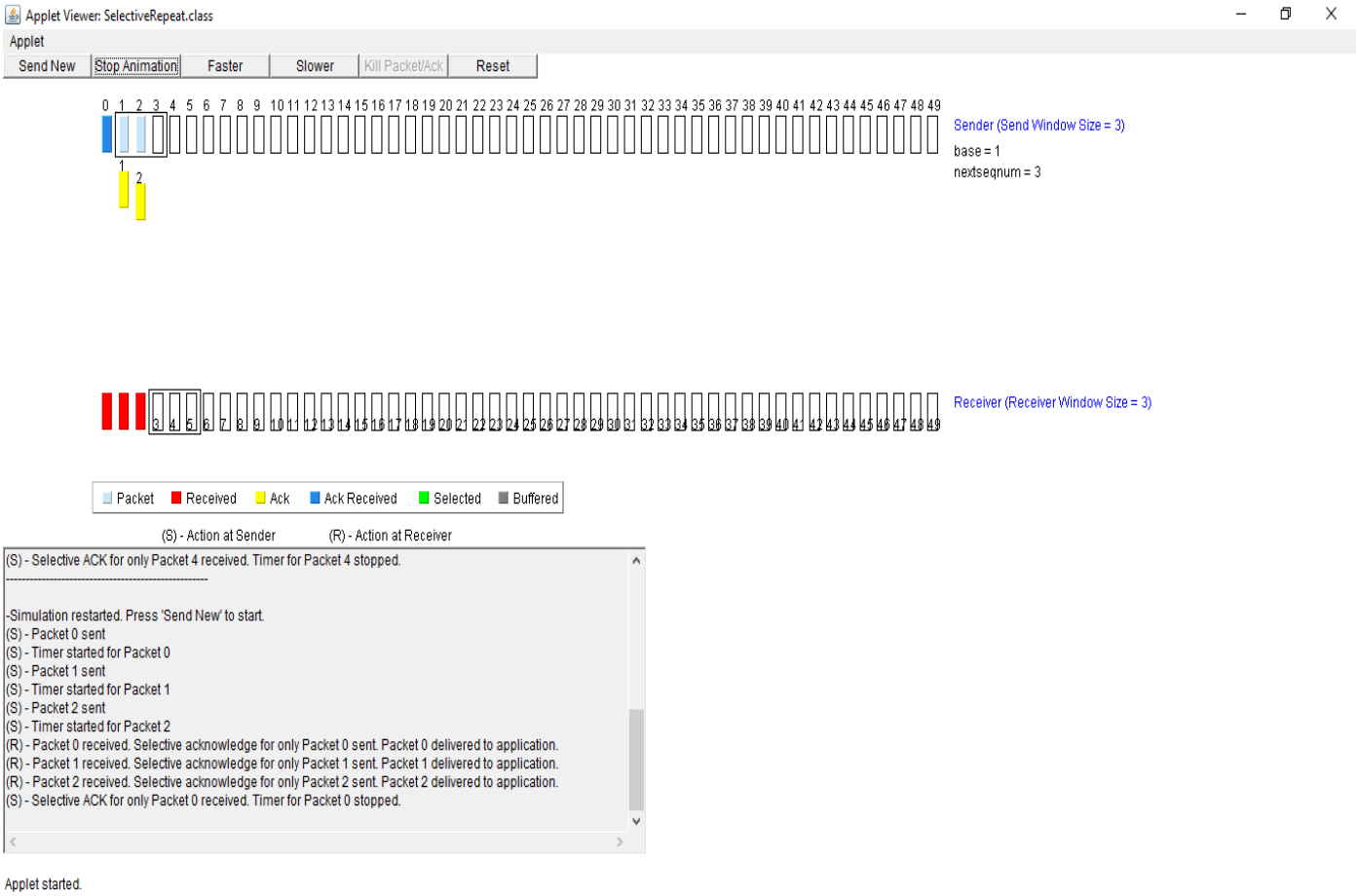
2) Packets have been transmitted.



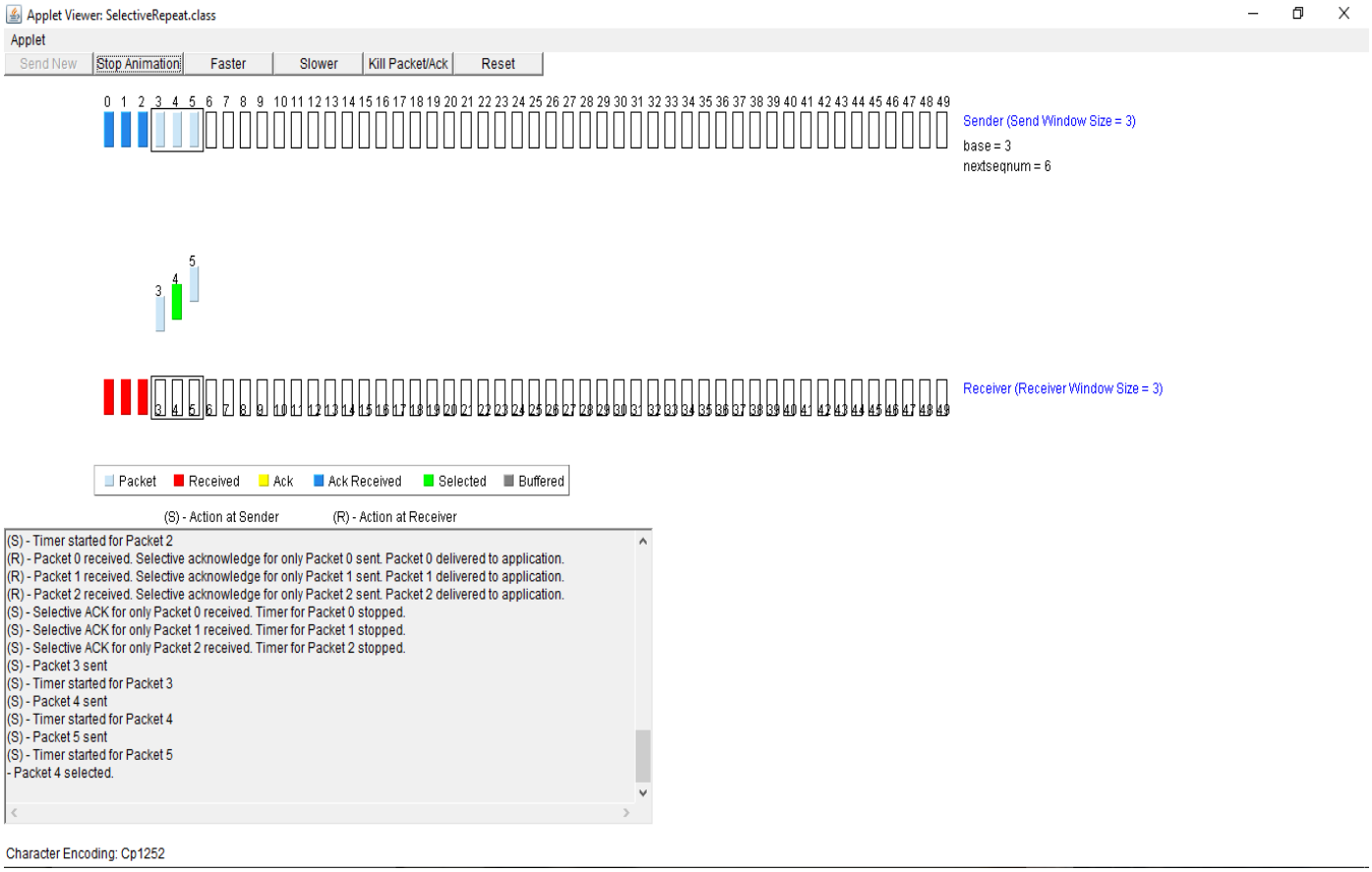
3) Packets received and acks being sent.



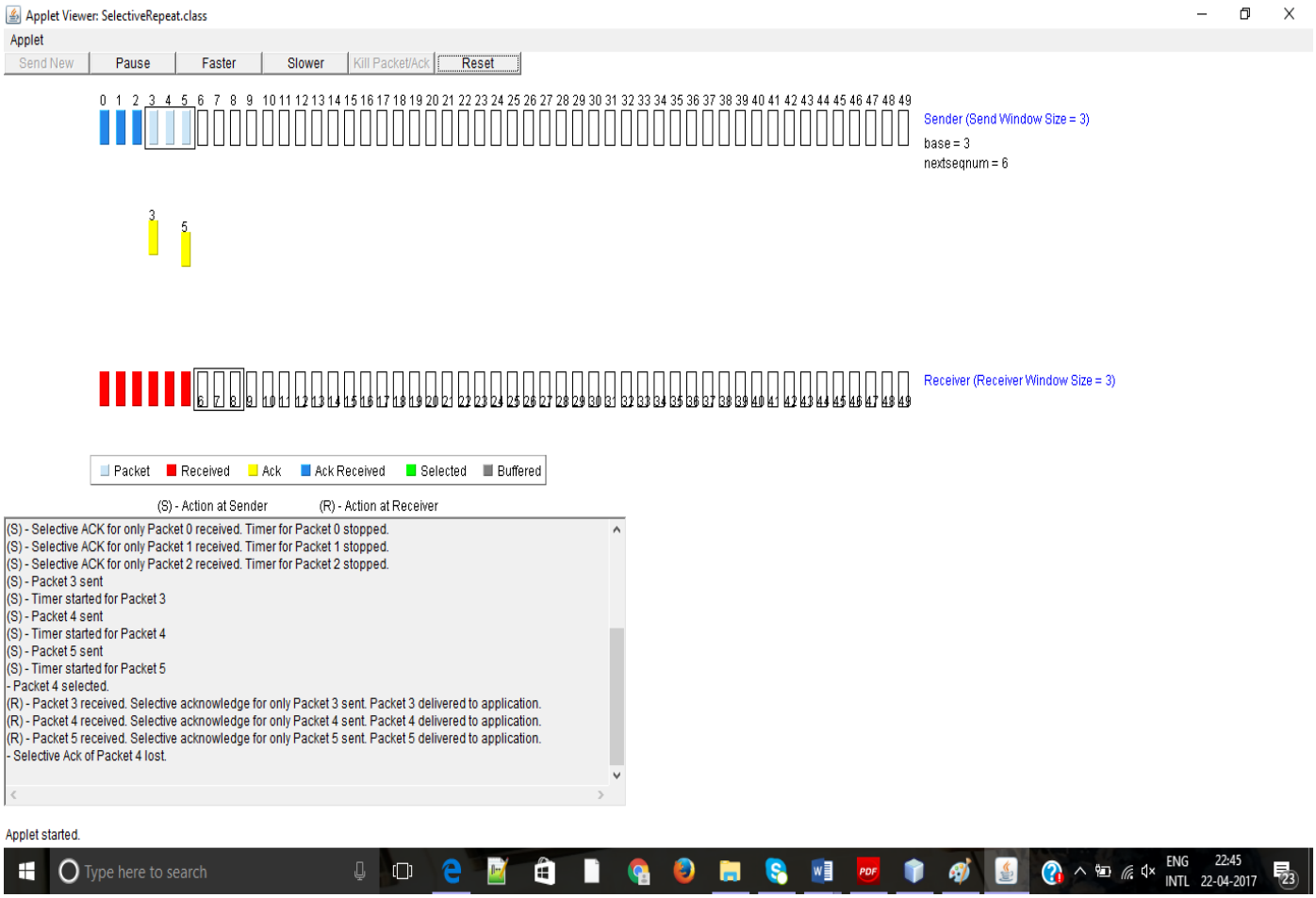
4) ack received and confirmation sent.



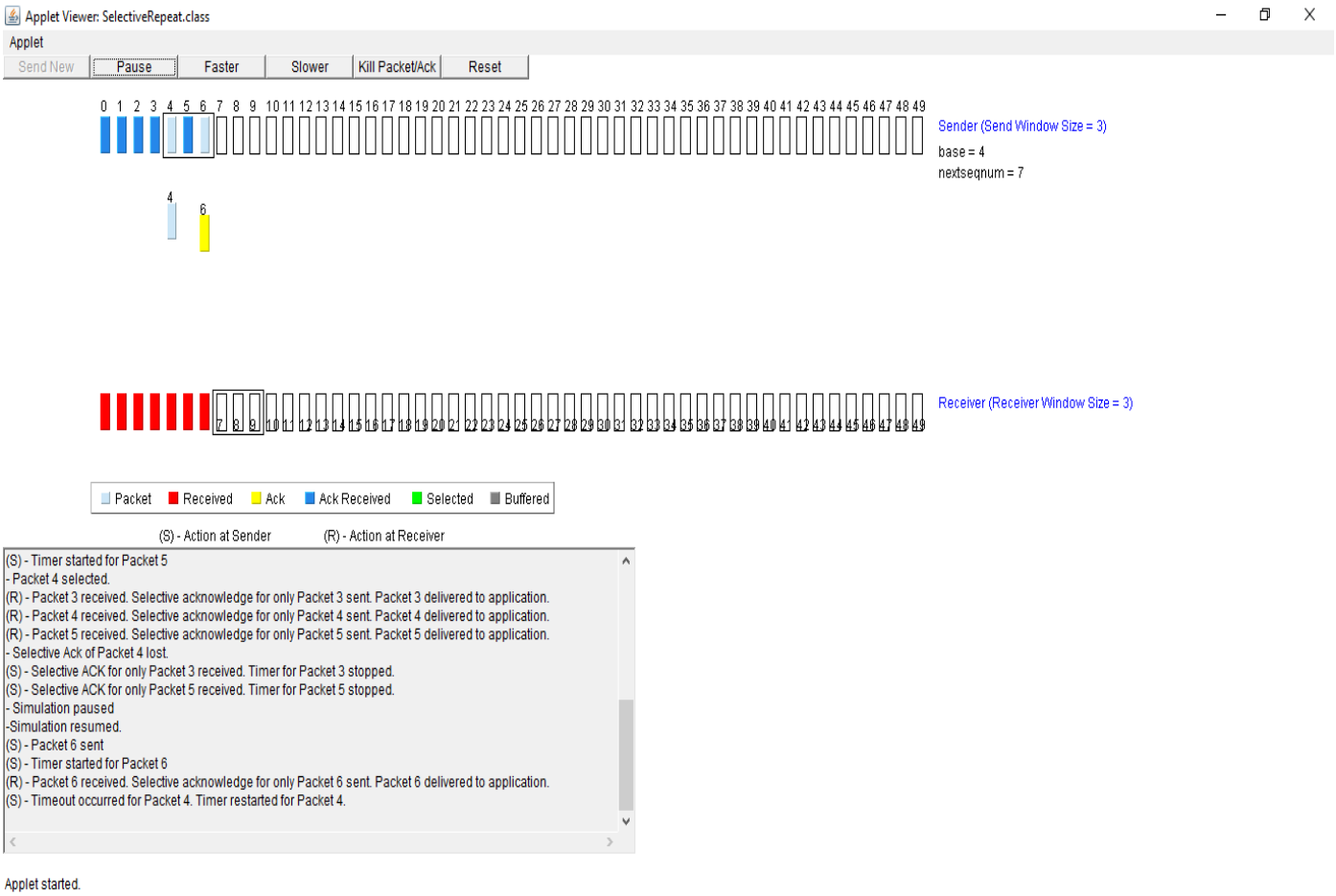
5) Packet is being selected to kill.



6)Packet is killed.

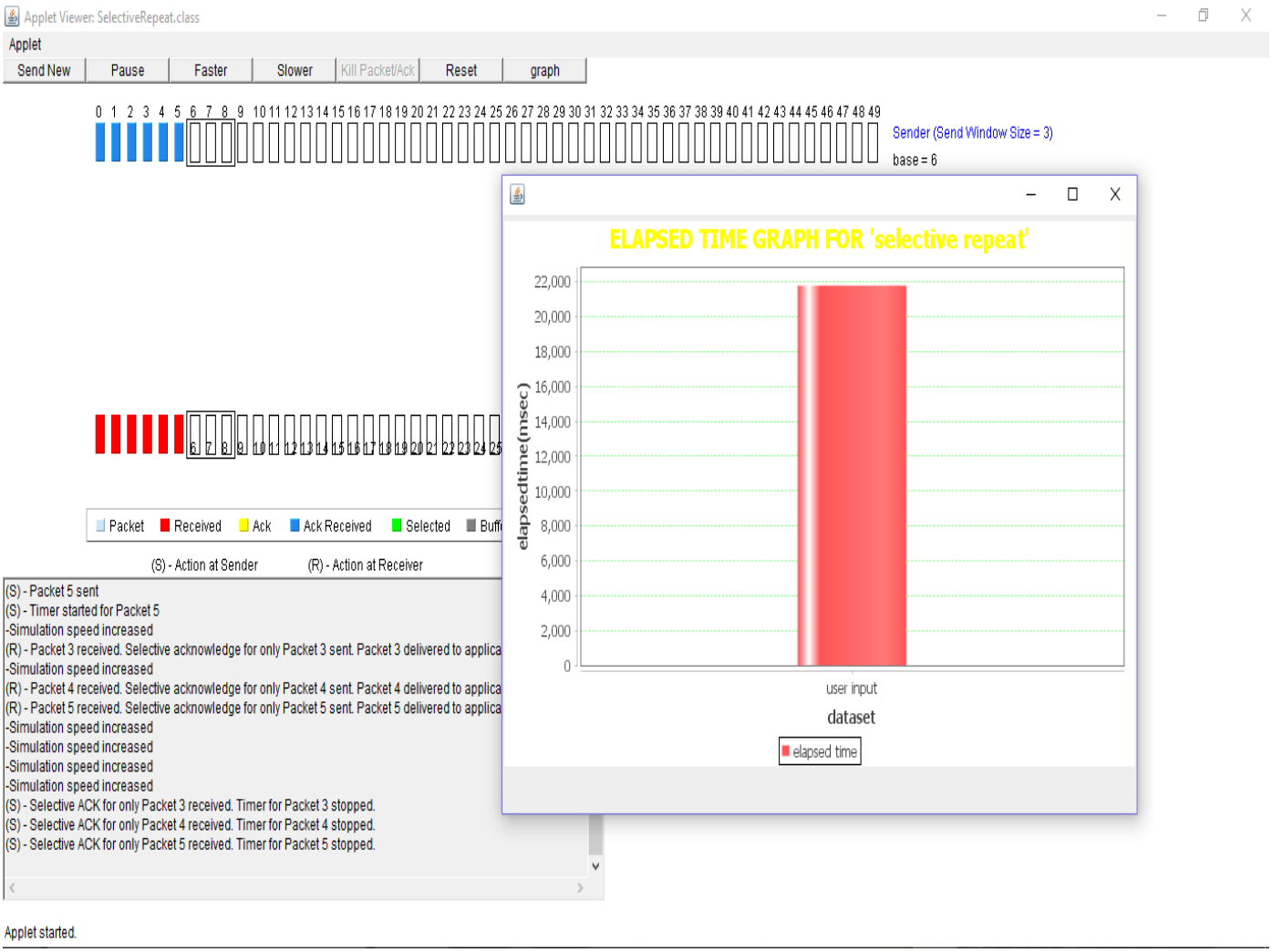

Applet Viewer: SelectiveRepeat.class
Applet
Send New
Pause
Faster
Slower
Kill Packet/Ack
Reset
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
Sender (Send Window Size = 3)
base = 3
nextseqnum = 6
3
5
Receiver (Receiver Window Size = 3)
Packet
Received
Ack
Ack Received
Selected
Buffered
(S) - Action at Sender
(R) - Action at Receiver
(S) - Selective ACK for only Packet 0 received. Timer for Packet 0 stopped.
(S) - Selective ACK for only Packet 1 received. Timer for Packet 1 stopped.
(S) - Selective ACK for only Packet 2 received. Timer for Packet 2 stopped.
(S) - Packet 3 sent
(S) - Timer started for Packet 3
(S) - Packet 4 sent
(S) - Timer started for Packet 4
(S) - Packet 5 sent
(S) - Timer started for Packet 5
- Packet 4 selected.
(R) - Packet 3 received. Selective acknowledge for only Packet 3 sent. Packet 3 delivered to application.
(R) - Packet 4 received. Selective acknowledge for only Packet 4 sent. Packet 4 delivered to application.
(R) - Packet 5 received. Selective acknowledge for only Packet 5 sent. Packet 5 delivered to application.
- Selective Ack of Packet 4 lost.
Applet started.
Type here to search
ENG
INTL
22:45
22-04-2017

7) Retransmitted after a time period.(only the lost packet)



8)All packets are sent.Window is slidded to the next packets.

9) Elapsed time of the protocol until the graph button in cicked.


Applet Viewer: SelectiveRepeat.class
Applet
Send New
Pause
Faster
Slower
Kill Packet/Ack
Reset
graph
Sender (Send Window Size = 3)
base = 6
ELAPSED TIME GRAPH FOR 'selective repeat'
elapsedtime(msec)
user input
dataset
elapsed time
Packet
Received
Ack
Ack Received
Selected
(S) - Action at Sender
(R) - Action at Receiver
(S) - Packet 5 sent
(S) - Timer started for Packet 5
-Simulation speed increased
(R) - Packet 3 received. Selective acknowledge for only Packet 3 sent. Packet 3 delivered to applica
-Simulation speed increased
(R) - Packet 4 received. Selective acknowledge for only Packet 4 sent. Packet 4 delivered to applica
(R) - Packet 5 received. Selective acknowledge for only Packet 5 sent. Packet 5 delivered to applica
-Simulation speed increased
-Simulation speed increased
-Simulation speed increased
-Simulation speed increased
(S) - Selective ACK for only Packet 3 received. Timer for Packet 3 stopped.
(S) - Selective ACK for only Packet 4 received. Timer for Packet 4 stopped.
(S) - Selective ACK for only Packet 5 received. Timer for Packet 5 stopped.
Applet started.

APPENDIX – A

1)INITIALIZATION AND RUN PHASE

```
public void init()
  {
        setLayout(null);
        output = new TextArea(150, 150); // setup output box
        // create text area for console output box
        output.setBounds(0, 400, 650, 250); // set bounds for output box
        output.setEditable(false); // prevent user from editing output written
        // to console
        add(output); // tell applet to draw our output box

        setupSimulationParams();

        pausableThreadPoolExecutor = new PausableThreadPoolExecutor(5);

        base = 0; // Defining our base to be 0 the first Packet number
        // expected
        receiver_base = 0; // Set the receiver base number to 0, which is the
        // first index in the receiver array
        nextseqsum = 0; // Defining next sequence number for next Packet
sent.
        fps = 5;



        sender = new SelectiveRepeatPacket[total_Packet];

        send = new Button("Send New");
        send.setActionCommand("rdt");
        send.addActionListener(this);
        send.setBounds(0, 0, 90, 20);

        // same structure as above
        stop = new Button("Pause");
        stop.setActionCommand("stopanim");
        stop.addActionListener(this);
        stop.setBounds(90, 0, 90, 20);

        fast = new Button("Faster");
        fast.setActionCommand("fast");
```

```
            fast.addActionListener(this);
            fast.setBounds(180, 0, 90, 20);

            slow = new Button("Slower");
            slow.setActionCommand("slow");
            slow.addActionListener(this);
            slow.setBounds(270, 0, 90, 20);

            kill = new Button("Kill Packet/Ack");
            kill.setActionCommand("kl");
            kill.addActionListener(this);
            kill.setEnabled(false);
            kill.setBounds(360, 0, 90, 20);

            reset = new Button("Reset");
            reset.setActionCommand("rst");
            reset.addActionListener(this);
            reset.setBounds(450, 0, 90, 20);
            add(send);
            add(stop);
            add(fast);
            add(slow);
            add(kill);
            add(reset);
            output.append("-Ready to run. Press 'Send New' button to start.\n");
  }// End init() method




 public void start()
   {
            // Creating GBNThread and starting execution. After start method is
run
            // the run method of this class is called
            if (gbnThread == null)
               gbnThread = new Thread(this);
            gbnThread.start();
   }// End start() method
```

```
   public void run()
   {
            //force garbage collection - depending on garbage collection threads
may be left
            //executing even though they have been killed leading to unexpected
behavior
            System.gc();

            boolean stopCheck = false;
            if (sender[total_Packet - 1] != null)
            {
               for (int i = total_Packet - window_len; i < total_Packet; i++)
               {
                              if (!sender[i].acknowledged)
                              {
                                 stopCheck = false;
                                 break;
                              }
                              else
                              {
                                 stopCheck = true;
                              }
               }

               if (stopCheck)
               {
                              output.append("Data Transferred - Simulation
completed.\n");
                              gbnThread = null;
                              return;
               }
            }









            Thread currenthread = Thread.currentThread();
            while (currenthread == gbnThread)
            {
```

```
            // While the animation is running
            if (onTheWay(sender)) // Checks if any of the Packets are traveling
                {
                  // Iterates through all of the Packet numbers (in this case
from
                  // 0 to 20)
                  for (int i = 0; i < total_Packet; i++)
                  {
                              // If the sender array for index[Packet number] is
not null,
                              // do the following, else do nothing
                              if (sender[i] != null)
                              {
                                    if(sender[i].packet_timer_task != null)
                                    {

sender[i].packet_timer_task.current_index = i;
                                    }

                              if (sender[i].on_way)
                                 {

                  if (sender[i].Packet_pos < (v_clearance - pack_height))
                        {
                           sender[i].Packet_pos += 5;
                        }
                  else if (sender[i].Packet_ack)
                        {
                                                sender[i].reached_dest = true;
                    if (check_upto_n(i))
                   {
                        sender[i].Packet_pos = pack_height + 5;
                        sender[i].Packet_ack = false;

                    if (sender[i].buffered || sender[i].acknowledged)
                   {
                                                      output.append("(R) -
Packet " + i + " received. Selective acknowledge for only Packet " + i + "
sent.\n");
                                                      sender[i].received =
true;
                                                }
                                                else if
(!sender[i].received)
```

```
                                                        {
                                                           output.append("(R) -
Packet " + i + " received. Selective acknowledge for only Packet " + i + " sent.
Packet " + i + " delivered to application.\n");
                                                           sender[i].received =
true;
                                                        }
                                                        else
                                                        {
                                                           output.append("(R) -
Packet " + i + " received out of order. Selective acknowledge for only Packet " +
i + " sent again(DUPACK)\n");
                                                        }

sender[i].received = true;
                                                                deliverBuffer(i);
                                                }

                                                else if (sender[i].acknowledged)
                                                {
                                                        sender[i].Packet_pos =
pack_height + 5;
                                                        sender[i].Packet_ack =
false;
                                                        output.append("(R) -
Packet " + i + " received. Selective acknowledge for only Packet " + i + "
sent.\n");
                                                        sender[i].received =
true;
                                                        deliverBuffer(i);
                                                }
                                                else
                                                {
                                                        sender[i].buffered =
true;
                                                        sender[i].Packet_pos =
pack_height + 5;
                                                        sender[i].Packet_ack =
false;

                                                        output.append("(R) -
Packet " + i + " received out of order.  Packet buffered. Selective acknowledge
for only Packet " + i + " sent.\n");
```

```
sender[i].received = true;

  deliverBuffer(i);
          if (i == selected)
          {
  selected = -1;
                                                          kill.setEnabled(false);
          }
              }
   } else if (!sender[i].Packet_ack)
   {

      output.append("(S) - Selective ACK for only Packet " + i + " received. Timer
for Packet " + i + " stopped.\n");
       sender[i].on_way = false;

              // In order check
      if (check_upto_n(i))
    {

sender[i].acknowledged = true;
sender[i].buffered = false;
      }
      else
   {

sender[i].acknowledged = true;
sender[i].buffered = true;
      }

   if (i == selected)
      {
                                                          selected = -1;
  kill.setEnabled(false);
                                                  }

                                                  // cancel the given packets
```

```
retransmission timer
                                                    if(sender[i].packet_timer_task !=
null)

sender[i].packet_timer_task.cancelTimer();




   for (int k = base; k < total_Packet; k++)
  {
  if (sender[k] != null)
          {
   if (sender[base].acknowledged)
     {

sender[base].buffered = false;
  if (k + window_len < total_Packet)
  {
                        base = base + 1;
  } }
  }
   else
   {
     break;
    }
  }

  if (nextseqsum < base + window_len)
  send.setEnabled(true);

    if (base != nextseqsum)
   {
          try
    {


if(sender[i].packet_timer_task != null)
                                                        {

sender[i].packet_timer_task.startTimer();
```

```
pausableThreadPoolExecutor.schedule(sender[i].packet_timer_task, 5,
TimeUnit.SECONDS);
                                                                        }
                                                                }
                                                                catch (IllegalStateException e)
                                                                {
                                                                        // ignore illegal state
exception and leave timer scheduled
                                                                }
                                                        }
                                                }

// End !sender[i].Packet_ack
                                        }// End sender[i] .onway
                                    }// End sender[i] != null
                        }// End for loop
                        repaint();

                        try
                        {
                              Thread.sleep(1000 / fps);
                        }
                        catch (InterruptedException e)
                        {
                              System.out.println("Help");
                        }
                    }
                else
                {
                        gbnThread = null;
                }
            }
    }
```

APPENDIX – B

BUFFERING AND RETRANSMISSION

```
    void deliverBuffer(int PacketNumber)
 {
int j = 0;

// Find our first buffered Packet in our array
while (j < PacketNumber)
{
   // error - all Packets up to PacketNumber should be created
   // if not something has gone horribly wrong
   if (sender[j] == null)
   {
         return;
   }
   // if Packet is ackd everythings fine keep looping
   else if (sender[j].acknowledged)
   {



                  sender[j].buffered = false;
                  j++;

   }
   else
   {
         break;
   }
}
```

```
  if (j > 0)
     j--;
  for (int k = j; k < total_Packet; k++)
  {
     // prevent indexing out of bounds
     if (sender[k] == null)
          break;
     // if packet is buffered deliver to application and advance window
     else if (sender[k].buffered)
     {
                    sender[k].buffered = false;
                    // sender[k].acknowledged = true;
                    output.append("(R) - Buffered Packet " + k + " delivered to
application.\n");
          // if this packet is ack'd already advance
     }
     else if (sender[k].acknowledged)
     {
                    sender[k].acknowledged = true;
                    sender[k].buffered = false;
     }
     else if (!sender[k].Packet_ack)
     {
          sender[k].buffered = false;
     }
     else
          break;
  }

  int count = 0;
  for (int i = 0; i < total_Packet; i++)
  {


    if (sender[i] != null)
     {
                    if (sender[i].received)
                    {
                       if (i + 1 <= (total_Packet - receiver_window_len))
                    count = i + 1;                              }
```

```
else
                    {
                       break;
                    }
      }
      else
      {
            break;
      }
   }
   receiver_base = count;
   }

    private void retransmitOutstandingPackets(int index)
    {
   int retransmitPacket = 0;

    if (sender[index] != null)
    {
            if (!sender[index].acknowledged && !sender[index].buffered)
            {
               sender[index].on_way = true;
               sender[index].Packet_ack = true;
               sender[index].Packet_pos = pack_height + 5;
               retransmitPacket++;
            }
            else if (!sender[index].acknowledged && sender[index].buffered)
            {
               sender[index].on_way = true;
               sender[index].Packet_ack = true;
               sender[index].Packet_pos = pack_height + 5;
               retransmitPacket++;
            }
    }

   if (gbnThread == null)
   {
      gbnThread = new Thread(this);
      gbnThread.start();
   }
```

```
   if (retransmitPacket == 0)
   {
            sender[index].packet_timer_task.cancelTimer();
   }
   else // Retransmitted packet has not been received, restart the timer
   {
      output.append("(S) - Timeout occurred for Packet " + index + ". Timer
 restarted for Packet " + index + ". \n");
      sender[index].packet_timer_task.startTimer();
      pausableThreadPoolExecutor.schedule(sender[index].packet_timer_task,
 5, TimeUnit.SECONDS);
   }
    }
```

APPENDIX – C

TIMER:

```
public class PacketTimerTask extends TimerTask

   {

   private boolean _cancel = false;

   private int current_index = 0;

   private boolean paused = false;


   public PacketTimerTask(){}


   // Method automatically called from the scheduler
(PausableThreadPoolExecutor)
```

```
@Override
public void run()
{
	int count = 0;
	while(count < time_out_sec_def)
	{
		try
		{
			while(paused)
			{
				Thread.sleep(100);
			}

				Thread.sleep(1000);
			}
		catch (InterruptedException e)
			{
				// ignore sleep interruption
			}
		count++;
	}
```

```
        if(!_cancel)
        {
                retransmitOutstandingPackets(current_index);
        }
}

public void cancelTimer()
{
        _cancel = true;
}

public void startTimer()
{
        _cancel = false;
}

        public synchronized void pause()
        {
                paused = true;
        }
        public synchronized void resume()
```

```
        {
                paused = false;
        }


class PausableThreadPoolExecutor extends ScheduledThreadPoolExecutor
{
  public PausableThreadPoolExecutor(int corePoolSize)
  {
        super(corePoolSize);
  }

  private boolean isPaused;
  private ReentrantLock pauseLock = new ReentrantLock();
  private Condition unpaused = pauseLock.newCondition();

  // Method called before scheduled timeout TimerTask executes
  @Override
  protected void beforeExecute(Thread t, Runnable r)
  {
        super.beforeExecute(t, r);
        pauseLock.lock();
        try
        {
                while(isPaused)
                        unpaused.await();
```

```
        }

        catch (InterruptedException ie)
        {
                t.interrupt();
        }
        finally
        {
                pauseLock.unlock();
        }
}
```

REFERENCES

1. Andrew S.Tannenbaum, David J.Wetherall, “Computer Networks”, 5th

Edition Pearson Education, ISBN-13: 978-0-13-212695-3, 2011

2. William Stallings, “Data and Computer Communications”, 9th Edition, Prentice-Hall of India (PHI), ISBN-81-203-1240-6, 2011

3. Forouzan, “Data Communication and Networking”, 5th Edition, Tata McGraw Hill , ISBN: 978-0-07-296775-3, 2012