Chapter 5 System Design, Implementation and Testing

A TCPIP component is a TCP with IP component. This component has both TCP features and IP features, which will be explained in section 5.1. In UMJaNetSim v0.5, this simulator's TCPIP component consists of two type of TCP version, i.e. Tahoe and Reno. This thesis adds an additional TCP version in the UMJaNetSim; that is NewReno.

This chapter is divided into five sections. The first section discusses on the component design of UMJaNetSim. A brief discussion on the TCPIP component class is included. Its class name is called TCPIPApp class.

The second section discusses the TCPIP component implementation. TCPIPApp class and NewReno's method (it is an OOP function) are also discussed. The network switch is also briefly tested on the lost segment that is purposely created. The switch class is called ATMLSR class.

The third section discusses the testing on the TCPIP component simulation. Each TCP version; including Tahoe, Reno and NewReno; is tested individually. There are three lost segments created purposely in the switch. The testing is on sender sequence number and its congestion window (cwnd).

The forth section discusses on the overall network with the same TCP version and a study on their average throughput is conducted.

Finally, there is a chapter summary.

5.1 Object Oriented Design

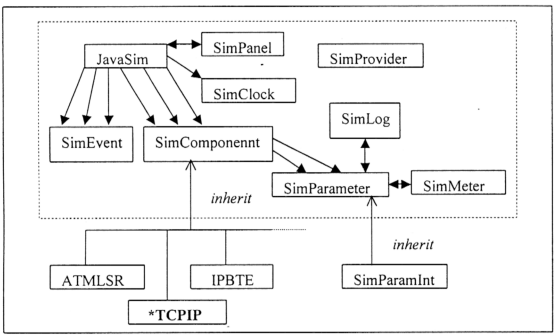

Figure 5.1 TCPIP Simulator Objects

Figure 5.1 shows the TCPIP simulator objects. There are three child SimComponent to be considered in this thesis, i.e. the TCPIP component, the IP terminal (IPBTE component) and the network switch (ATMLSR component). These components will inherit all the features from the SimComponent (See Section 4.2.2).

The TCPIP component has both TCP and IP features. TCP features are segmentation of packets and reassembly of packets. It also consists of sender buffers (congestion window) and receiver buffers (advertise window). In UMJaNetSim v0.5, this simulator's TCPIP component already consists of two types of TCP version; they are Tahoe and Reno. This thesis adds an additional TCP version in the UMJaNetSim; that is NewReno. The IP feature in this TCPIP component has the destination TCPIP component's IP address and port number. With the IP address, the TCPIP component is able to connect to the destination TCPIP component.

In Figure 5.1, the IPBTE component is an IP terminal. This component is used to connect between the TCPIP component and the network switch. This component is used to set the IP address. With this IP address, the TCPIP component is able to connect to the destination TCPIP component.

The ATMLSR component is a network switch. This component is used to route the packets sent by the TCPIP component to the target TCPIP component. From the target TCPIP component, acknowledgement packets are routed back to the sender TCPIP component.

The next section will briefly discuss on the TCPIP component class. This component's class is called TCPIPApp class.

5.2 Class Design

This section gives a description on the class designs of the UMJanetsim network simulator. The properties and the functionality of TCPIPApp class are discussed.

5.2.1 TCPIPApp class

The TCPIPApp class is the class that is used to simulate TCP features with IP. This class is called when the user selects TCPIP from the menu list. This class must perform the following function:

- It must let the user predefined TCP features automatically or manually.
- It must allow the user to input the destination IP address, destination subnet mask and destination port number.

Table 5.1 Attributes and Methods of TCPIPApp

Major Attribute	Major Methods
TCPIP Name	Set TCPIP name
TCPIP version	Segmentation packet
Segment ID	Process segment
Acknowledgement ID	Reassemble packet
Retransmission or timeout	Retransmission or timeout

Table 5.1 shows the attributes and methods of the TCPIPApp class. Each TCPIP component is assigned a unique name. This component consists of three TCPIP versions, including Tahoe, Reno and NewReno. Segment ID is used to identify the particular segment when the packet is sent. Acknowledgement ID is used to identify the particular segment being acknowledged. There is a need to create dynamic retransmission or timeout attributes.

The major methods are to set the TCPIP component name, segmentation of packets, manipulating the particular version feature in the segment process, reassemble packets, and calculate retransmission or timeout while the packets are being sent.

5.3 TCPIP Simulator Component Implementation

The following section discusses the implementation for TCPIP simulator components, especially for NewReno.

5.3.1 TCPIPApp class

The TCPIPApp class is the class for NewReno.

```
class TCPIPAPP extends SimComponent implements java.io.Serializable{
      private static final int TCP_NewRENO = 3;
      //define the NewReno
}
```

5.3.1.1 Process Segment Method

This method is used to determine which algorithm has to apply for the three consecutive lost segments; how to recalculate the Retransmission Time-out; how to process the data while the data is available; and when the lost segment is recovered, how the traffic is adjusted back to normal (exit fast retransmit and fast recovery). It can be applied to Tahoe, Reno and NewReno.

Step 1: Determine whether it is Tahoe, Reno or NewReno TCP version

- process_segment() or process_segment2()

Step 2: Determine whether it is a lost segment

If there are 3 duplicate acknowledgements received by the sender

Enter the fast retransmit and fast recovery process

Else

Perform slow start or congestion avoidance

Step 3: Recalculate Retransmission Time-out (RTO)

Jacobson [LEON], in [JACOBSON98] proposed a calculation of RTO algorithms. Refer section 2.1.2.4.

Step 4: Segment text processing

If data is available at TCP

Inform the user of need for buffers

Step 5: Exit fast retransmit and fast recovery

5.3.1.2 Retransmission and Timeout Methods

This method is used to calculate retransmission and timeout attributes while sending packets and receiving acknowledgements. The first step is to initialize the slow start theshold (ssthresh) and congestion window (cwnd) size. ssthresh is to determine whether the sender is in slow start or congestion avoidance mode. cwnd is the sender buffer size. The second step is to determine whether there are unacknowledged segments. If there is an unacknowledged-segment, the sender will retransmit the segment.

Step 1: Reset slow start theshold (ssthresh) and congestion window (cwnd) size
//initialize ssthresh and cwnd size

Step 2: Send lowest unAcked packet

Firstly, establish length to send.
Secondly, calculate upper band of window to send.

5.3.2 ATMLSR class

The ATMLSR class is the class that performs the network switch. This class inherits the methods and properties of SimComponent (see Section 4.2.2). The following list is the attribute and methods of this class. The modification, in ATMLSR class, is to create a lost packet for TCPIP packet flow.

5.3.2.1 Attributes

There are three lost segments created. The attributes used to create these lost segments are shown below:

```
class ATMLSR extends SimComponent implements Serializable {
```

```
//first drop packet
 private double last_drop_time;
   //capturing time for the last drop segment
 private double cur_time;
   //capturing the current time

 //second drop packet
 private double last_drop_time2;
   //capturing time for the last drop segment
 private double cur_time2;
   //capturing the current time

 //third drop packet
 private double last_drop_time3;
   //capturing time for the last drop segment
 private double cur_time3;
   //capturing the current time
}
```

5.3.2.2 Dropping packet Methods

This method is the function for dropping packets in the network switch.

This method creates a dropped packet.

```
private void sw_my_receive(SimEvent e)
```

Below is the time capture for creating dropped packets. (Refer Section 4.1.1.1 for the simulation time)

```
//set current time
   cur_time = SimClock.Tick2Sec(theSim.now());
   cur_time2 = SimClock.Tick2Sec(theSim.now());
   cur_time3 = SimClock.Tick2Sec(theSim.now());
   cur_time4 = SimClock.Tick2Sec(theSim.now());

if it is the first switch
   if (cur_time - last_drop_time is greather than 1.2 )
      //drop first packet
      last_drop_time = cur_time;
    else if (cur_time2 - last_drop_time2 is greather than 1.6 )
```

```
    //drop second packet
    last_drop_time2 = cur_time2;
  else if (cur_time3 - last_drop_time3 is greather than 2.4 )
    //drop third packet
    last_drop_time3 = cur_time3;
  else
    //send packet
    sw_receive_ip_datagram(cell,voport);
```

5.4 TCPIP component Testing

The following section discusses the details of the testing.

5.4.1 Testing

The testing, shown in Figure 5.2, is used to test the individual TCP versions, i.e. Tahoe, Reno and NewReno, for the TCP/IP simulator. The testing will include the feature testing and the interaction testing.

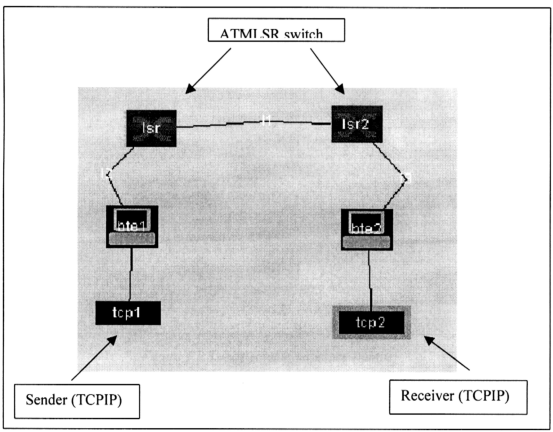

Figure 5.2 TCPIP component topology

Figure 5.2 is the TCPIP component topology used to test the TCPIP components. *lsr* and *lsr2* are the network switches. *bte1* and *bte2* are the IP Terminals. *tcp1* and *tcp2* are the TCPIP components. Refer to Section 5.1 for the network switch, IP Terminal and TCPIP component definitions.

The following are some of the tests that has been conducted:

5.4.1.1 Tahoe component test

Figure 5.3 shows the TCP Tahoe sender time elapsed versus the sequence number. The simulation created three lost segments at the network switch (lsr), which are labeled as 1, 2 and 3.

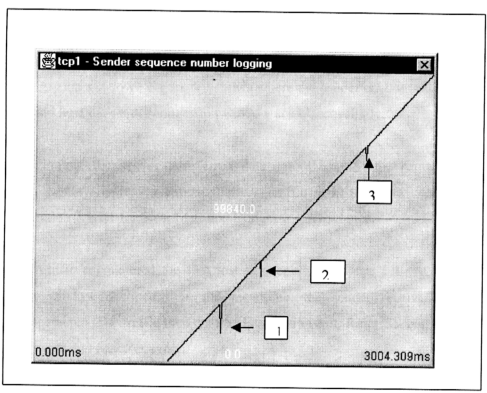

Figure 5.3 Tahoe sender sequence number

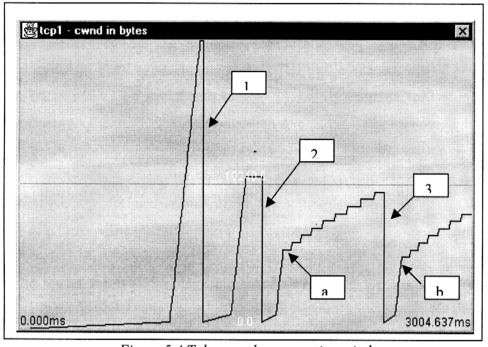

Figure 5.4 Tahoe sender congestion window

Figure 5.4 shows the time elapsed versus the traffic workload in the TCP Tahoe sender congestion window. Figure 5.4 shows where the sender encountered three lost segments and how the sender congestion window acted accordingly.

In the beginning, the sender transferred data smoothly. At the same time, the congestion window increased exponentially. Traffic is in the Slow Start fashion. Refer to Section 2.1.1.1 for Slow Start and Congestion Avoidance.

When the traffic encountered the first lost segment [1], Figure 5.4 shows that the congestion window declined. In the meantime, the sender received duplicate acknowledgements. The traffic then entered Fast Retransmit. Refer Section 2.1.1.2 for Fast Retransmit and Fast Recovery.

After the sender received the duplicate acknowledgement, the congestion window started to grow exponentially. The traffic entered Slow Start fashion again. However, the traffic encountered the second lost segment [2] . Figure 5.4 shows that the congestion window then declined again. In the meantime, the sender received

duplicate acknowledgements. The traffic reentered Fast Transmit. (See section 2.1.1.2)

After the sender received the duplicate acknowledgement again, the congestion window started to grow exponentially again. The traffic entered Slow Start. This time the traffic met a slow start threshold [a] . In the meantime, the traffic started to grow linearly. The traffic entered Congestion Avoidance. (See Section 2.1.1.1)

When the traffic encountered the third lost segment [3] , the figure shows the congestion window declined again. In the meantime, the sender received duplicate acknowledgements. The traffic reentered Fast Transmit again.

After the sender received the duplicate acknowledgement, the congestion window started to grow exponentially again. The traffic entered Slow Start. This time the traffic met a slow start threshold [b] again. In the meantime, the traffic started to grow linearly. The traffic then entered Congestion Avoidance. (See Section 2.1.1.1)

This figure shown is the expected figure. It fulfills the RFC2581 specification.

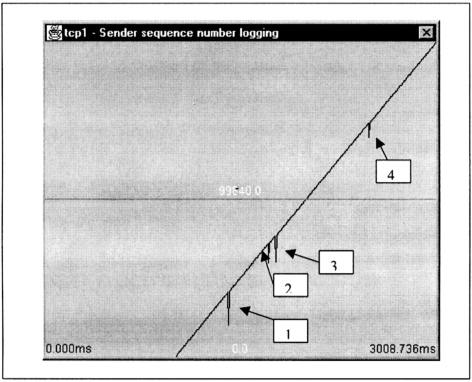

Figure 5.5 Reno sender sequence number

5.4.1.2 Reno component test

Figure 5.5 shows the TCP Reno time elapsed versus sender sequence number. The simulation created three lost segments at the network switch (lsr). In this figure, there are three lost segments, which were purposely created, shown as [1], [2], and [4]; a lost segment [3] was caused by timeout.

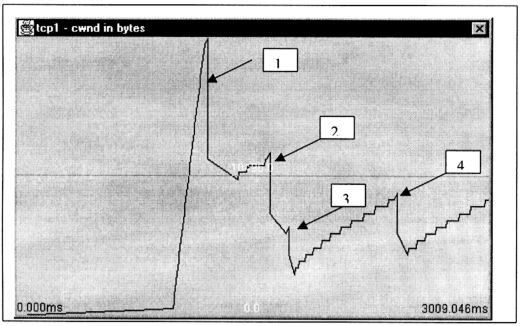

Figure 5.6 Reno sender congestion window

Figure 5.6 shows the time elapsed versus the traffic workload in the TCP Reno sender congestion window. Figure 5.6 shows where the sender encountered three lost segments, which is labled as [1], [2] and [4] ; and one timeout retransmit segment [3], and how the sender congestion window acted accordingly.

In the beginning, the sender transferred data smoothly. At the same time, the congestion window increased exponentially. Traffic is in the Slow Start fashion. Refer to Section 2.1.1.1 for Slow Start and Congestion Avoidance.

When the traffic encountered the first lost segment [1] , Figure 5.6 shows that the congestion window declined. In the meantime, the sender received duplicate acknowledgements. The traffic entered Fast Retransmit and Recovery. Refer Section 2.1.1.2 for Fast Retransmit and Fast Recovery.

After the sender received the duplicate acknowledgement, the congestion window started to grow linearly. The traffic entered Congestion Avoidance. Refer to Section 2.1.2.2 for the Introduction of Reno. However, the traffic encountered the second lost

segment 2 again. Figure 5.6 shows that the congestion window declined again. In the meantime, the sender received the duplicate acknowledgements. The traffic entered Fast Retransmit and Recovery. (See Section 2.1.1.2)

When the sender received acknowledgement of the lost segment, the traffic encountered timeout for a sent segment. The sender retransmitted immediately the timed out segment 3 .

After the sender received the acknowledgement of the timed out segment, the traffic started to grow linearly. The traffic entered Congestion Avoidance. (See Section 2.1.2.2)

The traffic encountered the forth lost segment 4 . Figure 5.6 shows the congestion window declined again. In the meantime, the sender received duplicate acknowledgements. The traffic entered Fast Retransmit and Recovery.

After the sender received the duplicate acknowledgement, the congestion window started to grow linearly. The traffic entered Congestion Avoidance. (See Section 2.1.2.2)

This figure shown is the expected figure. It fulfills the RFC2581 specification.

5.4.1.3 New Reno component test

Figure 5.7 shows the TCP NewReno time elapsed versus sender sequence number. The simulation created three lost segments at the network switch (lsr). In this figure there are three lost segments, which were created purposely, labeled 1 , 2 4 ; and a lost segment 3 was caused by timeout .

Figure 5.8 shows the time elapsed versus the traffic workload in the NewReno sender congestion window. Figure 5.8 shows where the sender encountered four lost segments and how the sender congestion window acted accordingly.

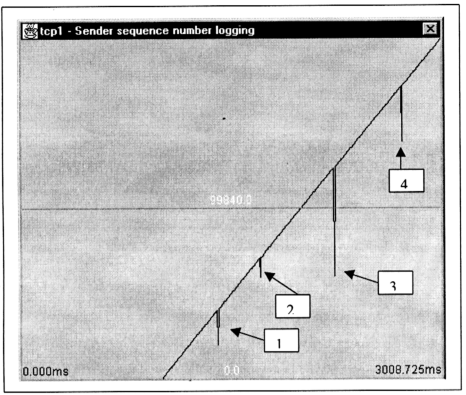

Figure 5.7 NewReno sender sequence number

Figure 5.8 shows that the sender congestion window started by growing exponentially. Traffic is in the Slow Start fashion. Refer Section 2.1.1.1 for Slow Start and Congestion Avoidance. When the traffic encountered the first lost segment [1], the congestion window declined. At the same time, the sender received three duplicate acknowledgments. The traffic entered Fast Retransmit and Recovery. During Fast Recovery, two segments were sent for each additional duplicate acknowledgment received. (See Section 2.1.2.3) This is to fully utilize the traffic.

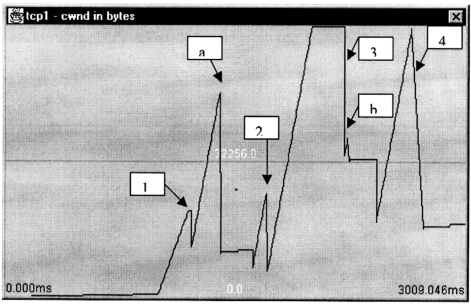

Figure 5.8 NewReno sender congestion window

The second decline [a] and flat line occurred because the sender sent the same value of sequence number before it left Fast Recovery and recovered from the lost segment. The congestion window regained exponential growth.

The second lost segment [2] was encountered. The line declined again. After the sender received three duplicate acknowledgments, the traffic entered Fast Retransmit and Recovery. The sender started to send two segments for every additional duplicate acknowledgment received. (See Section 2.1.2.3)

Before exiting Fast Recovery, a lost segment [3] occurred due to timeout. The timed out segment was retransmitted following which there was a flat line [b]. This was because the sender sent the same value of sequence number before it left Fast Recovery and recovered from the loss segment.

After leaving the Fast Recovery, the congestion window grew exponentially.

The forth lost segment [4] was encountered. The line showed a decline again. After the sender received three duplicate acknowledgments, the traffic entered Fast

Retransmit and Recovery. During Fast Recovery, the sender sent two segments for every additional duplicate acknowledgment received.

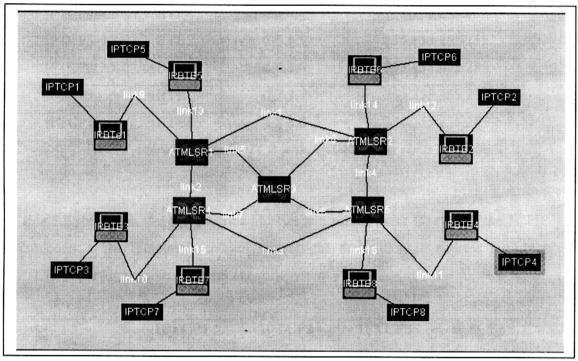

Figure 5.9 TCP/ IP Network Topology Structure

5.4.2 Performance Evaluation of Overall same TCP version Network Topology

Figure 5.9 shows the TCP/IP Network Topology structure with all TCPIP components having the same TCP version. ATMLSR*r* are the network switches, where r is in the range of 1 to 5. IPBTE*n* are the IP Terminals, where n is in the range of 1 to 8. IPTCP*m* are the TCPIP components, where m is in the range of 1 to 8. link*j* are links between IP Terminals and the network switchs, where j is in the range of 1 to 12. Refer to Section 5.1 for the network switch, IP Terminal and TCPIP component definitions.

IPTCP	BTE	Source Address	Subnet Mask
IPTCP1	BTE1	1.1.1.1	255.255.255.255
IPTCP2	BTE2	2.2.2.2	255.255.255.255
IPTCP3	BTE3	3.3.3.3	255.255.255.255
IPTCP4	BTE4	4.4.4.4	255.255.255.255
IPTCP5	BTE5	5.5.5.5	255.255.255.255
IPTCP6	BTE6	6.6.6.6	255.255.255.255
IPTCP7	BTE7	7.7.7.7	255.255.255.255
IPTCP8	BTE8	8.8.8.8	255.255.255.255

Table 5.2 Address Space for TCP with IP component (IPTCP)

Table 5.2 shows the source network addresses and subnet mask for each of the TCPIP component. The Simulator randomly selects the destination IP address for every IPTCP. It performed like an actual network system.

	IPTCP1 (Bits / Sec)	IPTCP2 (Bits / Sec)	IPTCP3 (Bits / Sec)	IPTCP4 (Bits / Sec)
Tahoe	804547	804547	804547	804547
Reno	804685	804685	804685	804685
NewReno	804547	804685	804547	804685

Table 5.3 Overall network IPTCP Average Throughput 1

	IPTCP5 (Bits / Sec)	IPTCP6 (Bits / Sec)	IPTCP7 (Bits / Sec)	IPTCP8 (Bits / Sec)
Tahoe	804685	804547	804547	804685
Reno	804685	804821	804547	804547
NewReno	804685	804822	804685	804685

Table 5.3 Overall network IPTCP Average Throughput 2

	Average Throughput (Bits / Sec)
Tahoe	804582
Reno	804667
NewReno	804668

Table 5.4 Overall network IPTCP Average Throughput

Table 5.3 shows the average throughput for each IPTCP component with the same TCP version.

Table 5.4 is a summary of the average throughput for all TCPIP components with the same TCP version. Table 5.4 shows that NewReno has the highest average throughput for overall network IPTCP, followed by Reno and Tahoe. This means that NewReno has the best transmission among the three TCP versions even though the network encountered congestion.

Case Study: TCPIP component network Topology

This network topology evaluated the all TCPIP component with the same TCP version. This network topology fixed every IPTCP with a fixed particular destination IPTCP. This is to make sure that all three of the overall network topology with same TCP version has the same topology and same destination.

There were two or three lost segments generated randomly at the ATMLSR1 and ATMLSR4 network switches.

The results showed that NewReno had the highest average throughput among the three TCP versions, followed by Reno and Tahoe. This means that NewReno is the best transmission TCP version when the sender encounters congestion.

The details of this testing will be listed in Appendix A Case Study.