CHAPTER FOUR

PETRA: A NEW TCP CONGESTION CONTROL MECHANSIM

This chapter introduces a new End-To-End congestion control mechanism based on TCP New Reno and TCP Westwood. The name 'PETRA' comes from the name of the author's home town Petra, Jordan. Three new modifications to TCP NewReno are presented in this chapter. Some modifications are based on the TCPW bandwidth estimation method.

Firstly, we introduced a new slow start enhancement. Two modifications are presented: a new method is proposed to calculate the initial slow start threshold (*ssthresh*) value, and a faster start phase to accelerate the congestion window growth. Secondly, we introduce two new modifications to the Fast Retransmission and Fast Recovery algorithm, including faster retransmission and best recovery methods. Finally, we introduce a new modification to the timeout procedure to prevent TCP from resting its congestion window size every timeout event unless real congestion is shaped. For each modification, we compare the results of the proposed enhancements with TCPW and TCP NewReno. For the comparison, we used the ns-3 simulator to obtain the results of two network performance metrics: the total throughput and the congestion window size. We used GNUPLOT version 4.2 to plot graphs to make the comparison more obvious.

In chapter five, we discussed implementing and evaluating the proposed modifications in chapter four over the LTE networks. Moreover, this chapter introduced the LTE technology, implementing LTE in ns-3, and TCP performance

UNIVERSITI SAINS ISLAM MALAYSIA
ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA
جامعة العلوم الإسلامية الماليزية

over LTE. Chapter five discusses the implementation of TCP PETRA in ns-3. The evaluation process included more network performance metrics: throughput, congestion window size, average delay, and jitters. Also, chapter five tested the proposed implementation 'PETRA' over LTE network.

4.1 Enhanced Slow Start Algorithm

The first congestion control mechanism was developed in TCP Tahoe. Two new mechanisms have been introduced: the slow start, and the congestion avoidance. Later, TCP Reno introduced another two new mechanisms: fast retransmission, and fast recovery. The TCP sender uses these mechanisms to control the sending rate. To implement these algorithms, two variables are added to the TCP pre-connection state (Allman et al., 1999). The congestion window (*cwnd*) is the amount of data the sender can transmit before receiving an acknowledgment (*ACK*), and the receiver's advertised window (*rwnd*) is the amount of outstanding data at the receiver side. The TCP uses the minimum of *cwnd* and *rwnd* to control data transmission.

The TCP starts transmission using the slow start algorithm to slowly probe the network condition (Hoe, 1996). The sender side has no information about the available link bandwidth. Therefore, it starts sending with an initial window (*IW*) of *cwnd* size less than or equal to 2*SMSS (Sender Maximum Segment Size) bytes. However, some TCP extensions allow the use of a larger *IW*, as defined in the following equation:

$$IW = min\ (4*SMSS,\ max\ (2*SMSS,\ 4380\ bytes)) \tag{4.1}$$

According to equation (4.1), the TCP sender may use two or four SMSS for the *IW* but not more than 4380 bytes. During the slow start phase, a TCP sender

UNIVERSITI SAINS ISLAM MALAYSIA
ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA

increments its *cwnd* by 1 SMSS bytes every time a new *ACK* arrives. This exponential incrementing in *cwnd* ends when *cwnd* exceeds *ssthresh* value, and then a congestion avoidance phase is started.

During the congestion avoidance phase TCP sender increments its *cwnd* by one full-sized segment per *RTT*, according to the following equation:

$$cwnd = cwnd + SMSS * SMSS/cwnd \quad (4.2)$$

A TCP sender starts a timer name 'Timeout' every time a new segment is transmitted. If the timeout expires before the receiver acknowledges receiving its segment, then the TCP sets the *ssthresh* value to half the *FlightSize* (the amount of outstanding data in the network), rests its *cwnd* to the *IW*, and starts the slow start phase again. One simple mistake is to set the *ssthresh* to half the current *cwnd* when the TCP sender detects segment loss, which in some implementations may incidentally exceed *rwnd*.

For the initial slow start, there is no safe *ssthresh* value. If the *ssthresh* value is too small, then the sender will quickly stop the exponential incremental of the *cwnd*. Thus, it will take a long time to reach the optimal *cwnd* size using the congestion avoidance linear incrementing. On the other hand, using large *ssthresh* value will aggressively increase the *cwnd*, which causes in some cases the *rwnd* to be exceeded.

Therefore, we propose the TCPW bandwidth estimation mechanism to properly set the initial *ssthresh* value, as discussed in section 4.1.1.

4.1.1 Setting-up Initial Slow Start Threshold Value

In this section, we present a new method to properly set the *ssthresh* value for the initial slow start phase. One simple End-To-End method to estimate the link capacity

UNIVERSITI SAINS ISLAM MALAYSIA
ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA

is to monitor the rate of the returning acknowledgments at the receiver side. Here, the time between two sequences of *ACKs* represents the link capacity of transmit a packet, buffering time in the intermediate nods, the receiver's processing, and sending the *ACK* back to the sender. Therefore, TCPW probes the link's bandwidth by counting the bytes acknowledged in every *ACK* divided by the period between two sequences of *ACKs*, according to equation (4.3).

$$EBW = \frac{Acked * SMSS}{t_k - t_{k-1}} \tag{4.3}$$

Where *Acked* is the total number of packets acknowledged with every *ACK*, t_k is the current *ACK* arrival time, and t_{k-1} is the last *ACK* arrival time. However, TCPW uses the same equation in (4.3), for every timeout occurred. Thus, in TCPW, t_k the loss time, and t_{k-1} is the time of the last estimation. However, we propose using the estimation of every *ACK* received. To eliminate randomness in the calculated *EBW*, we used the moving average method with ($\alpha = 0.9$) to obtain a smooth estimation all over the transmission period, according to equation (4.4).

$$SEBW = (1-\alpha)EBW_i + \alpha * EBW_{i-1} \tag{4.4}$$

Finally, to convert the *SEBW* in form of *ssthresh* value, we use the following equation:

$$ssthresh = SEBW * MinRTT \tag{4.5}$$

The computed *ssthresh* will provide an accurate value for the initial slow start threshold based on the link capacity. As seen, the *ssthresh* value is not a constant number, whereas it is varies according to the connection status.

UNIVERSITI SAINS ISLAM MALAYSIA ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA

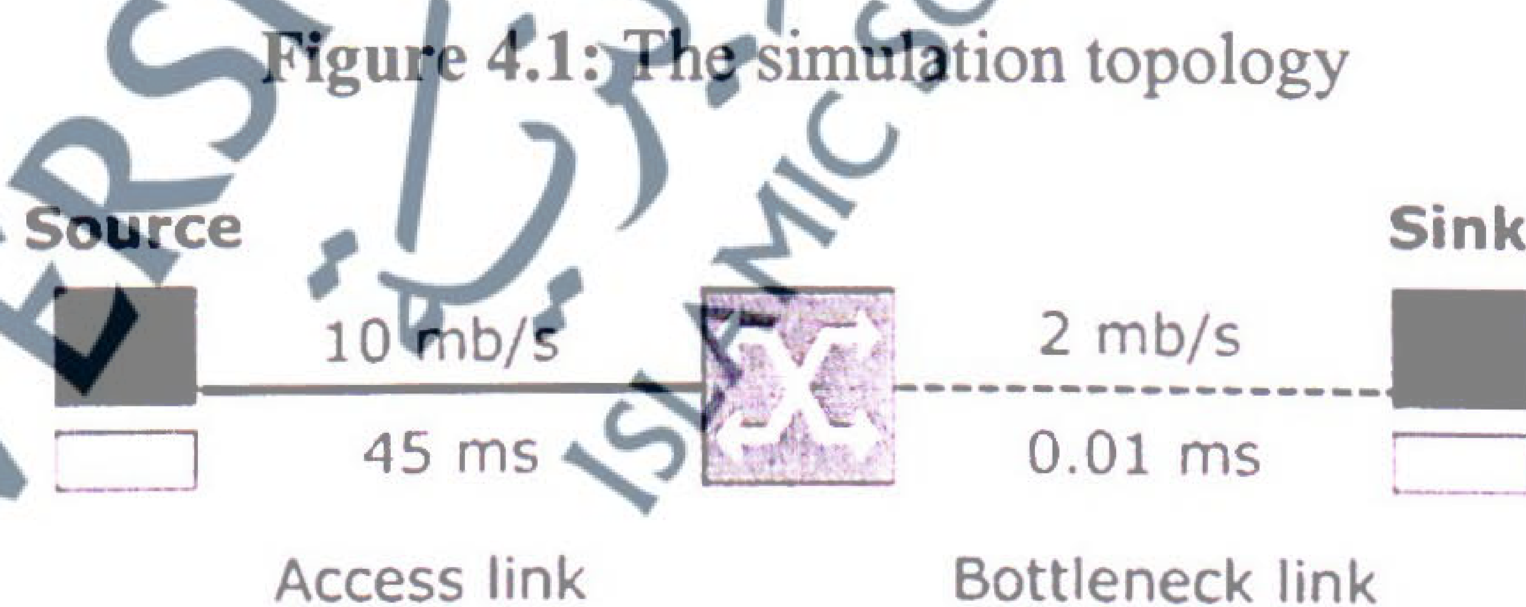
Whenever a loss event occurs (timeout occurred, or duplicate *ACKs*), we reset the *EBW* value to start a new estimation based on the new connection status to keep pace with the dynamic link changes. We implemented the changes required using ns-3. The following section elaborates the implementation and validation of the method.

4.1.1.1 Performance Evaluation

We implemented the changes proposed in section 4.1.1 using a modified version of TCP New Reno implementation included in ns-3. We included two new functions as presented in TCPW: a function to count the number of acknowledged packets in every received *ACK*, and a function to calculate the *EBW* using the formulae in (4.3, 4.4, and 4.5), respectively.

To evaluate the proposed modifications, we used ns-3 to obtain two main network performance metrics: total throughput, and the congestion window. Then we compared the results with TCP NewReno and TCPW. For constant simulation experiments, we employed the same topology used to present TCPW. The topology is presented in Figure 4.1.

Figure 4.1: The simulation topology



The topology in Figure 4.1 shows a mixed (wired-wireless) network which contains a single source and sink connected via Gat-way (PGW). There are two links: a source-router (wired portion), labelled access link, and a router-sink (wireless portion), labelled bottleneck link. For the access link, we set the link bandwidth 10Mbps with a

UNIVERSITI SAINS ISLAM MALAYSIA
ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA

propagation delay of 45ms, and for the bottleneck link we set the link bandwidth 2Mbps with propagation delay of 0.01ms. A peer-to-peer connection was used for both links using the PointToPointHelper provided by ns-3. For the bottleneck link, we used the built-in RateErrorModel class to generate sending errors. Errors were assumed to follow uniform distribution to simulate the wireless networks channel. ns-3's BulkSendApplication class was used to generate a single flow of traffic starting at the source and destined for the sink along the simulation period. We set the packet size 400 Bytes as in TCPW. Table 4.1 details the simulation parameters.

For the first experiment, we used a very short time of five seconds to clearly show the congestion window behaviour for the three implementations of TCPW, NewReno, and the new modification. The results are plotted in Figure 4.2.

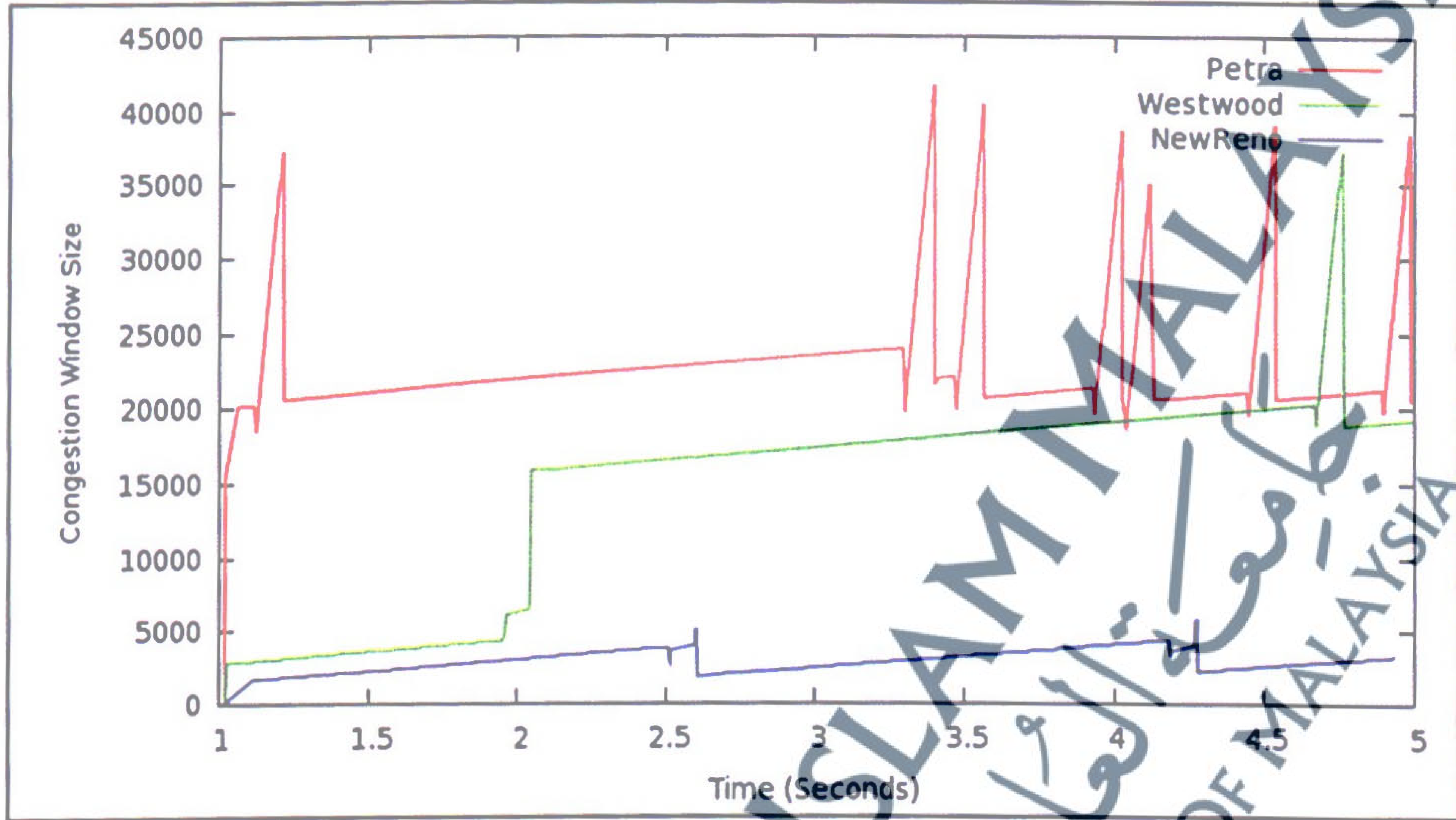
Table 4.1: Simulation parameters

Parameter	Value
Mobility	Fixed Position
Access link bandwidth	10Mbps
Access link Propagation Delay	45 ms
Bottleneck link bandwidth	2Mbps
Bottleneck link Propagation Delay	0.01 ms
Packet Size	400bytes
Error model	Uniform error model
Packet Error Rate (PER)	0.005
Application type	Bulk send application
Simulation time	100 seconds

UNIVERSITI SAINS ISLAM MALAYSIA
ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA

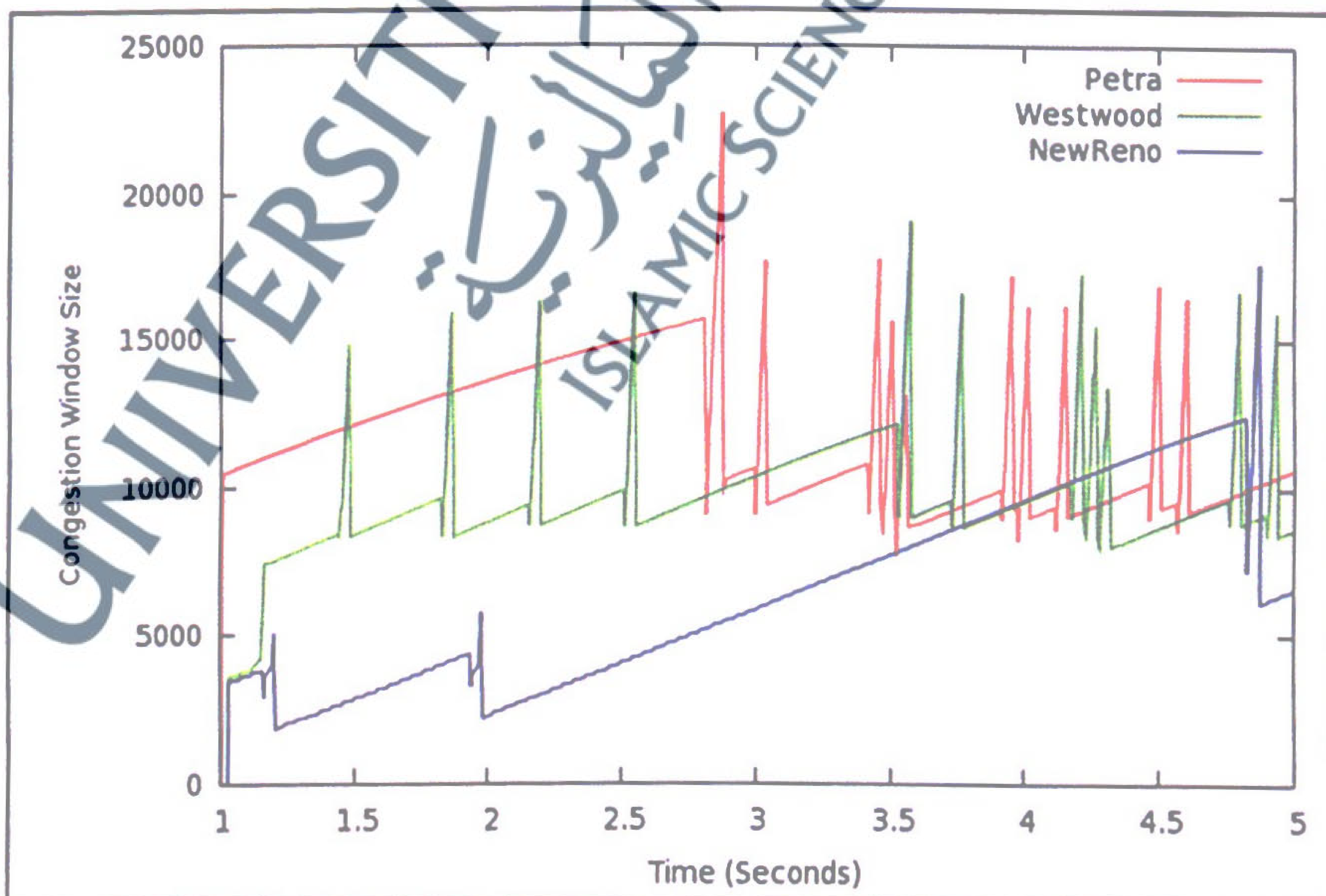
As we can see in Figure 4.2, the new modification shows a larger congestion window size compared to TCPW and TCP New Reno. Another observation is that the new modification reaches the optimal *cwnd* size faster compared to TCPW.

Figure 4.2: Congestion Window (Propagation Delay=45ms)



For the second experiment, we used the same scenario's parameters but we set the access link's propagation delay = 20ms and we plotted the results in Figure 4.3.

Figure 4.3: Congestion Window (Propagation Delay=20ms)

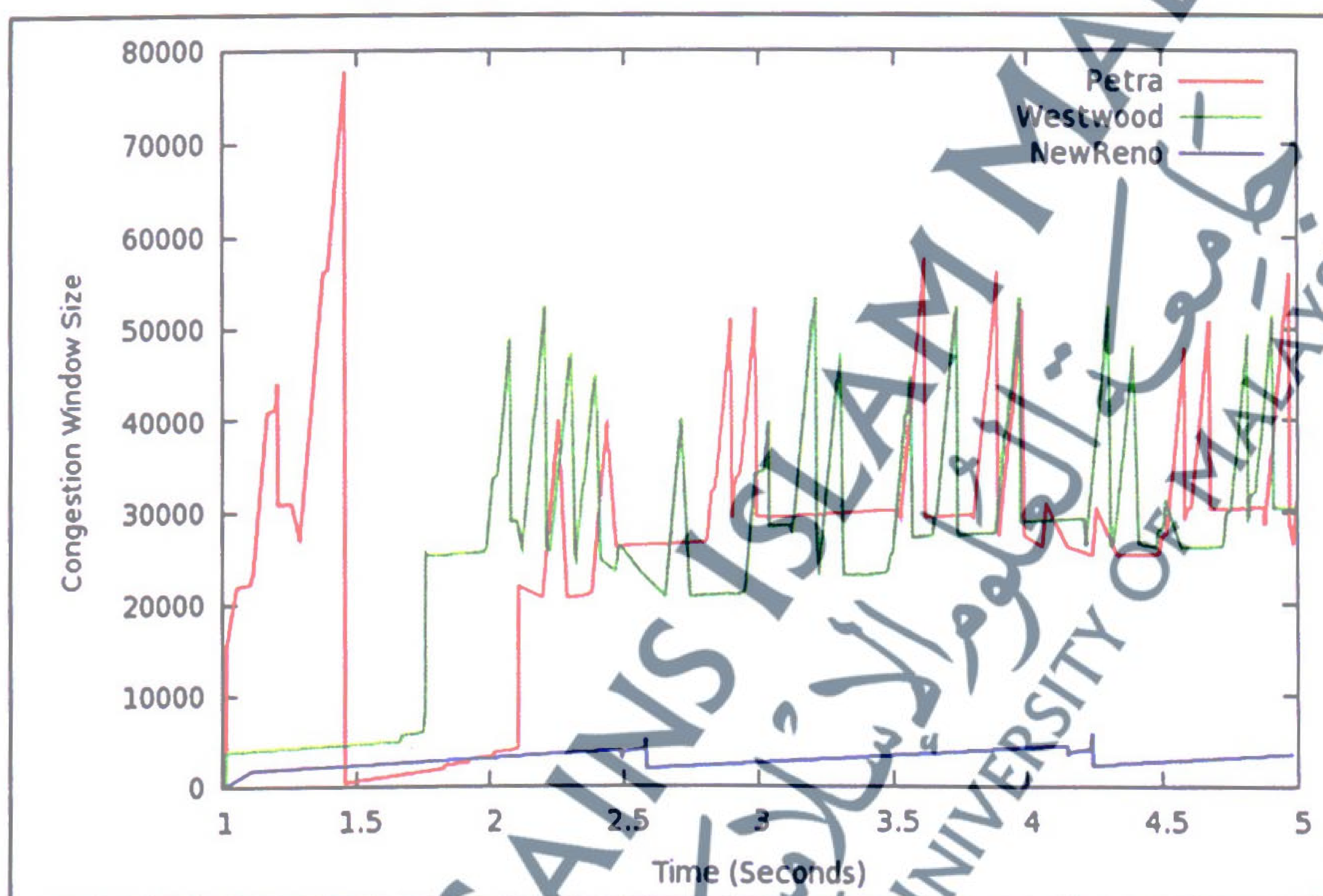


UNIVERSITI SAINS ISLAM MALAYSIA
ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA

In figure 4.3, the new modification recorded better *cwnd* size compared to other implementations. However, the difference was closer than in the first experiment.

For the next experiment, we increased the packet error rate (PER) to 0.009. We ran the same experiment and plotted the results in Figure 4.4.

Figure 4.4: Congestion Window (PER=0.009)



Again, the new modification recorded bigger *cwnd* over the two implementations. We conclude that the new modification improves the initial slow start in terms of congestion window size. We have extended our simulation experiments to compare the total throughput over 100 seconds in the simulation period. The results are presented in Table 4.2.

Table 4.2: Throughput Results

Experiment Number	Bottleneck Bandwidth (Mbps)	Packet Error Rate (PER)	Propagation Delay (ms)	PETRA (Mbytes)	TCPW (Mbytes)	New Reno (Mbytes)
1	2	0.005	45	**17.10**	16.84	4.27
2	4	0.005	45	**27.21**	27.03	4.33
3	8	0.005	45	**10.39**	9.06	4.35
4	10	0.005	45	**9.65**	9.48	4.35

UNIVERSITI SAINS ISLAM MALAYSIA
ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA

5	2	0.0075	45	14.86	**15.02**	3.47
6	2	0.009	45	**14.00**	13.68	3.17
7	2	0.01	45	**13.50**	13.47	3.01
8	2	0.025	45	**6.44**	6.44	1.26
9	2	0.005	30	**17.59**	17.24	6.12
10	2	0.005	20	**18.57**	18.35	8.93
11	2	0.005	10	**18.60**	18.50	15.19
12	2	0.005	1	18.65	18.65	**20.59**

Table 4.2 shows the total throughput results of 12 different experiments. We used different bottleneck bandwidth, different propagation delay, and different packet error rate values. We include in bold the maximum value recorded by the three implementations. As we can see, the new modification recorded higher throughput values compared to other implementations; however, TCPW recorded one maximum throughput in experiment 5, as well as New Reno in the last experiment.

4.1.2 Accelerated Slow Start

In the previous section, we introduced a new method to set the initial *ssthresh* value for the initial slow start phase. In this section, we introduced a new method to accelerate the slow start to better utilizing the available link's bandwidth. Therefore, we propose to increase the *cwnd* size according to the bandwidth utilisation. As mentioned before, during the slow start phase the sender increments its *cwnd* by one SMSS every time new *ACK* is received.

Recent studies (Benson et al., 2010) have revealed that most of the TCP connections are short lived (mice), yet a few long-lived connections carry the most the internet traffic (elephant). Usually, most of the short lived connections terminated very early, even before the slow start reaches its steady state. For example, TCP NewReno needs 100 seconds to reach its optimal *cwnd* over a bath with bandwidth

UNIVERSITI SAINS ISLAM MALAYSIA
ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA

100Mbps, *RTT* 100ms, and *ssthresh* of 32kbytes (Wang et al., 2004). Therefore, we propose to accelerate the *cwnd* growth, depending on the current bandwidth utilization to shorten the slow start phase period.

4.1.2.1 Faster Start

In this section, we introduced a new method to improve the slow start phase. We take advantage of the TCPW bandwidth estimation method presented in section 4.1.1. We proposed using the estimation to control the *cwnd* growth depending on the available bandwidth (*EBW*) calculated in equation (4.5). Therefore, we calculate the *EBW* for every *ACK* received, and then we compare it with the current *FlightSize*. The difference will indicate the bandwidth utilization. Depending on the bandwidth utilization, we increment the *cwnd* according to the following algorithm:

```
If (BytesInFlight() >=ssthresh) /* exit slow start and start
congestion avoidance */

   cwnd+=SMSS*SMSS/cwnd;        /*linear   congestion   avoidance
increment*/

elseIf (EBW div BytesInFlight() >2)

   cwnd+=EBW div BytesInFlight();

else /*normal slow start increment*/

  cwnd+=SMSS;;

endif
```

The new modification tries to exit the slow start with *cwnd* close to the optimal congestion window size. Therefore, the linear growth of *cwnd* during the congestion avoidance phase will continue with *cwnd* so close to the link capacity. As a result, the TCP sender side will preserve better *cwnd* size during the transmission period. The following section presents the results.

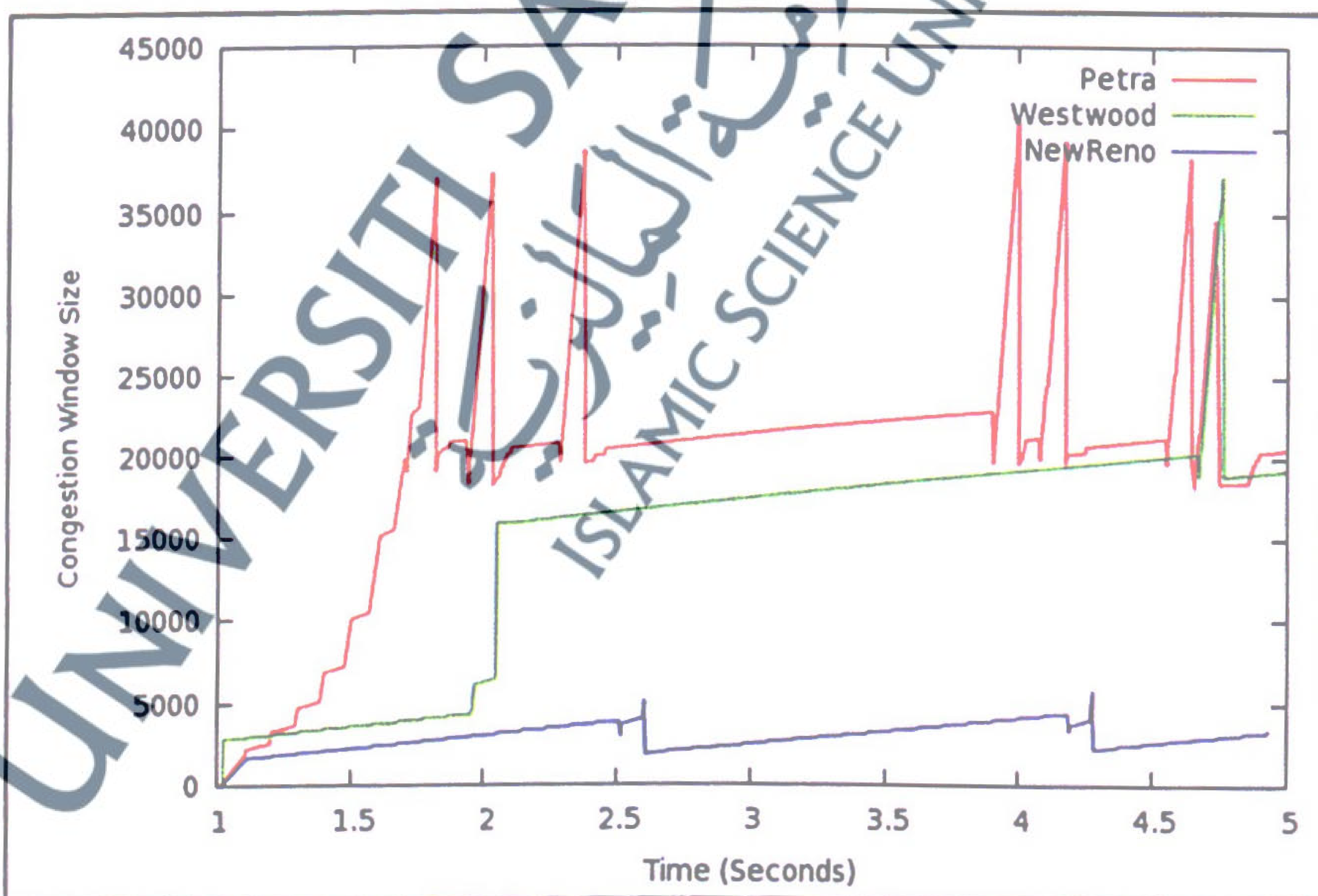
UNIVERSITI SAINS ISLAM MALAYSIA
ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA

4.1.2.2 Performance Evaluation

We have implemented the proposed modification in the previous section using ns-3. We modified the TCPWestwood: NewAck procedure in TCPW implementation as presented in the algorithm in section 4.1.2.1. For the first experiment, we used the same topology in section 4.1.1.1 to compare the *cwnd* size, and the total throughput achieved using the new modification with TCP NewReno, and TCPW. Figure 4.5 shows the congestion window size of the three implementations during the first five seconds of the experiment period.

In figure 4.5, we can clearly see how quickly the new modification achieved the optimal *cwnd* compared to TCPW and NewReno. Also, we can see the greater the *cwnd* values recorded by the new modification throughout the simulation period, compared to the other implementations.

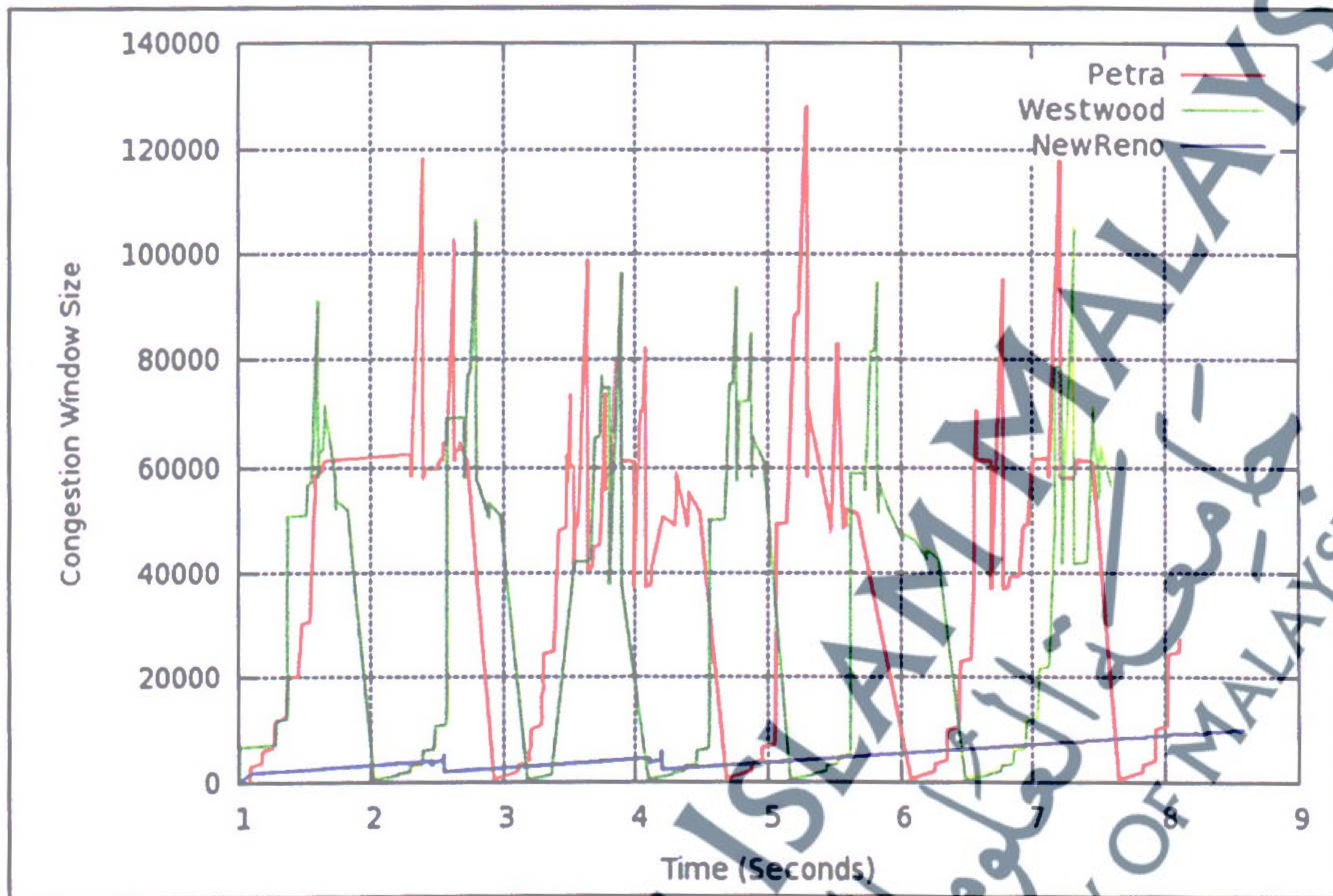
Figure 4.5: Congestion Window Size (bottleneck bandwidth=2Mpbs)



UNIVERSITI SAINS ISLAM MALAYSIA
ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA
جامعة العلوم الإسلامية الماليزية

For the second experiment, we changed the bottleneck bandwidth to 6Mbps to eliminate any restrictions on *cwnd* growth. We plotted the *cwnd* behaviour of the three implementations in figure 4.6.

Figure 4.6: Congestion Window Size (bottleneck bandwidth=6Mpbs)



In figure 4.6, TCPW reaches a larger *cwnd* size more quickly; however, the new implementation preserves better *cwnd* size for a longer period. The total throughput achieved by TCPW during the simulation period was 0.66Mbytes, whereas the new modification achieved better throughout as it recorded 1.34Mbytes, while NewReno recorded 0.34Mbytes.

We extended our experiments to use different values of bottleneck bandwidth, packet error rate (PER), and propagation delay. We recorded the total throughput achieved during 10 seconds as a simulation period using the three implementations in table 4.3.

UNIVERSITI SAINS ISLAM MALAYSIA
ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA

Table 4.3: Throughput Results

Experiment Number	Bottleneck Bandwidth (Mbps)	Packet Error Rate (PER)	Propagation Delay (ms)	PETRA (Mbytes)	TCPW (Mbytes)	New Reno (Mbytes)
1	2	0.005	45	**1.187**	1.023	0.33
2	4	0.005	45	**1.99**	**1.99**	0.34
3	6	0.005	45	**1.34**	0.66	0.34
4	8	0.005	45	**0.835**	0.66	0.34
5	10	0.005	45	**0.83**	0.823	0.342
6	2	0.0075	45	0.915	**0.947**	0.229
7	2	0.009	45	0.879	**0.901**	0.196
8	2	0.01	45	0.891	**0.926**	0.196
9	2	0.025	45	**0.422**	0.384	0.136
10	2	0.005	60	**0.924**	0.813	0.239
11	2	0.005	30	**1.237**	1.173	0.642
12	2	0.005	20	**1.354**	1.268	0.86
13	2	0.005	10	**1.4**	**1.4**	1.18

The new modification recorded better throughput results as we can see in table 4.3. However, TCPW performs better once PER is increased, as we can see in experiments six, seven, and eight. One reason for this could be the advantage of using the *EBW* method when every timeout occurred to set the value of *ssthresh*.

4.2 Enhanced Fast Retransmission and Fast Recovery mechanisms

In this section, we introduce two enhancements to fast retransmission and fast recovery mechanism presented in RFC 2581. Firstly, we introduce a new modification to make the fast retransmission procedure faster. Secondly, we introduce new metrics to improve the recovery of *cwnd*. For each modification, we implemented and evaluated the proposed methods using ns-3. To implement the new modifications, we modified the TCPW implementation included in ns-3 models library.

UNIVERSITI SAINS ISLAM MALAYSIA ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA

4.2.1 Fast Retransmission and Fast Recovery mechanism

A TCP connection uses a unique sequence number attached to every sent packet. A TCP receiver side should send *ACK* to acknowledge every packet received. Once a receiver side receives an out of order packet, it should immediately send a duplicate *ACK* to inform the sender side if a packet is missing. A duplicate *ACK* can be caused by several network faults. Firstly, they might be caused by a dropped packet. In this case, all packets received after the dropped packet will trigger duplicate *ACKs*. Secondly, some duplicate *ACKs* could be a cause of re-ordering the packets by network intermediate nodes. In this case, duplicate *ACKs* will be triggered until the delayed packet is received. Finally, duplicate *ACKs* could be a cause of duplicating *ACKs* or packets by the networks. Rarely, duplicate *ACKs* caused of receiving damaged packets. The TCP at the sender side should retransmit any duplicate *ACKs* packets as soon as they arrived. Therefore, the fast retransmission method is used to prevent the TCP from waiting for the Timeout timer expiration.

The fast retransmission method was used to detect and repair packets loss based on duplicate *ACKs*. Usually, the TCP sender awaits the arrival of three duplicate *ACKs* to trigger a fast retransmission procedure. On receiving the 3rd duplicate *ACK*, this indicates that a packet has been lost, and therefore the TCP retransmits which packet appears to be lost without waiting for the retransmission timer to expire. After retransmitting the lost packets, the fast recovery algorithm controls the transmission of new packets until the non-duplicate *ACK* arrives. Fast recovery assumes that receiving three duplicate *ACKs* indicates the link is not congested; therefore, the sender can continue transmitting new packets, but with a reduced *cwnd* size. The following steps describe Reno's fast retransmission and fast recovery in more detail:

UNIVERSITI SAINS ISLAM MALAYSIA
ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA

- When the 3rd duplicate *ACK* is received, the sender should set the *ssthresh* as in the following:

$$ssthresh = \max(FlightSize / 2, 2*SMSS) \quad (4.6)$$

- Retransmit the lost packet and set the *cwnd* as follows:

$$cwnd = ssthresh + 3*SMSS \quad (4.7)$$

  Where, 3*SMSS is assumed to be buffered at the receiver side during the time the 3rd duplicate *ACK* was arrived.

- For each additional duplicate *ACK*, the sender will increment its *cwnd* by one SMSS.

- Transmit the current *cwnd* if allowed by the recent value of the receiver's advertised window.

- Once the new *ACK* is received, set the *cwnd* to the *ssthresh* value computed in equation (4.6).

A modification to the Reno fast retransmission and fast recovery has been introduced in RFC 2582 as NewReno. The modification includes detecting and recovering multiple losses in a single sending window without using a selective *ACK* (SACK). A slight modification to the previous steps includes adding a new variable, '*recover*,' to record the highest sequence number transmitted just before receiving the 3rd duplicate *ACK*. After retransmitting the lost packet/packets, NewReno uses this variable to check if the received new *ACK* acknowledges all packets up to and including the '*recover*'. This means that the new *ACK* acknowledges all the packets between the lost packet retransmission and the receipt of the 3rd duplicate *ACK*. However, if the *ACK* does not acknowledge all the sent packets (partial *ACK*), then

the first unacknowledged packet is retransmitted and the *cwnd* is deflated to the amount of the *Acked* packets.

Many modifications have been proposed to enhanced fast retransmission and fast recovery algorithm, for example RFC 3042 (Allman et al., 2001). This modification introduced a solution for the small *cwnd* size by triggering the fast retransmission just after receiving the 2nd duplicate *ACK*. Another modification is presented in RFC 3522 (Ludwig & Meyer, 2003). This modification takes advantage of the Timestamps defined in RFC 1323. In this modification, a new algorithm is introduced to back out of fast recovery by restoring the congestion control state for the TCP sender side.

In the following two sections, we introduce two new modifications to the NewReno fast retransmission and fast recovery mechanism.

4.2.2 Faster Retransmission algorithm

In this section, we introduced a new modification to quicken the fast retransmission algorithm. As discussed in the previous section, the TCP sender waits receiving three duplicate *ACKs* to trigger the fast retransmission procedure. However, if the last round trip time (*LastRTT*) is too long, then the sender will keep waiting for an unachievable condition. Thus, the retransmission timer will expire before fast retransmission is triggered. Therefore, we proposed checking if the sender can receive the three duplicate *ACKs* based on the *LastRTT*. The following pseudo code illustrates the method:

```
If (count=3 && !infastRecovery) /* same as the standard
fast retransmission*/

  If (BytesInFlight() > ssthresh)
```

UNIVERSITI SAINS ISLAM MALAYSIA
ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA

```
    cwnd=ssthresh;
    infastrecovery =true;
    endif

  else If (LastRTT *(4-count) > Timeout &&!infastRecovery )
  /*4 used to check with every duplicate ACKs*/

    If (BytesInFlight() > ssthresh)
    cwnd=ssthresh;

    else
    infastrecovery =true; */
    endif

  do retransmit;
  endif
```

We have implemented the proposed algorithm using a modified version of the NewReno implementation provided by ns-3. We have taken advantage of the TCPW's bandwidth estimation to calculate *ssthresh* for every packet lost. The following section presents the results of using the new modification.

4.2.2.1 Performance Evaluation

In this section, we discussed evaluating the new modification presented in the previous section in terms of throughput and congestion window size. We used the same topology presented in 4.1.1.1. To implement the new modification, we modified the ns-3's TCPW implementation. Also, we used ns-3 to obtain the results of the throughput and congestion window of TCPW, NewReno, and the new modification. For the first experiment, we used the same parameters listed in table 4.1. We ran the simulation for a complete 100 seconds. Figure 4.7 shows the *cwnd* size of the three implantations in the first ten seconds.

UNIVERSITI SAINS ISLAM MALAYSIA
ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA

Figure 4.7: Congestion Window of PETRA, Westwood, and NewReno implementations.

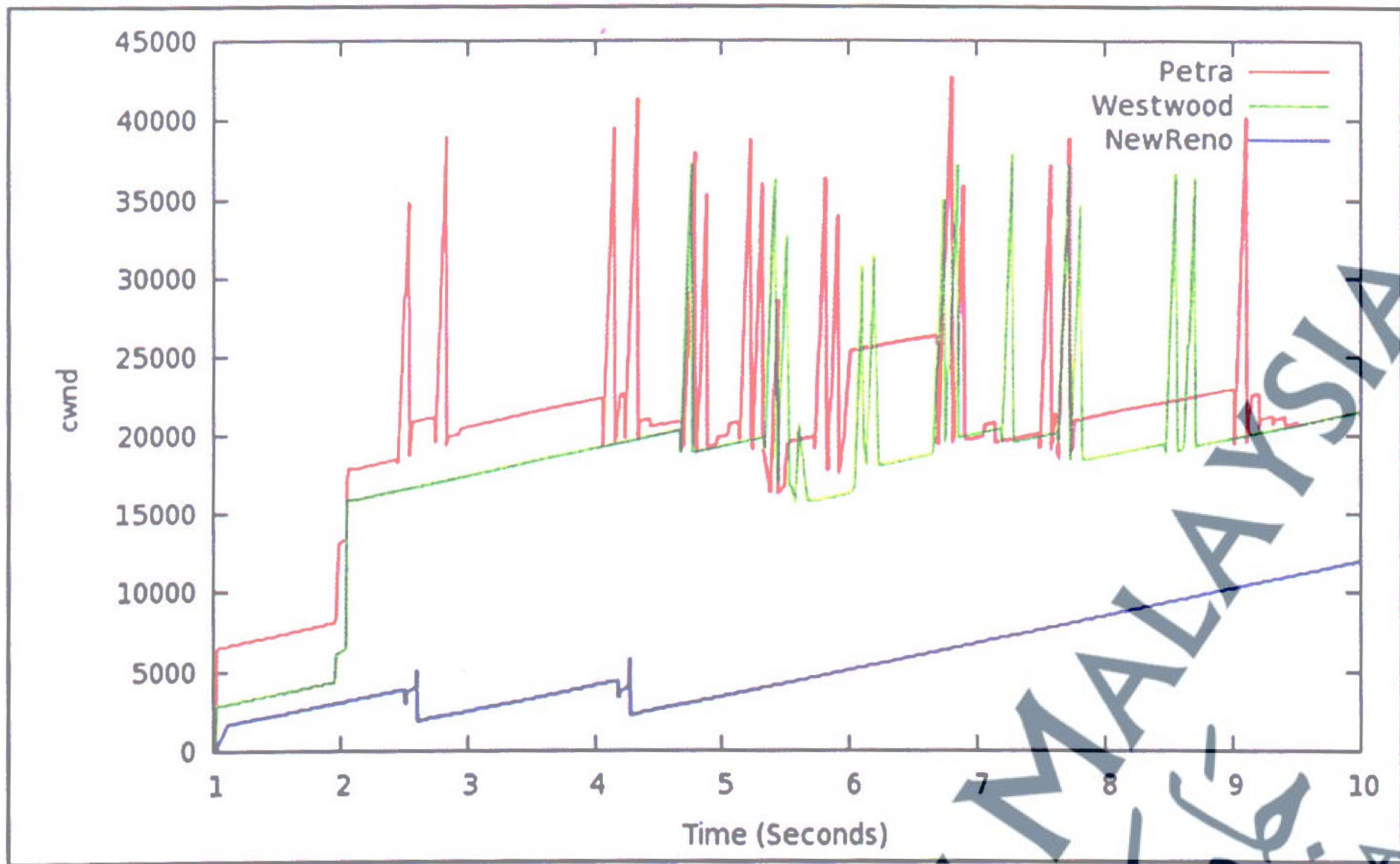
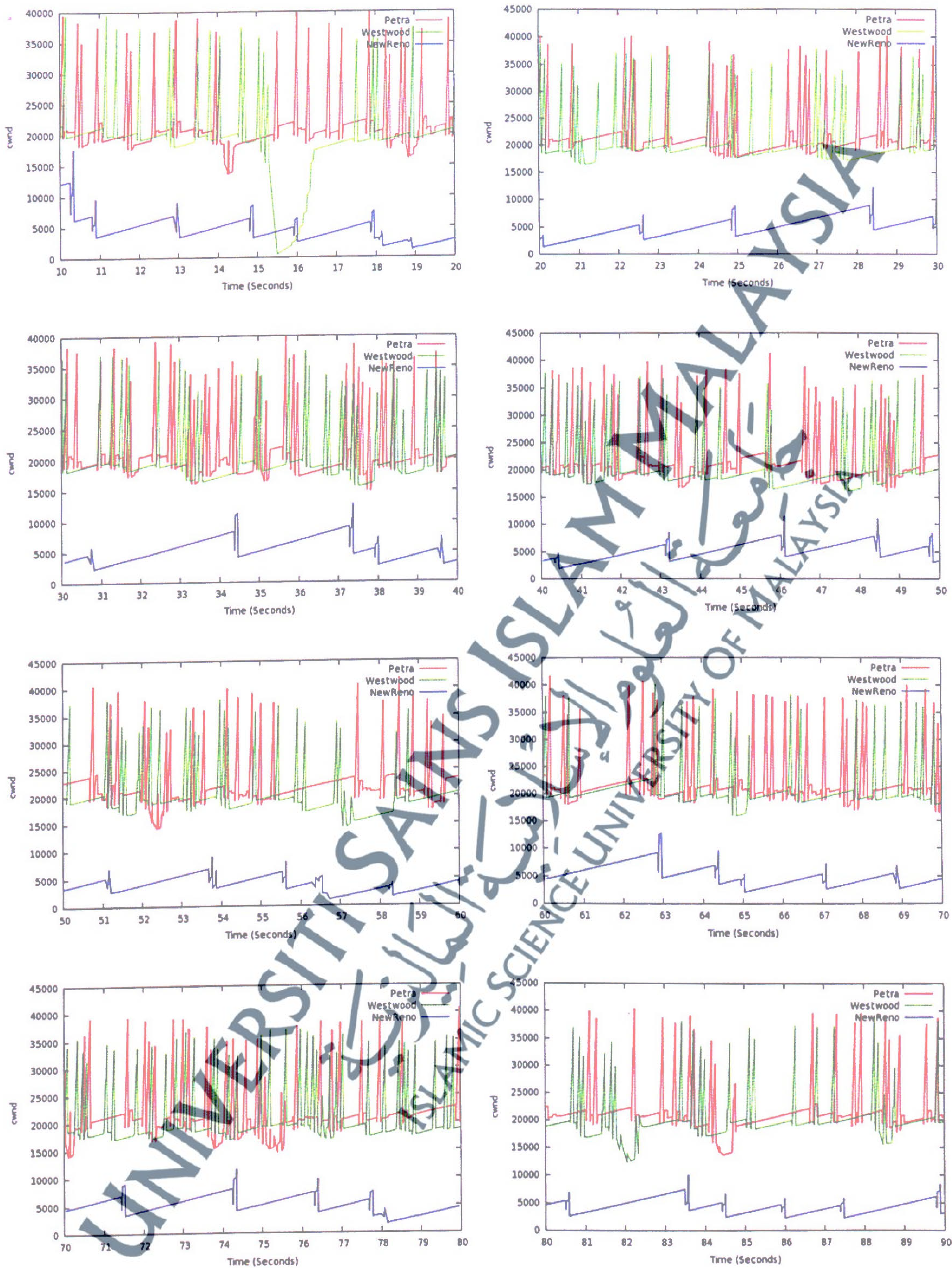


Figure 4.7 shows the improved *cwnd* size and the new modification recorded compared to other implementations. Figure 4.8 shows the *cwnd* recorded by the three implementations from 20 seconds to 90 seconds of the simulation period. We have plotted each ten second frame in a separate figure to make the comparison easier. From this, we can see how rare the connection reset is with the *cwnd* using the new modification, as it minimizes the retransmission timer expiry. Another observation which can be seen in figure 4.8 is that the new modification almost preserves the same behaviour all the simulation time. Also, we can see the smaller *cwnd* size recorded by the NewReno.

UNIVERSITI SAINS ISLAM MALAYSIA
ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA

Figure 4.8: Congestion Window (10-90 Seconds)



UNIVERSITI SAINS ISLAM MALAYSIA
ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA

For the second experiment, we used the same topology in the first experiment but we used different values of bottleneck bandwidth, PER, and propagation delay. We calculate the total throughput achieved using the three modifications and we recorded the results in Table 4.4.

Table 4.4: Throughput results

Experiment Number	Bottleneck Bandwidth (Mbps)	Packet Error Rate (PER)	Propagation Delay (ms)	PETRA (Mbytes)	TCPW (Mbytes)	New Reno (Mbytes)
1	1	0.005	45	**9.293**	8.94	4.19
2	2	0.005	45	**17.96**	16.84	4.27
3	3	0.005	45	**25.00**	22.81	4.30
4	4	0.005	45	**27.86**	27.03	4.35
5	2	0.0075	45	**16.24**	15.02	3.47
6	2	0.009	45	**15.64**	13.68	3.17
7	2	0.01	45	**15.28**	13.47	3.01
8	2	0.025	45	**8.71**	6.44	1.26
9	2	0.005	60	**17.19**	15.80	3.32
10	2	0.005	30	**18.00**	17.24	6.12
11	2	0.005	20	**18.45**	18.35	8.93
12	2	0.005	10	**19.30**	18.50	15.19

We can see in Table 4.4 that better throughput values were achieved using the new modification over TCPW and NewReno. One important observation is that over lower propagation delay values, the NewReno performance is obviously improved. However, the new modification performed better for all 12 experiments.

4.2.3 Enhanced TCPW's Fast Recovery

In this section, we introduce a modification to the standard fast recovery algorithm presented in NewReno. As discussed in previous sections, after every fast retransmission event, a TCP sender enters a fast recovery procedure awaiting the arrival of new *ACK*. If the *ACK* acknowledged all sent packets up to and including the

UNIVERSITI SAINS ISLAM MALAYSIA
ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA

'*recover*' –the sequence number of the packet where the 3rd duplicate *ACK* is received –then NewReno fast recovery sets the *cwnd* value to either:

1. min (*ssthresh*, *FightSize* + SMSS)
2. max (*FlightSize* / 2, 2*SMSS)

Then it exits the fast recovery procedure. If the *ACK* is partial –i.e. it has acknowledged some but not all packets –then the first unacknowledged packet is retransmitted and the *cwnd* deflated by the number of acknowledge packets, and 1 is SMSS is added back to the *cwnd* for every duplicate *ACK* which arrives. This is repeated until a new *ACK* is received that is an accumulative *ACK*. Then the fast recovery is exited.

It seems that the *cwnd* is likely to be inflated considerably for every duplicate *ACK* arriving during the fast recovery phase, where the TCP tries to prevent a predicated congestion situation. Therefore, we propose to use the last round trip time values along with the current *EBW* to confirm a congestion case. To do so, we compare the last value of the *RTT* with estimated round trip time calculated by the *rtt-estimator* in (Paxson et al., 2011). The following pseudo code explains the method:

```
DiffRtt=LastRtt - ERTT;
If (BytesInFlight() > ssthresh && DiffRtt >=0)
  cwnd= ssthresh;  /* we rest the cwnd size to ssthresh*/
infastrecovery=true;
DoRetransmit ( );
Endif
```

UNIVERSITI SAINS ISLAM MALAYSIA
ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA

The main idea in this code is to prevent fast recovery from resting the *cwnd* size unless real congestion is shaped based on the *EBW* and *RTT* values. We try to preserve a bigger *cwnd* size to retransmit after every partial *ACK*. The following section presents the results of implementing the new modification.

4.2.3.1 Performance Validation

To implement the proposed modification presented in the previous section we used a modified version of the TCPW implementation included in ns-3. We have modified the TcpWestwood::DupAck function as illustrated in the proposed pseudo code. To validate the proposed modification, we used the same topology presented in 4.1.1.1 as it is the same the topology used to introduce the original TCPW. We conducted 14 different experiments with different values of bottleneck bandwidth, packet error rate, and propagation delay. Then, we compared the throughput achieved using the new modification with TCPW and NewReno. Table 4.5 shows the results.

Table 4.5: Throughput results

Experiment Number	Bottleneck Bandwidth (Mbps)	Packet Error Rate (PER)	Propagation Delay (ms)	Petra (Mbytes)	TCPW (Mbytes)	New Reno (Mbytes)
1	1	0.005	45	**9.32**	8.94	4.19
2	2	0.005	45	**17.19**	16.84	4.27
3	3	0.005	45	**23.86**	22.81	4.30
4	4	0.005	45	**27.52**	27.03	4.35
5	2	0.0075	45	**15.2**	15.02	3.47
6	2	0.009	45	**14.00**	13.68	3.17
7	2	0.01	45	13.36	**13.47**	3.01
8	2	0.025	45	**6.77**	6.44	1.26
9	2	0.005	60	**16.01**	15.80	3.32
10	2	0.005	30	**18.20**	17.24	6.12
11	2	0.005	20	**19.04**	18.35	8.93
12	2	0.005	10	**19.84**	18.50	15.19
13	3	0.005	10	**29.23**	28.30	17.21
14	4	0.005	10	**37.75**	36.21	17.74

In Table 4.5, the new modification shows better throughput results compared to other implementations. One observation is that the new modification shows better performance over low propagation delay values, as we can see in experiments 10-14.

4.3 Enhanced TCP Retransmission Procedure

In this section, we present a new modification to the TCP retransmission procedure. The following section introduces the standard retransmission procedure with the TCP. Then, we discuss the TCPW retransmission procedure. We presented our proposed modifications followed by the performance evaluation. ns-3 has been used to evaluate and implement the new modification. Two network performance metrics were used to compare the results of the new modification with TCPW and NewReno. These metrics are the total throughput and the congestion window size.

4.3.1 The Standard Retransmission Procedure in NewReno

The TCP is a reliable transmission protocol. One fundamental of TCP is that it makes sure every sent packet has reached its destination. To do so, the TCP receiver must acknowledge every received packet using either piratical or cumulative *ACK*. The TCP uses duplicate *ACKs* when out of order packets arrive at the TCP receiver side to inform the sender that an out of order packet has been received and which packet is expected. The standard TCP uses the fast retransmission algorithm to detect and recover packet loss. The fast retransmission uses the arrival of 3 duplicate *ACKs* to confirm a packet loss. Therefore, after the 3rd duplicate *ACK*, the sender retransmits what appears to have been lost. However, if the sender does not receive the third duplicate *ACK* within a period of time called the 'retransmission timer', then the sender performs a hard congestion avoidance procedure as follows:

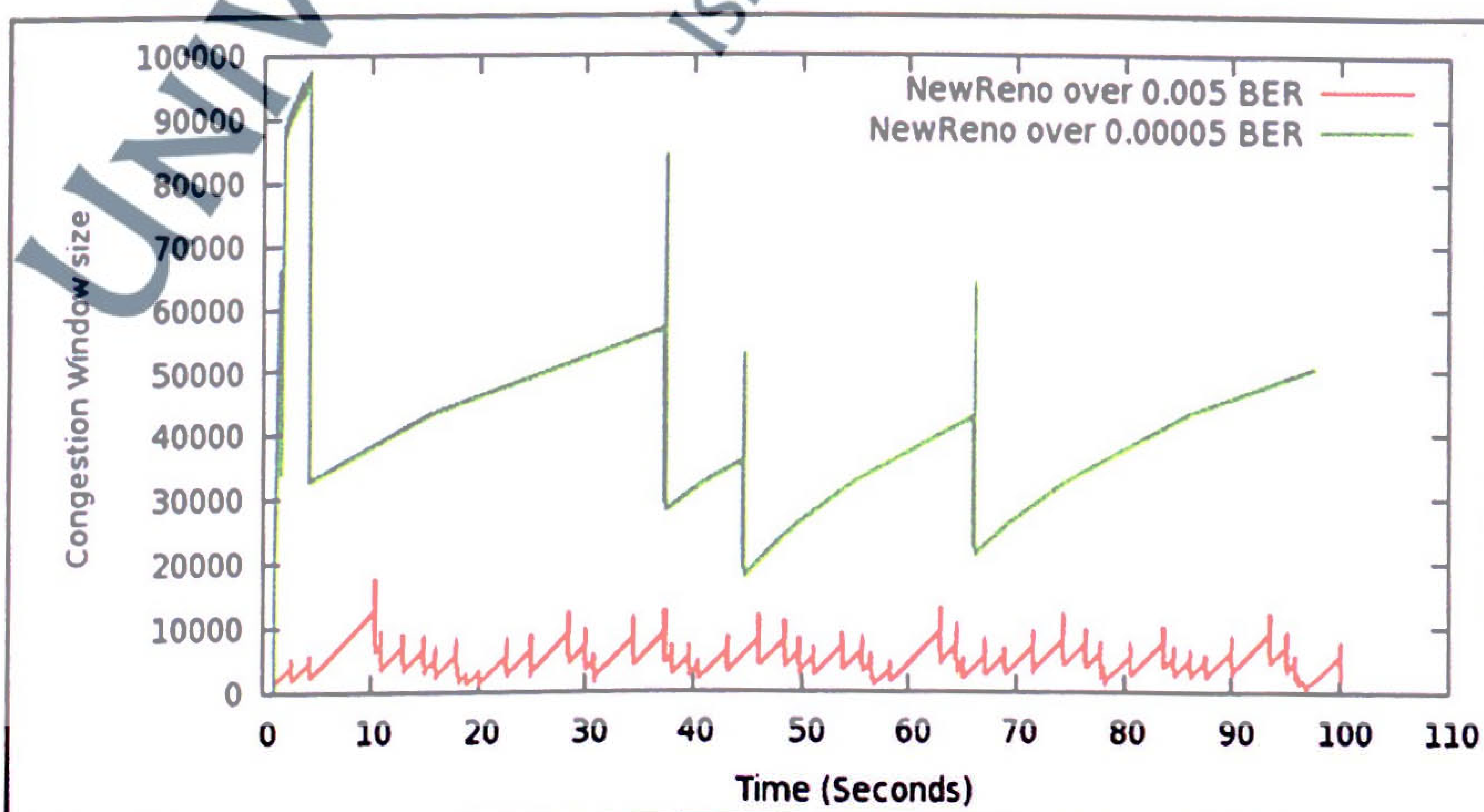
UNIVERSITI SAINS ISLAM MALAYSIA
ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA

- Set the *ssthresh* value to half the current *filghtSize*
- Reset the *cwnd* size to 1 SMSS
- Double the current value of *RTO*
- Trigger the slow start phase.

Over a high bit error rate (wireless links), this procedure will cause serious performance degradation in TCP connections, because a packet loss event is considered as a congestion situation. For example, a wireless link with PER of 0.005 and propagation delay of 20ms will trigger the same procedure six times every 100 seconds. In other words, the connection will retrace back to the initial state every 16.67 seconds. To illustrate the problem, we used ns-3 to record the congestion window size using a NewReno implantation over a wireless link with two different PER values: 0.005 and 0.00005. Figure 4.9 shows the results.

In figure 4.9, we can easily figure the difference between the two *cwnd* sizes. Also, we calculated the total throughput achieved in the two scenarios, and as expected in the first experiment with (PER=0.00005) NewReno transmitted 20.03Mbytes, whereas, in the second experiment NewReno transmitted 4.27Mbytes.

Figure 4.9: NewReno Congestion Control



UNIVERSITI SAINS ISLAM MALAYSIA
ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA

4.3.2 TCPW Retransmission Procedure

TCP Westwood introduced a new End-to-End mechanism to calculate the estimated bandwidth by monitoring the returning *ACKs* at the TCP sender side (Mascolo et al., 2001). TCPW uses the estimation to properly set the *ssthresh* value with every loss event. Using TCPW over high bit error networks shows significant improvement compared with many TCP implementations, including NewReno, Vegas, SACK, and etc. Although TCPW estimates the current link bandwidth, it still performs as NewReno in a timeout procedure. Firstly, TCPW sets the *ssthresh* value to the current estimation, then resets the *cwnd* size to one SMSS, and finally multiplies the retransmission time out (*RTO*) value. The following pseudo code shows the procedure:

```
If (coarse Time Out)
    ssthresh = current BWE;
    cWnd = SMSS;
    RTO = RTO*2;
    Retransmission();
Endif
```

UNIVERSITI SAINS ISLAM MALAYSIA
ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA

4.3.3 Modified TCPW Retransmission Procedure

In this section, we introduce a new modification to the TCPW retransmission procedure. The modification tries to prevent unnecessary congestion avoidance procedures every time a packet's *RTO* expires. To do so, we suggest using the current bandwidth estimation besides the latest values of *RTT* to confirm a congestion situation. We believe that by monitoring the *RTT* values we can probe the current link

status. A large *RTT* values indicates a congested links, whereas short *RTT* values indicate an uncongested link. We take advantage of the *RTT* Estimator proposed in (Paxson et al., 2011) to compare the last *RTT* values with the recent estimated *RTT*. The following pseudo code shows the proposed modification:

```
If (coarse Time Out)

    If (BytesInFlight() < EBW/2 && Last_RTT <ERTT )
        cWnd = cWnd + SMSS * SMSS/ cWnd;

    Else if (Last_RTT <ERTT)
         cWnd = ssthresh; /*Congestion Avoidance*/

    Else /* Congestion */
    cWnd=1;
    RTO=RTO*2;
    Endif

Retransmission();
Endif
```

As we can see above, we propose to increment the *cwnd* instead of resetting it to one SMSS, if both the estimated bandwidth and the *lastRTT* indicate no congestion. We implemented the new modification using the TCPW implementation in ns-3. We have modified the class TCPWestwood::Retransmit as proposed above. The next section discusses the performance evaluation of the proposed modification.

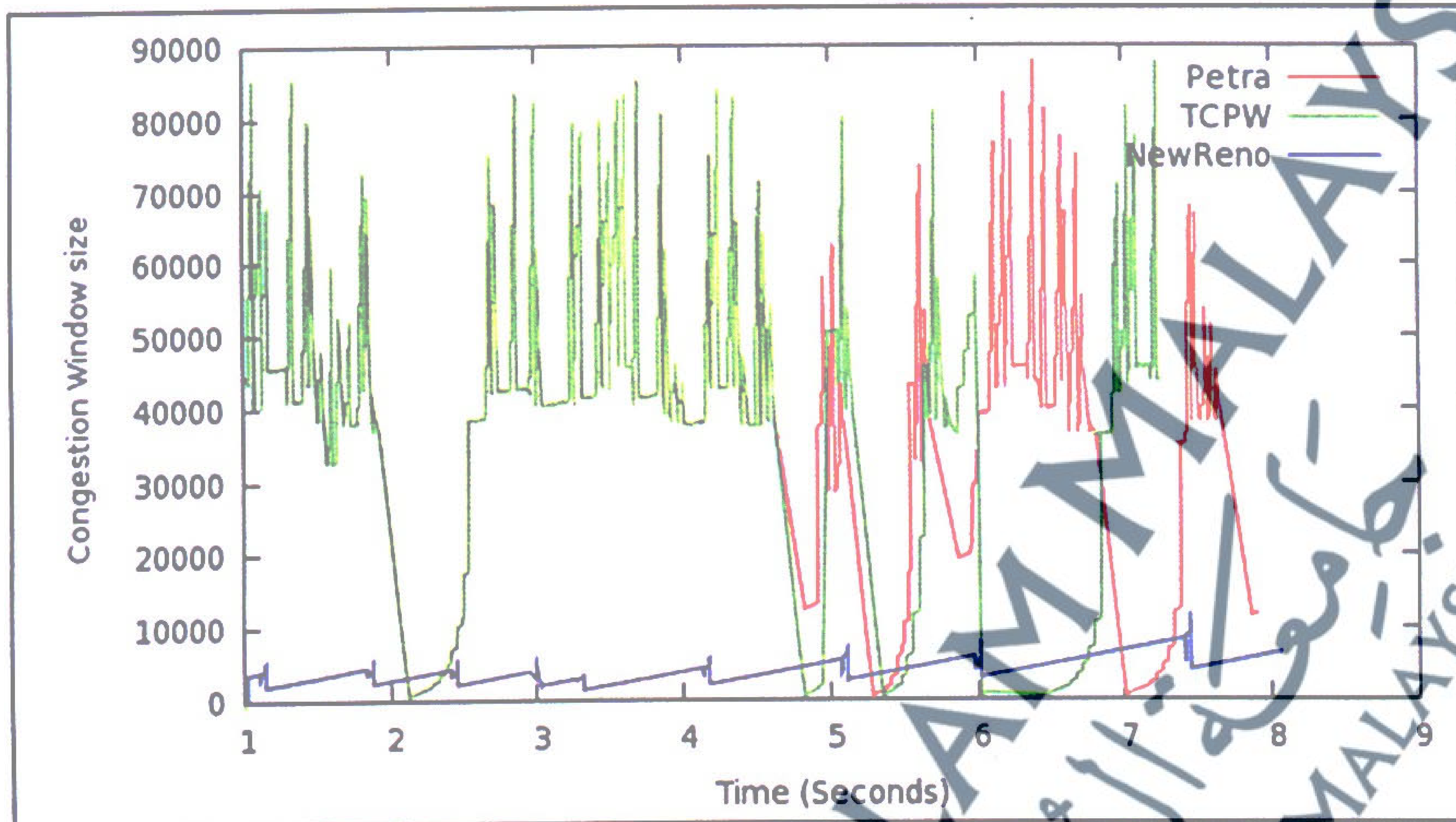
4.3.4 Performance Evaluation

This section discusses evaluating the retransmission modification proposed in section 4.3.3. To this end, we used the simulation methodology to test the performance of the new modification and then to compare its results with TCPW, and NewReno. We used ns-3 as a simulator to obtain the results of total throughput and the congestion window. For the simulation topology, we used the same topology used to introduce the original TCPW (Mascolo et al., 2001). The topology used is discussed in detail in

UNIVERSITI SAINS ISLAM MALAYSIA
ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA

section 4.1.1.1. For the first experiment, we used the same parameters as listed in table 4.1. We plotted the congestion window behaviour for the three implementations in figure 4.10.

Figure 4.10: Congestion Window size



UNIVERSITI SAINS ISLAM MALAYSIA
ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA

As we can see in the figure above, the new modification behaved like TCPW until the retransmission timer expired (at 4.6 and 5.8 seconds), we can see how the new modification preserved a higher *cwnd* size rather than resetting the *cwnd* to one SMSS.

We extended our simulation experiments with different values of bottleneck bandwidth and propagation delay, and different PER values over access link. We recorded the total throughput achieved by three implementations in table 4.6.

Table 4.6: Throughput Results

Experiment Number	Bottleneck Bandwidth (Mbps)	Packet Error Rate (PER)	Propagation Delay (ms)	Petra (Mbytes)	TCPW (Mbytes)	New Reno (Mbytes)
1	2	0.005	45	**16.84**	**16.84**	4.27
2	4	0.005	45	**27.28**	27.03	4.33
3	8	0.005	45	**10.00**	9.06	4.35
4	10	0.005	45	**10.34**	9.48	4.35

5	2	0.0075	45	**15.11**	15.02	3.47
6	2	0.009	45	**14.02**	13.68	3.17
7	2	0.01	45	**13.81**	13.47	3.01
8	2	0.025	45	**6.81**	6.44	1.26
9	2	0.005	30	**17.45**	17.24	6.12
10	2	0.005	20	**18.50**	18.35	8.93
11	2	0.005	10	**19.03**	18.50	15.19
12	2	0.0075	20	**17.11**	16.87	7.17
13	4	0.0075	20	**30.15**	29.77	7.34
14	8	0.0075	20	**43.69**	40.6	7.51
15	10	0.0075	20	**42.02**	39.16	7.55

In the above table, the new modification recorded higher throughput values compared to other implementations, whereas NewReno recorded the lowest throughput values in all 15 experiments.

4.4 Summary

In this chapter, we have introduced three main modifications to TCP Westwood. Firstly, we have introduced two modifications to enhance the slow start phase. The first modification was used to properly set the initial *ssthresh* value in the initial slow start phase using the average method to smooth the bandwidth estimation. The second modification was aimed at accelerating the *cwnd* growth depending on the link status.

Secondly, we introduced two modifications to the fast retransmission and fast recovery mechanism. The first modification introduced a faster retransmission method to accelerate the fast retransmission procedure. The second modification proposed using a congestion check method every time partial *ACKs* arrived during the fast recovery procedure. The key idea of this modification is to preserve greater *cwnd* size after every fast retransmission and fast recovery procedure.

UNIVERSITI SAINS ISLAM MALAYSIA
ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA

Finally, we modified the retransmission procedure in TCPW to prevent it from unnecessarily resting its *cwnd* with every time out unless real congestion is shaped. Therefore, we used both the current bandwidth estimation and the last round trip time values to confirm congestion cases. We do not claim that we have introduced the optimal solution here, but we have improved the TCPW performance.

The performance evaluations of the presented modifications were conducted using ns-3. For each modification, we compared the throughput and congestion window size with TCPW and NewReno variants. As a result, the new modifications show better performance compared to the two variants.

The next chapter will discuss implementing the modified TCPW over LTE Networks. Also, we extend the evaluation process to include more performance metrics: Jitter, average delay, and fairness with other TCP flows.

UNIVERSITI SAINS ISLAM MALAYSIA
ISLAMIC SCIENCE UNIVERSITY OF MALAYSIA