Bhargav Srinivasan (sbhargav3), Om Kale (okale6),
Vinish Chamrani (vchamrani3), Vivian Dsilva (vdsilva3)

Project 4: Comparing TCP Implementations

PROJECT SETUP

This project involved comparing the three flavors of TCP, namely Tahoe, Reno and NewReno, by performing an experiment that involved comparing the way in which each of these implementations react to packet drops at the receiver. The topology used for this experiment is the same as what was used in the Simulation based Comparison of Tahoe, Reno and SACK paper by Kevin Fall and Sally Floyd.

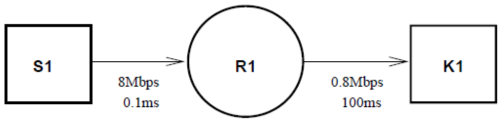

Figure 1: Simulation Topology

The nodes S1, R1 and K1 were placed as shown in the topology. The **PointToPointHelper** was used in the creation of the links as shown, one with a data rate and delay of 8Mbps, 0.1ms and 0.8Mbps, 100ms respectively. A Droptail queue installed on the links with a queue size of 64000 bytes. The node R1 in the middle had two ASCII traces set on either side of it. Therefore, packets at both the ends of the node were recorded in the trace, and the trace was further processed at the end of each run in order to plot the graphs.

In our implementation, we use the **BulkSendApplication** in order to create **one flow** which sends TCP packets from the source to the receiver.

The **ns3::ReceiveListErrorModel** has been used in order to introduce errors on specific packet numbers, which will be dropped at the receiver before it reaches the TCP layer such that it activates the congestion control mechanisms. This is our mechanism for dropping packets at the middle node, similar to what was mentioned in the paper.

We added two traces at both ends of the node R1 in order to record the packets going in and coming out of the queue. These traces are assigned to both the node devices containing the middle node.

PERL scripts are used to parse the data from the traces. The scripts also normalize the packets (with respect to packet size) and perform the mod 60 operation on them. These are then plotted using gnuplot, with a size of 1280x640. The X-axis label starts from 0 seconds and represents time in steps of seconds, and the Y-axis shows the packet numbers which are normalized by packet size and a modulo 60 operation is done.

The PERL scripts which were used are included in this report in Appendix A.

MAJOR DIFFERENCES

- The difference in the packet numbers being dropped for each case.

Case	Packet Number (ours)
One Packet Drop	10
Two Packet Drops	14,15
Three Packet Drops	14,15,16
Four Packet Drops	14,15,16,17

Table 1: Parameters for the Simulation

- Used only one flow instead of 3.
  - We do not have the data rate of the flows.
  - If we introduce 3 flows and set the data rate to be the same as the first flow, it results

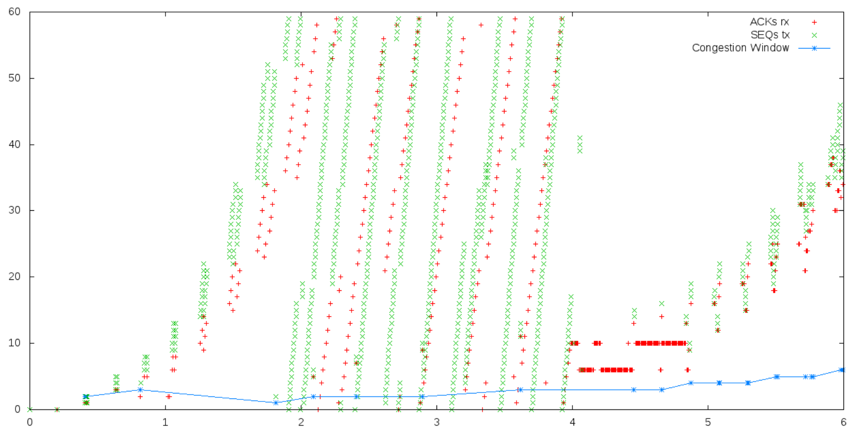

Figure 2: Output after introducing 3 flows

    in large queueing delays, as shown in figure 2, skewing the results. (Appendix B, Case 2)
- Queueing delay was negligible.
  - Additionally, we tried plotting the traces after enabling 2 flows with the same data rate, however we observed that the queuing delay was negligible and the points overlapped each other. (figure in Appendix B, Case 1)
- Our Graph starts from zero seconds, instead of one second as shown in the paper.
- Congestion window has been plotted.
- Time steps in ns3 are 1µs whereas in the paper the time step at which results were recorded was 1ms.

HOW PACKET LOSSES ARE HANDLED BY THE IMPLEMENTATIONS

TCP Tahoe recovers using fast retransmit and then entering slow start regardless of the number of packets dropped. The delay it takes while performing the slow start back to half the previous congestion window will cause some dropped packets to be retransmitted in the case of multiple losses.

TCP Reno is optimal in the case of single packet loss, it does a Fast retransmit and Fast Recovery and recovers quickly. In the case of two dropped packets, it goes through Fast Retransmit and Fast Recovery twice in succession, thereby reducing the congestion window two times for each dropped packet. However in the case of three and four consecutive dropped packets, the retransmission timer (RTO) is triggered and the sender waits for a period of time before transmitting again.

TCP NewReno works similar to Reno in the one packet drop case, it does a Fast Retransmit/Fast Recovery. It is optimized to handle multiple packet drops, and unlike Reno it does not timeout when it encounters three or four consecutive dropped packets. NewReno stays in the Fast Recovery phase and transmits one dropped packet per round trip time each time it receives a partial ACK from the receiver. The only constraint with New Reno is that it can transmit only one dropped packet per RTT in the Fast Recovery phase which results in a delay in transmitting the packets.

The paper talks about TCP SACK, however we haven't implemented this in our experiment. In TCP SACK, where selective acknowledgements are used and the sender knows exactly which packets to retransmit and transmits only when the number of packets in flight are less than the congestion window.

ONE PACKET LOSS

In TCP Tahoe, packets 1-10 are sent without errors and then the congestion window increases to a value of 11 MSS, and it is still in the Slow Start phase. After, the 10th packet gets dropped at the receiver, the sender receives three duplicate ACKs for this packet and the congestion window drops to the value of 1 MSS (512 bytes). The Slow Start threshold (ssthresh) is set to 5 and then the sender enters the slow start phase again as shown, the congestion window starts increasing exponentially till it hits ssthresh and after that it continues additive increase multiplicative decrease (AIMD), thus increasing the congestion window by 1 MSS every time an ACK is received.

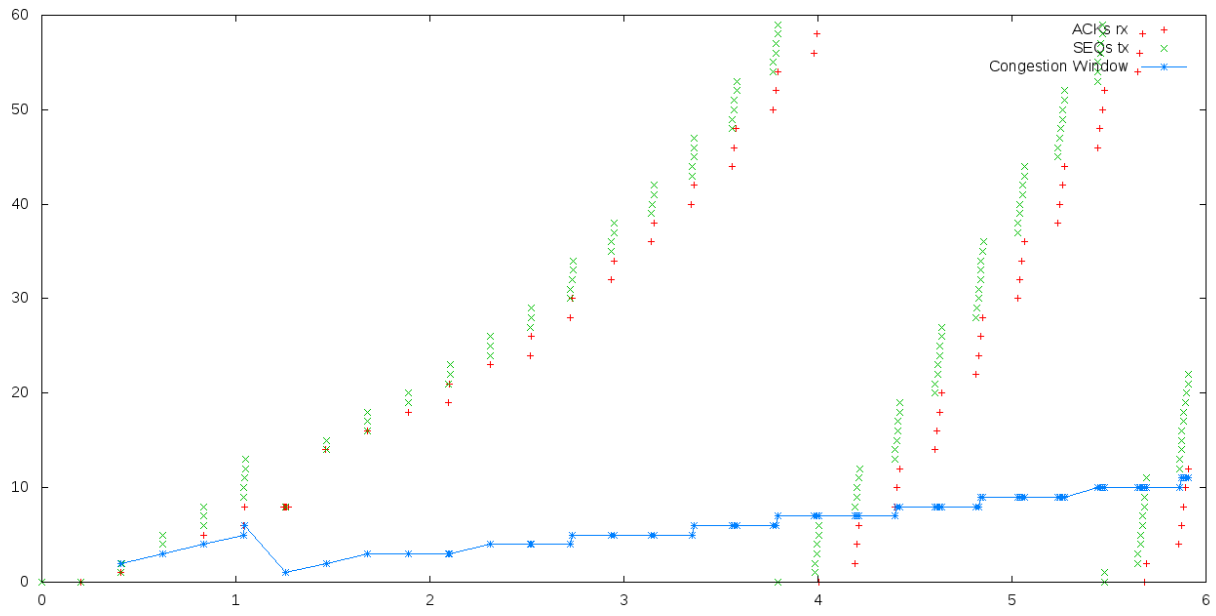

Figure 3: TCP Tahoe with one packet loss

In the case of TCP Reno, we observe that entering slow start is avoided and it performs Fast Recovery instead, the sender's congestion window is decreased to, 5, half its previous value and ssthresh is also set to this value. After the triple duplicate ACKs arrive, the sender uses Fast Retransmit and the lost packet is retransmitted and the sender continues to be in the Fast Recovery phase increasing the congestion window by 1MSS/RTT until it exits the Fast Recovery phase.

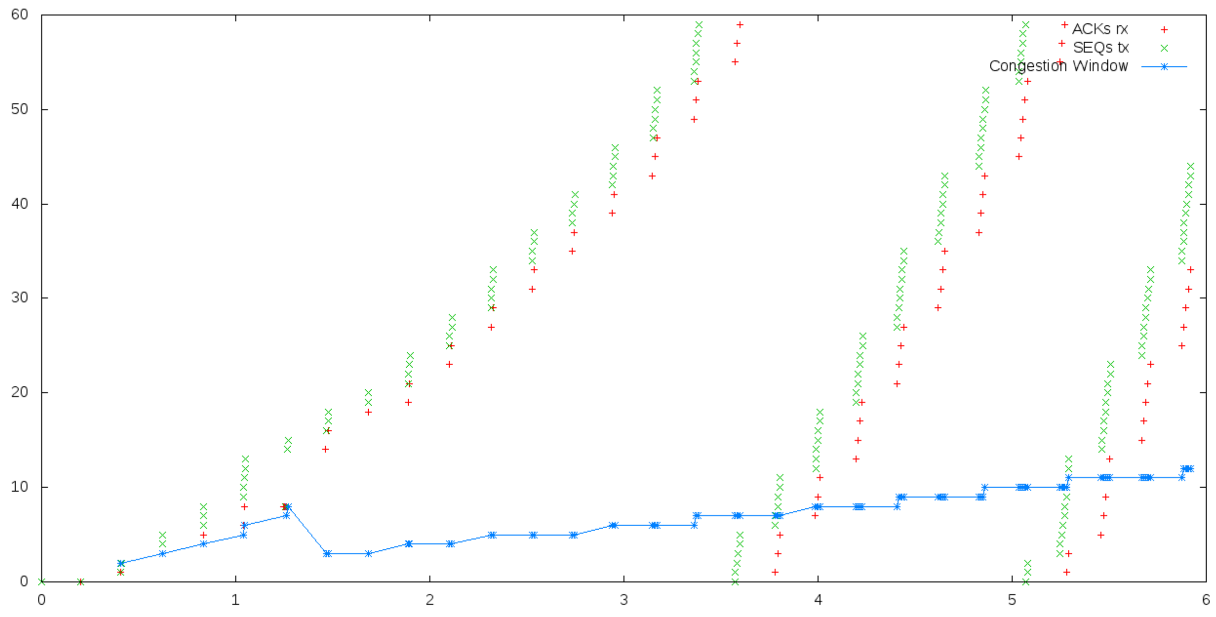

Figure 4: TCP Reno with one packet loss

TCP New Reno behaves exactly similar to TCP Reno, as there is no difference between the two, in the case of single losses.

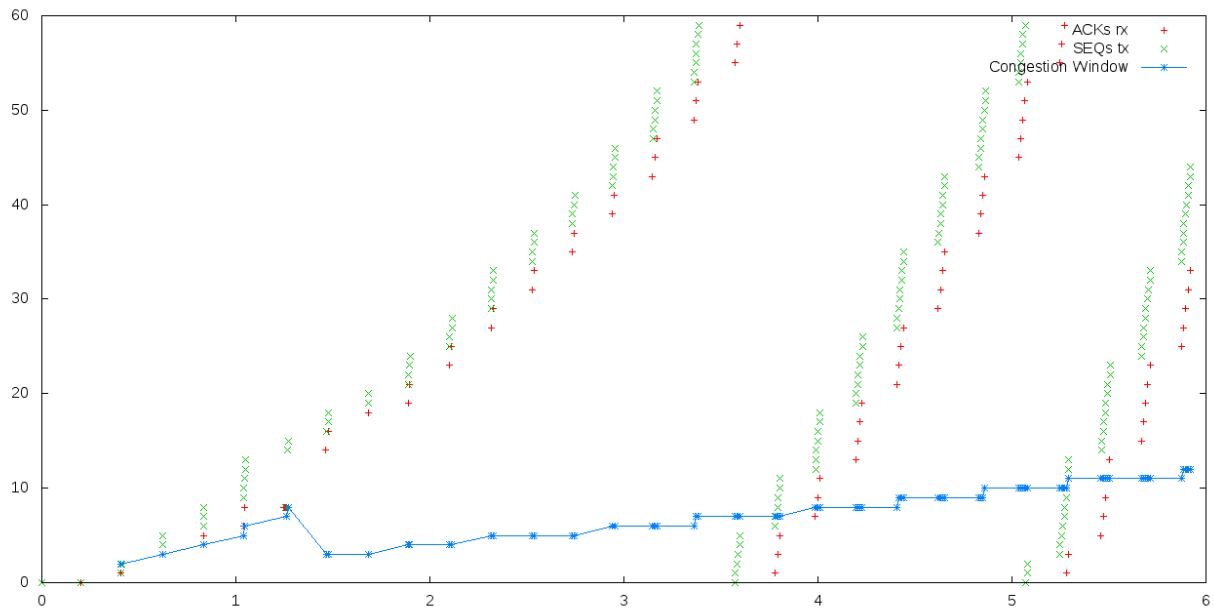

Figure 5: TCP New Reno with one packet loss

Two Packet Losses

In the case of TCP Tahoe, it behaves similar to the single loss case. After dropping packet 14 and receiving triple duplicate ACKs for it, the sender enters slow start with ssthresh set to 5, and the congestion window set to 1MSS. Then the sender receives an ACK for packet 14 that it sent using a Fast Retransmit, and this opens the congestion window to a value of 2, and then it just continues to transmit packet 15 forgetting the fact multiple losses have occurred. This happens because it entered the slow start phase again, and therefore it resends the lost packets anyway and the sender now continues in Slow Start.

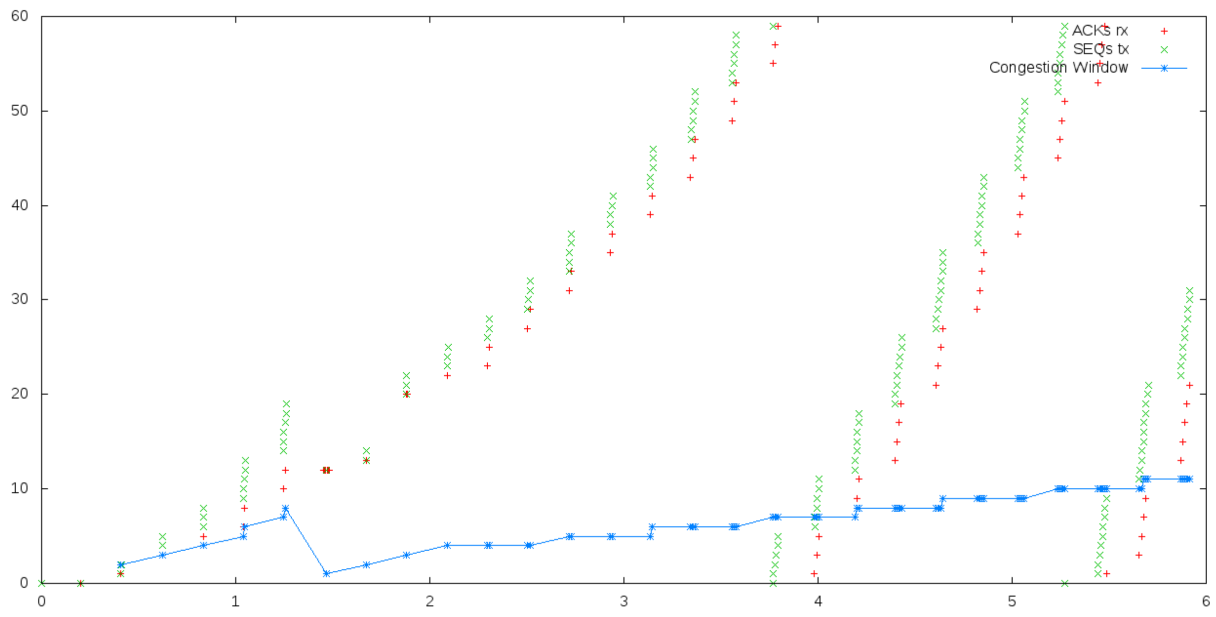

Figure 6: TCP Tahoe with two packet losses

In TCP Reno, the Fast Retransmit and Fast Recovery phases occur two times consecutively in two successive round trip times, and this slows down the connection considerably. The value of cwnd and ssthressh drop to a value of 3, and then Fast Recovery phase resumes by incrementing the cwnd by 1MSS/RTT till it eventually goes into the congestion avoidance phase and resumes Additive Increase, Multiplicative Decrease (AIMD).

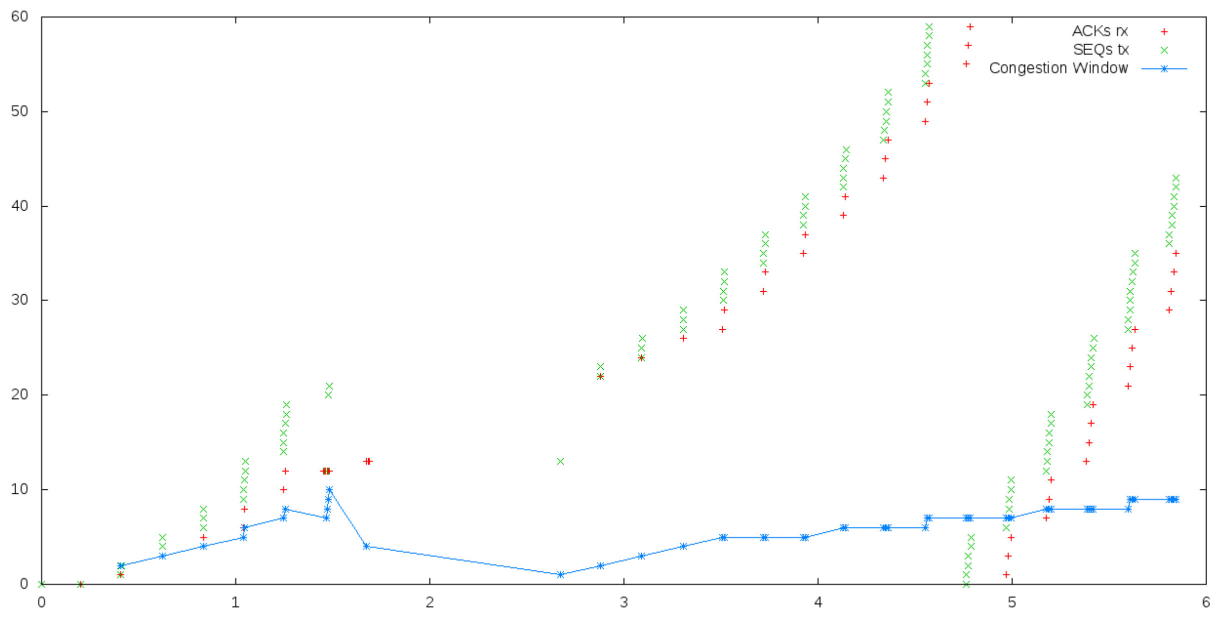

Figure 7: TCP Reno with two packet losses

In TCP NewReno however, its entry into two consecutive Fast Retransmit/Fast Recovery phases is avoided because it is designed to remain in the Fast Recovery phase in the case of multiple packet drops. This ensures that the congestion window is not affected and it receives partial ACKs until both the lost packets are retransmitted. The congestion window and ssthresh values are halved just like in the one packet loss case and the sender keeps retransmitting packets in the Fast Recovery phase. After this, the sender receives a cumulative ACK, and resumes operation in congestion avoidance.

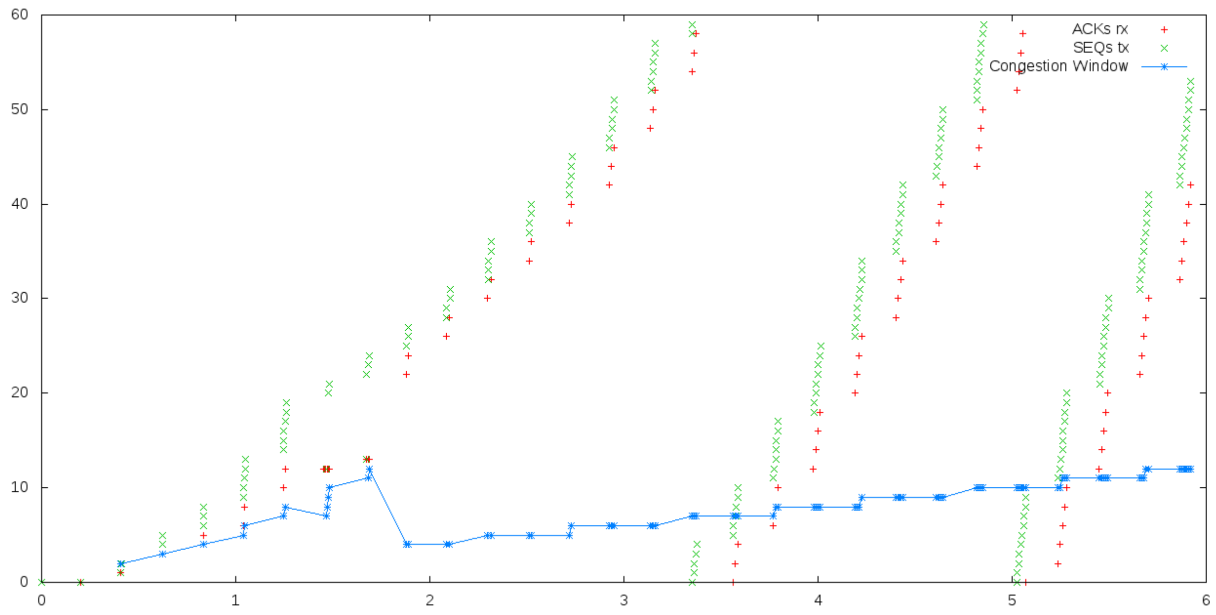

Figure 8: TCP New Reno with two packet losses

Three Packet Losses

In TCP Tahoe, the response is the same as that observed in the two packet loss case. The sender enters slow start which is triggered by the triple duplicate ACK received for packet 14, which it retransmitted using Fast Retransmit. Then it just continues to transmit packets 15 and 16 forgetting the fact multiple losses happened. This happens because it entered the slow start phase again, and therefore it resends the lost packets anyway and the sender now continues in Slow Start.

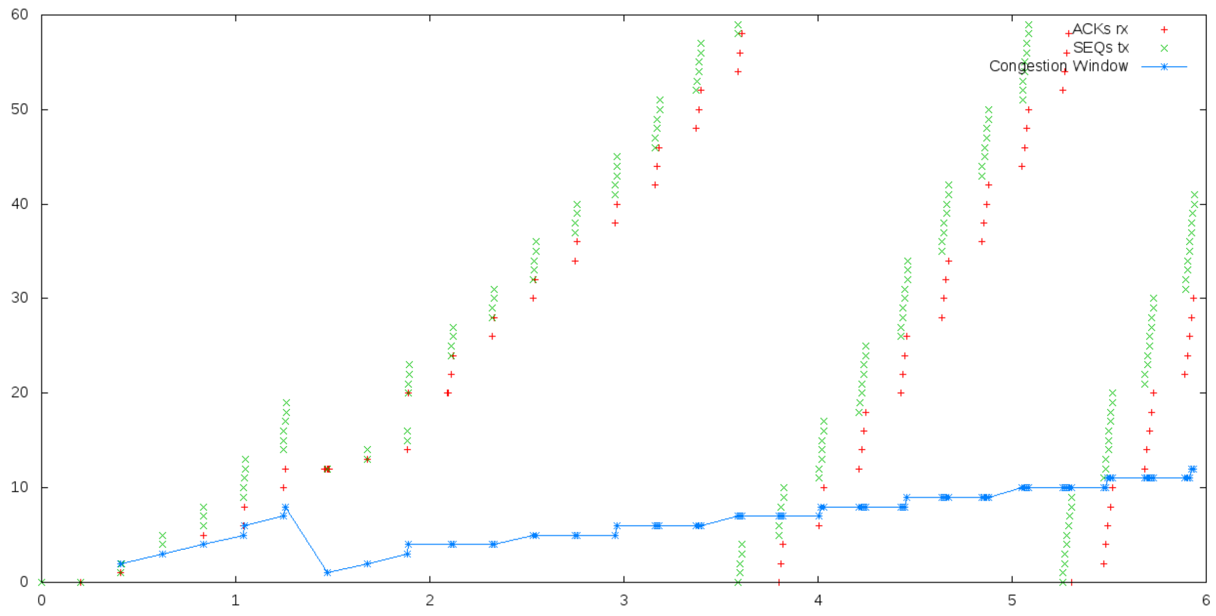

Figure 9: TCP Tahoe with three packet losses

In TCP Reno, this case is similar to the one observed in the two packet drop case, except for the fact that another additional packet drop (packet 17) occurs while the sender is still in the Fast Recovery phase. This causes the sender to time out and activate the retransmission timer (RTO). After waiting for the timer to go off, TCP Reno resumes the connection by entering the slow start phase again.

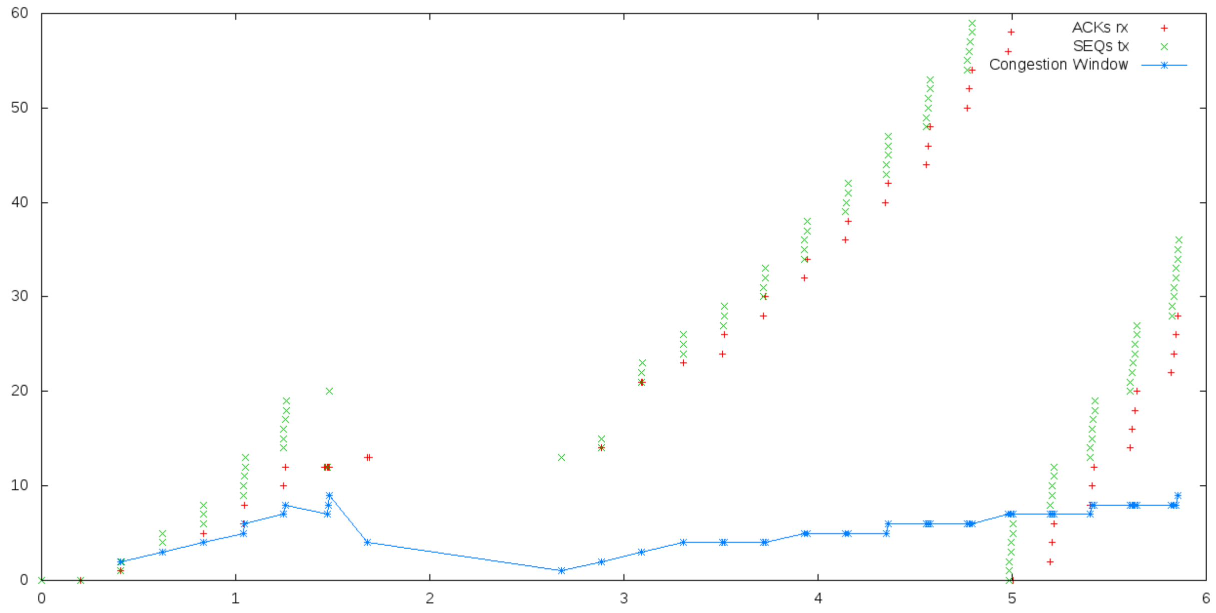

Figure 10: TCP Reno with three packet losses

In TCP NewReno, the retransmission timer is not triggered and it tackles the problem of multiple packet drops by remaining in the Fast Recovery phase instead of timing out. The sender receives partial ACKs for each retransmitted packet and continues increasing the congestion window by

1MSS/RTT till it receives an ACK for the last unACKed packet and resumes operation in the Congestion Avoidance phase.

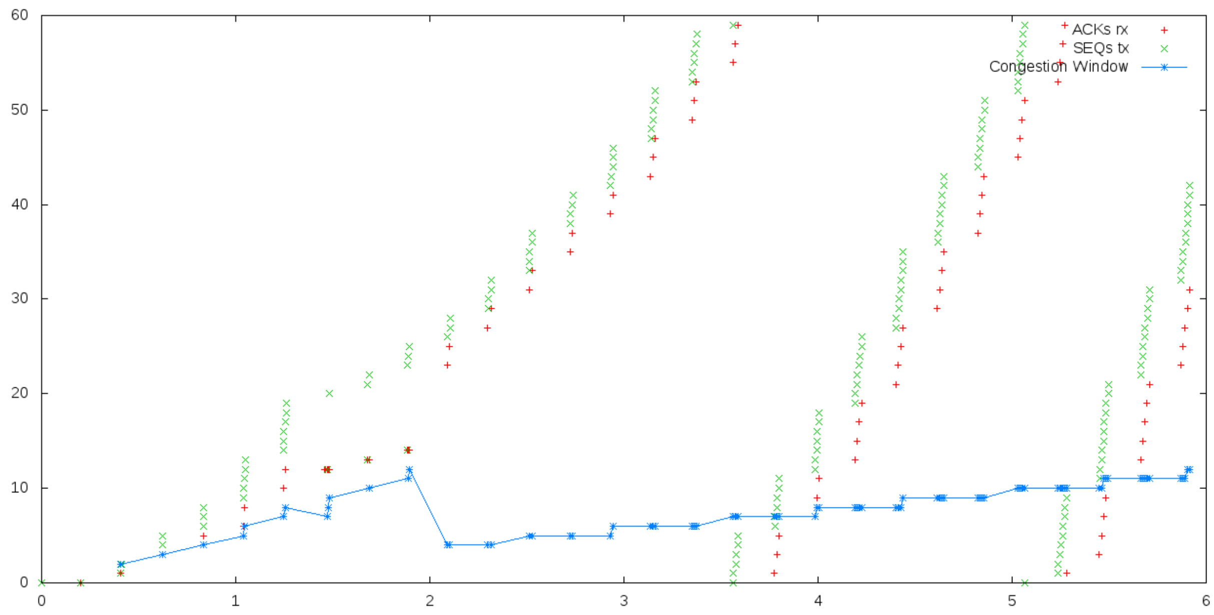

Figure 11: TCP New Reno with three packet losses

FOUR PACKET LOSSES

In TCP Tahoe, again the response is similar to that observed in the single drop case. The sender receives a triple duplicate ACK for packet 14 as mentioned and then it enters the slow start phase, after using Fast Retransmit to retransmit the packet. Then it just continues to transmit packets 15, 16 and 17 forgetting the fact multiple losses happened. This happens because it entered the slow start phase again, and therefore it resends the lost packets anyway and the sender now continues in Slow Start.

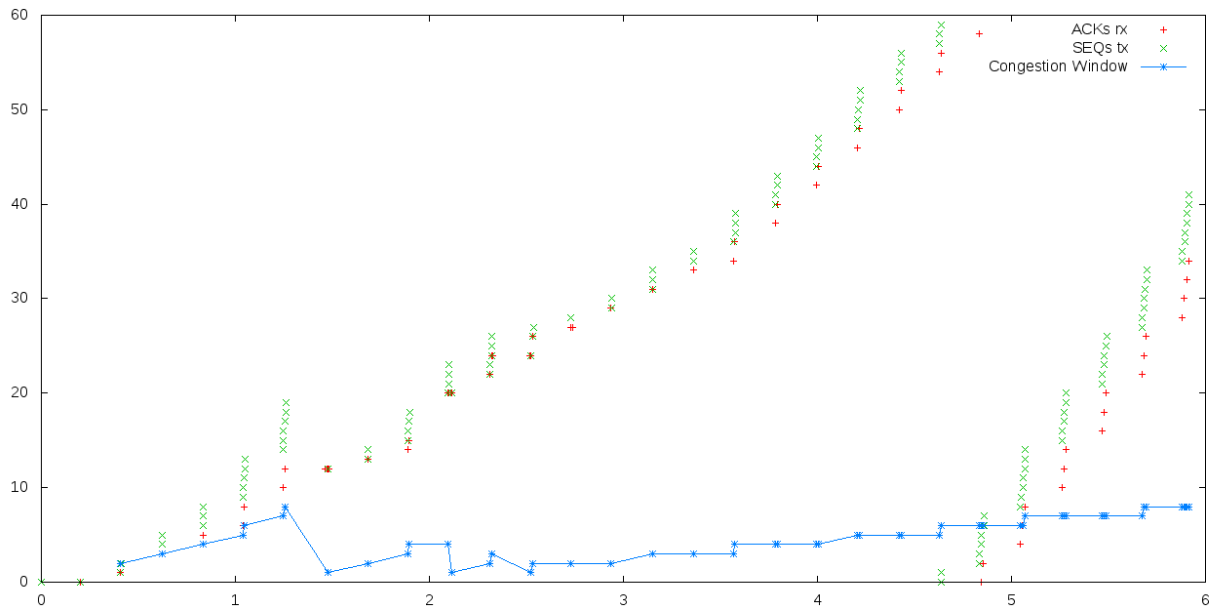

Figure 12: TCP Tahoe with four packet losses

For TCP Reno, the operation is similar to the three drops case where the sender triggers the retransmission timer (RTO) and waits before it resumes the slow start phase again. This happens due to the fact that multiple packet drops (15, 16, 17) happened while it was in the Fast Recovery phase.

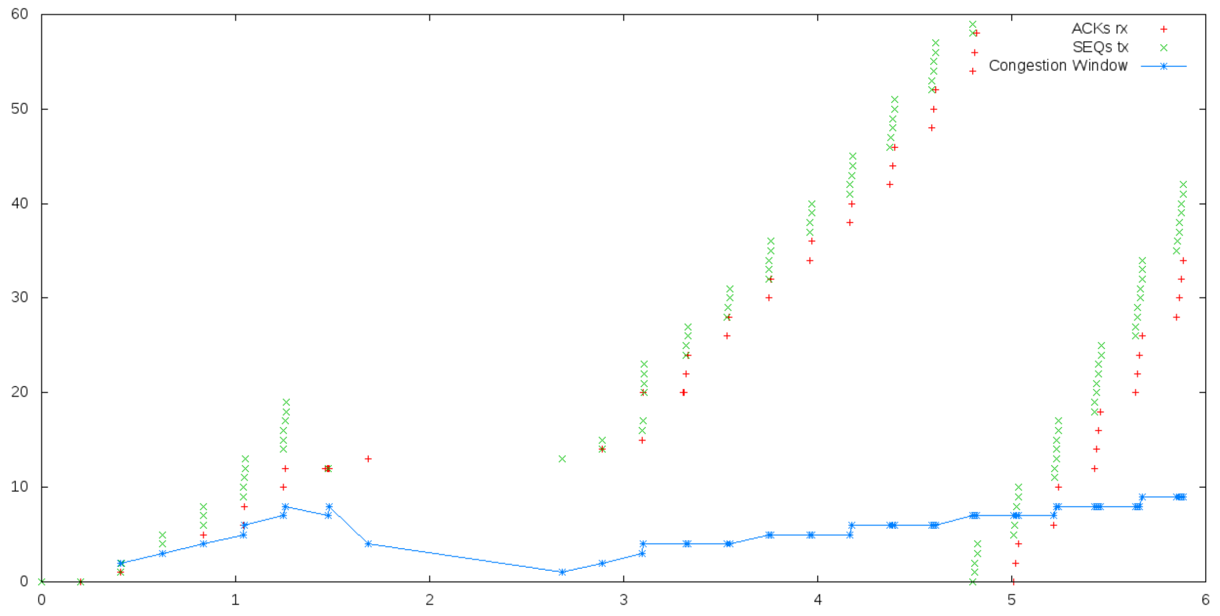

Figure 13: TCP Reno with four packet losses

In the case of TCP NewReno, again the multiple (burst) losses are handled well compared to Reno, because it does not trigger the RTO, and instead remains in the Fast Recovery phase. The sender receives partial ACKs for each retransmitted packet and continues increasing the cwnd by 1MSS/RTT till it receives an ACK for the last unACKed packet and resumes operation in the Congestion Avoidance phase.

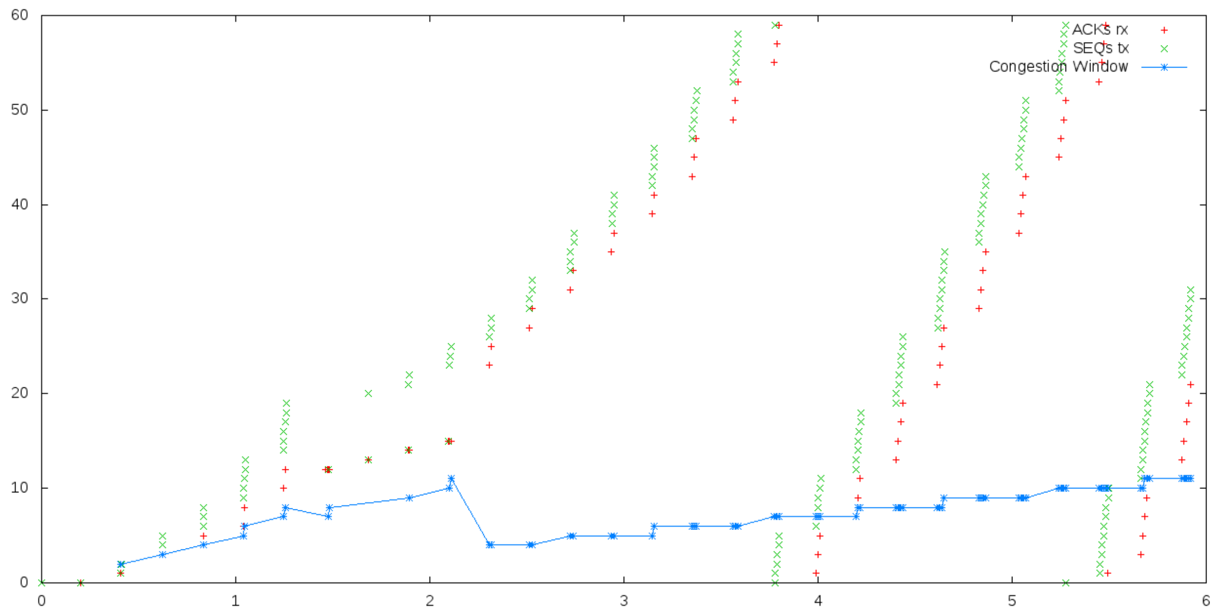

Figure 15: TCP New Reno with four packet losses

APPENDIX A

We used PERL scripts in order to process the trace data and plot the graphs easily. After the traces are recorded, we run the scripts in order to check for certain regexes, in order to pull out the sequence numbers, ACK numbers and the time at which they were recorded.

Script to parse and scale the congestion window:

```
#!/usr/bin/perl
use v5;
use strict;
use warnings;
open(FH,"<","cwnd.dat") or die "error happened: can't open file: $!";
open(WH,">","cwndextract.dat") or die "error happened: can't open file: $!";
while(<FH>){
        my @cwnd = split;
        $cwnd[1] = $cwnd[1] / 512; #packet size
        $cwnd[1] = $cwnd[1] % 60;
        print WH $cwnd[0],"\t\t", $cwnd[1],"\n";
}
close FH;
close WH;
```

Script to parse the trace file for ACK numbers and SEQ numbers at the entry of R1:
ACK:

```
#!/usr/bin/perl
use v5;
use strict;
use warnings;

my $matchstring;
my $matchr;
my $acktime;
my $ackcount;

open(FH,"<","node0long.tr") or die "error happened: can't open file: $!";
open(PH,">","ACKr.dat") or die "error happened: can't open file: $!";

while(<FH>){
        $matchstring = $_;
        if($matchstring =~ /^\+/){
                $matchr = $';
                $matchr =~ /\d+\.\d+|\d+/;
                $acktime = $&;
                print PH $acktime,"\t\t";
                if($matchr =~ /ns3::TcpHeader\s.*/){
                        $ackcount = $&;
                        $ackcount =~ s/.*Ack=(\d+)\sWin=.*/$1/;
                        $ackcount = $ackcount / 512; #packet size
                        $ackcount = $ackcount % 60;
                        print PH $ackcount,"\n";
                }
        }
}
close FH;
close PH;
```

SEQ:

```
#!/usr/bin/perl
use v5;
use strict;
use warnings;

my $matchstring;
my $matchr;
my $acktime;
my $ackcount;

open(FH,"<","node0long.tr") or die "error happened: can't open file: $!";
open(WH,">","SEQr.dat") or die "error happened: can't open file: $!";

while(<FH>){
        $matchstring = $_;
        if($matchstring =~ /^r/){
                $matchr = $';
                $matchr =~ /\d+\.\d+|\d+/;
                $acktime = $&;
                print WH $acktime,"\t\t";
                if($matchr =~ /ns3::TcpHeader\s.*/){
                        $ackcount = $&;
                        $ackcount =~ s/.*Seq=(\d+)\sAck=.*/$1/;
                        $ackcount = $ackcount / 512; #packet size
                        $ackcount = $ackcount % 60;
                        print WH $ackcount,"\n";
                }
        }
}
close FH;
close WH;
```

Script to parse the trace file for ACK numbers and SEQ numbers at the exit of R1:

ACK:

```
#!/usr/bin/perl
use v5;
use strict;
use warnings;

my $matchstring;
my $matchr;
my $acktime;
my $ackcount;

open(FH,"<","node1long.tr") or die "error happened: can't open file: $!";
open(PH,">","ACK+.dat") or die "error happened: can't open file: $!";

while(<FH>){
        $matchstring = $_;
        if($matchstring =~ /^r/){
                $matchr = $';
                $matchr =~ /\d+\.\d+|\d+/;
                $acktime = $&;
                print PH $acktime,"\t\t";
                if($matchr =~ /ns3::TcpHeader\s.*/){
                        $ackcount = $&;
                        $ackcount =~ s/.*Ack=(\d+)\sWin=.*/$1/;
                        $ackcount = $ackcount / 512; #packet size
                        $ackcount = $ackcount % 60;
                        print PH $ackcount,"\n";
                }
        }
}
close FH;
close PH;
```

SEQ:

```
#!/usr/bin/perl
use v5;
use strict;
use warnings;

my $matchstring;
my $matchr;
my $acktime;
my $ackcount;

open(FH,"<","node1long.tr") or die "error happened: can't open file: $!";
open(WH,">","SEQ+.dat") or die "error happened: can't open file: $!";
```

```
while(<FH>){
        $matchstring = $_;
        if($matchstring =~ /^\+/){
                $matchr = $';
                $matchr =~ /\d+\.\d+|\d+/;
                $acktime = $&;
                print WH $acktime,"\t\t";
                if($matchr =~ /ns3::TcpHeader\s.*/){
                        $ackcount = $&;
                        $ackcount =~ s/.*Seq=(\d+)\sAck=.*/$1/;
                        $ackcount = $ackcount / 512; #packet size
                        $ackcount = $ackcount % 60;
                        print WH $ackcount,"\n";
                }
        }
}
close FH;
close WH;
```

Finally, script to plot using gnuplot:

```
#!/usr/bin/perl
use v5;
use strict;
use warnings;
open my $GP, '|-', 'gnuplot' or die "$! : Can't open .dat file";

print {$GP} <<'__GNUPLOT__';
   set terminal png size 1280,640
   set output "<filename>.png"
   plot "ACKr.dat" using 1:2 title 'ACKs rx' with points, \
    "ACK+.dat" using 1:2 title 'ACKs q' with points, \
    "SEQ+.dat" using 1:2 title 'SEQs q' with points, \
    "SEQr.dat" using 1:2 title 'SEQs tx' with points, \
    "cwndextract.dat" using 1:2 title 'Congestion Window' with linespoints
   exit
__GNUPLOT__

close $GP;
```

APPENDIX B

Case 1: Negligible Queueing Delay
This figure shows the one packet loss case for TCP Tahoe and what happened when we tried using two flows in the BulkSendApplication for TCP on node (S1). The results show that the effects of queueing delay were negligible and we observed that the points plotted when the packets are

arriving at the queue overlap with the ones that are plotted after the packets leave the queue. We think that this happens because our queuesize is large and that the flows are not being queued and instead just traversing though the queue without any delay. This experiment was to study the nature of working of the different TCP flavors and hence we thought that neglecting queueing delay would not significantly affect the results of our experiment.

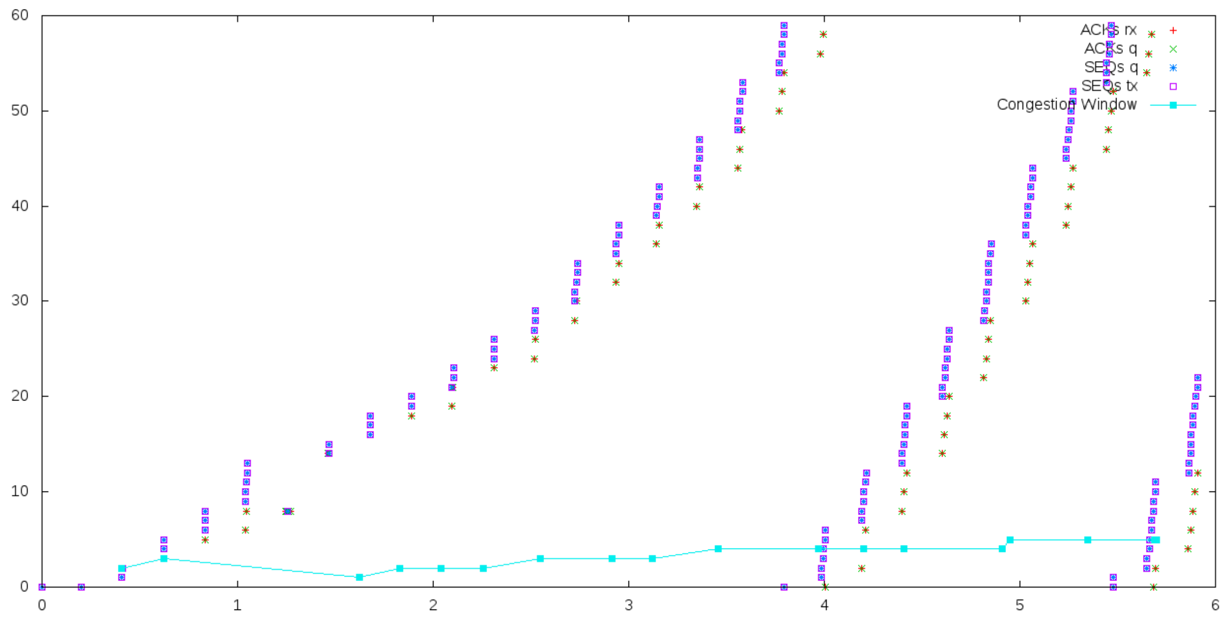

Figure 16: Negligible Queuing Delay

Case 2: Three Flows

We did not configure three flows in our experiment mainly due to the fact that the data rate of the additional flows used in the paper is not mentioned and we were getting skewed results when we used the same data rate for all three flows, as shown below.

This figure shows the one packet loss case for TCP Tahoe and what happened when we had 3 flows instead of 1 with the same data rate configured on node (S1) using the BulkSendApplication for TCP Tahoe. The results show that these additional flows constrain the throughput and we do observe queueing delay in this case. However, the results do not match the ones mentioned in the paper and hence we decided not to configure three flows and used only one flow instead.

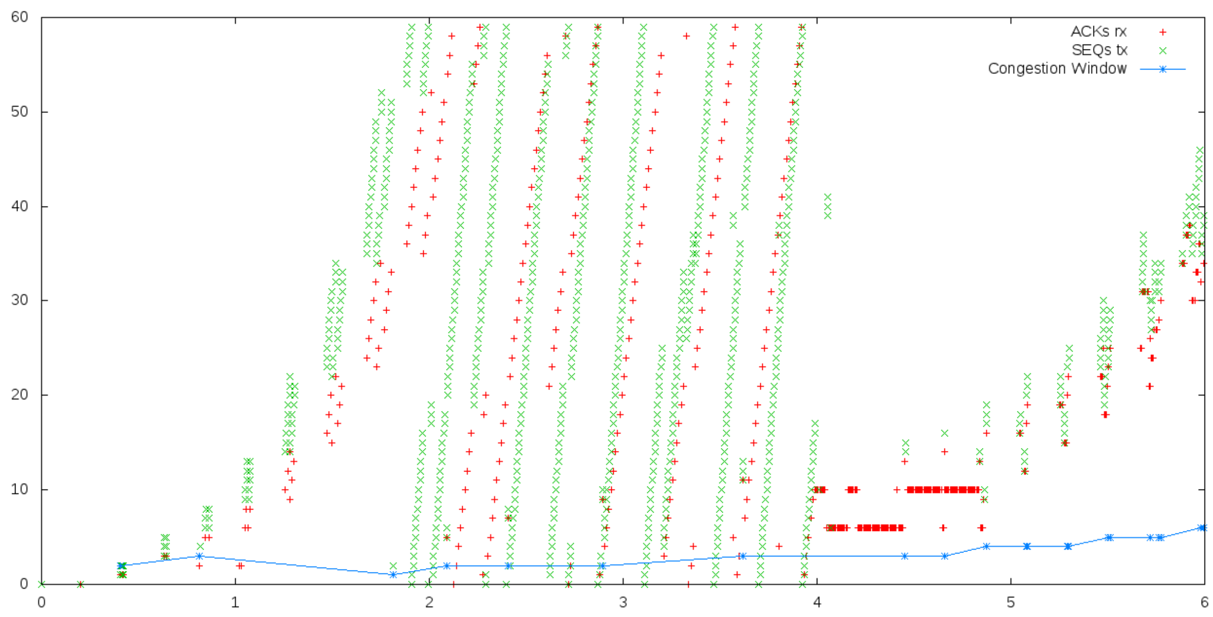

Figure 17: Output with 3 flows

REFERENCES

[1] Floyd, Sally, and Van Jacobson. "Random early detection gateways for congestion avoidance." *Networking, IEEE/ACM Transactions on* 1.4 (1993): 397-413.

[2] "New TCP Socket Architecture." Nsnam, 3 Sept. 2010. <https://www.nsnam.org/wiki/New_TCP_Socket_Architecture#Result_from_tcp-loss-response>

[3] "Perl Plotting." Misc-perl-info, 2012. <http://www.misc-perl-info.com/perl-plotting.html>

[4] "Ns-3 Tutorial: 5.3 Using the Tracing System." Ns-3 Tutorial. 20 Aug. 2010. <https://www.nsnam.org/docs/release/3.9/tutorial/tutorial_23.html >

[5] "Ns-3 Tutorial: 7.4 Using Trace Helpers." Ns-3 Tutorial. 20 Aug. 2010. <https://www.nsnam.org/docs/release/3.9/tutorial/tutorial_32.html >