A NEW FAIR WINDOW ALGORITHM
FOR ECN CAPABLE TCP (NEW-ECN)

by

Tilo Hamann

Memorandum No. UCB/ERL M99/35

29 June 1999
A NEW FAIR WINDOW ALGORITHM
FOR ECN CAPABLE TCP (NEW-ECN)

by

Tilo Hamann

Memorandum No. UCB/ERL M99/35

29 June 1999

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720
A New Fair Window Algorithm for ECN Capable TCP
(New-ECN)

Tilo Hamann
Department of Digital Communication Systems
Technical University of Hamburg-Harburg
Germany
t.hamann@tu-harburg.de

Advisor: Prof. Jean Walrand
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
USA
wlr@eecs.berkeley.edu

June 1999
Abstract

Although the TCP protocol is widely used in computer networks as a reliable data transfer protocol, the current implementations of TCP still have some drawbacks. One of the improvements to TCP's performance that were proposed is the Explicit Congestion Notification (ECN) [1] mechanism. ECN in conjunction with Random Early Detection (RED) [4] gateways avoids unnecessary packet drops and separates congestion detection and congestion notification. Another major issue in TCP has been shown in [5]: TCP is biased against flows with long round trip times (RTT). In this report we develop a new window control mechanism for TCP using the congestion information provided by ECN. This algorithm corrects the bias against TCP connections with longer RTTs and achieves a fair sharing of the available bandwidth of a bottleneck gateway. The idea is to prevent a fast connection from opening its congestion window too quickly and to enable a slow connection to open its window more aggressively. We demonstrate the effect of the New-ECN algorithm with a simple two flow model and verify the results with the network simulator “ns” [6] for different network topologies. We show that the New-ECN algorithm works also with many flows through one bottleneck gateway and study the TCP and ECN-TCP friendliness.

In this research report we only focus on the single bottleneck case. The behavior of New-ECN TCP-Reno with multiple congested gateways needs further research.
CONTENTS

Contents

1 Introduction and Problem Definition 6

2 Background 9

  2.1 Transport Control Protocol (TCP) .................................................. 9
     2.1.1 TCP-Tahoe ........................................................................ 10
     2.1.2 TCP-Reno ........................................................................ 11
     2.1.3 TCP-Vegas ........................................................................ 11
  2.2 Random Early Detection (RED) Gateways ........................................ 12
     2.2.1 The RED Algorithm .............................................................. 13
  2.3 Explicit Congestion Notification (ECN) .......................................... 15
     2.3.1 ECN-TCP Header ................................................................ 16
     2.3.2 ECN-TCP Gateways .............................................................. 16
     2.3.3 ECN-TCP Sender ................................................................ 17
     2.3.4 ECN-TCP Receiver .............................................................. 17
  2.4 Fairness ....................................................................................... 18

3 Developing a Fair Window Algorithm Using ECN 19

  3.1 New-ECN TCP-Reno Algorithm .................................................... 19
  3.2 Simulation Results for the Simple Model ....................................... 21

4 Verifying Results in the Network Simulator (NS) 25
CONTENTS

B.4 TCP-SINK.CC ......................................................... 62
B.5 NS-DEFAULT.TCL ................................................... 63

C TCL Scripts for the Simulations in Section 4 64
C.1 4 New-ECN TCP-Reno Connections ............................... 65
C.2 4 New-ECN TCP-Reno vs. 4 ECN TCP-Reno Connections ........ 72
C.3 Many New-ECN TCP-Reno Connections ......................... 74
C.4 One New-ECN TCP-Reno vs. one ECN/non-ECN TCP-Reno Connection 78
C.5 TCP Friendliness of New-ECN .................................... 81
C.6 ECN-TCP Friendliness of New-ECN ................................. 85
## LIST OF FIGURES

**List of Figures**

2.1 RED algorithm .................................................. 15
3.1 Simple model ...................................................... 19
3.2 Simple model: rates .............................................. 22
3.3 Simple model: windows .......................................... 23
3.4 Simple model: rate trace ....................................... 24
4.1 Simulation topology ............................................. 25
4.2 4 New-ECN TCP-Reno flows: ACK sequence number plot ... 28
4.3 4 New-ECN TCP-Reno flows: congestion window plot ........ 29
4.4 4 New-ECN TCP-Reno flows: $S_{wnd}$ plot .................... 29
4.5 4 New-ECN TCP-Reno flows: average window and $S_{wnd}$ .... 30
4.6 4 TCP-Reno flows: ACK sequence numbers plot for setup 2, $p_{max} = 0.1$ 32
4.7 4 TCP-Reno flows: ACK sequence numbers plot for setup 2, $p_{max} = 0.2$ 33
4.8 4 New-ECN vs. 4 ECN TCP-Reno flows: ACK sequence number plot . 34
4.9 16 New-ECN TCP-Reno connections: ACK sequence number plot ... 35
4.10 One New-ECN flow vs. one ECN/non-ECN flow ............... 38
4.11 TCP friendliness: ACK sequence number plot ................ 39
4.12 ECN-TCP friendliness: ACK sequence number plot .......... 40
List of Tables

4.1 4 New-ECN TCP-Reno flows: link parameter setup 1 ........................................ 26
4.2 4 New-ECN TCP-Reno flows: round trip delays ............................................. 26
4.3 4 New-ECN TCP-Reno flows: link parameter setup 2 ........................................ 27
4.4 4 New-ECN TCP-Reno flows: mean values ...................................................... 30
4.5 4 New-ECN TCP-Reno flows: mean value ratios ................................................ 31
4.6 4 New-ECN TCP-Reno flows: approximation for $\alpha$ ....................................... 31
4.7 Many New-ECN TCP-Reno flows: link parameter ............................................. 35
4.8 Many New-ECN TCP-Reno flows: round trip delays .......................................... 35
4.9 TCP / ECN-TCP friendliness: link parameter .................................................... 38
4.10 TCP / ECN-TCP friendliness: round trip delays ............................................. 39
1 Introduction and Problem Definition

The TCP protocol is widely used in computer networks as a reliable data transfer protocol. V. Jacobson [7, 8] introduced an adaptive retransmission and control mechanism, which has been subject to research and optimization since then. The main three versions of TCP are TCP-Tahoe, TCP-Reno/NewReno, and TCP-Vegas. Each of these versions has its advantages and disadvantages. The most commonly implemented version are TCP-Tahoe and TCP-Reno/NewReno.

There are still problems with these TCP congestion control schemes. The congestion control mechanisms slow start and congestion avoidance use packet loss or timeouts to discover congestion in the network. Usually, losses occur if the buffer at a network gateway has reached its capacity and every incoming packet is dropped (e.g., drop-tail gateways). In TCP, there are no means to discover congestion before a packet is lost. This leads to the problem of global synchronization [9]. When a gateway starts to drop packets due to a buffer overflow, many flows see a dropped packet at roughly the same time. Consequently, they all reduce their congestion window at the same time, which leads to a low utilization after a busy cycle and causes a greater variance in the queuing delay.

Besides these issues, TCP also has a bias against connection with longer round trip times [5]. A connection doubles its congestion window every round trip time (RTT) during the slow start phase and increases its congestion window by one every RTT during the congestion avoidance phase. This mechanism results in a non-uniform increase of the rates of connections with different RTT and enables a connection with a short RTT to discover and claim bandwidth faster than a connection with a longer RTT. Studies have shown that the bias in the claimed bandwidth for multiple connections is in the order of $RTT^\alpha$ with $\alpha < 2$ [10].

To improve the behavior of TCP during times of congestion, S. Floyd and V. Jacobson
introduced Random Early Detection (RED) Gateways [4]. The idea is to avoid multiple losses and global synchronization of TCP flows. A RED gateway controls its average queue size by dropping or marking arriving packets randomly before the buffer at the gateway overflows. Whether to mark or to drop a packet depends on the protocol used.

The RED algorithm allows us to identify misbehaving users [4] and penalize these connections, since they will see more dropped or marked packets. Nevertheless, a RED gateway cannot correct the bias of TCP against connections with long RTTs and assure that the bandwidth is shared fairly between all connections.

S. Floyd and K. Ramakrishnan proposed the Explicit Congestion Notification (ECN) mechanism [1, 2]. The idea of ECN is to avoid delays in a transmission due to unnecessary packet drops. The sending host receives a congestion feedback from the network by “marked” packets. A packet is marked at a RED gateway by setting a bit in the TCP header instead of being dropping by the RED algorithm. The mark gets echoed by the TCP receiver by setting an ECN echo flag in the header of the ACK and the TCP sender reacts then to the mark as it would to a dropped packet. The advantage of the mechanism is that it does not depend on the retransmit timeouts and the coarse granularity of the TCP timer. With ECN support, TCP does not need to wait for timeouts to react to a congestion indication. For flows with a small RTT, the coarse granularity of the TCP timer\(^1\) can delay the detection of a lost packet, with the result that the source lies idle and the link is underutilized. The improvement of the performance was studied in [1, 11]. ECN in conjunction with RED gateways addresses the issue of unnecessary and multiple packet losses and the resulting delays in the transmissions times, but it cannot guarantee a fair sharing of the available bandwidth.

The main goal of this research project was to investigate if it is possible to use the congestion information provided by ECN to correct the bias against connections with long RTTs. Using the idea of ECN with RED gateways, we developed an algorithm that

\(^1\)many implementations use a clock setting of $\geq 100\text{ms}$
achieves a fair sharing of the available bandwidth by using the congestion feedback of ECN. We used the main idea of ECN with RED gateways and made some modifications to the TCP window algorithm.

This report is organized as follows. Section 2 explains the used protocols and gateway mechanisms. Section 3 describes a simplified model with one bottleneck gateway and two TCP-Reno connections. We introduce a modified window algorithm for TCP and show that it can achieve a fair sharing of the available bandwidth. Section 4 verifies the results of the simple model with the network simulator "ns" [6] for more complex network topologies. Section 5 summarizes the results found in this report and discusses future work.
2 Background

2.1 Transport Control Protocol (TCP)

This section gives a more general overview over TCP. A detailed description on TCP/IP and its implementations can be found in [7, 8, 12, 13, 14, 15, 16]. TCP is a connection oriented, reliable byte stream protocol. It controls transmission errors and the flow of data packets. TCP is widely used to reliably transfer data in computer networks. In 1988 V. Jacobson [7] developed the congestion avoidance and control mechanisms which are used in TCP. The control mechanisms TCP uses are based on windows. It never injects more than a window size of packets into the network. That means that TCP never has more than a window size of unacknowledged packets in the network. Once it has sent the window size, the TCP sender only sends out new data when it receives an acknowledgment (ACK) for previously unacknowledged data from the receiver. In this way, TCP avoids overloading the network. This window size is called congestion window. The size of the congestion window is controlled by the two main algorithms of TCP: (1) slow start, (2) congestion avoidance. Later, a few more enhancements were made to TCP window algorithm: fast recovery and fast retransmit.

After a TCP connection is established the sender starts sending data packets with the slow start algorithm. During slow start, every incoming ACK increases the congestion window $cwnd$ by 1 packet, which leads to an exponential increase of the window size over time. During the congestion avoidance phase, TCP increases the congestion window linearly by roughly one packet each RTT. The congestion window is bounded above by the receiver-advertised window. With these two algorithms, TCP is constantly probing for bandwidth.

TCP infers from a packet loss the presence of congestion in the network. In case of a packet loss, TCP cuts the congestion window by $\frac{1}{2}$. The window algorithm used
after a packet loss depends on the version of TCP. In [17] D. Chiu and R. Jain showed that a linear increase and multiplicative decrease algorithm is a fair and efficient algorithm. The combination of the TCP window and congestion control algorithms is described below for the TCP versions TCP-Tahoe and TCP-Reno. TCP-Vegas uses a different approach for the window control, which is also described below. It has been shown that TCP-Reno and TCP-Tahoe are biased against connections with longer RTT [5], whereas TCP-Vegas enables the connections to achieve a fair sharing of the bandwidth [18].

2.1.1 TCP-Tahoe

TCP-Tahoe uses in its original version only the slow start and congestion avoidance algorithm. At the beginning of a connection, the congestion window is initialized with one packet and it is then increased by the slow start algorithm. Thus, TCP is probing the network for bandwidth in an exponential manner. The slow start phase ends if either a packet gets lost, which means that the sending rate exceeds the network capacity, or the window size is greater than the slow start threshold $ssthresh$. In the latter case, TCP enters the congestion avoidance phase. For the first case, a packet loss was discovered by a timeout, the slow start threshold is set to $ssthresh = \frac{1}{2} \text{cwnd}$ and the congestion window set to 1 packet again. As long the window size is below the slow start threshold, the slow start algorithm is used. Otherwise, the window size is increased by the congestion avoidance algorithm.

In recent TCP-Tahoe versions, packet loss is also discovered by *duplicate ACKs*. This means the sender received multiple (usually three) ACKs with the same (expected) sequence number. Receiving three or more duplicate ACKs in a row is a strong indication that a packet got lost. The missing packet is retransmitted immediately without wait-
ing for a timeout. This mechanism is called fast retransmit. After a fast retransmit TCP-Tahoe resets the congestion window size to 1 and performs slow start.

2.1.2 TCP-Reno

TCP-Reno uses the same mechanism as TCP-Tahoe. To improve the behavior of TCP in case of a packet loss, the fast retransmit algorithm in TCP-Reno is enhanced by the fast recovery algorithm. TCP-Reno does not perform a slow start after a fast retransmit. The congestion window is reduced by $\frac{1}{2}$ and the value of the new window is stored in the slow start threshold. The lost packet is retransmitted and further incoming duplicate ACKs are used to clock new subsequent outgoing packets. The congestion window is now $cwnd = cwnd + n_{dup,ACKs}$ where $n_{dup,ACKs}$ is the number of duplicate ACKs received so far. Thus, new data packets can be sent out. The fast recovery algorithm enables the connection to reduce backlog in the “pipe” (communication path) by half instead of flushing it completely. This provides a better throughput recovery after a (single) packet loss. After the first ACK that acknowledges new data arrives, the congestion window is reset to the slow start threshold\(^3\) and congestion avoidance is entered. The fast recovery algorithm is working efficiently only for a single packet loss in a window, but it does not recover very well from multiple packet losses. To cover this case, some enhancements have been proposed, called New-Reno. The idea is that the fast recovery phase is not left before all data that was outstanding at the beginning of the fast recovery phase is acknowledged. New-Reno is described in detail in [19, 16].

2.1.3 TCP-Vegas

TCP-Vegas uses a different approach to estimate the available bandwidth in the network. It uses the difference between the actual flow rate and the estimated flow rate\(^3\) which has the half value of the congestion window before a duplicate ACK was discovered.
to adjust the window size. The ACK scheme on the receiver side is not changed, TCP-Vegas just requires changes to the sender side. The idea behind the Vegas algorithm is that as long the network is not congested, the estimated flow rate is approximately the same as the actual flow rate. As soon as the network gets congested the estimate and the actual flow rate will differ. This difference is used to update the congestion window size in the following way:

\[
cwnd = \begin{cases} 
  cwnd + 1 & \text{if } \left[ (rate_{expected} - rate_{actual}) \cdot RTT_{minimum} \right] < \alpha \\
  cwnd - 1 & \text{if } \left[ (rate_{expected} - rate_{actual}) \cdot RTT_{minimum} \right] > \beta \\
  cwnd & \text{else}
\end{cases} 
\]  

with \( rate_{expected} = \frac{cwnd}{RTT_{minimum}} \) and \( rate_{actual} = \frac{cwnd}{RTT_{actual}} \). \( RTT_{minimum} \) is the minimum RTT, and \( RTT_{actual} \) is the actual (measured) RTT. Using equation 2.1 TCP-Vegas is trying to keep at least \( \alpha \) packets and at most \( \beta \) packets in transit. The advantage of TCP-Vegas is that, in contrast to TCP-Reno, it is not constantly probing for bandwidth. Rather, it discovers new bandwidth as soon it is available by the differences in the rates. Once it reaches the equilibrium, the window size does not change. TCP-Vegas is not a biased algorithm and can also achieve a higher throughput than TCP-Reno. Nevertheless, a major disadvantage of TCP-Vegas is that it gets about 50% less bandwidth than TCP-Reno when it is competing with TCP-Reno for the same resources [18]. Since the window adjustment algorithm is based on the round trip times, TCP-Vegas is sensitive to sudden changes of the RTT (e.g. rerouting). These issues in TCP-Vegas were studied further in [20] and [18].

### 2.2 Random Early Detection (RED) Gateways

The Random Early Detection (RED) mechanism was proposed for packet switched networks by S. Floyd and V. Jacobson in [4]. It was primarily designed for TCP networks, but it also could be used in other packet or cell-based environments. In contrast to drop-tail gateways, a RED gateway tries to detect congestion before the
2 BACKGROUND

gateway experiences a queue overflow. This early detection enables the gateway to prevent multiple packet losses and global synchronization. A RED gateway can either drop or mark a packet as congestion indication for the sources. Whether to mark or to drop a packet depends on the protocol used by the sources. In case of TCP, the gateway will drop the packets, since TCP infers congestion from packet drops. The RED gateway drops packets randomly as a function of the average queue size. The drop algorithm is based on the average rather than on the instantaneous queue size to avoid a bias against bursty traffic. By randomly dropping packets, RED limits synchronization across multiple flows through the gateway. The RED algorithm makes it possible to separate the congestion detection from the congestion notification. Another intention was to penalize flows with an excessive share of bandwidth. The idea was that a flow with a high rate has a higher number of packets arriving at the gateway and therefore a higher probability to see dropped packets in times of congestion. On the other hand, a low bandwidth connection periodically sees packet drops, which prevents the flow from reaching its fair share. This behavior is further studied in [21] by D. Lin and R. Morris. They proposed a flow random early random drop mechanism to correct this problem, but this requires a per flow accounting. The RED algorithm works well in times of moderate congestion. In case of heavy congestion, the RED fails to provide benefit to the network. This issue and enhancements to the RED algorithm are studied in [22, 23, 24]. These papers propose adaptive RED algorithms that estimate the number of active flows based on different information.

2.2.1 The RED Algorithm

The RED algorithm tries to keep the average queue length between two thresholds. No packet is dropped while the average queue size is below the lower threshold. If the average queue size exceeds the lower threshold and is still below the upper threshold, incoming packets get dropped with the probability $p_{\text{drop}}$, where $p_{\text{drop}}$ is a function of
the average queue length. As soon as the average queue size is greater than the upper threshold, every incoming packet is dropped. The two parts of the RED algorithm are: (1) calculating the average queue length, and (2) calculating the packet dropping probability. The first part determines the degree of burstiness allowed by the gateway and the second part determines how often packets will be dropped. The goal is to drop packets at roughly evenly spaced intervals in order to avoid biases and global synchronization, and also to drop packets with sufficient frequency to control the average queue size.

The average queue length $q_{avg}$ is an exponential weighted function of the instantaneous queue size $q$ with weight factor $w_q$

$$q_{avg} = (1 - w_q) q_{avg} + w_q q$$

which is calculated for every incoming packet. During idle times the RED algorithm tries to estimate the number of packets that could have been transmitted by the gateway in this idle period. For the first packet after an idle phase $q_{avg}$ is calculated as if the estimated number of packets had arrived to an empty queue during that period.

The computation of the dropping probability $p_{drop}$ is done in the following two steps:
First, the 'raw' dropping probability $p_{raw}$ is calculated which increases linearly from 0 to $p_{max}$ while $q_{avg}$ is increasing from $min_{th}$ to $max_{th}$

$$p_{raw} = p_{max} \frac{q_{avg} - min_{th}}{max_{th} - min_{th}}$$

where $min_{th}$ denotes the minimum threshold and $max_{th}$ the maximum threshold. The final dropping probability $p_{drop}$ is calculated by

$$p_{drop} = \frac{p_{raw}}{1 - count \cdot p_{raw}}$$

where $count$ is the number of packets arrived since the last drop. $p_{drop}$ is slowly increasing with the number of non-dropped packets.
2 BACKGROUND

The RED algorithm combines the previous two parts as in Figure 2.1. Further information on the implementation and a discussion on the parameters of the RED algorithm can be found in [4].

\begin{verbatim}
for each packet arrival {
    calculate the average queue size $q_{avg}$
    if $\min_{th} \leq q_{avg} < \max_{th}$ {
        calculate probability $p_{drop}$
        with probability $p_{drop}$
        drop the arriving packet
    }
    else if $\max_{th} \leq q_{avg}$
    drop the arriving packet
}\end{verbatim}

Figure 2.1: RED algorithm

2.3 Explicit Congestion Notification (ECN)

TCP infers that the network is congested only from a packet drop, which is indicated by retransmit timer timeouts or duplicate acknowledgements. This mechanism can be an 'expensive' way to detect the presence of congestion. Explicit Congestion Notification enhances active queue management, like RED, that drops packets probabilistically based on the queue state. In particular, when using ECN, packets are marked rather than dropped. This marking informs the source quickly about congestion, so that it does not need to wait for duplicate acknowledgements (ACKs) or for a timeout. Therefore, delay sensitive connections will benefit from ECN. Packet drops and especially multiple packet drops are reduced by the ECN mechanism.

S. Floyd and K. Ramakrishnan have proposed an extension to TCP/IP [2] to support ECN. In general, TCP should respond to a single marked packet as it would to a lost
packet. That means TCP cuts down the congestion window $cwnd$ by half and reduces
the slow start threshold $ssthresh$, with the exception that a marked packet should not
trigger a slow start in TCP-Tahoe and that TCP-Reno should not wait for roughly a
$\frac{1}{2}RTT$ in the fast recovery phase. The next four sections summarize the modifications
to TCP needed to implement ECN.

2.3.1 ECN-TCP Header

ECN-TCP makes use of two bits in the header: one to indicate that the connection is
ECN capable (ECT-bit) and one to indicate congestion (CE-bit). If a connection uses
ECN for congestion control the ECT-bit is set to 1 in all packets, otherwise it is set
to 0. The CE-bit is set at a congested router\(^4\). How a gateway decides when to mark
packets is described in Section 2.3.2. A discussion on whether to use two bits or just
one bit can be found in [2] along with a detailed description of the position of the bits.
Furthermore, ECN introduces two new flags in the reserved field of the TCP header.
The first flag is the ECN echo flag, which is set in an ACK by the receiver if an ECN
packet\(^5\) has the CE-bit set. The second flag is the congestion window reduced (CWR)
flag. This flag is set by the sender after it has reduced its congestion window for any
reason (i.e. retransmission timer timeouts, duplicate ACKs, and ECN echo ACKs).

2.3.2 ECN-TCP Gateways

ECN-TCP uses RED gateways with a modified algorithm to set the CE-bit in the TCP
header. The underlying RED algorithm is described in Section 2.2. The difference is
that the gateway marks packets instead of dropping them. A packet is always dropped
if the average queue length exceeds an upper threshold, even if it is an ECN packet [3].

\(^4\)the terms gateway and router are used interchangeably in this report
\(^5\)an ECN packet is a packet of a ECN capable connection (ECT-bit is set to 1)
\(^6\)an ECN echo ACK is an ACK of a ECN capable connection with the ECN echo flag set to 1
2.3.3 ECN-TCP Sender

In the connection setup phase, the sender and receiver need to negotiate their ECN capability. This is done by the sender setting the CWR and ECN echo flags in the SYN packet and by the receiver setting the ECN echo flag in the SYN-ACK packet. In the case that just the sender is ECN capable, the connection should not make use of the ECT and CE bit. Otherwise, the ECT bit should be set by the sender in every packet and must react to ECN echo ACKs.

The sender should treat an ECN echo ACK as a lost packet, but without the need to retransmit the marked packet. Although this ACK acknowledges a data packet, it does not increase the congestion window. If the sender receives multiple congestion indications, including timeout, duplicate ACKs and ECN echo ACK, it should react just once per RTT to the congestion indication. In the case that a retransmitted packet is marked or dropped, the sender should react to congestion indication again. However, it should not reduce the slow start threshold ssthresh if it has been reduced within the last RTT. After the sender responds to a congestion indication, it sets the CWR flag in the next data packet sent after the reduction of the window.

2.3.4 ECN-TCP Receiver

The receiver echoes the congestion indication of a CE packet\(^7\) back to the sender. After it has received a CE packet, it sets the ECN echo flag of the subsequent ACK packet. The receiver continues to set the echo flag until it receives a data packet with the CWR flag set. This provides robustness in case an ACK packet with the echo flag set gets lost.

\(^7\)a CE packet is a packet of a ECN capable connection with the CE bit set to 1
2.4 Fairness

One of the most common fairness definitions is the *max-min fairness* [25]. A rate vector $x$ is max-min fair if any rate $x_i$ of flow $i$ cannot be increased without decreasing the rate $x_j$ of some flow $j$ with $x_j \leq x_i$. To date, no reliable decentralized algorithm is known that achieves max-min fairness. Another definition of fairness, the *proportional fairness* was defined by F. Kelly in [26]: A vector of rates is proportionally fair if it is feasible (that is $x \geq 0$ and $Ax \leq C$), and if for any other feasible vector $x^*$, the aggregate of proportional changes is zero or negative:

$$\sum_i \frac{x_i^* - x_i}{x_i} \leq 0$$  \hspace{1cm} (2.5)

This report focuses only on a single bottleneck topology. For this case, the proportional fairness is equal to the max-min fairness. For a single bottleneck gateway, fairness means that all connections through this gateway share the available bandwidth equally. For the case that the flows do not share the bandwidth equally, D. Chiu and R. Jain developed a fairness criterion to quantify the fairness [27, 17]. For a bottleneck link with $n$ flows going through, the fairness index is

$$F_l := \frac{(\sum_{i=1}^n x_i)^2}{n \cdot \sum_{i=1}^n x_i^2}$$  \hspace{1cm} (2.6)

This index is 1 if all flows share the bandwidth equally and it is $\frac{1}{n}$ if one flow uses up all the bandwidth. The fairness index is used in this report to measure and compare the fairness of New-ECN TCP-Reno.

---

---

$^8C$ is the vector of link capacities, $A_{ij} = 1$ if flow $i$ uses resource $j$, $A_{ij} = 0$ otherwise
3 Developing a Fair Window Algorithm Using ECN

ECN-TCP inherits the bias against connections with longer RTTs from TCP. We developed a window control algorithm that corrects this bias and achieves a fair sharing of the available bandwidth by using the congestion information provided by the ECN algorithm. We studied the behavior of a simplified network model with two ECN TCP-Reno connections sharing one bottleneck router (see Figure 3.1). For this model we wrote a simulation in Matlab and modified the TCP and ECN algorithm to achieve an almost fair bandwidth sharing. The idea was to prevent the fast connection\(^9\) from opening its congestion window too quickly and to enable the slow connection to open its window more aggressively. We experimented with a few algorithms based on this idea and developed the New-ECN algorithm described in the next subsection.

\[\text{TCP-Source}\]

\[\text{X} \quad \text{Link 1} \quad \text{RED Gateway with ECN} \quad \text{Link 3} \quad \text{TCP-Sink for both sources}\]

\[\text{Y} \quad \text{Link 2} \quad \text{TCP-Source}\]

Figure 3.1: Simple model

3.1 New-ECN TCP-Reno Algorithm

The New-ECN TCP-Reno algorithm tries to slow down a fast connection and gives more bandwidth to a slow connection. The algorithm we developed modifies not only the window size as a response to a marked packet\(^{10}\) but also the slope of the congestion

\(^9\)a fast (slow) connection is a connection with a short (long) RTT

\(^{10}\)a marked packet means that the sender received a ECN echo ACK as a respond of the receivers to a CE packet
TCP increases the window size by \( \frac{1}{c_{wnd}} \) for every received ACK, which is roughly an increase by 1 every RTT. In New-ECN TCP-Reno this increase is modified based on the type of the received ACK, an ECN echo ACK or a normal ACK. New-ECN TCP-Reno uses TCP-Reno with ECN support as underlying TCP code with the following enhancements.

While the TCP sender receives regular ACKs it increases its congestion window using the formula

\[
c_{wnd} = c_{wnd} + \frac{1}{c_{wnd}} \cdot S_{wnd}
\]

Thus, the window increases by roughly \( S_{wnd} \) per RTT. In the connection setup phase this value is initialized with \( S_{wnd} = 1 \) and it is not used in the slow start phase of TCP. Hence, in the beginning of a connection (until the flow sees a marked packet) the increase is TCP like. After the first marked packet has been received, this value gets updated once every RTT as a function of the round trip time\(^{12}\):

\[
S_{wnd} = S_{wnd} + \beta \cdot RTT^2
\]

\( S_{wnd} \) is only used in the congestion avoidance algorithm and does not affect the slow start phase.

In case the sender receives a marked packet, it reduces the congestion window

\[
c_{wnd} = \alpha_{wnd} \cdot c_{wnd}
\]

along with the slow start threshold

\[
s_{thresh} = \alpha_{wnd} \cdot s_{thresh}
\]

and decreases the slope of the window

\[
S_{wnd} = \alpha_{slp} \cdot S_{wnd}
\]

\(^{11}\)called \textit{window slope}

\(^{12}\)In the implementation of the New-ECN algorithm in the network simulator 'ns', the smoothed round trip time \(sRTT\) is used.
Similarly to normal ECN TCP, New-ECN TCP reacts at most once per RTT to a congestion notification. In case of a loss, indicated by duplicate ACKs or a retransmit timer timeout, New-ECN TCP-Reno reduces the congestion window \( cwnd \) by \( \frac{1}{2} \) and also, if \( S_{\text{wnd}} > 1 \), \( S_{\text{wnd}} \) by \( \frac{1}{2} \). Furthermore, the TCP receiver should send out only one ECN echo ACK after a received CE packet\(^{13}\). The intuition behind this algorithm is that a fast connection will most likely see more marks in the beginning of a connection than a connection with a long RTT. Therefore, it reduces its window and its slope more frequently. Additionally, the slow connection will increase its value of \( S_{\text{wnd}} \) more significantly, since it will not be repeatedly ‘interrupted’ in the increase by marked packets. After some time both flows will see roughly the same number of marks and the congestion window will stabilize with some variance at a mean value. The following simulation will show that the mean value of the window size is roughly linear in RTT. The description in this section and implementation in the next sections demonstrates the performance characteristics of New-ECN with the objective to achieve fairness across multiple flows sharing the same resources. Nevertheless, since the development of the algorithm assumed rather simple networks, the parameters of the algorithm require further refinement for the complex topologies found in the “real Internet world”.

### 3.2 Simulation Results for the Simple Model

The following simulation results show that this algorithm can achieve a fair sharing of bandwidth. In this simulation the following assumptions were made:

- constant packet size of 512 bytes
- constant round trip times
- TCP sources always have data to send
- the packet interarrival times at the gateway are uniformly distributed
- the queue size is large enough such that no packets get lost

\(^{13}\)This might be not robust enough. The number of ECN echo ACKs could be increased to a few packets, say three. If too many ECN echo ACK are send out, the TCP sender might interpret them as a new instant of congestion an reduces the window again.
The parameters for New-ECN TCP and the RED gateway were:

- $p_{max} = 0.25$ : RED gateway: maximum marking probability
- $max_{th} = 20$ : RED gateway: maximum threshold level (in packets)
- $min_{th} = 10$ : RED gateway: minimum threshold level (in packets)
- $w_q = 0.002$ : RED gateway: weighting factor for the queue average
- $packet_{size} = 512$ bytes
- $\alpha_{wnd} = 0.9$ : window reduce factor
- $\alpha_{slp} = 0.7$ : window slope reduce factor
- $\beta = 16$ : window slope increase factor
- Link 1–3: 5 Mbps
- Connection X: $RTT = 40$ms
- Connection Y: $RTT = 20$ms

The Figure 3.2 shows that the rates for each connection are almost the same. In this figure, we can recognize one problem with this model: the connections are synchronized, which is most likely an artifact caused by the implementation of the model. Computing...
the fairness index for this simulation

\[ F_I = \frac{\left( \sum_{i=1}^{2} rate_i \right)^2}{2 \cdot \sum_{i=1}^{2} rate_i} \approx 0.99 \]

(3.6)
demonstrates that the algorithm already achieves a good fair sharing. In Figure 3.3 the window size is traced over the time. Computing the average window size for both connections, we get \( \overline{\text{cwnd}_x} = 18.1 \text{ packets} \) and \( \overline{\text{cwnd}_y} = 10.4 \text{ packets} \). The ratio of the average window is slightly higher than the ratio of the RTT’s. Thus, the mean window size is almost linear in RTT. Looking at Figure 3.4 we can see that the bandwidth allocation for each connection is stabilizing close to the fairness line.

This simulation shows some limitations of the implementation in Matlab. To gain more confidence about the results and test the algorithm for other network topologies, the New-ECN algorithm was implemented in the UCB/LBL/VINT Network Simulator "ns" [6].
Rates of the connections $x$ and $y$.

<table>
<thead>
<tr>
<th>Line Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>solid</td>
<td>rates $(x, y)$</td>
</tr>
<tr>
<td>dashed</td>
<td>rate $x+y=5$ Mbps</td>
</tr>
<tr>
<td>dashed dotted</td>
<td>Fairness Line</td>
</tr>
</tbody>
</table>

Figure 3.4: Simple model: rate trace
4 Verifying Results in the Network Simulator (NS)

4.1 Network Simulation Topology and Parameters

For verifying the results of the previous model, the following network topology with one bottleneck gateway was used:

![Simulation topology diagram]

The number of flows varied from one to six. In all simulations, TCP-Reno was used for the TCP flows and each flow was fed with an FTP traffic generator. The RED gateway used the following parameters:

- $p_{max} = 0.1$ : maximum marking probability
- $max_{th} = 20$ : maximum threshold level (in packets)
- $min_{th} = 10$ : minimum threshold level (in packets)
- $w_q = 0.002$ : weighting factor for the queue average

The New-ECN TCP-Reno used the following parameters:

- $\text{packetsize} = 512$ bytes
- $\text{tcpTick} = 0.01$ : TCP clock (in seconds)
- $\alpha_{wnd} = 0.9$ : window reduce factor
- $\alpha_{slp} = 0.9$ : window slope reduce factor
4 VERIFYING RESULTS IN THE NETWORK SIMULATOR (NS)

- $\beta = 16$ : window slope increase factor
- $wnd_{max} = 128$ packets (64Kb) : maximum window size

4.2 Fairness of New-ECN

In this subsection, we look into the fairness of the New-ECN algorithm. We studied the behavior of the New-ECN algorithm with multiple connections and compared New-ECN TCP-Reno with ECN TCP-Reno.

4.2.1 4 New-ECN TCP-Reno Connections

Simulation Setup 1 This simulation uses 4 New-ECN TCP-Reno connection. The parameters for the links can be found in Table 4.1. The four TCP-Reno flows with the New-ECN algorithm are started at the times 0s, 0.5s, 1s, 1.5s (TCP 1, TCP 2, TCP 3, TCP 4 respectively). These four flows compete for the bandwidth of the gateway (link 7). The round trip delays for each flow can be found in Table 4.2.

<table>
<thead>
<tr>
<th></th>
<th>Link 1</th>
<th>Link 2</th>
<th>Link 3</th>
<th>Link 4</th>
<th>Link 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay</td>
<td>94ms</td>
<td>34ms</td>
<td>14ms</td>
<td>1ms</td>
<td>1ms</td>
</tr>
<tr>
<td>Bandwidth</td>
<td>10Mbps</td>
<td>10Mbps</td>
<td>10Mbps</td>
<td>10Mbps</td>
<td>10Mbps</td>
</tr>
</tbody>
</table>

Table 4.1: 4 New-ECN TCP-Reno flows: link parameter setup 1

<table>
<thead>
<tr>
<th></th>
<th>TCP 1</th>
<th>TCP 2</th>
<th>TCP 3</th>
<th>TCP 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection</td>
<td>S1 → D</td>
<td>S2 → D</td>
<td>S3 → D</td>
<td>S4 → D</td>
</tr>
<tr>
<td>$RTT_{min}$</td>
<td>190ms</td>
<td>70ms</td>
<td>30ms</td>
<td>4ms</td>
</tr>
</tbody>
</table>

Table 4.2: 4 New-ECN TCP-Reno flows: round trip delays
Simulation Setup 2 This simulation is similar to the previous one, but it uses a different link speed for link 7. The link parameters for this setup can be found in Table 4.3. The minimums RTTs for the TCP connections are the same as for the previous setup (Table 4.2).

<table>
<thead>
<tr>
<th>Link 1</th>
<th>Link 2</th>
<th>Link 3</th>
<th>Link 4</th>
<th>Link 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay</td>
<td>94ms</td>
<td>34ms</td>
<td>14ms</td>
<td>1ms</td>
</tr>
<tr>
<td>Bandwidth</td>
<td>10Mbps</td>
<td>10Mbps</td>
<td>10Mbps</td>
<td>2Mbps</td>
</tr>
</tbody>
</table>

Table 4.3: 4 New-ECN TCP-Reno flows: link parameter setup 2

Analysis of the Results To analyze the simulations we look at the ACK sequence numbers. The sequence numbers of each connection are plotted over time in one graph. The slope of the lines represents the rate of the flow. Hence, if two flows have the same slope in the sequence number plot, they send at the same rate. Figure 4.2 shows the ACK sequence number plot for the simulation setup 1. After a few seconds, the TCP flows 1, 2 and 3 have similar slopes, so they send at similar rates. The fourth flow has a slightly higher sending rate. Some experiments showed that it is possible to get the fourth flow closer to the other flows by modifying the value of $p_{max}$ of the RED gateway. The role of this parameter is further discussed later in this section. To evaluate the performance of the algorithm we calculate the fairness index for this simulation:

$$F_l = \frac{\left(\sum_{i=1}^{4} rate_i\right)^2}{4 \cdot \sum_{i=1}^{4} rate_i^2} \approx 0.99$$  \hspace{1cm} (4.1)

This confirms the result found with the simple model in the previous section.

Figure 4.3 shows the evolution of congestion window for each flow. The peak in the beginning of the graph is caused by the slow start algorithm. After this phase, all connections entered the congestion avoidance phase and the New-ECN algorithm is activated. The graph also shows that the congestion window is oscillating after a few seconds around an almost constant value and is staying close to the mean value. As
Figure 4.2: 4 New-ECN TCP-Reno flows: ACK sequence number plot

Figure 4.4 shows the value of $S_{\text{wnd}}$ is also converging to a nearly constant value. The mean congestion window is plotted in the upper part of Figure 4.5. The ratio of the mean congestion windows is roughly linear in the round trip time. In the lower part of that figure, the mean value of the window slope is shown. Comparison of the ratios of these mean values reveals that these ratios are not linear in RTT, but proportional to $RTT^\alpha$ with $1 < \alpha < 2$. In Table 4.6 the difference between the ratio of the mean (smoothed) RTT raised to $\alpha$ and the ratio of the measured mean value of $S_{\text{wnd}}$ of two flows $i$ and $j$ is calculated in order to find an approximation for $\alpha$. From this Table it appears that the value of $\alpha$ is roughly $\frac{3}{2}$. Table 4.4 contains the mean values after 60 seconds and Table 4.5 the ratios of $c_{\text{wnd}}$, $sRTT^{14}$, and $S_{\text{wnd}}$.

Further simulations have shown that the maximum achievable fairness is bounded by the maximum window size. If the maximum window size is set too small, a slow connection can be prevented from claiming a fair share of the bandwidth.

$^{14}sRTT$ is the smoothed round trip time, calculated every time when $RTT_{\text{measured}}$ changes by $sRTT = \alpha sRTT + (1 - \alpha) RTT_{\text{measured}}$.
Figure 4.3: 4 New-ECN TCP-Reno flows: congestion window plot

Figure 4.4: 4 New-ECN TCP-Reno flows: $S_{wnd}$ plot
Running the same simulation with a smaller link speed of link 7 (Table 4.3) reveals a certain sensitivity to the choice of $p_{max}$. In Figure 4.6 it is obvious that there is no longer a fair sharing of the bandwidth. The effect of the New-ECN algorithm is minimal; it behaves similarly to the normal ECN algorithm. Changing the value of $p_{max}$ to 0.2 we can again achieve a fair sharing. The sequence number plot for the simulation with this value is shown in Figure 4.7.

As this simulation and the simulation with many flows in Section 4.2.3 show, the value of $p_{max}$ is dependent on the grade of congestion, namely on the number of connec-
## VERIFYING RESULTS IN THE NETWORK SIMULATOR (NS)

<table>
<thead>
<tr>
<th></th>
<th>sRTT ratio</th>
<th>cwnd ratio</th>
<th>$\overline{S_{\text{cwnd}}}$ ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>$TCP_1 : TCP_4$</td>
<td>13.02</td>
<td>13.91</td>
<td>47.22</td>
</tr>
<tr>
<td>$TCP_2 : TCP_4$</td>
<td>5.00</td>
<td>5.70</td>
<td>12.68</td>
</tr>
<tr>
<td>$TCP_3 : TCP_4$</td>
<td>2.34</td>
<td>2.82</td>
<td>4.69</td>
</tr>
<tr>
<td>$TCP_1 : TCP_3$</td>
<td>5.56</td>
<td>4.94</td>
<td>10.07</td>
</tr>
<tr>
<td>$TCP_2 : TCP_3$</td>
<td>2.14</td>
<td>2.02</td>
<td>2.7</td>
</tr>
<tr>
<td>$TCP_1 : TCP_2$</td>
<td>2.60</td>
<td>2.44</td>
<td>3.72</td>
</tr>
</tbody>
</table>

Table 4.5: 4 New-ECN TCP-Reno flows: mean value ratios

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$\Delta r_{1,4}(\alpha)$</th>
<th>$\Delta r_{2,4}(\alpha)$</th>
<th>$\Delta r_{3,4}(\alpha)$</th>
<th>$\Delta r_{1,3}(\alpha)$</th>
<th>$\Delta r_{2,3}(\alpha)$</th>
<th>$\Delta r_{1,2}(\alpha)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>-34.20</td>
<td>-7.67</td>
<td>-2.35</td>
<td>-4.50</td>
<td>-0.56</td>
<td>-1.12</td>
</tr>
<tr>
<td>1.1</td>
<td>-30.99</td>
<td>-6.80</td>
<td>-2.14</td>
<td>-3.46</td>
<td>-0.40</td>
<td>-0.86</td>
</tr>
<tr>
<td>1.2</td>
<td>-25.46</td>
<td>-5.77</td>
<td>-1.92</td>
<td>-2.22</td>
<td>-0.21</td>
<td>-0.57</td>
</tr>
<tr>
<td>1.3</td>
<td>-19.10</td>
<td>-4.57</td>
<td>-1.67</td>
<td>-0.76</td>
<td>-0.02</td>
<td>-0.26</td>
</tr>
<tr>
<td>1.4</td>
<td>-10.87</td>
<td>-3.15</td>
<td>-1.40</td>
<td>0.99</td>
<td>0.19</td>
<td>0.09</td>
</tr>
<tr>
<td>1.5</td>
<td>-0.24</td>
<td>-1.49</td>
<td>-1.11</td>
<td>3.06</td>
<td>0.42</td>
<td>0.47</td>
</tr>
<tr>
<td>1.6</td>
<td>13.51</td>
<td>0.47</td>
<td>-0.79</td>
<td>5.51</td>
<td>0.67</td>
<td>0.89</td>
</tr>
<tr>
<td>1.7</td>
<td>31.28</td>
<td>2.77</td>
<td>-0.45</td>
<td>8.43</td>
<td>0.94</td>
<td>1.36</td>
</tr>
<tr>
<td>1.8</td>
<td>54.24</td>
<td>5.47</td>
<td>-0.07</td>
<td>11.89</td>
<td>1.22</td>
<td>1.87</td>
</tr>
<tr>
<td>1.9</td>
<td>83.93</td>
<td>8.63</td>
<td>0.34</td>
<td>16.00</td>
<td>1.53</td>
<td>2.43</td>
</tr>
<tr>
<td>2.0</td>
<td>122.30</td>
<td>12.36</td>
<td>0.79</td>
<td>20.89</td>
<td>1.87</td>
<td>3.05</td>
</tr>
</tbody>
</table>

$$\Delta r_{i,j}(\alpha) := \frac{sRTT_i^\alpha}{sRTT_j^\alpha} - \frac{S_{\text{cwnd},i}}{S_{\text{cwnd},j}}$$  \hspace{1cm} (4.2)

Table 4.6: 4 New-ECN TCP-Reno flows: approximation for $\alpha$
Figure 4.6: 4 TCP-Reno flows: ACK sequence numbers plot for setup 2, $p_{\text{max}} = 0.1$

...and the capacity of the bottleneck link. One solution to this problem could be an adaptive RED algorithm, which modifies the marking probability based on an estimate of the number of active connections as proposed in [24], or based on the average queue length as proposed in [22, 23]. In [23] it is shown that no single set of RED parameters will work well for different scenarios. If the value of $p_{\text{max}}$ is too small for a high load (many active flows) the average queue size of the RED gateway will be close to the upper threshold and often even exceed it. Therefore, most of the arriving packets are dropped and the RED gateway starts to behave like a drop-tail gateway. On the other hand, if the marking probability is too high, the connection will see too many marks and 'overreact'. Experimenting with the value of $p_{\text{max}}$ showed that simulation setup 1 can achieve a fair sharing for $\frac{1}{10} < p_{\text{max}} < \frac{1}{3}$. For $p_{\text{max}} > \frac{1}{3}$ we start seeing a bias against fast connections. Fortunately, the sensitivity to $p_{\text{max}}$ is only moderate. The interaction of a adaptive scheme as in [23] with New-ECN needs further study.
4.2.2 4 New-ECN TCP-Reno vs. 4 ECN TCP-Reno Connections

Simulation Setup In this simulation, two independent runs of four flows are compared. The first run is done with New-ECN TCP-Reno and the second with ECN capable TCP-Reno. The setup parameters can be found in Table 4.1 and Table 4.2. The TCP flows are started at the times 0s, 0.5s, 1s, 1.5s (TCP 1, TCP 2, TCP 3, TCP 4 respectively).

Analysis of the Results These simulations show when the benefit of the New-ECN algorithm begins. The sequence number plots for both simulations can be found in Figure 4.8. The upper part of this figure displays the complete plot for 60 seconds and the lower part zooms into the first ten seconds. After a few seconds, the flow with the longer RTT is sending faster than the flow with the same RTT without the New-ECN algorithm. In contrast, the flow with the short RTT is slowed down. After roughly ten seconds, all flows with the New-ECN algorithm are sending with a similar rate.
Comparing the fairness indices of the New-ECN with the normal ECN algorithm

\[
F_{I, \text{New-ECN}} = \frac{\left(\sum_{i=1}^{4} \frac{rate_i}{rate_i^2}\right)^2}{4 \cdot \sum_{i=1}^{4} \frac{rate_i}{rate_i^2}} \approx 0.99 \quad F_{I, \text{ECN}} = \frac{\left(\sum_{i=1}^{4} \frac{rate_i}{rate_i^2}\right)^2}{4 \cdot \sum_{i=1}^{4} \frac{rate_i}{rate_i^2}} \approx 0.48
\]  

(4.3)

shows that the fairness improved approximately by factor 2.

![Diagram](image)

Figure 4.8: 4 New-ECN vs. 4 ECN TCP-Reno flows: ACK sequence number plot

### 4.2.3 Many New-ECN TCP-Reno Connections

**Simulation Setup** This simulation is done to test the New-ECN algorithm with many flows. In particular, the algorithm is tested with 16 flows. The network topology is similar to the one shown in Figure 4.1, but now 8 sources are connected to the gateway (link 1 to link 8). The gateway is connected with one link to the TCP sink (link 9). For each source, two connections are established with the TCP sink. The parameters for the setup are given in Table 4.7 and Table 4.8. As earlier in Section 4.2.1 observed, the parameter \( p_{\text{max}} \) has a dependency on the grade of congestion. For this simulation a value of \( p_{\text{max}} = 0.2 \) was needed to achieve a fair sharing of bandwidth.
Table 4.7: Many New-ECN TCP-Reno flows: link parameter

<table>
<thead>
<tr>
<th>Link</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay [ms]</td>
<td>44</td>
<td>34</td>
<td>28</td>
<td>24</td>
<td>14</td>
<td>9</td>
<td>9</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Bandwidth [Mbps]</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 4.8: Many New-ECN TCP-Reno flows: round trip delays

<table>
<thead>
<tr>
<th>TCP</th>
<th>1, 9</th>
<th>2, 10</th>
<th>3, 11</th>
<th>4, 12</th>
<th>5, 13</th>
<th>6, 14</th>
<th>7, 15</th>
<th>8, 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow source</td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>S4</td>
<td>S5</td>
<td>S6</td>
<td>S7</td>
<td>S8</td>
</tr>
<tr>
<td>Flow sink</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>RTT\text{min}</td>
<td>90ms</td>
<td>70ms</td>
<td>58ms</td>
<td>50ms</td>
<td>30ms</td>
<td>20ms</td>
<td>20ms</td>
<td>10ms</td>
</tr>
<tr>
<td>Start time</td>
<td>0.0s, 0.25s, 0.5s, 0.75s, 2.0s, 2.25s, 2.5s, 2.75s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4.0s, 4.25s, 4.5s, 4.75s, 6.0s, 6.25s, 6.5s, 6.75s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.9: 16 New-ECN TCP-Reno connections: ACK sequence number plot
Analysis of the Results This simulation demonstrates that the algorithm also works with many active flows. Figure 4.9 shows the sequence number plot for this simulation. All flows have a similar slope in the sequence number plot and therefore they have similar rates. These results are consistent with the results found earlier in this section.

4.3 TCP / ECN-TCP Friendliness of New-ECN TCP-Reno

This subsection investigates the TCP and ECN-TCP friendliness of the New-ECN algorithm. The majority of the Internet traffic is based on different versions of TCP and a new protocol should not affect these established protocols. Therefore, the TCP friendliness is an important issue to be studied for a new protocol in order to determine if a protocol is deployable.

4.3.1 One New-ECN TCP-Reno vs. one ECN/non-ECN TCP-Reno Connection

Simulation Setup This simulation compares one New-ECN TCP-Reno flow with one TCP-Reno flow with and without ECN support in independent runs. Each flow is started at 0.0s at TCP source S1 and connected with the TCP sink. The parameters for the links are:

- Link 1: 10Mbps, 14ms delay
- Link 7: 5Mbps, 1ms delay

Analysis of the Results In Figure 4.10 the sequence number plots and the mean rates are compared. We can see that the New-ECN algorithm performs slightly better than the ECN extension to TCP and much better than simple TCP-Reno. ECN and New-ECN TCP react before packets get lost, so these connections see fewer packets dropped. Thus, ECN and New-ECN do not suffer from multiple losses and these
connections can achieve a better throughput. Additionally, New-ECN TCP can achieve a slightly better performance than ECN because in “steady state” New-ECN has a smaller variance in the window size, since it cuts the window with \( \alpha_{\text{wnd}} = 0.9 \). In contrast, ECN TCP-Reno cuts the window by \( \frac{1}{2} \).

S. Floyd and K. Fall defined TCP-friendly flows \([28]\) as a flow with an arrival rate that does not exceed the bandwidth of a corresponding TCP flow in the same environment. They developed the following bound on the maximum sending rate

\[
T \leq \frac{1.5 \sqrt{\frac{2}{3} \cdot B}}{R \cdot \sqrt{p}} \tag{4.4}
\]

where \( B \) is the maximum packet size, \( p \) the packet drop rate and \( R \) the minimum RTT. Calculating this bound for the New-ECN algorithm using the values from the previous simulation

- packet size \( B = 512 \text{ bytes} \)
- total packets sent: 72583 \text{ packets} \n- total dropped packets: 90 \text{ packets} \n- minimum RTT \( R = 30 \text{ ms} \)

we get

\[
T_{\text{New-ECN}} = \frac{1.5 \sqrt{\frac{2}{3} \cdot (512 \cdot 8) \text{ bit}}}{30 \text{ ms} \cdot \sqrt{\frac{90 \text{ dropped packets}}{72583 \text{ total packets}}}} \approx 4.75 \text{ Mbps} \approx 1160 \text{ packets/s} \tag{4.5}
\]

As we can see from Figure 4.10 the flows achieve the following mean rates

- TCP-Reno : 1077 \text{ packets/s} \n- ECN TCP-Reno : 1157 \text{ packets/s} \n- New-ECN TCP-Reno : 1208 \text{ packets/s} \n
The New-ECN flow is slightly higher than the TCP-friendliness bound. S. Floyd and K. Fall derived the definition under the assumption that only packet drops are cause for reducing the sending rate. In New-ECN and ECN in general, packet drops are reduced in favor of marked packets. Thus, the bound in this form might not be applicable to the New-ECN algorithm.
4.3.2 TCP Friendliness of New-ECN

Simulation Setup In this simulation, the TCP friendliness is studied. Six flows are competing for the bandwidth of a bottleneck gateway. Three of them are using the New-ECN algorithm and the rest are using TCP-Reno without ECN support. The parameters for this simulation can be found in Table 4.9 and Table 4.10. The flows TCP 1, TCP 2, and TCP 3 are using the New-ECN algorithm and TCP 4, TCP 5, and TCP 6 TCP-Reno without ECN. This simulation is compared with a simulation with all six flows using TCP-Reno without ECN support.

<table>
<thead>
<tr>
<th></th>
<th>Link 1</th>
<th>Link 2</th>
<th>Link 3</th>
<th>Link 4</th>
<th>Link 5</th>
<th>Link 6</th>
<th>Link 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay</td>
<td>34ms</td>
<td>14ms</td>
<td>1ms</td>
<td>34ms</td>
<td>14ms</td>
<td>1ms</td>
<td>1ms</td>
</tr>
<tr>
<td>Bandwidth</td>
<td>10Mbps</td>
<td>10Mbps</td>
<td>10Mbps</td>
<td>10Mbps</td>
<td>10Mbps</td>
<td>10Mbps</td>
<td>10Mbps</td>
</tr>
</tbody>
</table>

Table 4.9: TCP / ECN-TCP friendliness: link parameter
4 VERIFYING RESULTS IN THE NETWORK SIMULATOR (NS)

<table>
<thead>
<tr>
<th>Connection</th>
<th>TCP 1</th>
<th>TCP 2</th>
<th>TCP 3</th>
<th>TCP 4</th>
<th>TCP 5</th>
<th>TCP 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1 → D</td>
<td>70ms</td>
<td>30ms</td>
<td>4ms</td>
<td>70ms</td>
<td>30ms</td>
<td>4ms</td>
</tr>
<tr>
<td>S2 → D</td>
<td>0.0s</td>
<td>0.0s</td>
<td>0.0s</td>
<td>0.0s</td>
<td>0.0s</td>
<td>0.0s</td>
</tr>
<tr>
<td>S3 → D</td>
<td>0.0s</td>
<td>0.0s</td>
<td>0.0s</td>
<td>0.0s</td>
<td>0.0s</td>
<td>0.0s</td>
</tr>
<tr>
<td>S4 → D</td>
<td>0.0s</td>
<td>0.0s</td>
<td>0.0s</td>
<td>0.0s</td>
<td>0.0s</td>
<td>0.0s</td>
</tr>
<tr>
<td>S5 → D</td>
<td>0.0s</td>
<td>0.0s</td>
<td>0.0s</td>
<td>0.0s</td>
<td>0.0s</td>
<td>0.0s</td>
</tr>
<tr>
<td>S6 → D</td>
<td>0.0s</td>
<td>0.0s</td>
<td>0.0s</td>
<td>0.0s</td>
<td>0.0s</td>
<td>0.0s</td>
</tr>
</tbody>
</table>

Table 4.10: TCP / ECN-TCP friendliness: round trip delays

Analysis of the Results  This simulation studies the effect of New-ECN flows to normal TCP-Reno flows. Figure 4.11 shows the sequence number plots for this test. The upper part of this figure shows the sequence number plot for the mixed (New-ECN and TCP-Reno) flows and the lower part for the TCP-Reno only simulation. We can see that the three New-ECN flows have a slight effect on the TCP-Reno flows. The fast connection loses some bandwidth in favor of the New-ECN flows. After a few seconds, the three New-ECN flows try to share the bandwidth almost equally, whereas the three TCP-Reno flows claim the bandwidth in the expected biased way. The New-ECN flows see roughly half the capacity of the bottleneck link and share it in a fair way.

Figure 4.11: TCP friendliness: ACK sequence number plot
4.3.3 ECN-TCP Friendliness of New-ECN

Simulation Setup  This simulation is similar to the previous one. Now the ECN-TCP friendliness is studied. Again, six flows are competing for the bandwidth of a bottleneck gateway. This time three of them are using the New-ECN algorithm and the remaining are using the TCP-Reno with ECN support. The parameters for this simulation are the same and can be found in Table 4.9 and Table 4.10. The flows TCP 1, TCP 2, and TCP 3 are using the New-ECN algorithm and TCP 4, TCP 5, and TCP 6 are using the ECN algorithm. This simulation is then compared with all six flows using TCP-Reno with ECN support.

![Figure 4.12: ECN-TCP friendliness: ACK sequence number plot](image)

Analysis of the Results  From Figure 4.12 we can see that the three New-ECN flows do not significantly reduce the sending rate of the three ECN flows. The upper part of this figure shows the sequence number plot for the mixed (New-ECN and ECN) simulation and the lower part for the ECN only simulation. As in the previous simulation, the three New-ECN flows try again to share the bandwidth equally. The
bandwidth sharing of the ECN flows is also biased.

These simulations show that the New-ECN algorithm performs well even in a mixed protocol environment. Furthermore, it has no considerable effect on flows of other versions of TCP. In the TCP and ECN TCP friendliness simulations the New-ECN flows share roughly half the capacity of the bottleneck link a fair way.
5 Summary

5.1 Conclusions and Future Work

This report introduced an enhancement called New-ECN to TCP-Reno with ECN support. The aim of this algorithm was to achieve a fair sharing of available bandwidth of a bottleneck router. The performance and behavior of this enhancement have been studied for the single bottleneck case. With a variety of simulation scenarios, we have demonstrated that the New-ECN algorithm achieves a fair sharing of bandwidth. Furthermore, the simulations have shown that the New-ECN TCP-Reno algorithm is TCP friendly and ECN TCP friendly while competing with them for bandwidth. It does not affect other TCP-Reno (with or without ECN) flows. Additionally, New-ECN also performed well with many flows going through one bottleneck router. This report also revealed a sensitivity of the New-ECN algorithm to the marking probability $p_{\text{max}}$ at the RED gateway. We suggested an adaptive mechanism for the value of $p_{\text{max}}$ as a solution to this problem. The interaction and performance of New-ECN with an adaptive RED algorithm need further study.

So far, we have focused our research on the single bottleneck case. In the ‘real Internet world’ a network topology with multiple congested gateways is more likely. Therefore, we need to extend the study to this case. Preliminary results with a topology similarly to the one used in [5] have shown that the New-ECN algorithm provides also some benefit. It performs better than a TCP-Reno connection with and without ECN support in the same topology. Interestingly, in our simulation, the TCP-Reno flow without ECN support performs better than a flow with ECN. New-ECN does not have this problem in the multiple congested gateway simulation. These results need further research to verify a benefit of the New-ECN algorithm.

Intuitively, we believe that the New-ECN algorithm has an advantage over the constant increase algorithm [5], since it tries to ‘find’ the right window increase value in an adaptive way.
5 SUMMARY

The parameters used for the New-ECN algorithm were chosen initially arbitrarily. They were refined in many simulation runs, but we do not claim that the values of the parameters we used in this report are the optimal set of parameters. To find an optimal set, the New-ECN algorithm needs further research. An important field of future research is the development of a stochastic model for the New-ECN algorithm. This model would make the understanding and the analysis of the dependency on the parameters much easier.

Altogether, this report shows the potential benefit that the New-ECN algorithm can bring to TCP flows. Although we have run many simulations with different scenarios, further testing is required. One question is what tests should a new protocol pass before it can be deployed in the network.

5.2 Acknowledgements

First, I would like to thank Prof. Jean Walrand for being my advisor during my research project. The guidance and support I got from him helped me finding my way through this project. His enthusiasm and intuition about networking have been very inspiring. I appreciate that he had time for regular meetings although he was on an industrial leave. I would also like to thank Matt Siler for helping me getting started in my research and for his technical help. Thanks to Richard La for the few very helpful discussions on the New-ECN algorithm.

I thank my officemates, Yogesh Bhumralkar, Lawrence Ip, Rene Vidal and Kiran, for the technical advice, especially for the help with LATEX. They made my stay at the University of California at Berkeley very interesting and unforgettable.

The research was supported in part by SBC Communications, a MICRO grant from the State of California, and Odysia Systems. Tilo Hamann was supported by a scholarship provided by the Ditze-Foundation of the Technical University of Hamburg-Harburg.
References


REFERENCES


A Matlab-File for the Simple Model Simulation

clear;

% Basic parameters
Packet=512*8;
t_incr=10e-3;
MaxSteps=500;
RTTt=40e-3;
RTTy=20e-3;
MaxRateX=5e6;
MaxRateY=5e6;

% New-ECN TCP parameters
alpha_slp=0.70;
alpha_vnd=0.9;
beta=16;

% TCP connection parameters
% Connection x
MaxIncrX=t_incr/RTTt;
MaxCondX=MaxRateX*RTTt/Packet;
xO=1;
ssthreshX=65535*8/Packet;
counterX=1;
jX=1;
xv(1)=xO;
DelayX=floor(RTTt/t_incr);
FlagX=1;
Swnd_X=1;

% Connection y
MaxIncrY=t_incr/RTTt;
MaxCondY=MaxRateY*RTTt/Packet;
yO=1;
ssthreshY=65535*8/Packet;
counterY=1;
jY=1;
yv(1)=yO;
DelayY=floor(RTTY/t_incr);
FlagY=1;
Swnd_Y=1;

% Outgoing link
c=5e6;
Servicetime=1/(c/Packet);

% RED parameters:
MaxTh=20;
MinTh=10;
MaxP=0.25;
wq=0.002;
C1=MaxP/(MaxTh-MinTh);
C2=(MaxP+MinTh)/(MaxTh-MinTh);
q_i_time=0;
counter=1;
p_dro=0;
R=rand(1,1);
queue(1)=0;
avgqueue(1)=0;

% Other parameters
HandlingPacket=-1;
% MATLAB-FILE FOR THE SIMPLE MODEL SIMULATION

npx=0; % number of packets received on connection x [p]
npy=0; % number of packets received on connection y [p]
npx(l)=0; % number of marked packets for connection x [p]
npy(1)=0; % number of marked packets for connection y [p]
MaxServedPackets=(c/Packetsize)*t_incr; % max number of packets that c be served during one step

CarryX=0; % variable used to calculate the right number of packets at the router
CarryY=0; % variable used to calculate the right number of packets at the router

% Init log variables
npx_log(l)=npx;
npy_log(l)=npy;
markx_log(l)=markx(l);
marky_log(l)=marky(l);
npt_log(l)=0;
qi_log(l)=0;
mmx(l)=xw(l)*Packetsize/RTTx;
mmx(l)=yw(l)/Packetsize/RTTy;
S_wnd_X_log(l)=l;
S_wnd_Y_log(l)=l;

% variable used to calculate the right number of packets at the router
% log number of packets received on connection x [p]
% log number of packets received on connection y [p]
% log marks received on x
% log marks received on y
% log number of packets arrived at the router
% log queue idle time
% mean rate for connection x
% mean rate for connection y
% log window slope connection x
% log window slope connection x
% variable to handle the increase of the slope
% variable to handle the increase of the slope

% Start loop
for k=2:MaxSteps
    % updating TCP parameters for connection X, window based
    if (k-DelayX<l)
        % marks not effective yet
        if (xw(k-1)<ssthreshX)
            % slow start
            xw(k)=min(MaxCwndX,xw(jx)+(xw(jx)*2-xw(jx))*MaxIncrX*counterX);
            if (counterx=1/MaxIncrX)
                jx=k;
                counterx=1;
            else
                counterx=counterx+1;
            end;
        else
            % congestion avoidance
            xw(k)=min(MaxCwndX,xw(k-1)+MaxIncrX);
        end;
    else
        % marks are effective now
        if ((npx(k-DelayX)==0))
            % no marks received
            npx(k-DelayX)=0; % reset marks
            Time_Now_X = k * t_incr;
            if ((FlagX<k) & (Time_Now_X >= (Last_Time_X+RTTx))) % increase slope once per RTT
                S_wnd_X=S_wnd_X+beta*RTTx^2;
                Last_Time_X = Time_Now_X;
            end;
            if (xw(k-1)<ssthreshX)
                % slow start
                xw(k)=min(MaxCwndX,xw(jx)+(xw(jx)*2-xw(jx))*MaxIncrX*counterX*S_wnd_X);
                if (counterx=1/MaxIncrX)
                    jx=k;
                    counterx=1;
                else
                    counterx=counterx+1;
                end;
            else
                % marks are effective now
                if ((npx(k-DelayX)==0))
                    % no marks received
                    npx(k-DelayX)=0; % reset marks
                    Time_Now_X = k * t_incr;
                    if ((FlagX<k) & (Time_Now_X >= (Last_Time_X+RTTx))) % increase slope once per RTT
                        S_wnd_X=S_wnd_X+beta*RTTx^2;
                        Last_Time_X = Time_Now_X;
                    end;
                    if (xw(k-1)<ssthreshX)
                        % slow start
                        xw(k)=min(MaxCwndX,xw(jx)+(xw(jx)*2-xw(jx))*MaxIncrX*counterX*S_wnd_X);
                        if (counterx=1/MaxIncrX)
                            jx=k;
                            counterx=1;
                        else
                            counterx=counterx+1;
                        end;
                    else
                        % congestion avoidance
                        xw(k)=min(MaxCwndX,xw(k-1)+MaxIncrX);
                    end;
                end;
            end;
        end;
    end;
end;
A MATLAB-FILE FOR THE SIMPLE MODEL SIMULATION

end;
else
    % congestion avoidance
    xw(k)=min(MaxCwndX,xw(k-1)+MaxIncrX*S_wnd_X);
end;
else
    % marks received
    % update window and threshold
    Last_Time_X = k*t_incr;
    Time_Now_2_X=k*t_incr;
    if (FlagX==1) % first marks received
        FlagX=0;
        xw(k)=xw(k-1)*(alpha_wnd);
        Last_Time_2_X=Time_Now_2_X;
    else
        if (Time_Now_2_X >= (Last_Time_2_X+RTTx)) % update window once per RTT
            S_wnd_X=S_wnd_X*(alpha_slp); % decrease slope
            xw(k)=xw(k-1)*(alpha_wnd); % decrease window
            ssthreshX=min(xw(k)/2,1); % decrease ssthresh
            Last_Time_2_X=Time_Now_2_X;
        else
            xw(k)=xw(k-1);
        end;
    end;
end;
end;

% log variables for connection x
ssthreshX_log(k)=ssthreshX; % log ssthresh
x(k)=x(k)*Packetsize/RTTx; % calculate corresponding rate
S_wnd_X_log(k)=S_wnd_X; % log window slope
mmx(k)=mean(x); % log mean rate

% updating TCP parameters for connection y, window based
if (k-DelayY<1)
    % marks not effective yet
    if (yw(k-1)<ssthreshY)
        % slow start
        yw(k)=min(MaxCwndY,yw(jy)+(yw(jy)*2-yw(jy))*MaxIncrY*countery);
        if (countery>=1/MaxIncrY)
            jy=k; countery=1;
        else
            countery=countery+1;
        end;
    else
        % congestion avoidance
        yw(k)=min(MaxCwndY,yw(k-1)+MaxIncrY);
    end;
else
    % marks are effective now
    if ((npmy(k-DelayY)==0)) % no marks received
        npmy(k-DelayY)=0; % reset marks
        Time_Now_Y = k * t_incr;
        if ((FlagY<1)&(Time_Now_Y >= (Last_Time_Y+RTTy))) % increase slope once per RTT
            S_wnd_Y=S_wnd_Y*beta*RTTy^2;
            Last_Time_Y = Time_Now_Y;
        end;
        if (yw(k-1)<ssthreshY)
            % slow start
            yw(k)=min(MaxCwndY,yw(jy)+(yw(jy)*2-yw(jy))*MaxIncrY*countery*S_wnd_Y);
            if (countery>=1/MaxIncrY)
                jy=k; countery=1;
            else
                countery=countery+1;
            end;
        else
            % congestion avoidance
            yw(k)=min(MaxCwndY,yw(k-1)+MaxIncrY);
        end;
    else
        % marks not effective yet
        if (yw(k-1)<ssthreshY)
            % slow start
            yw(k)=min(MaxCwndY,yw(jy)+(yw(jy)*2-yw(jy))*MaxIncrY*countery*S_wnd_Y);
            if (countery>=1/MaxIncrY)
                jy=k; countery=1;
            else
                countery=countery+1;
            end;
        else
            % congestion avoidance
            yw(k)=min(MaxCwndY,yw(k-1)+MaxIncrY);
        end;
    end;
end;
countery=countery+l;
end;
else
   % congestion avoidance
   yw(k)=min(MaxCwndY,yw(k-1)+MaxIncrY*S_wnd_y);
end;
else
   % marks received
   % update window and threshold
   Last_Time_Y=k*t_incr;
   Time_Now_2_Y=k*t_incr;
   if (FlagY==1) % first marks received
      FlagY=0;
      yw(k)=yw(k-1)*(alpha_wnd);
      Last_Time_2_Y=Time_Now_2_Y;
   else
      if (Time_Now_2_Y >= (Last_Time_2_2*RTTy)) % update window once per RTT
         S_wnd_y=S_wnd_y*(alpha_slp); % decrease slope
         yw(k)=yw(k-1)*(alpha_wnd); % decrease window
         ssthreshY=max(yw(k)/2,1); % decrease ssthresh
         Last_Time_2_Y=Time_Now_2_Y;
      else
         yw(k)=yw(k-1);
      end;
   end;
   npmy(k-DelayY)=0; % reset marks
end;
end;

% log variables for connection y
ssthreshY_log(k)=ssthreshY; % log ssthresh
y(k)=yw(k)*Packetsize/RTTy; % calculate corresponding rate
S_wnd_Y_log(k)=S_wnd_Y; % log window slope
mmy(k)=mean(y); % log mean rate
qi_log(k)=q_i_time; % log queue idle time

% queue management:
TempX=floor(xw(k))*t_incr/RTTx; % Calculating the number of arrived packets
TempY=floor(yw(k))*t_incr/RTTy; % Calculating the number of arrived packets
CarryX=CarryX+mod(TempX,1); % Calculating the number of arrived packets
CarryY=CarryY+mod(TempY,1); % Calculating the number of arrived packets
Temp2X=floor(CarryX); % Calculating the number of arrived packets
Temp2Y=floor(CarryY); % Calculating the number of arrived packets
CarryX=CarryX-Temp2X; % Calculating the number of arrived packets
CarryY=CarryY-Temp2Y; % Calculating the number of arrived packets
npX=floor(Temp1X)+Temp2X; % number of packets arrived from X during one step
npY=floor(Temp1Y)+Temp2Y; % number of packets arrived from Y during one step
npt=npX+npY; % total number of packets arrived during one step
npt_log(k)=npt; % log total number of packets arrived during one step
npX_log(k)=npX; % log number of packets arrived during one step for connection x
npY_log(k)=npY; % log number of packets arrived during one step for connection y
qi_log(k)=q_i_time; % log queue idle time
ServedPackets=0;

if (npt>0)
   ServePerRun(k)=(MaxServedPackets/npt); % number of packets served from the queue in each
   countto=(1/ServePerRun(k)); % run of while
else
   ServePerRun(k)=0;
end;

% serving queue for no arrival
if (npt==0)
   if (queue(k-1)==0)
      q_i_time=q_i_time+t_incr; % setting queue idle time for the avg calculations
   end;
end;
queue(k)=0;
else
  if(queue(k-1)>floor(MaxServedPackets)) % serve queue
    queue(k)=queue(k-1)-floor(MaxServedPackets);
  else
    queue(k)=0;
  end;
end;
if (queue(k)>0) % queue is not empty
  avgqueue(k)=avgqueue(k-1)+wq*(queue(k)-avgqueue(k-1));
else % queue is empty
  avgqueue(k)= (1-wq)*(q_i_time/Servicetime)+avgqueue(k-1);
end;
end;

npnz(k)=0; npsy(k)=0; % reset marks for each connection
NoX=0; NoY=0;
QCarry=0;

% some calculation in order to handle packets in some kind of random way
if (npt>0)
  if (npx==0)
    NoX=1;
  else
    if (npy==0)
      NoY=1;
    else
      Ratio=npx/npy;
      if (Ratio>1)
        help1=floor(Ratio);
        QCarry=mod(Ratio,1);
      else
        help1=floor(1/Ratio);
        QCarry=mod(1/Ratio,1);
      end;
    end;
  end;
else
  if (Ratio>1)
    if ((help1>1)&(npx>0))
      HandlingPacket=0;
      help1=help1-1;
    else
      HandlingPacket=1;
      help1=floor(Ratio+QCarry);
      QCarry=mod(Ratio+QCarry,1);
    end;
  else
    if ((help1>=1)&(npy>0))
      HandlingPacket=1;
      help1=help1-1;
    else
      HandlingPacket=0;
      help1=floor((1/Ratio)+QCarry);
    end;
end;
QCarry = mod((1/Ratio) + QCarry, 1);
end;
end;
end;
end;

queue(k) = queue(k) + 1;
% increase queue

if (queue(k) > 0)
    if (avgqueue(k) = avgqueue(k-1) * wq * (queue(k) - avgqueue(k-1));
    q_i_time = 0;
else
    avgqueue(k) = (1-wq)*(q_i_time/Servicetime)*avgqueue(k-1);
end;

% check for marking
if (((avgqueue(k) > MinTh) && (avgqueue(k) < MaxTh)) % queue size is between the thresholds
    count = count + 1;
p_drop = C1 * avgqueue(k) - C2;
    if ((count > 0) && (count >= (R/p_drop)))
        if (HandlingPacket == 0)
            npmx(k) = npmx(k) + 1; % mark packet x
        end;
        if (HandlingPacket == 1)
            npmy(k) = npmy(k) + 1; % mark packet y
        end;
        count = 0;
    end;
    if (count == 0) R = rand(1, 1);
else
    if (avgqueue(k) >= MaxTh) % queue size is beyond the upper threshold
        if (HandlingPacket == 0)
            npmx(k) = npmx(k) + 1;
        end;
        if (HandlingPacket == 1)
            npmy(k) = npmy(k) + 1;
        end;
        count = 1;
    else
        count = 1;
    end;
end;

% serving queue
if ((queue(k) > 0) && (ServedPackets <= (MaxServedPackets)))
    queue(k) = max(0, queue(k) - (ServePerRun(k)));
    ServedPackets = ServedPackets + (ServePerRun(k));
end;

% reducing number of unserved packets left
if (HandlingPacket == 0) npx = npx - 1;
if (HandlingPacket == 1) npy = npy - 1;
end;

% log variables
ServedPerRun(k) = ServePerRun(k) * npt;
marksx_log(k) = npmx(k);
marksy_log_log(k) = npmy(k);
end; % of for
B Modified Network Simulator TCP Source Code

The following subsections describe the changes to the source code to implement the New-ECN TCP-Reno algorithm in the ns Network Simulator distribution V.21b5. Each subsection shows only the modified function in the particular file. In the function the modified parts are framed by //--begin----- and //--end----- . To use the New-ECN algorithm, the TCP option windowOption_ and the new option windowOptionS_ are set to the value 8. The first option tells the source to use the New-ECN algorithm and the second changes the TCP-sink behavior for New-ECN. Additionally, the ECN option in sources and gateways needs to be turned on.

B.1 TCP.H

At the start of the file the block

```c
#define CLOSE_SSTHRESH_HALF 0x00000001
#define CLOSE_CWND_HALF 0x00000002
#define CLOSE_CWND_RESTART 0x00000004
#define CLOSE_CWND_INIT 0x00000008
#define CLOSE_CWND_ONE 0x00000010
#define CLOSE_SSTHRESH_halve 0x00000020
#define CLOSE_CWND_HALF 0x00000040
```

is extended by

```c
#define CLOSE_CWND_NEW 0x00000080
#define CLOSE_SSTHRESH_NEW 0x00000100
```

In class TcpAgent : public Agent the following declarations are added in the protected part:

```c
int First_Time_;  // flag, set to one after the first received mark
double Last_Time_;  // variable for handling window slope increase
double Slope_Window_;  // window slope
double Mark_Received_;  // flag, set to one for a received mark
```
B.2 TCP.CC

TcpAgent::TcpAgent() : Agent(PT_TCP),
t_seqno_(0), t_rtt_(0), t_srtt_(0), t_rttvar_(0),
t_backoff_(0), tsPeer_(0),
rtx_timer_(this), delsnd_timer_(this), burstsnd_timer_(this),
dupsacks_(0), curseq_(0), highest_ack_(0), cwnd_(0), sssthresh_(0),
count_(0), fcnt_(0), rtt_active_(0), rtt_seq_(-1), rtt_ts_(0,0),
maxseq_(0), cong_action_(0), ecn_burst_(0), ecn_backoff_(0),
ect_(0), restart_bugfix_(1), closed_(0), nrexmit_(0),
//begin---------------------------------------------------------------
Slope_Window_(1), Last_Time_(0), First_Time_(0), Mark_Received_(0)
//end-----------------------------------------------------------------
{
    // Defaults for bound variables should be set in ns-default.tcl.
    bind("window.", &wnd_);
    bind("windowInit.", &wnd_init_);
    bind("windowInitOption.", &wnd_init_option_);
    bind(bool("syn.", &syn_);
    bind("windowOption.", &wnd_option_);
    bind("windowConstant.", &wnd_const_);
    bind("windowThreshold.", &wnd_th_);
    bind("delay-growth.", &delay_growth_);
    bind("overhead.", &overhead_);
    bind("tcpTick.", &tcp_tick_);
    bind(bool("ecn.", &ecn_);
    bind(bool("old_ecn.", &old_ecn_);
    bind("eln.", &eln_);
    bind("eln_rxmit_thres.", &eln_rxmit_thres_);
    bind("packetSize.", &size_);
    bind("tcpip_base_hdr.size.", &tcpip_base_hdr_size_);
    bind(bool("bugFix.", &bug_fix_);
    bind(bool("slow_start_restart.", &slow_start_restart_);
    bind(bool("restart_bugfix.", &restart_bugfix_);
    bind(bool("timestamps.", &ts_option_);
    bind("maxburst.", &maxburst_);
    bind("maxcwnd.", &maxcwnd_);
    bind("maxrto.", &maxrto_);
    bind("rttt_init.", &rttt_init_);
    bind("rttvar_init.", &rttvar_init_);
    bind("rtxcur_init.", &rtxcur_init_);
    bind("T_SRTT_BITS", &T_SRTT_BITS);
    bind("T_RTTVAR_BITS", &T_RTTVAR_BITS);
    bind("rttvar_exp.", &rttvar_exp_);
    bind("dupsacks.", &dupsacks_);
    bind("seqno.", &seqno_);
    bind("t_seqno.", &t_seqno_);
    bind("ack.", &highest_ack_);
    bind("cwnd.", &cwnd_);
    bind("wnd.", &wnd_);
    bind("sssthresh.", &sssthresh_);
    bind("rtt.", &rtt_);
    bind("srtt.", &srtt_);
    bind("rttvar.", &rttvar_);
    bind("backoff.", &backoff_);
    bind("maxseq.", &maxseq_);
    //begin---------------------------------------------------------------
    bind("slope_window.", &slope_window_); // window slope
    bind("last_time.", &last_time_); // variable for handling window slope increase
    bind("first_time.", &first_time_); // flag, set to one after the first received mark
    bind("mark.", &mark_); // flag, set to one for a received mark
```c
//ifdef OFF_HDR
    bind("off_ip_", &off_ip_);
    bind("off_tcp_", &off_tcp_);
#endif
#endif

// reset used for dynamically created agent
reset();

void TcpAgent::reset() {
    rtt_init();
    /*XXX lookup variables */
```
dupacks_ = 0;
curseq_ = 0;
set_initial_window();

double Time_Now;
double f;

if (cvnd_ < ssthresh_) {
    /* slow-start (exponential) */
    cwnd_ = 1;
} else {
    /* linear */
    switch (wnd_option_) {
        case 0:
            if (++count_ >= cwnd_)
                count_ = 0;
            ++cwnd_;
            break;

        case 1:
            /* This is the standard algorithm. */
            cwnd_ = 1 / cwnd_;
            break;

        case 2:
            /* These are window increase algorithms
                * for experimental purposes only. */
            f = (t_srtt_ >> T_SRTT_BITS) * tcp_tick_;
            f *= f;
            if (f > cwnd_)
                fcnt_ = 0;
            ++cwnd_;
            break;

        case 3:
f = awnd_;  
f *= f;  
f *= wnd_const_;  
f += fcnt_;  
if (f > cwnd_) {
    fcnt_ = 0;  
    ++cwnd_;  
} else  
    fcnt_ = f;  
break;

case 4:  
f = awnd_;  
f *= wnd_const_;  
f += fcnt_;  
if (f > cwnd_) {
    fcnt_ = 0;  
    ++cwnd_;  
} else  
    fcnt_ = f;  
break;

case 5:  
f = (t_srtt_ >> T_SRTT_BITS) * tcp_tick_;  
f *= wnd_const_;  
f += fcnt_;  
if (f > cwnd_) {
    fcnt_ = 0;  
    ++cwnd_;  
} else  
    fcnt_ = f;  
break;

//begin-----------------------------------------------------------------------
case 8:  
    /* New-ECN TCP algorithm */  
    Time_Now = Scheduler::instance().clock();  // get time  
    f = (t_srtt_ >> T_SRTT_BITS) * tcp_tick_;  // convert time to [s]  
    // increase slope once per RTT  
    if (((Time_Now - f) >= Last_Time_) && (First_Time_)) {  
        Slope_Window_ = (Slope_Window_) + (16) * pow(f, 2);  
        Last_Time_ = Time_Now;  
    }  
    cwnd_ += ((1/cwnd_) * Slope_Window_);  // increase window  
    break;
//end-----------------------------------------------------------------------

default:  
    #ifdef notdef
        #define XXX
        error("illegal window option \%d", wnd_option_);  
    #endif
    abort();
}  

// if maxcwnd is set (nonzero), make it the cwnd limit  
if (maxcwnd_ && (int(cwnd_) > maxcwnd_))  
    cwnd_ = maxcwnd_;  

return;

void TcpAgent::slowdown(int how) {

int halfwin = int(window() / 2);

//begin-----------------------------------------------
double factorwnd;  // alpha wnd
double factorslope;  // alpha slp
factorwnd=0.9;
factorslope=0.9;
int smallwin = int(window() * factorwnd);  // reduced value of the congestion window
//end-----------------------------------------------

if (how & CLOSE_SSTHRESH_HALF)
    ssthresh_ = halfwin;
if (how & CLOSE_SSTHRESH_NEW)
    ssthresh_ = int(ssthresh_ * factorwnd);
if (ssthresh_ < 2)
    ssthresh_ = 2;
if (how & CLOSE_CWND_HALF)
    cwnd_ = halfwin;
else if (how & CLOSE_CWND_RESTART) {
    cwnd_ = int(wnd_restart_);
}
else if (how & CLOSE_CWND_INIT) {
    cwnd_ = int(wnd_init_);
}
else if (how & CLOSE_CWND_ONE) {
    cwnd_ = 1;
}
//begin-----------------------------------------------
else if (how & CLOSE_CWND_NEW) {  // reduce window (mark received)
    if (smallwin>=0)
        cwnd_ = smallwin;
}
if (how & (CLOSE_CWND_HALF|CLOSE_CWND_NEW)) {  // reduce window slope (mark received)
    if (First_Time_ & (wnd_option_ == 8))
        Slope_Window_ = Slope_Window_ * factorslope;
}
if (how & (CLOSE_CWND_INIT|CLOSE_CWND_RESTART|CLOSE_CWND_ONE)) {
    // reduce window after a lost packet
    if (First_Time_ & (wnd_option_ == 8) & (Slope_Window_>.0))
        Slope_Window_ = Slope_Window_ *0.5;
}
if (how & (CLOSE_CWND_HALF|CLOSE_CWND_RESTART|CLOSE_CWND_INIT|CLOSE_CWND_ONE|CLOSE_CWND_NEW))
    cong_action_ = TRUE;
Last_Time_ = Scheduler::instance().clock();  // Time now, reset increase
//end-----------------------------------------------
fcnt_ = count_ = 0;
}

void TcpAgent::ecn(int seqno) {
if (seqno > recover_ ||
    last_cwnd_action_ == CWND_ACTION_TIMEOUT)>({wnd_option_ ==7}) ) {
    if (wnd_option_ ==7) {
        recover_ = maxseq_;//(int( maxseq_ - highest_ack_)*/0.75));
        last_cwnd_action_ = CWND_ACTION_ECN;
    }
    if (cwnd_ <= 1.0) {
        if (ecn_backoff_)
            rtt_backoff();
        else ecn_backoff_ = 1;
    } else ecn_backoff_ = 0;
//begin-----------------------------------------------
if (vnd_option_ == 8) {
    slowdown(CLOSE_CWND_NEW|CLOSE_SSTHRESH_NEW);
    First_Time_=l;
} else {
    slowdown(CLOSE_CWND_HALF|CLOSE_SSTHRESH_HALF);
}

void TcpAgent::recv_newack_helper(Packet *pkt) {
    hdr.tcp *tcp = hdr.tcp::access(pkt);
    newack(pkt);

    if (vnd_option_ == 8) {
        if (!ect_ && !hdr_flags::access(pkt)->ecnecho()) {
            opencwnd();
        }
    } else {
        /* If "old_ecn", this is not the first ACK carrying ECN-Echo
         * after a period of ACKs without ECN-Echo.
         * Therefore, open the congestion window. */
        if (!ect_ && !hdr_flags::access(pkt)->ecnecho() || (old_ecn_ && ecn_burst_)) {
            opencwnd();
        }
    }
}

if (ect_) {
    if (!hdr_flags::access(pkt)->ecnecho())
        ecm_backoff_ = 0;
    if (!ecn_burst_ && hdr_flags::access(pkt)->ecnecho())
        ecm_burst_ = TRUE;
    else if (ecn_burst_ && !hdr_flags::access(pkt)->ecnecho())
        ecm_burst_ = FALSE;
}

if (!ect_ && hdr_flags::access(pkt)->ecnecho() &&
    hdr_flags::access(pkt)->cong_action())
    ect_= 1;
/* if the connection is done, call finish() */
if ((highest_ack_ >= curseq_-1) && !closed_) {
    closed_ = 1;
    finish();
}
}

void TcpAgent::recv(Packet *pkt, Handler*) {
    hdr.tcp *tcp = hdr_tcp::access(pkt);
    #ifdef notdef
    if (pkt->type_ != PT_ACK) {
        Tel::instance().evalf("% error \"received non-ack\"",
                              name());
        Packet::free(pkt);
        return;
    }
    #endif
    ++nackpack_;
    ts_peer_ = tcp->ts();
}
int ecnecho = hdr_flags::access(pkt)->ecnecho;

//begin----------------------------------------------------------------------------------------
if (ecnecho & ecn_)
  ecn(tcph->seqno());
  Mark_Received_ = 1;  // trace marks
} else if (ecnecho & ecn_)
  Mark_Received_ = 0;
//end------------------------------------------------------------------------------------------

recv_helper(pkt);

/* grow cwnd and check if the connection is done */
if (tcph->seqno() > last_ack_)
  recv_newack_helper(pkt);
  if (last_ack_ == 0 & delay_growth_)
    cwnd_ = initial_window();
} else if (tcph->seqno() == last_ack_)
  if (hdr_flags::access(pkt)->ecn_ & ecn_)
    tcp_ecn(pkt);
    return;
  if (++dupacks_ == NUMDUPACKS)
    dupack_action();
}
Packet::free(pkt);
/*
 * Try to send more data.
 */
send_much(0, 0, maxburst_);

B.3 TCP-RENO.CC

void RenoTcpAgent::recv(Packet *pkt, Handler*){
  hdr_tcp *tcph = (hdr_tcp*)pkt->access(off_tcp_);
  if (pkt->type_ != PT_ACK) {
    fprintf(stderr,
      "ns: configuration error: tcp received non-ack\n");
    exit(1);
  }
  ++nackpack_;
  ts_peer_ = tcph->ts();
  //begin----------------------------------------------------------------------------------------
  if (((hdr_flags*)pkt->access(off_flags_))->ecn_ & ecn_)
    ecn(tcph->seqno());
    Mark_Received_ = 1;  // trace marks
} else if (!((hdr_flags*)pkt->access(off_flags_))->ecn_ & ecn_)
  Mark_Received_ = 0;
//end------------------------------------------------------------------------------------------
recv_helper(pkt);
if (tcph->seqno() > last_ack_)
  dupwnd_ = 0;
recv_newack_helper(pkt);
if (last_ack_ == 0 && delay_growth_)
    cwnd_ = initial_window();
} else if (tcph->seqno() == last_ack_)
    if (((hdr_flags*)pkt->access(off_flags_))->ecn_ && eln_) {
        tcp_ecn(pkt);
        return;
    }
    if (++dupacks_ == NUMDUPACKS) {
        dupack_action();
        dupwnd_ = NUMDUPACKS;
    } else if (dupacks_ > NUMDUPACKS) {
        ++dupwnd_;// fast recovery
    }
} Packet::free(pkt);
#ifdef notyet
if (trace_)
    plot();
#endif

/**
 * Try to send more data
 */
if (dupacks_ == 0 || dupacks_ > NUMDUPACKS - 1)
    send_much(0, 0, maxburst_);

B.4 TCP-SINK.CC

void TcpSink::ack(Packet* opkt) {
    Packet* npkt = allocpkt();
double now = Scheduler::instance().clock();
    hdr_flags *sf;
    hdr_tcp *otcp = hdr_tcp::access(opkt);
    hdr_tcp *ntcp = hdr_tcp::access(npkt);
    ntcp->seqno() = acker_->Seqno();
    ntcp->ts() = now;
    if (ts_echo_bugfix_)/\TCP/IP Illustrated, Vol. 2, pg. 870*/
        ntcp->ts_echo() = acker_->ts_to_echo();
    else
        ntcp->ts_echo() = otcp->ts();
    hdr_ip* oip = (hdr_ip*)opkt->access(off_ip_);
    hdr_ip* nip = (hdr_ip*)npkt->access(off_ip_);
    nip->flowid() = oip->flowid();
    hdr_flags* of = (hdr_flags*)opkt->access(off_flags_);
    hdr_flags* nf = (hdr_flags*)npkt->access(off_flags_);
    if (save_ != NULL)
        sf = (hdr_flags*)save_->access(off_flags_);
    // Look at delayed packet being acked.
    if ((save_ != NULL && sf->cong_action()) || of->cong_action() )
        // Sender has responded to congestion.
        acker_->update_scm_unacked(0);
    if ((save_ != NULL && sf->ect() && sf->ce()) ||
...
B MODIFIED NETWORK SIMULATOR TCP SOURCE CODE

(of->ect() & of->ce())
// New report of congestion.
acker->update_ecn_unacked();

//BEGIN---------------------------------------------------------------------------
if (wnd_option_ == 8) {
  if ((save_ != NULL & sf->ect() & sf->ce()) || (of->ect() & of->ce())
    // Set EcnEcho bit.
    nf->ecnecho() = 1;
} else {
  if ((save_ != NULL & sf->ect()) || of->ect())
    // Set EcnEcho bit.
    nf->ecnecho() = acker->ecn_unacked();
}
//END---------------------------------------------------------------------------

if ((of->ect() & of->ecn_echo() ||
    (save_ != NULL & sf->ect() & sf->ecn_echo())
  // This is the negotiation for ECN-capability.
  // We are not checking for of->cong_action() also.
  // In this respect, this does not conform to the
  // specifications in the internet draft
  nf->ecn_echo() = 1;
  acker->append_ack((hdr.cmm*)npkt->access(off.cmn_),
    ntcp, otcp->seqno());
  add_to_ack(npkt);
  send(npkt, 0);
}

B.5 NS-DEFAULT.TCL

The following lines have been added to the ns-default.tcl file. These lines set the
default values for the new parameters of New-ECN.

Agent/TCP set slope_window_ 1.0
Agent/TCP set mark_ 0.0
Agent/TCP set last_time_ 0.0
Agent/TCP set first_time_ 0
Agent/TCPSink set windowOptionS_ 1
C TCL Scripts for the Simulations in Section 4

To activate the New-ECN algorithm for TCP flows in the network simulator, the following parameters need to be set:

- **Agent/TCP:** `windowOption`.
  - 8: New-ECN window algorithm
  - 1: standard TCP window algorithm

- **Agent/TCP:** `ecn`.
  - 1: ECN and New-ECN support for TCP flows
  - 0: no ECN or New-ECN support for TCP flows

- **Agent/TCPSink:** `windowOptionS`.
  - 8: New-ECN window algorithm
  - 1: standard TCP window algorithm

- **queue:** `setbit`.
  - 1: mark packets if they are from a ECN or New-ECN flow
  - 0: drop packets

All of the ns simulation scripts are using the following function to create data files from the trace-files for a Matlab analysis:

```tcl
# create data from the trace-files for a Matlab analysis
proc makeplot number { num
    set awkCodeCwnd {
        { if ($6 == "cwnd_") {
            print $1 >> "temp.t.cwnd";
            print $7 >> "temp.v.cwnd";
        } }
    }
    set awkCodeSrtt {
        { if ($6 == "srtt_") {
            print $1 >> "temp.t.srtt";
            print $7 >> "temp.v.srtt";
        } }
    }
    set awkCodeAck {
        { if ($6 == "maxseq_") {
            print $1 >> "temp.t.ack";
            print $7 >> "temp.v.ack";
        } }
    }
    set awkCodeMark {
        { if ($6 == "ack_") {
            print $8 >> "temp.m.ack";
        } }
    }
}
C TCL SCRIPTS FOR THE SIMULATIONS IN SECTION 4

C.1 4 New-ECN TCP-Reno Connections

The next script was used for the simulation setup 1 in Section 4.2.1:

```
# ns simulator script for section 4.2.1: 4 New-ECN TCP-Reno connections
# Setup 1: 4 New-ECN TCP-Reno connections, Link 7: 10Mbps, 1ms
set ns [new Simulator]

# create trace-files and define the trace parameters
```
C TCL SCRIPTS FOR THE SIMULATIONS IN SECTION 4

proc tracestart {} {
    global ns Itrace 2trace 3trace 4trace tcp1 tcp2 tcp3 tcp4

    set Itrace [open tcp1.tr v]
    set 2trace [open tcp2.tr w]
    set 3trace [open tcp3.tr w]
    set 4trace [open tcp4.tr w]

    $tcp1 tracevar cwnd_; $tcp1 tracevar srtt_
    $tcp1 tracevar ack_; $tcp1 tracevar maxseq_
    $tcp2 tracevar cwnd_; $tcp2 tracevar srtt_
    $tcp2 tracevar ack_; $tcp2 tracevar maxseq_
    $tcp3 tracevar cwnd_; $tcp3 tracevar srtt_
    $tcp3 tracevar ack_; $tcp3 tracevar maxseq_
    $tcp4 tracevar cwnd_; $tcp4 tracevar srtt_
    $tcp4 tracevar ack_; $tcp4 tracevar maxseq_

    $tcp1 attach $Itrace ; $tcp2 attach $2trace
    $tcp3 attach $3trace ; $tcp4 attach $4trace
}

# create data from the trace-file for a Matlab analysis
proc makeplot number {
    ...
}

# end the simulation
proc finish {} {
    global ns Itrace 2trace 3trace 4trace

    # close trace-files
    $ns flush-trace
    close $Itrace ; close $2trace
    close $3trace ; close $4trace

    # create Matlab data
    makeplot "1" ; makeplot "2"
    makeplot "3" ; makeplot "4"
    exit 0
}

# Create network topology:
# create nodes:
set node_(r1) [ns node]
set node_(d) [ns node]
set node_(s1) [ns node]
set node_(s2) [ns node]
set node_(s3) [ns node]
set node_(s4) [ns node]

# create links
$ns duplex-link $node_(s1) $node_(r1) 10Mb 1ms DropTail
$ns duplex-link $node_(s2) $node_(r1) 10Mb 14ms DropTail
$ns duplex-link $node_(s3) $node_(r1) 10Mb 34ms DropTail
$ns duplex-link $node_(s4) $node_(r1) 10Mb 94ms DropTail

# setup RED router
$ns duplex-link-op $node_(r1) $node_(d) queuePos 0.5
$ns queue-limit $node_(r1) $node_(d) 500
$ns queue-limit $node_(d) $node_(r1) 500

# set RED router parameters
# TCL SCRIPTS FOR THE SIMULATIONS IN SECTION 4

```tcl
set redq [[$ns link $node_(r1) $node_(d)] queue] $redq set setbit_true $redq set lintern_10 $redq set thresh_10 $redq set maxthresh_20 $redq set q_weight 0.002

# set general TCP parameters
Agent/TCP set tcpTick 0.01 Agent/TCP set windowOption 8 Agent/TCPSink set windowOptionS 8

# create TCP connections and set individual parameters
set tcpl [$ns create-connection TCP/Reno $node_(s1) TCPSink $node_(d) 0]
$tcpl set window 128 $tcpl set ecn 1 $tcpl set packetSize 512
set tcp2 [$ns create-connection TCP/Reno $node_(s2) TCPSink $node_(d) 1]
$tcp2 set window 128 $tcp2 set ecn 1 $tcp2 set packetSize 512
set tcp3 [$ns create-connection TCP/Reno $node_(s3) TCPSink $node_(d) 2]
$tcp3 set window 128 $tcp3 set ecn 1 $tcp3 set packetSize 512
set tcp4 [$ns create-connection TCP/Reno $node_(s4) TCPSink $node_(d) 3]
$tcp4 set window 128 $tcp4 set ecn 1 $tcp4 set packetSize 512

# attach ftp traffic generators

# schedule events
$ns at 0.0 "tracestart" $ns at 1.5 "ftp1 start" $ns at 1.0 "ftp2 start" $ns at 0.5 "ftp3 start" $ns at 0.0 "ftp4 start" $ns at 60.0 "finish"

# start simulation
$ns run
```

The next script was used for the simulation setup 2 with 4 New-ECN TCP-Reno flows in Section 4.2.1. One simulation was done with $p_{max} = 0.1$ ($\text{redq set lintern} 10$) and one with $p_{max} = 0.2$ ($\text{redq set lintern} 5$).

# ns simulator script for section 4.2.1: 4 New-ECN TCP-Reno connections
# Setup 2: 4 New-ECN TCP-Reno connections, Link 7: 2Mbps, 1ms
set ns [new Simulator]

# create trace-files and define the trace parameters
proc tracestart {} { global ns itrace 2trace 3trace 4trace tcp1 tcp2 tcp3 tcp4
set itrace [open tcp1.tr w] ; set 2trace [open tcp2.tr w]
C  TCL SCRIPTS FOR THE SIMULATIONS IN SECTION 4

```tcl
set 3trace [open tcp3.tr w] ; set 4trace [open tcp4.tr w]

$tcp1 tracevar cwnd_ ; $tcp1 tracevar srtt_  
$tcp2 tracevar cwnd_ ; $tcp2 tracevar srtt_  
$tcp2 tracevar ack_ ; $tcp2 tracevar maxseq_  
$tcp3 tracevar cwnd_ ; $tcp3 tracevar srtt_  
$tcp3 tracevar ack_ ; $tcp3 tracevar maxseq_  
$tcp4 tracevar cwnd_ ; $tcp4 tracevar srtt_  
$tcp4 tracevar ack_ ; $tcp4 tracevar maxseq_  
$tcp1 attach $1trace ; $tcp2 attach $2trace  
$tcp3 attach $3trace ; $tcp4 attach $4trace
```

# create data from the trace-file for a Matlab analysis
proc makeplot number {
  ...
}

# end the simulation
proc finish {} {
  global ns ltrace 2trace 3trace 4trace

  # close trace-files
  $ns flush-trace
  close $ltrace ; close $2trace
  close $3trace ; close $4trace

  # create Matlab data
  makeplot "1" ; makeplot "2"
  makeplot "3" ; makeplot "4"
  exit 0
}

# Create network topology:
# create nodes:
set node_(rl) [$ns node]
set node_(d} [$ns node]
set node_(s1) [$ns node]
set node_(s2) [$ns node]
set node_(s3) [$ns node]
set node_(s4) [$ns node]

# create links
$ns duplex-link $node_(s1) $node_(rl) 10Mb 1ms DropTail
$ns duplex-link $node_(s2) $node_(rl) 10Mb 14ms DropTail
$ns duplex-link $node_(s3) $node_(rl) 10Mb 34ms DropTail
$ns duplex-link $node_(s4) $node_(rl) 10Mb 94ms DropTail

# setup RED router
$ns duplex-link $node_(rl) $node_(d} 2Mb 1ms RED
$ns queue-limit $node_(rl) $node_(d) 500
$ns queue-limit $node_(d) $node_(rl) 500

# setup for "nam"
$ns duplex-link-op $node_(rl) $node_(d} queuePos 0.5
$ns duplex-link-op $node_(d) $node_(rl) queuePos 0.5

# set RED router parameters
set reqd [[$ns link $node_(rl) $node_(d}]] queue
$roqd set setbit_ true

# one simulation was done with "set linterm_ 10" (p_max=0.1)
# and one with "set linterm_ 5" (p_max=0.2)
$roqd set linterm_ 10
```
The next script was used for the simulation setup 2 with 4 ECN TCP-Reno flows in Section 4.2.1. One simulation was done with $p_{\text{max}} = 0.1$ ($\texttt{set \ linterm\_10}$) and one with $p_{\text{max}} = 0.2$ ($\texttt{set \ linterm\_5}$).

# ns simulator script for section 4.2.1: 4 New-ECN TCP-Reno connections
# Setup 2: 4 ECN TCP-Reno connections, Link 7: 2Mbps, 1ms
set ns [new Simulator]
# create trace-files and define the trace parameters
proc tracestart {} { 
    set ns [setns]
    set Itrace [open tcp1.tr w] ; set 2trace [open tcp2.tr w]
    set 3trace [open tcp3.tr w] ; set 4trace [open tcp4.tr w]
}
C TCL SCRIPTS FOR THE SIMULATIONS IN SECTION 4

```tcl
$tcp1 tracevar cwnd_ ; $tcp1 tracevar srtt_ ;
$tcp1 tracevar ack_ ; $tcp1 tracevar maxseq_
$tcp2 tracevar cwnd_ ; $tcp2 tracevar srtt_ ;
$tcp2 tracevar ack_ ; $tcp2 tracevar maxseq_
$tcp3 tracevar cwnd_ ; $tcp3 tracevar srtt_ ;
$tcp3 tracevar ack_ ; $tcp3 tracevar maxseq_
$tcp4 tracevar cwnd_ ; $tcp4 tracevar srtt_ ;
$tcp4 tracevar ack_ ; $tcp4 tracevar maxseq_
$tcp1 attach $Itrace ; $tcp2 attach $2trace
$tcp3 attach $3trace ; $tcp4 attach $4trace
}

# create data from the trace-file for a Matlab analysis
proc makeplot number {} {
   end the simulation
   proc finish O {
      global ns $Itrace 2trace 3trace 4trace
      # close trace-files
      $ns flush-trace
      close $Itrace ; close $2trace
      close $3trace ; close $4trace
      # create Matlab data
      makeplot "1" ; makeplot "2"
      makeplot "3" ; makeplot "4"
      exit 0
   }
   makeplot "1"
   makeplot "2"
   makeplot "3"
   makeplot "4"
   exit 0
}

# Create network topology:
# create nodes:
set node_(rl) [$ns node]
set node_(d) [$ns node]
set node_(sl) [$ns node]
set node_(s2) [$ns node]
set node_(s3) [$ns node]
set node_(s4) [$ns node]
# create links
$ns duplex-link $node_(sl) $node_(rl) 10Mb 1ms DropTail
$ns duplex-link $node_(s2) $node_(rl) 10Mb 14ms DropTail
$ns duplex-link $node_(s3) $node_(rl) 10Mb 34ms DropTail
$ns duplex-link $node_(s4) $node_(rl) 10Mb 94ms DropTail

# setup RED router
$ns duplex-link $node_(rl) $node_(d) 2Mb 1ms RED
$ns queue-limit $node_(rl) $node_(d) 500
$ns queue-limit $node_(rl) $node_(d) 500

# setup for "narm"
$ns duplex-link-op $node_(rl) $node_(d) queuePos 0.5
```
```tcl
$ns duplex-link-op $node_(d) $node_(rl) queuePos 0.5

# set RED router parameters
set redq [[$ns link $node_(rl) $node_(d)] queue]
$redq set setbit_true
# one simulation was done with "set linterm_10" (p_max=0.1)
# and one with "set linterm_5" (p_max=0.2)
$redq set linterm_10
$redq set thresh_10
$redq set maxthresh_20
$redq set q_weight_0.002

# set general TCP parameters
Agent/TCP set tcpTick_0.01
Agent/TCP set windowOption_1
Agent/TCPSink set windowOptionS_1

# create TCP connections and set individual parameters
set tcp1 [[$ns create-connection TCP/Reno $node_(s1) TCPSink $node_(d) 0]
$tcp1 set window_128
$tcp1 set ecn_1
$tcp1 set packetSize_512

set tcp2 [[$ns create-connection TCP/Reno $node_(s2) TCPSink $node_(d) 1]
$tcp2 set window_128
$tcp2 set ecn_1
$tcp2 set packetSize_512

set tcp3 [[$ns create-connection TCP/Reno $node_(s3) TCPSink $node_(d) 2]
$tcp3 set window_128
$tcp3 set ecn_1
$tcp3 set packetSize_512

set tcp4 [[$ns create-connection TCP/Reno $node_(s4) TCPSink $node_(d) 3]
$tcp4 set window_128
$tcp4 set ecn_1
$tcp4 set packetSize_512

# attach ftp traffic generators
set ftp1 [[$tcp1 attach-app FTP]]
set ftp2 [[$tcp2 attach-app FTP]]
set ftp3 [[$tcp3 attach-app FTP]]
set ftp4 [[$tcp4 attach-app FTP]]

# schedule events
$ns at 0.0 "tracestart"
$ns at 1.5 "ftp1 start"
$ns at 1.0 "ftp2 start"
$ns at 0.5 "ftp3 start"
$ns at 0.0 "ftp4 start"
$ns at 60.0 "finish"

# start simulation
$ns run
```
C.2 4 New-ECN TCP-Reno vs. 4 ECN TCP-Reno Connections

The next script was used for the simulations with 4 New-ECN/ECN TCP-Reno flows in Section 4.2.2. One simulation was done with 4 New-ECN TCP flows (Agent/TCP set windowOption_8 and Agent/TCPSink set windowOptionS_8) and one was done with 4 ECN TCP flows (Agent/TCP set windowOption_1 and Agent/TCPSink set windowOptionS_1).

```tcl
# ns simulator script for section 4.2.2: 4 New-ECN TCP-Reno vs.
# 4 ECN TCP-Reno connections
set ns [new Simulator]

# create trace-files and define the trace parameters
proc tracestart {} {
    global ns Itrace 2trace 3trace 4trace tcp1 tcp2 tcp3 tcp4
    set Itrace [open tcp1.tr w] ; set 2trace [open tcp2.tr w]
    set 3trace [open tcp3.tr w] ; set 4trace [open tcp4.tr w]
    $tcp1 tracevar cwnd_ ; $tcp1 tracevar srtt_
    $tcp1 tracevar ack_ ; $tcp1 tracevar maxseq_
    $tcp2 tracevar cwnd_ ; $tcp2 tracevar srtt_
    $tcp2 tracevar ack_ ; $tcp2 tracevar maxseq_
    $tcp3 tracevar cwnd_ ; $tcp3 tracevar srtt_
    $tcp3 tracevar ack_ ; $tcp3 tracevar maxseq_
    $tcp4 tracevar cwnd_ ; $tcp4 tracevar srtt_
    $tcp4 tracevar ack_ ; $tcp4 tracevar maxseq_
    $tcp1 attach $Itrace ; $tcp2 attach $2trace
    $tcp3 attach $3trace ; $tcp4 attach $4trace
}

# create data from the trace-file for a Matlab analysis
proc makeplot number {
    # end the simulation
    proc finish {
        global ns Itrace 2trace 3trace 4trace
        # close trace-files
        $ns flush-trace
        close $Itrace ; close $2trace
        close $3trace ; close $4trace
        # create Matlab data
        makeplot "1" ; makeplot "2"
        makeplot "3" ; makeplot "4"
        exit 0
    }
}

# Create network topology:
# create nodes:
```

```
# attach ftp traffic generators
set node_(r1) [$ns node]
set node_(d) [$ns node]
set node_(s1) [$ns node]
set node_(s2) [$ns node]
set node_(s3) [$ns node]
set node_(s4) [$ns node]

# create links
$ns duplex-link $node_(s1) $node_(r1) 10Mb 1ms DropTail
$ns duplex-link $node_(s2) $node_(r1) 10Mb 14ms DropTail
$ns duplex-link $node_(s3) $node_(r1) 10Mb 34ms DropTail
$ns duplex-link $node_(s4) $node_(r1) 10Mb 94ms DropTail

# setup RED router
$ns duplex-link $node_(rl) $node_(d) 10Mb 1ms RED
$ns queue-limit $node_(rl) $node_(d) 500
$ns queue-limit $node_(d) $node_(rl) 500

# setup for "nam"
$ns duplex-link-op $node_(rl) $node_(d) queuePos 0.5
$ns duplex-link-op $node_(d) $node_(rl) queuePos 0.5

# set RED router parameters
set reqq [[$ns link $node_(rl) $node_(d)] queue]
$reqq set setbit_ true
$reqq set linterm_ 10
$reqq set thresh_ 10
$reqq set maxthresh_ 20
$reqq set q_weight_ 0.002

# set general TCP parameters
Agent/TCP set tcpTick_ 0.01
# one simulation was done with "set windowOption_ 8" and "set windowOptionS_ 8" (New-ECN)
# one was done with "set windowOption_ 1" and "set windowOptionS_ 1" (ECN)
Agent/TCP set windowOption_ 1
Agent/TCPSink set windowOptionS_ 1

# create TCP connections and set individual parameters
set tcp1 [[$ns create-connection TCP/Reno $node_(s1) TCPSink $node_(d) 0]
$tcp1 set window_ 128
$tcp1 set ecn_ 1
$tcp1 set packetSize_ 512

set tcp2 [[$ns create-connection TCP/Reno $node_(s2) TCPSink $node_(d) 1]
$tcp2 set window_ 128
$tcp2 set ecn_ 1
$tcp2 set packetSize_ 512

set tcp3 [[$ns create-connection TCP/Reno $node_(s3) TCPSink $node_(d) 2]
$tcp3 set window_ 128
$tcp3 set ecn_ 1
$tcp3 set packetSize_ 512

set tcp4 [[$ns create-connection TCP/Reno $node_(s4) TCPSink $node_(d) 3]
$tcp4 set window_ 128
$tcp4 set ecn_ 1
$tcp4 set packetSize_ 512

# attach ftp traffic
set ftp1 [[$tcp1 attach-app FTP] ; set ftp2 [[$tcp2 attach-app FTP]
set ftp3 [[$tcp3 attach-app FTP] ; set ftp4 [[$tcp4 attach-app FTP]

# schedule events
C.3 Many New-ECN TCP-Reno Connections

The next script was used for the simulation in Section 4.2.3:

```tcl
# ns simulator script for section 4.2.3: many New-ECN TCP-Reno connections
set ns [new Simulator]
# create trace-files and define the trace parameters
proc tracestart {} {
    global ns 1trace 2trace 3trace 4trace 5trace 6trace 7trace 8trace tcp1 tcp2 tcp3 tcp4 tcp5 tcp6 tcp7 tcp8 tcp9 tcp10 tcp11 tcp12 tcp13 tcp14 tcp15 tcp16
    set 1trace [open tcp1.tr w]
    set 2trace [open tcp2.tr w]
    set 3trace [open tcp3.tr w]
    set 4trace [open tcp4.tr w]
    set 5trace [open tcp5.tr w]
    set 6trace [open tcp6.tr w]
    set 7trace [open tcp7.tr w]
    set 8trace [open tcp8.tr w]
    set 9trace [open tcp9.tr w]
    set 10trace [open tcp10.tr w]
    set 11trace [open tcp11.tr w]
    set 12trace [open tcp12.tr w]
    set 13trace [open tcp13.tr w]
    set 14trace [open tcp14.tr w]
    set 15trace [open tcp15.tr w]
    set 16trace [open tcp16.tr w]
    $tcp1 tracevar cwnd
    $tcp2 tracevar cwnd
    $tcp3 tracevar cwnd
    $tcp4 tracevar cwnd
    $tcp5 tracevar cwnd
    $tcp6 tracevar cwnd
    $tcp7 tracevar cwnd
    $tcp8 tracevar cwnd
    $tcp9 tracevar cwnd
    $tcp10 tracevar cwnd
    $tcp11 tracevar cwnd
    $tcp12 tracevar cwnd
    $tcp13 tracevar cwnd
    $tcp14 tracevar cwnd
    $tcp15 tracevar cwnd
    $tcp16 tracevar cwnd
    $tcp1 attach $1trace
    $tcp2 attach $2trace
    $tcp3 attach $3trace
    $tcp4 attach $4trace
    $tcp5 attach $5trace
    $tcp6 attach $6trace
    $tcp7 attach $7trace
    $tcp8 attach $8trace
    $tcp9 attach $9trace
    $tcp10 attach $10trace
    $tcp11 attach $11trace
    $tcp12 attach $12trace
    $tcp13 attach $13trace
    $tcp14 attach $14trace
    $tcp15 attach $15trace
    $tcp16 attach $16trace
}
# create data from the trace-file for a Matlab analysis
proc makeplot number {
    $ns at 0.0 "tracestart"
    $ns at 1.5 "$ftp1 start"
    $ns at 1.0 "$ftp2 start"
    $ns at 0.5 "$ftp3 start"
    $ns at 0.0 "$ftp4 start"
    $ns at 60.0 "finish"
    # start simulation
    $ns run
}
```
```tcl
C TCL SCRIPTS FOR THE SIMULATIONS IN SECTION 4

...}

$ end the simulation
proc finish {} {
    global ns 1trace 2trace 3trace 4trace 5trace 6trace 7trace 8trace 9trace \
         10trace 11trace 12trace 13trace 14trace 15trace 16trace
    # close trace-files
    $ns flush-trace
    close $1trace ; close $2trace ; close $3trace ; close $4trace
    close $5trace ; close $6trace ; close $7trace ; close $8trace
    close $9trace ; close $10trace ; close $11trace ; close $12trace
    close $13trace ; close $14trace ; close $15trace ; close $16trace

    # create Matlab data
    makeplot "1" ; makeplot "2" ; makeplot "3" ; makeplot "4"
    makeplot "5" ; makeplot "6" ; makeplot "7" ; makeplot "8"
    makeplot "9" ; makeplot "10" ; makeplot "11" ; makeplot "12"
    makeplot "13" ; makeplot "14" ; makeplot "15" ; makeplot "16"
    exit 0
}

# Create network topology:
# create nodes:
set node_(s1) [\$ns node]
set node_(s2) [\$ns node]
set node_(s3) [\$ns node]
set node_(s4) [\$ns node]
set node_(s5) [\$ns node]
set node_(s6) [\$ns node]
set node_(s7) [\$ns node]
set node_(s8) [\$ns node]

# create links
$ns duplex-link $node_(s1) $node_(s2) 10Mb 44ms DropTail
$ns duplex-link $node_(s2) $node_(s3) 10Mb 34ms DropTail
$ns duplex-link $node_(s3) $node_(s4) 10Mb 28ms DropTail
$ns duplex-link $node_(s4) $node_(s5) 10Mb 24ms DropTail
$ns duplex-link $node_(s5) $node_(s6) 10Mb 14ms DropTail
$ns duplex-link $node_(s6) $node_(s7) 10Mb 9ms DropTail
$ns duplex-link $node_(s7) $node_(s8) 10Mb 4ms DropTail

# setup RED router
$ns duplex-link $node_(s1) $node_(s2) queuePos 0.5
$ns queue-limit $node_(s1) $node_(s2) 500
$ns queue-limit $node_(s2) $node_(s1) 500

# setup for "nam"
$ns duplex-link-op $node_(s1) $node_(s2) queuePos 0.5
$ns duplex-link-op $node_(s2) $node_(s1) queuePos 0.5

# set RED router parameters
set redq [[\$ns link $node_(s1) $node_(s2)]] queue
$redq set setbit_ true
$redq set linterm_ 5
$redq set thresh_ 10
$redq set maxthresh_ 20
$redq set q_weight_ 0.002

# set general TCP parameters
```
Agent/TCP set tcpTick 0.01
Agent/TCP set windowOption 8
Agent/TCP set windowOption2 8

# create TCP connections and set individual parameters
set tcp1 [$ns create-connection TCP/Reno $node_(s1) TCPSink $node_(d) 0]
$tcp1 set window 128
$tcp1 set ecn 1
$tcp1 set packetSize 512

set tcp2 [$ns create-connection TCP/Reno $node_(s2) TCPSink $node_(d) 1]
$tcp2 set window 128
$tcp2 set ecn 1
$tcp2 set packetSize 512

set tcp3 [$ns create-connection TCP/Reno $node_(s3) TCPSink $node_(d) 2]
$tcp3 set window 128
$tcp3 set ecn 1
$tcp3 set packetSize 512

set tcp4 [$ns create-connection TCP/Reno $node_(s4) TCPSink $node_(d) 3]
$tcp4 set window 128
$tcp4 set ecn 1
$tcp4 set packetSize 512

set tcp5 [$ns create-connection TCP/Reno $node_(s5) TCPSink $node_(d) 4]
$tcp5 set window 128
$tcp5 set ecn 1
$tcp5 set packetSize 512

set tcp6 [$ns create-connection TCP/Reno $node_(s6) TCPSink $node_(d) 5]
$tcp6 set window 128
$tcp6 set ecn 1
$tcp6 set packetSize 512

set tcp7 [$ns create-connection TCP/Reno $node_(s7) TCPSink $node_(d) 6]
$tcp7 set window 128
$tcp7 set ecn 1
$tcp7 set packetSize 512

set tcp8 [$ns create-connection TCP/Reno $node_(s8) TCPSink $node_(d) 7]
$tcp8 set window 128
$tcp8 set ecn 1
$tcp8 set packetSize 512

set tcp9 [$ns create-connection TCP/Reno $node_(s9) TCPSink $node_(d) 8]
$tcp9 set window 128
$tcp9 set ecn 1
$tcp9 set packetSize 512

set tcp10 [$ns create-connection TCP/Reno $node_(s10) TCPSink $node_(d) 10]
$tcp10 set window 128
$tcp10 set ecn 1
$tcp10 set packetSize 512

set tcp11 [$ns create-connection TCP/Reno $node_(s11) TCPSink $node_(d) 11]
$tcp11 set window 128
$tcp11 set ecn 1
$tcp11 set packetSize 512

set tcp12 [$ns create-connection TCP/Reno $node_(s12) TCPSink $node_(d) 12]
$tcp12 set window 128
$tcp12 set ecn 1
C TCL SCRIPTS FOR THE SIMULATIONS IN SECTION 4

$tcpl2 set packetSize 512

set tcpl3 [$ns create-connection TCP/Reno $node_(s3) TCPSink $node_(d) 12]
$tcpl3 set window 128
$tcpl3 set ecn 1
$tcpl3 set packetSize 512

set tcpl4 [$ns create-connection TCP/Reno $node_(s4) TCPSink $node_(d) 13]
$tcpl4 set window 128
$tcpl4 set ecn 1
$tcpl4 set packetSize 512

set tcpl5 [$ns create-connection TCP/Reno $node_(s5) TCPSink $node_(d) 14]
$tcpl5 set window 128
$tcpl5 set ecn 1
$tcpl5 set packetSize 512

set tcpl6 [$ns create-connection TCP/Reno $node_(s6) TCPSink $node_(d) 15]
$tcpl6 set window 128
$tcpl6 set ecn 1
$tcpl6 set packetSize 512

# attach ftp traffic generators
set ftp1 [$tcpl attach-app FTP]
set ftp2 [$tcp2 attach-app FTP]
set ftp3 [$tcp3 attach-app FTP]
set ftp4 [$tcp4 attach-app FTP]
set ftp5 [$tcp5 attach-app FTP]
set ftp6 [$tcp6 attach-app FTP]
set ftp7 [$tcp7 attach-app FTP]
set ftp8 [$tcp8 attach-app FTP]
set ftp9 [$tcp9 attach-app FTP]
set ftp10 [$tcp10 attach-app FTP]
set ftp11 [$tcp11 attach-app FTP]
set ftp12 [$tcp12 attach-app FTP]
set ftp13 [$tcp13 attach-app FTP]
set ftp14 [$tcp14 attach-app FTP]
set ftp15 [$tcp15 attach-app FTP]
set ftp16 [$tcp16 attach-app FTP]

# schedule events
$nsl at 0.0 "tracestart"
$nsl at 0.0 "$ftp1 start"
$nsl at 0.25 "$ftp2 start"
$nsl at 0.5 "$ftp3 start"
$nsl at 0.75 "$ftp4 start"
$nsl at 2.0 "$ftp5 start"
$nsl at 2.25 "$ftp6 start"
$nsl at 2.5 "$ftp7 start"
$nsl at 2.75 "$ftp8 start"
$nsl at 4.0 "$ftp9 start"
$nsl at 4.25 "$ftp10 start"
$nsl at 4.50 "$ftp11 start"
$nsl at 4.75 "$ftp12 start"
$nsl at 6.0 "$ftp13 start"
$nsl at 6.25 "$ftp14 start"
$nsl at 6.5 "$ftp15 start"
$nsl at 6.75 "$ftp16 start"
$nsl at 80.0 "finish"

# start simulation
$nsl run
C.4 One New-ECN TCP-Reno vs. one ECN/non-ECN TCP-Reno Connection

The next script was used for the simulation with one New-ECN TCP-Reno flow in Section 4.3:

```tcl
# ns simulator script for section 4.3.1: One New-ECN TCP-Reno vs.
# one ECN/non-ECN TCP-Reno connections
set ns [new Simulator]

# create trace-files and define the trace parameters
proc tracestart {} {
    global ns ltrace tcpl
    set ltrace [open tcpl.tr w]
    $tcpl tracevar cvnd_ ; $tcpl tracevar srtt_ ; $tcpl tracevar ack_
    $tcpl attach $ltrace
}

# create data from the trace-file for a Matlab analysis
proc makeplot number {
    ...
}

# count packet drops
proc CountDrops {} {
    set awkCode {
        if ($1 == "d") {
            print $1, $2 >> "drops";
        }
    }
    exec awk $awkCode queue1.tr
}

# end the simulation
proc finish {} {
    global ns ltrace qtrace
    # close trace-files
    $ns flush-trace
    close $ltrace
    close qtrace
    # create Matlab data
    CountDrops
    makeplot "1"
    exit 0
}

# create queue trace
set qtrace [open queue1.tr w]
$ns trace-all $qtrace

# Create network topology:
# create nodes:
set node_r1 [$ns node]
set node_d [$ns node]
```

C TCL SCRIPTS FOR THE SIMULATIONS IN SECTION 4 78
The next script was used for the simulation with one ECN/non-ECN TCP-Reno flow in Section 4.3. One simulation was done with ECN TCP-Reno ($\texttt{tcp1 set ecn. 1}$) and one with TCP-Reno ($\texttt{tcp1 set ecn. 0}$).
# create data from the trace-file for a Matlab analysis
proc makeplot number {
    ... 
}

# end the simulation
proc finish () {
    global ns itrace
    # close trace-files
    puts "in finish"
    $ns flush-trace
    close $itrace
    # create Matlab data
    makeplot "1"
    exit 0
}

# Create network topology:
# create nodes:
set node_{ri} [$ns node]
set node_{d} [$ns node]
set node_{s1} [$ns node]

# create links
$ns duplex-link $node_{s1} $node_{ri} 10Mb 14ms DropTail

# setup RED router
$ns duplex-link $node_{ri} $node_{d} 5Mb 1ms RED
$ns queue-limit $node_{ri} $node_{d} 500
$ns queue-limit $node_{d} $node_{ri} 500

# setup for "nam"
$ns duplex-link-op $node_{ri} $node_{d} queuePos 0.5
$ns duplex-link-op $node_{d} $node_{ri} queuePos 0.5

# set RED router parameters
set roi $[[$ns link $node_{ri} $node_{d}]] queue
$roi set setbit_ true
$roi set linterm_ 10
$roi set thresh_ 10
$roi set maxthresh_ 20
$roi set q_weight_ 0.002

# set general TCP parameters
Agent/TCP set tcpTick_ 0.01

# create TCP connections and set individual parameters
set tcpi $[[$ns create-connection TCP/Reno $node_{s1} TCPSink $node_{d} 0]]
$tcpi set window_ 128
# one simulation was done with "set ecn_ 1" (TCP-Reno with ECN)
# and one with "set ecn_ 0" (TCP-Reno without ECN)
$tcpi set ecn_ 1
$tcpi set windowOption_ 1
$tcpi set windowOptionS_ 1
$tcpi set packetSize_ 512

# attach ftp traffic generators
set ftpi [$tcpi attach-app FTP]

# schedule events
$ns at 0.0 "tracestart"
C TCL SCRIPTS FOR THE SIMULATIONS IN SECTION 4

$ns at 0.0 "$ftp1 start"
$ns at 60.0 "finish"

# start simulation
$ns run

C.5 TCP Friendliness of New-ECN

The next script was used for the simulation with three New-ECN flows and three TCP-Reno flows in Section 4.3.2.

# ns simulator script for section 4.3.2: TCP Friendliness of New-ECN
set ns [new Simulator]

# create trace-files and define the trace parameters
proc tracestart {} {
    global ns Itrace 2trace 3trace 4trace Strace Strace Gtrace tcpl tcp2 tcp3 tcp4 tcp5 tcp6
    set Itrace [open tcpl.tr v]
    set 3trace [open tcp3.tr v]
    set Strace [open tcp5.tr v]
    set 2trace- [open tcp2.tr w]
    set 4trace [open tcp4.tr v]
    set Strace [open tcp6.tr v]
    $tcpl tracevar cwnd_
    $tcp2 tracevar cwnd_
    $tcp3 tracevar cwnd_
    $tcp4 tracevar cwnd_
    $tcp5 tracevar cwnd_
    $tcp6 tracevar cwnd_
    $tcpl attach $ltrace
    $tcp2 attach $2trace
    $tcp3 attach $3trace
    $tcp4 attach $4trace
    $tcp5 attach $5trace
    $tcp6 attach $6trace

    $tcp1 tracevar srtt_
    $tcp2 tracevar srtt_
    $tcp3 tracevar srtt_
    $tcp4 tracevar srtt_
    $tcp5 tracevar srtt_
    $tcp6 tracevar srtt_
    $tcp2 attach $2trace
    $tcp5 attach $5trace

    ft create data from the trace-file for a Matlab analysis
    proc makeplot number {
        ...
    }

    # end the simulation
    proc finish {} {
        global ns Itrace 2trace 3trace 4trace Strace Strace Gtrace tcpl tcp2 tcp3 tcp4 tcp5 tcp6
        # close trace-files
        $ns flush-trace
        close $1trace ; close $2trace ; close $3trace
        close $4trace ; close $5trace ; close $6trace

        # create Matlab data
        makeplot "1" ; makeplot "2" ; makeplot "3"
        makeplot "4" ; makeplot "5" ; makeplot "6"
        exit 0
    }

    # Create network topology:
    # create nodes:
    set node_(r1) [$ns node]
    set node_(d) [$ns node]
set node_s1 [$ns node]
set node_s2 [$ns node]
set node_s3 [$ns node]
set node_s4 [$ns node]
set node_s5 [$ns node]
set node_s6 [$ns node]

# create links
$ns duplex-link $node_s1 $node_s1 $node_r1 10Mb 1ms DropTail
$ns duplex-link $node_s2 $node_s2 $node_r1 10Mb 14ms DropTail
$ns duplex-link $node_s3 $node_s3 $node_r1 10Mb 34ms DropTail
$ns duplex-link $node_s4 $node_s4 $node_r1 10Mb 1ms DropTail
$ns duplex-link $node_s5 $node_s5 $node_r1 10Mb 14ms DropTail
$ns duplex-link $node_s6 $node_s6 $node_r1 10Mb 34ms DropTail

# setup RED router
$ns duplex-link $node_r1 $node_r1 $node_d 10Mb 1ms RED
$ns queue-limit $node_r1 $node_d 500
$ns queue-limit $node_d $node_r1 500

# setup for "nam"
$ns duplex-link-op $node_r1 $node_d queuePos 0.5
$ns duplex-link-op $node_d $node_r1 queuePos 0.5

# set RED router parameters
set redq [$ns link $node_r1 $node_d] queue
$redq set setbit_true
$redq set linterm_10
$redq set thresh_10
$redq set maxthresh_20
$redq set q_weight_0.002

# set general TCP parameters
Agent/TCP set tcpTick_0.01

# create TCP connections and set individual parameters
set tcp1 [$ns create-connection TCP/Reno $node_s1 TCPSink $node_d 0]
tcp1 set window_128
tcp1 setecn_1
tcp1 set windowOption_8
tcp1 set windowOptionS_8
$tcp1 set packetSize_512

set tcp2 [$ns create-connection TCP/Reno $node_s2 TCPSink $node_d 1]
tcp2 set window_128
tcp2 setecn_1
tcp2 set windowOption_8
tcp2 set windowOptionS_8
$tcp2 set packetSize_512

set tcp3 [$ns create-connection TCP/Reno $node_s3 TCPSink $node_d 2]
tcp3 set window_128
tcp3 setecn_1
tcp3 set windowOption_8
tcp3 set windowOptionS_8
$tcp3 set packetSize_512

set tcp4 [$ns create-connection TCP/Reno $node_s4 TCPSink $node_d 3]
tcp4 set window_128
tcp4 setecn_0
$tcp4 set windowOption_1
$tcp4 set windowOptionS_1
$tcp4 set packetSize_512
C TCL SCRIPTS FOR THE SIMULATIONS IN SECTION 4

The next script was used for the simulation with six TCP-Reno flows in Section 4.3.2.

# ns simulator script for section 4.3.2: TCP Friendliness of New-ECN
set ns [new Simulator]

# create trace-files and define the trace parameters
proc tracestart {} {
    global ns ltrace 2trace 3trace 4trace 5trace 6trace tcpl tcp2 tcp3 tcp4 tcp5 tcp6
    set ltrace [open tcp1.tr u] ; set 2trace [open tcp2.tr w]
    set 3trace [open tcp3.tr w] ; set 4trace [open tcp4.tr w]
    set 5trace [open tcp5.tr w] ; set 6trace [open tcp6.tr w]
    $tcpl tracevar cwnd_ ; $tcp2 tracevar cwnd_ ; $tcp3 tracevar cwnd_ ; $tcp4 tracevar cwnd_ ; $tcp5 tracevar cwnd_ ; $tcp6 tracevar cwnd_
    $tcpl tracevar srtt_ ; $tcp2 tracevar srtt_ ; $tcp3 tracevar srtt_ ; $tcp4 tracevar srtt_ ; $tcp5 tracevar srtt_ ; $tcp6 tracevar srtt_
    $tcpl tracevar ack_ ; $tcp2 tracevar ack_ ; $tcp3 tracevar ack_ ; $tcp4 tracevar ack_ ; $tcp5 tracevar ack_ ; $tcp6 tracevar ack_
    $tcpl attach $ltrace ; $tcp2 attach $2trace ; $tcp3 attach $3trace
    $tcp4 attach $4trace ; $tcp5 attach $5trace ; $tcp6 attach $6trace
}

# create data from the trace-file for a Matlab analysis
proc makeplot number {
    ...
}

# end the simulation
proc finish {} {
    ...
}
global ns 1trace 2trace 3trace 4trace 5trace 6trace

# close trace-files
$ns flush-trace
close $1trace ; close $2trace ; close $3trace
close $4trace ; close $5trace ; close $6trace

# create Matlab data
makeplot "1" ; makeplot "2" ; makeplot "3"
makeplot "4" ; makeplot "5" ; makeplot "6"
exit 0

# Create network topology:
# create nodes:
set node_(r1) [$ns node]
set node_(d) [$ns node]
set node_(s1) [$ns node]
set node_(s2) [$ns node]
set node_(s3) [$ns node]
set node_(s4) [$ns node]
set node_(s5) [$ns node]
set node_(s6) [$ns node]

# create links
$ns duplex-link $node_(s1) $node_(r1) 10Mb 1ms DropTail
$ns duplex-link $node_(s2) $node_(r1) 10Mb 14ms DropTail
$ns duplex-link $node_(s3) $node_(r1) 10Mb 34ms DropTail
$ns duplex-link $node_(s4) $node_(r1) 10Mb 1ms DropTail
$ns duplex-link $node_(s5) $node_(r1) 10Mb 14ms DropTail
$ns duplex-link $node_(s6) $node_(r1) 10Mb 34ms DropTail

# setup RED router
$ns duplex-link $node_(r1) $node_(d) 10Mb 1ms RED
$ns queue-limit $node_(r1) $node_(d) 500
$ns queue-limit $node_(d) $node_(r1) 500

# setup for "nam"
$ns duplex-link-op $node_(r1) $node_(d) queuePos 0.5
$ns duplex-link-op $node_(d) $node_(r1) queuePos 0.5

# set RED router parameters
set rodq [[($ns link $node_(r1) $node_(d)) queue]
$rodq set setbit_ true
$rodq set linterm_ 10
$rodq set thresh_ 10
$rodq set maxthresh_ 20
$rodq set q_weight_ 0.002

# set general TCP parameters
Agent/TCP set tcpTick_ 0.01

# create TCP connections and set individual parameters
set tcp1 [$ns create-connection TCP/Reno $node_(s1) TCPSink $node_(d) 0]
$tcp1 set window_ 128
$tcp1 set ecn_ 0
$tcp1 set windowOption_ 1
$tcp1 set windowOptionS_ 1
$tcp1 set packetSize_ 512

set tcp2 [$ns create-connection TCP/Reno $node_(s2) TCPSink $node_(d) 1]
$tcp2 set window_ 128
$tcp2 set ecn_ 0
$tcp2 set windowOption_ 1
C TCL SCRIPTS FOR THE SIMULATIONS IN SECTION 4

```tcl
$tcp2 set windOptionS_1
$tcp2 set packetSize_512

set tcp3 [ns create-connection TCP/Reno $node_(s3) TCPSink $node_(d) 2]
$tcp3 set window_128 $tcp3 set ecn_0 $tcp3 set windowOption_1 $tcp3 set windowOptionS_1 $tcp3 set packetSize_512

set tcp4 [ns create-connection TCP/Reno $node_(s4) TCPSink $node_(d) 3]
$tcp4 set window_128 $tcp4 set ecn_0 $tcp4 set windowOption_1 $tcp4 set windowOptionS_1 $tcp4 set packetSize_512

set tcp5 [ns create-connection TCP/Reno $node_(s5) TCPSink $node_(d) 4]
$tcp5 set window_128 $tcp5 set ecn_0 $tcp5 set windowOption_1 $tcp5 set windowOptionS_1 $tcp5 set packetSize_512

set tcp6 [ns create-connection TCP/Reno $node_(s6) TCPSink $node_(d) 5]
$tcp6 set window_128 $tcp6 set ecn_0 $tcp6 set windowOption_1 $tcp6 set windowOptionS_1 $tcp6 set packetSize_512

# attach ftp traffic generators
set ftp1 [$tcp1 attach-app FTP] ; set ftp2 [$tcp2 attach-app FTP]
set ftp3 [$tcp3 attach-app FTP] ; set ftp4 [$tcp4 attach-app FTP]
set ftp5 [$tcp5 attach-app FTP] ; set ftp6 [$tcp6 attach-app FTP]

# schedule events
$ns at 0.0 "tracestart"
$ns at 0.0 "$ftp1 start"
$ns at 0.0 "$ftp2 start"
$ns at 0.0 "$ftp3 start"
$ns at 0.0 "$ftp4 start"
$ns at 0.0 "$ftp5 start"
$ns at 0.0 "$ftp6 start"
$ns at 60.0 "finish"

# start simulation
$ns run
```

C.6 ECN-TCP Friendliness of New-ECN

The next script was used for the simulation with three New-ECN flows and three ECN TCP-Reno flows in Section 4.3.3.

```tcl
# ns simulator script for section 4.3.3: ECN-TCP Friendliness of New-ECN
set ns [new Simulator]
```
# create trace-files and define the trace parameters
proc tracestart {} {
    global ns Itrace 2trace 3trace 4trace 5trace 6trace tcp1 tcp2 tcp3 tcp4 tcp5 tcp6

    set Itrace [open tcp1.tr w]; set 2trace [open tcp2.tr w]
    set 3trace [open tcp3.tr w]; set 4trace [open tcp4.tr w]
    set 5trace [open tcp5.tr w]; set 6trace [open tcp6.tr w]

    $tcp1 tracevar cwnd_; $tcp1 tracevar srtt_; $tcp1 tracevar ack_
    $tcp2 tracevar cwnd_; $tcp2 tracevar srtt_; $tcp2 tracevar ack_
    $tcp3 tracevar cwnd_; $tcp3 tracevar srtt_; $tcp3 tracevar ack_
    $tcp4 tracevar cwnd_; $tcp4 tracevar srtt_; $tcp4 tracevar ack_
    $tcp5 tracevar cwnd_; $tcp5 tracevar srtt_; $tcp5 tracevar ack_
    $tcp6 tracevar cwnd_; $tcp6 tracevar srtt_; $tcp6 tracevar ack_

    $tcp1 attach $Itrace; $tcp2 attach $2trace; $tcp3 attach $3trace
    $tcp4 attach $4trace; $tcp5 attach $5trace; $tcp6 attach $6trace

    # close trace-files
    $ns flush-trace
    close $Itrace; close $2trace; close $3trace
    close $4trace; close $5trace; close $6trace

    # create Matlab data
    makeplot "1" ; makeplot "2" ; makeplot "3"
    makeplot "4" ; makeplot "5" ; makeplot "6"
    exit 0
}

# create data for the trace-file for a Matlab analysis
proc makeplot number {
    ...
}

# end the simulation
proc finish {} {
    global ns Itrace 2trace 3trace 4trace 5trace 6trace

    # close trace-files
    $ns flush-trace
    close $Itrace; close $2trace; close $3trace
    close $4trace; close $5trace; close $6trace

    # create Matlab data
    makeplot "1" ; makeplot "2" ; makeplot "3"
    makeplot "4" ; makeplot "5" ; makeplot "6"
    exit 0
}

# Create network topology:
# create nodes
set node_(rl) [$ns node]
set node.(d) [$ns node]
set node.(si) [$ns node]
set node.Cs2) [$ns node]
set node.CsS) [$ns node]
set node.(s4) [$ns node]
set node.(sS) [$ns node]
set node.(sS) [$ns node]

# create links
$ns duplex-link $node.(si) $node.(rl) 10Mb 1ms DropTail
$ns duplex-link $node.(s2) $node.(rl) 10Mb 14ms DropTail
$ns duplex-link $node.(s3) $node.(rl) 10Mb 34ms DropTail
$ns duplex-link $node.(s4) $node.(rl) 10Mb 1ms DropTail
$ns duplex-link $node.(s5) $node.(rl) 10Mb 14ms DropTail
$ns duplex-link $node.(s6) $node.(rl) 10Mb 34ms DropTail

# setup RED router
$ns duplex-link $node.(rl) $node.(d) 10Mb 1ms RED
$ns queue-limit $node.(rl) $node.(d) 600
$ns queue-limit $node.(d) $node.(rl) 600

# setup for "nam"
C  TCL SCRIPTS FOR THE SIMULATIONS IN SECTION 4 87

```tcl
# $ns duplex-link-op $node_(rl) $node_(d) queuePos 0.5
# $ns duplex-link-op $node_(d) $node_(rl) queuePos 0.5

# set RED router parameters
set redq [{[ns link $node_(rl) $node_(d)] queue]}
set bit_ true
$redq set linterm_ 10
$redq set thresh_ 10
$redq set maxthresh_ 20
$redq set q_weight_ 0.002

# set general TCP parameters
Agent/TCP set tcpTick_ 0.01

# create TCP connections and set individual parameters
set tcp1 [ns create-connection TCP/Reno $node_(s1) TCPSink $node_(d) 0]
$tcp1 set window_ 128
$tcp1 set ecn_ 1
$tcp1 set windowOption_ 8
$tcp1 set windowOptionS_ 8
$tcp1 set packetSize_ 512

set tcp2 [ns create-connection TCP/Reno $node_(s2) TCPSink $node_(d) 1]
$tcp2 set window_ 128
$tcp2 set ecn_ 1
$tcp2 set windowOption_ 8
$tcp2 set windowOptionS_ 8
$tcp2 set packetSize_ 512

set tcp3 [ns create-connection TCP/Reno $node_(s3) TCPSink $node_(d) 2]
$tcp3 set window_ 128
$tcp3 set ecn_ 1
$tcp3 set windowOption_ 8
$tcp3 set windowOptionS_ 8
$tcp3 set packetSize_ 512

set tcp4 [ns create-connection TCP/Reno $node_(s4) TCPSink $node_(d) 3]
$tcp4 set window_ 128
$tcp4 set ecn_ 1
$tcp4 set windowOption_ 8
$tcp4 set windowOptionS_ 8
$tcp4 set packetSize_ 512

set tcp5 [ns create-connection TCP/Reno $node_(s5) TCPSink $node_(d) 4]
$tcp5 set window_ 128
$tcp5 set ecn_ 1
$tcp5 set windowOption_ 8
$tcp5 set windowOptionS_ 8
$tcp5 set packetSize_ 512

set tcp6 [ns create-connection TCP/Reno $node_(s6) TCPSink $node_(d) 5]
$tcp6 set window_ 128
$tcp6 set ecn_ 1
$tcp6 set windowOption_ 8
$tcp6 set windowOptionS_ 8
$tcp6 set packetSize_ 512

# attach ftp traffic generators
set ftp1 [tcp1 attach-app FTP] ; set ftp2 [tcp2 attach-app FTP]
set ftp3 [tcp3 attach-app FTP] ; set ftp4 [tcp4 attach-app FTP]
set ftp5 [tcp5 attach-app FTP] ; set ftp6 [tcp6 attach-app FTP]

# schedule events
$ns at 0.0 "tracestart"
```
C TCL SCRIPTS FOR THE SIMULATIONS IN SECTION 4

$ns at 0.0 "$ftp1 start"
$ns at 0.0 "$ftp2 start"
$ns at 0.0 "$ftp3 start"
$ns at 0.0 "$ftp4 start"
$ns at 0.0 "$ftp5 start"
$ns at 0.0 "$ftp6 start"
$ns at 60.0 "finish"

# start simulation
$ns run

The next script was used for the simulation with six ECN TCP-Reno flows in Section 4.3.3.

# ns simulator script for section 4.3.3: ECN-TCP Friendliness of New-ECN
set ns [new Simulator]

# create trace-files and define the trace parameters
proc tracestart {} {
    global ns 1trace 2trace 3trace 4trace 5trace 6trace tcp1 tcp2 tcp3 tcp4 tcp5 tcp6
    set 1trace [open tcp1.tr w] ; set 2trace [open tcp2.tr w]
    set 3trace [open tcp3.tr w] ; set 4trace [open tcp4.tr w]
    set 5trace [open tcp5.tr w] ; set 6trace [open tcp6.tr w]

    $tcp1 tracevar cwnd_ ; $tcp2 tracevar cwnd_ ; $tcp3 tracevar cwnd_ ; $tcp4 tracevar cwnd_ ; $tcp5 tracevar cwnd_ ; $tcp6 tracevar cwnd_
    $tcp2 tracevar srtt_ ; $tcp3 tracevar srtt_ ; $tcp4 tracevar srtt_ ; $tcp5 tracevar srtt_ ; $tcp6 tracevar srtt_
    $tcp3 attach $1trace ; $tcp4 attach $2trace ; $tcp5 attach $3trace ; $tcp6 attach $4trace
    $tcp4 attach $4trace ; $tcp5 attach $5trace ; $tcp6 attach $6trace
}

# create data from the trace-file for a Matlab analysis
proc makeplot number {
    $tcp1 tracevar ack_; $tcp2 tracevar ack_; $tcp3 tracevar ack_; $tcp4 tracevar ack_; $tcp5 tracevar ack_; $tcp6 tracevar ack_
    $tcp2 attach $3trace ; $tcp3 attach $4trace ; $tcp5 attach $5trace ; $tcp6 attach $6trace
}

# end the simulation
proc finish {} {
    global ns 1trace 2trace 3trace 4trace 5trace 6trace

    # close trace-files
    $ns flush-trace
    close $1trace ; close $2trace ; close $3trace ;
    close $4trace ; close $5trace ; close $6trace

    # create Matlab data
    makeplot "1" ; makeplot "2" ; makeplot "3"
    makeplot "4" ; makeplot "5" ; makeplot "6"
    exit 0
}

# Create network topology:
# create nodes:
set node_01 [list] $ns node
set node_02 [list] $ns node
set node_03 [list] $ns node
set node_04 [list] $ns node
set node_05 [list] $ns node
set node_06 [list] $ns node

# Create network topology:
# create nodes:
set node_01 [list] $ns node
set node_02 [list] $ns node
set node_03 [list] $ns node
set node_04 [list] $ns node
set node_05 [list] $ns node
set node_06 [list] $ns node
C TCL SCRIPTS FOR THE SIMULATIONS IN SECTION 4

set node_(s2) [$ns node]
set node_(s3) [$ns node]
set node_(s4) [$ns node]
set node_(s5) [$ns node]
set node_(s6) [$ns node]

# create links
$ns duplex-link $node_(s1) $node_(s1) 10Mb 1ms DropTail
$ns duplex-link $node_(s2) $node_(s2) 10Mb 13ms DropTail
$ns duplex-link $node_(s3) $node_(s3) 10Mb 34ms DropTail
$ns duplex-link $node_(s4) $node_(s4) 10Mb 1ms DropTail
$ns duplex-link $node_(s5) $node_(s5) 10Mb 13ms DropTail
$ns duplex-link $node_(s6) $node_(s6) 10Mb 34ms DropTail

# setup RED router
$ns duplex-link $node_(s1) $node_(s1) 10Mb 1ms RED
$ns queue-limit $node_(s1) $node_(d) 500
$ns queue-limit $node_(d) $node_(s1) 500

# setup for "nam"
$ns duplex-link-op $node_(s1) $node_(d) queuePos 0.5
$ns duplex-link-op $node_(d) $node_(s1) queuePos 0.5

# set RED router parameters
set reqd [list [$ns link $node_(s1) $node_(d)]] queue]
$reqd set setbit true
$reqd set linterm 10
$reqd set thresh 10
$reqd set maxthresh 20
$reqd set q_weight 0.002

# set general TCP parameters
Agent/TCP set tcpTick 0.01

# create TCP connections and set individual parameters
set tcp1 [$ns create-connection TCP/Reno $node_(s1) TCPSink $node_(d) 0]
$tcp1 set window 128
$tcp1 set ecn 1
$tcp1 set windowOption 1
$tcp1 set windowOptionS 1
$tcp1 set packetSize 512

set tcp2 [$ns create-connection TCP/Reno $node_(s2) TCPSink $node_(d) 1]
$tcp2 set window 128
$tcp2 set ecn 1
$tcp2 set windowOption 1
$tcp2 set windowOptionS 1
$tcp2 set packetSize 512

set tcp3 [$ns create-connection TCP/Reno $node_(s3) TCPSink $node_(d) 2]
$tcp3 set window 128
$tcp3 set ecn 1
$tcp3 set windowOption 1
$tcp3 set windowOptionS 1
$tcp3 set packetSize 512

set tcp4 [$ns create-connection TCP/Reno $node_(s4) TCPSink $node_(d) 3]
$tcp4 set window 128
$tcp4 set ecn 1
$tcp4 set windowOption 1
$tcp4 set windowOptionS 1
$tcp4 set packetSize 512

set tcp5 [$ns create-connection TCP/Reno $node_(s5) TCPSink $node_(d) 4]
C  TCL SCRIPTS FOR THE SIMULATIONS IN SECTION 4

$tcp5 set window_ 128  
$tcp5 set ecn_ 1  
$tcp5 set windowOption_ 1  
$tcp5 set windowOptionS_ 1  
$tcp5 set packetSize_ 512  

set tcp6 [$ns create-connection TCP/Reno $node_(s6) TCPSink $node_(d) 5]  
$tcp6 set window_ 128  
$tcp6 set ecn_ 1  
$tcp6 set windowOption_ 1  
$tcp6 set windowOptionS_ 1  
$tcp6 set packetSize_ 512  

# attach ftp traffic generators  
set ftp1 [$tcp1 attach-app FTP] ; set ftp2 [$tcp2 attach-app FTP]  
set ftp3 [$tcp3 attach-app FTP] ; set ftp4 [$tcp4 attach-app FTP]  
set ftp5 [$tcp5 attach-app FTP] ; set ftp6 [$tcp6 attach-app FTP]  

# schedule events  
$ns at 0.0 "tracestart"  
$ns at 0.0 "$ftp1 start"  
$ns at 0.0 "$ftp2 start"  
$ns at 0.0 "$ftp3 start"  
$ns at 0.0 "$ftp4 start"  
$ns at 0.0 "$ftp5 start"  
$ns at 0.0 "$ftp6 start"  
$ns at 60.0 "finish"  

# start simulation  
$ns run