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

Project Report

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

List of Figures

List of Tables

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^{α} 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 connection 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 $\geq 100ms$

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 computers 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][2] 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 $ssthresh = \frac{1}{2}\ cwnd$ 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-

[2]The analysis was done with the assumption that all connections are synchronized (i.e. have the same delay)

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:

$$cwnd = \begin{cases} cwnd + 1 & if\ [(rate_{expected} - rate_{actual}) \cdot RTT_{minimum}] < \alpha \\ cwnd - 1 & if\ [(rate_{expected} - rate_{actual}) \cdot RTT_{minimum}] > \beta \\ cwnd & else \end{cases} \tag{2.1}$$

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 α packets and at most β 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

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_{drop}, where p_{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

$$q_{avg} = (1 - w_q)\, q_{avg} + w_q\, q \tag{2.2}$$

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}

$$p_{raw} = p_{max}\, \frac{q_{avg} - min_{th}}{max_{th} - min_{th}} \tag{2.3}$$

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

$$p_{drop} = \frac{p_{raw}}{1 - count \cdot p_{raw}} \tag{2.4}$$

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

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].

```
for each packet arrival {
  calculate the average queue size q_avg
  if min_th ≤ q_avg < max_th {
     calculate probability p_drop
     with probability p_drop
        drop the arriving packet
  }
  else if max_th ≤ q_avg
     drop the arriving packet
}
```

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 it congestion window for any reason (i.e. retransmission timer timeouts, duplicate ACKs, and ECN echo ACK[6]).

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 $Ax \leq C$)[8], and if for any other feasible vector x^*, the aggregate of proportional changes is zero or negative:

$$\sum_i \frac{x_i^* - x_i}{x_i} \leq 0 \tag{2.5}$$

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

$$F_I := \frac{(\sum_{i=1}^n x_i)^2}{n \cdot \sum_{i=1}^n {x_i}^2} \tag{2.6}$$

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.

[8] C 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.

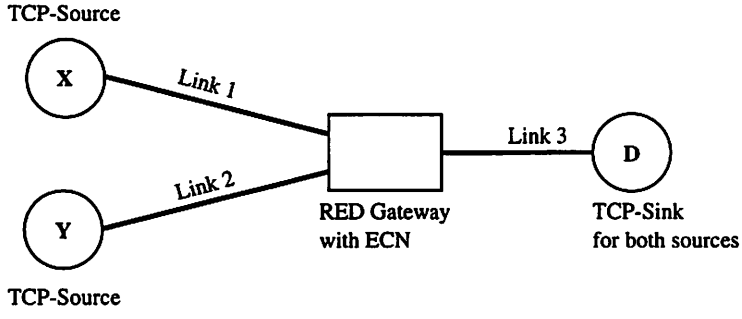

Figure 3.1: Simple model

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

window increase. TCP increases the window size by $\frac{1}{cwnd}$ 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

$$cwnd = cwnd + \frac{1}{cwnd} \cdot S_{wnd} \tag{3.1}$$

Thus, the window increases by roughly S_{wnd}[11] 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]:

$$S_{wnd} = S_{wnd} + \beta \cdot RTT^2 \tag{3.2}$$

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

$$cwnd = \alpha_{wnd} \cdot cwnd \tag{3.3}$$

along with the slow start threshold

$$ssthresh = \alpha_{wnd} \cdot ssthresh \tag{3.4}$$

and decreases the slope of the window

$$S_{wnd} = \alpha_{slp} \cdot S_{wnd} \tag{3.5}$$

[11] called *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 $S_{wnd} > 1$, S_{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_{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
- $packetsize = 512$ 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: $RTT = 40ms$
- Connection Y: $RTT = 20ms$

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

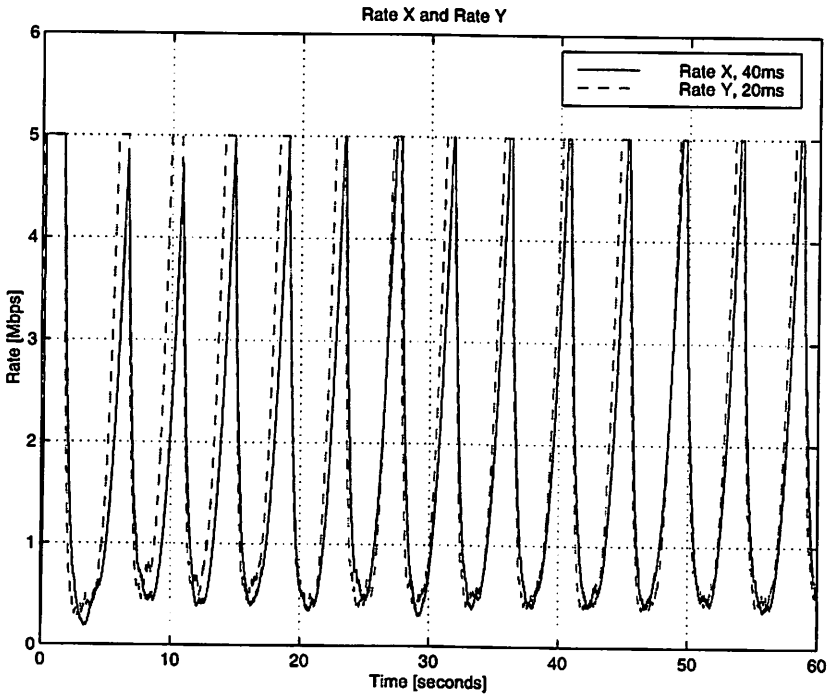

Figure 3.2: Simple model: rates

the fairness index for this simulation

$$F_I = \frac{(\sum_{i=1}^{2} \overline{rate_i})^2}{2 \cdot \sum_{i=1}^{2} \overline{rate_i}^2} \approx 0.99 \tag{3.6}$$

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

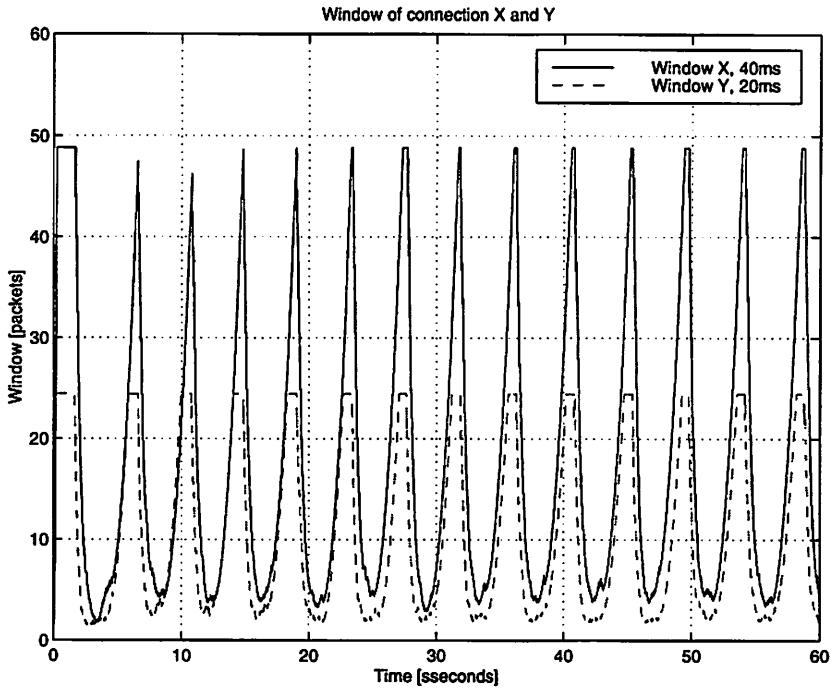

Figure 3.3: Simple model: windows

connections, we get $\overline{cwnd_x} = 18.1$ *packets* and $\overline{cwnd_y} = 10.4$ *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].

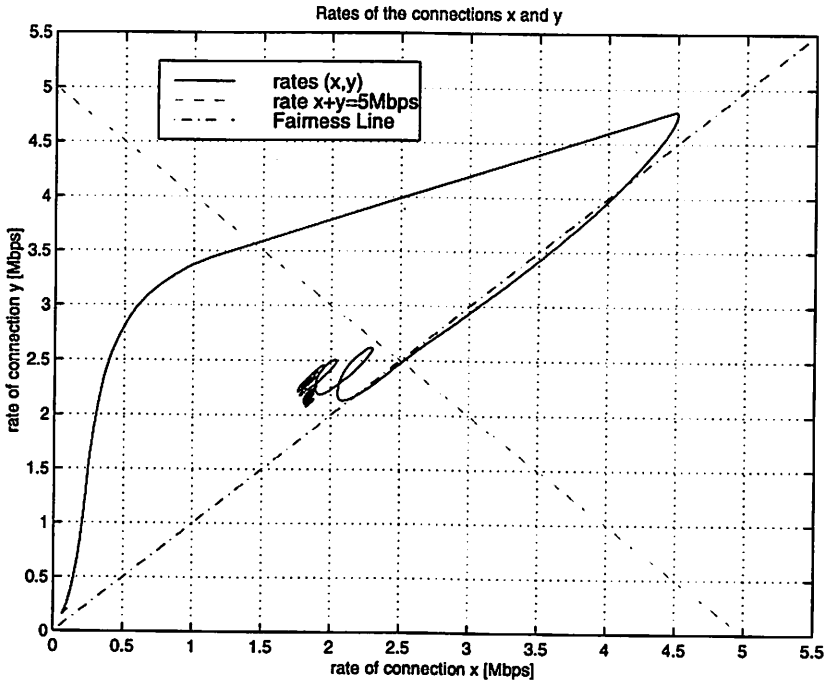

Figure 3.4: Simple model: rate trace

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:

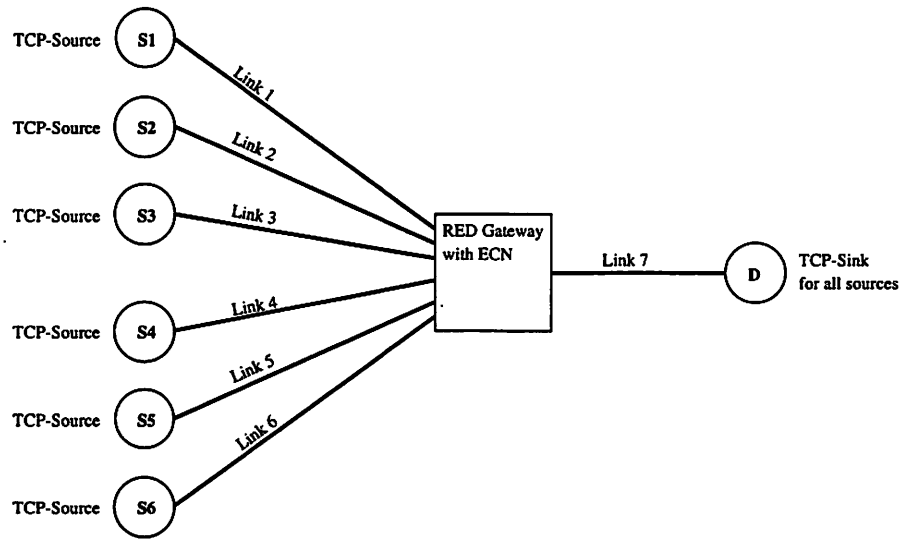

Figure 4.1: Simulation topology

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

- $packetsize = 512$ bytes
- $tcpTick\_ = 0.01$: TCP clock (in seconds)
- $\alpha_{wnd} = 0.9$: window reduce factor
- $\alpha_{slp} = 0.9$: window slope reduce factor

- $\beta = 16$: 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.

	Link 1	Link 2	Link 3	Link 4	Link 7
Delay	$94ms$	$34ms$	$14ms$	$1ms$	$1ms$
Bandwidth	$10Mbps$	$10Mbps$	$10Mbps$	$10Mbps$	$10Mbps$

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

	TCP 1	TCP 2	TCP 3	TCP 4
Connection	S1 $\rightarrow$ D	S2 $\rightarrow$ D	S3 $\rightarrow$ D	S4 $\rightarrow$ D
RTT_{min}	$190ms$	$70ms$	$30ms$	$4ms$

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).

	Link 1	Link 2	Link 3	Link 4	Link 7
Delay	$94ms$	$34ms$	$14ms$	$1ms$	$1ms$
Bandwidth	$10Mbps$	$10Mbps$	$10Mbps$	$10Mbps$	$2Mbps$

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:

$$F_I = \frac{(\sum_{i=1}^{4} \overline{rate_i})^2}{4 \cdot \sum_{i=1}^{4} \overline{rate_i}^2} \approx 0.99 \tag{4.1}$$

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

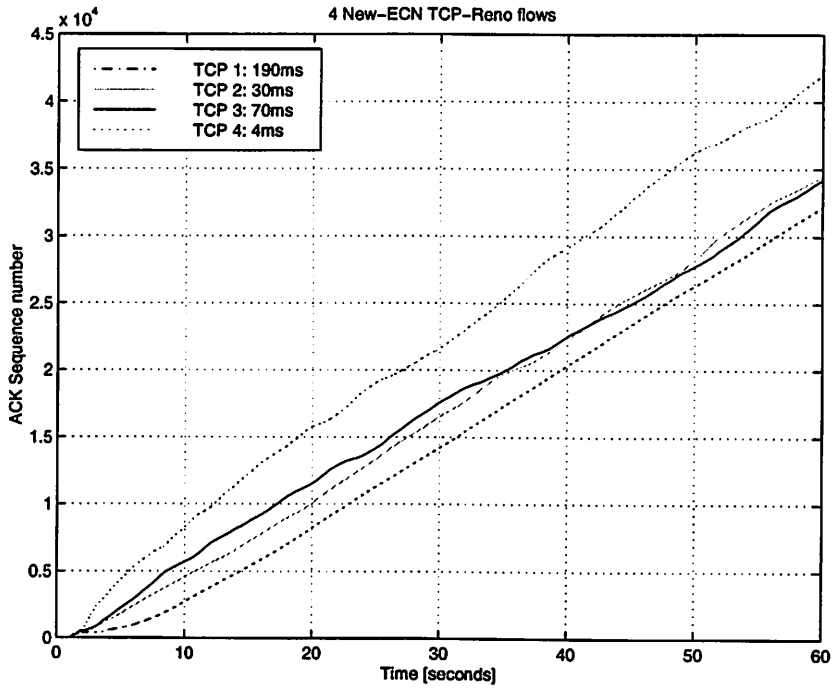

Figure 4.2: 4 New-ECN TCP-Reno flows: ACK sequence number plot

Figure 4.4 shows the value of S_{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^{α} with $1 < \alpha < 2$. In Table 4.6 the difference between the ratio of the mean (smoothed) RTT raised to α and the ratio of the measured mean value of S_{wnd} of two flows i and j is calculated in order to find an approximation for α. From this Table it appears that the value of α is roughly $\frac{3}{2}$. Table 4.4 contains the mean values after 60 seconds and Table 4.5 the ratios of $\overline{cwnd}$, $\overline{sRTT}$[14], and $\overline{S_{wnd}}$.

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 $RTT_{measured}$ changes by $sRTT = \alpha\ sRTT + (1 - \alpha)\ RTT_{measured}$

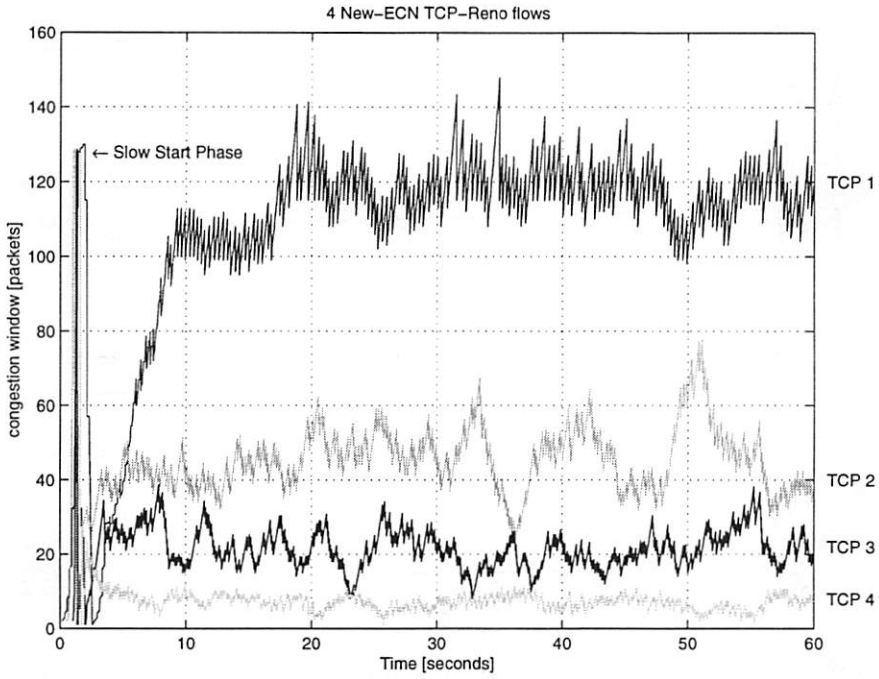

Figure 4.3: 4 New-ECN TCP-Reno flows: congestion window plot

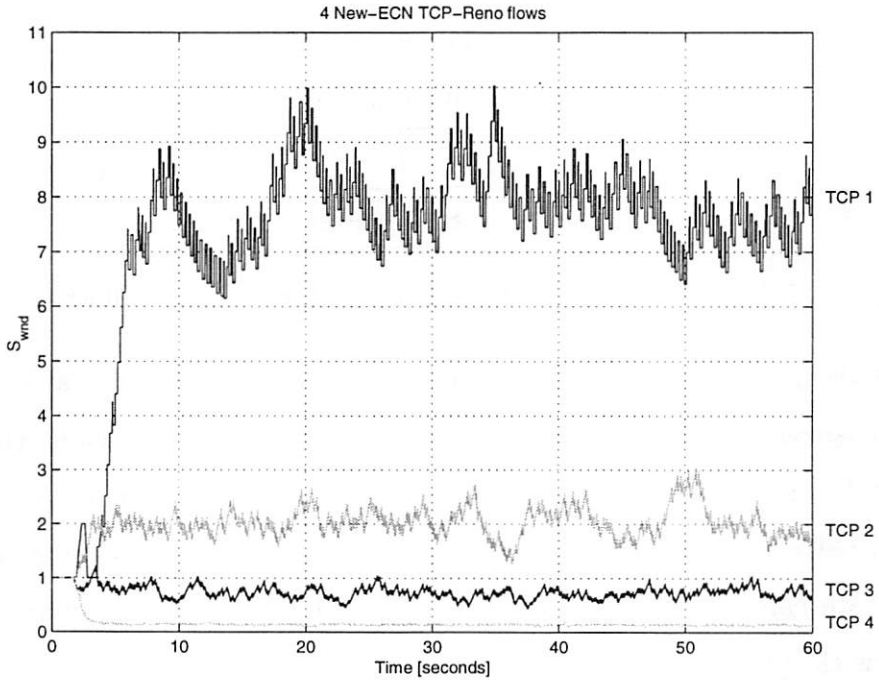

Figure 4.4: 4 New-ECN TCP-Reno flows: S_{wnd} plot

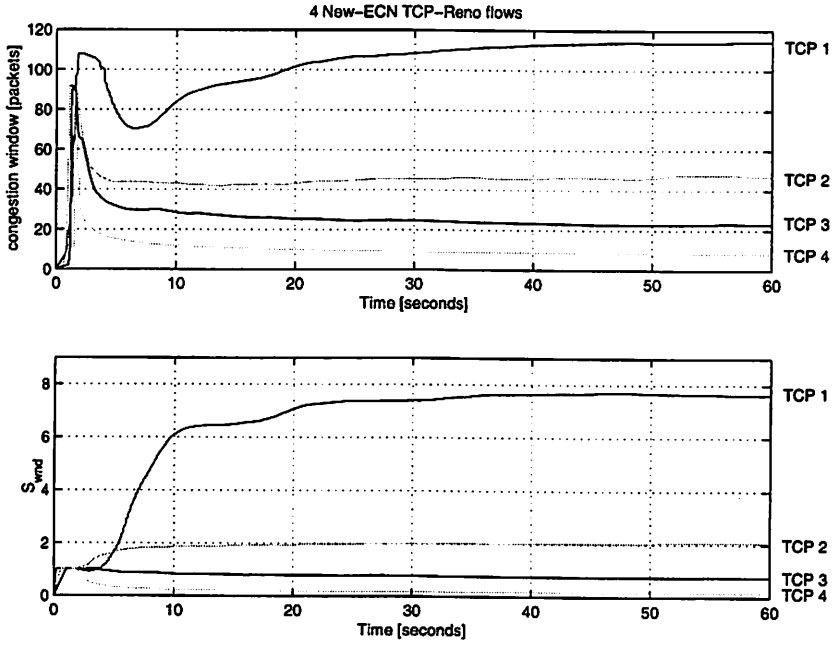

Figure 4.5: 4 New-ECN TCP-Reno flows: average window and S_{wnd}

TCP	$\overline{sRTT}$	$\overline{cwnd}$	$\overline{S_{wnd}}$
1	195*ms*	114.26 *packets*	7.627
2	75*ms*	46.86 *packets*	2.048
3	35*ms*	23.14 *packets*	0.758
4	15*ms*	8.21 *packets*	0.162

Table 4.4: 4 New-ECN TCP-Reno flows: mean values

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-

	$\overline{sRTT}$ ratio	$\overline{cwnd}$ ratio	$\overline{S_{wnd}}$ ratio
$TCP_1 : TCP_4$	13.02	13.91	47.22
$TCP_2 : TCP_4$	5.00	5.70	12.68
$TCP_3 : TCP_4$	2.34	2.82	4.69
$TCP_1 : TCP_3$	5.56	4.94	10.07
$TCP_2 : TCP_3$	2.14	2.02	2.7
$TCP_1 : TCP_2$	2.60	2.44	3.72

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

α	$\Delta r_{1,4}(\alpha)$	$\Delta r_{2,4}(\alpha)$	$\Delta r_{3,4}(\alpha)$	$\Delta r_{1,3}(\alpha)$	$\Delta r_{2,3}(\alpha)$	$\Delta r_{1,2}(\alpha)$
1.0	*-34.20*	*-7.67*	*-2.35*	*-4.50*	*-0.56*	*-1.12*
1.1	*-30.39*	*-6.80*	*-2.14*	*-3.46*	*-0.40*	*-0.86*
1.2	*-25.46*	*-5.77*	*-1.92*	*-2.22*	*-0.21*	*-0.57*
1.3	*-19.10*	*-4.57*	*-1.67*	*-0.76*	*-0.02*	*-0.26*
1.4	*-10.87*	*-3.15*	*-1.40*	0.99	0.19	0.09
1.5	*-0.24*	*-1.49*	*-1.11*	3.06	0.42	0.47
1.6	13.51	0.47	*-0.79*	5.51	0.67	0.89
1.7	31.28	2.77	*-0.45*	8.43	0.94	1.36
1.8	54.24	5.47	*-0.07*	11.89	1.22	1.87
1.9	83.93	8.63	0.34	16.00	1.53	2.43
2.0	122.30	12.36	0.79	20.89	1.87	3.05

$$\Delta r_{i,j}(\alpha) := \frac{\overline{sRTT_i^{\alpha}}}{\overline{sRTT_j^{\alpha}}} - \frac{\overline{S_{wnd,i}}}{\overline{S_{wnd,j}}} \tag{4.2}$$

Table 4.6: 4 New-ECN TCP-Reno flows: approximation for α

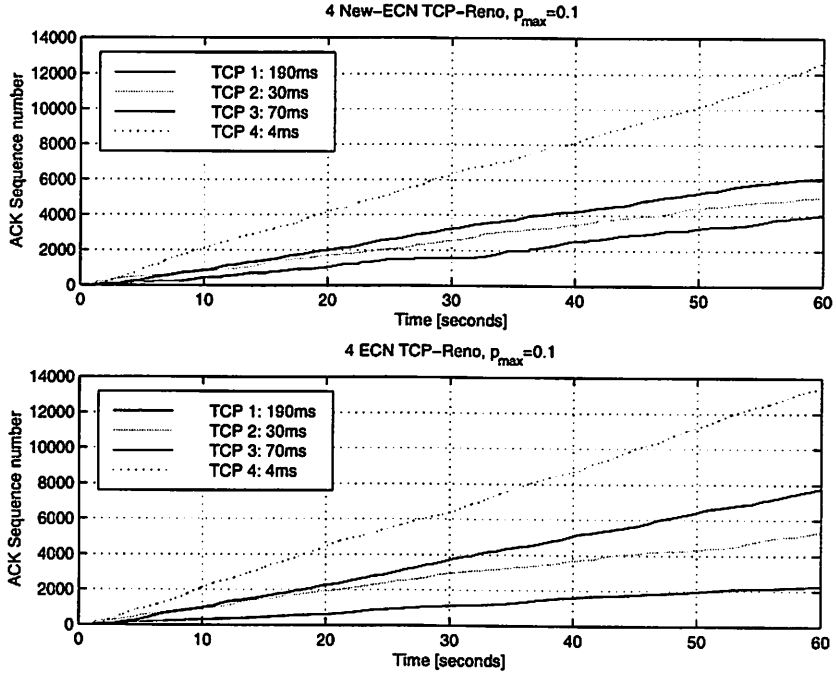

Figure 4.6: 4 TCP-Reno flows: ACK sequence numbers plot for setup 2, $p_{max} = 0.1$

tions 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_{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_{max} showed that simulation setup 1 can achieve a fair sharing for $\frac{1}{10} < p_{max} < \frac{1}{3}$. For $p_{max} > \frac{1}{3}$ we start seeing a bias against fast connections. Fortunately, the sensitivity to p_{max} is only moderate. The interaction of a adaptive scheme as in [23] with New-ECN needs further study.

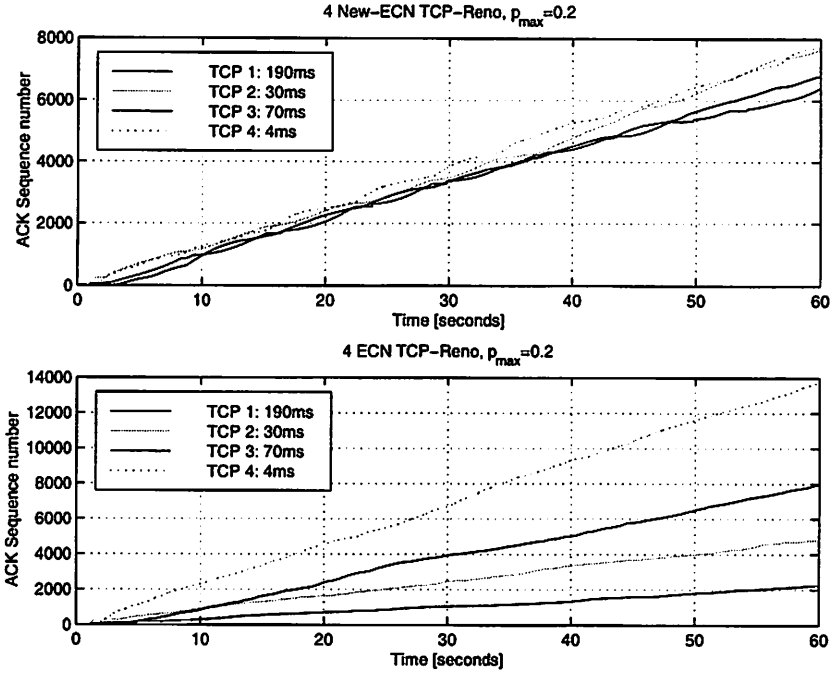

Figure 4.7: 4 TCP-Reno flows: ACK sequence numbers plot for setup 2, $p_{max} = 0.2$

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

$$F_{I,New-ECN} = \frac{(\sum_{i=1}^{4} \overline{rate_i})^2}{4 \cdot \sum_{i=1}^{4} \overline{rate_i}^2} \approx 0.99 \qquad F_{I,ECN} = \frac{(\sum_{i=1}^{4} \overline{rate_i})^2}{4 \cdot \sum_{i=1}^{4} \overline{rate_i}^2} \approx 0.48 \qquad (4.3)$$

shows that the fairness improved approximately by factor 2.

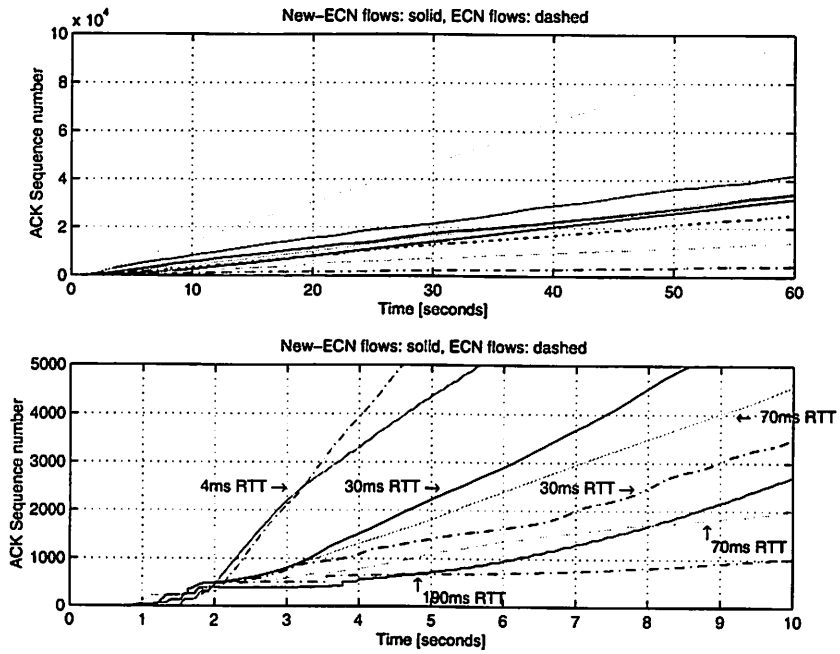

Figure 4.8: 4 New-ECN vs. 4 ECN TCP-Reno flows: ACK sequence number plot

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_{max} has a dependency on the grade of congestion. For this simulation a value of $p_{max} = 0.2$ was needed to achieve a fair sharing of bandwidth.

Link	1	2	3	4	5	6	7	8	9
Delay $[ms]$	44	34	28	24	14	9	9	4	1
Bandwidth $[Mbps]$	10	10	10	10	10	10	10	10	10

Table 4.7: Many New-ECN TCP-Reno flows: link parameter

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

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

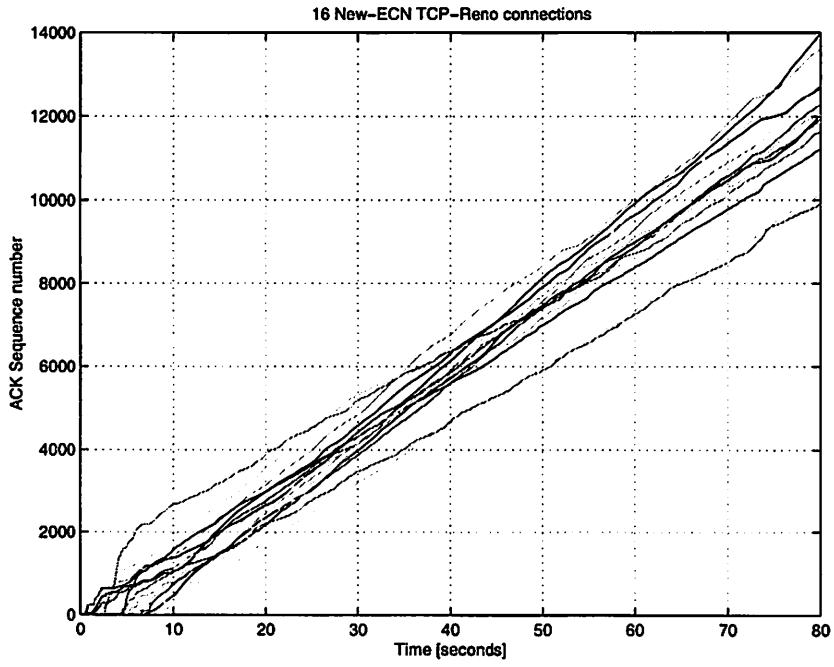

Figure 4.9: 16 New-ECN TCP-Reno connections: ACK sequence number plot

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 $\alpha_{wnd} = 0.9$. 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 a 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 $B = 512$ *bytes*
- total packets sent: 72583 *packets*
- total dropped packets: 90 *packets*
- minimum RTT $R = 30$ *ms*

we get

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

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

- TCP-Reno : 1077 *packets/s*
- ECN TCP-Reno : 1157 *packets/s*
- New-ECN TCP-Reno : 1208 *packets/s*

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.

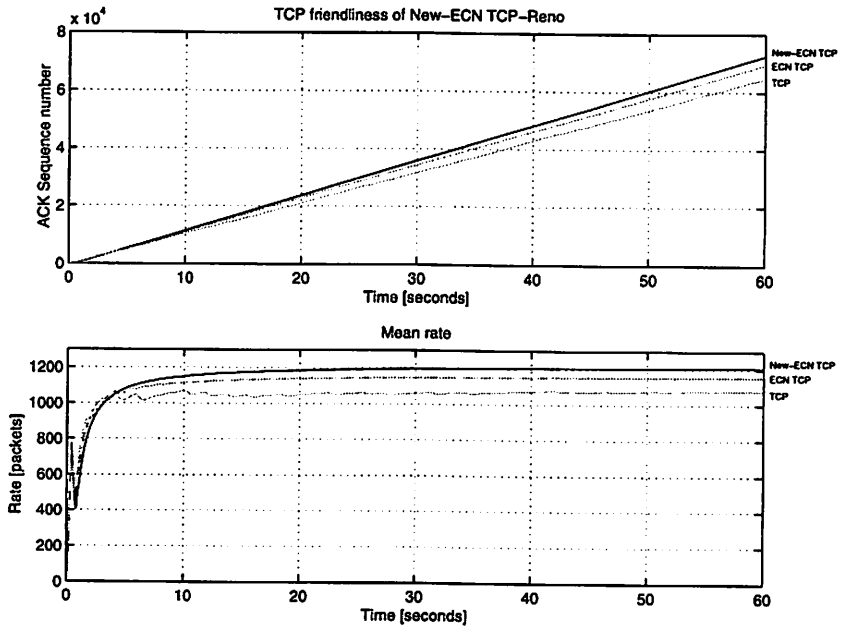

Figure 4.10: One New-ECN flow vs. one ECN/non-ECN flow

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.

	Link 1	Link 2	Link 3	Link 4	Link 5	Link 6	Link 7
Delay	$34ms$	$14ms$	$1ms$	$34ms$	$14ms$	$1ms$	$1ms$
Bandwidth	$10Mbps$	$10Mbps$	$10Mbps$	$10Mbps$	$10Mbps$	$10Mbps$	$10Mbps$

Table 4.9: TCP / ECN-TCP friendliness: link parameter

	TCP 1	TCP 2	TCP 3	TCP 4	TCP 5	TCP 6
Connection	S1 → D	S2 → D	S3 → D	S4 → D	S5 → D	S6 → D
RTT_{min}	$70ms$	$30ms$	$4ms$	$70ms$	$30ms$	$4ms$
Start time	$0.0s$	$0.0s$	$0.0s$	$0.0s$	$0.0s$	$0.0s$

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.

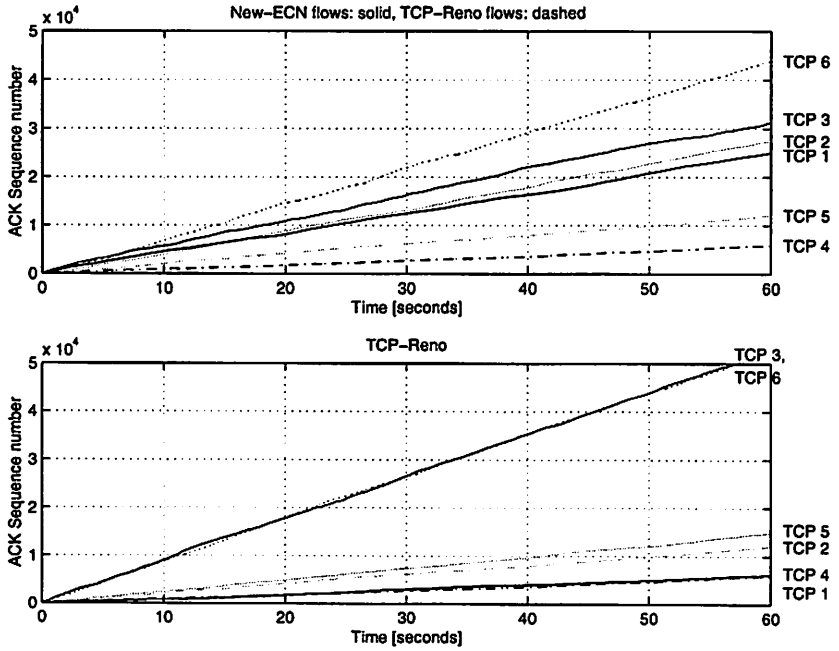

Figure 4.11: TCP friendliness: ACK sequence number plot

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.

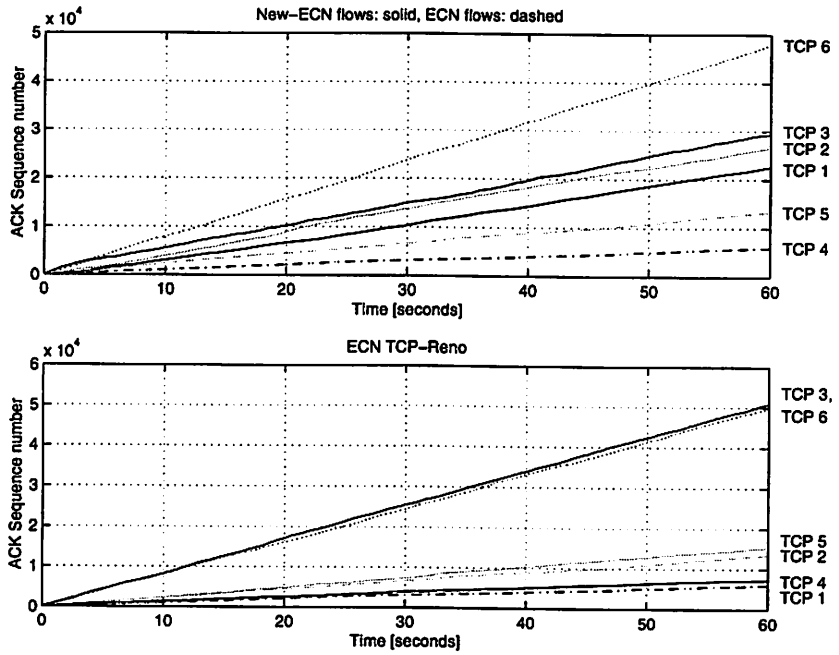

Figure 4.12: ECN-TCP friendliness: ACK sequence number plot

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_{max} at the RED gateway. We suggested an adaptive mechanism for the value of p_{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.

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 Odyssia Systems. Tilo Hamann was supported by a scholarship provided by the Ditze-Foundation of the Technical University of Hamburg-Harburg.

References

[1] S. Floyd. TCP and Explicit Congestion Notification. *ACM Computer Communication Review*, 24(5):8–23, October 1994.

[2] S. Floyd and K. Ramakrishnan. A Proposal to add Explicit Congestion Notification (ECN) to IP. *RFC 2481*, January 1999.

[3] S. Floyd, D. Black, and K. Ramakrishnan. IPsec Interactions with ECN. *Internet-Draft: draft-ipsec-ecn-00.txt*, April 1999.

[4] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):pp.397–413, August 1993.

[5] S. Floyd. Connection with Multiple Congested Gateways in Packet-Switched Networks, Part 1: One-Way Traffic. *ACM Computer Communication Review*, 21(5):30–47, October 1991.

[6] The UCB/LBNL/VINT Network Simulator. http://www-mash.cs.berkeley.edu/ns.

[7] V. Jacobson. Congestion Avoidance and Control. *Porc. Computer Communication Review*, 18(4):314–329, August 1998.

[8] V. Jacobson. Modified TCP Congestion Avoidance Algorithm. Technical report, End-2-End-Interest Mailing List, April 1990.

[9] L. Zhang and D. Clark. Oscillating Behavior of Network Traffic: A Case Study Simulation. *Internetworking: Research and Experience*, 1(2):101–112, December 1990.

[10] T. Lakshman and U. Madhow. The Performance of TCP/IP for Networks with High Bandwidth-Delay Products and Random Loss. *IEEE/ACM Transactions on Networking*, 5(3):336–350, June 1997.

[11] H. Krishnan. Analyzing Explicit Congestion Notification (ECN) Benefits for TCP. Master's thesis, UCLA, 1998.

[12] W. Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. *RFC 2001*, January 1997.

[13] W. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols.* Addison-Wesley, 1994.

[14] G. Wright and W.Stevens. *TCP/IP Illustrated, Volume 2: The Implementation.* Addison-Wesley, 1995.

[15] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. *RFC 2581*, April 1999.

[16] S. Floyd and T. Henderson. The NewReno Modification to TCP's FastRecovery Algorithm. *RFC 2582*, April 1999.

[17] D . Chiu and R. Jain. Analysis of the Increase/Decrease Algorithms for Congestion Avoidance in Computer Networks. *Journal of Computer Networks and ISDN*, 17(1):1–14, June 1989.

[18] J. Mo, R. La, V. Anantharam, and J. Walrand. Analysis and Comparison of TCP Reno and Vegas. *INFOCOM 99*, March 1999.

[19] J. Hoe. Start-up Dynamics of TCP's Congestion Control and Avoidance Schemes. Master's thesis, MIT, June 1995.

[20] L. Brakmo and L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, October 1995.

[21] D. Lin and R. Morris. Dynamics of Random Early Detection. *ACM Computer Communication Review*, 27(4):127–137, October 1997.

[22] W. Feng, D. Kandlur, D. Saha, and K. Shin. Techniques for Eliminating Packet Loss in Congested TCP/IP Networks. Technical report, CSE-TR-349-97, U. Michigan, 1997.

[23] W. Feng, D. Kandlur, D. Saha, and K. Shin. A Self-Configuring RED Gateway. *INFOCOM 99*, March 1999.

[24] T. Ott, T. Lakshman, and L. Wang. SRED: Stabilized RED. *INFOCOM 99*, March 1999.

[25] D. Bersekas and R. Gallager. *Data Networks.* Prentice-Hall, Englwood Cliffs, N.J., 1992.

[26] F. Kelly. Charging and Rate Control for Elastic Traffic. *European Transactions on Telecommunications*, 8(1):33–37, 1997.

[27] R. Jain, D. Chiu, and W. Hawe. A Quantitative Measure of Fairness and Discrimination for Resources Allocation in Shared Systems. Technical report, DEC TR-301, Digital Equipment Corporation, Littleton, MA, September 1984.

[28] S. Floyd and K. Fall. Promoting the Use of End-to-End Congestion Control in the Internet. *To appear in IEEE/ACM Transactions on Networking*, August 1999.

[29] K. Fall and S. Floyd. Simulation-based Comparisons of Tahoe, Reno, and SACK TCP. *Computer Communication Review*, 26(3):5–21, July 1996.

[30] S. Floyd and V. Jacobson. On Traffic Phase Effects in Packet-Switched Gateways. *Internetworking: Research and Experience*, 3(3):115–156, September 1992.

[31] R. Gibbens and F. Kelly. Resource Pricing and the Evolution of Congestion Control. *Automatica*, 35, 1999.

[32] T. Henderson, E. Sahouria, S. McCanne, and R. Katz. On Improving the Fairness of TCP Congestion Avoidance. *Proceedings of IEEE Globecom '98*, November 1998.

[33] F. Kelly, A. Maulloo, and D. Tan. Rate Control in Communication Networks: Shadow Prices, Proportional Fairness and Stability. *Journal of the Operational Research Society*, 49(3):237–252, March 1998.

[34] D. Tan. Rate Control and User Behaviour in Communication Networks. *4th INFORMS Telecommunications Conference*, 1998.

A Matlab-File for the Simple Model Simulation

```
clear;

% Basic parameters
Packetsize=512*8;                % packet size [bytes]
t_incr=10e-3;                    % time increasing factor [s]
MaxSteps=500;                    % number of cycles
RTTx=40e-3;                      % round trip time of connection x  [s]
RTTy=20e-3;                      % round trip time of connection y  [s]
MaxRateX=5e6;                    % line rate connection x [bps]
MaxRateY=5e6;                    % line rate connection y [bps]

% New-ECN TCP parameters
alpha_slp=0.70;                  % reduce factor for the slope of the window
alpha_wnd=0.9;                   % reduce factor for the window
beta=16;                         % window slope increase factor

% TCP connection parameters
% Connection x
MaxIncrX=t_incr/RTTx;            % window increment per t_incr
MaxCwndX=MaxRateX*RTTx/Packetsize;   % maximum window size
xw0=1;                           % initial window
ssthreshX=65535*8/Packetsize;    % slow start threshold
counterx=1;                      % variable for handling slow start window evolution
jx=1;                            % variable for handling slow start window evolution
xw(1)=xw0;                       % initialize window vector
DelayX=floor(RTTx/t_incr);       % variable for handling the delay of the first packets
FlagX=1;                         % flag, set to zero after the first received mark
S_wnd_X=1;                       % window slope

% Connection y
MaxIncrY=t_incr/RTTy;            % window increment per t_incr
MaxCwndY=MaxRateY*RTTy/Packetsize;   % maximum window size
yw0=1;                           % initial window
ssthreshY=65535*8/Packetsize;    % slow start threshold
countery=1;                      % variable for handling slow start window evolution
jy=1;                            % variable for handling slow start window evolution
yw(1)=yw0;                       % initialize window vector
DelayY=floor(RTTy/t_incr);       % variable for handling the delay of the first packets
FlagY=1;                         % flag, set to zero after the first received mark
S_wnd_Y=1;                       % window slope

% Outgoing link
c=5e6;                           % outgoing link rate of the bottleneck router [bps]
Servicetime=1/(c/Packetsize);    % service time for one packet

% RED parameters:
MaxTh=20;                        % upper threshold
MinTh=10;                        % lower threshold
MaxP=0.25;                       % maximum marking probability
wq=0.002;                        % queue weight factor for calculating the avg. queue size
C1=MaxP/(MaxTh-MinTh);           % values needed to calculate p_drop
C2=(MaxP*MinTh)/(MaxTh-MinTh);   % values needed to calculate p_drop
q_i_time=0;                      % queue idle time
count=-1;                        % packets since last marked packet
p_drop=0;                        % current marking probability
R=rand(1,1);                     % a random number, uniform on [0,1]
queue(1)=0;                      % instantaneous queue size
avgqueue(1)=0;                   % average queue size

% Other parameters
HandlingPacket=-1;               % which packet is handled by the router: 0: X, 1:Y
```

```
npx=0;                          % number of packets received on connection x [p]
npy=0;                          % number of packets received on connection y [p]
npmx(1)=0;                      % number of marked packets for connection x [p]
npmy(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(1)=npx;                 % log number of packets received on connection x [p]
npy_log(1)=npy;                 % log number of packets received on connection y [p]
marksx_log_log(1)=npmx(1);      % log marks received on x
marksy_log(1)=npmy(1);          % log marks received on y
npt_log(1)=0;                   % log number of packets arrived at the router
qi_log(1)=0;                    % log queue idle time
mmx(1)=xw(1)*Packetsize/RTTx;   % mean rate for connection x
mmy(1)=yw(1)*Packetsize/RTTy;   % mean rate for connection y
S_wnd_X_log(1)=1;               % log window slope connection x
S_wnd_Y_log(1)=1;               % log window slope connection x
Time_Now_X=0;                   % variable to handle the increase of the slope
Time_Now_X=0;                   % variable to handle the increase of the slope
Last_Time_X=0;                  % variable to handle the increase of the slope
Time_Now_Y=0;                   % variable to handle the increase of the slope
Last_Time_Y=0;                  % variable to handle the increase of the slope
Time_Now_2_X=0;                 % variable to handle the decrease of the slope and window
Last_Time_2_X=0;                % variable to handle the decrease of the slope and window
Time_Now_2_Y=0;                 % variable to handle the decrease of the slope and window
Last_Time_2_Y=0;                % variable to handle the decrease of the slope and window

% Start loop
for k=2:MaxSteps
   % updating TCP parameters for connection X, window based
   if (k-DelayX<1)
      % marks not effective yet
      if (xw(k-1)<ssthreshX)
         % slow start7
         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 ((npmx(k-DelayX)==0))
         % no marks received
         npmx(k-DelayX)=0;   % reset marks
         Time_Now_X = k *t_incr;
         if ((FlagX<1)&(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*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=max(xw(k)/2,1);     % decrease ssthresh
            Last_Time_2_X=Time_Now_2_X;
         else
            xw(k)=xw(k-1);
         end;
      end;
      npmx(k-DelayX)=0;   % reset marks
   end;
end;
%log variables for connection x
ssthreshX_log(k)=ssthreshX;       % log ssthresh
x(k)=xw(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*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_Y+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:
Temp1X=floor(xw(k))*t_incr/RTTx;  % Calculating the number of arrived packets
Temp1Y=floor(yw(k))*t_incr/RTTy;  % Calculating the number of arrived packets
CarryX=CarryX+mod(Temp1X,1);      % Calculating the number of arrived packets
CarryY=CarryY+mod(Temp1Y,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
```

```
      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;

npmx(k)=0; npmy(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;
end;

% handling packets at the queue
while ((npx+npy)>0)
   % decide which packet is placed in the queue
   if (NoX==1)
      HandlingPacket=1;
   else
      if (NoY==1)
         HandlingPacket=0;
      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);
```

```
            QCarry=mod((1/Ratio)+QCarry,1);
          end;
        end;
      end;
    end;

    queue(k)=queue(k-1)+1;              % increase queue
    % calculating average queuesize
    if (queue(k)>0)                     % queue is not empty
      avgqueue(k)=avgqueue(k-1)+wq*(queue(k)-avgqueue(k-1));
      q_i_time=0;
    else                                % queue is empty
      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;         % calculating the marking probability
      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);end;
    else
      if (avgqueue(k)>=MaxTh)           % queue size is beyond the upper threshold
        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=-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;end;
    if (HandlingPacket==1) npy=npy-1;end;
  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

```
#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_HALVE      0x00000040
```

is extended by

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

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

```
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), ts_peer_(0),
    rtx_timer_(this), delsnd_timer_(this), burstsnd_timer_(this),
    dupacks_(0), curseq_(0), highest_ack_(0), cwnd_(0), ssthresh_(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("windowThresh_", &wnd_th_);
    bind_bool("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_thresh_", &eln_rxmit_thresh_);
    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("srtt_init_", &srtt_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("dupacks_", &dupacks_);
    bind("seqno_", &curseq_);
    bind("t_seqno_", &t_seqno_);
    bind("ack_", &highest_ack_);
    bind("cwnd_", &cwnd_);
    bind("awnd_", &awnd_);
    bind("ssthresh_", &ssthresh_);
    bind("rtt_", &t_rtt_);
    bind("srtt_", &t_srtt_);
    bind("rttvar_", &t_rttvar_);
    bind("backoff_", &t_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_Received_);         // flag, set to one for a received mark
```

```
    //-end-------------------------------------------------------------------------------

#ifdef OFF_HDR
    bind("off_ip_", &off_ip_);
    bind("off_tcp_", &off_tcp_);
#else
    off_ip_ = hdr_ip::offset();
    off_tcp_ = hdr_tcp::offset();
#endif
    bind("ndatapack_", &ndatapack_);
    bind("ndatabytes_", &ndatabytes_);
    bind("nackpack_", &nackpack_);
    bind("nrexmit_", &nrexmit_);
    bind("nrexmitpack_", &nrexmitpack_);
    bind("nrexmitbytes_", &nrexmitbytes_);
    bind_bool("trace_all_oneline_", &trace_all_oneline_);
    bind_bool("nam_tracevar_", &nam_tracevar_);

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


TcpAgent::traceVar(TracedVar* v) {
    double curtime;
    Scheduler& s = Scheduler::instance();
    char wrk[500];
    int n;

    //-begin-----------------------------------------------------------------------------
    curtime = &s ? s.clock() : 0;
    if (!strcmp(v->name(), "cwnd_") || !strcmp(v->name(), "maxrto_"))
        // added trace for Slope_Window_
        sprintf(wrk,"%-8.8f %-2d %-2d %-2d %-2d %s %-6.3f %-8.8f", curtime, addr_/256, addr_%256,
          dst_/256, dst_%256, v->name(), double(*((TracedDouble*) v)),double(Slope_Window_));
    else if (!strcmp(v->name(), "rtt_"))
        // added trace for First_Time_
        sprintf(wrk,"%-8.8f %-2d %-2d %-2d %-2d %s %-6.3f FT=%f", curtime, addr_/256, addr_%256,
          dst_/256, dst_%256, v->name(), int(*((TracedInt*) v))*tcp_tick_, double(First_Time_));
    else if (!strcmp(v->name(), "srtt_"))
        sprintf(wrk,"%-8.8f %-2d %-2d %-2d %-2d %s %-6.3f", curtime, addr_/256, addr_%256,
          dst_/256, dst_%256, v->name(), (int(*((TracedInt*) v)) >> T_SRTT_BITS)*tcp_tick_);
    else if (!strcmp(v->name(), "rttvar_"))
        sprintf(wrk,"%-8.8f %-2d %-2d %-2d %-2d %s %-6.3f", curtime, addr_/256, addr_%256,
          dst_/256, dst_%256, v->name(), int(*((TracedInt*) v))*tcp_tick_/4.0);
    else
        // added trace for Mark_Received_
        sprintf(wrk,"%-8.8f %-2d %-2d %-2d %-2d %s %d %-8.8f", curtime, addr_/256, addr_%256,
          dst_/256, dst_%256, v->name(), int(*((TracedInt*) v)),double(Mark_Received_));
    //-end-------------------------------------------------------------------------------

    n = strlen(wrk);
    wrk[n] = '\n';
    wrk[n+1] = 0;
    if (channel_)
        (void)Tcl_Write(channel_, wrk, n+1);
    wrk[n] = 0;
    return;
}


void TcpAgent::reset() {
    rtt_init();
    /*XXX lookup variables */
```

```
    dupacks_ = 0;
    curseq_ = 0;
    set_initial_window();

    //-begin-----------------------------------------------------------------------------------
    Last_Time_=0.0;         // initial values
    Slope_Window_=1.0;      // initial values
    First_Time_=0;          // initial values
    Mark_Received_=0.0;     // initial values
    //-end-------------------------------------------------------------------------------------

    t_seqno_ = 0;
    maxseq_ = -1;
    last_ack_ = -1;
    highest_ack_ = -1;
    ssthresh_ = int(wnd_);
    wnd_restart_ = 1.;
    awnd_ = wnd_init_ / 2.0;
    recover_ = 0;
    closed_ = 0;
    last_cwnd_action_ = 0;
    boot_time_ = Random::uniform(tcp_tick_);
}


void TcpAgent::opencwnd() {
  //-begin-----------------------------------------------------------------------------------
  double Time_Now;
  double f;
  //-end-------------------------------------------------------------------------------------

  if (cwnd_ < 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;
           f *= wnd_const_;
           f += fcnt_;
           if (f > cwnd_) {
               fcnt_ = 0;
               ++cwnd_;
           } else
               fcnt_ = f;
           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
            /*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_>1.0))
            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 (wnd_option_ == 8) {
          slowdown(CLOSE_CWND_NEW|CLOSE_SSTHRESH_NEW);
          First_Time_=1;
      }
      else {
        slowdown(CLOSE_CWND_HALF|CLOSE_SSTHRESH_HALF);
      }
   //-end-----------------------------------------------------------------------------
   }
}


void TcpAgent::recv_newack_helper(Packet *pkt) {
   //hdr_tcp *tcph = hdr_tcp::access(pkt);
   newack(pkt);

   //-begin---------------------------------------------------------------------------
   if (wnd_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();
      }
   }
   //-end-----------------------------------------------------------------------------

  if (ect_) {
       if (!hdr_flags::access(pkt)->ecnecho())
           ecn_backoff_ = 0;
       if (!ecn_burst_ && hdr_flags::access(pkt)->ecnecho())
           ecn_burst_ = TRUE;
       else if (ecn_burst_ && ! hdr_flags::access(pkt)->ecnecho())
           ecn_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 *tcph = hdr_tcp::access(pkt);
#ifdef notdef
   if (pkt->type_ != PT_ACK) {
       Tcl::instance().evalf("%s error \"received non-ack\"",
                     name());
       Packet::free(pkt);
       return;
   }
#endif
   ++nackpack_;
   ts_peer_ = tcph->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)->eln_ && eln_) {
                  tcp_eln(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_);
#ifdef notdef
    if (pkt->type_ != PT_ACK) {
        fprintf(stderr,
            "ns: confiuration error: tcp received non-ack\n");
        exit(1);
    }
#endif
    ++nackpack_;
    ts_peer_ = tcph->ts();

   //-begin-------------------------------------------------------------------------------
    if (((hdr_flags*)pkt->access(off_flags_))->ecnecho() && ecn_) {
        ecn(tcph->seqno());
        Mark_Received_=1;       // trace marks
    } else if (!((hdr_flags*)pkt->access(off_flags_))->ecnecho() && 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_))->eln_ && eln_) {
            tcp_eln(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 responsed to congestion.
        acker_->update_ecn_unacked(0);
    if ( (save_ != NULL && sf->ect() && sf->ce())  ||
```

```
            (of->ect() && of->ce()) )
        // New report of congestion.
        acker_->update_ecn_unacked(1);

    //-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->ecnecho() ||
         (save_ != NULL && !sf->ect() && sf->ecnecho()) )
         // 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->ecnecho() = 1;
    acker_->append_ack((hdr_cmn*)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:

```
# create data from the trace-files for a Matlab analysis
proc makeplot number {
        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";
```

```
    } }
  }
  set awkCodeSD {
    {
     if ($6 == "cwnd_") {
         print $1 >> "temp.t.sd";
         print $8 >> "temp.v.sd";
    } }
  }

  set cwndt [open "tcp$number.tcwnd" w] ; set cwndv [open "tcp$number.vcwnd" w]
  set srttt [open "tcp$number.tsrtt" w] ; set srttv [open "tcp$number.vsrtt" w]
  set ackt [open "tcp$number.tack" w]   ; set ackv [open "tcp$number.vack" w]
  set sdft [open "tcp$number.tsd" w]    ; set sdfv [open "tcp$number.vsd" w]
  set mark [open "tcp$number.vmark" w]

  set ct "temp.t.cwnd" ;  set cv "temp.v.cwnd"
  set st "temp.t.srtt" ;  set sv "temp.v.srtt"
  set at "temp.t.ack"  ;  set av "temp.v.ack"
  set am "temp.m.ack"  ;  set sdt "temp.t.sd"
  set sdv "temp.v.sd"

  exec rm -f $am   ;  exec touch $am
  exec rm -f $ct   ;  exec touch $ct
  exec rm -f $st   ;  exec touch $st
  exec rm -f $at   ;  exec touch $at
  exec rm -f $sdt  ;  exec touch $sdt
  exec rm -f $cv   ;  exec touch $cv
  exec rm -f $sv   ;  exec touch $sv
  exec rm -f $av   ;  exec touch $av
  exec rm -f $sdv  ;  exec touch $sdv

  exec awk $awkCodeCwnd "tcp$number.tr"
  exec awk $awkCodeSD "tcp$number.tr"
  exec awk $awkCodeSrtt "tcp$number.tr"
  exec awk $awkCodeAck "tcp$number.tr"
  exec awk $awkCodeMark "tcp$number.tr"

  exec cat $ct >@ $cwndt  ;  exec cat $cv >@ $cwndv
  exec cat $st >@ $srttt  ;  exec cat $sv >@ $srttv
  exec cat $at >@ $ackt   ;  exec cat $av >@ $ackv
  exec cat $am >@ $mark   ;  exec cat $sdt >@ $sdft
  exec cat $sdv >@ $sdfv

  close $cwndt ; close $cwndv
  close $srttt ; close $srttv
  close $ackt  ; close $ackv
  close $sdft  ; close $sdfv
  close $mark
}
```

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
```

```
proc tracestart {} {
        global ns 1trace 2trace 3trace 4trace tcp1 tcp2 tcp3 tcp4

        set 1trace [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 $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 1trace 2trace 3trace 4trace

        # close trace-files
        $ns flush-trace
        close $1trace ;  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 $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
Agent/TCP set windowOption_ 8
Agent/TCPSink set windowOptionS_ 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

#    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
```

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 $p_{max} = 0.1$ (*$redq set linterm_ 10*) and one with $p_{max} = 0.2$ (*$redq set linterm_ 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 1trace 2trace 3trace 4trace tcp1 tcp2 tcp3 tcp4

        set 1trace [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 $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 1trace 2trace 3trace 4trace

        # close trace-files
        $ns flush-trace
        close $1trace ;  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 $node_(r1) $node_(d) 2Mb 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
 # 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_ 8
Agent/TCPSink set windowOptionS_ 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

#    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
```

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 $p_{max} = 0.1$ (*$redq set linterm_ 10*) and one with $p_{max} = 0.2$ (*$redq 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 {} {
        global ns 1trace 2trace 3trace 4trace tcp1 tcp2 tcp3 tcp4

        set 1trace [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 $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 1trace 2trace 3trace 4trace

        # close trace-files
        $ns flush-trace
        close $1trace ;  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_(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_(r1) $node_(d) 2Mb 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
 # 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*).

```
# 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 1trace 2trace 3trace 4trace tcp1 tcp2 tcp3 tcp4

        set 1trace [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 $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 1trace 2trace 3trace 4trace

        # close trace-files
        $ns flush-trace
        close $1trace ;  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_(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
 # 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
```

```
$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.3 Many New-ECN TCP-Reno Connections

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

```
# 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 9trace 10trace 11trace 12trace \
               13trace 14trace 15trace 16trace 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_ ;   $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_
        $tcp7 tracevar cwnd_ ;   $tcp7 tracevar srtt_ ;   $tcp7 tracevar ack_
        $tcp8 tracevar cwnd_ ;   $tcp8 tracevar srtt_ ;   $tcp8 tracevar ack_
        $tcp9 tracevar cwnd_ ;   $tcp9 tracevar srtt_ ;   $tcp9 tracevar ack_
        $tcp10 tracevar cwnd_ ;  $tcp10 tracevar srtt_ ;  $tcp10 tracevar ack_
        $tcp11 tracevar cwnd_ ;  $tcp11 tracevar srtt_ ;  $tcp11 tracevar ack_
        $tcp12 tracevar cwnd_ ;  $tcp12 tracevar srtt_ ;  $tcp12 tracevar ack_
        $tcp13 tracevar cwnd_ ;  $tcp13 tracevar srtt_ ;  $tcp13 tracevar ack_
        $tcp14 tracevar cwnd_ ;  $tcp14 tracevar srtt_ ;  $tcp14 tracevar ack_
        $tcp15 tracevar cwnd_ ;  $tcp15 tracevar srtt_ ;  $tcp15 tracevar ack_
        $tcp16 tracevar cwnd_ ;  $tcp16 tracevar srtt_ ;  $tcp16 tracevar ack_

        $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 {
```

```
 ...
}

# 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_(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]
set node_(s7) [$ns node]
set node_(s8) [$ns node]

#    create links
$ns duplex-link $node_(s1) $node_(r1) 10Mb 44ms DropTail
$ns duplex-link $node_(s2) $node_(r1) 10Mb 34ms DropTail
$ns duplex-link $node_(s3) $node_(r1) 10Mb 29ms DropTail
$ns duplex-link $node_(s4) $node_(r1) 10Mb 24ms DropTail
$ns duplex-link $node_(s5) $node_(r1) 10Mb 14ms DropTail
$ns duplex-link $node_(s6) $node_(r1) 10Mb 9ms DropTail
$ns duplex-link $node_(s7) $node_(r1) 10Mb 9ms DropTail
$ns duplex-link $node_(s8) $node_(r1) 10Mb 4ms 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 redq [[$ns link $node_(r1) $node_(d)] 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/TCPSink set windowOptionS_ 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_(s7) TCPSink $node_(d) 16]
$tcp9 set window_ 128
$tcp9 set ecn_ 1
$tcp9 set packetSize_ 512

set tcp10 [$ns create-connection TCP/Reno $node_(s1) TCPSink $node_(d) 8]
$tcp10 set window_ 128
$tcp10 set ecn_ 1
$tcp10 set packetSize_ 512

set tcp11 [$ns create-connection TCP/Reno $node_(s8) TCPSink $node_(d) 17]
$tcp11 set window_ 128
$tcp11 set ecn_ 1
$tcp11 set packetSize_ 512

set tcp12 [$ns create-connection TCP/Reno $node_(s2) TCPSink $node_(d) 9]
$tcp12 set window_ 128
$tcp12 set ecn_ 1
```

```
$tcp12 set packetSize_ 512

set tcp13 [$ns create-connection TCP/Reno $node_(s3) TCPSink $node_(d) 12]
$tcp13 set window_ 128
$tcp13 set ecn_ 1
$tcp13 set packetSize_ 512

set tcp14 [$ns create-connection TCP/Reno $node_(s4) TCPSink $node_(d) 13]
$tcp14 set window_ 128
$tcp14 set ecn_ 1
$tcp14 set packetSize_ 512

set tcp15 [$ns create-connection TCP/Reno $node_(s5) TCPSink $node_(d) 14]
$tcp15 set window_ 128
$tcp15 set ecn_ 1
$tcp15 set packetSize_ 512

set tcp16 [$ns create-connection TCP/Reno $node_(s6) TCPSink $node_(d) 15]
$tcp16 set window_ 128
$tcp16 set ecn_ 1
$tcp16 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]
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
$ns at 0.0 "tracestart"
$ns at 0.0 "$ftp1 start"
$ns at 0.25 "$ftp2 start"
$ns at 0.5 "$ftp3 start"
$ns at 0.75 "$ftp4 start"
$ns at 2.0 "$ftp5 start"
$ns at 2.25 "$ftp6 start"
$ns at 2.5 "$ftp7 start"
$ns at 2.75 "$ftp8 start"
$ns at 4.0 "$ftp9 start"
$ns at 4.25 "$ftp10 start"
$ns at 4.50 "$ftp11 start"
$ns at 4.75 "$ftp12 start"
$ns at 6.0 "$ftp13 start"
$ns at 6.25 "$ftp14 start"
$ns at 6.5 "$ftp15 start"
$ns at 6.75 "$ftp16 start"
$ns at 80.0 "finish"

# start simulation
$ns 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:

```
# 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 1trace tcp1

        set 1trace [open tcp1.tr w]
        $tcp1 tracevar cwnd_ ; $tcp1 tracevar srtt_ ; $tcp1 tracevar ack_
        $tcp1 attach $1trace
}

# 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 1trace qtrace

        # close trace-files
        $ns flush-trace
        close $1trace
        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]
```

```
set node_(s1) [$ns node]

#    create links
$ns duplex-link $node_(s1) $node_(r1) 10Mb 14ms DropTail

#    setup RED router
$ns duplex-link $node_(r1) $node_(d) 5Mb 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 set ecn_ 1
$tcp1 set windowOption_ 8
$tcp1 set windowOptionS_ 8
$tcp1 set packetSize_ 512

#    attach ftp traffic generators
set ftp1 [$tcp1 attach-app FTP]

# schedule events
$ns at 0.0 "tracestart"
$ns at 0.0 "$ftp1 start"
$ns at 60.0 "finish"

# start simulation
$ns run
```

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 (*$tcp1 set ecn_ 1*) and one with TCP-Reno (*$tcp1 set ecn_ 0*)

```
# 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 1trace tcp1

        set 1trace [open tcp1.tr w]
        $tcp1 tracevar cwnd_ ; $tcp1 tracevar srtt_ ; $tcp1 tracevar ack_
        $tcp1 attach $1trace
}
```

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

# end the simulation
proc finish {} {
        global ns 1trace

        # close trace-files
        #puts "in finish"
        $ns flush-trace
        close $1trace

        # create Matlab data
        makeplot "1"
        exit 0
}

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

#    create links
$ns duplex-link $node_(s1) $node_(r1) 10Mb 14ms DropTail

#    setup RED router
$ns duplex-link $node_(r1) $node_(d) 5Mb 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
 # one simulation was done with "set ecn_ 1" (TCP-Reno with ECN)
 # and one with "set ecn_ 0" (TCP-Reno without ECN)
$tcp1 set ecn_ 1
$tcp1 set windowOption_ 1
$tcp1 set windowOptionS_ 1
$tcp1 set packetSize_ 512

#    attach ftp traffic generators
set ftp1 [$tcp1 attach-app FTP]

# schedule events
$ns at 0.0 "tracestart"
```

```
$ns at 0.0 "$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 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_ ;   $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 $1trace ;   $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 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 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_ 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
```

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 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_ ;   $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 $1trace ;   $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 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 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
```

```
$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_ 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.

```
# 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_ ;   $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 $1trace ;   $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 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 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_ 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_ 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
```

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_ ;   $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 $1trace ;   $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 13ms 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 13ms 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 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 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]
```

```
$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
```