A study of TCP-RED congestion control using ns2

Arijit Ganguly Pasi Lassila

July 20, 2001

Contents

Chapter 1

The Network Simulator-ns2

1.1 Introduction

ns2 is an object-oriented simulator, written in C++, with an OTcl interpreter as the frontend. The simulator supports a class hierarchy in C++ (also called the compiled hierarchy), and a similar hierarchy within the OTcl interpreter (also called the interpreted hierarchy). The two hierarchies are closely related to each other, from the user's perspective, there is a one-to-one correspondence between a class in the interpreted hierarchy and one in the compiled hierarchy. The root of this hierarchy is the class TclObject. Users create new simulator objects through the interpreter, these object are instantiated within the interpreter, and are closely mirrored by a corresponding object object in the compiled hierarchy.

1.2 Why two languages?

ns2 uses two languages, because it has two things to do. On one hand, detailed simulations of protocols requires a system programming language which can efficiently manipulate bytes, packet headers, and implement algorithms that run over large data sets. For these tasks, run-time speed is important. C++ is slow to change, but its speed makes it suitable for protocol implementation.

On the other hand, a large part of network research involves slightly varying parameters and configurations, or exploring a number of scenarios. In these cases, iteration time (change the model and re-run) is more important. Since configuration runs once (at the beginning of the simulations), run-time of this part is less important. OTcl runs slower but can be changed very quickly making it ideal for simulation configuration.

1.3 Getting started

```
set ns [new Simulator]
#create a new simulator object

set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
#create nodes in the network
```

```
$ns duplex-link $n0 $n2 100k 10m DropTail
$ns duplex-link $n1 $n2 100k 10m DropTail
$ns duplex-link $n2 $n3 100k 10m RED

#link up the nodes in the network topology
#we specify for each point-to-point link the endpoints of the link,
#the bandwidth of the link, the propagation delay on the link, and finally
#the queue type associated with the link.

set tcp1[$ns create-connection TCP $n0 TCPSink $n3 1]
set tcp2[$ns create-connection TCP $n1 TCPSink $n3 2]

#set up a TCP agents at n0 and n1, set up TCPSink agent at node3
#connect the agents to each other

set ftp1[$tcp1 attach-app FTP]
set ftp2[$tcp2 attach-app FTP]
#run ftp as an appplication

$ns at 1.0 ‘‘$ftp1 start’’
$ns at 1.4 ‘‘$ftp2 start’’

$ns at 1.0 ‘‘$ns halt; exit 0’’
$ns run
```

To an ns2 script one just have type the following command:

```
ns filename args
```

1.4 Adding a new protocol to the ns2

Adding a new protocol to the ns2 is simple and can be checked up in the Marc Greis tutorial on ns2 homepage (http://www.isi.edu/nsnam/ns/).

Chapter 2

The RED-TCP interaction

Random Early Detection (RED) is an Active Queue Management (AQM) method introduced aiming at reducing the risk of global synchronization of TCP sources. RED also reduces bias against the bursty traffic. Currently, it is recommended as one of the mechanisms for AQM by IETF.

2.1 The TCP congestion control

The TCP congestion control consists of four intertwined algorithms: slow start, congestion avoidance, fast retransmit and fast recovery.

2.1.1 Slow Start and Congestion Avoidance

The slow start and congestion avoidance algorithms are used by a TCP sender to control the amount of outstanding data being injected into the network. To implement this, two state variables are added to TCP per-connection state. The congestion window (cwnd) is a sender-side limit on the amount of data the sender can transmit into the network before receiving an acknowledgement (ACK), while the receiver's advertised window (rwnd) is a receiver-side limit on the amount of outstanding data. The minimum of cwnd and rwnd governs the data transmission.

Another variable, the slow start threshold (sstresh), is used to determine wheteher slow start or congestion avoidance is used. When a TCP connection starts, it tries to probe the network capacity by using a maximum congestion window of one segment. The slow start algorithm is used when $cwnd < ssthresh$, else congestion avoidance is used.

During slow start, a TCP increments cwnd by one segment for every ACK received. The slow start ends when cwnd exceeds ssthresh, and congestion avoidance takes over. During congestion avoidance, cwnd is increased linearly, i.e. by 1 segment per round-trip time (RTT).

When a TCP sender detects segment loss using retransmission timer, the value of ssthresh is halved, and the cwnd is reset to 1 segment. The lost segment is retransmitted.

2.1.2 Fast Retransmit/Fast Recovery

A TCP-receiver immediately sends a duplicate ACK when an out-of-order segment arrives. In addition, a TCP receiver should also send an immediate ACK when an incoming segment fills in all or part of a gap in the sequence space.

The TCP-sender uses fast retransmit to detect and repair, loss based on incoming duplicate ACKs. After receiving 3 duplicate ACKs, TCP retransmits the missing segment without waiting for retransmission timer to expire.

After the fast retransmit, the fast recovery algorithm governs the transmission of new data until a non-duplicate ACK is received. After the receipt of third duplicate ACK, the value of ssthresh is halved and cwnd is set to $ssthresh + 3$ segemnts.

For each duplicate ACK received, the cwnd is inflated by 1 segment. A segment is transmitted if allowed by the new value of congestion window and receivers advertised window.

When the next ACK arrives that acknowledges new data, cwnd is set to ssthresh. This is termed as "deflating" the window.

2.2 The RED Algorithm

The ns2 RED implementation has been altered in the following mannner. The effect of computing the average queue length differently when the queue becomes idle is not considered. The counter measuring the no. of packets since the last drop is not used. The simplified RED used for the simulations works as follows.

Consider a queue with a finite buffer which can hold up to K packets at a time. Let q_n be the queue length and s_n EWA queue length (to be defined below) at the n^{th} arrival. Then the RED algorithm probabilistically drops packets in the following manner.

For each arriving packet we compute the EWA queue length s_n with

$$s_n = (1 - w)s_{n-1} + wq_n,$$

where s_{n-1} is the EWA queue length seen by the $(n-1)^{th}$ arrival, and q_n is the instantaneous queue lenth seen by the packet, and w is the weighting parameter. If the buffer is full, the packet is dropped. If $s_n < T_{min}$, the arriving packet is accepted. If $s_n > T_{max}$, the packet is discarded. However, if $T_{min} \leq s_n \leq T_{max}$, the packet is dropped with probability

$$p_n = pmax\frac{(s_n - T_{min})}{(T_{max} - T_{min})}.$$

Chapter 3

Implementation details

The behavior of the TCP-RED congestion control can be studied by analysing data about the arrival rate, average queue length and the instantaneous queue length, of a RED queue, at a bottleneck link.

This can be easily accomplished by sampling the RED state variables at regular intervals. But the data has to be collected, at the time of a packet arrival, a packet departure and a packet drop occur, in order to get a complete picture of the queue state.

3.1 Problems with ns2 trace utility

From the ns2 trace utility, it is possible to determine the arrivals, departures and the drops of packets at a queue, and then use postprocessing to find the queue lengths and other RED state variables, from these events. But this would mean, implementing RED controller again. The trace directly, doesn't provide any information about the queue state. It only provides a very general information about the packet events.

3.2 Changes made to the ns2 code for extracting queue state

The files that have been changed to extract the RED queue state are:

queue.h, queue.cc A new variable dumpcallback_ has been added to the Queue class, and bound this variable to a OTcl variable dumpcallback_. OTcl provides a bind function, that allows a variable of a OTcl class, and that of the compiled shadow class, to be bound together. It ensures that the values of the compiled variable and the interpreted variable are always consistent.

Another method dumpstate(), has been added to the Queue class. This dumps the information about the queue, into a user specified file. This method is automatically invoked at the time of packet arrival, departure and drop.

red.h, red.cc The dumpstate() method for RED dumps the average queue size, instantaneous queue length and the current time into the dumpfile.

drop-tail.h, drop-tail.cc The dumpstate() method is the same as in earlier case, except that there is no field for the average queue length.

ns-default.tcl The dumpcallback_ instance variable is set to a default value of false.

The dumpstate() method has to be defined to be virtual. Any class deriving from the Queue class must implement this method in order to be instantiated.

Note: The changes made to the ns2 code are just aimed at extracting the RED queue state.

3.3 Changes made to ns2 RED implementation

The model for RED proposed by Kuusela et al. in [1], is based on the implementation were the number of packets between two drops is a geometric random variable. But the ns2 RED implementation keeps track of the number of packets since the last drop, and uses this count to modify the drop probability, in effect making the inter-drop count a uniform random variable.

Therefore, changes have been made to the probability calculation functions of the REDQueue class, in the file red.cc. The functions ns2 uses to calculate drop probability are:

1. caclculate_p
2. modify_p

For the analysis, the system consists of:

1. RED queue holding the customers (packets) waiting for service (transmission).
2. A server which services the packets by transmitting them.

The information obtained from the packet enque and deque in ns2 does not give all the information about the system. The RED implementation in ns2 confines the system to the RED queue. As soon as a packet is dequed, it is considered to have left the system.

But the analysis require that a packet be considered inside the system even after its deque, until it is transmitted. This required another hack into the ns2 RED code.

The queue class has a state variable, blocked_, which tells whether a packet is in transmission After a packet is dequed, the queue is blocked. Any packets arriving at the queue, can be transmitted only when the queue is unblocked, i.e. when no packet is in trasmission. The ns2 accomplishes this, by scheduling, at the time of deque, an event that fires after the transmission time, and resumes the queue.

In the calculation of average queue size, the value of the blocked_ state variable has been used to find out the actual number of packets in the system.

3.4 Changes made to the TcpAgent

The following files have been modified so that slow start could be turned off and on from the ns2 script.

tcp.h tcp.cc A new class variable called arislowstart_, has been added to the class TcpAgent. This variable is bound to the OTcl variable with the same name. When this is set to true (from ns2 tcl script), the agent implements slow start, otherwise it doesn't implement slow start. The following methods that have been modified:

- opencwnd

- slowdown

ns-default.tcl The value of the variable arislowstart_ is set to true, so that by default the TcpAgent always implements slow start.

Note: The changes made to the TcpAgent will work fine only when the TcpAgent uses the default value for the windowOption_ variable. The default value for windowOption_ is 1.

3.5 Putting changes into effect

The modified C++ and OTcl files are available in

- /projekti/Akatemia/Comcom/Arijitwork/DynamicTCPRED.

Making the changes work requires the following steps:

1. Obtain a fresh ns-allinone package, unzip the package and install it in some directory.
2. Copy the C++ files into the directory ns-allinone/ns/.
3. Copy the OTcl files into the directory ns-allinone/ns/tcl/lib.
4. Run make in the /ns-allinone/ns directory.

3.6 Postprocessing of results with Python

For post-processing, a programming language called Python has been used. Python's easy-to-use features and powerful constructs make it an ideal language for post-processing. The version used for post-processing is the Python 2.1. Besides, all the post-processing scripts also require the Numeric module for Python to be installed. This module has to be installed separately as it does not accompany the Python 2.1 installation.

The topology parameters are passed to The OTcl script in form of command-line arguments, which include the link characteristics, RED parameters, TCP connection parameters, simulation time. The OTcl script dumps the entire queue state variables to the standard output.

All the ns2 simulations can be run through a single python program. This program reads the topology information from a file and invokes the ns2 simulation script from within. The output of the OTcl script, is read through a pipe, and post-processed, without generating any temporary trace files. The output of the python program is in form of the following ASCII files.

deq.out throughput

arr.out normalized arrival rate of packets

len.out instantaneous queue length

ave.out average queue length

These files contained the average queue size, instantaneous queue length, throughput and the normalized arrival rate of packets as functions of time.

3.7 Running the python program

Running the python program that in turn invokes the ns2 and post-processes the results, requires the following command:

```
./simulate [topology file] [no. of runs] [delta] [simulation time] [prefix]
```

The first argument to the program is the topology file. The second argument is the no. of simulations, followed by the size of the interval over which we want to average the simulation data. The fourth argument is the simulation time for the ns2. The last argument is the name of the directory in which the output files are created.

3.8 The topology file format

The Figure 3.1 shows the topology that has been realised in the simulations.

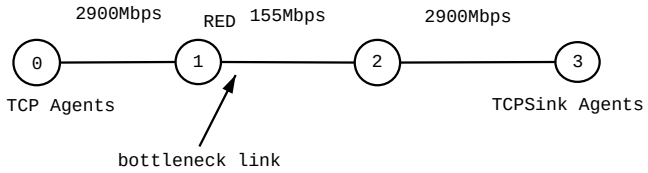

Figure 3.1: The Network Topology

The first two lines of the topology file specify the banwidths and the delay characteristics of the non-bottleneck links. The bandwidth value specified is, by default taken to be bps, but by adding suitable suffixes (M/m or K/k), bandwidth can also be specified in Mbps and Kbps respectively.

The next line specifies the queue length for all the non-bottleneck links. This is set to quite a high value, so that the other links don't run out of buffer space.

The next line specifies the bottleneck link characteristics, including the queue limit, for that link. The next line contains the various TCP parameters, the packet size in bytes for the TCP connections, the maximum senders window in packets, whether the connection should go back to slow start after a long idle period, and whether the TCP agent should use slow-start. This parameter takes a boolean value specified by true/false. But the last parameter is immaterial as in the simulations, the FTP sources always have data to send. So the idle periods do not occur. But a value has to be specified for this field. Tha parameter specifying the slow-start on/off also takes a boolean value specified by true/false.

Finally, the topology file contains the RED parameters, in the order min-threshold(T_{min}), max-threshold (T_{max}), weighting parameter (w) and the reciprocal of the maximum drop probability ($\frac{1}{pmax}$). There is another boolean (true/false) parameter that specifies whether RED should maintain an interval between drops.

Following is the sample of a topology file:

```
0 1 [b/w] delay

2 3 [b/w] delay

[non-bottleneck link queue limit]
```

```
[b/w for bottleneck link] [bottleneck-link delay] [bottleneck-link queue limit]
[no. of TCP connections]
[meanpktsize] [max_window] [slow-start restart after idle periods] [slow-start on]
[minthresh] [maxthresh] [w] [1/pmax] [maintain an interval between drops]
```

Chapter 4

Results and Conclusion

The simulations have been carried out for the topology as shown in Figure 4.1.

The following parameters have been fixed for all the simulations:

- non-bottleneck link queue limit = 1000
- $T_{max} = 50$
- $w = 0.002$
- $\frac{1}{pmax} = 5$
- meanpktsize = 1000 bytes, max_window = 40 segments

All other links use DropTail and have a queue limit of 1000. The TCP packet size was set to 1000 bytes. The average queue size, instantaneous queue length and the throughput were studied for different RED parameters, no. of connections, and round-trip times.

The graphs on the RED queue state, have been obtained by averaging over 100 simulation runs, with the simulation time being 5.0 msec.

The model proposed by Kuusela et al. in [1], assumes that the congestion window is always inflated linearly and TCP never resorts to slow-start. This required making changes to the methods for inflating and deflating the congestion window. The results shown are the ones from the simulations carried out without the slowstart and exponential rise. When TCP detects congestion it halves its congestion window, which it always inflates linearly.

So the results presented in this report are those obtained without the TCP slow-start. A comparison of results with and without slow-start is shown in a later section.

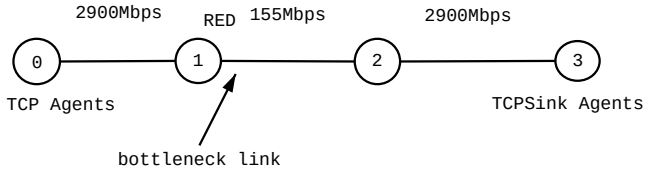

Figure 4.1: The Network Topology

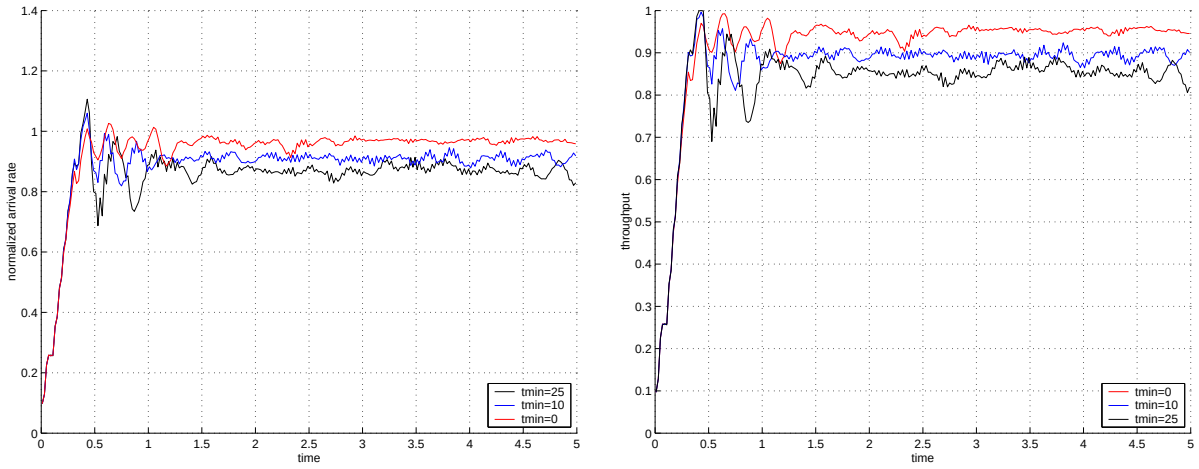

Figure 4.2: Effect of varying T_{min} on queue load and throughput

4.1 Tests on RED parameters

This section shows the effect of varying T_{min} on the behaviour of the RED queue. For the simulations, the TCP slow-start has been turned off and the various topology parameters are:

- link 0-1 5ms
- link 1-2 10ms
- link 2-3 5ms
- $T_{max} = 50$

The behaviour of RED has been studied using the topology of Figure 4.1, by varying the T_{min} for the RED queue. The results are shown in in the graphs (Figure 4.2 and 4.3).

Its clear from Figure 4.3 that the instantaneous and the average queue lengths increase when the T_{min} for the RED queue is increased. This is because increasing T_{min} allows room for more packets to come into the queue.

However, Figure 4.2 shows an unintuitive result. The increase in the T_{min} causes a decrease in the normalized arrival rate of packets at the congested link. The normalized arrival rate gives an estimate of the load on the queue. This result shows that even with a decreased load, a RED queue may function at a higher queue length. Besides, the throughput of the link also goes down as the T_{min} is increased, although the queue now functions at a higher length.

The results also show that the throughput behaves much the same as the load (normalized arrival rate) on the queue. Similarly, the behaviour of the average queue length is same as that of the instantaneous queue length. In the next section, a comparison of only the load and the instantaneous queue length is shown.

4.2 Tests on TCP parameters

This section shows the effect of varying the roud-trip time for the TCP connections on the behavior of the RED queue. The results have been shown for round-trip times of 40 msec and 8 msec. The results are shown in Figure 4.4 and 4.5.

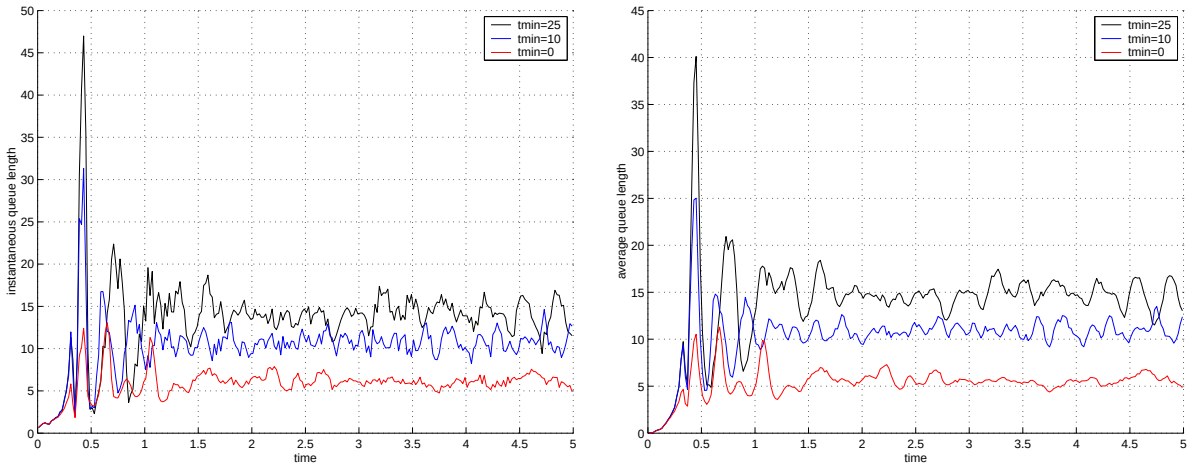

Figure 4.3: Effect of varying T_{min} on instantaneous and average queue length

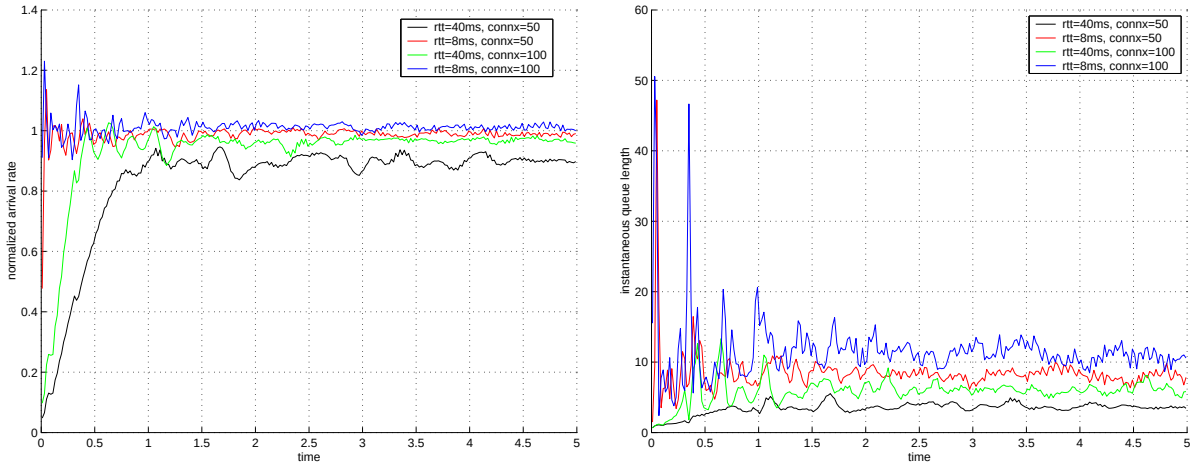

Figure 4.4: Effect of varying rtt on queue load and instantaneous queue length ($T_{min} = 0$)

The breakup for the round-trip times used in the simulations is as follows:

40 ms link 0-1 5msec, link 1-2 10msec, link 2-3 5msec

8 ms link 0-1 1sec, link 1-2 2msec, link 2-3 1msec

The results show that as the round-trip time for TCP connections decreases, the dynamics of the system tend to become faster. The load and the queue length attain their maximum values much faster than in a case where round-trip time is small. This is because, the response time of the TCP connections to the RED behaviour and vice-versa also decreases with small round-trip times.

It can also be observed that small round-trip times result in a higher load on the RED queue. As a result, the queue length also increases.

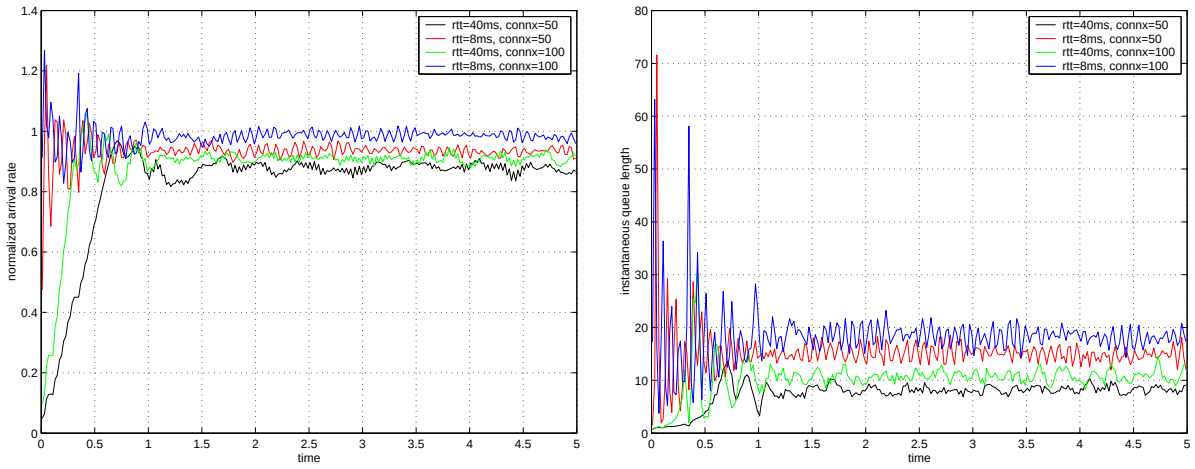

Figure 4.5: Effect of varying rtt on queue load and instantaneous queue length ($T_{min} = 10$)

4.3 Effect of slow-start

The results shown above are obtained from a TCP source without the slow-start feature. The TCP source halves its congestion window, whenever it detects congestion, which it always increases linearly.

The various parameters for the simulation are:

- link 0-1 5ms
- link 1-2 10ms
- link 2-3 5ms
- $T_{min} = 10$
- connx = 100

But actual TCP implements the slow-start feature, and hence this section compares the RED behaviour to the TCP sources implementing the slow-start with the one used in the simulations above.

The Figure 4.6 shows the normalized arrival rate and the instantaneous queue length with and without slow-start. In a system with slow-start feature turned on, it can be observed that an initial spike is observed in the load and the queue length. This spike however doesn't appear when the slow-start is turned off. If the initial spike is ignored, the dynamics of the system look much same.

The initial spike can be attributed to the exponential rise in the congestion windows of the TCP connections. This is followed by a fall in the load, as well as the queue length. This happens because RED hasn't yet been able to break the global synchronization of TCP sources. A large number of connections observe congestion and close down their congestion windows, resulting in lower load and queue length.

The graphs are similar, except for the initial spike, because TCP agents seldom resort to exponential rise, once they have already responded to congestion. This because a TCP agent can grow its congestion window exponentially only when its congestion window is below the slow-start threshold. This can occur, only when it goes into slow-start. The samples of TCP windows (Figure

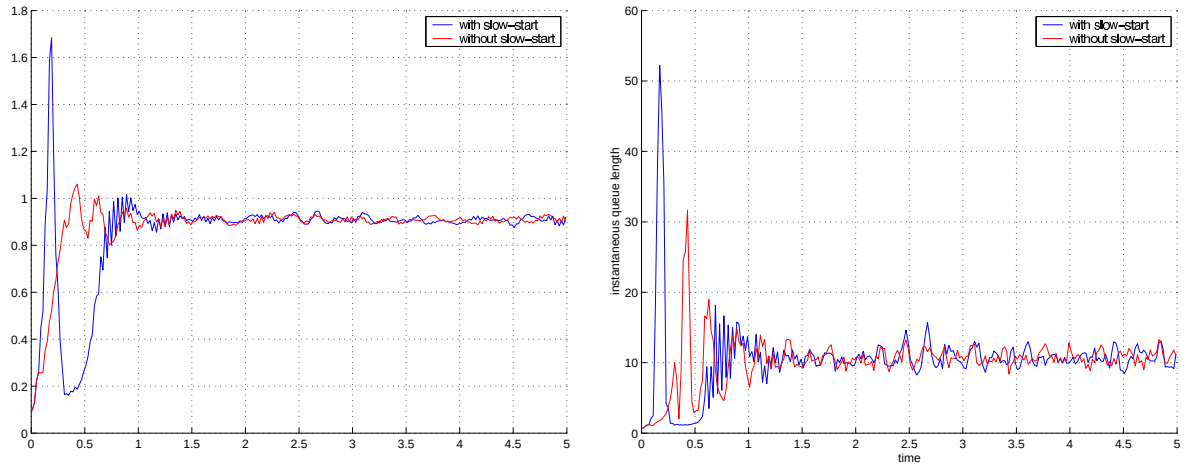

Figure 4.6: Effect of slow-start on queue load and instantaneous queue length ($T_{min} = 10$)

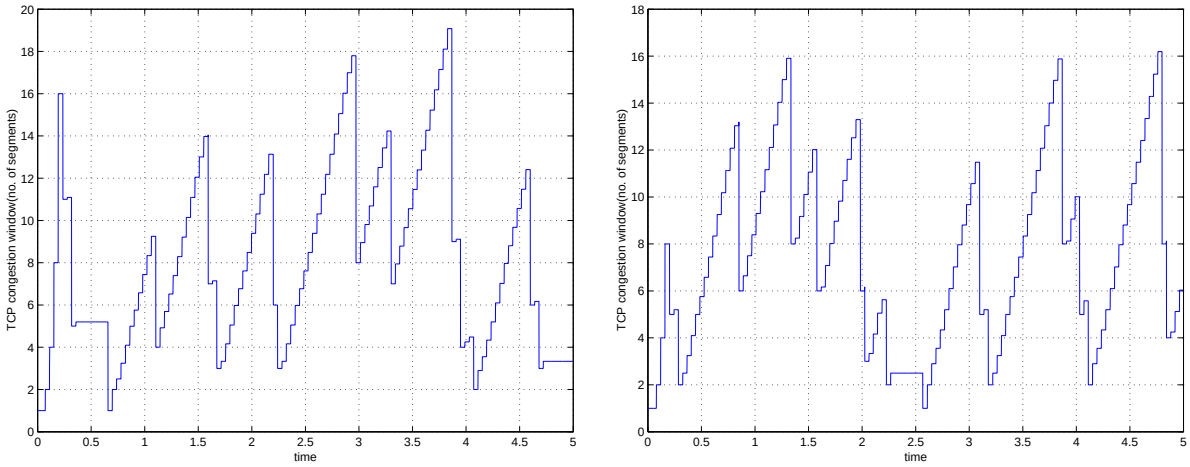

Figure 4.7: Congestion window samples from two TCP sources (with slow-start)

4.7) taken at a 0.001 second interval show that, there are very few slow-starts as the simulations proceed. In most of the cases, TCP just halves its congestion window when it detects congestion. The congestion window, doesn't fall below the slow-start threshold and the rise is always linear (congestion avoidance phase). This explains the similarity between the graphs obtained with and without slow-start.

4.4 Instability caused by very heavy loads

The RED algorithm has been designed for co-operative data sources, that detect and respond to congestion. Under normal loads, RED shows stable behaviour, keeping the queue at a fairly constant length. But under very heavy loads, wild oscillations can be observed in the queue length and the normalized arrival rate, as shown in the Figure 4.8.

The simulations use the following parameters:

- link 0-1 5ms

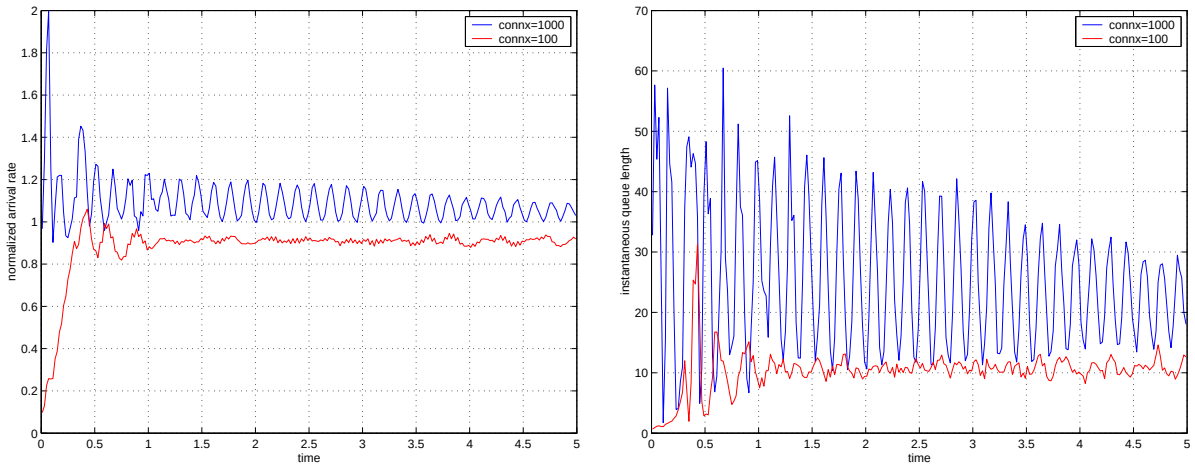

Figure 4.8: Effect of a heavy load on the RED-TCP interaction

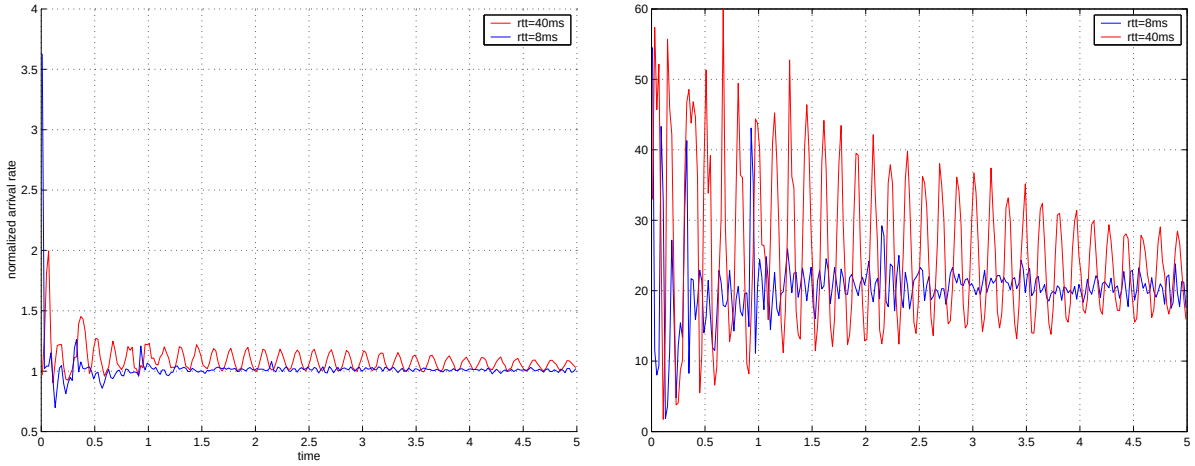

Figure 4.9: Effect of a round-trip time under heavy load

- link 1-2 10ms
- link 2-3 5ms
- $T_{min} = 10$
- connx = 100 or 1000

As the round-trip time is decreased from 40ms to 8ms, it can be observed from Figure 4.9 that the system behaves in a stable manner (shown for 1000 connections only).

Bibliography

[1] P. Kuusela, P. Lassila, J. Virtamo, P. Key, "Modeling RED with Idealized TCP Sources", in Proceedings of the 9^{th} IFIP Conference on Performance Modeling and Evaluation and ATM & IP Networks, Budapest, June 17-19, 2001, pp. 155–166.