PERFORMANCE ANALYSIS OF HIGH-SPEED TRANSPORT CONTROL PROTOCOLS

A Thesis

Presented to

the Graduate School of

Clemson University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Computer Science

---

by

Pankaj Sharma

August 2006

Advisor: Dr. Michele C. Weigle

ABSTRACT

TCP suffers from poor performance on high bandwidth delay product links. This is largely due to TCP's congestion control algorithm, which can be slow in taking advantage of large amounts of available bandwidth. A number of high-speed variants have been proposed recently, the major ones being HS-TCP, Scalable TCP, BIC-TCP, CUBIC, FAST and H-TCP. These protocols follow very different approaches in trying to improve the utilization of available bandwidth over that of TCP. We test the efficacy of these approaches on a number of metrics, such as the time taken to reach full link utilization and the time taken to recover from packet drops. We believe that the Internet of tomorrow will continue to have multiple high-speed TCP variants. It becomes important in such a scenario for the flows using one variant to be fair in sharing the available bandwidth with flows using other variants in order to avoid a congestion collapse. We test the new high-speed TCP proposals against each other to see how fairly they share bandwidth with competing flows using simulations. Our results show that the protocols with dynamic approaches to managing their sending rates perform much better in terms of fairness than those with static approaches.

ACKNOWLEDGEMENTS

I am deeply indebted to my advisor, Dr. Michele Weigle, for her help and guidance throughout the course of this study. I am also grateful to my parents for their love and support. I am what I am today only because of their efforts and sacrifices.

I thank Injong Rhee and Lisong Xu for the ns-2 source code for S-TCP, BIC-TCP, and CUBIC and for the use of simulation scripts [RX05]. I thank Tony Cui and Lachlan Andrew from the University of Melbourne's Centre for Ultra-Broadband Information Networks (CUBIN) for ns-2 source code for FAST [CA04], and I thank Douglas Leith *et al.* for the ns-2 source code for H-TCP [LS+05].

TABLE OF CONTENTS

Table of Contents (Continued)

Table of Contents (Continued)

LIST OF FIGURES

List of Figures (Continued)

List of Figures (Continued)

List of Figures (Continued)

List of Figures (Continued)

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The Internet has become an essential part of our everyday lives. It has redefined the way we communicate and lead our lives. TCP is the most widely-used transport protocol on the Internet. It has performed remarkably well over the past two decades with little change in its specification. It has proven to be robust in the face of changing and growing traffic demands and has helped the Internet to avoid a congestion collapse. Network bandwidth has been steadily growing over the past years. High-speed networks with large bandwidths have become increasingly available. Grid computing and research networks both employ high-speed networks. Scientists are connected to their colleagues in other locations via fast, long distance links. Distributed, collaborative applications for analyzing large data sets require a reliable and fast mechanism for distributing the data. Networks of connected research labs, such as the National LambdaRail project [NLR], are being developed to address this issue.

1.1 System Model

In order to discuss the problems that TCP has with high-speed networks, we first present a simple system model that we use to define the terms and metrics used in this work.

Figure 1: Simple Network

Figure 1 displays a simple computer network where node A is the sender and node B is the receiver. They are connected via a network link that has various characteristics such as link speed, propagation delay, round-trip time, throughput and bandwidth-delay product. These terms are explained below:

- The *link speed*, or *bandwidth*, of the link between node A and and B describes the number of bits that the link can transmit per second. We will describe the bandwidth in units of megabits per second (Mbps).
- *Propagation delay* is the time it takes for a bit to travel the link between node A to node B. This will be expressed in terms of milliseconds (ms).
- The *round-trip time* (RTT) is the time it takes for a packet to travel from node A to node B and for an acknowledgement (ACK) of its receipt to return to node A. The minimum amount of time for the RTT is twice the total propagation delay on the path from node A to node B. The RTT will also be expressed in terms of milliseconds (ms).
- *Throughput* is the sending rate, or the number of bits that a sender (node A, for example) transmits across the network per second. The throughput will be limited by

network characteristics such as the bandwidth and RTT as well as the congestion control algorithm employed at the sender. We will express this in terms of megabits per second (Mbps).

- *Link utilization* is the percentage of link bandwidth that is currently being used. It can be calculated by dividing the sum of throughput of all flows sharing the link by the link bandwidth.
- The *bandwidth-delay product* (BDP) for a network path is the maximum number of bytes that it can carry at any instant. With an acknowledgement-based protocol such as TCP, new packets may only be sent once ACKs for previous packets return to the sender, so the BDP depends on the flow's RTT, *i.e.* how fast new packets can leave the sender. The BDP is computed as the product of the RTT and the bandwidth. As an example, consider a path with bandwidth of 100 Mbps and an RTT of 100 ms. This will give the path a BDP of 10 megabits (Mb). Since the bandwidth on high-speed networks is large, so is the BDP.

1.2 TCP Congestion Control

TCP is an ACK-based protocol, in that every packet sent is acknowledged by the receiver sending an ACK back to the sender. Only when an ACK is received can new packets be sent into the network. One of the main responsibilities of TCP is congestion control. Congestion control is in charge of controlling the sending rate so as to not congest the network with excess traffic. The primary purpose of this module is to make the TCP flow as aggressive as possible and at the same time be fair in sharing the link

bandwidth with other competing flows. For this reason, TCP senders keep a congestion window, *cwnd*, which is the number of unacknowledged bytes allowed to be in the network in terms of packets. This roughly corresponds to the number of bytes in flight. In this way, TCP tracks the current share of a particular flow on a link. It manages this share by expanding or contracting the congestion window via its congestion control algorithm. A larger congestion window corresponds to a greater throughput for the flow. The congestion windows have to scale to increasingly higher values to fully utilize large network BDPs. TCP's congestion control algorithm is conservative in increasing the congestion window and therefore TCP is not able to fully utilize available bandwidth resulting in poor performance in high BDP networks. To understand why, we look at the congestion window control algorithm for TCP.

There are two main phases of TCP congestion control: slow-start and congestion avoidance. Every TCP connection begins in slow-start but switches to congestion avoidance after the first packet loss.

On the arrival of an ACK, the congestion window at time i+1, $cwnd_{i+1}$, is adjusted as follows:

$$cwnd_{i+1} = cwnd_i + 1 \quad \text{[slow-start]}$$

$$= cwnd_i + 1/cwnd_i \quad \text{[congestion avoidance]},$$

where $cwnd_i$ is the size of the congestion window at time i. Since roughly *cwnd* ACKs arrive at the sender in one RTT, the congestion window of a sender during slow-start will double each RTT, and the congestion window of a sender during congestion avoidance will increase by 1 packet each RTT. It can be observed that the congestion window

increases very quickly initially, in slow-start, and then increases slowly after a packet loss is detected, which is an indication of having surpassed full link utilization.

On discovering a packet loss at time i+1, the congestion window $cwnd_{i+1}$ is adjusted as follows:

$$cwnd_{i+1} = 0.5 * cwnd_i$$

where $cwnd_i$ is the size of the congestion window at time i.

This is an additive-increase/multiplicative-decrease (AIMD) algorithm. The congestion window increases in increments of 1 packet every RTT once the slow-start is exited, and decreases by half whenever a packet loss is detected.

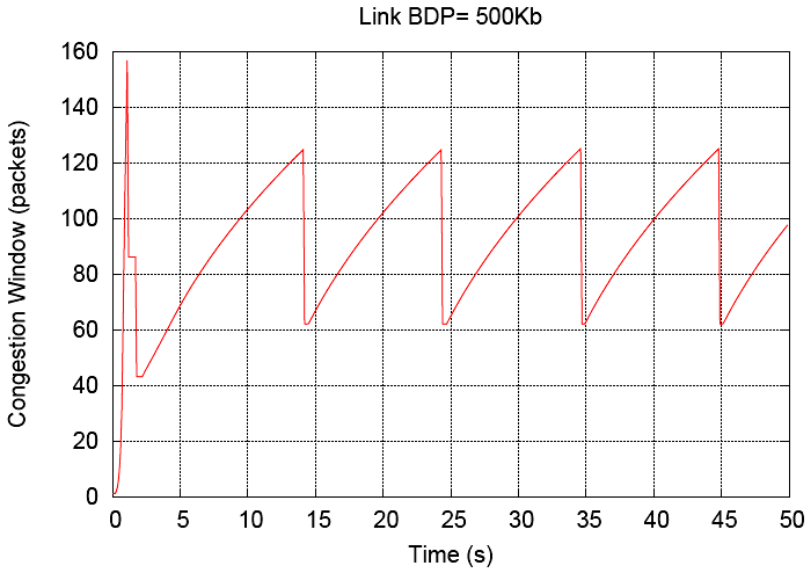

Figure 2: TCP performance on Low BDP link

To demonstrate this problem more clearly we ran two simulations in the ns-2 [[MF] network simulator, version 2.27. A network topology similar to the one presented in Figure 1 was used. The link connecting the sender, source, to the receiver, sink, was configured to have 100 ms RTT and 5 Mbps bandwidth. This configuration gave the

network path a BDP of 500 Kb. A TCP flow was run across this link and its congestion window was observed. In Figure 2 we show the congestion window of the sender over time. We can see that on a 500 Kb BDP path, after a packet drop the congestion window value drops by half and is able to get back to the maximum value relatively quickly, in less than 10 seconds.

In the second simulation, the bandwidth was increased to 622 Mbps resulting in a higher BDP of 62 Mb. The congestion window of the sender is shown in Figure 3. With a higher BDP, the congestion window value has greater space available for growth. To utilize the network efficiently, it must be able to be close to the maximum possible value for most of the time. The packet drop forces the window to reduce by half. It is unable to get back to the maximum in the remaining duration of the simulation. If another packet drop was to happen during this recovery phase then it would be set back further still. Therefore, there is a pressing need for making changes to TCP's congestion control algorithm to better utilize high BDP links.

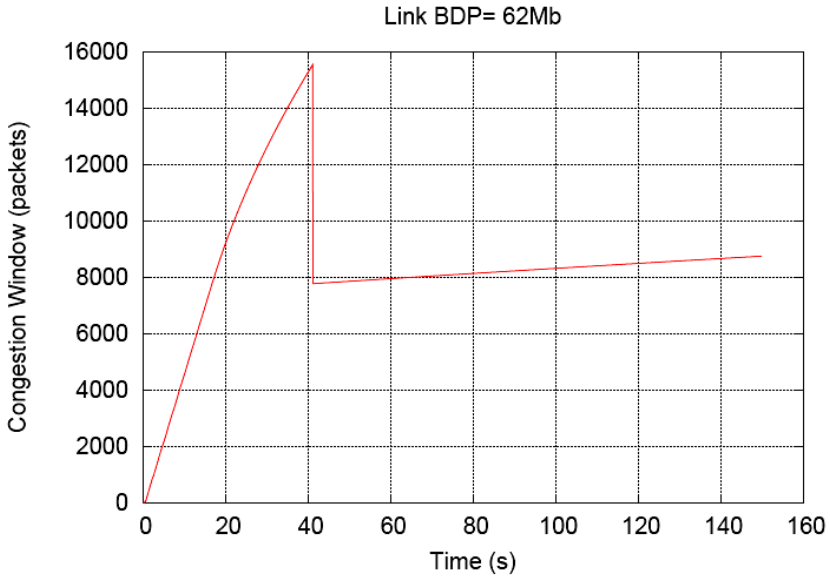

Figure 3: TCP performance on high BDP link

1.3 High-Speed TCPs

A way to deal with the conservativeness of TCP's increase algorithm is to make it more aggressive in garnering bandwidth and less aggressive in giving up bandwidth. A number of high-speed TCP proposals have been put forward in the recent past, the most prominent of which are High-Speed TCP (HS-TCP) [Flo03,10], Scalable TCP [Kel03], Binary Increase TCP (BIC-TCP) [XHR04], CUBIC [RX05], FAST [JWL04] and Hamilton TCP (H-TCP) [LLS05]. The basic aim of all these proposals is to obtain bandwidth faster than standard TCP. However, this aggressiveness has important implications for other flows sharing the network links. Fairness has become an important goal now primarily because all the high-speed protocols are very quick in taking up all the bandwidth and if fairness is not adequately addressed then it will lead to starvation of the less aggressive flows. RTT-fairness is another aspect that some of the protocol designers have concentrated upon. It is the fairness between flows having different RTTs. Since the congestion window is increased every time an ACK returns, flows with smaller RTTs will be able to increase their congestion windows faster than flows with longer RTTs. Competing flows rarely ever have similar RTTs and the differences in RTT affect the robustness of the flows.

The primary purpose of high speed networks is to transport large amounts of data as quickly as possible. On very high-speed links we expect there to be little congestion, so speed rather than congestion, is the primary concern. However, we want to make sure that if these protocols were used in an environment where congestion losses occurred

more frequently, they would still be responsive. We therefore focus on the speed and congestion avoidance abilities of the protocols. While most studies rely on the background traffic and other high speed flows on the network link to generate all the different situations that may arise on a typical network, this may not always be true owing to the great randomness associated with Internet traffic. Additionally, it is helpful to have tailored scenarios to test specific features of the protocols while trying to understand the implications of the protocol's congestion control algorithm design. This study specifically checks for protocol performance in the different scenarios like dropping packets in different stages of the congestion window growth, different bottleneck router queue sizes and different round trip times.

This study, based on simulations performed with the ns-2 network simulator, is directed towards developing a good understanding of the dynamics of the various high-speed transport protocols. It is an effort to understand the behavior of the protocols in simpler situations and based on that, to predict the behavior in more complex situations. We believe that the Internet of the future will not have just one high-speed TCP but a number of these will co-exist. In such a situation, it becomes imperative for the protocols to be fair to each other. We study the inter-protocol behavior with two flows from different protocols competing against each other after one of the flows has had the opportunity to obtain the full link bandwidth. We have previously published this portion of the study [WSF06]. In this thesis, we concentrate on the inter-protocol behavior after testing the protocols individually on different metrics. Some of the protocols have very different approaches in dealing with the same problems and we try to find which

approaches are better. The congestion control approaches of the protocols can be classified as being dynamic or static. Static approaches are those where fixed parameters are used for increasing or decreasing the congestion window, while dynamic approaches are those where the increase and decrease parameters are modeled on some variable(s) and thus depend upon the value(s) of those variable(s).

> **Thesis Statement:** High-speed transport control protocols with dynamic congestion-control algorithms have better performance compared to those that have static approaches. We measure performance in terms of fairness and robustness in different network scenarios.

The related work is presented in Chapter 2. The experiment setup and the tests that will be carried out are listed and explained in Chapter 3. In Chapter 4, a thorough explanation of the various high-speed protocols is provided along with results from the experiments for the individual protocol issues. The inter-protocol test results are in Chapter 5. We conclude our study in Chapter 6.

CHAPTER 2

RELATED WORK

The six high-speed TCPs under study will be introduced in this section and a much broader and an in-depth discussion will follow in Chapter 4. Previous work in evaluating high-speed TCPs will also be introduced and the unique points of this study will be highlighted.

2.1 Overview of High-Speed TCP Proposals

The high-speed protocols can be divided into two groups, loss-based i.e. those that depend upon packet-drops for detecting congestion, and delay-based i.e. those that depend on queuing delays to gather congestion information. All protocols except FAST are loss-based. The loss-based protocols can be further subdivided into two categories, those that depend on the previous congestion window value, *cwnd*, for calculating the next *cwnd* and those that depend on the time since last packet drop for finding the next *cwnd.* HS-TCP, Scalable-TCP and BIC-TCP fall in the first categorization and H-TCP and CUBIC fall in the second categorization. Below we briefly describe the high-speed TCPs.

2.1.1 High-Speed TCP (HS-TCP)

HS-TCP [Flo03, FRS02] interprets high loss-rate networks as legacy low BDP networks and low loss-rate networks as high BDP networks. It strives to behave more aggressively than TCP in low loss-rate environments and becomes quite passive in high loss rate environments for compatibility with legacy networks. Additionally, it uses TCP's congestion control algorithm below a threshold value, referred to as *LowWindow,* for backward compatibility. It uses lookup tables for determining the values of the increase and decrease parameters.

2.1.2 Scalable-TCP

Scalable-TCP [Kel03] does away with HS-TCP's table driven approach and switches to fixed increase and decrease parameters. It uses TCP's congestion control algorithm below *LowWindow,* similar to the approach followed by HS-TCP.

2.1.3 Binary Increase TCP (BIC-TCP)

RTT-fairness was one of the major goals in the design of BIC-TCP [XHR04]. It makes an estimate of the maximum link capacity and increases the congestion window, *cwnd* to that value using a binary search strategy. After attaining the maximum value, it does a slow start, *Max Probing,* until it sees a loss. It uses a *Fast Convergence* strategy to converge quickly to a fair share with the other flows. In this, a BIC-TCP flow reduces its

estimate of the maximum link capacity if it finds that its share of the bandwidth is going down and thus becomes gentler.

2.1.4 CUBIC

CUBIC [RX05] is a modification of BIC-TCP and it tries to improve on BIC-TCP's fairness. A cubic window increase function is used as opposed to the binary search strategy employed by BIC-TCP. This makes the *cwnd* increases less aggressive. Another major change is that the congestion window increases are based on the time since the last congestion event rather than on previous congestion window value.

2.1.5 FAST TCP

FAST TCP [JWL04] is a delay-based protocol which monitors the flow's RTT continuously to detect the level of congestion in the network. FAST updates the congestion window, *cwnd,* every other RTT depending on the RTT observed and α, which is the number of packets it tries to maintain in the router queues. It increases *cwnd* very aggressively as long as the RTT remains at the observed minimum value. As congestion builds up, the router queues begin to fill up leading to larger RTTs. FAST then switches to a less aggressive mode. FAST uses TCP's congestion control mechanism to handle packet drops.

2.1.6 Hamilton TCP (H-TCP)

H-TCP [LLS05] aims to be RTT-fair by using RTT-scaling. It increases congestion window based on the time between successive congestion events and the ratio of the minimum-observed RTT to the maximum-observed RTT. It adopts an adaptive back-off strategy when it sees a loss and reduces its congestion window based on the ratio of the minimum and maximum observed RTT.

2.2 Previous Evaluation Studies

The research community has been very active in evaluating the performance of the various high-speed TCPs. There has however been no clear consensus on which protocols perform the best. We believe that there will be no clear winner in the near future amongst the high-speed TCPs and that they will continue to co-exist. Inter-protocol fairness is therefore a very important aspect, yet most of the previous studies have ignored it. Bullot *et al.* [BCH03] did look at inter-protocol fairness but they start the competing high-speed TCP flows at the same time which is a highly unlikely scenario. The previous work has used simulations, testbeds and limited live networks to evaluate the protocols. We summarize this work below.

2.2.1 Simulation - Xu *et al.*

Xu *et al.* [XHR04] proposed BIC-TCP and conducted a performance comparison against HS-TCP, Scalable-TCP and an additive-increase/multiplicative-decrease (AIMD) algorithm with the increase and decrease parameters set to 32 and 0.125, respectively. They have run their simulations in ns-2 on a simple dumbbell network, which included both forward and reverse path traffic. They simulate their background traffic as a mixture of web traffic and short-term and long-term TCP flows. They measure network utilization, RTT fairness and TCP- friendliness in a DropTail as well as RED environment. DropTail and RED are different router queue management schemes. The results show BIC-TCP as the best performer in all the scenarios. The experiments run are very basic, and the background traffic generated is not a realistic estimation of true background traffic on a high-speed network link. The authors did not consider inter-protocol fairness issues. They have used BDP-sized router buffers which are too large for actual routers. The study does not cover a number of other major high speed TCP variants like H-TCP and FAST.

2.2.2 Live Network – Bullot *et al.*

Bullot *et al*. [BCH03] have done their tests on actual network paths and have used different Linux kernel versions patched for the different high-speed TCPs. Li *et al*. [LLS05] have pointed out that the performance variations may be caused by different Linux kernel versions rather than just the different congestion control algorithms.

Therefore, the results in the Bullot paper might not depict the true picture. In our study however, only the congestion control algorithm differs across the high-speed TCP implementations since the same version of *ns* has been used for all the protocols. In addition to this, even though Bullot *et al.* have run inter-protocol fairness where they use the same start time for all the competing flows. We stagger the start times in our study so that one flow is sending at a high data rate before another high-speed flow enters the network, which is much more realistic. They have also looked at the effects of reverse path congestion on the flow's throughput on the forward path. High-speed protocols like FAST that depend on RTT for their congestion window control are affected the most by this. Recent additions to FAST allow the protocol to change its congestion control depending on one-way transit times instead of round trip times to neutralize this weakness [CTF]. One of Bullot *et al.*'s main results is that a large RTT accentuates the performance difference of the high-speed TCPs. Their results show BIC-TCP to have the best behavior amongst all tested protocols. They also found Scalable-TCP to be too aggressive and HS-TCP to be too gentle. They also find Scalable-TCP to be unfair at smaller RTTs (<10ms). They calculate the asymmetry metric and then present the average of this measure for all the protocols. The asymmetry metric is defined as $(x_1 - x_2) / (x_1 + x_2)$, where x_i is the average throughput of flow *i*. The value of this metric ranges from -1 to 1 with 0 signifying perfectly fair sharing. A value close to 1 indicates that the first flow is more aggressive while a value close to -1 means that flow 2 was more aggressive. The average of these values can be quite misleading as we see from our own experiments. A flow can be very aggressive against one protocol and very gentle with another leading to a good average fairness, which is however not the case. In our

study, we have therefore not considered the average of the asymmetric metric but instead rank the protocols based on the sum of the absolute value. Their study is informative but falls short in providing a true picture of how all these high speed protocols measure up against each other when run together. Their background traffic solely consists of UDP which does not realistically compare with real-network traffic.

2.2.3 Testbed – Li *et al.*

Li *et al.* [LLS05] have done a performance evaluation of Scalable-TCP, FAST, HS-TCP, H-TCP and BIC-TCP. The study is based on protocol runs on an experimental testbed and emphasizes the need for using identical network stack implementations for all the high-speed protocols. They have run a wide variety of tests, where they measure fairness, efficiency, packet drops and backward compatibility. They found FAST and Scalable-TCP to have very poor fairness in a simple dumbbell topology where the competing flows had the same RTTs. They also found H-TCP to have the best RTT-fairness. Convergence times were the best for H-TCP and FAST after the start of a new flow. FAST however diverged later on. The study misses out on doing inter-protocol comparisons. They do however look at flow fairness when the competing flows have different RTTs and when the router queue sizes vary.

2.2.4 Testbed – Low *et al.*

In [HLW+04], Low *et al.* test FAST against Scalable-TCP, HS-TCP, H-TCP, BIC-TCP and TCP-Reno. The protocols have been tested on an experimental testbed employing Dummynet to control various network characteristics like bandwidth and delay. The emphasis of the study is on flow throughput, queue occupancy and packet drops as a function of the maximum available router queue size. The study does not look at protocol fairness and behavior in the face of background traffic on the forward path. It shows that all the loss-based protocols fill up the queue at the bottleneck router since they use packet drops as a signal of congestion. FAST however maintains a steady queue since it depends on RTT variation for detecting congestion. The study also studies a modified form of FAST, FAST-BQD that improves FAST's performance in reverse path traffic scenarios. FAST-BQD performs acceptably well in a reverse path traffic environment but still lags behind the other high-speed protocols. Strangely, FAST-BQD is not used for any tests other than the reverse path traffic test. A very simple traffic mix of TCP Reno streams is used.

Low *et al.* perform another study [JWL04] on the performance of FAST in comparison to Scalable-TCP, HS-TCP and TCP-Reno. The tests have been done with Dummynet on the router and Linux TCP implementations on the sender and receiver. They have run multiple experiments with different sets of RTTs and different number of flows from same protocols and show FAST to have best performance in terms of

throughput, responsiveness, stability and fairness. Background traffic is not considered and inter-protocol issues are not addressed.

2.2.5 Testbed – Ha *et al.*

Recently, Ha *et al*. [HKL+06] have performed an extensive study in which they have included background traffic to create a dynamic network environment as opposed to a static network environment. They argue that the addition of background traffic results in a fluctuation of network bandwidth which is a better model of a real world link. It also removes the phase effect *i.e.* the extreme loss synchronization of network flows. They also added background traffic on the reverse path to simulate ACK-compression. They have measured a number of metrics, including utilization, fairness, stability and time to convergence. They run their experiments in a lab with iperf to generate the network flows and dummynet to enforce the network parameters. They use a modified form of Linux 2.6.13. They have capped the maximum buffer space on the routers to 2MB which is much less (12%) than the actual link BDP in some of the experiments. They use iperf to generate long term background flows. For the short term flows, they use a log-normal (body) and a Pareto (tail) distribution for file sizes and an exponential distribution for inter-arrival times. One of their main conclusions is that H-TCP, Scalable-TCP and FAST have significantly lower network utilization in most situations than BIC-TCP and CUBIC. They conjecture that flows that have a convex congestion window curve i.e. increase sharply initially and become less aggressive afterwards as they near link capacity, have the best utilization. FAST also has a convex congestion window curve but

still behaves poorly because of its dependency on RTT. They reason that as the flow throughput approaches its maximum possible value, then because of the convex increase curve, the increments in the congestion window become smaller and smaller and result in a very small overflow at the bottleneck buffer when the maximum capacity is breached. On the other hand, protocols with a concave congestion window curve have bigger and bigger increments as the maximum throughput is approached and the overflow is much more, which results in a lot of other flows also experiencing losses. For their fairness study, they have only looked at intra-protocol fairness and have found that all the protocols except FAST show good fairness after the addition of background traffic. The strength of the paper lies in the emphasis it places on dynamic network conditions rather that static network conditions. The study has a major flaw in that it injects enough background traffic to fill up the entire network link on its own in the absence of any high speed flow. This assumption goes against previous studies which have showed that network utilization on backbone/high-speed links seldom increases beyond 50% [FMD+01]. With a lower level of background traffic most of the interesting results might not hold true. Another weakness in the paper is that it has only tested in scenarios where protocols with a convex increase function perform well. We argue that when a convex increase protocol sees a packet drops very early on, it stabilizes pre-maturely and performs poorly. We test this later in our study.

CHAPTER 3

EXPERIMENT SETUP

The objective of the experiments is to first study the features of the high-speed protocols individually and then to correlate their behavior when they compete against each other with their specific implementations. We try to discover the implementation features that result in better performances.

There is no randomness involved in the simulations. We use a very simple model: two nodes, one or two flows traveling in the same direction, and no background traffic. The RTTs of the flows are fixed based on the propagation delay on the link, and the starting times of the flows are also fixed. The goal is to study the mechanisms of the high-speed TCP proposals in isolation.

We first test the protocols individually to test the efficacy of the different aspects of their congestion control algorithms. We note the time they take (i) to grow to full link utilization and (ii) to recover from forced packet losses during their first ascent. We also examine their behavior at different link delays and bottleneck router queue sizes. These are discussed in Chapter 4 and listed in Section 3.5.

We believe that there will always be multiple variants of high-speed TCP around. In such a scenario it is necessary to test the protocols against each other to see the robustness of their approaches. These experiments are presented in Chapter 5.

All the experiments have been run in the *ns-2* network simulator [[MF] version 2.27. *Ns-2* is a simulation tool that lets one simulate networks of various topologies and characteristics. It is widely used in the research community. A large number of protocols including TCP have been implemented in it. Most of the new protocols, including the ones that we will be testing, also have *ns-2* implementations. Ns-2 implementations of Scalable-TCP, BIC-TCP and CUBIC from [RX05], FAST from [CA04] and H-TCP from [LS+05] have been used in this study.

3.1 Network Topologies

Three types of topologies have been employed for the simulations. They are presented below.

3.1.1 Network Topology 1

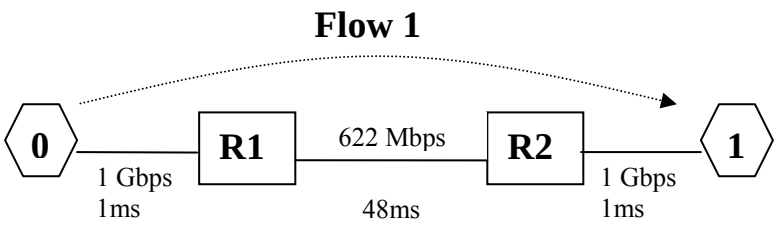

Figure 4: Network Topology 1

Network topology 1 is pictured in Figure 4. Node *0* is the source (sender) and node *1* is the sink (receiver). Each end node is connected to a router by a 1 Gbps link with a propagation delay of 1 ms. Link *R1-R2* is the bottleneck with a 622 Mbps link speed and 48 ms propagation delay. This topology results in a 100 ms roundtrip time (RTT). The network has a bandwidth-delay product (BDP) of 7775 1000-byte segments, and drop-tail queuing is used at both routers. In traditional networks, the rule of thumb for setting router queue size is to set it to be equal to BDP [VS94] but in high-speed networks, 20% of BDP is often used [HLW+04]. To ensure that TCP window size is not a limiting factor, each TCP connection has a maximum window size of 67,000 segments, which is about 64 MB. Each simulation is run for 500 seconds.

3.1.2 Network Topology 2

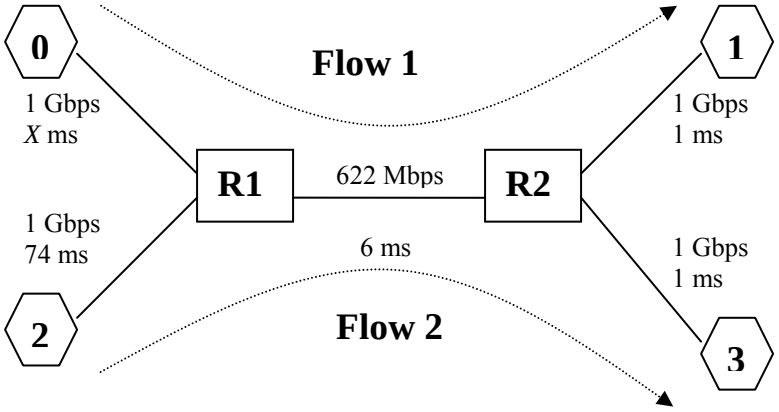

Figure 5: Network Topology 2

In the second topology, shown in Figure 5, nodes *0* and *2* are the sources, and nodes *1* and *3* are the sinks. Each end node is connected to a router by a 1 Gbps link with

different propagation delays. In this topology, there are two flows, one from node 0 to node 1 and another from node 2 to node 3. Link *R1-R2* is the bottleneck with a delay of 6 ms and link speed of 622 Mbps. Flow 2 has a constant RTT of 162 ms while the RTT of Flow 1 is varied from 16 ms to 162 ms. This is done by varying *X,* which is the delay for link *0-R1.* This topology is used for the RTT-fairness study. The flow with the longer RTT, flow 2, is always started first and the flow with the shorter RTT, flow 1, is started 50 seconds after that. Simulations were also done with the longer flow starting second, but the results were similar for all protocols except Scalable-TCP. The BDP for the network is 12,596 1000-byte packets. The router queue for *R1* is again limited to 20% of this value *i.e.*, 2519 packets.

3.1.3 Network Topology 3

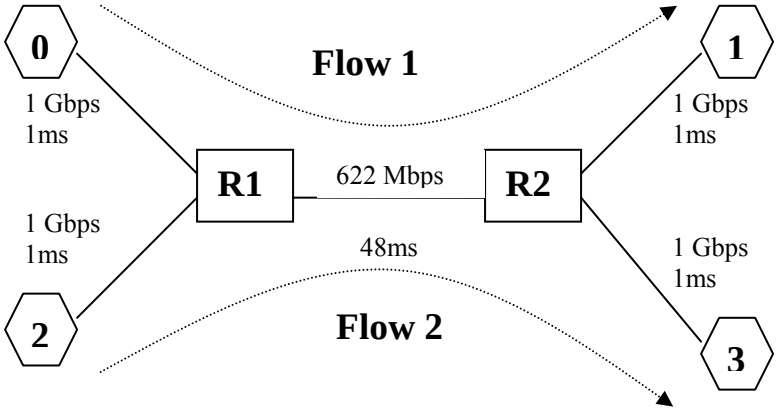

Figure 6: Network Topology 3

In the third topology, shown in Figure 6, link R1-R2 is the bottleneck and has a delay of 48ms with a bandwidth of 622 ms. Each end node is connected to a router by a 1

Gbps link with 1ms propagation delay. There are two flows, one from node 0 to node 1, and another from node 2 to node 3. Each sender has 100 ms RTT. The network has a BDP of 7775 1000-byte segments, and drop-tail queuing is used at both routers. Even though using the same RTT for both flows could cause synchronized loss, we were more concerned with factoring out the effects of RTTs on our results. Previous work [XHR04] has shown that many of the high-speed protocols are not RTT-fair, meaning that flows with shorter RTTs achieve higher throughput than flows with longer RTTs. In order to concentrate on the effects of competing flows, we chose to equalize the RTTs.

3.2 Performance Evaluation

We test the protocols for the following metrics:

1. The maximum throughput the protocol can obtain on an empty link
2. The time it takes for the protocol to go from zero to its maximum throughput on an empty link
3. The time it takes for the protocol to recover from a packet drop during its first ascent
4. The effect of router buffer values on the protocol's link utilization
5. The effect of different RTTs on the robustness of a protocol
6. The effect of different RTTs on intra-protocol fairness
7. Inter-protocol fairness

It is necessary to differentiate the cases where there is a packet drop during the first ascent and packet drops during other stages since some of the protocols, like BIC-TCP, depend heavily on the first packet drop to estimate the available capacity.

We use the asymmetry metric from Bullot *et al.* [BCH03] to evaluate fairness. This metric, as opposed to the Chiu and Jain fairness index [CJ89], provides additional information about which is the more aggressive flow rather than just conveying the level of fairness in sharing. The asymmetry metric is defined as:

$$A = (x_1 - x_2) / (x_1 + x_2)$$

where x_i is the average throughput obtained for flow i.

Average throughput was measured starting at time 250 seconds to focus on steady-state throughput. The closer the asymmetry metric A is to 0, the fairer is the distribution of link bandwidth. Value of A closer to 1 indicates the dominance of flow 1 while a value of A close to -1 indicates the dominance of flow 2.

CHAPTER 4

INTRA-PROTOCOL SIMULATIONS

In this chapter, the high-speed TCP protocols we studied are introduced, and an in-depth study is conducted to understand their various features. All of these protocols are sender-side alterations of TCP and are loss-based, with the exception of FAST which is delay-based. Congestion results in increased RTTs as the router queues fill up. When the queues overflow, packets get dropped. Loss-based protocols are those that depend on packet drops to detect congestion. On the other hand, delay-based protocols depend on the RTT variations to detect the congestion levels. Some of the protocols use very different forms of congestion control algorithms, and it is essential to test the efficacy of their approaches. We have tested the protocols individually and have tried to understand their behavior under different circumstances. The following sets of experiments have been conducted for this purpose:

1. Single flow, no enforced packet drops
2. Single flow, single packet drop
3. Single flow, different queue buffer sizes
4. Single flow, different RTT
5. Two flows of the same protocol, different RTTs

The purpose of these experiments is explained below:

1. *Single flow, no enforced packet* drops: This shows us how fast the protocol approaches link capacity and how stable it is at that optimum level. Topology 1 (Figure 4) has been used for these simulations.

2. *Single flow, single packet* drop: The protocols should be able to handle packet drops efficiently by making a quick recovery to full link utilization. We test the speed of recovery with this set of experiments. A packet is artificially dropped during the ascent phase of the congestion window growth before the congestion window reaches its maximum value. Recovery from a burst of drops is not examined as that is handled efficiently by TCP-SACK which has been used in all the simulations. Network topology 1 (Figure 4) has been used for these simulations.

3. *Single flow, different queue buffer* sizes: The buffer size for bottleneck router queues should be equal to at least the Bandwidth Delay Product (BDP) for 100% utilization during congestion [VS94]. This is because of TCP's back off factor of 0.5 that works in emptying the queue when it becomes full and overflows. With the router queue size equal to BDP, the router will always have something to send and hence the link utilization would be 100%. This setting is however not always optimal. The first reason is that the RTTs of the flows sharing the link differ leading to different BDP values for different flows. Secondly, with high BDP networks, there is an additional performance issue since such a provisioning would lead to extremely large queue size leading to vastly varying RTTs. Thirdly, it is expensive to have large router buffers. In our simulations with our network parameters, the BDP is:
   BDP = RTT x Link Bandwidth
   = 100ms * 622Mbps
   = 10^5 * 622 bits
   = 7775 packets, where 1 packet = 1000 Bytes
   As can be seen, this value is very large. It is expensive for router manufacturers to have such large buffers. It is therefore often advantageous to have smaller buffers at routers in high BDP environments [AKM04]. We test the performance of the different protocols with different router buffer sizes. Network topology 1 (Figure 4) has been used for these simulations. The router buffer at *R1* is varied from 40 to 7776 packets.

4. *Single flow, different* RTT: The increments to the congestion window are driven by the arrival of ACKs. We suspect that the effect of longer RTTs might be more pronounced for some of the protocols and test that with this set of simulations. Network topology 1 (Figure 4) has been used for these simulations with a slight variation wherein the delay of link R1-R2 is varied from 10ms to 110ms to have a different RTT for each run.

5. *Two flows of the same protocol, different* RTTs: RTT-fairness [XHR04] is a very important feature that has often been neglected in the design of transport protocols. Competing flows usually have widely varying RTTs, and the share of a flow is inversely dependent on its RTT in the absence of explicit RTT-fairness enforcement mechanisms. This is because the congestion window is usually incremented when the ACKs arrive, and the ACK arrival rate is dependent on the RTT. So with longer RTTs, the ACKs arrive at a slower rate resulting in slower increments for the farther hosts and faster increments for the nearer hosts. Here we test how well the protocols deal with the issue of the competing flows having different RTTs. A base RTT of

162ms is used for one of the flows, and the other flow's RTT is varied from 16ms to 162ms. The fairness might vary substantially in different network topologies but this still conveys meaningful information about how much variation in RTT the protocols can handle. Network topology 2 (Figure 5) has been used for these simulations.

The sections that follow include a description of the features of the protocol followed by the experiment results and their explanations. The results are summarized in Section 4.7.

4.1 HS-TCP

HighSpeed TCP (HS-TCP) is a high-speed TCP protocol developed by Sally Floyd [Flo03, FRS02]. It is one of the first major high-speed TCP protocols to have been proposed. The major goal is to make an HS-TCP flow more aggressive without any explicit feedback from either the link routers or the receiver. HS-TCP associates low loss rates, lower than 10^{-2}, with a high bandwidth environment and therefore strives to be more aggressive than TCP during times of low loss. HS-TCP is also designed to be fair to TCP at medium to high loss rates, greater than 10^{-2}. When the congestion window is lower than a minimum threshold value *LowWindow*, HS-TCP behaves the same as standard TCP. When the congestion window is above *LowWindow*, HS-TCP uses a modified response function. The standard TCP response function is $1.2/\sqrt{p}$ where p is the loss rate. With the appropriate parameters, HS-TCP's response function is $0.12/ (p^{0.835})$. We can observe that this is more aggressive than TCP's response function for p less than 10^{-3}. This function has been translated into increase and decrease parameters that are introduced in the next subsection.

4.1.1 Congestion Control Algorithm

In this discussion, we use the following terminology:

cwnd : Current congestion window

LowWindow: Congestion window threshold below which behavior is identical to TCP

a(cwnd) : Increase parameter

b(cwnd) : Decrease parameter

On the arrival of an ACK:

cwnd = *cwnd* + 1 *cwnd* <= *LowWindow* and slow start

= *cwnd* + 1/*cwnd* *cwnd* <= *LowWindow* and congestion avoidance

= *cwnd* + *a(cwnd)*/*cwnd* *cwnd* > *LowWindow*

On a packet loss:

cwnd = 0.5 *cwnd* *cwnd* <= *LowWindow*

= (1−*b(cwnd)*) *cwnd* *cwnd* > *LowWindow*

This algorithm is an additive increase, multiplicative decrease (AIMD) algorithm with *a(cwnd)*and *b(cwnd)* as the increase and decrease parameters. Current implementations of HS-TCP use a lookup table to determine the values of *a(cwnd)* and *b(cwnd)*. Recommended settings allow *a(cwnd)* in the range of [1, 72] segments and *b(cwnd)* in the range [0.1, 0.5]. *LowWindow* has been set to 38 in our experiments.

4.1.2 Unique Points

1. The algorithm uses table-based increase and decrease parameters. This complicates the implementation but allows the algorithm to have a layered approach.
2. HS-TCP is aggressive only in low-loss rate environments. In this aspect it is similar to H-TCP which also becomes less aggressive if the loss rate is high.

4.1.3 Examples of the Protocol Operation

A number of simulations were run to test the various features of HS-TCP. Network topology 1 (Figure 4) was used for the single flow simulations, and network topology 2 (Figure 5) was used for the two flows with different RTT simulations. The results are presented below.

4.1.3.1 Single Flow, No Enforced Packet Drops

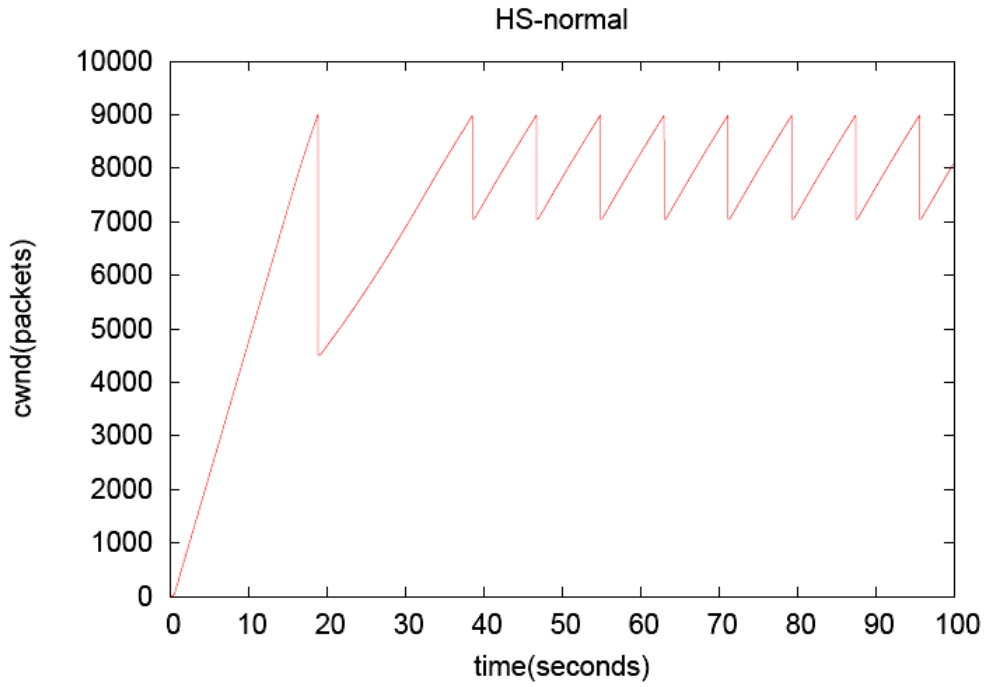

Figure 7: HS-TCP normal graph

The congestion window graph from the single flow, no enforced packet drop experiment is shown in Figure 7. The reductions in the congestion window are due to losses that the flow itself is causing at the bottleneck link. The variation is less than with standard TCP, as *cwnd* decreases by factor that is less than 0.5. The maximum *cwnd* is achieved in 19 seconds. For a normal packet drop, when a flow overflows the router queues and reduces *cwnd* by its decrease factor, the time to recover is 8 seconds. In ns-2, the first drop is treated the same as standard TCP and the window is reduced by half except in FAST, BIC, and CUBIC. This is the reason for the halving of *cwnd* after the first packet loss.

4.1.3.2 Single Flow, Single Drop

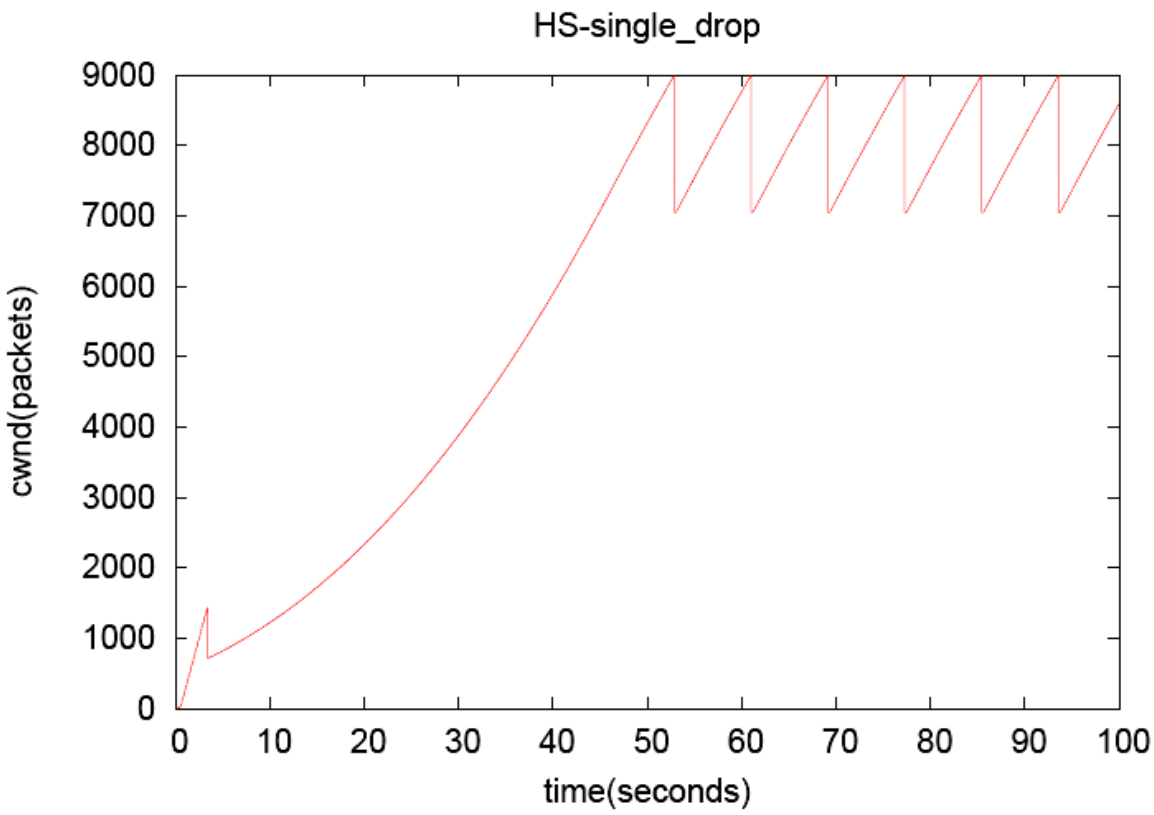

Figure 8: HS-TCP Single drop

Figure 8 shows the congestion window graph of the sender in the single flow, single drop experiment. A packet is artificially dropped in the ascent phase. This delays the rise of *cwnd* to the maximum substantially. Now, it takes 53 seconds to reach the maximum value which is 34 seconds later than the normal case. The reason why this happens is that a higher loss rate causes HS-TCP to become less aggressive resulting in a gentler slope.

4.1.3.3 Single Flow, Different Buffer Sizes

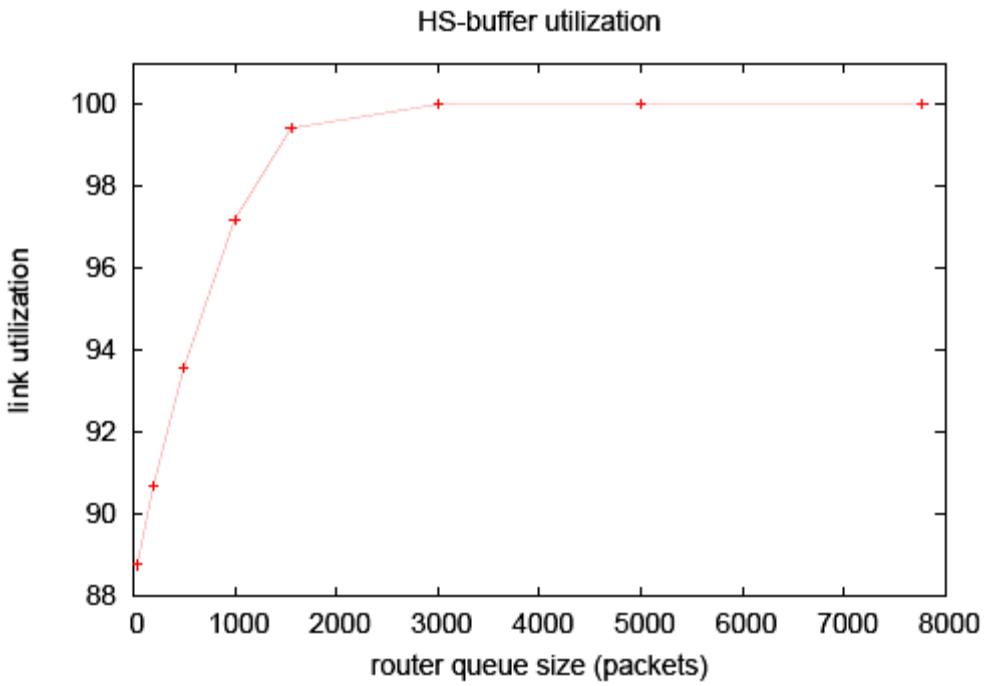

Figure 9: HS-TCP Link utilization at different router queue sizes

In the single flow, different buffer sizes experiment where a single flow is run through the bottleneck link with varying router buffer sizes; the link utilization is very good even with a 40-packet queue (Figure 9). The router queue generally has a non-zero size leading to a good utilization (Figure 10).

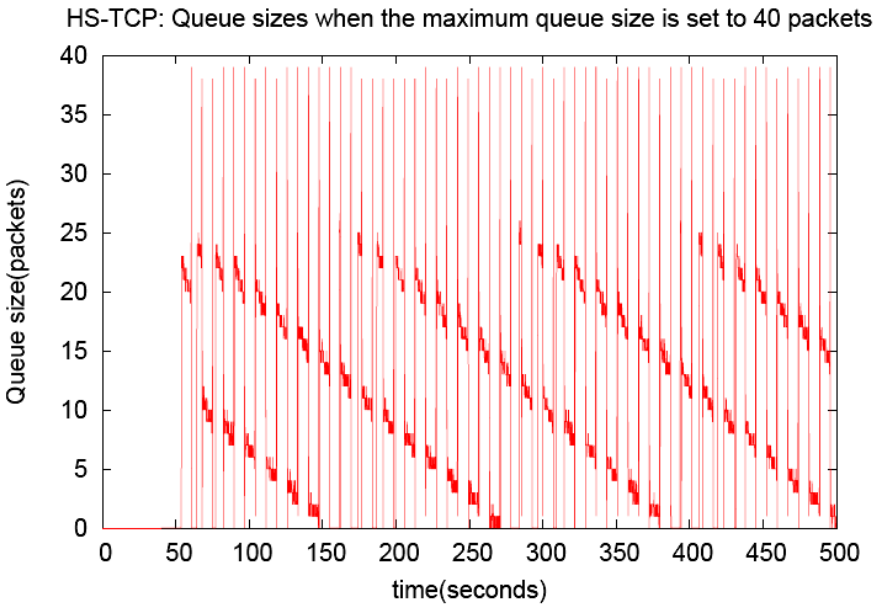

Figure 10: HS-TCP's queue occupancy graph in small router buffer environment

4.1.3.4 Single Flow, Different RTT

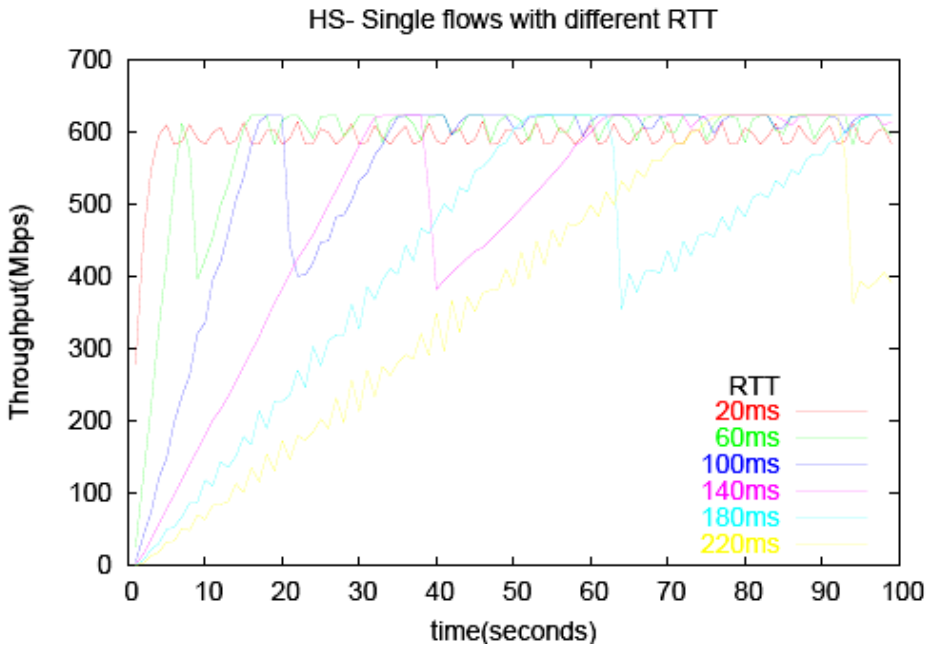

Figure 11: HS-TCP single flows with different RTTs

In this set of experiments, single HS-TCP flows were run with varying link RTTs. The results are shown in Figure 11 and we can see that the RTT has a substantial effect on the aggressiveness of the HS-TCP flows. The 20ms flow is able to fully utilize the link in 5 seconds while the 220ms flow takes 77 seconds to do that. Based on this observation we expect HS-TCP to be RTT-unfair.

4.1.3.5 Two Flows, Different RTTs

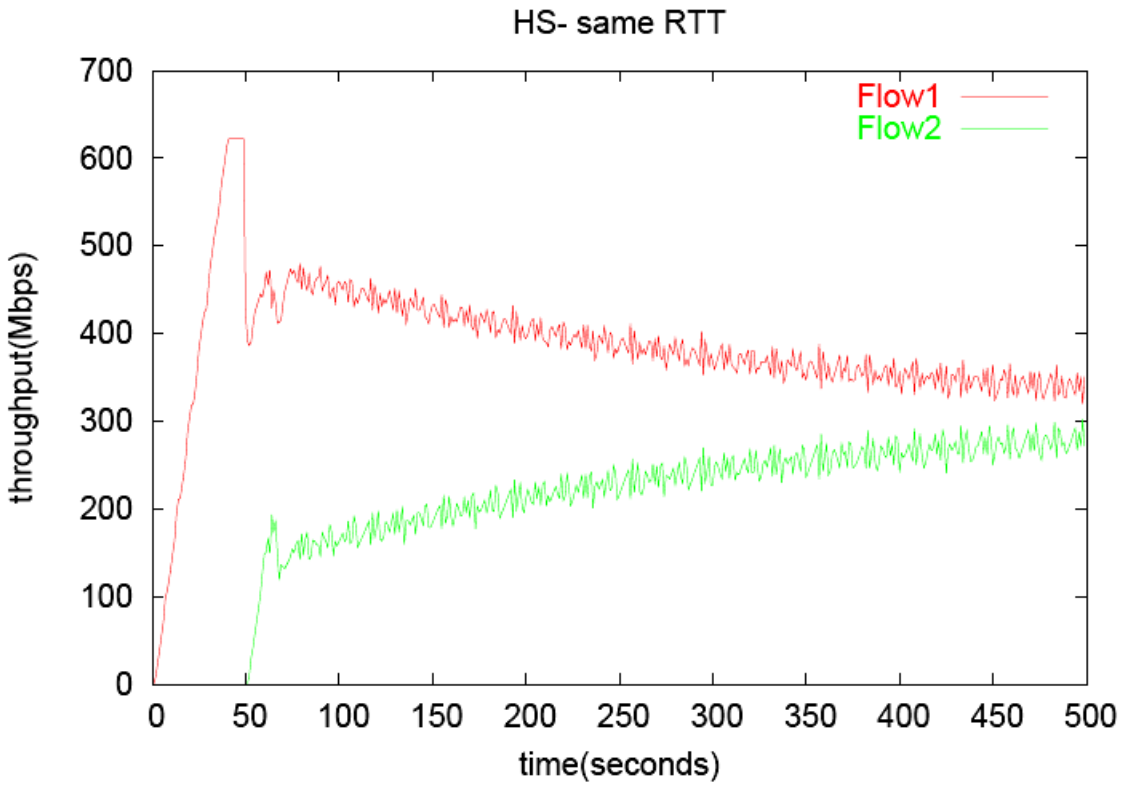

Figure 12: HS-TCP 2 flows with same RTT started 50s apart

In this set of experiments the RTT of the first flow is fixed at 162ms while the RTT of the second flow is varied from 162ms to 16ms. Figure 12 displays the throughput

of the competing flows when both the flows have the same RTT of 162ms. We can see that the flows slowly converge to a fair share.

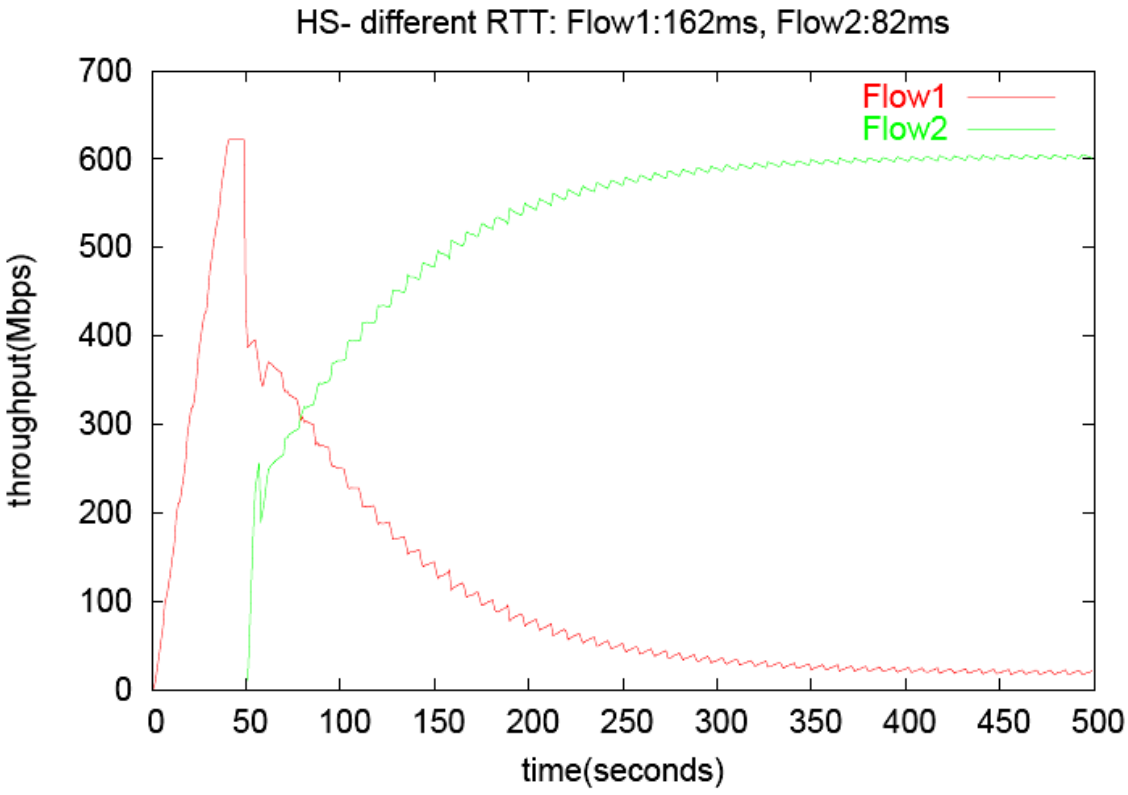

Figure 13: HS-TCP 2 flows with different RTT started 50s apart

We shorten the RTT of the flow starting second to 82ms and the throughputs of the flows are shown in Figure 13. We can see that HS-TCP becomes unfair and the second flow starves the first flow over time and completely dominates the link. Further shortening the RTT of the second flow aggravates this unfairness.

4.1.4 Summary

HS-TCP was able to reach maximum link utilization in 19 seconds. A packet drop during its initial ascent had a substantial affect on its performance and it took 34 more seconds to reach complete link utilization. HS-TCP showed good performance even with a bottleneck router buffer queue of size 40 packets. HS-TCP was fair in sharing bandwidth with itself. However when the RTT of the second flow was halved to 82ms, the sharing became very unfair.

4.2 Scalable TCP

Scalable TCP [Kel03] was proposed by Tom Kelly in 2003 as an improvement to HS-TCP. It is much more aggressive than TCP in increasing the congestion window and less aggressive in reducing the congestion window. This allows it to quickly garner a larger share of the available bandwidth. It aims to be backward compatible by behaving just like TCP when the congestion window is below a certain threshold value. It is called *scalable* since it is able to double its sending rate for any rate in about 70 RTTs.

4.2.1 Congestion Control Algorithm

In this discussion, we use the following terminology:

cwnd : Current congestion window

LowWindow: Congestion window threshold below which behavior is identical to TCP

Scalable TCP has a very simple congestion control algorithm. For every ACK, *cwnd* is incremented by 0.01. On the detection of a congestion event, *cwnd* is reduced by a factor of 0.125. Congestion events are identified by packet losses. Precisely,

On the arrival of an ACK:

cwnd	=	*cwnd* + 1	*cwnd* <= *LowWindow* and slow start
	=	*cwnd* + 1/*cwnd*	*cwnd* <= *LowWindow* and congestion avoidance
	=	*cwnd* + 0.01	*cwnd* > *LowWindow*

On a packet loss:

cwnd	=	0.5 *cwnd*	*cwnd* <= *LowWindow*
	=	(1 – 0.125) *cwnd*	*cwnd* > *LowWindow*

An increment of 0.01 per ACK would result in a total increment of 0.01**cwnd* per RTT as *cwnd* packets will be acknowledged in one RTT. This algorithm is thus a multiplicative-increase/multiplicative-decrease, MIMD, algorithm with 1.01 and 0.875 as the increase and decrease parameters in comparison to TCP's algorithm, which is additive-increase/multiplicative-decrease or AIMD. *LowWindow* has been set to 16 in our experiments.

4.2.2 Unique Points

1. The protocol does not use any tables to calculate the increase and decrease factors as does HS-TCP. It uses a very simple approach of making the increase and decrease parameters more aggressive.

2. The congestion window algorithm is scalable. The sending rate is doubled in a fixed amount of time for all rates. However, the downside of this is that for large *cwnd* values, the doubling will lead to a larger increase, while for a small *cwnd* value the doubling will result in a smaller increase. We expect the larger flows to be more aggressive than smaller flows in our simulations.

4.2.3 Examples of the Protocol Operation

A number of simulations were run to test the various features of HS-TCP. Network topology 1 (Figure 4) was used for the single flow simulations, and network topology 2 (Figure 5) was used for the two flows with different RTT simulations. The results are presented below.

4.2.3.1 Single Flow, No Enforced Packet Drops

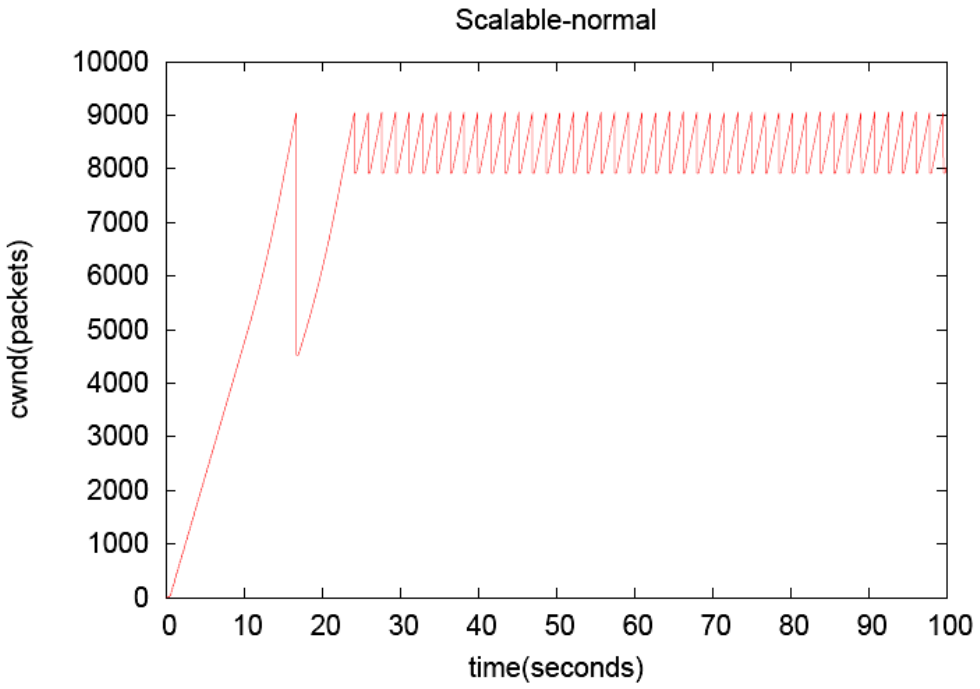

Figure 14: Scalable TCP normal graph

In the single flow, no enforced packet drop simulation, where the flow is started on an empty link with no external interference, the algorithm results in the *cwnd* graph (Figure 14) being again similar to the saw-tooth graph associated with TCP. The variation is however less, as the losses bring down the *cwnd* by a multiple of just 0.875 instead of 0.5. For the first packet drop, the *cwnd* drops down by half and then drops down by a factor of 0.875 subsequently. This first *cwnd* decrease is the same as HS-TCP and is due to the way these protocols have been implemented in ns-2. The maximum *cwnd* is achieved in 16.74s. For a normal packet drop the time to recover is 1.74s.

4.2.3.2 Single Flow, Single Drop

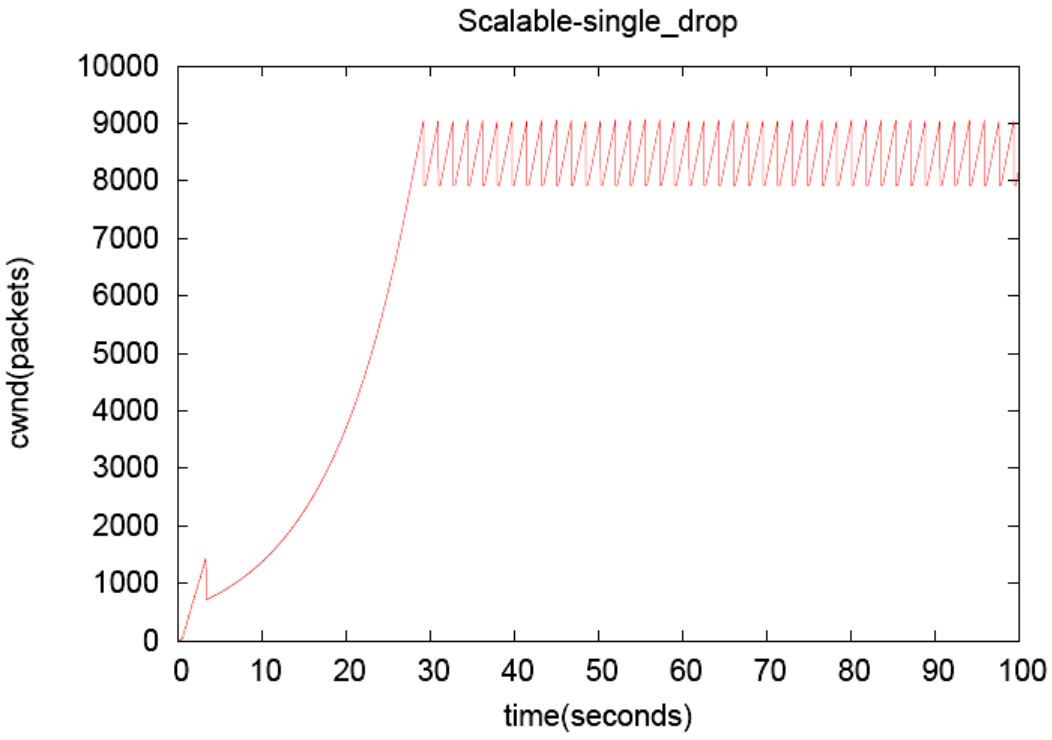

Figure 15: Scalable TCP Single drop

Figure 15 shows Scalable-TCP's *cwnd* graph when an artificial packet drop is made in the initial ascent of the *cwnd*. The first drop is seen at 3.41s and the maximum value of *cwnd* is achieved in 29.29s. This is 12.55s later than the normal case.

4.2.3.3 Single Flow, Different Buffer Sizes

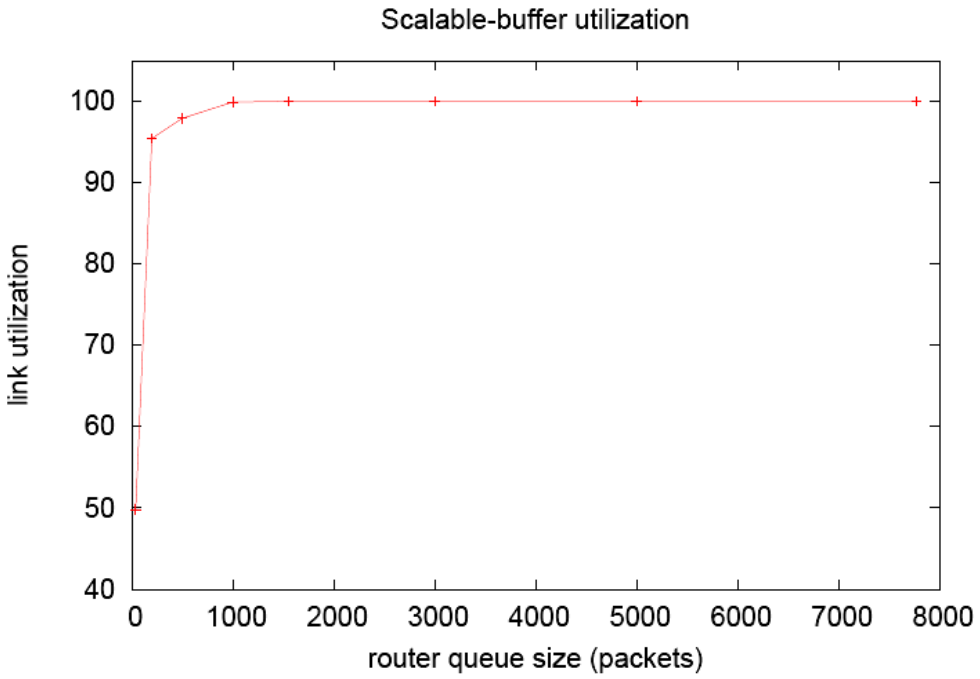

Figure 16: Scalable TCP Link utilization at different queue sizes

The bottleneck router buffer was varied to test the performance of Scalable-TCP in different router buffer size scenarios. Scalable-TCP showed good performance even with small buffers. Figure 16 shows the link utilization graph. Scalable-TCP had over 90% link utilizations in all experiments except in the case of 40 packet router queue where the utilization was only 49.8%.

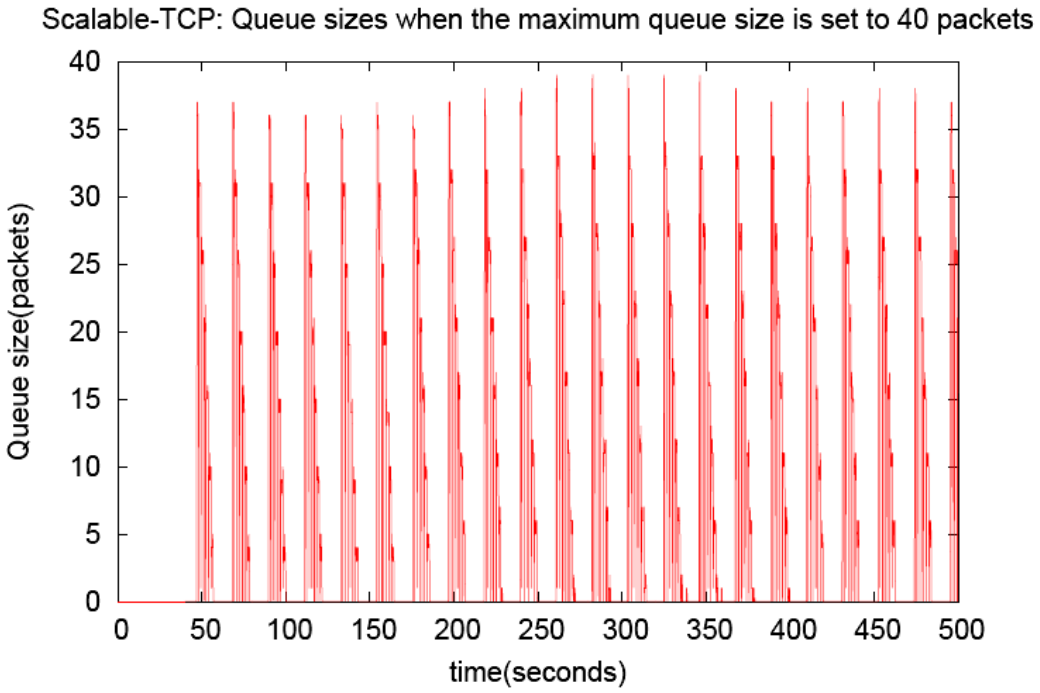

Figure 17: Scalable TCP Queue occupancy when the maximum size is limited to 40 packets

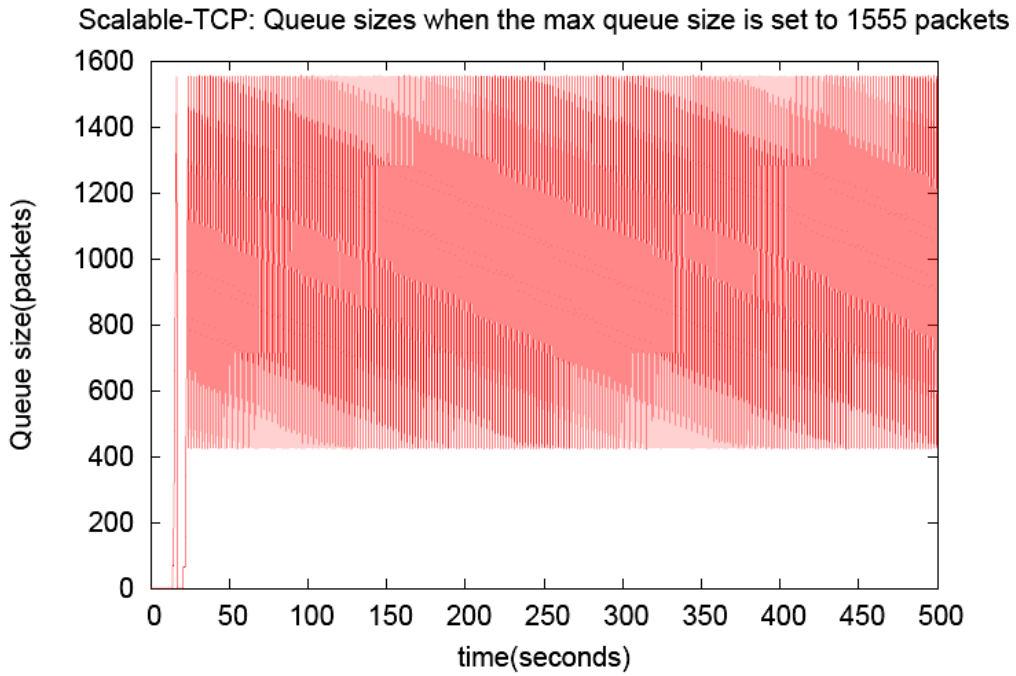

Figure 18: Scalable TCP Queue occupancy when the maximum size is limited to 1555 packets

Figure 17 shows the bottleneck router queue when it is limited to 40 packets. We can see that it is empty for large amounts of time. This is because as the queue overflows and drops packets; Scalable-TCP reduces its sending rate. As the queue size is small, it drains out completely and is empty for a large amount of time before Scalable-TCP can fill it up again. Router queues should always have something to send for the link utilization to be 100%. We see from the router queue sizes graph shown in Figure 18 that the queue never drains out completely when the maximum size is set to 1555 packets. The network utilization for that case is therefore 100%.

4.2.3.4 Single Flow, Different RTT

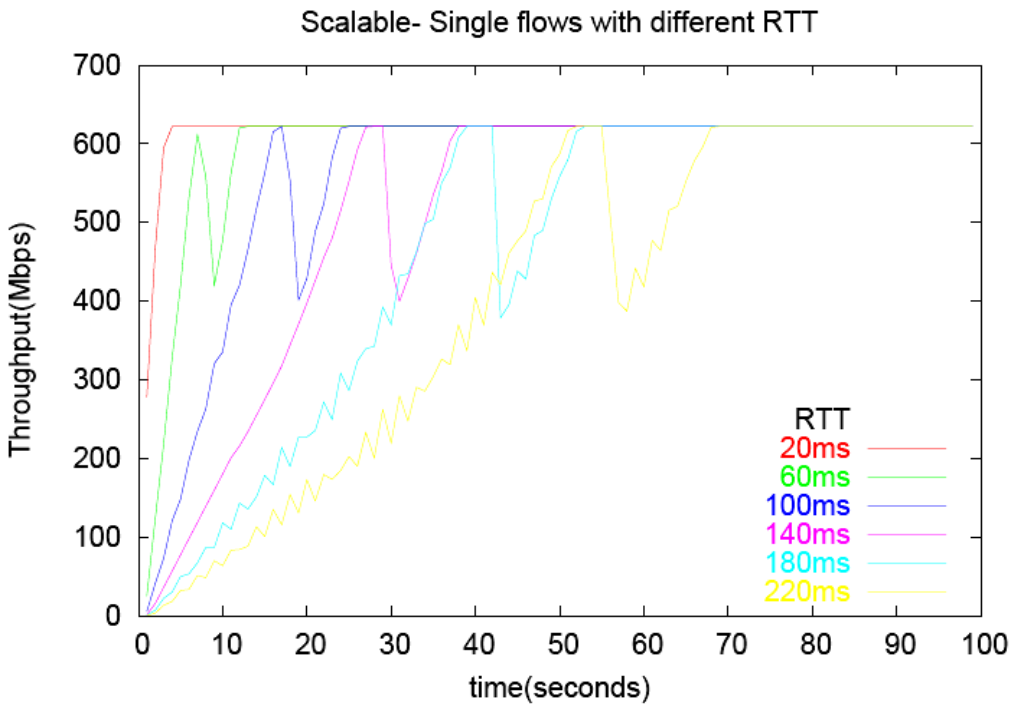

Figure 19: Scalable TCP Single flows with different RTTs

Figure 19 shows the throughput of Scalable-TCP flows at different RTTs. With increasing RTTs, Scalable-TCP flows become less and less aggressive. This is the reason why Bullot *et al*. [BCH03] found Scalable-TCP to be fairer on network links with long delays. The 20ms flow reaches the maximum in 4s while the 220ms flow reaches the maximum in 52s.

4.2.3.5 Two Flows, Different RTTs

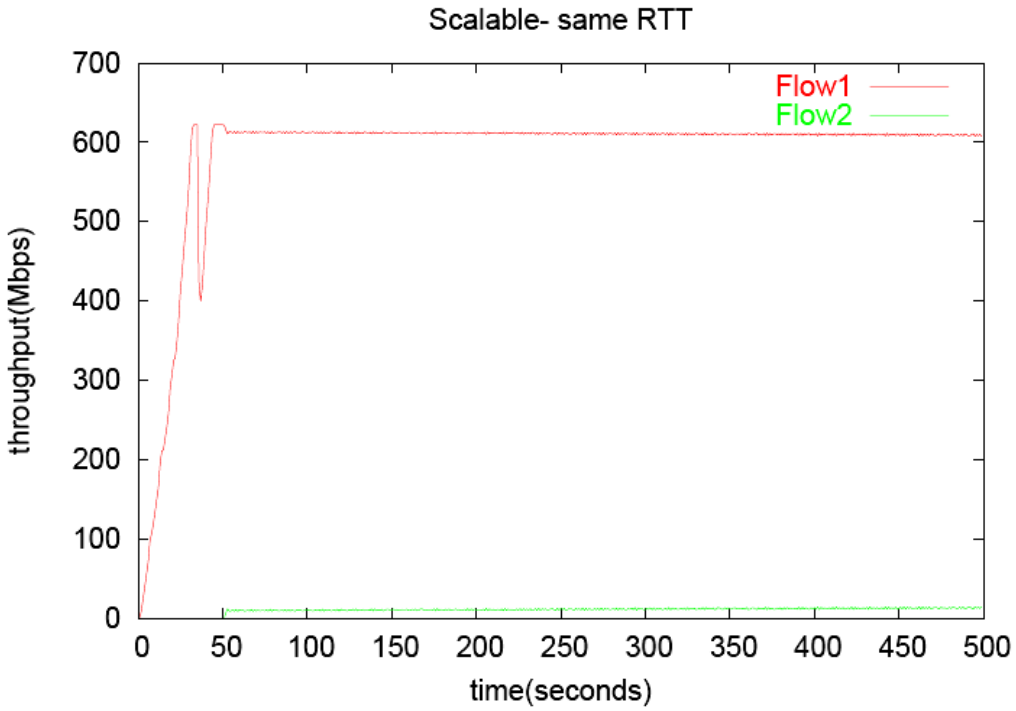

Figure 20: Scalable TCP 2 flows with same RTT started 50s apart

The RTT of the second flow was shortened from 162ms to 16ms keeping the RTT of the first flow fixed at 162ms. In the first case the RTT of the second flow was set to 162ms. The throughput of the two competing Scalable-TCP flows is shown above in

Figure 20. The sharing was very unfair even when the two flows had the same RTT. The reason for this is that as mentioned before, Scalable-TCP is more aggressive at higher *cwnd* window values. This gives an advantage to the larger flow and starves the smaller flow which always gets smaller increments. Hence we see an unfair sharing.

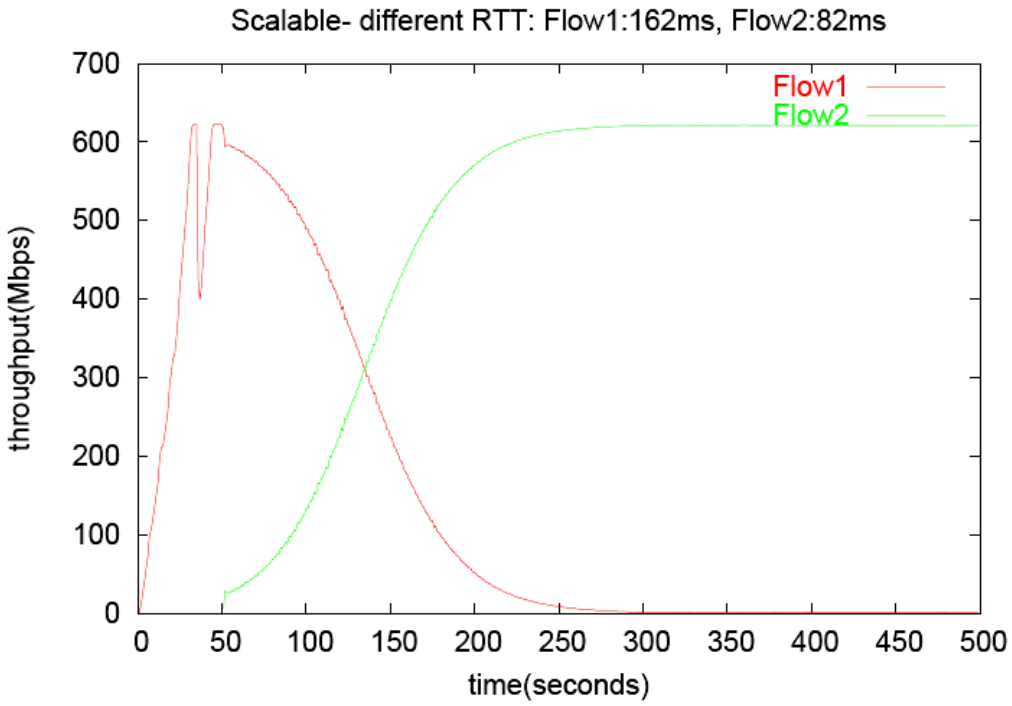

Figure 21: Scalable TCP 2 flows with different RTTs started 50s apart

As the second flow's RTT is shortened to 82ms, it gains an advantage over the bigger flow as it starts getting its ACKs sooner and increments the congestion window faster. This causes the longer flow to loose its domination of the link. Figure 21 shows the throughputs of the flows for this case. The two major factors that govern the share of a flow are its intensity of increase and decrease. With a shorter RTT both the increments as well as the decrements come faster. As we shorten the RTT of the second flow further, Figure 22, we see that it starts to struggle and loses its advantage of faster increments owing to faster decrements.

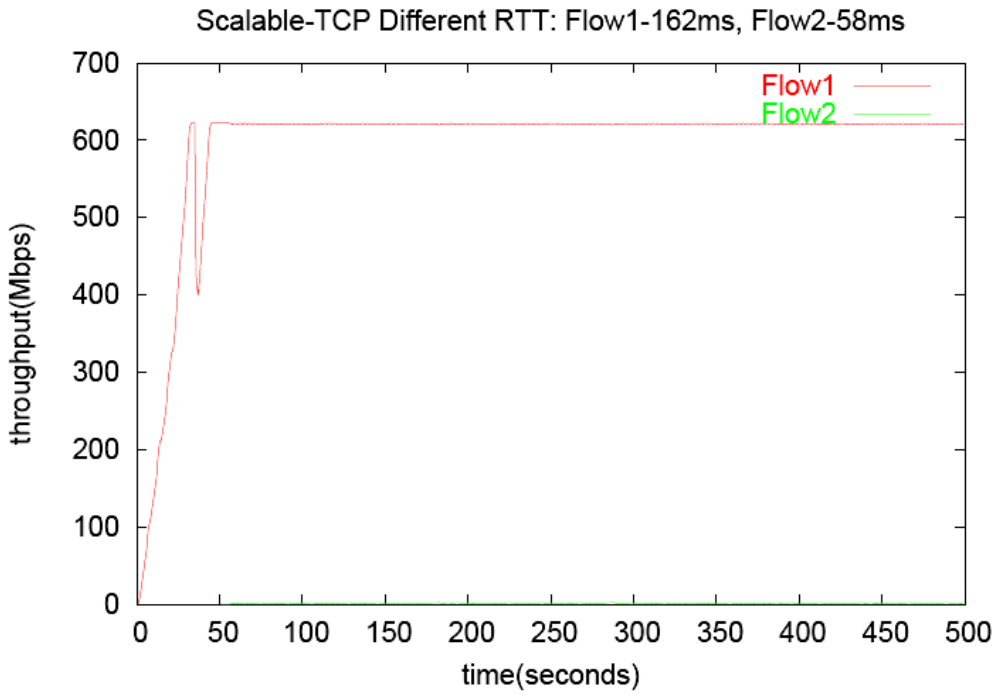

Figure 22: Scalable TCP 2 flows with different RTTs started 50s apart

Scalable-TCP provides larger increments to the flow with the larger *cwnd*. We suspected that if the flow with the shorter RTT was already dominant on the link, then it would always be dominant because of the added advantage of receiving the larger of the two increments. We re-ran this set of experiments with the flow with the shorter RTT starting first to test this.

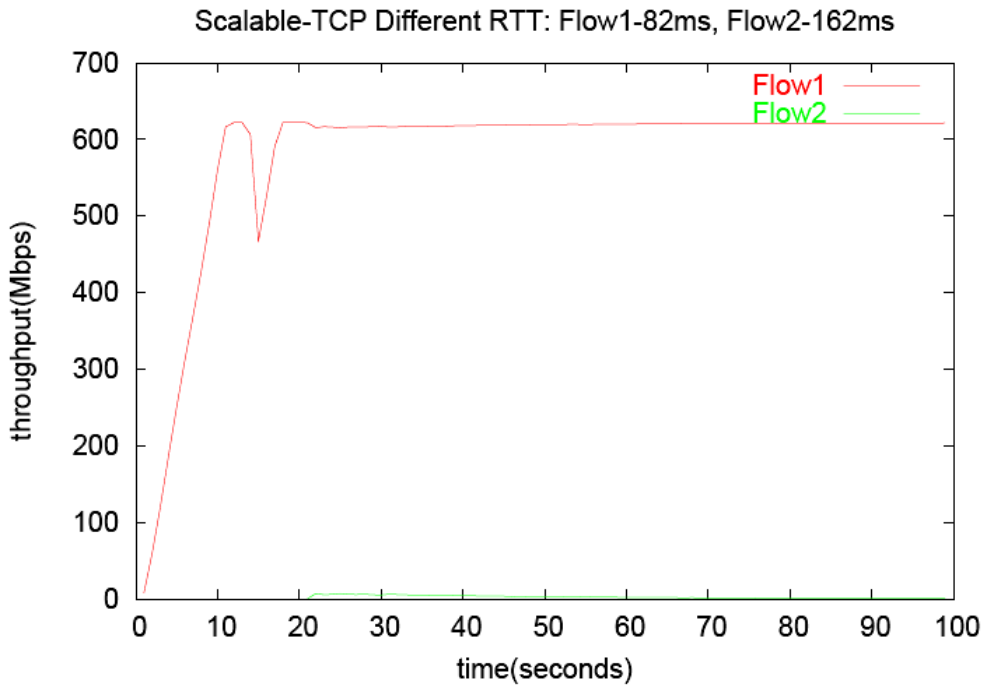

Figure 23: Scalable TCP 2 flows with different RTTs started 50s apart, shorter flow started first

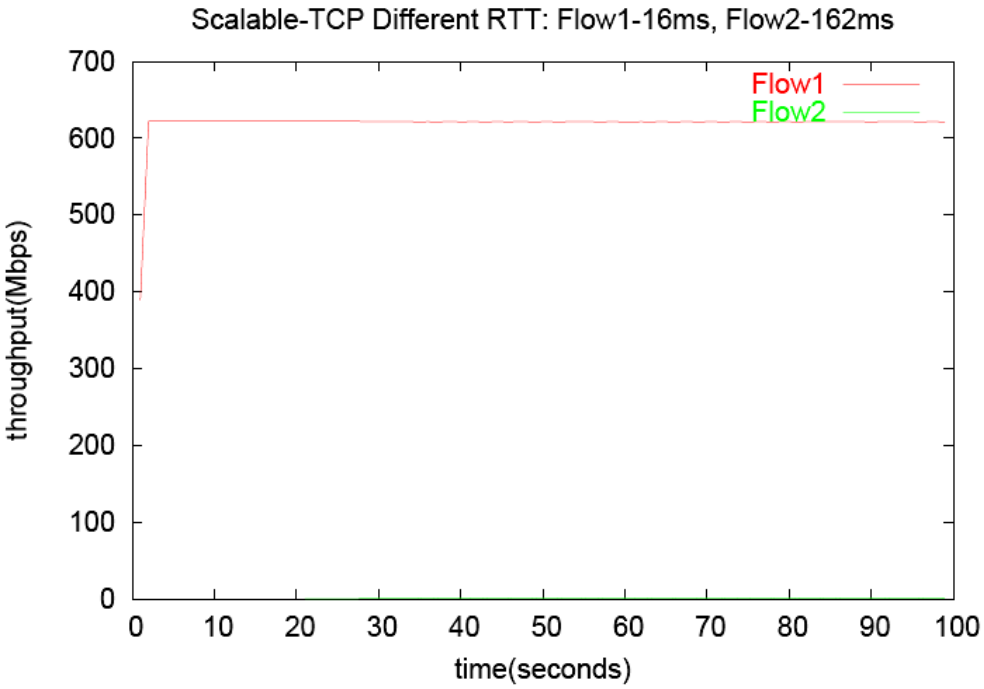

Figure 24: Scalable TCP 2 flows with different RTTs started 50s apart, shorter flow started first

Figure 23 and Figure 24 show throughput of Scalable-TCP flows when the shorter RTT flow is started first. We can see from these graphs that the first flow was dominant across all other values of the second flow's RTT. It always had the larger and faster increments and the second longer RTT flow always got starved.

4.2.4 Summary

Scalable-TCP was quick in increasing to the maximum link capacity and an occurrence of a packet loss during the ascent phase did not have much affect on its performance. It did not perform well when the bottleneck router buffer size was reduced to 40 packets because of its aggressive nature. It continuously overflowed the router queue resulting in a large number of packet drops and subsequent window reductions and poor link utilization. It has no provision for RTT-fairness and its weakness in environments where competing flows had different RTT was exposed. It was never able to provide a fair sharing to competing flows, with the result always skewed towards one end.

4.3 BIC-TCP

BIC-TCP was proposed by Xu *et al* [XHR04] from North Carolina State University (NCSU) in 2004. An additional constraint of "*RTT fairness*" was added to *TCP friendliness* and *bandwidth scalability*. The protocol has characteristics similar to other AIMD protocols but differs in behavior when it approaches link capacity. In

contrast to just incrementing by a fixed constant, BIC-TCP's congestion control mechanism switches to smaller and smaller increments as it gets closer to its estimate of the maximum link capacity. Thus, even when the protocol completely fills up the link and overflows the bottleneck router queue, the overflow is small and the affect is limited to a smaller extent.

4.3.1 Congestion Control Algorithm

We use the following terminology in the discussion below:

cwnd : Current congestion window

S_{max} : Maximum Increment

S_{min} : Minimum Increment

Mx_{cwnd}: Congestion window value where the protocol fully utilizes the available link capacity

Mn_{cwnd} : Congestion window value where there are no losses

T_{cwnd} : Target congestion window value, usually ((Mx_{cwnd} + Mn_{cwnd}) / 2)

β : Multiplicative decrease factor

BIC-TCP uses a *Binary Increase* strategy for increasing *cwnd* when the Mx_{cwnd} is known and a *slow start* strategy when Mx_{cwnd} is unknown. In the *Binary Increase*, if the increments are too steep, *i.e.* greater than S_{max}, then an *Additive Increase* strategy is employed. To ensure a quick convergence to an equitable sharing, *Fast Convergence* is used. The phases are explained in greater detail below:

- *Binary Increase*: Mx_{cwnd} is known and *cwnd* is raised to that value using a binary search technique. At each step, T_{cwnd} is calculated as $(Mn_{cwnd}+Mx_{cwnd})/2$ and the increment, *i.e.* (T_{cwnd} –*cwnd*), is found. If the increment is greater than S_{max}, then the increment is set to S_{max}. If the increment is below S_{min}, the window is raised to T_{cwnd} and the Mn_{cwnd} is also set to T_{cwnd}. After this T_{cwnd} is freshly calculated and the process is repeated until T_{cwnd} becomes equal to Mx_{cwnd}, after which the slow start phase is entered. In case the increment exceeds S_{max}, the window is increased only by S_{max}.
- *Additive Increase*: The difference between T_{cwnd} and *cwnd* exceeds S_{max}, and the increment is capped at S_{max}. The *cwnd* increases linearly in this phase.
- *Multiplicative Decrease*: In case of a packet loss, *cwnd* is reduced by a fixed factor β, usually 0.875, the same as Scalable-TCP. Mx_{cwnd} is set to the last *cwnd* value, Mn_{cwnd} is set to the reduced *cwnd* value and T_{cwnd} is recalculated. The congestion control algorithm then switches the *Binary Increase* phase.
- *Max Probing*: When Mx_{cwnd} is unknown, *cwnd* increases exponentially similar to TCP's slow start phase, till the increment becomes S_{max}, at which point *Binary Increase* is entered.
- *Fast Convergence*: This aims at converging quickly to a fair link share when there are other flows on the link. When a window reduction happens, it attempts to determine whether the flow's share of the bandwidth is smaller or larger than its actual fair share value. It does so by observing whether the Mx_{cwnd} is on a downward trend, *i.e.* the flow has greater than fair share, or an upward trend, *i.e*. the flow has a smaller than fair share. In case the flow has a greater share, rather than setting Mx_{cwnd} to the last

cwnd value, it is set to (Mx_{cwnd} + Mn_{cwnd}) / 2. This way the flow with the greater share will recover much slower from the previous loss than the flow with the smaller share, allowing the sharing to become more equitable.

The algorithm results in the *cwnd* graph being logarithmic in the *Binary Increase* phase and inverted logarithmic, or exponential, in the *Slow Start* phase. We use the following settings for BIC-TCP in our experiments:

- $S_{max} = 32$
- $S_{min} = 0.01$
- $\beta = 0.125$

4.3.2 Unique Points

1. The congestion window increases very quickly when it is far away from BIC-TCP's estimate of maximum congestion window value. This allows the protocol to increase its share quickly early on and also to increase rapidly when there is spare capacity.
2. *Fast Convergence* helps the protocol to be RTT fair.
3. Even in case of losses, since the increments are small as BIC-TCP approaches its estimated maximum, the queue is overflowed by only a small amount. Thus the loss rate is low and the effect on other competing flows is also much less.

4.3.3 Examples of the Protocol Operation

Various simulations were run to test the performance of BIC-TCP. Network topology 1 (Figure 4) was used for the single flow simulations and network topology 2 (Figure 5) was used for the two flows with different RTT simulations. BIC-TCP and

CUBIC become very gentle as they approach their estimate of the link capacity. We therefore note the time that they take to reach 90% link utilization and not 100% link utilization. The results from the simulations are presented below:

4.3.3.1 Single Flow, No Enforced Packet Drops

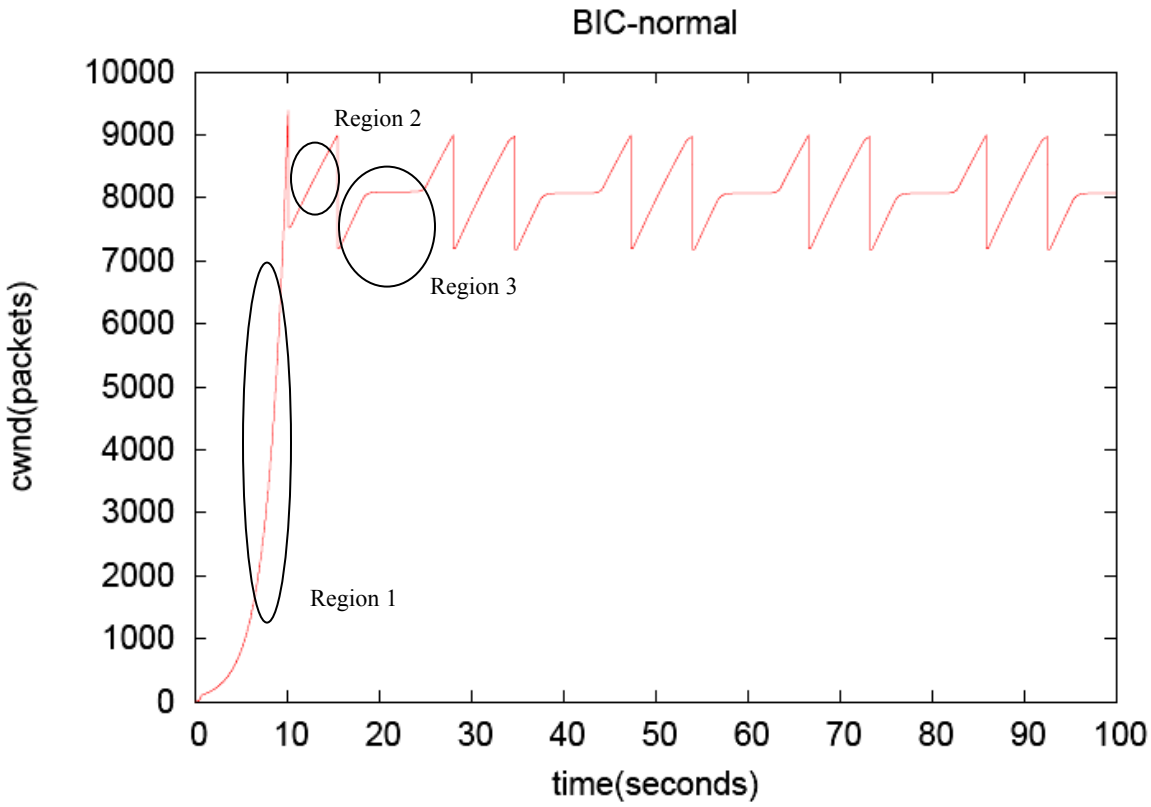

Figure 25: BIC-TCP normal graph

BIC-TCP's *cwnd* graph in the normal case is shown in Figure 25**.** The initial curve in region 1 is exponential corresponding to the *Slow Start* since S_{max} is unknown. A packet drop happens when the flow overshoots the maximum available bandwidth. Mx_{cwnd} is set to the *cwnd* value right before the loss. Since the graph only captures the

congestion window every 0.1 seconds, it misses the maximum *cwnd* value. Looking at the *cwnd* value in the *cwnd* trace file, the maximum is revealed as 9409.98. The flow then tries to get to Mx_{cwnd} value using *Binary Increase*. The *Additive Increase* phase can be seen in region 2 of the graph. Since the first Mx_{cwnd} was an over-estimate a packet drop is seen in the *Additive Increase* phase itself leading to another window reduction in region 3. The protocol then does a *Fast Convergence* since the new Mx_{cwnd}, 8993.95, is smaller than 9409.98, the last one. Hence the next flow increase happens only to ((Mx_{cwnd} + Mn_{cwnd}) / 2) and not Mx_{cwnd}. BIC-TCP takes 9.53 seconds to reach 90% network utilization.

4.3.3.2 Single Flow, Single Drop

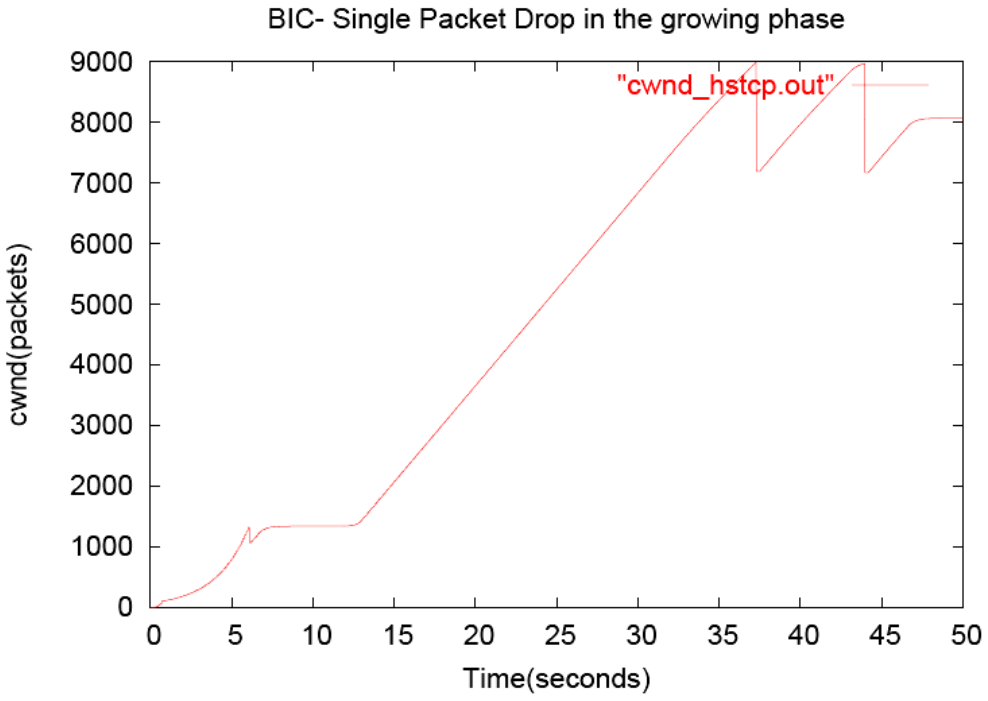

Figure 26: BIC-TCP Single drop

Figure 26 shows BIC-TCP's *cwnd* graph when a packet is dropped in the protocol's growth phase. The rise to the maximum is delayed considerably. The packet drop deceives the protocol into thinking that it has reached the link maximum and its estimate of the maximum link capacity gets set to an incorrect low value. The flow goes into a plateau lasting approximately 7 seconds and then does an *Additive Increase* to reach the 90% link utilization in 30.74 seconds. Thus a packet drop early on ends up in delaying the rise by around 21.21 seconds.

4.3.3.3 Single Flow, Different Buffer Sizes

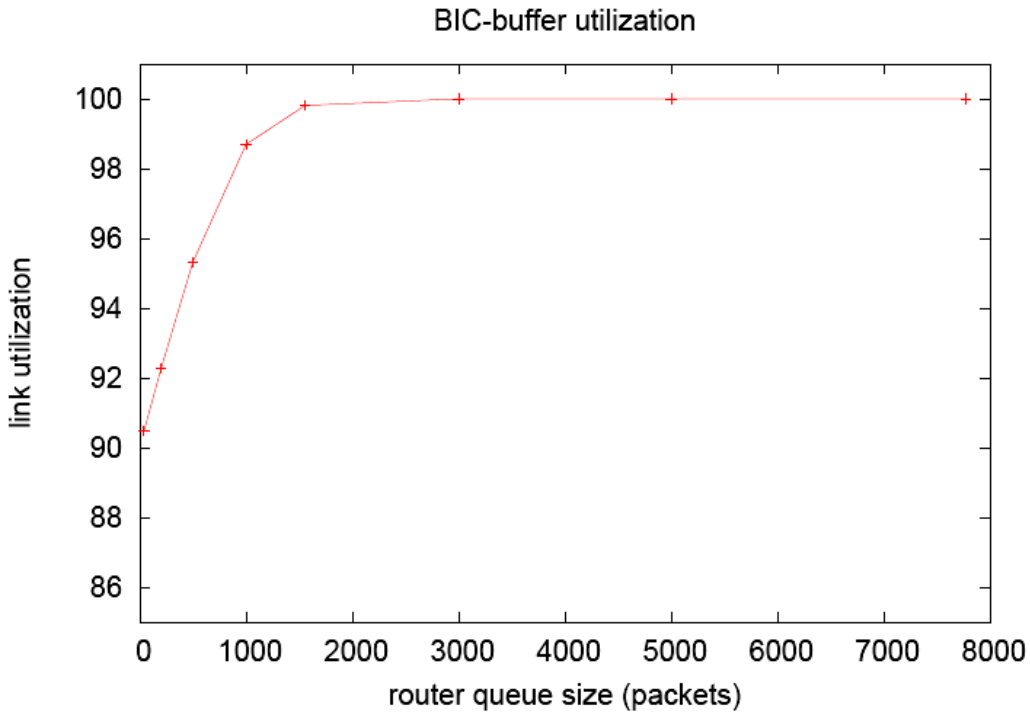

Figure 27: BIC-TCP Link utilization at different queue sizes

Figure 27 shows the link utilization when BIC-TCP is run in varying router queue size scenarios. BIC-TCP performs impressively for all buffer sizes. The link utilization is always greater than 90%.

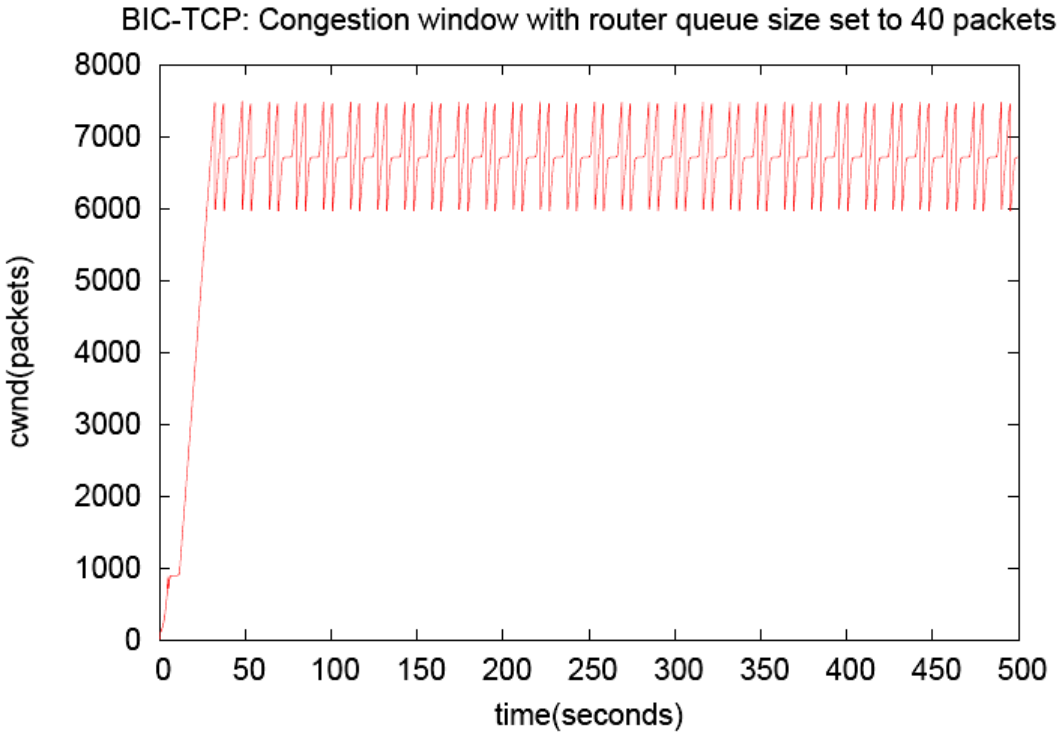

Figure 28: BIC-TCP's cwnd graph in small router buffer environment

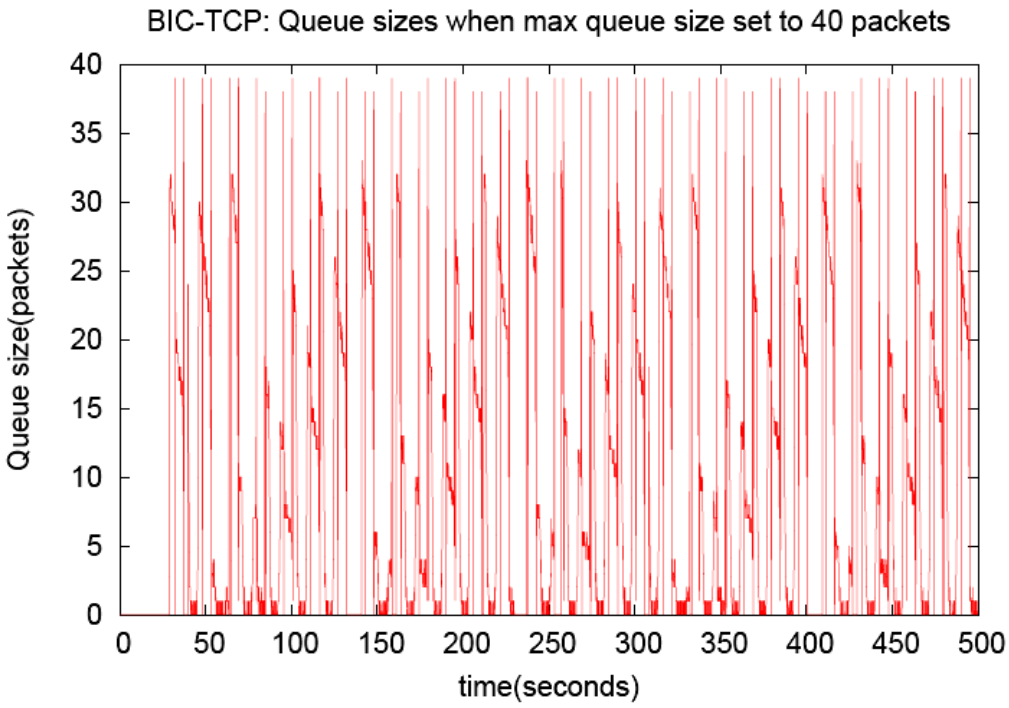

Figure 29: BIC-TCP's queue occupancy graph in small router buffer environment

We see in Figure 29 that the router queue size is non-zero for most of the time. This is because BIC-TCP stays stable and has very small increments as it nears its estimate of the link capacity. This helps it to have a good performance. The *cwnd* graph, Figure 28, is also stable.

4.3.3.4 Single Flow, Different RTT

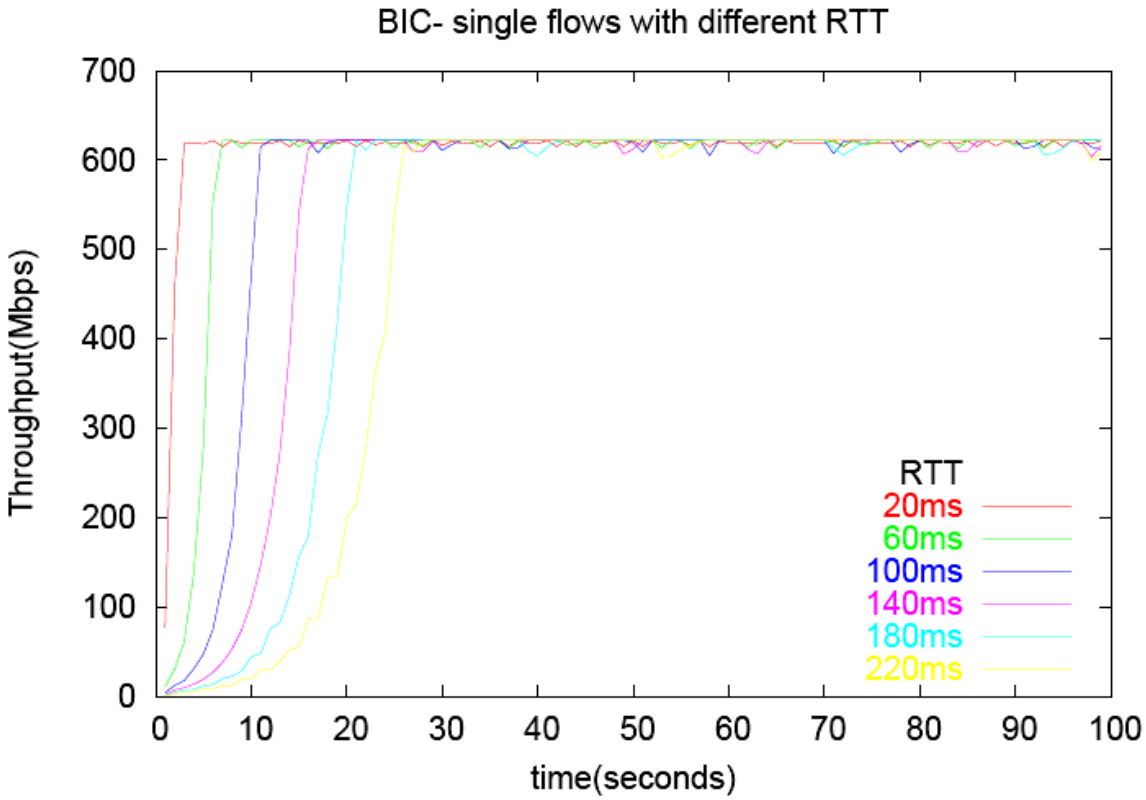

Figure 30: BIC-TCP single flows with different RTTs

Figure 30 shows the throughput of a single BIC-TCP for different link RTTs. When the windows are increasing, each *ACK* received acts as an increment to the window size. For flows with small delays, these *ACKs* come faster and hence they grow faster. A flow with a large delay tends to grow slower for the same reason. The RTT of the network path was varied to see the effect it would have on the aggressiveness of BIC-TCP. The 20ms flow is able to reach 90% link utilization in 1.7 seconds while the 220ms flow takes 21.6 seconds to do so. BIC-TCP attempts to balance this unfairness when

flows are competing against each other by introducing *Fast Convergence* and we will test the efficiency of this technique in the next section.

4.3.3.5 Two Flows, Different RTTs

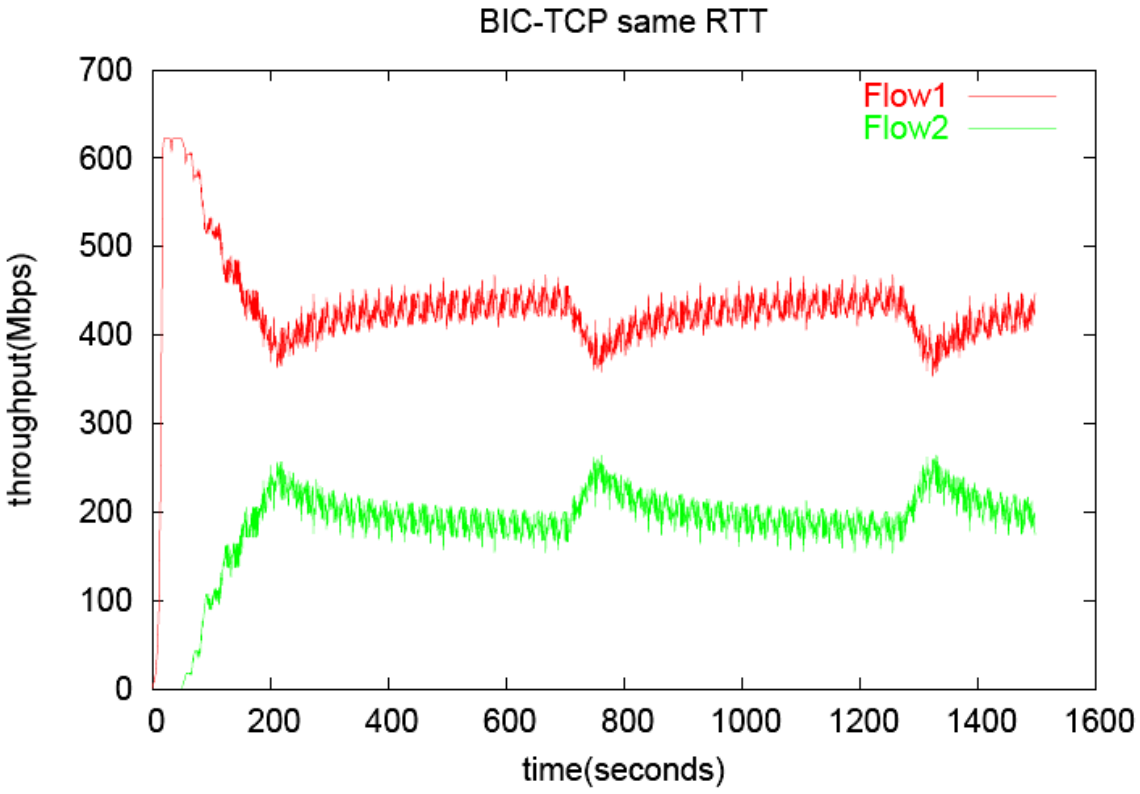

Figure 31: BIC-TCP 2 flows with same RTTs

The two flows never share the bandwidth fairly even when their RTTs are similar. We see the first flow coming down a little to make space for the second flow in the throughput graph shown in Figure 31. The second flow garners a substantial share of the bandwidth. This behavior can be explained by focusing on how BIC-TCP provides RTT-fairness. The protocol follows the *Fast Convergence* strategy if the Mx_{cwnd} is on a

downward trend. When two flows are competing on the link and the maximum bandwidth is surpassed, both of them will see a loss and will reduce their *cwnd* by a factor of 0.875. The reduction will clearly be greater for the flow with the larger share of the bandwidth. *Fast Convergence* will also cap the Mx_{cwnd} of the larger flow to a smaller value. This will give the flow with the smaller share of bandwidth an advantage over the larger flow leading to a fairer state. At higher differences in RTTs however, the disadvantage due to longer delay far exceeds the advantage due to *Fast Convergence*. Figure 32 shows the throughput of two competing BIC-TCP flows when the RTT of the second flow is set to 82ms. We can see that the share of the second flow gets greatly reduced.

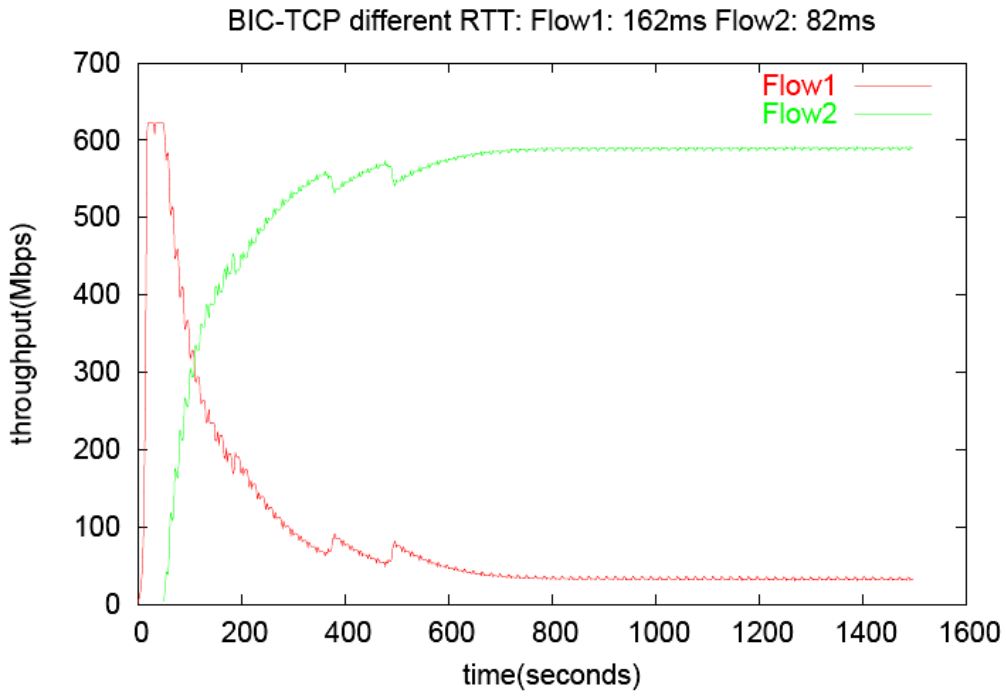

Figure 32: BIC-TCP 2 flows with different RTTs started 50s apart

4.3.4 Summary

BIC-TCP was quick in increasing its *cwnd* to fully occupy the link. It suffered when there was a packet drop early on in its initial ascent because it made an incorrect estimate of the maximum link capacity. It had very good performance in the limited bottleneck router buffer size scenario because of its stability as it nears its estimate of the link capacity. In spite of its *Fast Convergence* strategy, it was not able to share the link capacity fairly with other competing flows.

4.4 CUBIC

CUBIC is another proposal from Rhee *et al*. from NCSU [RX05]. It has been designed as an enhancement to BIC-TCP. It tries to improve BIC-TCP's fairness by becoming less aggressive. It uses a cubic increase function rather than the binary increase strategy that BIC-TCP uses allowing it to be gentler when it nears the plateau. The shape of its congestion window curve still resembles that of BIC, with the difference that the increases are gentler.

Another major change is that the congestion window has been made dependent on the time elapsed since the last congestion event rather than the previous congestion window value. This approach is similar to the one used by H-TCP. The approach to achieving TCP compatibility has also been completely changed. There is no minimum threshold value for the congestion window value now. Instead, the window size of a TCP

connection in a similar situation is found and that value is used if it is more than CUBIC's congestion window value.

4.4.1 Congestion Control Algorithm

In this discussion, we use the following terminology:

cwnd : Congestion window value

C : constant used for scaling

T : the time since the last congestion event

W_{max} : size of the window just before the window was last reduced

β : multiplicative decrease factor, currently set to 0.8

K : $(W_{max}\ \beta/C)^{1/3}$

S_{max} : Maximum rate of increase

RTT_{max}: Maximum observed RTT

RTT_{min}: Minimum observed RTT

On the arrival of an ACK:

$$cwnd = C(T - K)^3 + W_{max}$$

On the detection of a packet loss:

$$cwnd = \beta\ W_{max}$$

CUBIC monitors the rate of increase of *cwnd* and limits it to S_{max}. We use the following settings for CUBIC's parameters:

- $\beta = 0.8$
- $C = 0.4$
- $W_{max} = 160$

4.4.2 Unique Points

1. CUBIC retains the advantages of BIC-TCP's congestion window curve. As the flow nears its estimate of its share in the network bandwidth, it becomes gentler in its increases. Thus when the maximum link capacity is reached, the router queues are overflowed by only a small number of packets, which helps in reducing the loss rate.

2. A lot of effort has been made to make CUBIC TCP-friendly. The rate of increase has been moderated significantly. This sometimes makes CUBIC gentler in comparison to TCP connections especially on small RTT network paths. Therefore, instead of having a *LowWindow* parameter for switching between high speed and low speed TCP compatible modes, *cwnd* is monitored and is set to *cwnd* from TCP's algorithm if that value is greater. This ensures fair behavior.
3. Protocols with increments dependent on current *cwnd* values usually suffer from intra-protocol unfairness as the flows with a larger share of bandwidth will always have bigger increments. Binding the *cwnd* value to time since last packet drop helps in avoiding this problem.

4. Updates to *cwnd* are inversely proportional to W_{max}. This means that bigger flows will get updated to smaller values compared to smaller flows which would get larger updates. This results in a quick convergence between different flows.

5. CUBIC also ensures RTT-fairness because of two reasons. Firstly, *cwnd* updates are directly proportional to T. T would be same for all the flows on a link irrespective of their RTTs after a synchronized packet drop. Thus all the flows would see similar increments. Secondly, all flow expansions are rate limited. When the flow windows are far away from W_{max}, they will increase linearly in time. This is better than BIC's approach where the individual increments were capped to a similar value since shorter flows would add up the increments faster owing to quicker receipt of RTTs. However real-time rate limitation does not suffer from this problem.

6. Since CUBIC increases the congestion window quite slowly compared to the other high-speed protocols, there is a provision for faster increments during low utilization. If the RTT is larger than $0.1(RTT_{max} - RTT_{min})$, then *cwnd* updates are no longer rate limited. This condition is checked only in the max probing phase. This allows the flows to grow faster.

4.4.3 Examples of the Protocol Operation

Various simulations were run to test the performance of CUBIC. Network topology 1 (Figure 4) was used for the single flow simulations and network topology 2 (Figure 5) was used for the two flows with different RTT simulations. CUBIC becomes very gentle as it approaches its estimate of the link capacity. We therefore treat it similarly to BIC-TCP and note the time that it takes to reach 90% link utilization and not 100% link utilization. The results from the simulations are presented below:

4.4.3.1 Single Flow, No Enforced Packet Drops

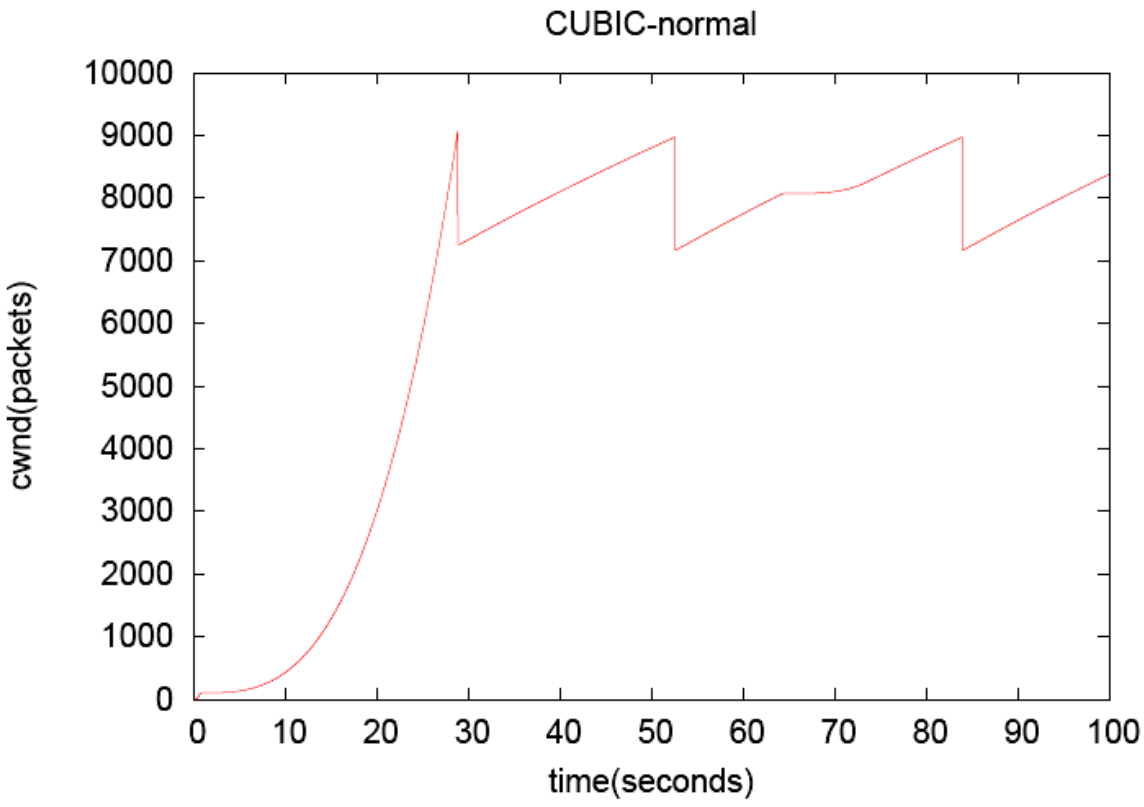

Figure 33: CUBIC normal graph

In the single flow, no enforced packet drop experiment, the congestion window graph, Figure 33, is similar to BIC-TCP's graph. It is however gentler and 90% link utilization is achieved in 26.55 seconds.

4.4.3.2 Single Flow, Single Drop

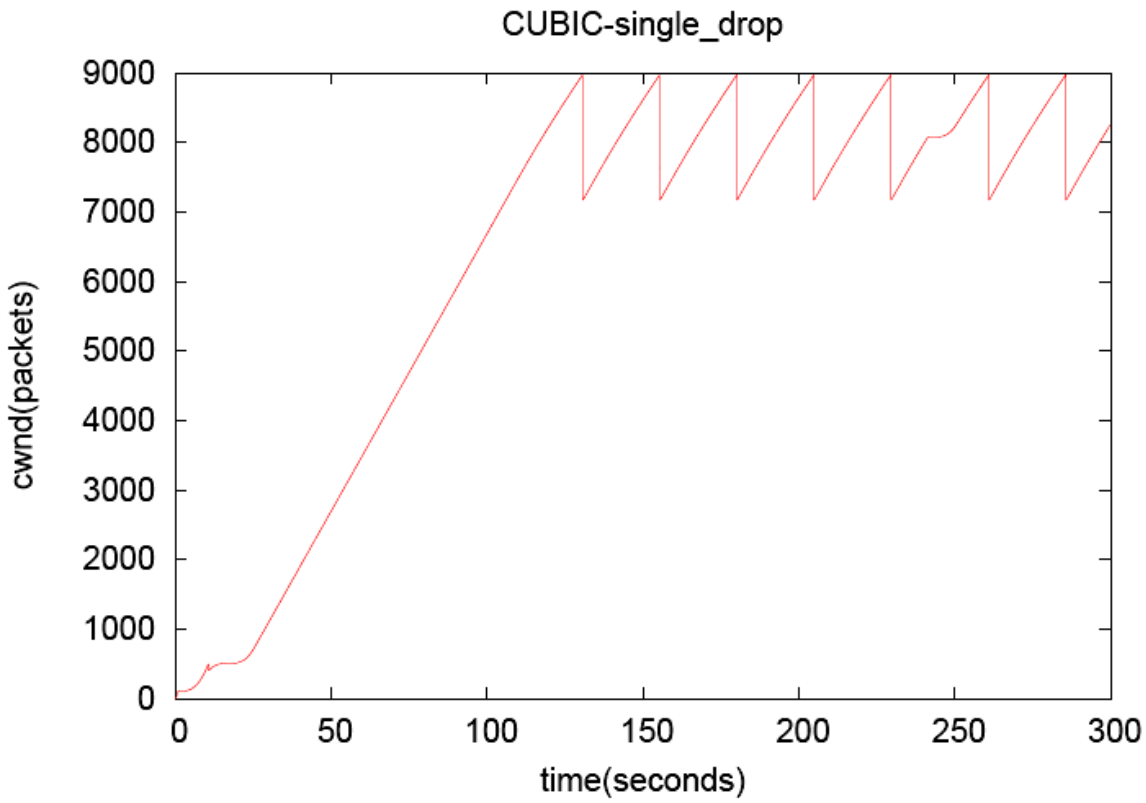

Figure 34: CUBIC Single drop

In the single flow, single drop experiment a packet is artificially dropped in the ascent phase. Figure 34 shows the *cwnd* graph for this simulation. The rise of *cwnd* to the maximum is substantially delayed and it takes CUBIC 104.01 seconds to reach 90% link

utilization which is 77.46 seconds slower than the normal case. The packet drop is seen by CUBIC as a queue overflow and is interpreted as having reached full link capacity. CUBIC sets its estimate of the maximum link capacity to this value. This is an incorrect estimation and is much lower than the actual link capacity. CUBIC takes a long time to recover from it and the performance suffers.

4.4.3.3 Single Flow, Different Buffer Sizes

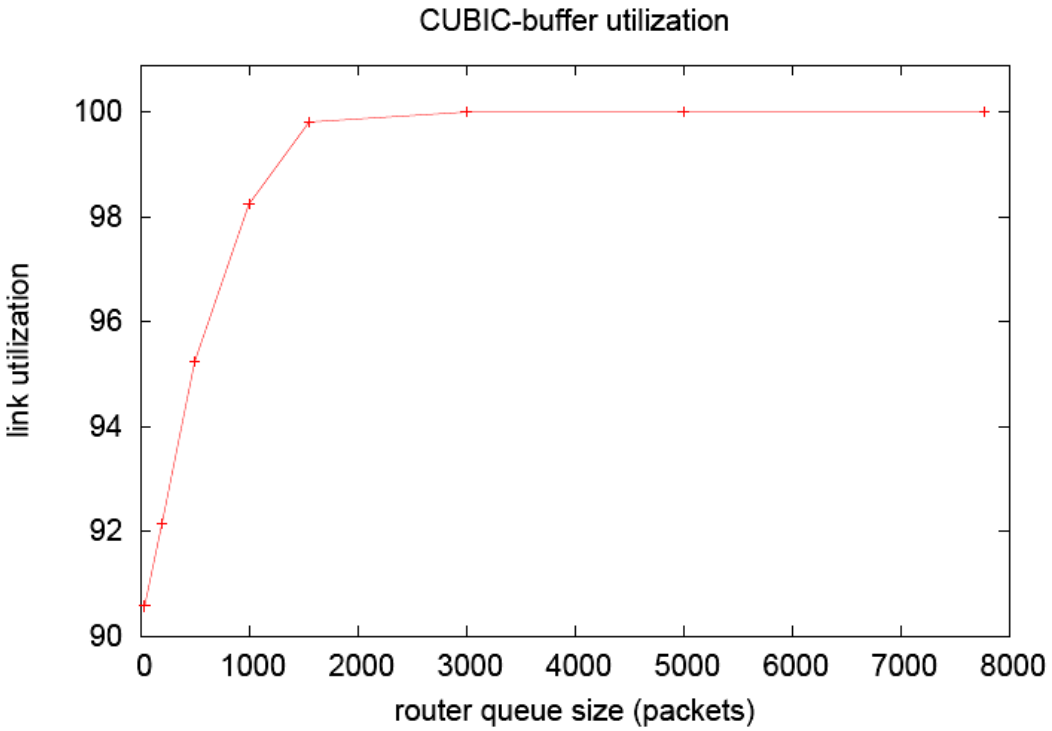

Figure 35: CUBIC Link utilization at different queue sizes

In the single flow, different buffer sizes experiment where a single flow is run through the bottleneck link with varying router buffer sizes; the utilization is very good

even with a 40-packet queue (Figure 35). This is similar to the results from BIC-TCP. This was expected since the *cwnd* graph of CUBIC closely resembles that of BIC-TCP. It also increases very slowly as it nears its estimate of link capacity.

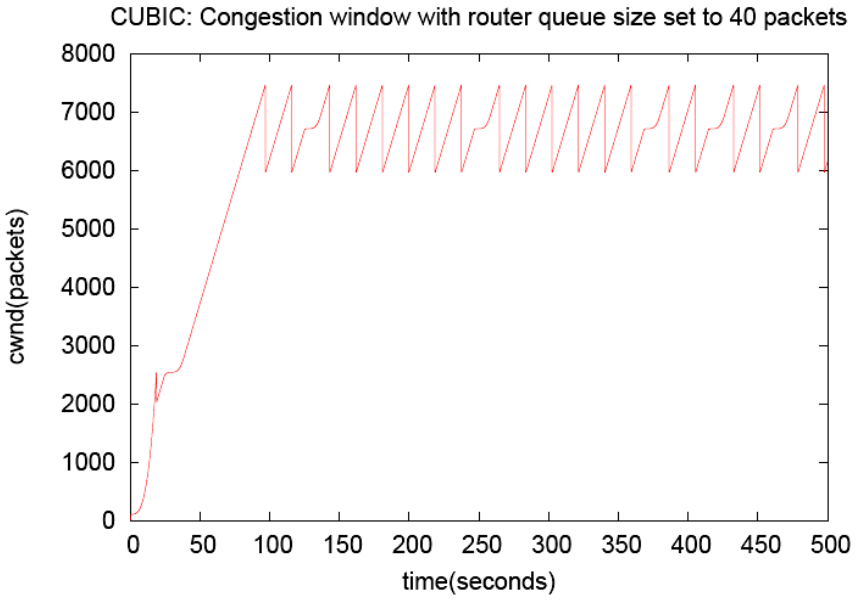

Figure 36: CUBIC's cwnd graph in small router buffer environment

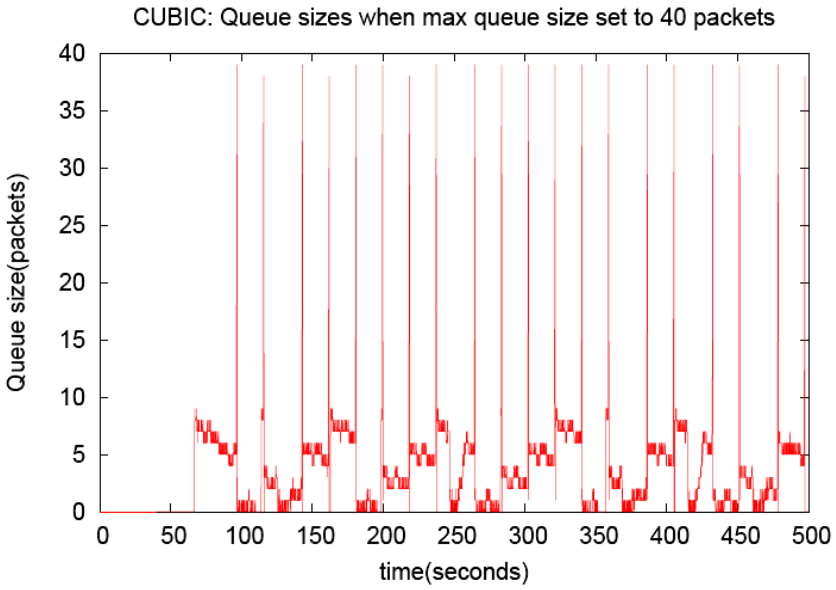

Figure 37: CUBIC's queue occupancy graph in small router buffer environment

Figure 37 shows router queue occupancy when the maximum queue size is set to 40 packets. We can see that the router queue is non-empty for most of the time. This is more stable than the corresponding graph for BIC-TCP, Figure 29. This is because CUBIC is gentler than BIC-TCP and stays close to the maximum link capacity for longer. The *cwnd* graph, Figure 36, is also stable and similar to its *cwnd* graph in a large-sized buffer scenario.

4.4.3.4 Single Flow, Different RTT

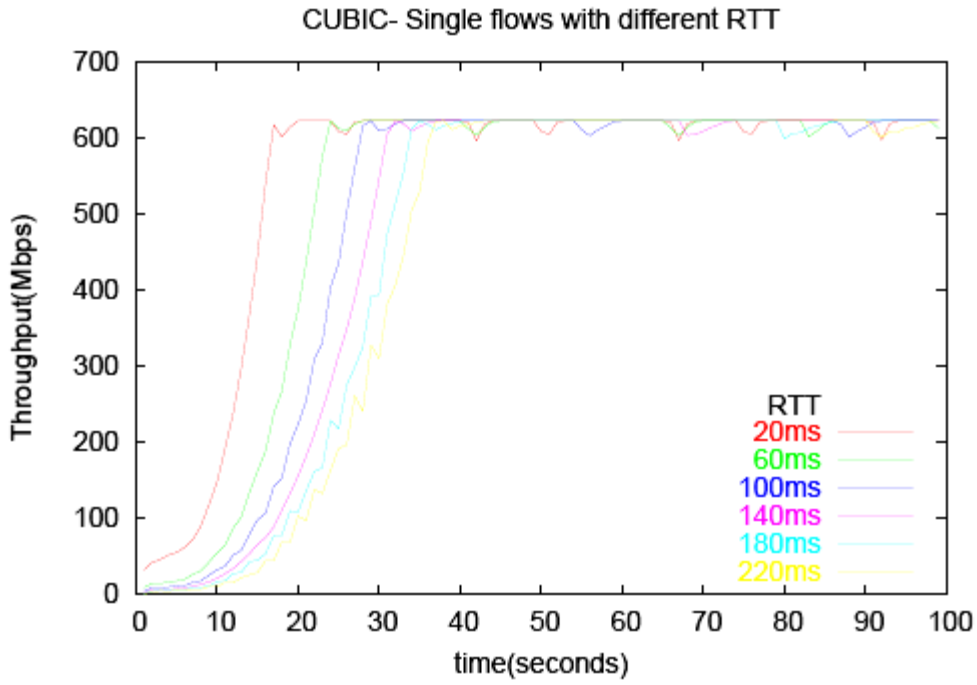

Figure 38: CUBIC Single flows with different RTTs

Figure 38 shows the throughput of a CUBIC flow when the RTT of the link is varied from 220ms to 20ms. CUBIC has a number of features to deal with RTT-fairness. In this scenario however, there are no competing flows and the difference in performance

of the flows is due to the difference in arrival rate of ACKs. Performance is expected to be better when there are competing flows on the link. We will test this in the next set of simulations.

4.4.3.5 Two Flows, Different RTTs

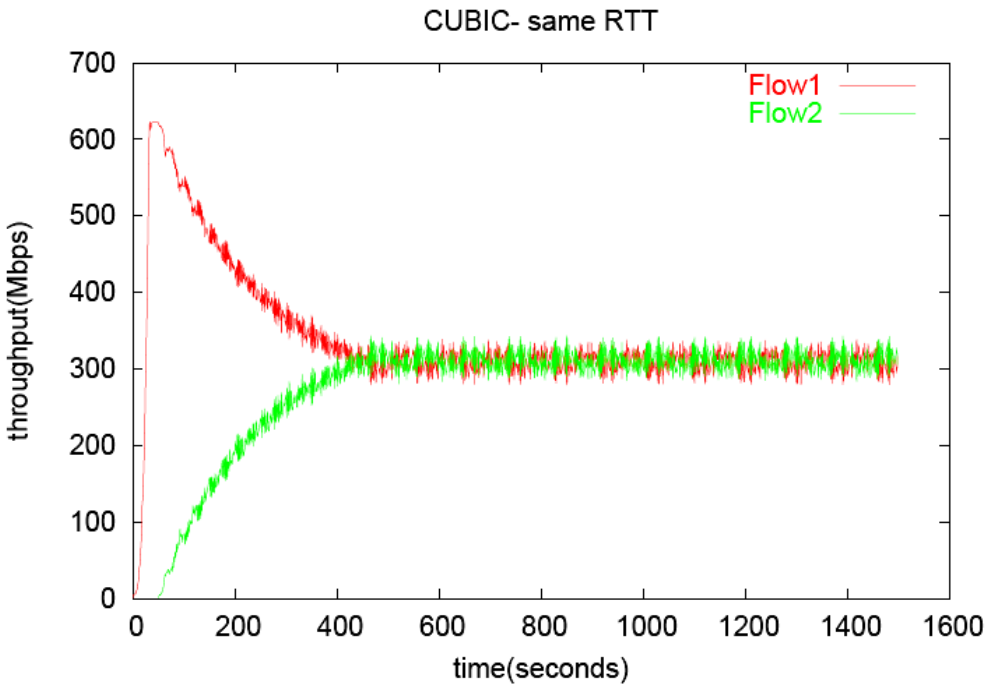

Figure 39: CUBIC 2 flows with same RTT started 50s apart

The RTT of the second flow was shortened from 162ms to 16ms. CUBIC performed impressively in this set of experiments. The sharing was always very fair. Figures 39, 40 and 41 show the throughputs of the competing CUBIC flows with different RTTs.

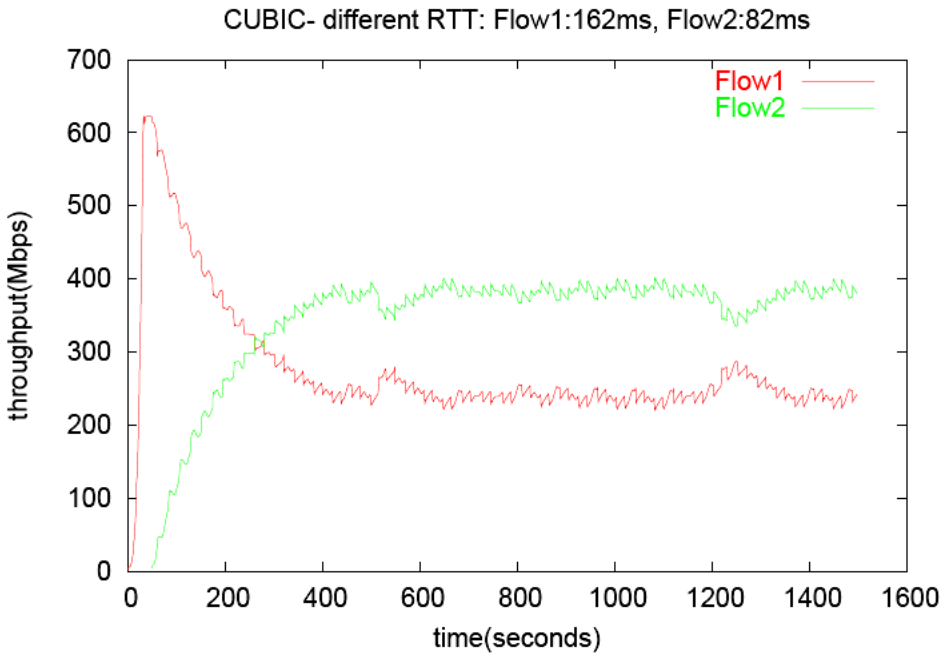

Figure 40: CUBIC 2 flows with different RTTs started 50s apart

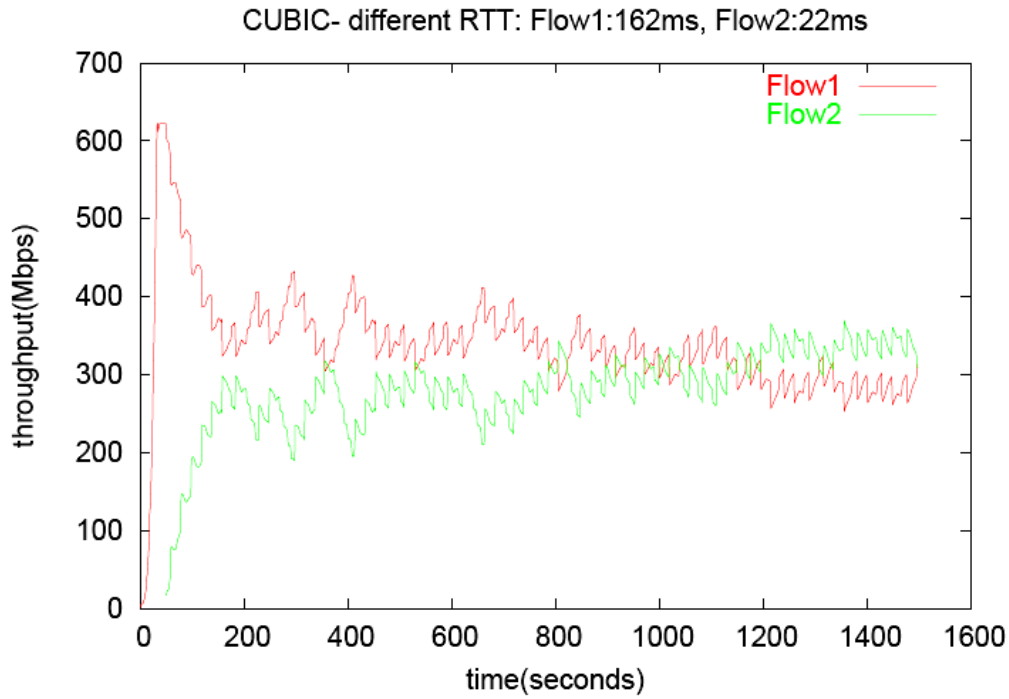

Figure 41: CUBIC 2 flows with different RTTs started 50s apart

4.4.4 Summary

CUBIC was slow in increasing its *cwnd* to fully occupy the link. It suffered a lot, more than BIC, when there was a packet drop early on in its first ascent because of the incorrect estimation of maximum link capacity. It had better performance than BIC-TCP in the limited bottleneck router buffer size scenario because of its greater stability as it nears its estimate of the link capacity. Its RTT-fairness mechanisms worked well in enforcing fairness.

4.5 FAST

FAST is a high-speed TCP protocol that is currently being developed at Caltech by Steven Low *et al*. [JWL04]. It has a delay-based approach that uses queuing delay as an additional indicator of congestion. It has four different modules: the window control module, the burstiness control module, the estimation module and a data control module. We are interested in the window control component since that is the one that is responsible for managing the congestion window. The estimation module calculates the *qdelay*, explained later, and also monitors packet losses. It provides these metrics as inputs to the window control module.

4.5.1 Congestion Control Algorithm

In this discussion, we use the following terminology:

avgRTT: current average RTT

baseRTT: the minimum RTT observed so far

α: the number of packets that a FAST flow attempts to maintain in the network buffer(s) at equilibrium

γ: ranges from 0+ to 1. In the current implementation gamma is set to 0.5.

cwnd: congestion window value

qdelay: weighted queuing delay

The window control component uses the following equation to manage *cwnd*:

$$cwnd = \min\{ 2cwnd, (1-\gamma)cwnd + \gamma(cwnd(\text{baseRTT}/\text{avgRTT}) + \alpha(cwnd, qdelay)) \}$$

α (*cwnd*, qdelay)= α**cwnd* when *qdelay* = 0

α (*cwnd*, qdelay)= α when *qdelay* $\neq$ 0

From the equation above, we can see that when *qdelay* is 0, *cwnd* will be usually doubled (for α>2). On the first sign of congestion, an immediate switch is made to a much less aggressive additive increase mode. The increments range from 0.5*α to possible negative values up to (*cwnd* -α)/2 (negative for α> *cwnd*). Thus, as the flow approaches link capacity, the increments becomes smaller and smaller leading to a healthy equilibrium. However, in the current implementation, α (*cwnd*, *qdelay*) always equals α irrespective of *qdelay*. *Cwnd* is updated periodically, for every other received ACK in the current implementation. It is also important to realize the importance of baseRTT in the equation. An over-estimation of its value will result in larger than normal increments and possible unfair sharing.

FAST currently uses the TCP loss recovery mechanism. Whenever it sees a loss, it deactivates the window update function and switches to TCP's loss recovery. It switches back to its window update function once TCP exits recovery.

The implementers of FAST in ns-2 recommend setting α such that α/C is greater than 5*mithresh, where C is the bottleneck speed in packets/ms and mithresh is the threshold of queuing delay that controls when FAST enters its multiplicative increase phase (set by default to 0.75 ms). With a 622 Mbps bottleneck, the link speed is 78 1000-byte packets/ms, so we set α to 4C, or 312 packets. This results in alpha/C = 4 ms, which is greater than 5*0.75=3.75 ms.

4.5.2 Unique Points

1. Queuing delay occurs much earlier than packet loss. The sequence is that the queue at the bottleneck link starts to build up on account of congestion caused by too much pressure from the feeding links. This increases the time a packet has to spend at the queue waiting for its turn to get transferred. This increase in queuing delay is an early, as well as a fine-grained indication of congestion. FAST uses queuing delay as a signal of congestion while the other protocols we study do not.

2. Another important aspect of FAST is that unlike the other protocols, it bases it decisions over a period of time (the current average RTT). This helps in ensuring that transient congestion does not harm the health of a FAST flow. Only when the congestion is sustained does a FAST flow really start to back off.

3. Even though FAST uses TCP's loss recovery mechanism, it still works well because it avoids losses well. It starts reacting at the first signs of congestion and this early reaction helps in easing of the pressure at the bottleneck which again results in a drop free environment.

4. FAST strives to have minimal (0) queuing delay. Packets do not have to wait in the queues, and the RTT comes down to the minimal level. This is clearly very advantageous.

4.5.3 Examples of the Protocol Operation

A number of simulations were run to test the various features of FAST. Network topology 1 (Figure 4) was used for the single flow simulations, and network topology 2 (Figure 5) was used for the two flows with different RTT simulations. The results are presented below.

4.5.3.1 Single Flow, No Enforced Packet Drops

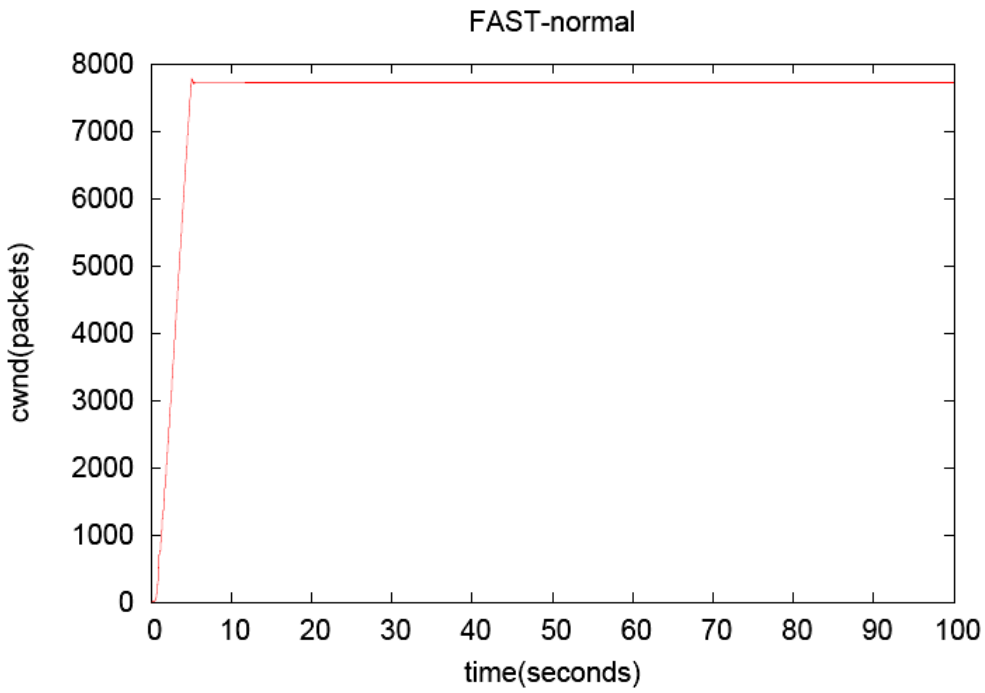

Figure 42: FAST Normal graph

In the single flow, no enforced packet drop experiment, the congestion window *cwnd* increases rapidly to the maximum value in 5.05 seconds and holds steady. Figure 42 shows the *cwnd* graph for the FAST flow.

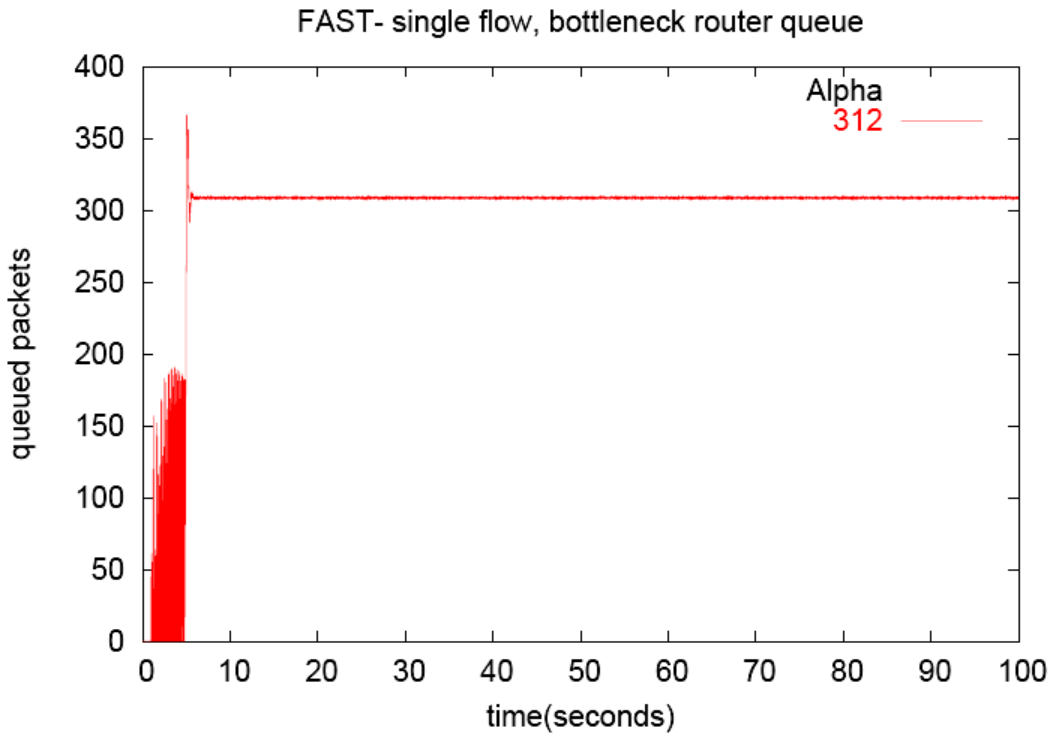

Figure 43: FAST Normal case bottleneck router queue size

In Figure 43, we show the number of queued packets at the router for the experiment shown in Figure 42. α is the number of packets a FAST flow attempts to keep in the router queue at equilibrium. Increments to *cwnd* are directly proportional to α. Looking at the queue size with α set to 312, which is the optimum value based on the network topology, we can see that there is a lot of volatility in the beginning. However, the queue size quickly stabilizes to the α value. This is the point that corresponds to the previous graph at which the maximum throughput is reached.

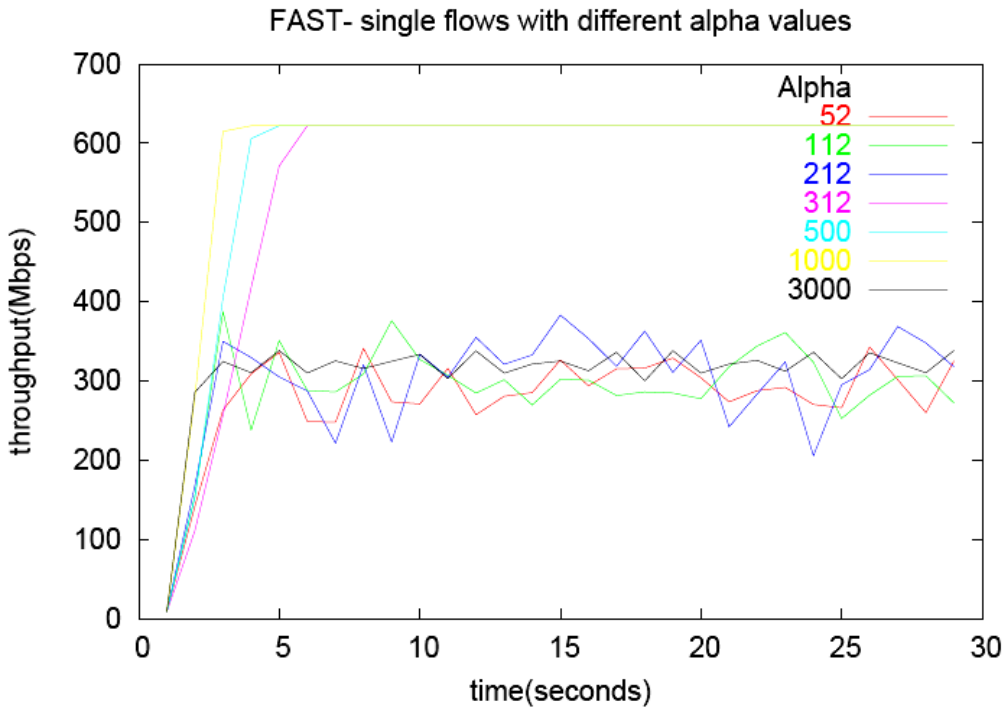

Figure 44: FAST Single flows with different alpha values

We ran tests with different α values to see how much effect that setting has on the performance. A setting of 312 is the optimum setting based on the network topology. We can see from the throughput graph in Figure 44 above that the flows with α below this setting perform poorly. Figure 45 shows queue sizes for α set to 212. The queue size never stabilizes and the network utilization is poor. As the value is increased to 312 and beyond, the utilization reaches 100%. α set to 312, Figure 43, which is the optimal setting and 500, Figure 46, which is greater than the optimal setting but less than the maximum queue size, 1555 packets, of the bottleneck router, result in 100% utilization. An α of 3000, Figure 47, results in the flow trying to have 3000 packets in the router queue causing packet drops and window halving resulting in a poor performance. So we see that α below the optimum value as well as α in excess of the queue capacity results in poor

performance. Another point to be noted here is that larger α values result in steeper rises to the maximum throughput. This is in line with our previous estimations.

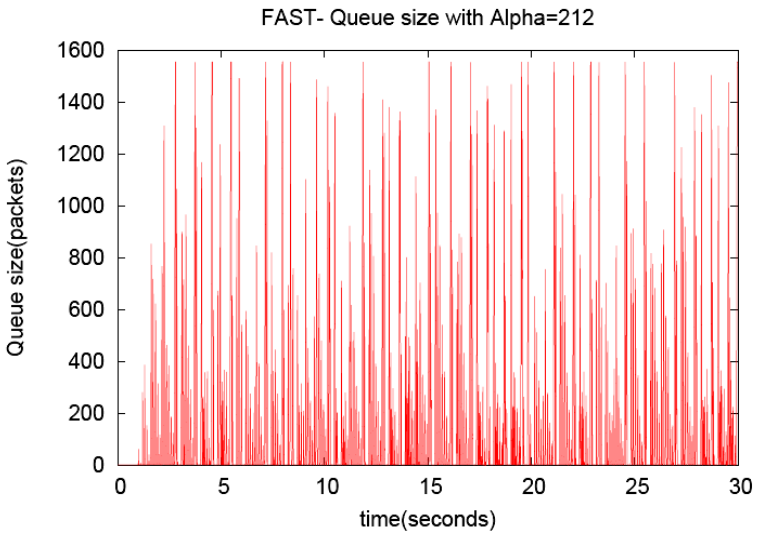

Figure 45: FAST Alpha value less than optimal setting

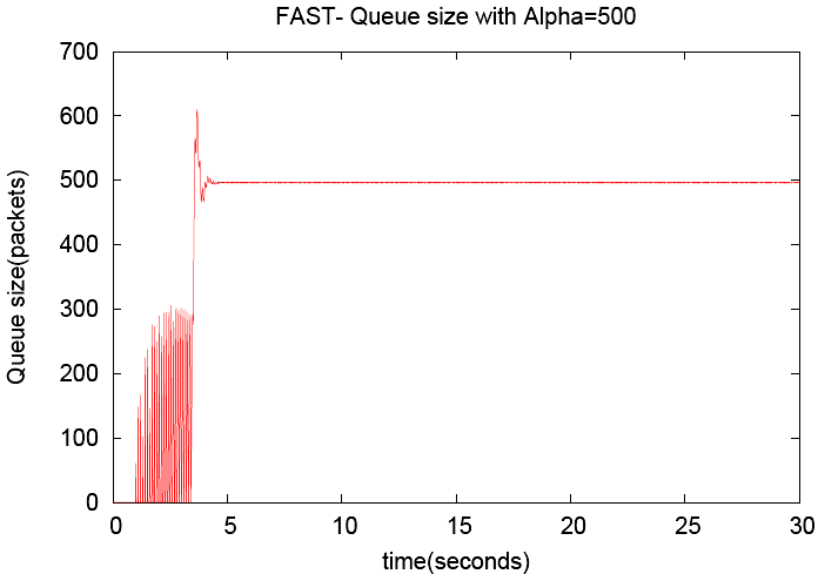

Figure 46: FAST Alpha value greater than optimal setting but less than maximum queue size

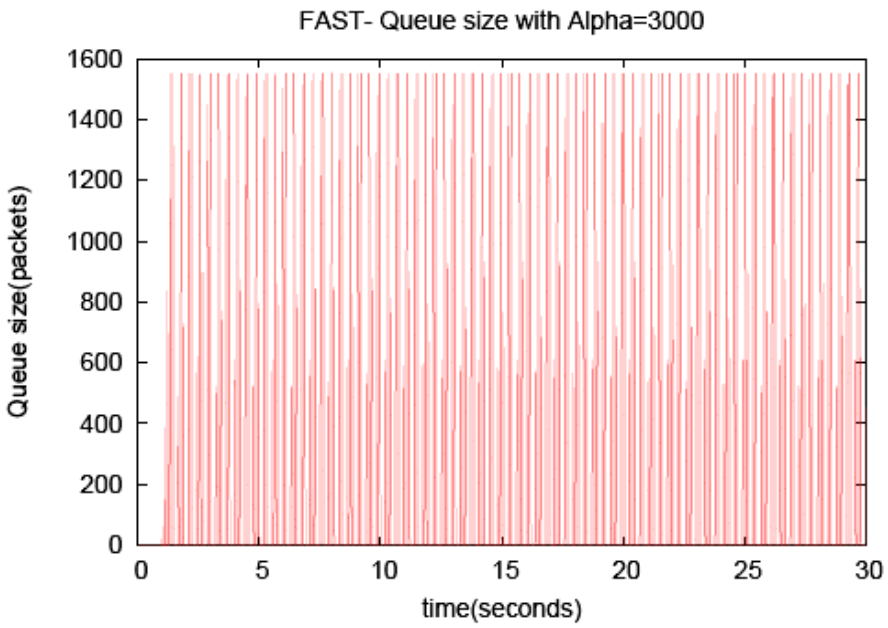

Figure 47: FAST Alpha value greater than maximum queue size

4.5.3.2 Single Flow, Single Drop

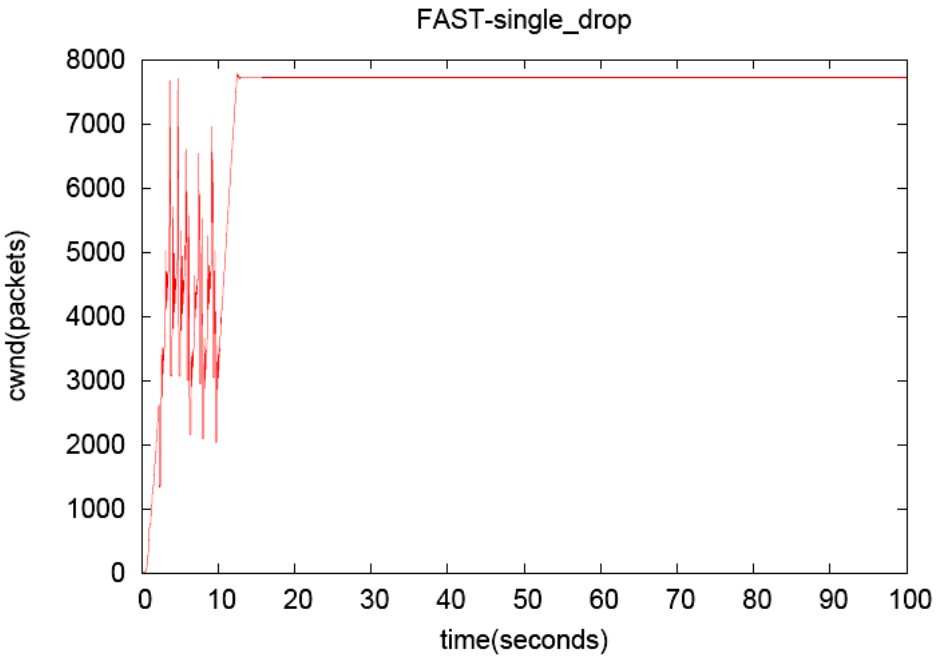

Figure 48: FAST Single drop

A packet was dropped when the FAST flow was in its initial ascent phase in this simulation. Figure 48 shows the *cwnd* of the FAST flow. It now reaches the maximum throughput in 13 seconds, which is 8 seconds slower than the normal case.

4.5.3.3 Single Flow, Different Buffer Sizes

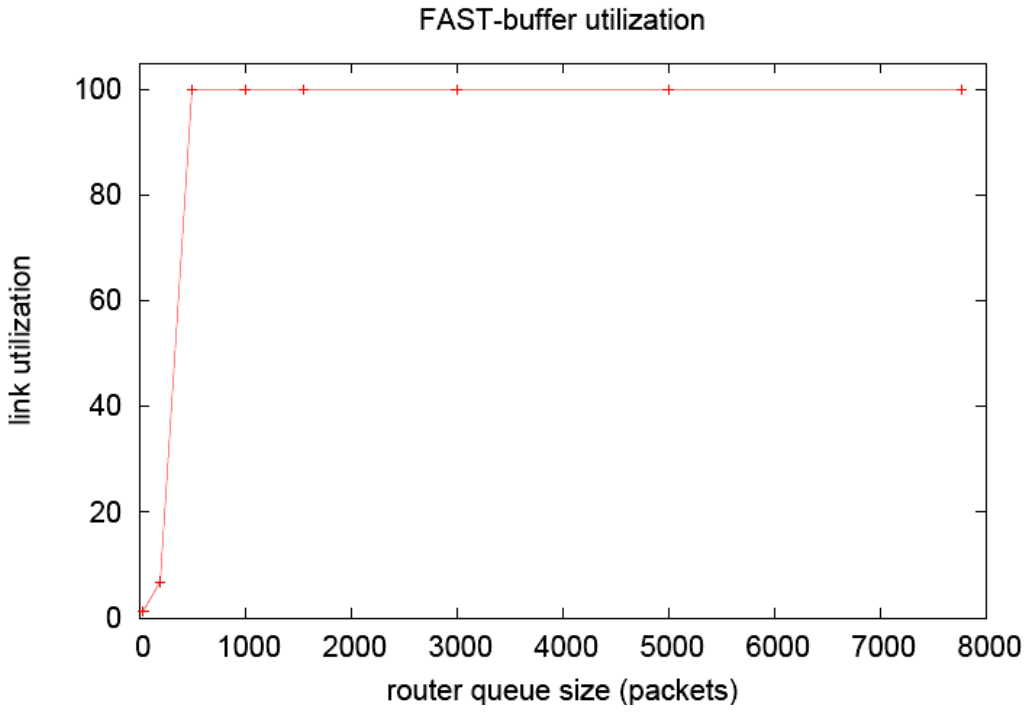

Figure 49: FAST Link utilization at different router queue sizes

Figure 49 shows link utilization for the different router queue sizes. FAST performs poorly for the smaller sizes. This is also expected as the α value is set based on the link capacity. In large bandwidth links, because of the enormity of the BDP, smaller queue sizes are typically used. We see poor performance whenever the router queue size

is smaller than α. The first two points in the graph are for the queue sizes corresponding to 40 and 200 packets which are smaller than 312, the α value.

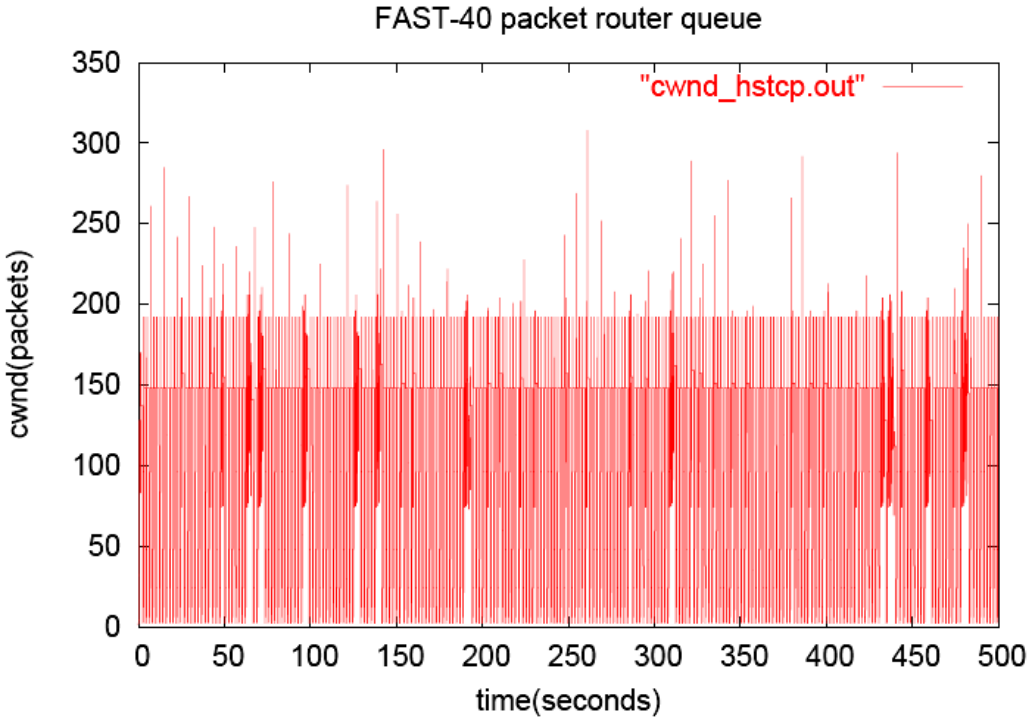

Figure 50: FAST Congestion window of flow with alpha value set to greater than router queue size

We see the *cwnd* of the FAST flow oscillating wildly in the 40-packet queue size scenario (Figure 50). It never settles to any sort of equilibrium and performance is very poor.

4.5.3.4 Single Flow, Different RTT

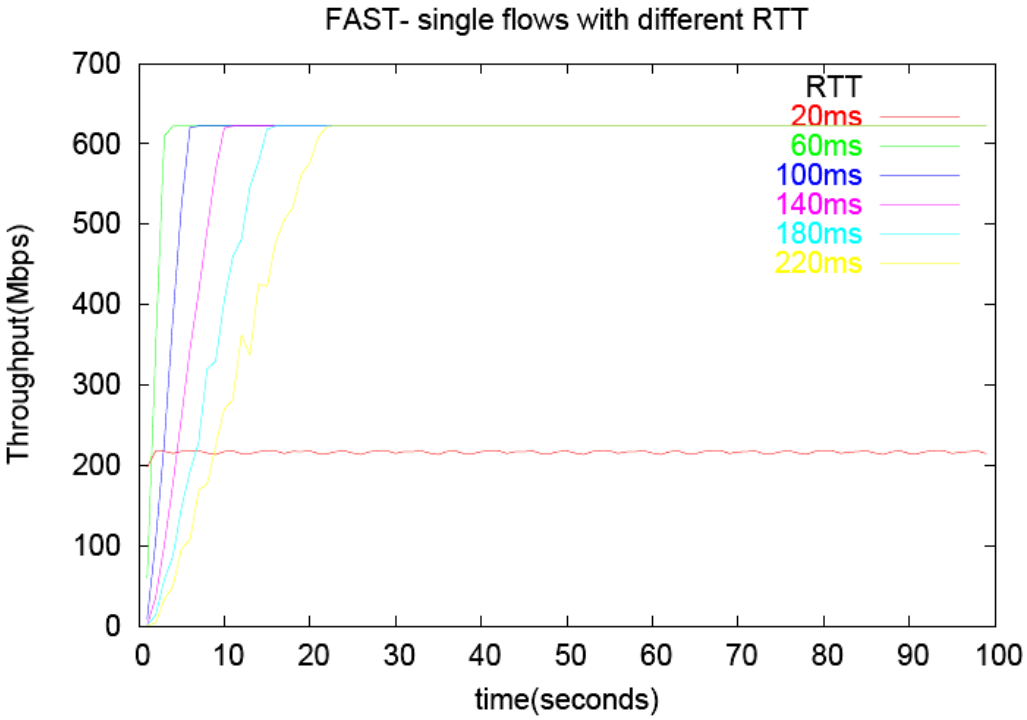

Figure 51: FAST single flows with different RTTs

Figure 51 shows the throughput of single FAST flow when the link RTT is varied from 220 ms to 20 ms. There was little difference in the performance of FAST flows. At very small RTTs however, FAST was unstable as we see in the *cwnd* graph below in Figure 52, and hence was unable to capture the entire available bandwidth. It stabilized when the RTT was increased to 60 ms and was able to reach 100% link utilization. Figure 53 shows the *cwnd* of the FAST flow with RTT of the link set to 60 ms. The difference in the amount of time it takes for the 60 ms and the 220 ms flows to reach the maximum link capacity was 18 seconds.

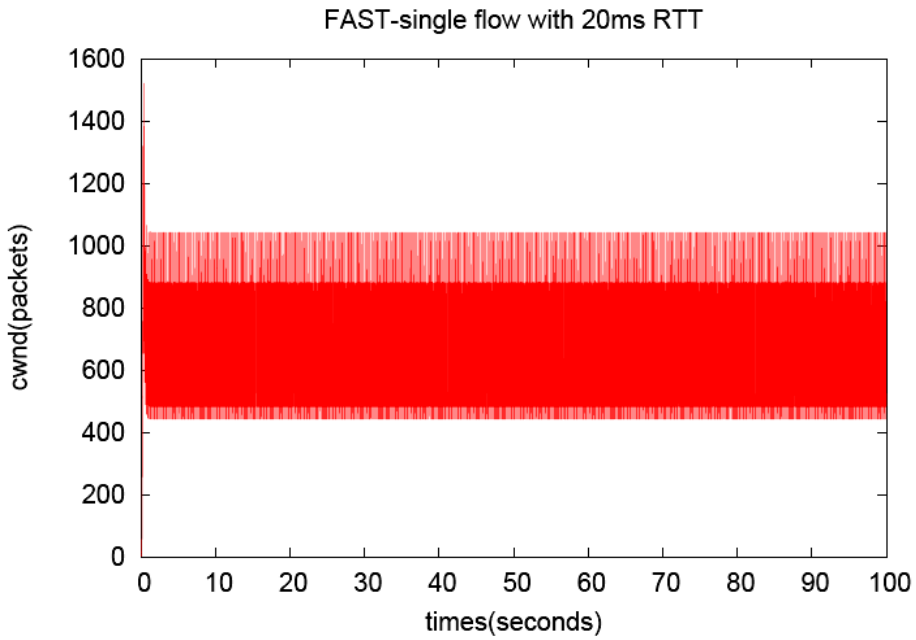

Figure 52: FAST Congestion window of flow on link with RTT 20ms

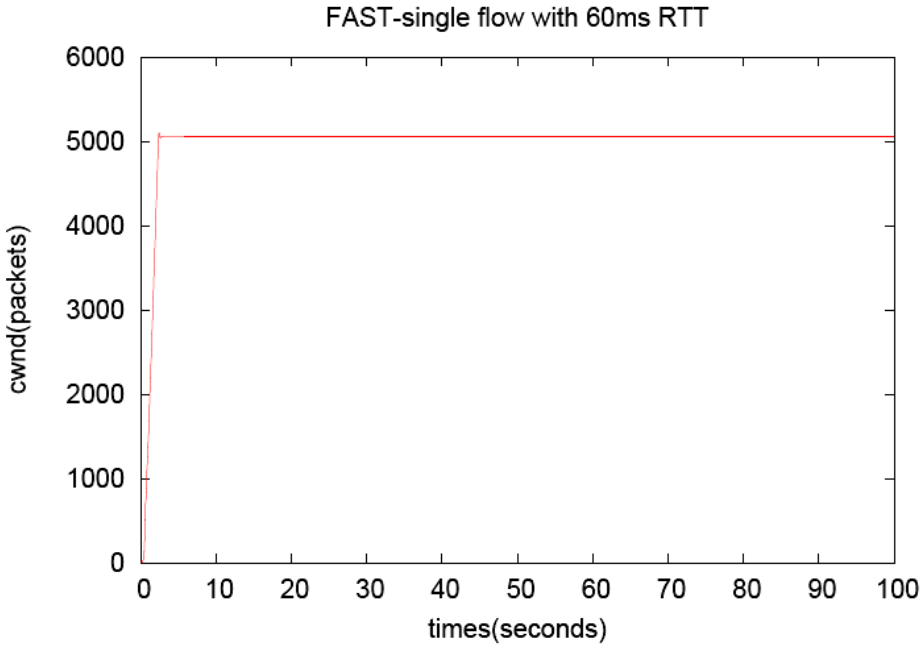

Figure 53: FAST Congestion window of flow on link with RTT 60ms

A possible reason for this difference in performance for the 20ms and 60ms RTT flows is that FAST is a delay-based protocol, and with smaller and smaller RTTs, the percentage of the queuing delay in the total delay increases. As it crosses a point, which in our experiments is between 60ms and 20ms, FAST is unable to maintain its stability resulting in this behavior.

4.5.3.5 Two Flows, Different RTTs

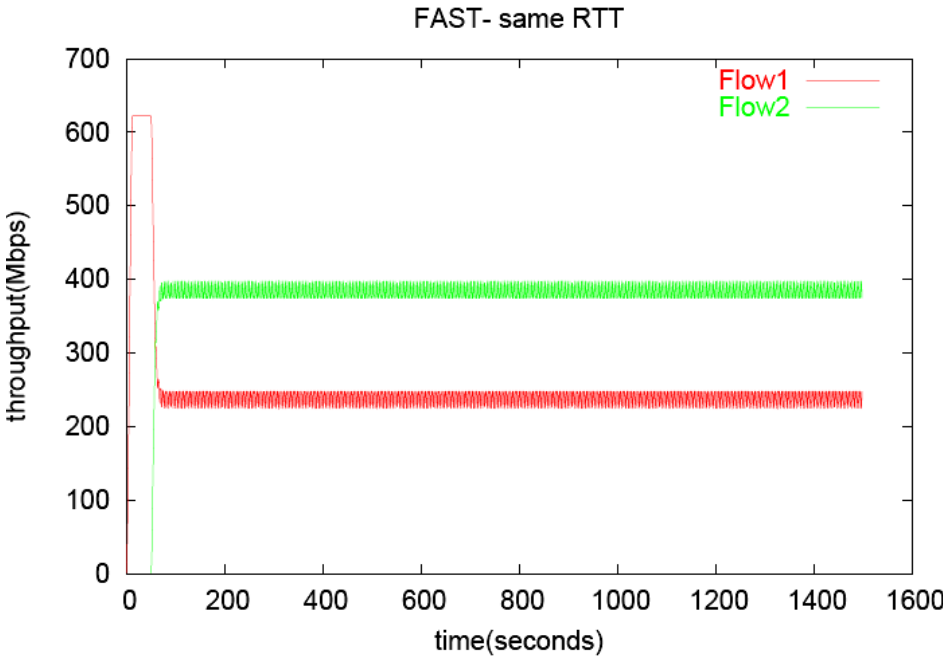

Figure 54: FAST 2 flows with same RTT started 50s apart

FAST shares unfairly even when the two flows have the same RTT. Figure 54 shows the throughput of the competing FAST flows that have the same RTT. This is because of the incorrect estimation of baseRTT by the second flow. When the first flow starts there are no packets in the router queue and a correct estimation of baseRTT is

made. However as the second flow starts, since the first flow is already occupying the link and has α packets queued in the bottleneck router queue, a larger incorrect estimate of baseRTT is made. This results in the second flow having a larger share of the link bandwidth. However, had there been numerous other flows on the link, this result would not have been true. We would have seen a different result since every flow would see an approximately equal amount of delay while starting up assuming there was even traffic throughout. Sharing would be fair in that case.

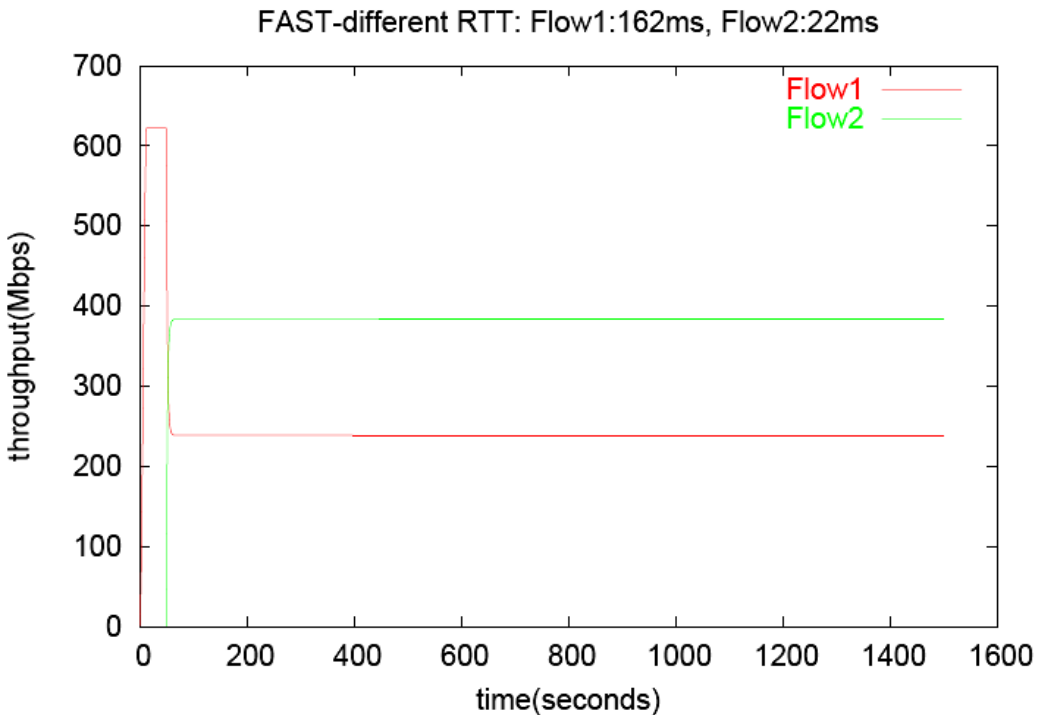

Figure 55: FAST 2 flows with different RTT started 50s apart

The results do not change a lot as we shorten the RTT of the second flow to 22ms. Figure 55 displays the throughput of FAST flows for this case. The only difference is because of incorrect estimation of the RTT as explained earlier.

4.5.4 Summary

FAST had the fastest initial *cwnd* increase. The artificial packet drop in the ascent phase did not have much effect on its performance as it makes not estimate of the link capacity but only of link RTT. Severe link congestion early on would have had a substantial effect on its performance. We see this in the multiple flows on a single link scenario where the second flow makes an incorrect estimate of minimum RTT and is more aggressive every time. The performance in limited router buffer environment is very dependant on the α value. It is essential to know the link bandwidth to have the correct value for α. This might not always be possible in the real world and would hamper FAST's performance.

4.6 H-TCP

H-TCP is a TCP congestion control modification proposal from the Hamilton Institute, Ireland [LLS05]. It has been designed to function efficiently in both high BDP and low BDP environments. It differs from most of the other congestion control algorithms in its modeling of the congestion window increase and decrease functions on the time differences between congestion events rather than on previous congestion window values.

4.6.1 Congestion Control Algorithm

In this discussion, we use the following terminology:

α**:** Increase Parameter

β: Decrease Parameter

Δ: Time since last congestion event

Δ^L: Threshold for switching from low speed to high speed mode

$RTT_{min:}$ Minimum Round Trip Time

$RTT_{max:}$ Maximum Round Trip Time

cwnd: Congestion window value

On the arrival of an ACK:

(i) $\alpha = 1 \qquad \Delta \leq \Delta^L$

$= 1 + 10(\Delta - \Delta^L) + ((\Delta - \Delta^L)/ 2)^2 \qquad \Delta > \Delta^L$

(ii) $\alpha = 2(1- \beta)\, \alpha$

(iii) $cwnd = cwnd + \alpha/cwnd$

On a packet loss:

(i) $\beta = RTT_{min} / RTT_{max} \qquad \beta \in [0.5,0.8]$

(ii) $cwnd = \beta * cwnd$

The algorithm is an AIMD algorithm.

4.6.2 Unique Points

1. H-TCP models the increase parameter on basis of time difference between successive congestion epochs (Δ) as opposed to functions of *cwnd*. The minimum threshold also has a time value rather than a *cwnd* value. If the time evolved since the last congestion (Δ) is below 1 second, then H-TCP behaves like standard TCP. The regular conventional networks (small BDP) typically have small time differences in successive congestion events. H-TCP is thus able to adapt its behavior based on the size of the link BDPs. For smaller BDP links it behaves like standard TCP and for higher BDP link it behaves differently.

2. H-TCP uses an *Adaptive Back-off Algorithm* to deal with backing off in face of congestion. Rather than having a fixed value for backing-off, it calculates the value based on the RTTs before and after the back-off, $\beta = RTT_{min} / RTT_{max}$. In addition to this, for situations that call for a rapid back-off (change in throughput before and after the congestion event exceeds 20%), H-TCP uses a β of 0.5. A larger β value helps in quicker response while a larger value results in a more efficient performance. The maximum value is capped at 0.8. This value gives good performance when queue size is limited to up to 25% of BDP.

3. ***RTT scaling*:** H-TCP employs *RTT-scaling* to overcome RTT unfairness issues in scenarios where flows with different RTTs compete for link bandwidth. It uses $k*\alpha$ instead of just α to handle this, where $k = RTT/RTT_{ref}$. RTT scaling is switched off when Δ is below the threshold to maintain backward compatibility with conventional low BDP network links. This approach is more dynamic compared to BIC's approach where there was a just a fixed amount of advantage that the longer flow got. Once the disadvantage due to the longer RTT outgrew the advantage, there was again unfairness. However, since H-TCP increases the counterweight proportionally to the RTT, this is handled much better.

4.6.3 Examples of the Protocol Operation

A number of simulations were run to test the various features of H-TCP. Network topology 1 (Figure 4) was used for the single flow simulations, and network topology 2 (Figure 5) was used for the two flows with different RTT simulations. The results are presented below.

4.6.3.1 Single Flow, No Enforced Packet Drops

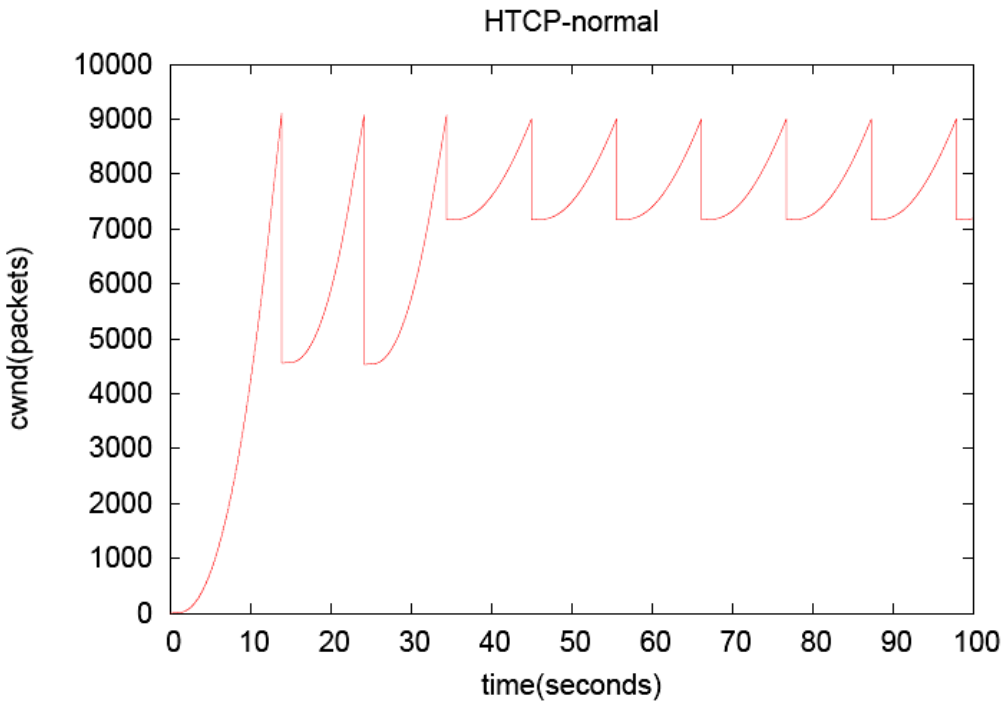

Figure 56: H-TCP Normal graph

In the single flow, no enforced packet drop experiment, the congestion window, shown in Figure 56, grows rapidly to its maximum value in 13 seconds. The reductions in the congestion window are due to losses that the flow itself is causing at the bottleneck link. The first halving of *cwnd* is similar to the behavior seen with HS-TCP and Scalable-TCP and is due to the way the protocol has been implemented in ns-2. The second halving is because of H-TCP's adaptive back-off algorithm which sets the back-off parameter to 0.5 when the difference in throughput after a congestion event exceeds 20%.

4.6.3.2 Single Flow, Single Drop

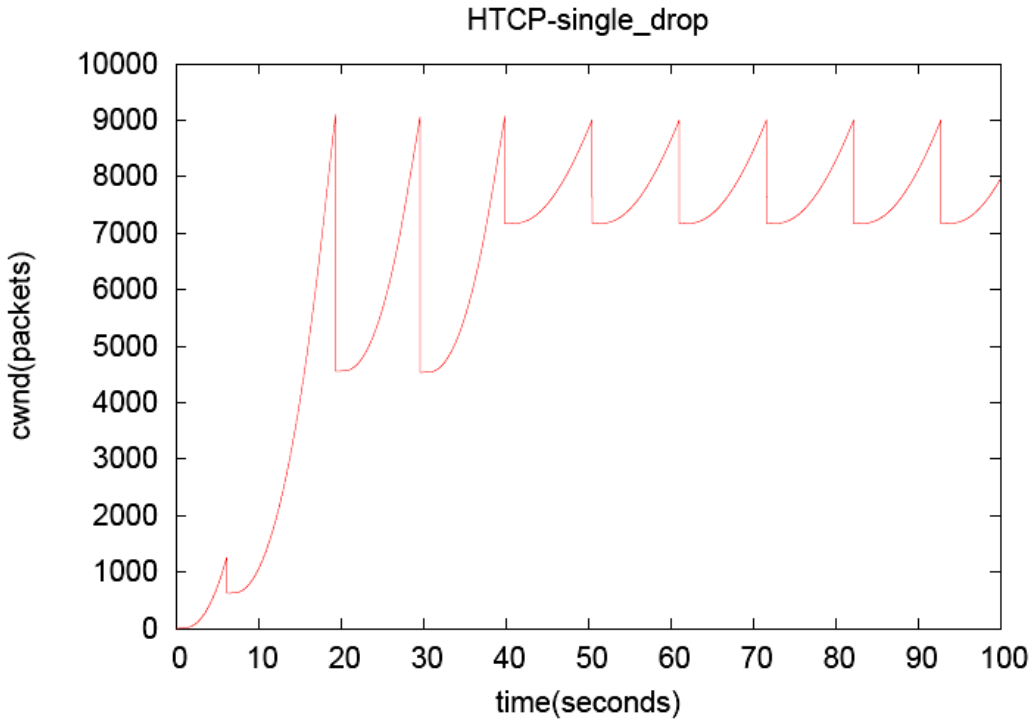

Figure 57: H-TCP Single drop

In the single flow, single drop experiment a packet is artificially dropped in the ascent phase. This does not have a substantive affect on the time taken to reach link capacity which is 19 seconds in this case. Figure 57 shows the *cwnd* of the H-TCP flow.

4.6.3.3 Single Flow, Different Buffer Sizes

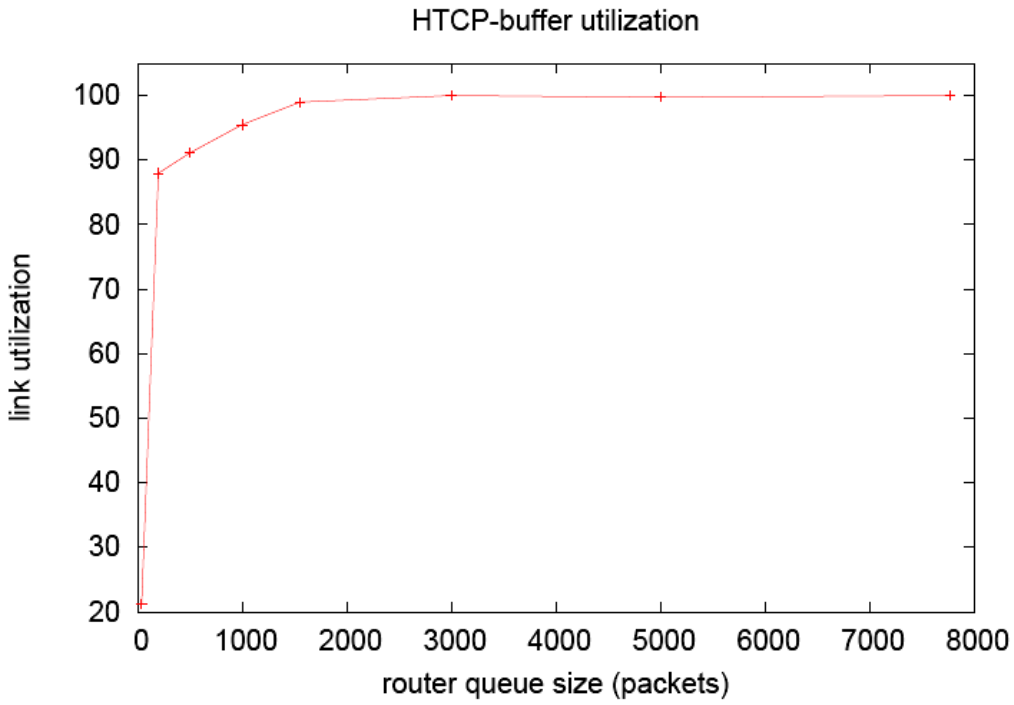

Figure 58: H-TCP Link utilization with different router queue sizes

In the single flow, different buffer sizes experiment where a single flow is run through the bottleneck link with varying router buffer sizes; the link utilization, shown in Figure 58, is very poor for the 40-packet router queue case. With such a small router queue, the value of RTT_{min} / RTT_{max} approaches 1. However β is capped at 0.8 to have a faster response time. This makes the H-TCP flow back-off too much leaving the queue empty for long periods of time.

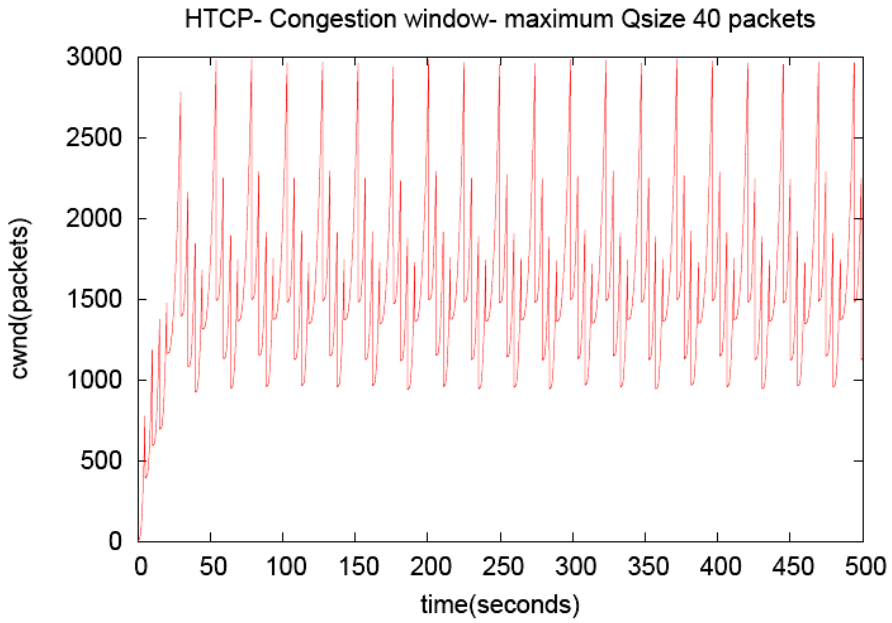

Figure 59: H-TCP's cwnd graph in small router buffer environment

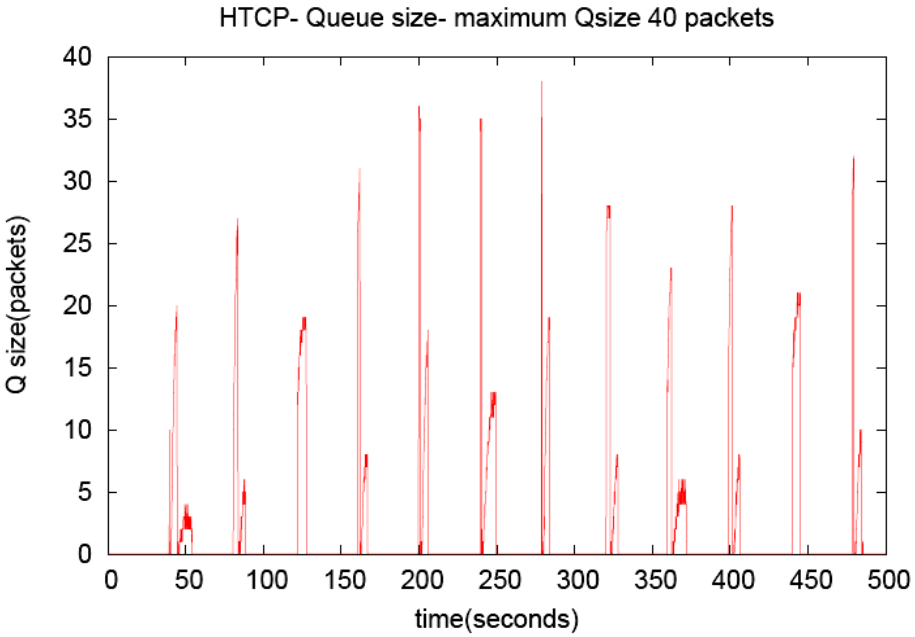

Figure 60: H-TCP's queue occupancy graph in small router buffer environment

Figures 59 and 60 display the *cwnd* and router queue size graphs when the router queue size is limited to 40 packets. The queue is empty for large amounts of time leading

to a poor utilization. As the router queue size is increased to 500 packets, performance improves considerably. Figures 61 and 62 display the *cwnd* and router queue size graphs for this case. We can see in Figure 61 that *cwnd* is reduced by a factor of 0.8 every time a loss happens. This is again due to RTT_{min} and RTT_{max} being close together.

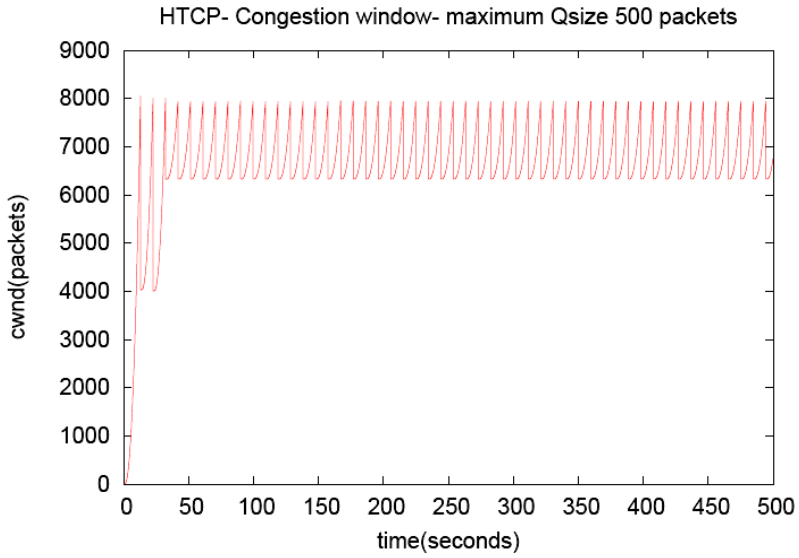

Figure 61: H-TCP's cwnd graph in 500 packet router buffer environment

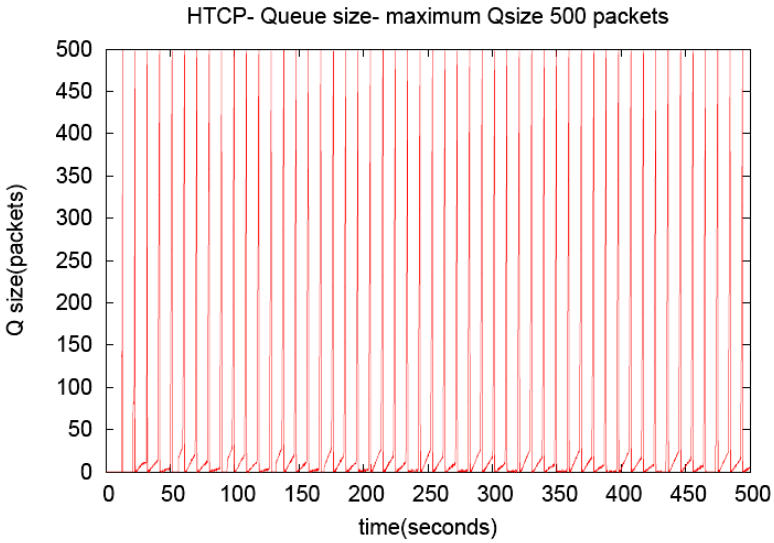

Figure 62: H-TCP's queue occupancy graph in 500 packet router buffer environment

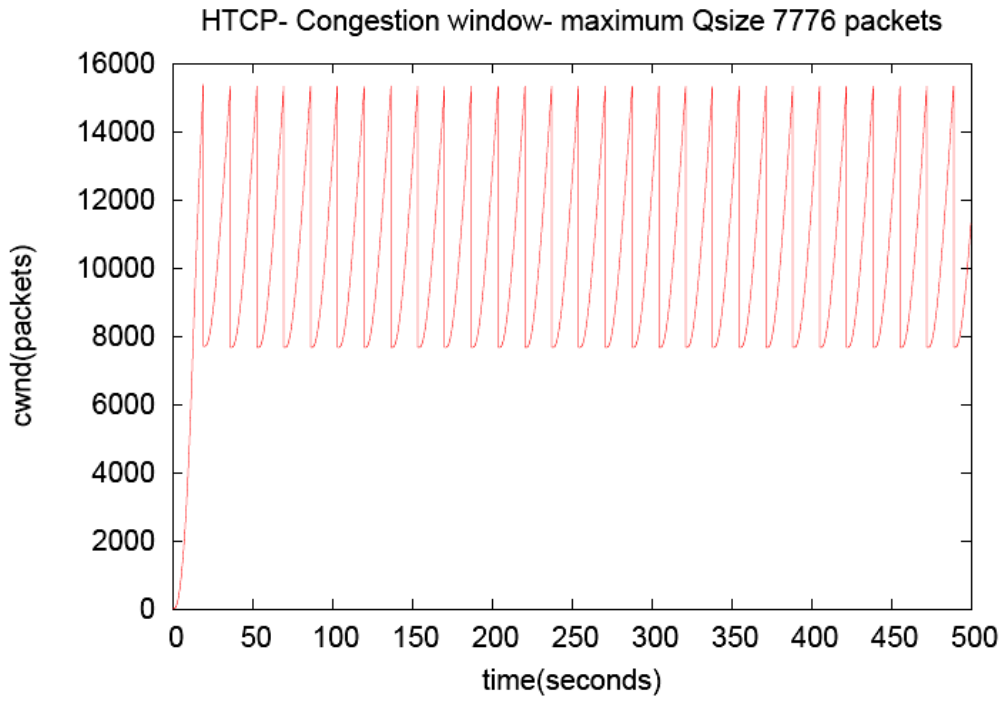

Figure 63: H-TCP's cwnd graph in large router buffer environment

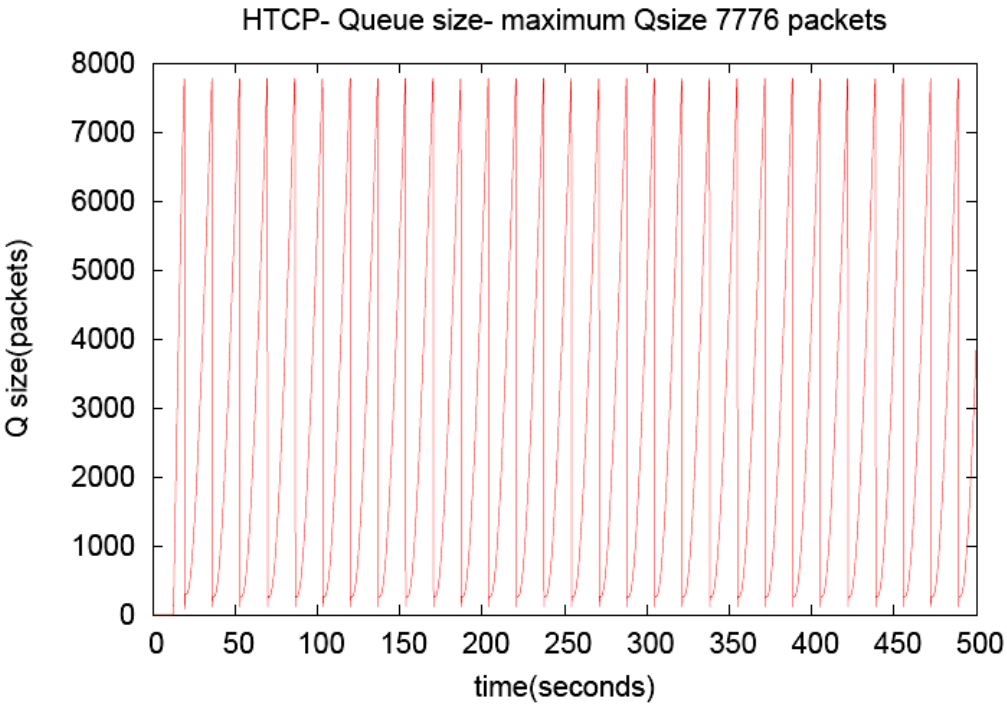

Figure 64: H-TCP's queue occupancy graph in large router buffer environment

Figure 63 shows H-TCP's *cwnd* graph when the router buffer size is 7776 packets. There is ample queuing delay in this case and the value of RTT_{min} / RTT_{max} is significantly lower. This results in β being set to 0.5 every time. This is evident from the graph as the packet losses reduce *cwnd* by half whenever packet drops happen. Figure 64 shows the graph for the router queue sizes for the same scenario. We can see that the queue never drains out completely. This is due to β being limited to 0.5. A smaller value would have been able to drain the queue completely.

4.6.3.4 Single Flow, Different RTT

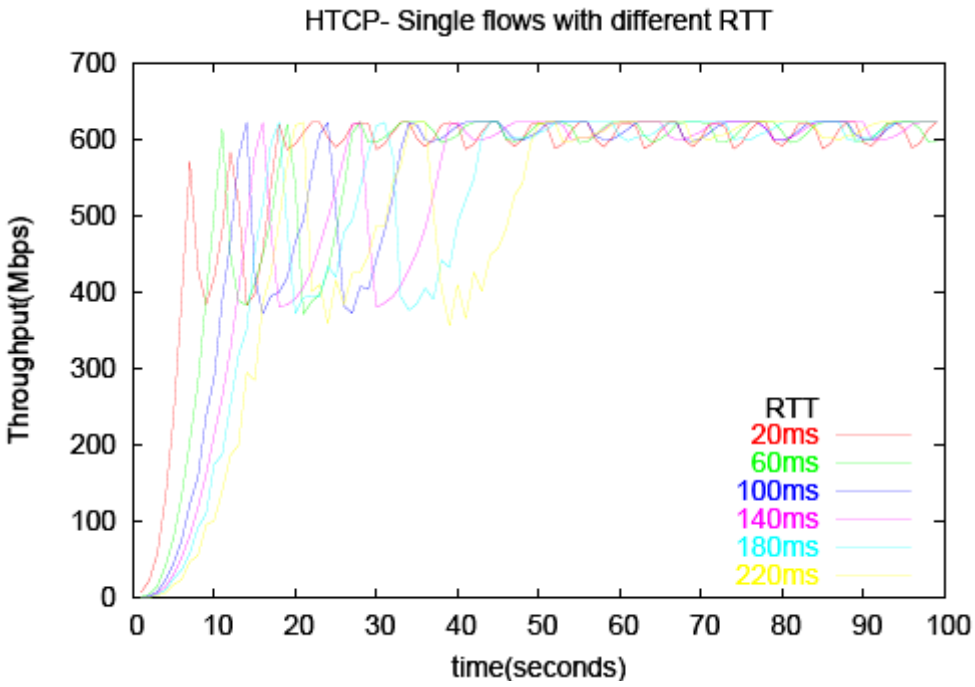

Figure 65: H-TCP Single flows with different RTTs

In this set of experiments, single H-TCP flows were run with varying link RTTs. H-TCP has explicit RTT scaling mechanisms. This should translate in to flows showing

the same degree of aggressiveness at different RTTs. Figure 65 shows the throughput of a single H-TCP flow for the different RTTs. We can see from the graph that this does happen. There is very little difference in performance of flows at different RTTs. The 20ms flow is able to fully utilize the link in 18 seconds while the 220ms flow takes 20 seconds to do that. Based on this observation we expect H-TCP to be RTT-fair.

4.6.3.5 Two Flows, Different RTTs

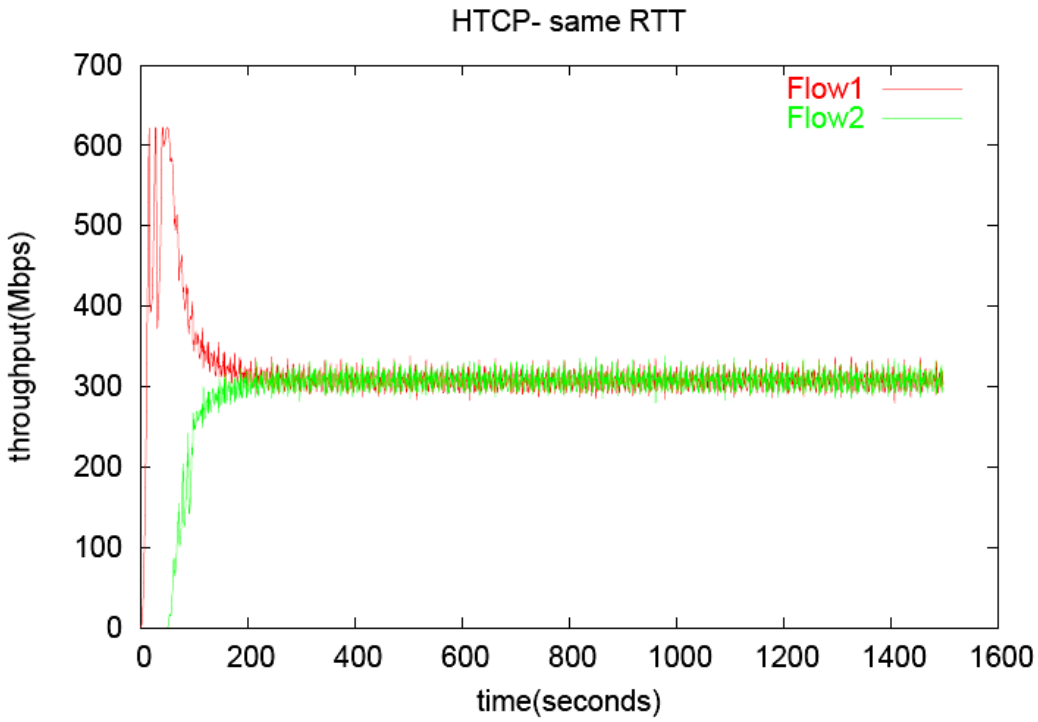

Figure 66: H-TCP 2 flows with same RTT started 50s apart

H-TCP is completely fair when the two flows have the same RTT. Figure 66 shows the throughput of the competing H-TCP flows when both have the same RTT. The convergence is much faster than CUBIC (Figure 39). This is due to the fast

responsiveness of H-TCP. Figures 67 – 69 present fairness results for different RTTs. We can see that H-TCP is generally fair compared to the other protocols and the convergence time for the two flows is also smaller.

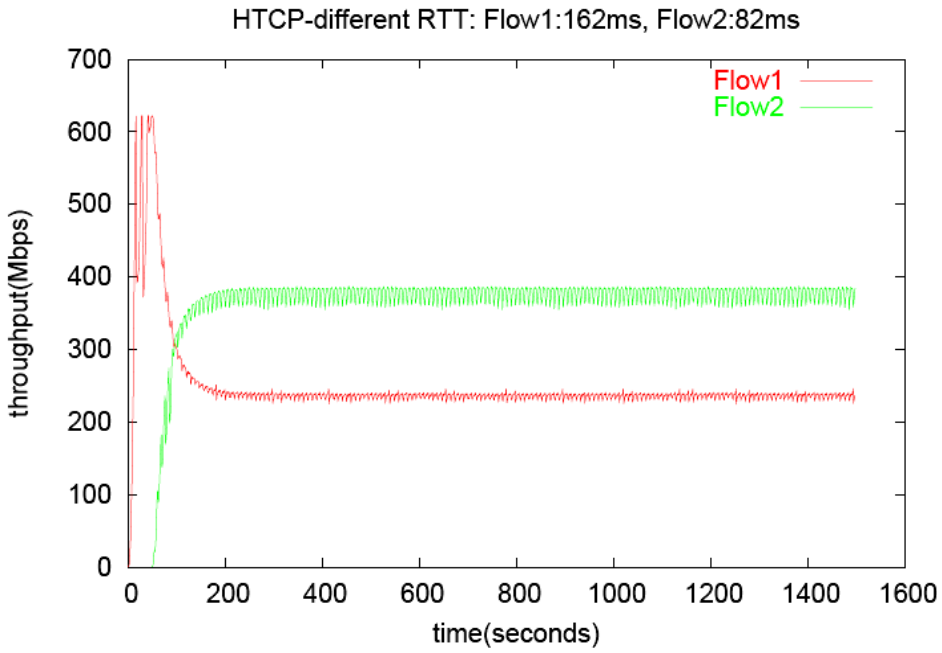

Figure 67: H-TCP 2 flows with different RTT started 50s apart

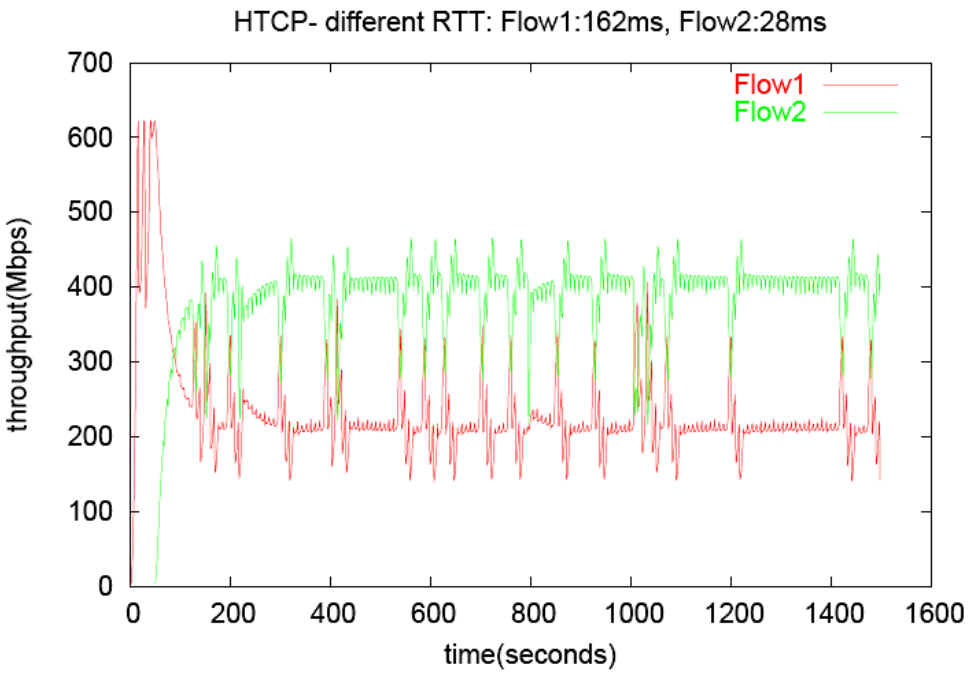

Figure 68: H-TCP 2 flows with different RTT started 50s apart

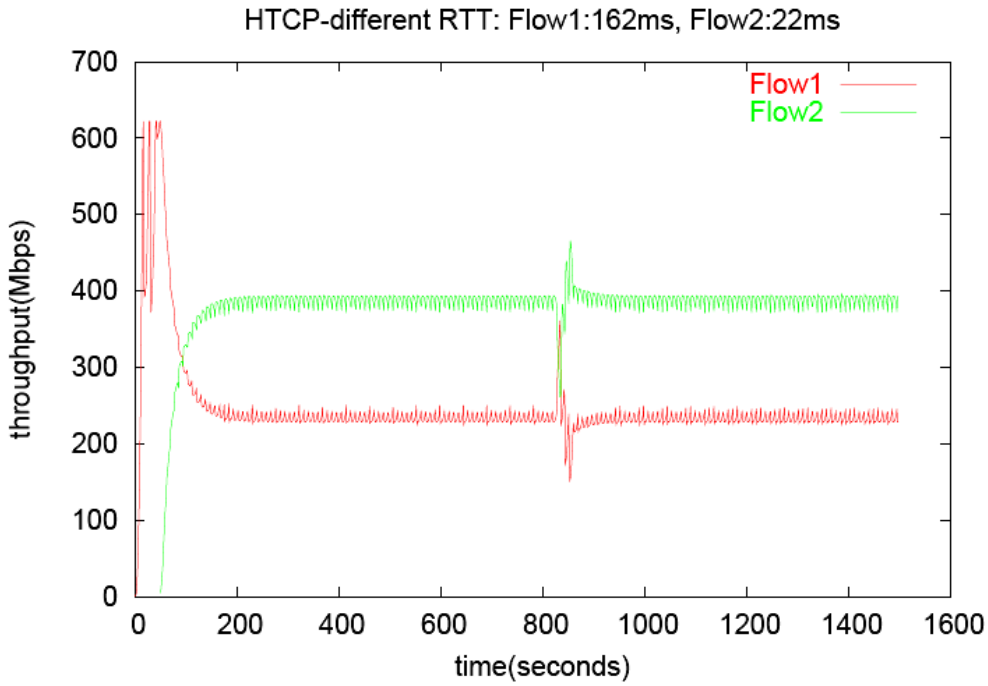

Figure 69: H-TCP 2 flows with different RTT started 50s apart

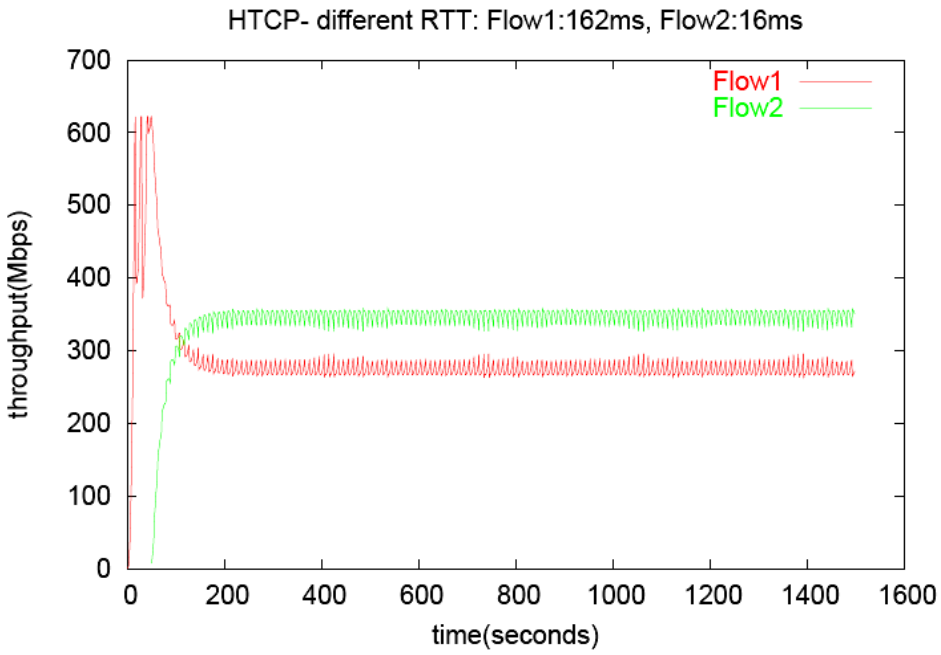

Figure 70: H-TCP 2 flows with different RTT started 50s apart

4.6.4 Summary

The artificial packet drop in the initial ascent phase of H-TCP does not affect its performance much since it does not make any estimates of the link capacity. It performs poorly in very small router queue environments because it does not stabilize as it approaches link capacity. Its back-off parameter is constricted in the range [0.5,0.8]. The upper limit is imposed to have good responsiveness while the lower limit is imposed to have good utilization. However, the upper limit forces it to back-off too much when the router queue is very small leading to a poor utilization. RTT-scaling is very effective in imparting the same level of aggressiveness to all H-TCP flows. H-TCP also showed good RTT-fairness.

4.7 Summary

The results from the intra-protocol simulations are summarized here. Section 4.7.1 shows the results from the single flow experiments, while Section 4.7.2 has the results from the multiple flows with different RTTs simulations.

4.7.1 Single Flow Results

Table 1: Results from intra-protocol simulations -1

Protocol	Max tput (Mbps)	Time to reach 90% link utilization (seconds)	Time to reach 90% link utilization when loss occurs during the first ascent (seconds)	Link utilization for buffer size of 40 packets	Buffer for which more than 90% utilization (packets)	Time (seconds) to reach 90% link utilization with RTT as		Time difference in reaching maximum utilization with the two border RTTs (seconds)
						20ms	220 ms	
HS-TCP	622	14.58	44.73	88.76	200	2.7	33.7	31
Scalable TCP	622	13.96	26.50	49.80	200	1	34.6	33.6
BIC-TCP	622	9.53	30.74	90.47	40	1.7	21.6	19.9
CUBIC	622	26.55	104.01	90.57	40	15.7	27.2	11.5

FAST	622	4.64	12.08	1.27	500	*60ms*[1] - 2.1	11	8.9
H-TCP	622	12.39	17.78	21.18	500	16.8	13.9	-2.9

Results are shown for 90% link utilization to be fair to BIC-TCP and CUBIC which slow down considerably as they near their estimate of link capacity. We can see from Table 1 that all the protocols were able to reach full link utilization in the normal case, where RTT and bottleneck router queue size were not limited. FAST was the first to get to 90% link utilization in 4.64 seconds. CUBIC was slowest. A packet drop in the first ascent had the largest effect on CUBIC and it was able to reach 90% link utilization only in 104.01 seconds. As we saw previously, BIC-TCP and CUBIC were the best performers when the bottleneck router buffer queue size was limited to 40 packets owing to their nature of becoming very gentle when they near their estimate of the link capacity. H-TCP's RTT-scaling mechanism helped it to have the best performance in the single flow with different RTT set of experiments.

[1] FAST never reaches 90% link utilization with 20ms RTT. The result from 60ms RTT experiment is used.

4.7.2 Intra-Protocol Results

Table 2: Results from intra-protocol simulations-2

Protocol	Fairness values for flows with different RTTs Flow1 RTT (ms) : Flow2 RTT(ms)						
	162:16	162:22	162:28	162:42	162:58	162:82	162:168
HS-TCP	-0.964	-0.950	-0.882	-0.915	-0.867	-0.727	0.241
Scalable TCP	-0.307	0.993	0.995	0.995	0.997	-0.651	0.961
Scalable TCP-2[2]	0.998	0.998	0.996	0.998	0.998	0.997	0.577
BIC-TCP	-0.968	-0.950	-0.746	-0.915	-0.886	-0.762	0.370
CUBIC	-0.296	0.080	0.204	-0.056	-0.082	-0.133	0.079
FAST	-0.467	-0.232	-0.233	-0.231	-0.232	-0.231	-0.230
H-TCP	-0.107	-0.235	-0.273	-0.322	-0.285	-0.213	0.014

Table 2 shows the fairness results when two flows from the same high-speed TCP competed against each other for the share of the bottleneck link bandwidth. The flow with the larger RTT was started first except for the case of Scalable-TCP-2 where the flow with shorter RTT was started first. A result value close to 0 indicates a fair sharing. Values close to 1 indicate that the first flow was dominant while values close to -1 indicate that the second flow was dominant. Most of the results showed very unfair sharing. H-TCP, CUBIC and FAST were the exceptions. H-TCP and CUBIC have explicit mechanisms for RTT-fairness and their efficiency was proven in this set of

[2] **Scalable TCP-2:** Case when the shorter RTT flow is started first

experiments. H-TCP was however always faster in converging to a fair share and thus had a better overall performance. FAST's intra-protocol sharing was independent of the RTT.

CHAPTER 5

INTER-PROTOCOL EXPERIMENTS

We test the various high-speed TCP protocols against each other to see how fair they are when confronted by different congestion control algorithms. Network topology 2 will be used to run this set of tests. Two flows are run across a single bottleneck link. We stagger the start times of the two flows by 50 seconds to allow one flow to dominate the link before the second flow starts. We calculate the *asymmetry metric* at the end of 250 seconds to determine how fair the sharing was. All 36 combinations of HS-TCP, Scalable TCP, BIC-TCP, CUBIC, FAST and H-TCP are tested.

5.1 Expected Results

We would ideally want the congestion control algorithms to be robust enough to be able to deal with flows that are more/less aggressive in a suitable manner by still only taking their fair share of the bandwidth. However, this is a very difficult problem to handle as there is no feedback from the network link to the hosts on their actual fair share. Queuing delays and packet drops are the only signals of congestion that the end hosts get and they have to base their decision of how aggressive they need to be solely on this. We say that a parameter is fixed if a single value is always used for that parameter. An example for this is the increase parameter for Scalable-TCP which is always 1.01. We call a parameter variable when its value differs in different scenarios. H-TCP's back-off

parameter is an example of this as it varies in the range of [0.5,0.8] based on RTT_{min} / RTT_{max}. Consider a scenario where there are two flows on a link each belonging to a different protocol. Let us also suppose that both of these protocols have fixed increase and decrease parameters and one of the protocols is more aggressive than the other one. Both the protocols will spot congestion only when the link approaches its full capacity. At this point, the more aggressive protocol would have enabled its flow to take up majority of the bandwidth. They would then cut back with different factors, assume these to be same, thinking that they have crossed their fair share and once again start increasing until the link gets full again. The sharing would be quite unfair and would be dominated by the more aggressive protocol. It is easy to see that this problem gets compounded as more and more flows join in. In such scenarios, the performance of one flow is dependent to a large extent on the aggressive level of the competing flows. Flows that have robust algorithms which have different increase and decrease parameters depending on their estimates of link utilization would be expected to work better in comparison to protocols that have fixed increase and decrease parameters.

Table 3: Nature of increase and decrease parameters

Protocol	Increase Parameter	Decrease Parameter
HS-TCP	Variable	Variable
Scalable TCP	Fixed	Fixed
FAST	Variable	Fixed
H-TCP	Variable	Variable
BIC-TCP	Variable	Fixed
CUBIC	Variable	Fixed

Based on the information in Table 3 we would expect HS-TCP and H-TCP to be more robust. They should have relatively similar behavior with all the other protocols. BIC-TCP and CUBIC should also have good performance with BIC-TCP being more aggressive because of the binary increase function. Scalable TCP should show the poorest performance.

FAST is the only protocol here that is delay-based. Delay-based protocols depend on the queue occupancy on the link routers to gather information about the intensity of congestion on the link. FAST sees increasing queuing delays as signs of congestion and becomes less and less aggressive till the queues start draining. This approach would have been excellent had there been only delay based protocols in competition. However, it is tested against loss based protocols here. Loss based protocols drive the link router queues to overflow to detect link capacity. This approach would cause FAST to keep cutting back as the queue would grow. Once the queue became full, synchronized losses would happen and the queue would drain out. We saw before in chapter 4 that FAST ramped up

very quickly on empty links. We would expect it to behave similarly and take up the major share of bandwidth lost to the packet drops. It would then again cut back as the queue filled up again. We expect its performance to be seriously disadvantaged by the delay based competitors.

5.2 Results

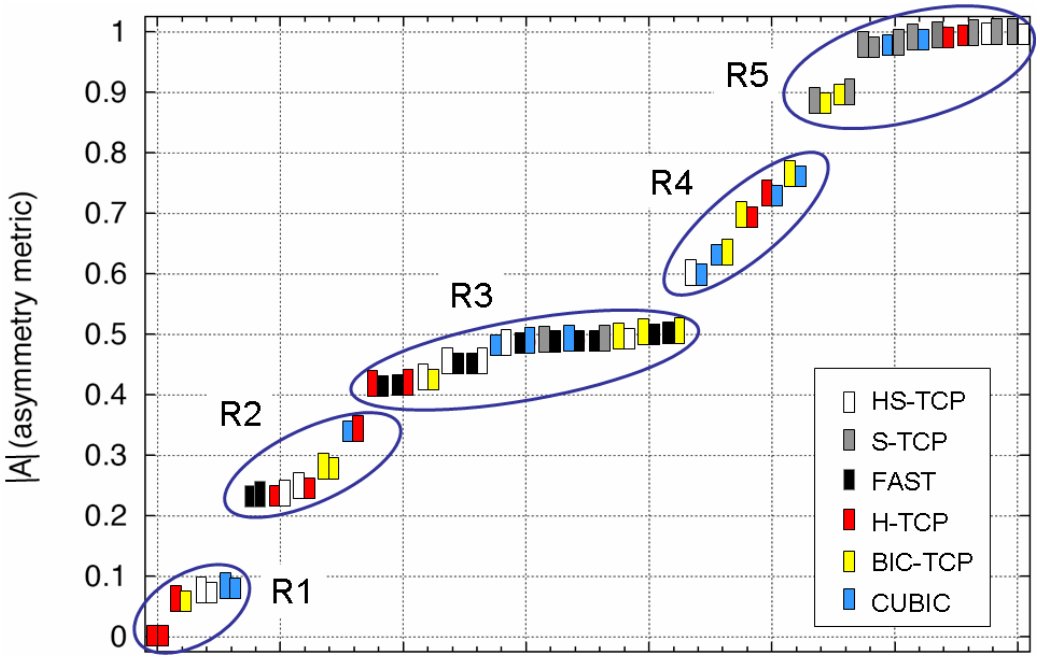

Figure 71: Inter-protocol fairness results

The graph in the Figure 71 shows the consolidated results. Every simulation is represented by a node in the graph. Each node is composed of two columns representing the two protocols that were run in that simulation. The first column represents the flow that was started first and the taller of the two columns represents the more aggressive protocol. The y-coordinate of the node represents the fairness in sharing bandwidth.

Nodes close to the x-axis represent fairer sharing. The nodes have been bunched into 5 regions, R1 to R5, based on their y-coordinate. We explain each of these regions in greater detail below.

5.2.1 Region 1

Most of the intra-protocol runs fell in this region. This is expected as both the competing flows had same increase and decrease intensity. FAST and BIC-TCP were in Region 2 while the Scalable-TCP experiments resulted in a Region 5 spot.

BIC and H-TCP was the only inter-protocol pairing that fell in Region 1. The throughput of the two flows in this simulation is shown in Figure 72. This was when H-TCP was started first. H-TCP was able to give up the bandwidth efficiently to make way for the BIC-TCP flow. However, BIC-TCP was not able to give up bandwidth when it was started first. That result is shown later in section 5.2.4.

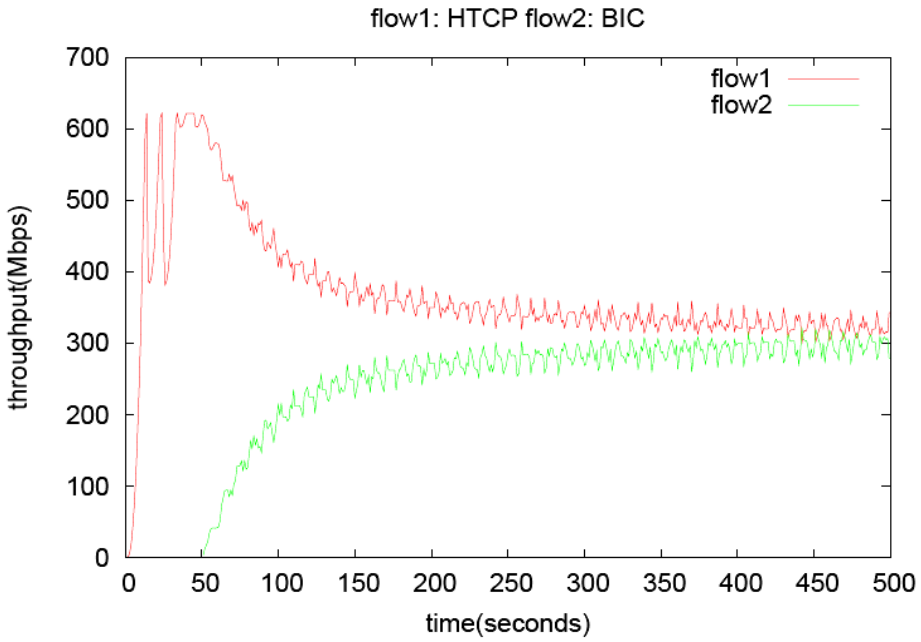

Figure 72: H-TCP-BIC inter-protocol simulation

5.2.2 Region 2

BIC-TCP and FAST intra-protocol results fell in this region. Three H-TCP pairings were also in this region of the result graph. When H-TCP was competed against CUBIC we again saw good behavior on H-TCP's side. Figure 73 shows the throughput of the two flows in this simulation. Even though CUBIC turned out to be the least aggressive of all the flow, H-TCP was still able to back off. Both pairings of H-TCP and HS-TCP, Figure 74 and Figure 75, were also in this region. We can see from the graphs that HS-TCP was more aggressive than H-TCP. The sharing was however stable.

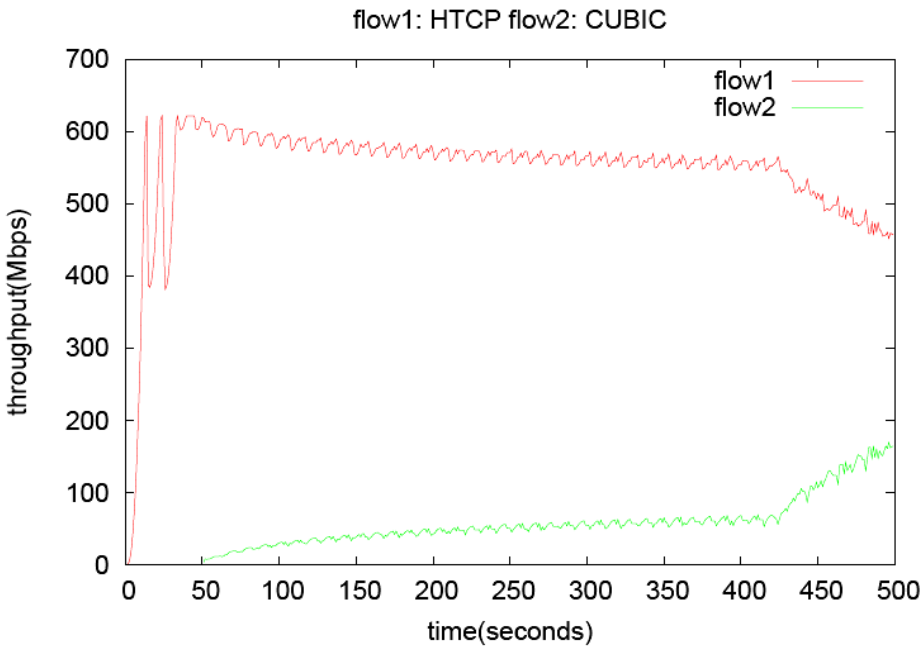

Figure 73: H-TCP-CUBIC inter-protocol simulation

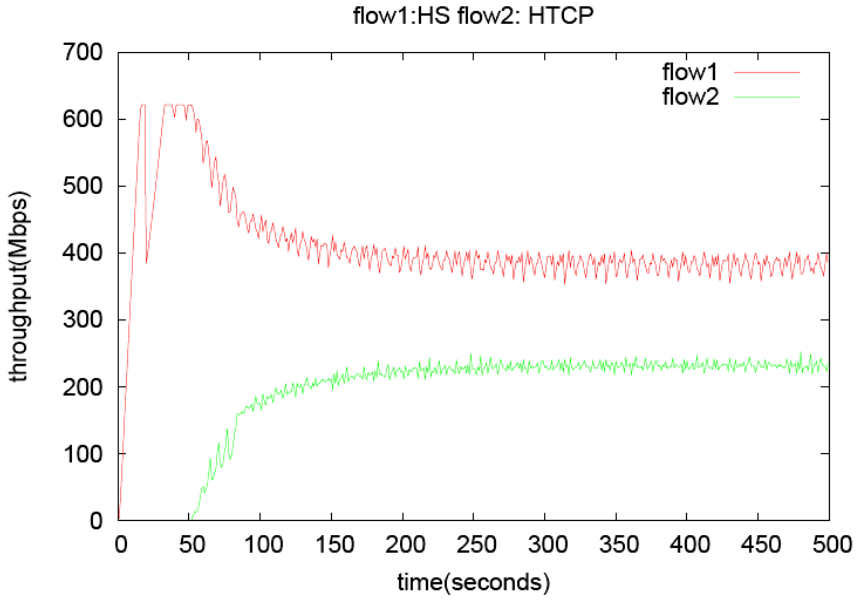

Figure 74: HS-H-TCP inter-protocol simulation

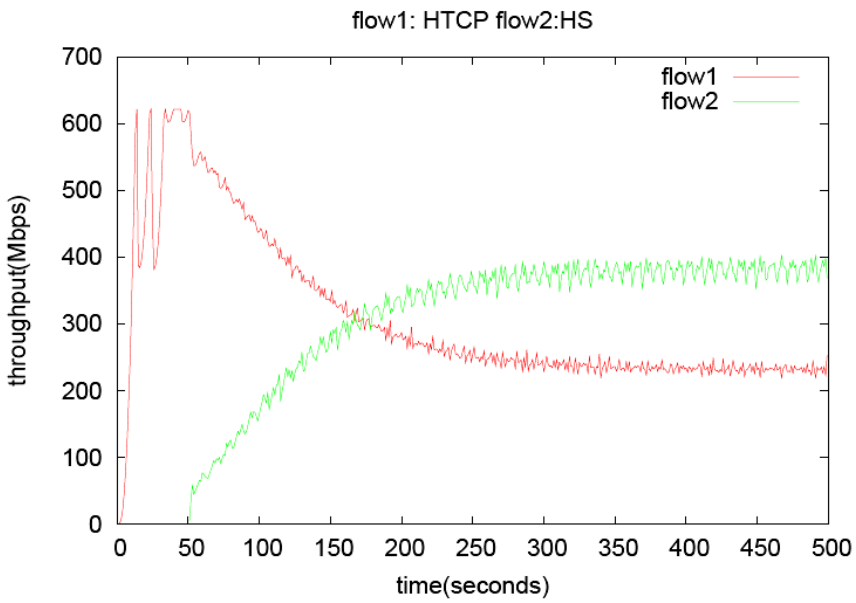

Figure 75: H-TCP-HS-TCP inter-protocol simulation

5.2.3 Region 3

FAST is a delay-based protocol. It rises very rapidly, faster than any of the other protocols, when the router queue is empty and eases of considerably as the router queue size becomes non-zero. All the loss-based protocols drive the router queue to overflow to detect congestion. When the router queue overflows, the loss-based protocols back-off leading to the queue emptying out. FAST increases during this time and garners some share in the bandwidth for itself. We see this behavior of FAST against all other loss-based protocols. Even though there is consistent unfairness in all the results, FAST does not get starved because of its rapid increase when router queue is empty.

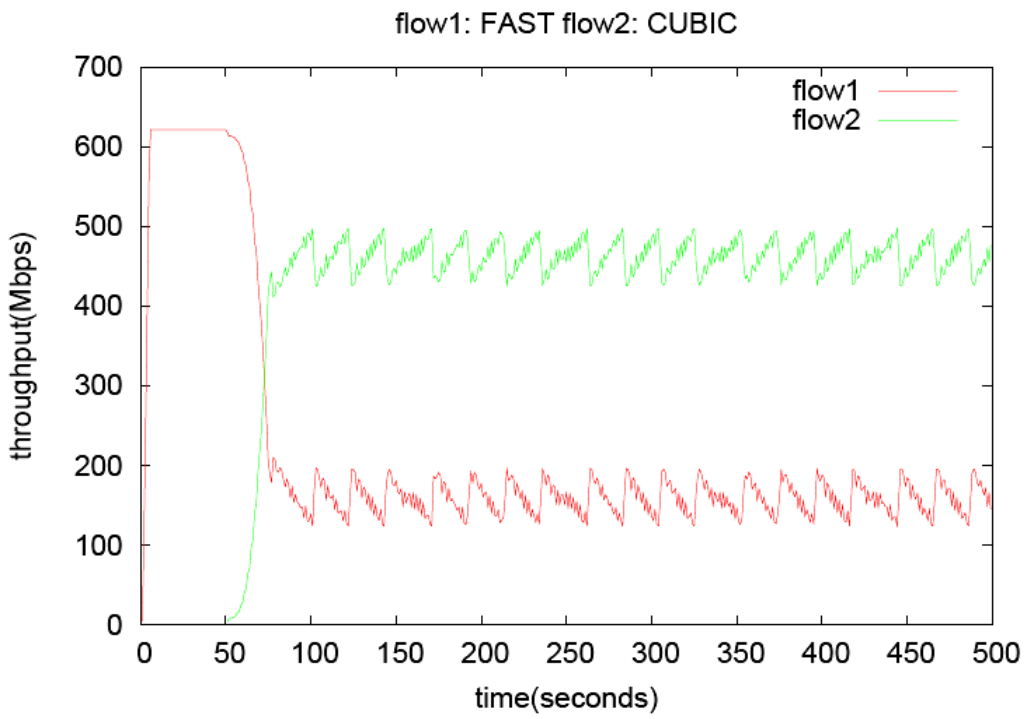

Figure 76: FAST-CUBIC inter-protocol simulation

Scalable-TCP was the most aggressive protocol in our experiments while CUBIC was the mildest. We show throughput results from the runs where CUBIC was started with FAST already occupying the link, Figure 76, and Scalable-TCP was started with FAST already on the link, Figure 77. We see that the results are quite similar.

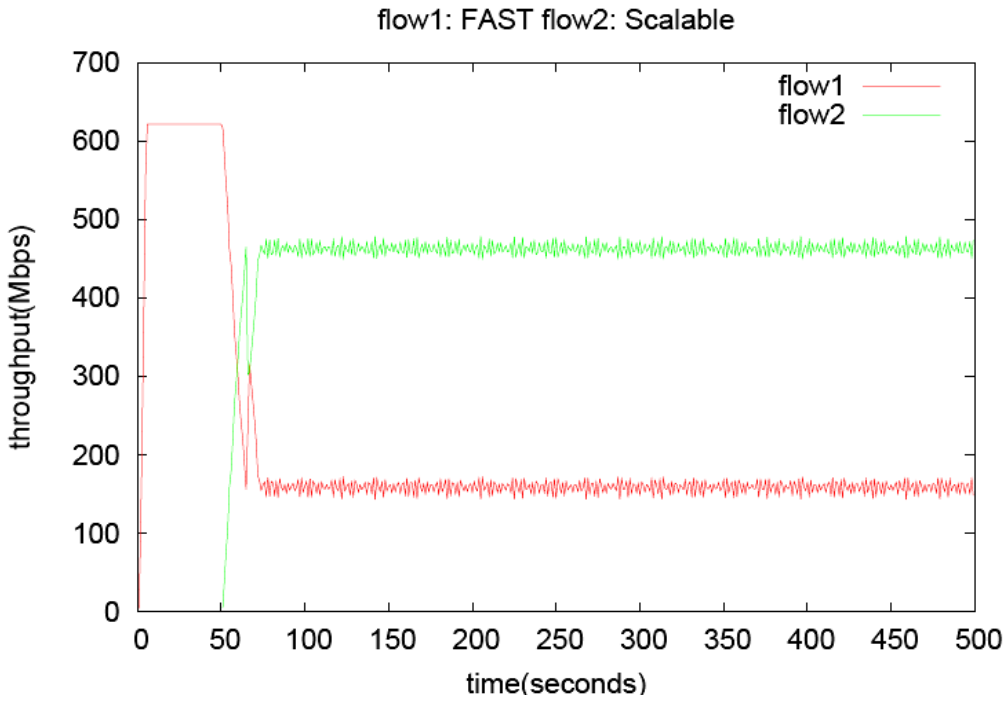

Figure 77: FAST-Scalable inter-protocol simulation

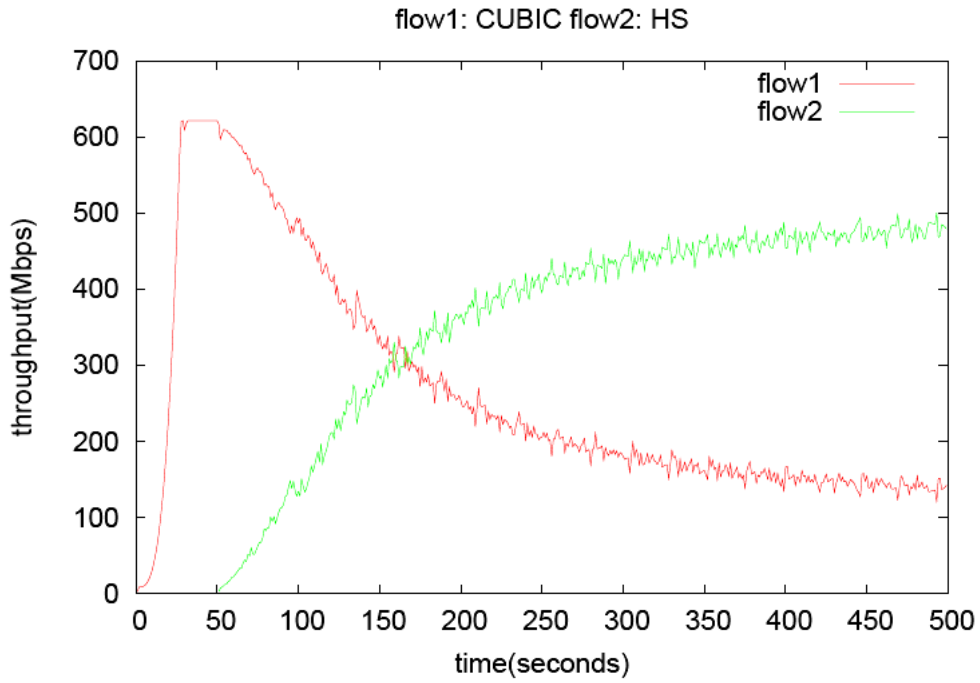

Figure 78: CUBIC-HS-TCP inter-protocol simulation

CUBIC gives way to HS-TCP quickly as shown in the throughput graph in Figure 78. There is a distinct mismatch in the aggressiveness of CUBIC and HS-TCP. However, CUBIC does not get completely starved and is able to retain some bandwidth.

5.2.4 Region 4

We see similar results for CUBIC when it is started with either HS-TCP or H-TCP on the link. Figure 79 shows the throughputs for CUBIC *vs.* HS-TCP and Figure 80 shows the throughputs for CUBIC *vs*. H-TCP. It takes a long time for CUBIC to gain any substantial share in both the cases.

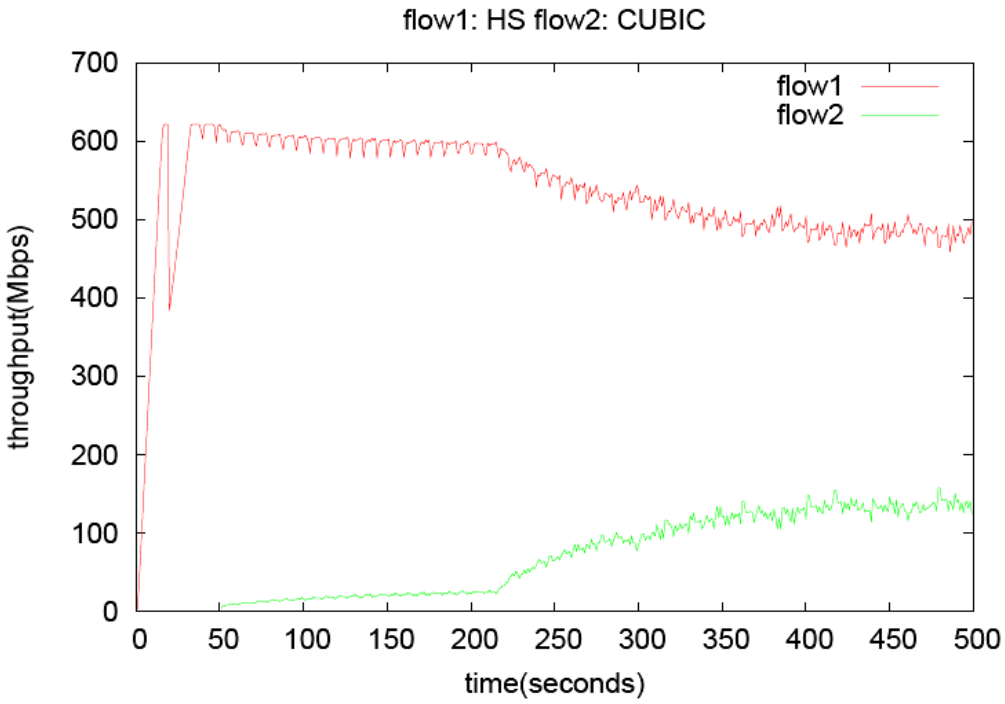

Figure 79: HS-CUBIC inter-protocol simulation

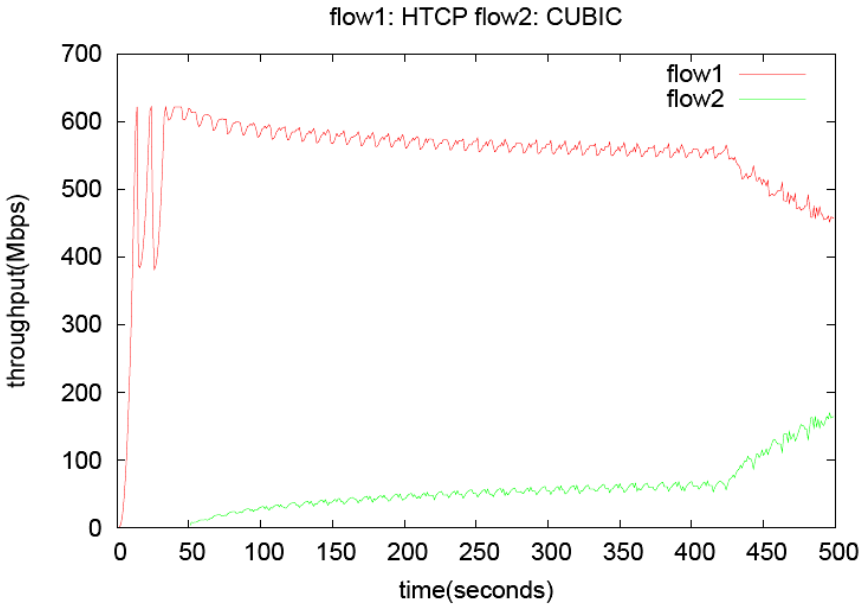

Figure 80: H-TCP-CUBIC inter-protocol simulation

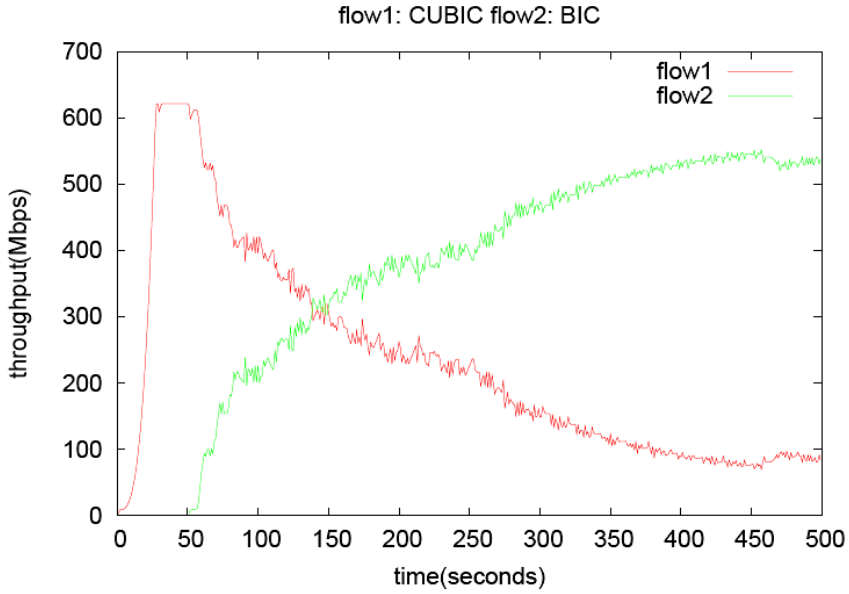

Figure 81: CUBIC-BIC inter-protocol simulation

BIC-TCP and CUBIC simulations gave expected results with BIC-TCP dominating in both the cases. Figure 81 and Figure 82 show the throughput of the

competing BIC-TCP and CUBIC flows. CUBIC was designed to be less aggressive than BIC-TCP and that showed in the results that we got.

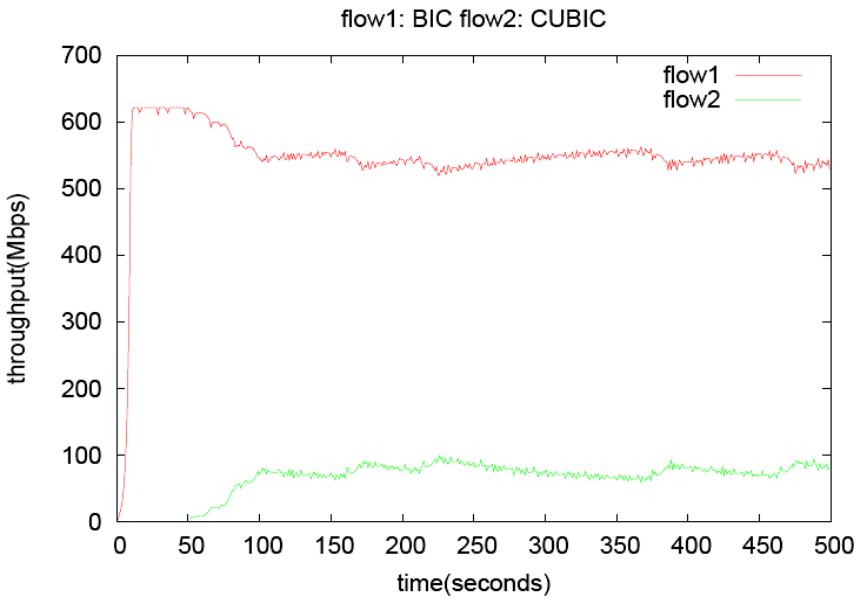

Figure 82: BIC-CUBIC inter-protocol simulation

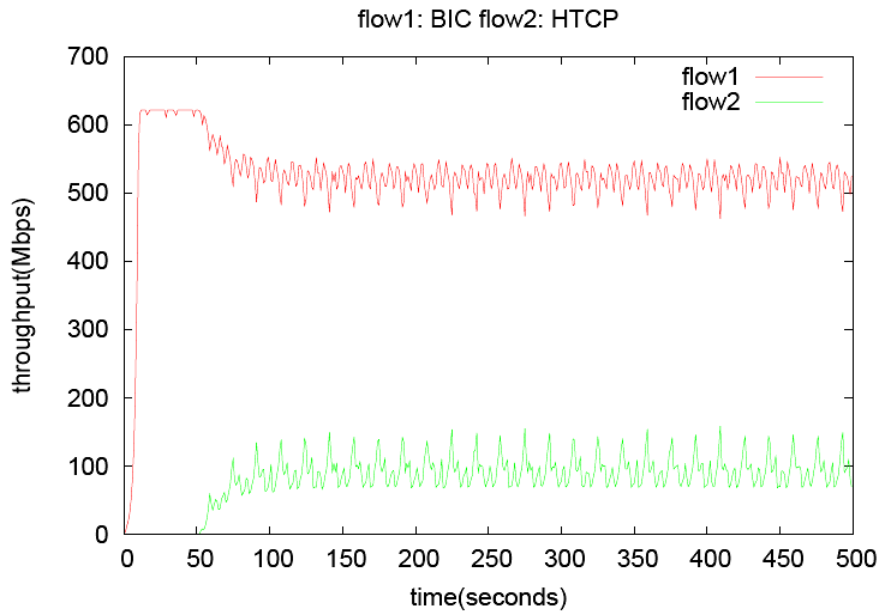

Figure 83: H-TCP-BIC inter-protocol simulation

BIC-TCP did not back-off sufficiently and H-TCP was not able to increase aggressively enough when H-TCP was started on a link with BIC-TCP already occupying the link (Figure 83). The sharing was unfair but stable.

5.2.5 Region 5

All Scalable-TCP pairings fell in this region except the pairing with FAST. Scalable-TCP was unfair in sharing bandwidth with itself too. Figure 84 shows results from HS-TCP *vs.* Scalable-TCP run where Scalable-TCP was started second. We can see that the aggressiveness of Scalable-TCP completely overpowers and starves the HS-TCP flow.

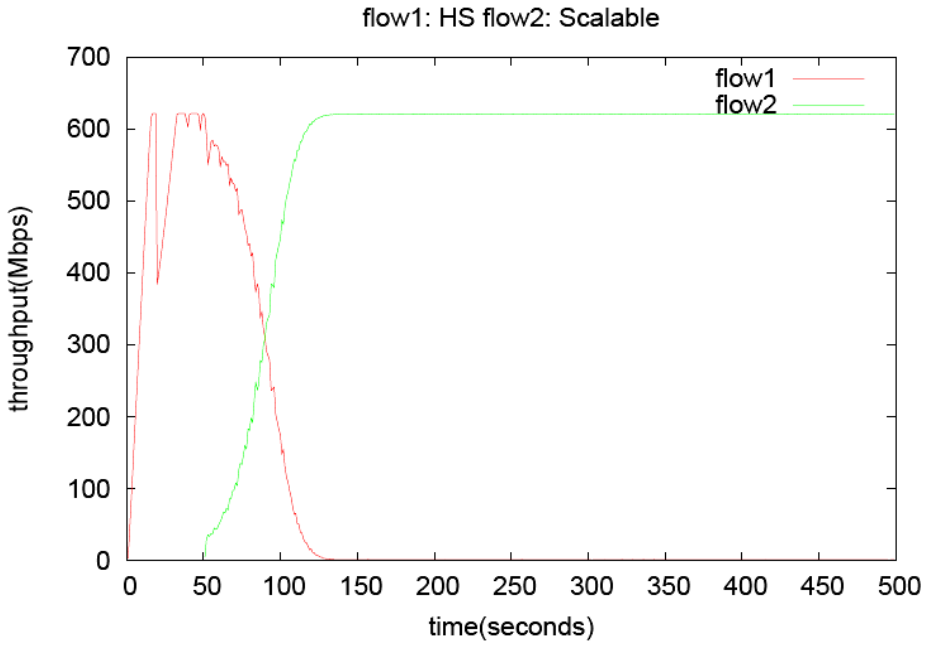

Figure 84: HS-Scalable inter-protocol simulation

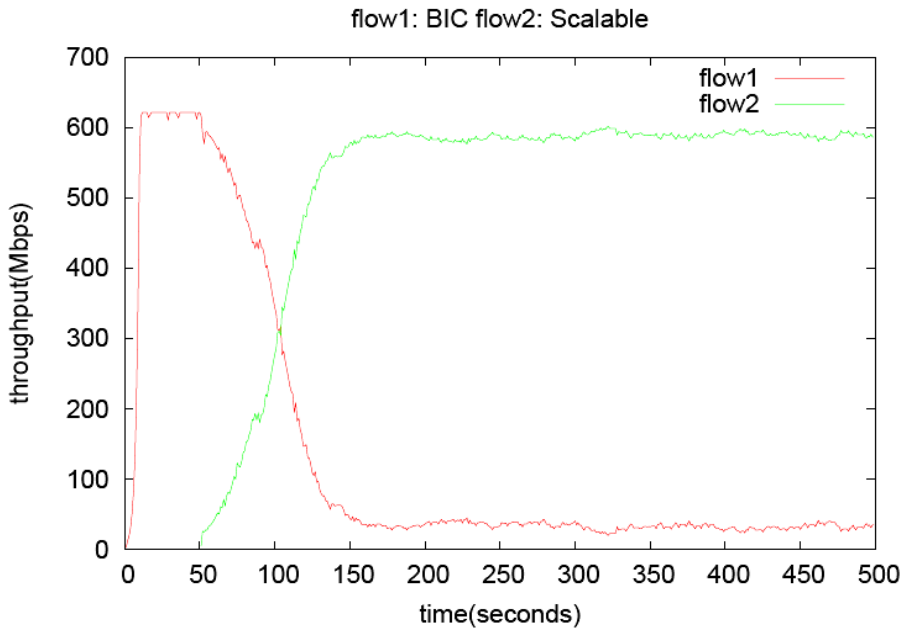

Figure 85: BIC-Scalable inter-protocol simulation

We see a similar result for the BIC-TCP *vs.* Scalable-TCP run, Figure 85. However, BIC-TCP does not get starved like HS-TCP in the previous run. This is because HS-TCP becomes less and less aggressive at high loss rates while BIC-TCP maintains its level of aggressiveness.

CHAPTER 6

CONCLUSIONS

We first conducted an in-depth study of the six high-speed protocols: HS-TCP, Scalable-TCP, BIC-TCP, CUBIC, FAST and H-TCP. We evaluated their performance by simulating artificial packet drops, using different RTTs and using different router buffer queue sizes. The findings from the intra-protocol simulations are presented in section 6.1. Flows belonging to different high-speed protocols were then competed against each other to see the degree of adaptability of the different protocols. The findings from these simulations are presented in section 6.2. Based on our study we propose a set of guidelines for future high-speed transport control proposals in section 6.3. Future work is given in section 6.4.

6.1 Findings from Intra-Protocol Simulations

These results have been summarized previously in section 4.7. We found that all the protocols were able to raise their congestion windows to fully utilize the available bandwidth. FAST was the fastest in its initial increase even with the artificial packet drop. This was because it has an extremely aggressive increase rate when the router queuing delay, *qdelay*, is zero. Even with the artificial packet drop, the *qdelay* was still unchanged at 0 resulting in a fast recovery a good performance. The protocols that were most affected by the artificial packet drop were HS-TCP, BIC-TCP and CUBIC-TCP.

HS-TCP is less aggressive at high-loss environments and a packet loss early on led it to interpret that the network had a high-loss rate resulting in poor performance. BIC-TCP and CUBIC make estimates of what the maximum link capacity is based on where they see the first packet drop. They then increase rapidly to around that point and probe slowly after that. Link bandwidth however fluctuates a lot and the effectiveness of such estimates has limited usefulness. The poor performance of BIC-TCP and CUBIC for this set of tests was because of this strategy.

BIC-TCP and CUBIC had the best performance when the network bandwidth was limited to 40 packets and they both had more than 90% utilization. They have smaller and smaller increments as they approach link capacity and only overflow the router buffer queue by a small amount when they surpass link capacity. Thus they do not require a large buffer for good performance. H-TCP and FAST were able to surpass the 90% utilization with 500 queue size. FAST tries to keep α packets, set to 312 in our experiments, in the router queue at all times. With queue sizes below that value, packets were continuously dropped and hence there was bad performance at queue sizes smaller than α.

At small RTT, 20ms, Scalable-TCP was the fastest in getting to 90% network utilization. FAST performed very poorly and was not able to get to 90% utilization. This was because of FAST's delay based approach. An increase in the RTT difference to 220ms had the maximum influence on Scalable-TCP. H-TCP has explicit RTT-fairness

mechanisms and it showed the best performance with the least difference in performance with changes in RTT.

For the intra-protocol simulations where two flows belonging to the same protocol were simulated with different RTTs, the protocols with smaller difference in performance with different RTT performed much better. Fairness mechanisms where the bigger flow makes space to accommodate the smaller flow were also an important factor in the performance. Scalable-TCP and HS-TCP were the worst performers where the longer flow was starved when the RTT of the smaller flow was reduced to 82ms. BIC-TCP performed slightly better but was still very unfair at large differences in RTT. FAST, CUBIC and H-TCP performed very well and the RTT difference did not have much affect on the bandwidth sharing.

6.2 Findings from Inter-Protocol Simulations

These results have been previously summarized in section 5.2. We looked at how fairly the high-speed flows shared the network bandwidth with each other. We ran our tests with three different router queue lengths. We present the results with the queue length set to 20% BDP as it gave a high utilization and was small enough to be in the range of most commercial routers. We stated our expectations from this study beforehand based on the results and performance observed in the intra-protocol experiments. The results that we got matched our expectations. We found the protocols that had dynamic approaches to be more efficient in terms of sharing bandwidth.

Based on the regions in which the protocols fell in the consolidated inter-protocol fairness result graph, Figure 71, we ranked the protocols on their fairness. A spot in Region 1 gave a protocol 1 point; a spot in Region 2 gave it 2 points and so on. Protocols with the less number of points at the end were awarded a better rank. The ranking gives an approximate indication of the fairness capabilities and adaptability of the protocols when they were competing against other protocols. The protocols are ranked as follows:

1. H-TCP
2. FAST
3. HS-TCP, BIC-TCP
4. CUBIC
5. S-TCP

The rankings match our earlier expectations that were made based on the careful study of the various high-speed protocols. H-TCP shows overall good behavior and Scalable TCP is overly aggressive in all situations. FAST had similar less aggressive behavior against all protocols because of its delay-based design. CUBIC was also very gentle in all the inter-protocol runs and perhaps needs to tune its parameters to be more aggressive. It will be very interesting to see how it performs with a bigger *C* value.

6.3 Guidelines for Future High-Speed Transport Control Protocol Proposals

Based on our study we present the following guidelines for the design of an efficient high-speed transport control protocol.

1. Since congestion severity is variable and not fixed it is important that the mechanisms that deal with it are also variable in nature. Having a solution that caters to median values or the border values will necessarily be inefficient compared to the solutions that offer different optimized solutions for the breadth of the variable's values. Protocol designers should therefore have dynamic approaches in designing high-speed TCPs for efficient and fair utilization of available bandwidth. The increase and decrease parameters should therefore be dynamic and linked to the severity of congestion on the network link.

2. A good way for measuring the severity of congestion is measuring the RTT on the network path. This would be the delay-based approach that has been implemented in FAST. We have seen that loss-based protocols severely hampered the performance of FAST in section 5.2.3. We should therefore additionally include other mechanisms like CUBIC's and H-TCP's approaches which monitor the time since the last congestion epoch to decide on the congestion window value. This would result in an effective hybrid approach that will tackle the problem of congestion control effectively.

3. RTT-fairness mechanisms like those employed by CUBIC and H-TCP should be incorporated as competing flows rarely ever have similar RTTs.

4. FAST had the best results for the initial increase. FAST is very aggressive when there is no congestion on the link, qdelay is 0. We could take this approach into our hybrid approach where a flow would ramp up very quickly till it sees 0 qdelay and then switch to hybrid mode when qdelay would become non-zero.

5. Performance of high-speed protocols that become gentle as they reach link capacity is the best in small bottleneck buffer scenarios (section 4.7.1). The shape of BIC-TCP's and CUBIC's *cwnd* graph should be emulated to have good performance in differently sized router buffer scenarios.

6. Link utilization is highly variable in nature and the high-speed protocols that rely on estimations of link capacity or link propagation delay are affected when the estimations are incorrect. This was evident in the intra-protocol fairness results from

FAST in section 4.5.5 and BIC-TCP's and CUBIC's poor performance when there was a packet drop during the initial ascent of the congestion window (section 4.3.3.2 and section 4.4.3.2). Even in cases where the estimations are correct, there is no guarantee that those values will be the same in the future. Thus, protocols should try to update their estimates often enough to dynamically adapt to changing network conditions.

6.4 Future Work

As part of our future work, we intend to explore the effect of background traffic on fairness. The topologies we used are very basic and the effect of complex network topologies could also be interesting. Active Queue Management (AQM) are router queue management schemes where packets are dropped from the queue before the queue overflows. Many of the AQM schemes incorporate fairness enforcement schemes where they drop packets belonging to the larger flows. Behavior of the high-speed TCPs in such an environment should be fairer. It would be interesting to see how much effect this has on their performance. Other things that we want to look at are more number of flows on the link at the same time and also inter-protocol fairness with different RTT flows.

APPENDICES

Appendix A

Tcl script for running standard TCP in low bandwidth environment in ns-2

```
#############################################################
# High-Speed TCP Studies - hstcp.tcl                        #
# Michele Weigle and Pankaj Sharma                          #
#  - based on scripts from L. Xu, K. Harfoush, I. Rhee #
#  - based on scripts from S. Floyd and E. Souza       #
#############################################################

set hstcp_type [lindex $argv 0];   # SACK, HS, Scalable, FAST, BIC, CUBIC
set endtime 50
set alpha 156
proc print-cwnd {flow interval fp} {
  global ns
  set now [$ns now]
  puts $fp "[format %.2f $now] [$flow set cwnd_]"
  set nexttime [expr $now+$interval]
  $ns at $nexttime "print-cwnd $flow $interval $fp"
}

proc print-cwnd-bic {flow interval fp} {
  global ns
  set now [$ns now]
  puts $fp "[format %.2f $now] [$flow set cwnd_] [$flow set bic_delay_min_] [$flow set
bic_delay_avg_] [$flow set bic_delay_max_] [$flow set
bic_low_utilization_indication_]"
  set nexttime [expr $now+$interval]
  $ns at $nexttime "print-cwnd-bic $flow $interval $fp"
}

proc print-th-one {fid fmon interval} {
  global frep tput_hs lastKBytes ns
  set now [$ns now]
  set fcl [$fmon classifier]; # flow classifier
  set flow [$fcl lookup auto 0 0 $fid]
  if {$flow != "" } {
    set bytes [$flow set bdepartures_]
    set bytesDbl [ns-int64todbl $bytes]
```

```
    set Kbytes [expr $bytesDbl / 1000 ]
    set thruLastPeriod [ expr ( $Kbytes - $lastKBytes)*8 /$interval/1000]
    set lastKBytes $Kbytes
    puts $tput_hs "[format "%.2f" $now] [format %.2f $thruLastPeriod]"
   }

   $ns at [expr $now+$interval] "print-th-one $fid $fmon $interval"
}

proc finish {} {
  global ns tput_hs cfp_hs trace_file
  flush $cfp_hs
  close $cfp_hs
  flush $tput_hs
  close $tput_hs
  flush $trace_file
  close $trace_file
  exit 0
}

set bandwidth 5;   # 622 Mbps
set delay 50;       # total of 100 ms RTT
set buffer 1;       # qlen is 1 x BDP

# parameters for CUBIC
set cubic_tcp_mode 1

set low_window 0
set high_window 83000
set high_p 0.0000001
set high_decrease 0.1
set hstcp_fix 1

remove-all-packet-headers       ; # removes all except common
add-packet-header Flags IP TCP  ; # hdrs reqd for TCP
set ns [new Simulator]
$ns use-scheduler Heap
global defaultRNG
$defaultRNG seed 9999;
set n1 [$ns node]
set n2 [$ns node]

Agent/TCP set timestamps_ 1
Agent/TCP set window_ 67000
Agent/TCP set packetSize_ 1000
Agent/TCP set overhead_ 0.000008
```

```
Agent/TCP set max_ssthresh_ 100
Agent/TCP set maxburst_ 2

# BIC
Agent/TCP set bic_beta_ 0.8
Agent/TCP set bic_B_ 4
Agent/TCP set bic_max_increment_ 32
Agent/TCP set bic_min_increment_ 0.01
Agent/TCP set bic_fast_convergence_ 1
Agent/TCP set bic_low_utilization_threshold_ 0
Agent/TCP set bic_low_utilization_checking_period_ 2
Agent/TCP set bic_delay_min_ 0
Agent/TCP set bic_delay_avg_ 0
Agent/TCP set bic_delay_max_ 0
Agent/TCP set bic_low_utilization_indication_ 0

# CUBIC
Agent/TCP set cubic_beta_ 0.8
Agent/TCP set cubic_max_increment_ 16
Agent/TCP set cubic_fast_convergence_ 1
Agent/TCP set cubic_scale_ 0.4
Agent/TCP set cubic_tcp_friendliness_ $cubic_tcp_mode
Agent/TCP set cubic_low_utilization_threshold_ 0
Agent/TCP set cubic_low_utilization_checking_period_ 2
Agent/TCP set cubic_delay_min_ 0
Agent/TCP set cubic_delay_avg_ 0
Agent/TCP set cubic_delay_max_ 0
Agent/TCP set cubic_low_utilization_indication_ 0

set bf_size [expr $bandwidth*$delay*2*1000*$buffer/[Agent/TCP set packetSize_]/8]

# setup link
$ns duplex-link $n1 $n2 [expr $bandwidth]Mb [expr $delay]ms DropTail
$ns queue-limit $n1 $n2 $bf_size
$ns queue-limit $n2 $n1 $bf_size

# setup packet tracing
set trace_file [open "$hstcp_type/pckts.trq" w]
$ns trace-queue $n1 $n2 $trace_file
$ns trace-queue $n2 $n1 $trace_file

set rng_ [new RNG]

set starttime 0.1

set hsnode_s [$ns node]
```

```
set hsnode_r [$ns node]

set del 1.0

$ns duplex-link $hsnode_s $n1 1000Mb ${del}ms DropTail
$ns duplex-link $hsnode_r $n2 1000Mb 1ms DropTail
$ns queue-limit $hsnode_s $n1 67000
$ns queue-limit $n1 $hsnode_s 67000
$ns queue-limit $n2 $hsnode_r 67000
$ns queue-limit $hsnode_r $n2 67000

 if {$hstcp_type != "FAST"} {
          set hstcp [$ns create-connection TCP/SackTS $hsnode_s TCPSink/Sack1
$hsnode_r 0]

       if {$hstcp_type == "HS"} {
             set hstcpflows_type 8
             set low_window 38
          } elseif {$hstcp_type == "Scalable"} {
             set hstcpflows_type 9
             set low_window 16
             $hstcp set scalable_lwnd_ $low_window
          } elseif {$hstcp_type == "BIC"} {
             set hstcpflows_type 12
             set low_window 14
          } elseif {$hstcp_type == "CUBIC"} {
             set hstcpflows_type 13
          }
       if {$hstcp_type != "SACK"} {
            $hstcp set windowOption_ $hstcpflows_type
       }
          $hstcp set low_window_ $low_window
          $hstcp set high_window_ $high_window
          $hstcp set high_p_ $high_p
          $hstcp set high_decrease_ $high_decrease
          $hstcp set hstcp_fix_ $hstcp_fix
   } else {
          set hstcp [$ns create-connection TCP/Fast $hsnode_s TCPSink $hsnode_r 0]
          hstcp set alpha_ $alpha
          $hstcp set beta_ $alpha
   }

  set hsftp [$hstcp attach-app FTP]
  $ns at $starttime "$hsftp start"
  $ns at [expr $endtime+5] "$hsftp stop"
  $ns at $endtime "finish"
```

```
  set cfp_hs [open $hstcp_type/cwnd.out w]
  set tput_hs [open $hstcp_type/tput.out w]

if {$hstcp_type != "BIC"} {
    print-cwnd $hstcp 0.1 $cfp_hs
} elseif { $hstcpflows_type == 12 } {
    print-cwnd-bic $hstcp 0.1 $cfp_hs
}

set fmon [$ns makeflowmon Fid]
$ns attach-fmon [$ns link $n1 $n2] $fmon
set lastKBytes 0
print-th-one 0 $fmon 1

$ns run
```

Appendix B

Tcl script for running standard TCP in high bandwidth environment in ns-2

```
#############################################################
# High-Speed TCP Studies - hstcp.tcl                #
# Michele Weigle and Pankaj Sharma                  #
#  - based on scripts from L. Xu, K. Harfoush, I. Rhee #
#  - based on scripts from S. Floyd and E. Souza       #
#############################################################

set hstcp_type [lindex $argv 0];   # SACK, HS, Scalable, FAST, BIC, CUBIC
set endtime 150
set alpha 156
proc print-cwnd {flow interval fp} {
  global ns
  set now [$ns now]
  puts $fp "[format %.2f $now] [$flow set cwnd_]"
  set nexttime [expr $now+$interval]
  $ns at $nexttime "print-cwnd $flow $interval $fp"
}

proc print-cwnd-bic {flow interval fp} {
  global ns
  set now [$ns now]
  puts $fp "[format %.2f $now] [$flow set cwnd_] [$flow set bic_delay_min_] [$flow set
bic_delay_avg_] [$flow set bic_delay_max_] [$flow set
bic_low_utilization_indication_]"

  set nexttime [expr $now+$interval]
  $ns at $nexttime "print-cwnd-bic $flow $interval $fp"
}

proc print-th-one {fid fmon interval} {
  global frep tput_hs lastKBytes ns

  set now [$ns now]

  set fcl [$fmon classifier]; # flow classifier
  set flow [$fcl lookup auto 0 0 $fid]
  if {$flow != "" } {
    set bytes [$flow set bdepartures_]
```

```
    set bytesDbl [ns-int64todbl $bytes]
    set Kbytes [expr $bytesDbl / 1000 ]
    set thruLastPeriod [ expr ( $Kbytes - $lastKBytes)*8 /$interval/1000]
    set lastKBytes $Kbytes
    puts $tput_hs "[format "%.2f" $now] [format %.2f $thruLastPeriod]"
   }
   $ns at [expr $now+$interval] "print-th-one $fid $fmon $interval"
}

proc finish {} {
  global ns tput_hs cfp_hs trace_file
  flush $cfp_hs
  close $cfp_hs
  flush $tput_hs
  close $tput_hs
  flush $trace_file
  close $trace_file
  exit 0
}

set bandwidth 622;   # 622 Mbps
set delay 50;        # total of 100 ms RTT
set buffer 1;        # qlen is 1 x BDP

# parameters for CUBIC
set cubic_tcp_mode 1

set low_window 0
set high_window 83000
set high_p 0.0000001
set high_decrease 0.1
set hstcp_fix 1

remove-all-packet-headers       ; # removes all except common
add-packet-header Flags IP TCP  ; # hdrs reqd for TCP

set ns [new Simulator]
$ns use-scheduler Heap

global defaultRNG
$defaultRNG seed 9999;

set n1 [$ns node]
set n2 [$ns node]

Agent/TCP set timestamps_ 1
```

```
Agent/TCP set window_ 67000
Agent/TCP set packetSize_ 1000
Agent/TCP set overhead_ 0.000008
Agent/TCP set max_ssthresh_ 100
Agent/TCP set maxburst_ 2

# BIC
Agent/TCP set bic_beta_ 0.8
Agent/TCP set bic_B_ 4
Agent/TCP set bic_max_increment_ 32
Agent/TCP set bic_min_increment_ 0.01
Agent/TCP set bic_fast_convergence_ 1
Agent/TCP set bic_low_utilization_threshold_ 0
Agent/TCP set bic_low_utilization_checking_period_ 2
Agent/TCP set bic_delay_min_ 0
Agent/TCP set bic_delay_avg_ 0
Agent/TCP set bic_delay_max_ 0
Agent/TCP set bic_low_utilization_indication_ 0

# CUBIC
Agent/TCP set cubic_beta_ 0.8
Agent/TCP set cubic_max_increment_ 16
Agent/TCP set cubic_fast_convergence_ 1
Agent/TCP set cubic_scale_ 0.4
Agent/TCP set cubic_tcp_friendliness_ $cubic_tcp_mode
Agent/TCP set cubic_low_utilization_threshold_ 0
Agent/TCP set cubic_low_utilization_checking_period_ 2
Agent/TCP set cubic_delay_min_ 0
Agent/TCP set cubic_delay_avg_ 0
Agent/TCP set cubic_delay_max_ 0
Agent/TCP set cubic_low_utilization_indication_ 0

set bf_size [expr $bandwidth*$delay*2*1000*$buffer/[Agent/TCP set packetSize_]/8]

# setup link
$ns duplex-link $n1 $n2 [expr $bandwidth]Mb [expr $delay]ms DropTail
$ns queue-limit $n1 $n2 $bf_size
$ns queue-limit $n2 $n1 $bf_size

# setup packet tracing
set trace_file [open "$hstcp_type/pckts.trq" w]
$ns trace-queue $n1 $n2 $trace_file
$ns trace-queue $n2 $n1 $trace_file

set rng_ [new RNG]
```

```
set starttime 0.1

set hsnode_s [$ns node]
set hsnode_r [$ns node]

set del 1.0

$ns duplex-link $hsnode_s $n1 1000Mb ${del}ms DropTail
$ns duplex-link $hsnode_r $n2 1000Mb 1ms DropTail

  $ns queue-limit $hsnode_s $n1 67000
  $ns queue-limit $n1 $hsnode_s 67000

  $ns queue-limit $n2 $hsnode_r 67000
  $ns queue-limit $hsnode_r $n2 67000

 if {$hstcp_type != "FAST"} {
          set hstcp [$ns create-connection TCP/SackTS $hsnode_s TCPSink/Sack1
$hsnode_r 0]

       if {$hstcp_type == "HS"} {
             set hstcpflows_type 8
             set low_window 38
          } elseif {$hstcp_type == "Scalable"} {
             set hstcpflows_type 9
             set low_window 16
             $hstcp set scalable_lwnd_ $low_window
          } elseif {$hstcp_type == "BIC"} {
             set hstcpflows_type 12
             set low_window 14
          } elseif {$hstcp_type == "CUBIC"} {
             set hstcpflows_type 13
          }
       if {$hstcp_type != "SACK"} {
            $hstcp set windowOption_ $hstcpflows_type
       }
          $hstcp set low_window_ $low_window
          $hstcp set high_window_ $high_window
          $hstcp set high_p_ $high_p
          $hstcp set high_decrease_ $high_decrease
          $hstcp set hstcp_fix_ $hstcp_fix
    } else {
          set hstcp [$ns create-connection TCP/Fast $hsnode_s TCPSink $hsnode_r 0]
       $hstcp set alpha_ $alpha
       $hstcp set beta_ $alpha
    }
```

```
  set hsftp [$hstcp attach-app FTP]

  $ns at $starttime "$hsftp start"
  $ns at [expr $endtime+5] "$hsftp stop"
  $ns at $endtime "finish"

  set cfp_hs [open $hstcp_type/cwnd.out w]
  set tput_hs [open $hstcp_type/tput.out w]

if {$hstcp_type != "BIC"} {
    print-cwnd $hstcp 0.1 $cfp_hs
} elseif { $hstcpflows_type == 12 } {
    print-cwnd-bic $hstcp 0.1 $cfp_hs
}

set fmon [$ns makeflowmon Fid]
$ns attach-fmon [$ns link $n1 $n2] $fmon
set lastKBytes 0
print-th-one 0 $fmon 1

$ns run
```

Appendix C

Tcl script for running "Single Flow, No Enforced Packet Drops" experiments in ns-2

```
#############################################################
# High-Speed TCP Studies - hstcp.tcl                  #
# Pankaj Sharma                                             #
#  - based on scripts from Michele Weigle             #
#  - based on scripts from L. Xu, K. Harfoush, I. Rhee      #
#  - based on scripts from S. Floyd and E. Souza            #
#############################################################

#ARGS: endtime flow_type fastAlpha run DATADIR

set begintime 0

#simulation endtime
set endtime [lindex $argv 0]
set flow_type [lindex $argv 1]
set alpha [lindex $argv 2]
set run [lindex $argv 3]
set DATADIR [lindex $argv 4]


set hstcpflows_num 1
set hstcpflows_type 0
set flow_id 0
set bandwidth 622;      #622 Mbps
set delay 48;           #2x48+2x2= 100ms (RTT)

remove-all-packet-headers       ; # removes all except common
add-packet-header Flags IP TCP  ; # hdrs reqd for TCP

source utils.tcl

set frep [open $DATADIR/report.txt w]
set urep [open $DATADIR/util.out w]

set ns [new Simulator]
global defaultRNG
$defaultRNG seed 9999

for {set i 1} {$i < $run} {incr i} {
```

```
    $defaultRNG next-substream
}


Agent/TCP set timestamps_ 1
Agent/TCP set window_ 67000
Agent/TCP set packetSize_ 1000
Agent/TCP set overhead_ 0.000008
Agent/TCP set max_ssthresh_ 100
Agent/TCP set maxburst_ 2

set cubic_tcp_mode 1
set low_window 0
set high_window 83000
set high_p 0.0000001
set high_decrease 0.1
set hstcp_fix 1


#Parameters for BI-TCP
Agent/TCP set bic_beta_ 0.8
Agent/TCP set bic_B_ 4
Agent/TCP set bic_max_increment_ 32
Agent/TCP set bic_min_increment_ 0.01
Agent/TCP set bic_fast_convergence_ 1
Agent/TCP set bic_low_utilization_threshold_ 0
Agent/TCP set bic_low_utilization_checking_period_ 2
Agent/TCP set bic_delay_min_ 0
Agent/TCP set bic_delay_avg_ 0
Agent/TCP set bic_delay_max_ 0
Agent/TCP set bic_low_utilization_indication_ 0


# CUBIC
Agent/TCP set cubic_beta_ 0.8
Agent/TCP set cubic_max_increment_ 16
Agent/TCP set cubic_fast_convergence_ 1
Agent/TCP set cubic_scale_ 0.4
Agent/TCP set cubic_tcp_friendliness_ $cubic_tcp_mode
Agent/TCP set cubic_low_utilization_threshold_ 0
Agent/TCP set cubic_low_utilization_checking_period_ 2
Agent/TCP set cubic_delay_min_ 0
Agent/TCP set cubic_delay_avg_ 0
Agent/TCP set cubic_delay_max_ 0
Agent/TCP set cubic_low_utilization_indication_ 0;
```

```
###############N/w Setup################

set n1 [$ns node]
set n2 [$ns node]

set bf_size 1555

$ns duplex-link $n1 $n2 [expr $bandwidth]Mb [expr $delay]ms DropTail

$ns queue-limit $n1 $n2 $bf_size
$ns queue-limit $n2 $n1 $bf_size

set qmon [$ns monitor-queue $n1 $n2 ""]
set qfp [open $DATADIR/queue.out w]
print-queue 0.01 $qfp

set fmon [$ns makeflowmon Fid]
$ns attach-fmon [$ns link $n1 $n2] $fmon

# back path fmon
set fmon2 [$ns makeflowmon Fid]
$ns attach-fmon [$ns link $n2 $n1] $fmon2

set src_hsnode [$ns node]
set recv_hsnode [$ns node]
$ns duplex-link $src_hsnode $n1 1000Mb 1ms DropTail
$ns duplex-link $recv_hsnode $n2 1000Mb 1ms DropTail
$ns queue-limit $src_hsnode $n1 100000
$ns queue-limit $n1 $src_hsnode 100000
$ns queue-limit $n2 $recv_hsnode 100000
$ns queue-limit $recv_hsnode $n2 100000

if {$flow_type != "FAST"} {
      set hstcp [$ns create-connection TCP/Sack1 $src_hsnode TCPSink/Sack1
$recv_hsnode $flow_id]

      if {$flow_type == "HS"} {
         set hstcpflows_type 8
         set low_window 38
      } elseif {$flow_type == "Scalable"} {
         set hstcpflows_type 9
         set low_window 16
         [set hstcp] set scalable_lwnd_ $low_window
      } elseif {$flow_type == "BIC"} {
         set hstcpflows_type 12
         set low_window 14
```

```
        } elseif {$flow_type == "CUBIC"} {
            set hstcpflows_type 13
        } elseif {$flow_type == "HTCP"} {
            set hstcpflows_type -10
        }

            if {$flow_type != "HTCP"} {
            [set hstcp] set max_ssthresh_ 100
            } else  {
            [set hstcp] set max_ssthresh_ 1
            }

        [set hstcp] set windowOption_ $hstcpflows_type
        [set hstcp] set low_window_ $low_window
        [set hstcp] set high_window_ $high_window
        [set hstcp] set high_p_ $high_p
        [set hstcp] set high_decrease_ $high_decrease
        [set hstcp] set hstcp_fix_ $hstcp_fix
} else {
        set hstcp [$ns create-connection TCP/Fast $src_hsnode TCPSink/Sack1
$recv_hsnode $flow_id]
        [set hstcp] set alpha_ $alpha
        [set hstcp] set beta_ $alpha
}

set hsftp [[set hstcp] attach-app FTP]
set lastBytes 0

$ns at $begintime "[set hsftp] start"
$ns at [expr $endtime+5] "[set hsftp] stop"

# print this connection start time
puts $frep "hstcp starttime: $begintime"

set cfp_hs [open $DATADIR/cwnd_hstcp.out w]
set tput_hs [open $DATADIR/tput_hstcp.out w]
if { $hstcpflows_type == 12 } {
    print-cwnd-bic [set hstcp] 0.01 $cfp_hs
} else {
    print-cwnd [set hstcp] 0.01 $cfp_hs
}

  set lastKBytes 0
  print-th-one $flow_id $fmon 1

set halftime [expr $endtime / 2]
```

```
puts stderr "halftime: $halftime   endtime: $endtime"

$ns at $halftime "print-stat-one-pre $flow_id $fmon"
$ns at $endtime "print-stat-one $flow_id $fmon"

$ns at $halftime "print-stat-all-pre"

$ns at $endtime "printlegend"
$ns at $endtime "finish"

$ns at 0 "timeReport 1"
$ns at 0.1 "print-util 0.1 0"

$ns run
```

Appendix D

Tcl script for running "Single Flow, Single Drop" experiments in ns-2

```
#############################################################
# High-Speed TCP Studies - hstcp.tcl                  #
# Pankaj Sharma                                             #
#  - based on scripts from Michele Weigle             #
#  - based on scripts from L. Xu, K. Harfoush, I. Rhee      #
#  - based on scripts from S. Floyd and E. Souza            #
#############################################################

#ARGS: endtime flow_type fastAlpha run DATADIR

set begintime 0
set offset1 20000
set period 100000000

#simulation endtime
set endtime [lindex $argv 0]
set flow_type [lindex $argv 1]
set alpha [lindex $argv 2]
set run [lindex $argv 3]
set DATADIR [lindex $argv 4]

set hstcpflows_num 1
set hstcpflows_type 0
set flow_id 0
set bandwidth 622;      #622 Mbps
set delay 48;           #2x48+2x2= 100ms (RTT)

remove-all-packet-headers       ; # removes all except common
add-packet-header Flags IP TCP  ; # hdrs reqd for TCP

source utils.tcl

set frep [open $DATADIR/report.txt w]
set urep [open $DATADIR/util.out w]

set ns [new Simulator]
global defaultRNG
$defaultRNG seed 9999
```

```
for {set i 1} {$i < $run} {incr i} {
   $defaultRNG next-substream
}

Agent/TCP set timestamps_ 1
Agent/TCP set window_ 67000
Agent/TCP set packetSize_ 1000
Agent/TCP set overhead_ 0.000008
Agent/TCP set max_ssthresh_ 100
Agent/TCP set maxburst_ 2

set cubic_tcp_mode 1
set low_window 0
set high_window 83000
set high_p 0.0000001
set high_decrease 0.1
set hstcp_fix 1

#Parameters for BI-TCP
Agent/TCP set bic_beta_ 0.8
Agent/TCP set bic_B_ 4
Agent/TCP set bic_max_increment_ 32
Agent/TCP set bic_min_increment_ 0.01
Agent/TCP set bic_fast_convergence_ 1
Agent/TCP set bic_low_utilization_threshold_ 0
Agent/TCP set bic_low_utilization_checking_period_ 2
Agent/TCP set bic_delay_min_ 0
Agent/TCP set bic_delay_avg_ 0
Agent/TCP set bic_delay_max_ 0
Agent/TCP set bic_low_utilization_indication_ 0

# CUBIC
Agent/TCP set cubic_beta_ 0.8
Agent/TCP set cubic_max_increment_ 16
Agent/TCP set cubic_fast_convergence_ 1
Agent/TCP set cubic_scale_ 0.4
Agent/TCP set cubic_tcp_friendliness_ $cubic_tcp_mode
Agent/TCP set cubic_low_utilization_threshold_ 0
Agent/TCP set cubic_low_utilization_checking_period_ 2
Agent/TCP set cubic_delay_min_ 0
Agent/TCP set cubic_delay_avg_ 0
Agent/TCP set cubic_delay_max_ 0
Agent/TCP set cubic_low_utilization_indication_ 0;

################N/w Setup#################
```

```
set n1 [$ns node]
set n2 [$ns node]
set bf_size 1555

$ns duplex-link $n1 $n2 [expr $bandwidth]Mb [expr $delay]ms DropTail
$ns queue-limit $n1 $n2 $bf_size
$ns queue-limit $n2 $n1 $bf_size

# setup lossy link
set lossylink_ [$ns link $n1 $n2]

set em [new ErrorModule Fid]
set errmodel [new ErrorModel/Periodic]
$errmodel unit pkt
$errmodel set offset_ $offset1
$errmodel set burstlen_ 1
$errmodel set period_ $period
$lossylink_ errormodule $em
$em insert $errmodel
$em bind $errmodel 0
set errmodule [$lossylink_ errormodule]
set errmodel [$errmodule errormodels]

set qmon [$ns monitor-queue $n1 $n2 ""]
set qfp [open $DATADIR/queue.out w]
print-queue 0.01 $qfp

set fmon [$ns makeflowmon Fid]
$ns attach-fmon [$ns link $n1 $n2] $fmon

# back path fmon
set fmon2 [$ns makeflowmon Fid]
$ns attach-fmon [$ns link $n2 $n1] $fmon2

set src_hsnode [$ns node]
set recv_hsnode [$ns node]
$ns duplex-link $src_hsnode $n1 1000Mb 1ms DropTail
$ns duplex-link $recv_hsnode $n2 1000Mb 1ms DropTail
$ns queue-limit $src_hsnode $n1 100000
$ns queue-limit $n1 $src_hsnode 100000
$ns queue-limit $n2 $recv_hsnode 100000
$ns queue-limit $recv_hsnode $n2 100000

if {$flow_type != "FAST"} {
        set hstcp [$ns create-connection TCP/Sack1 $src_hsnode TCPSink/Sack1
$recv_hsnode $flow_id]
```

```
        if {$flow_type == "HS"} {
            set hstcpflows_type 8
            set low_window 38
        } elseif {$flow_type == "Scalable"} {
            set hstcpflows_type 9
            set low_window 16
            [set hstcp] set scalable_lwnd_ $low_window
        } elseif {$flow_type == "BIC"} {
            set hstcpflows_type 12
            set low_window 14
        } elseif {$flow_type == "CUBIC"} {
            set hstcpflows_type 13
        } elseif {$flow_type == "HTCP"} {
            set hstcpflows_type -10
        }
            if {$flow_type != "HTCP"} {
            [set hstcp] set max_ssthresh_ 100
            } else  {
            [set hstcp] set max_ssthresh_ 1
            }

        [set hstcp] set windowOption_ $hstcpflows_type
        [set hstcp] set low_window_ $low_window
        [set hstcp] set high_window_ $high_window
        [set hstcp] set high_p_ $high_p
        [set hstcp] set high_decrease_ $high_decrease
        [set hstcp] set hstcp_fix_ $hstcp_fix
} else {
        set hstcp [$ns create-connection TCP/Fast $src_hsnode TCPSink/Sack1
$recv_hsnode $flow_id]
        [set hstcp] set alpha_ $alpha
        [set hstcp] set beta_ $alpha
}

set hsftp [[set hstcp] attach-app FTP]
set lastBytes 0

$ns at $begintime "[set hsftp] start"
$ns at [expr $endtime+5] "[set hsftp] stop"

# print this connection start time
puts $frep "hstcp starttime: $begintime"

set cfp_hs [open $DATADIR/cwnd_hstcp.out w]
set tput_hs [open $DATADIR/tput_hstcp.out w]
```

```
if { $hstcpflows_type == 12 } {
   print-cwnd-bic [set hstcp] 0.01 $cfp_hs
} else {
   print-cwnd [set hstcp] 0.01 $cfp_hs
}

  set lastKBytes 0
  print-th-one $flow_id $fmon 1

set halftime [expr $endtime / 2]

puts stderr "halftime: $halftime   endtime: $endtime"

$ns at $halftime "print-stat-one-pre $flow_id $fmon"
$ns at $endtime "print-stat-one $flow_id $fmon"

$ns at $halftime "print-stat-all-pre"

$ns at $endtime "printlegend"
$ns at $endtime "finish"

$ns at 0 "timeReport 1"
$ns at 0.1 "print-util 0.1 0"

$ns run
```

Appendix E

Tcl script for running "Single Flow, Different Buffer Sizes" experiments in ns-2

```
#############################################################
# High-Speed TCP Studies - hstcp.tcl                  #
# Pankaj Sharma                                             #
#  - based on scripts from Michele Weigle             #
#  - based on scripts from L. Xu, K. Harfoush, I. Rhee      #
#  - based on scripts from S. Floyd and E. Souza            #
#############################################################

#ARGS: endtime flow_type fastAlpha run DATADIR Qbuff
set begintime 0

#simulation endtime
set endtime [lindex $argv 0]
set flow_type [lindex $argv 1]
set alpha [lindex $argv 2]
set run [lindex $argv 3]
set DATADIR [lindex $argv 4]
set Qbuff [lindex $argv 5]

set hstcpflows_num 1
set hstcpflows_type 0
set flow_id 0
set bandwidth 622;     #622 Mbps
set delay 48;          #2x48+2x2= 100ms (RTT)

remove-all-packet-headers       ; # removes all except common
add-packet-header Flags IP TCP  ; # hdrs reqd for TCP

source utils.tcl

set frep [open $DATADIR/report.txt w]
set urep [open $DATADIR/util.out w]

set ns [new Simulator]
global defaultRNG
$defaultRNG seed 9999

for {set i 1} {$i < $run} {incr i} {
   $defaultRNG next-substream
```

```
}

Agent/TCP set timestamps_ 1
Agent/TCP set window_ 67000
Agent/TCP set packetSize_ 1000
Agent/TCP set overhead_ 0.000008
Agent/TCP set max_ssthresh_ 100
Agent/TCP set maxburst_ 2

set cubic_tcp_mode 1
set low_window 0
set high_window 83000
set high_p 0.0000001
set high_decrease 0.1
set hstcp_fix 1


#Parameters for BI-TCP
Agent/TCP set bic_beta_ 0.8
Agent/TCP set bic_B_ 4
Agent/TCP set bic_max_increment_ 32
Agent/TCP set bic_min_increment_ 0.01
Agent/TCP set bic_fast_convergence_ 1
Agent/TCP set bic_low_utilization_threshold_ 0
Agent/TCP set bic_low_utilization_checking_period_ 2
Agent/TCP set bic_delay_min_ 0
Agent/TCP set bic_delay_avg_ 0
Agent/TCP set bic_delay_max_ 0
Agent/TCP set bic_low_utilization_indication_ 0


# CUBIC
Agent/TCP set cubic_beta_ 0.8
Agent/TCP set cubic_max_increment_ 16
Agent/TCP set cubic_fast_convergence_ 1
Agent/TCP set cubic_scale_ 0.4
Agent/TCP set cubic_tcp_friendliness_ $cubic_tcp_mode
Agent/TCP set cubic_low_utilization_threshold_ 0
Agent/TCP set cubic_low_utilization_checking_period_ 2
Agent/TCP set cubic_delay_min_ 0
Agent/TCP set cubic_delay_avg_ 0
Agent/TCP set cubic_delay_max_ 0
Agent/TCP set cubic_low_utilization_indication_ 0;

################N/w Setup#################
```

```
set n1 [$ns node]
set n2 [$ns node]


$ns duplex-link $n1 $n2 [expr $bandwidth]Mb [expr $delay]ms DropTail

$ns queue-limit $n1 $n2 $Qbuff
$ns queue-limit $n2 $n1 $Qbuff

set qmon [$ns monitor-queue $n1 $n2 ""]
set qfp [open $DATADIR/queue.out w]
print-queue 0.1 $qfp

set fmon [$ns makeflowmon Fid]
$ns attach-fmon [$ns link $n1 $n2] $fmon

# back path fmon
set fmon2 [$ns makeflowmon Fid]
$ns attach-fmon [$ns link $n2 $n1] $fmon2

set src_hsnode [$ns node]
set recv_hsnode [$ns node]
$ns duplex-link $src_hsnode $n1 1000Mb 1ms DropTail
$ns duplex-link $recv_hsnode $n2 1000Mb 1ms DropTail
$ns queue-limit $src_hsnode $n1 100000
$ns queue-limit $n1 $src_hsnode 100000
$ns queue-limit $n2 $recv_hsnode 100000
$ns queue-limit $recv_hsnode $n2 100000

if {$flow_type != "FAST"} {
      set hstcp [$ns create-connection TCP/Sack1 $src_hsnode TCPSink/Sack1
$recv_hsnode $flow_id]

      if {$flow_type == "HS"} {
         set hstcpflows_type 8
         set low_window 38
      } elseif {$flow_type == "Scalable"} {
         set hstcpflows_type 9
         set low_window 16
         [set hstcp] set scalable_lwnd_ $low_window
      } elseif {$flow_type == "BIC"} {
         set hstcpflows_type 12
         set low_window 14
      } elseif {$flow_type == "CUBIC"} {
         set hstcpflows_type 13
      } elseif {$flow_type == "HTCP"} {
```

```
            set hstcpflows_type -10
        }

            if {$flow_type != "HTCP"} {
            [set hstcp] set max_ssthresh_ 100
            } else  {
            [set hstcp] set max_ssthresh_ 1
            }

        [set hstcp] set windowOption_ $hstcpflows_type
        [set hstcp] set low_window_ $low_window
        [set hstcp] set high_window_ $high_window
        [set hstcp] set high_p_ $high_p
        [set hstcp] set high_decrease_ $high_decrease
        [set hstcp] set hstcp_fix_ $hstcp_fix
} else {
        set hstcp [$ns create-connection TCP/Fast $src_hsnode TCPSink/Sack1
$recv_hsnode $flow_id]
        [set hstcp] set alpha_ $alpha
        [set hstcp] set beta_ $alpha
}

set hsftp [[set hstcp] attach-app FTP]
set lastBytes 0

$ns at $begintime "[set hsftp] start"
$ns at [expr $endtime+5] "[set hsftp] stop"

# print this connection start time
puts $frep "hstcp starttime: $begintime"

set cfp_hs [open $DATADIR/cwnd_hstcp.out w]
set tput_hs [open $DATADIR/tput_hstcp.out w]
if { $hstcpflows_type == 12 } {
    print-cwnd-bic [set hstcp] 0.01 $cfp_hs
} else {
    print-cwnd [set hstcp] 0.01 $cfp_hs
}

  set lastKBytes 0
  print-th-one $flow_id $fmon 1

set halftime [expr $endtime / 2]

puts stderr "halftime: $halftime   endtime: $endtime"
```

```
$ns at $halftime "print-stat-one-pre $flow_id $fmon"
$ns at $endtime "print-stat-one $flow_id $fmon"

$ns at $halftime "print-stat-all-pre"

$ns at $endtime "printlegend"
$ns at $endtime "finish"

$ns at 0 "timeReport 1"
$ns at 0.1 "print-util 0.1 0"

$ns run
```

Appendix F

Tcl script for running "Single Flow, Different RTT" experiments in ns-2

```
#############################################################
# High-Speed TCP Studies - hstcp.tcl                  #
# Pankaj Sharma                                             #
#  - based on scripts from Michele Weigle             #
#  - based on scripts from L. Xu, K. Harfoush, I. Rhee      #
#  - based on scripts from S. Floyd and E. Souza            #
#############################################################

#ARGS: endtime flow_type fastAlpha run DATADIR delay

set begintime 0


#simulation endtime
set endtime [lindex $argv 0]
set flow_type [lindex $argv 1]
set alpha [lindex $argv 2]
set run [lindex $argv 3]
set DATADIR [lindex $argv 4]
set delay [lindex $argv 5]


set hstcpflows_num 1
set hstcpflows_type 0
set flow_id 0
set bandwidth 622;      #622 Mbps

remove-all-packet-headers       ; # removes all except common
add-packet-header Flags IP TCP  ; # hdrs reqd for TCP

source utils.tcl

set frep [open $DATADIR/report.txt w]
set urep [open $DATADIR/util.out w]

set ns [new Simulator]
global defaultRNG
$defaultRNG seed 9999
```

```
for {set i 1} {$i < $run} {incr i} {
   $defaultRNG next-substream
}


Agent/TCP set timestamps_ 1
Agent/TCP set window_ 67000
Agent/TCP set packetSize_ 1000
Agent/TCP set overhead_ 0.000008
Agent/TCP set max_ssthresh_ 100
Agent/TCP set maxburst_ 2

set cubic_tcp_mode 1
set low_window 0
set high_window 83000
set high_p 0.0000001
set high_decrease 0.1
set hstcp_fix 1


#Parameters for BI-TCP
Agent/TCP set bic_beta_ 0.8
Agent/TCP set bic_B_ 4
Agent/TCP set bic_max_increment_ 32
Agent/TCP set bic_min_increment_ 0.01
Agent/TCP set bic_fast_convergence_ 1
Agent/TCP set bic_low_utilization_threshold_ 0
Agent/TCP set bic_low_utilization_checking_period_ 2
Agent/TCP set bic_delay_min_ 0
Agent/TCP set bic_delay_avg_ 0
Agent/TCP set bic_delay_max_ 0
Agent/TCP set bic_low_utilization_indication_ 0


# CUBIC
Agent/TCP set cubic_beta_ 0.8
Agent/TCP set cubic_max_increment_ 16
Agent/TCP set cubic_fast_convergence_ 1
Agent/TCP set cubic_scale_ 0.4
Agent/TCP set cubic_tcp_friendliness_ $cubic_tcp_mode
Agent/TCP set cubic_low_utilization_threshold_ 0
Agent/TCP set cubic_low_utilization_checking_period_ 2
Agent/TCP set cubic_delay_min_ 0
Agent/TCP set cubic_delay_avg_ 0
Agent/TCP set cubic_delay_max_ 0
Agent/TCP set cubic_low_utilization_indication_ 0;
```

```
################N/w Setup#################

set n1 [$ns node]
set n2 [$ns node]

set bf_size [expr $bandwidth*$delay*2*1000/[Agent/TCP set packetSize_]/40]; #20% so
40 instead of 8

$ns duplex-link $n1 $n2 [expr $bandwidth]Mb [expr $delay]ms DropTail

$ns queue-limit $n1 $n2 $bf_size
$ns queue-limit $n2 $n1 $bf_size

set qmon [$ns monitor-queue $n1 $n2 ""]
set qfp [open $DATADIR/queue.out w]
print-queue 0.01 $qfp

set fmon [$ns makeflowmon Fid]
$ns attach-fmon [$ns link $n1 $n2] $fmon

# back path fmon
set fmon2 [$ns makeflowmon Fid]
$ns attach-fmon [$ns link $n2 $n1] $fmon2

set src_hsnode [$ns node]
set recv_hsnode [$ns node]
$ns duplex-link $src_hsnode $n1 1000Mb 1ms DropTail
$ns duplex-link $recv_hsnode $n2 1000Mb 1ms DropTail
$ns queue-limit $src_hsnode $n1 100000
$ns queue-limit $n1 $src_hsnode 100000
$ns queue-limit $n2 $recv_hsnode 100000
$ns queue-limit $recv_hsnode $n2 100000

if {$flow_type != "FAST"} {
      set hstcp [$ns create-connection TCP/Sack1 $src_hsnode TCPSink/Sack1
$recv_hsnode $flow_id]

      if {$flow_type == "HS"} {
         set hstcpflows_type 8
         set low_window 38
      } elseif {$flow_type == "Scalable"} {
         set hstcpflows_type 9
         set low_window 16
         [set hstcp] set scalable_lwnd_ $low_window
      } elseif {$flow_type == "BIC"} {
```

```
            set hstcpflows_type 12
            set low_window 14
        } elseif {$flow_type == "CUBIC"} {
            set hstcpflows_type 13
        } elseif {$flow_type == "HTCP"} {
            set hstcpflows_type -10
        }

            if {$flow_type != "HTCP"} {
            [set hstcp] set max_ssthresh_ 100
            } else  {
            [set hstcp] set max_ssthresh_ 1
            }

        [set hstcp] set windowOption_ $hstcpflows_type
        [set hstcp] set low_window_ $low_window
        [set hstcp] set high_window_ $high_window
        [set hstcp] set high_p_ $high_p
        [set hstcp] set high_decrease_ $high_decrease
        [set hstcp] set hstcp_fix_ $hstcp_fix
} else {
        set hstcp [$ns create-connection TCP/Fast $src_hsnode TCPSink/Sack1
$recv_hsnode $flow_id]
        [set hstcp] set alpha_ $alpha
        [set hstcp] set beta_ $alpha
}

set hsftp [[set hstcp] attach-app FTP]
set lastBytes 0

$ns at $begintime "[set hsftp] start"
$ns at [expr $endtime+5] "[set hsftp] stop"

# print this connection start time
puts $frep "hstcp starttime: $begintime"

set cfp_hs [open $DATADIR/cwnd_hstcp.out w]
set tput_hs [open $DATADIR/tput_hstcp.out w]
if { $hstcpflows_type == 12 } {
   print-cwnd-bic [set hstcp] 0.01 $cfp_hs
} else {
   print-cwnd [set hstcp] 0.01 $cfp_hs
}

  set lastKBytes 0
  print-th-one $flow_id $fmon 1
```

```
set halftime [expr $endtime / 2]

puts stderr "halftime: $halftime   endtime: $endtime"

$ns at $halftime "print-stat-one-pre $flow_id $fmon"
$ns at $endtime "print-stat-one $flow_id $fmon"

$ns at $halftime "print-stat-all-pre"

$ns at $endtime "printlegend"
$ns at $endtime "finish"

$ns at 0 "timeReport 1"
$ns at 0.1 "print-util 0.1 0"

$ns run
```

Appendix G

Tcl script for running "Two Flows, Different RTTs" experiments in ns-2

```
#############################################################
# High-Speed TCP Studies - hstcp.tcl                #
# Michele Weigle and Pankaj Sharma                  #
#  - based on scripts from L. Xu, K. Harfoush, I. Rhee #
#  - based on scripts from S. Floyd and E. Souza       #
#############################################################

#############################################################
#TOPOLOGY:                                                  #
#       N-------|                RTT: ((diff+6+1)*2)ms      |----N  #
#       (diff)ms                                           1ms    #
#                (467pkts)     6ms, 622Mbps                      #
#                -------- N   ------------------------------  N--------    #
#                                                                 #
#       74ms                                                1ms    #
#       N------|                 RTT: (162ms base)          |----N  #
#                                                                 #
#############################################################

# ARGS: endtime  flow1_type  flow2_type  FASTalpha  flow2_start run  DATADIR
RTTdiff which

set begintime  [clock second]

# simulation end time
set endtime [lindex $argv 0]

# high-speed flow 1 type, flow 2 type
set hstcp_type(0) [lindex $argv 1];   # HS, Scalable, FAST, BIC, CUBIC
set hstcp_type(1) [lindex $argv 2];   # HS, Scalable, FAST, BIC, CUBIC
set hstcpflows_num 2
set hstcpflows_type 0

# FASTalpha
set alpha [lindex $argv 3]

# flow 2 start time
set flow2_start [lindex $argv 4]
```

```
# run
set run [lindex $argv 5]

# datadir
set DATADIR [lindex $argv 6]

#RTT difference between the two flows
set RTTdiff [lindex $argv 7]

#which flow to add the difference to, 0- 1st flow, 1- 2nd flow
set which [lindex $argv 8]

set bandwidth 622;   # 622 Mbps
set delay 6;        # bottleneck link
set buffer 1;        # qlen is 1 x BDP
set fixed_delay 74;     #delay for the second incoming link which is not varied

# parameters for CUBIC
set cubic_tcp_mode 1

set low_window 0
set high_window 83000
set high_p 0.0000001
set high_decrease 0.1
set hstcp_fix 1

remove-all-packet-headers       ; # removes all except common
add-packet-header Flags IP TCP  ; # hdrs reqd for TCP

source utils.tcl

set frep [open $DATADIR/report.txt w]
set urep [open $DATADIR/util.out w]

puts "------------------------------------------------------------"
puts "------------------------------------------------------------"
puts [pwd]

printlegend

set ns [new Simulator]

global defaultRNG
$defaultRNG seed 9999;

for {set i 1} {$i < $run} {incr i} {
```

```
    $defaultRNG next-substream
}

set n1 [$ns node]
set n2 [$ns node]

Agent/TCP set timestamps_ 1
Agent/TCP set window_ 67000
Agent/TCP set packetSize_ 1000
Agent/TCP set overhead_ 0.000008
Agent/TCP set max_ssthresh_ 100
Agent/TCP set maxburst_ 2

# BIC
Agent/TCP set bic_beta_ 0.8
Agent/TCP set bic_B_ 4
Agent/TCP set bic_max_increment_ 32
Agent/TCP set bic_min_increment_ 0.01
Agent/TCP set bic_fast_convergence_ 1
Agent/TCP set bic_low_utilization_threshold_ 0
Agent/TCP set bic_low_utilization_checking_period_ 2
Agent/TCP set bic_delay_min_ 0
Agent/TCP set bic_delay_avg_ 0
Agent/TCP set bic_delay_max_ 0
Agent/TCP set bic_low_utilization_indication_ 0

# CUBIC
Agent/TCP set cubic_beta_ 0.8
Agent/TCP set cubic_max_increment_ 16
Agent/TCP set cubic_fast_convergence_ 1
Agent/TCP set cubic_scale_ 0.4
Agent/TCP set cubic_tcp_friendliness_ $cubic_tcp_mode
Agent/TCP set cubic_low_utilization_threshold_ 0
Agent/TCP set cubic_low_utilization_checking_period_ 2
Agent/TCP set cubic_delay_min_ 0
Agent/TCP set cubic_delay_avg_ 0
Agent/TCP set cubic_delay_max_ 0
Agent/TCP set cubic_low_utilization_indication_ 0

set bf_size 2519;  #20% of bdp. bdp= 162ms* 622Mbps/8*1000  packets
puts "BufferSize(Packets) $bf_size"
puts $frep "BufferSize(Packets) $bf_size"

$ns duplex-link $n1 $n2 [expr $bandwidth]Mb [expr $delay]ms DropTail
$ns queue-limit $n1 $n2 $bf_size
$ns queue-limit $n2 $n1 $bf_size
```

```
set qmon [$ns monitor-queue $n1 $n2 ""]
set qfp [open $DATADIR/queue.out w]
print-queue 0.1 $qfp

set fmon [$ns makeflowmon Fid]
$ns attach-fmon [$ns link $n1 $n2] $fmon

# back path fmon
set fmon2 [$ns makeflowmon Fid]
$ns attach-fmon [$ns link $n2 $n1] $fmon2

set starttime(0) 0.1
set starttime(1) $flow2_start

for {set i 0} {$i < $hstcpflows_num} {incr i} {
  set hsnode(s$i) [$ns node]
  set hsnode(r$i) [$ns node]

if {$i == $which} {
        $ns duplex-link $hsnode(s$i) $n1 1000Mb [expr $RTTdiff/2]ms DropTail
} else {
        $ns duplex-link $hsnode(s$i) $n1 1000Mb [expr $fixed_delay]ms DropTail
}
  $ns duplex-link $hsnode(r$i) $n2 1000Mb 1ms DropTail
  puts "hstcp$i, forwardlink delay=1 ms"
  puts $frep "hstcp$i, forwardlink delay=1 ms"
  puts "hstcp$i, RTT = 100.00 ms"
  puts $frep "hstcp$i, RTT = 100.00 ms"

  $ns queue-limit $hsnode(s$i) $n1 67000
  $ns queue-limit $n1 $hsnode(s$i) 67000

  $ns queue-limit $n2 $hsnode(r$i) 67000
  $ns queue-limit $hsnode(r$i) $n2 67000

   if {$hstcp_type($i) != "FAST"} {
        set hstcp$i [$ns create-connection TCP/Sack1 $hsnode(s$i) TCPSink/Sack1
$hsnode(r$i) $i]

        if {$hstcp_type($i) == "HS"} {
           set hstcpflows_type 8
           set low_window 38
        } elseif {$hstcp_type($i) == "Scalable"} {
           set hstcpflows_type 9
           set low_window 16
```

```
            [set hstcp$i] set scalable_lwnd_ $low_window
        } elseif {$hstcp_type($i) == "BIC"} {
            set hstcpflows_type 12
            set low_window 14
        } elseif {$hstcp_type($i) == "CUBIC"} {
            set hstcpflows_type 13
        } elseif {$hstcp_type($i) == "HTCP"} {
        set hstcpflows_type -10
      }

if {$hstcp_type($i) != "HTCP"} {
      [set hstcp$i] set max_ssthresh_ 100
} else  {
      [set hstcp$i] set max_ssthresh_ 1
}

        [set hstcp$i] set windowOption_ $hstcpflows_type
        [set hstcp$i] set low_window_ $low_window
        [set hstcp$i] set high_window_ $high_window
        [set hstcp$i] set high_p_ $high_p
        [set hstcp$i] set high_decrease_ $high_decrease
        [set hstcp$i] set hstcp_fix_ $hstcp_fix
   } else {
        set hstcp$i [$ns create-connection TCP/Fast $hsnode(s$i) TCPSink/Sack1
$hsnode(r$i) $i]
      [set hstcp$i] set alpha_ $alpha
      [set hstcp$i] set beta_ $alpha
   }

  set hsftp$i [[set hstcp$i] attach-app FTP]

  set lastBytes($i) 0

  $ns at $starttime($i) "[set hsftp$i] start"
  $ns at [expr $endtime+5] "[set hsftp$i] stop"

  # print this connection start time
  puts "hstcp$i starttime: $starttime($i)"
  puts $frep "hstcp$i starttime: $starttime($i)"

  set cfp_hs($i) [open $DATADIR/cwnd_hstcp$i.out w]
  set tput_hs($i) [open $DATADIR/tput_hstcp$i.out w]
  if { $hstcpflows_type == 12 } {
    print-cwnd-bic [set hstcp$i] 0.1 $cfp_hs($i)
  } else {
    print-cwnd [set hstcp$i] 0.1 $cfp_hs($i)
```

```
  }

  set lastKBytes($i) 0
  print-th-one $i $fmon 1
}

set halftime [expr $endtime / 2]

puts stderr "halftime: $halftime   endtime: $endtime"

for {set i 0} {$i < $hstcpflows_num} {incr i} {
  $ns at $halftime "print-stat-one-pre $i $fmon"
  $ns at $endtime "print-stat-one $i $fmon"
}

$ns at $halftime "print-stat-all-pre"
$ns at $endtime "print-stat-all"

$ns at $endtime "printlegend"
$ns at $endtime "finish"

$ns at 0 "timeReport 1"
$ns at 0.1 "print-util 0.1 0"

$ns run
```

Appendix H

Tcl script for running Inter-Protocol experiments in ns-2

```
#############################################################
# High-Speed TCP Studies - hstcp.tcl                        #
# Michele Weigle and Pankaj Sharma                          #
#  - based on scripts from L. Xu, K. Harfoush, I. Rhee #
#  - based on scripts from S. Floyd and E. Souza       #
#############################################################

# ARGS: endtime  flow1_type  flow2_type  FASTalpha  flow2_start run  DATADIR

set begintime  [clock second]

# simulation end time
set endtime [lindex $argv 0]

# high-speed flow 1 type, flow 2 type
set hstcp_type(0) [lindex $argv 1];   # HS, Scalable, FAST, BIC, CUBIC
set hstcp_type(1) [lindex $argv 2];   # HS, Scalable, FAST, BIC, CUBIC
set hstcpflows_num 2
set hstcpflows_type 0

# FASTalpha
set alpha [lindex $argv 3]

# flow 2 start time
set flow2_start [lindex $argv 4]

# run
set run [lindex $argv 5]

# datadir
set DATADIR [lindex $argv 6]

set bandwidth 622;   # 622 Mbps
set delay 48;        # total of 100 ms RTT
set buffer 1;        # qlen is 1 x BDP

# parameters for CUBIC
set cubic_tcp_mode 1
```

```
set low_window 0
set high_window 83000
set high_p 0.0000001
set high_decrease 0.1
set hstcp_fix 1

remove-all-packet-headers       ; # removes all except common
add-packet-header Flags IP TCP  ; # hdrs reqd for TCP

source utils.tcl

set frep [open $DATADIR/report.txt w]
set urep [open $DATADIR/util.out w]

puts "------------------------------------------------------------"
puts "------------------------------------------------------------"
puts [pwd]

printlegend

set ns [new Simulator]
#$ns use-scheduler Heap

global defaultRNG
#$defaultRNG seed 0;   # set the seed to current time/date
$defaultRNG seed 9999;

for {set i 1} {$i < $run} {incr i} {
   $defaultRNG next-substream
}

set n1 [$ns node]
set n2 [$ns node]

Agent/TCP set timestamps_ 1
Agent/TCP set window_ 67000
Agent/TCP set packetSize_ 1000
Agent/TCP set overhead_ 0.000008
Agent/TCP set max_ssthresh_ 100
Agent/TCP set maxburst_ 2

# BIC
Agent/TCP set bic_beta_ 0.8
Agent/TCP set bic_B_ 4
Agent/TCP set bic_max_increment_ 32
Agent/TCP set bic_min_increment_ 0.01
```

```
Agent/TCP set bic_fast_convergence_ 1
Agent/TCP set bic_low_utilization_threshold_ 0
Agent/TCP set bic_low_utilization_checking_period_ 2
Agent/TCP set bic_delay_min_ 0
Agent/TCP set bic_delay_avg_ 0
Agent/TCP set bic_delay_max_ 0
Agent/TCP set bic_low_utilization_indication_ 0

# CUBIC
Agent/TCP set cubic_beta_ 0.8
Agent/TCP set cubic_max_increment_ 16
Agent/TCP set cubic_fast_convergence_ 1
Agent/TCP set cubic_scale_ 0.4
Agent/TCP set cubic_tcp_friendliness_ $cubic_tcp_mode
Agent/TCP set cubic_low_utilization_threshold_ 0
Agent/TCP set cubic_low_utilization_checking_period_ 2
Agent/TCP set cubic_delay_min_ 0
Agent/TCP set cubic_delay_avg_ 0
Agent/TCP set cubic_delay_max_ 0
Agent/TCP set cubic_low_utilization_indication_ 0

#set bf_size [expr $bandwidth*$delay*2*1000*$buffer/[Agent/TCP set packetSize_]/8]
set bf_size 1555
puts "BufferSize(Packets) $bf_size"
puts $frep "BufferSize(Packets) $bf_size"

$ns duplex-link $n1 $n2 [expr $bandwidth]Mb [expr $delay]ms DropTail
$ns queue-limit $n1 $n2 $bf_size
$ns queue-limit $n2 $n1 $bf_size

set qmon [$ns monitor-queue $n1 $n2 ""]
set qfp [open $DATADIR/queue.out w]
print-queue 0.1 $qfp

set fmon [$ns makeflowmon Fid]
$ns attach-fmon [$ns link $n1 $n2] $fmon

# back path fmon
set fmon2 [$ns makeflowmon Fid]
$ns attach-fmon [$ns link $n2 $n1] $fmon2

#set rng_ [new RNG]

#for {set i 1} {$i < $run} {incr i} {
#    $rng_ next-substream
#}
```

```
set starttime(0) 0.1
set starttime(1) $flow2_start

for {set i 0} {$i < $hstcpflows_num} {incr i} {
  set hsnode(s$i) [$ns node]
  set hsnode(r$i) [$ns node]

#  set del [$rng_ uniform 0.1 1.9]

#  $ns duplex-link $hsnode(s$i) $n1 1000Mb ${del}ms DropTail
  $ns duplex-link $hsnode(s$i) $n1 1000Mb 1ms DropTail
  $ns duplex-link $hsnode(r$i) $n2 1000Mb 1ms DropTail
#  puts "hstcp$i, forwardlink delay=[format %.2f $del] ms"
#  puts $frep "hstcp$i, forwardlink delay=$del ms"
#  puts "hstcp$i, RTT = [format %.2f [expr ($del+1.0+$delay)*2 ]] ms"
#  puts $frep "hstcp$i, RTT = [format %.2f [expr ($del+1.0+$delay)*2 ]] ms"
  puts "hstcp$i, forwardlink delay=1 ms"
  puts $frep "hstcp$i, forwardlink delay=1 ms"
  puts "hstcp$i, RTT = 100.00 ms"
  puts $frep "hstcp$i, RTT = 100.00 ms"

  $ns queue-limit $hsnode(s$i) $n1 67000
  $ns queue-limit $n1 $hsnode(s$i) 67000

  $ns queue-limit $n2 $hsnode(r$i) 67000
  $ns queue-limit $hsnode(r$i) $n2 67000

   if {$hstcp_type($i) != "FAST"} {
        set hstcp$i [$ns create-connection TCP/Sack1 $hsnode(s$i) TCPSink/Sack1
$hsnode(r$i) $i]

        if {$hstcp_type($i) == "HS"} {
           set hstcpflows_type 8
           set low_window 38
        } elseif {$hstcp_type($i) == "Scalable"} {
           set hstcpflows_type 9
           set low_window 16
           [set hstcp$i] set scalable_lwnd_ $low_window
        } elseif {$hstcp_type($i) == "BIC"} {
           set hstcpflows_type 12
           set low_window 14
        } elseif {$hstcp_type($i) == "CUBIC"} {
           set hstcpflows_type 13
        } elseif {$hstcp_type($i) == "HTCP"} {
        set hstcpflows_type -10
```

```
        }

if {$hstcp_type($i) != "HTCP"} {
        [set hstcp$i] set max_ssthresh_ 100
} else  {
        [set hstcp$i] set max_ssthresh_ 1
}

            [set hstcp$i] set windowOption_ $hstcpflows_type
            [set hstcp$i] set low_window_ $low_window
            [set hstcp$i] set high_window_ $high_window
            [set hstcp$i] set high_p_ $high_p
            [set hstcp$i] set high_decrease_ $high_decrease
            [set hstcp$i] set hstcp_fix_ $hstcp_fix
    } else {
#           set hstcp$i [$ns create-connection TCP/Fast $hsnode(s$i) TCPSink $hsnode(r$i)
$i]
            set hstcp$i [$ns create-connection TCP/Fast $hsnode(s$i) TCPSink/Sack1
$hsnode(r$i) $i]
        [set hstcp$i] set alpha_ $alpha
        [set hstcp$i] set beta_ $alpha
    }

  set hsftp$i [[set hstcp$i] attach-app FTP]

  set lastBytes($i) 0

  $ns at $starttime($i) "[set hsftp$i] start"
  $ns at [expr $endtime+5] "[set hsftp$i] stop"

  # print this connection start time
  puts "hstcp$i starttime: $starttime($i)"
  puts $frep "hstcp$i starttime: $starttime($i)"

  set cfp_hs($i) [open $DATADIR/cwnd_hstcp$i.out w]
  set tput_hs($i) [open $DATADIR/tput_hstcp$i.out w]
  if { $hstcpflows_type == 12 } {
    print-cwnd-bic [set hstcp$i] 0.1 $cfp_hs($i)
  } else {
    print-cwnd [set hstcp$i] 0.1 $cfp_hs($i)
  }

  set lastKBytes($i) 0
  print-th-one $i $fmon 1
}
```

```
set halftime [expr $endtime / 2]

puts stderr "halftime: $halftime   endtime: $endtime"

for {set i 0} {$i < $hstcpflows_num} {incr i} {
  $ns at $halftime "print-stat-one-pre $i $fmon"
  $ns at $endtime "print-stat-one $i $fmon"
}

$ns at $halftime "print-stat-all-pre"
$ns at $endtime "print-stat-all"
$ns at $endtime "printlegend"
$ns at $endtime "finish"

$ns at 0 "timeReport 1"
$ns at 0.1 "print-util 0.1 0"

$ns run
```

REFERENCES

[BCH03] H. Bullot, R. L. Cottrell, and R. Hughes-Jones. Evaluation of advanced TCP stacks on fast long-distance production networks. Journal of Grid Computing, 1(4):345–359, 2003. Graphs available at http://www-iepm.slac.stanford.edu/bw/tcp-eval/.

[CTF] http://netlab.caltech.edu/FAST/

[LLS05] Yee-Ting Li, Douglas Leith, and Robert Shorten, "Experimental Evaluation of TCP Protocols for High-Speed Networks", Technical report, Hamilton Institute, 2005

[HLW+04] S. Hegde, D. Lapsley, B.Wydrowski, J. Lindheim, D.Wei, C. Jin, S. Low,H. Newman, "FAST TCP in High Speed Networks: An Experimental Study", Proceedings GridNets, San Jose, 2004

[JWL04] C. Jin, D.X. Wei, S,H. Low, "FAST TCP: motivation, architecture, algorithms, performance", Proceedings IEEE INFOCOM 2004

[HKL+06] S. Ha, Y. Kim, L. Le, I. Rhee, and L. Xu. "A Step toward Realistic Performance Evaluation of High-Speed TCP Variants", in International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet 2006), Nara, Japan, February 2006

[XHR04] L. Xu, K. Harfoush, and I. Rhee, "Binary increase congestion control for fast long-distance networks", In Proceedings of IEEE INFOCOM, Hong Kong, Mar. 2004.

[NLR] National LambdaRail Project. http://www.nlr.net/.

[Flo03] S. Floyd. Highspeed TCP for large congestion windows, Dec. 2003. RFC 3649, Experimental.

[FRS02] S. Floyd, S. Ratnasamy, and S. Shenker. Modifying TCP's congestion control for high speeds, May 2002. Technical Note, available at http://www.icir.org/floyd/papers/hstcp.pdf.

[Kel03] T. Kelly. Scalable TCP: Improving performance in highspeed wide area networks. In Proceedings of PFLDnet, Geneva, Switzerland, Feb. 2003.

[RX05] I. Rhee and L. Xu. CUBIC: A new TCP-friendly high-speed TCP variant. In Proceedings of PFLDnet, Lyon, France, Feb. 2005.

[WSF06] M. Weigle, P Sharma and J. Freeman, Performance of Competing High-Speed TCP Flows, Proceedings of IFIP Networking2006, May 2006.

[CJ89] D. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. Computer Networks and ISDN Systems, pages 1–14, June 1989.

[FMD+01] C. Fraleigh, S. Moon, C. Diot, B. Lyles and F. Tobagi, "Packet-Level Traffic Measurements from a Tier-1 IP Backbone", Sprint ATL Technical Report TR01-ATL-110101, November 2001.

[AKM04] G. Appenzeller, I. Keslassy, and N. Mckeown, "Sizing router buffers," in *Proceedings of ACM SIGCOMM'04*, 2004.

[VS94] C. Villamizar and C. Song. High performance tcp in ansnet. ACM Computer Communications Review, 24(5):45{60, 1994 1994.

[MF] S.McCanne and S. Floyd. ns Network Simulator. Software available at http://www.isi.edu/nsnam/ns/.

[RX05] I. Rhee and L. Xu. Simulation code and scripts for CUBIC, 2005. Available at http://www.csc.ncsu.edu/faculty/rhee/export/bitcp/cubic-script/script.htm.

[LS+05] D. Leigh, R. Shorten, et al. H-TCP ns-2 implementation, 2005. Available at http://www.hamilton.ie/net/research.htm#software.

[CA04] T. Cui and L. Andrew. FAST TCP simulator module for ns-2, version 1.1, 2004. Available at http://www.cubinlab.ee.mu.oz.au/ns2fasttcp