NOVEL PLACEMENT OF CONGESTION CONTROL FUNCTIONS IN THE INTERNET

By

Kartikeya Chandrayana

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer and Systems Engineering

Approved by the
Examining Committee:

Prof. Shivkumar Kalyanaraman, Thesis Adviser

Prof. Christopher D. Carothers, Member

Prof. Koushik Kar, Member

Prof. Biplab Sikdar, Member

Prof. John T. Wen, Member

Rensselaer Polytechnic Institute
Troy, New York

April 2004
(For Graduation May 2004)

NOVEL PLACEMENT OF CONGESTION CONTROL FUNCTIONS IN THE INTERNET

By

Kartikeya Chandrayana

An Abstract of a Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer and Systems Engineering

The original of the complete thesis is on file
in the Rensselaer Polytechnic Institute Library

Examining Committee:

Prof. Shivkumar Kalyanaraman, Thesis Adviser
Prof. Christopher D. Carothers, Member
Prof. Koushik Kar, Member
Prof. Biplab Sikdar, Member
Prof. John T. Wen, Member

Rensselaer Polytechnic Institute
Troy, New York

April 2004
(For Graduation May 2004)

© Copyright 2004
by
Kartikeya Chandrayana
All Rights Reserved

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

Foremost, I would like to thank my adviser Prof. Shivkumar Kalyanaraman. He has been very patient with me, allowing me to work in different areas of networking. I am deeply indebted to him for consistently reminding me of my priorities and often correcting my drift. His continuous guidance and infinite wisdom are primarily the reasons behind the successful completion of this work.

I would also like to thank my committee members, specifically Prof. Christopher Carothers, Prof. Koushik Kar, Prof. Biplab Sikdar and Prof. John T. Wen for their guidance, insightful comments and encouragement. I would especially like to thank Prof. Biplab Sikdar, part-mentor and part-friend who has been eminently accessible for anything at any time of the day.

This thesis would not have been possible without the generous help received from my friends. Satish Raghunath has been my man friday for all systems and programming related problems, which I had in abundance. I will always cherish the endless discussions I had with him on a variety of subjects including some critical and valuable assessment of my research. During the past couple of years, Omesh Tickoo has been there for various discussions and critical reviews over coffee, at all time after daylight. I cannot thank him enough.

Sthanu, Omesh, Anand, Manoj and Alex Newman need a special mention as they have helped me at various points in time with some of the research presented in this thesis. I would also like to express my gratitude to Jagoron, Omesh and Ayesha for taking good care of me in the past three years, especially while I wrote my candidacy and thesis documents. I would also like to thank all my colleagues in the Networks Lab for tolerating all my idiosyncrasies and musical tastes.

Finally, I would like to thank my family for coping up with all my mood swings and frequent lack of communication. They have taught me important and good things in life including the value of education.

ABSTRACT

Over the years the Internet has become a critical part of modern world. Prudent congestion control mechanisms have been primarily responsible for the stability and robustness of the Internet. However, these lessons about the necessity of congestion control mechanisms were learnt after a series of failures in the formative years of the Internet. As a result of these failures, considerable research efforts have been spent at understanding congestion control and many solutions to avoid and control congestion in the Internet have been proposed. Of the proposed solutions, end-system based congestion avoidance and control is now an integral part of the most widely used transport protocol, TCP. However, subsequent research which provided complementary network based solutions for managing congestion are yet to be deployed on the Internet, for a variety of reasons.

In this thesis we re-evaluate the function placement of the building blocks of the Internet congestion control architecture. Specifically, we attempt to bridge the gap between the deployment costs and desirable congestion control features. Towards this end, this thesis proposes a series of deployable end-and-edge based solutions which combine almost all the beneficial properties of existing congestion control solutions. An essential step for achieving such deployable solution lies in *decoupling the congestion control tasks from their placement* in the network. As such, this thesis proposes end-and-edge based architectures which help us de-construct AQM schemes, thus disassociating congestion control tasks from their placement.

In this thesis we proposes an end-system based solution, *Randomized TCP*, which can alleviate the network and end-to-end performance degradations which arise out of use of TCP on simple FIFO queueing. *Uncooperative Congestion Control* is a proposed network-based architecture which de-couples management of selfish behavior of flows in the Internet from router based AQM policies. Specifically, this framework helps prevent traffic volume based denial of service, enforces congestion response conformance and provides an architecture for implementing simple differentiated services. Finally, this thesis also proposes *virtual AQM* an end-or-edge

based solution for managing bottleneck queues in the Internet. The virtual AQM framework can help us send early congestion indications and also reduce steady state queue size and latency.

CHAPTER 1
Introduction

Over the years as the Internet has evolved TCP has formed the backbone of its stability. However, a decade ago the present Internet suffered from a severe congestion control problem which was called the "Internet meltdown". To prevent such a situation Jacobson [49] proposed the congestion avoidance and control mechanisms for TCP which has subsequently become the de-facto transport protocol for the Internet.

However as the application needs changed newer rate control schemes were proposed. As such we now have an Internet which operates with a spectrum of congestion control schemes, even though TCP remains the most widely used transport protocol. In [35] the authors have argued that these new congestion control schemes can lead to a new congestion collapse and pose the problem of congestion response conformance (wherein selfish schemes get an unfavorable share of bandwidth in comparison to TCP).

Though end-system based congestion control mechanisms have helped prevent Internet meltdown they are not sufficient to provide good service under all circumstances. Specifically, network and end-user performance may degrade in presence of Drop Tail queues and different rate control schemes [39, 37, 4, 17, 18, 19, 60]. Also end system based solutions constrain the choices of flow control protocols which might be available to any application. Towards addressing these issue, network (or router) based schemes like Active Queue Management (AQM) have been proposed to complement the end-system based congestion control schemes. Thus, together with end-system based congestion avoidance and control schemes, AQM form the building blocks of the Internet congestion control architecture.

However these AQM proposals are beset with configuration problems [31, 28, 60, 22, 20] and also require significant upgrade of the network (i.e. each bottleneck must have the AQM enabled) which is not only expensive but also infeasible. Lack of deployment of ECN (Explicit Congestion Notification) is another reason which

holds back the deployment of AQM schemes. Further, providers do not want to drop a packet unless the network is congested and this further hampers the deployment of AQM proposals. As a result of these implementation drawbacks and deployment considerations, the Internet still operates with Drop Tail queues.

Considering that AQM schemes are still to be widely deployed on the Internet, presence of different congestion control schemes and ubiquity of Drop Tail queueing in the Internet, in this thesis we re-evaluate the function placement of building blocks of congestion control. In particular, we in this thesis we explore *deployable* end system and network based solutions which combine all beneficial properties of both end-and-network based congestion control building blocks. In other words, we propose end-and-edge based solutions to emulate the beneficial properties of AQM over network of Drop Tail queues, while allowing end users the flexibility to choose their congestion control schemes. In summary, in this thesis we argue that almost all tasks involving Internet congestion control can be done at the network edge or end-system without any support from the core of the network.

In the following sections we first present the various end-and-network based solutions for preventing congestion collapse in the Internet and their current deployment status in the Internet. Thereafter we present our end-and-edge based proposals for managing congestion in the Internet. While we keep a holistic view of congestion control architectures, in this chapter we will focus on smaller issues which arise out of management of congestion in the Internet. In particular, we will discuss issues related to management of bottleneck queue lengths, fairness and congestion response conformance (or protection of TCP flows from malicious users) in the Internet.

1.1 A Macroscopic view of Congestion Control: Solutions for Preventing Congestion Collapse in the Internet

The Internet protocol architecture is based on a connectionless end-to-end packet service using the IP protocol. The advantages of its connectionless design, flexibility and robustness have been amply demonstrated. However, these advantages are not without cost: careful design is required to provide good service under heavy load. In fact, lack of attention to the dynamics of packet forwarding can

result in severe service degradation or "Internet meltdown".

As a result of this meltdown considerable research has been done on Internet dynamics and many solutions have been suggested to avoid it. These proposals can be broadly classified into two categories a) end-system based solutions, e.g. TCP and other congestion control schemes and b) network based solution. In this section we briefly discuss these proposals, their advantages and disadvantages and their current deployment in the Internet.

1.1.1 End System Based Approaches

The end-system based solutions consists of source or receiver based congestion control schemes. These schemes try to avoid congestion in the Internet by cutting down their transmission rate, whenever congestion is detected. The original fix for the congestion collapse (or Internet meltdown) proposed by Jacobson in 1988 [49] is one such scheme. In particular, Jacobson proposed the congestion avoidance and control features in TCP and since then TCP has been the mainstay of the Internet.

These end-system based solutions can operate with and without network support. In absence of network support the network employs simple queuing at the routers in which the packets are admitted till the queue has space. This queuing policy is called Drop Tail. Though simple to implement, Drop Tail queue do not try to manage congestion in the network, in fact it is left to the end-system based application.

Though TCP has served the Internet community well it is known to suffer from a number of phenomena which limits its effectiveness when operated over a network of Drop Tail queues. The main problem which degrades TCP and network performance are: synchronization of congestion windows (or correspondingly the loss instances) causing alternate overloading and under-loading of the bottleneck [70, 86, 101]; phase effects wherein a certain section of flows face recurrent losses [38]; unfairness to flows with higher RTTs [32]; bias against bursty traffic [39] ; delays and losses due to the bursty nature of TCP traffic [101, 4].

The tail-drop discipline allows queues to maintain a full status for long periods of time. This is because Drop Tail signals congestion only when the queue has

become full. If the queue is full, an arriving burst will cause multiple packets (from same or different flows) to be dropped causing global synchronization [39]. This synchronization can be attributed to two reasons: (1) the sliding window flow control of the TCP, which produces bursts of packets and (2) the Drop Tail queue at the bottleneck, which drops all packets when the buffer is full [45]. *Synchronization* of windows and loss events for flows sharing common links causes alternating periods of overload and under-load thereby leading to inefficient resource utilization. In some situations Drop Tail queuing allows a single connection or a few flows to monopolize queue space, preventing other connections from getting room in the queue. This "lock-out" phenomenon is often the result of synchronization [39, 14] and full queues.

Phase effects refer to conditions where in the bandwidth-delay product of the path of a flow is not an integral multiple of the packet size [38]. Phase effects cause a specific section of competing flows to experience recurrent drops causing unfair distribution of bandwidth and increased latency. Phase effects are manifested in the network preferentially dropping packets from a specific subset of flows thereby reducing their throughput.

Drop Tail queues suffer from a problem called, "full queues", which implies that Drop Tail queuing maintains sustained long or full queues. The sliding window protocol of TCP and the persistent full queues often results in burst losses. These burst losses causes Drop Tail queues to differentiate against TCP like schemes [39]. It has been widely shown that TCP can recover well from a single packet loss but with burst losses it often times out [89]. Consequently, these burst losses also increase the delays. It has also been reported that these burst losses are the primary reason for the bias against flows with longer RTT [3].

Drop Tail queues also do not protect flows. As noted earlier because of synchronization Drop Tail queues can let some flows monopolize the buffer space. Also, given that there are various congestion control schemes in the network, by not differentiating amongst flows, Drop Tail queues allow aggressive sources to get more bandwidth. Flows which do not react to congestion indications will push the responsive flows out of the queue and will always take up bandwidth worth their

transmission rate [60, 17, 18]. Thus by introducing burst losses and by not protecting flows, Drop Tail queues aggravate the problem of unfair equilibrium rate allocations in the network.

1.1.2 Network Based Approaches

Though the end-system based congestion avoidance and control mechanisms are necessary and powerful, they are not sufficient to provide good service under all circumstances. Primarily, there is a limit to how much control that can be accomplished from the end of the network. Specifically, these problems were highlighted in the previous section and can be chiefly attributed to the full queues and lock-out behavior of the Drop Tail queues. Thus some mechanisms are needed in the routers to complement the endpoint congestion control and avoidance mechanisms.

Active Queue Management (AQM) was suggested as a pro-active way of managing queue at the bottleneck router. The pro-activeness was defined to be able to drop few packets before the queue gets full thereby signaling sources to cut their rates on account of impending congestion. This in turn help solved the problem of full queues. The solution to the full queues problem implied that there would be space in the queue to enque packets which consequently solved the lock-out problem of Drop Tail queues and avoid burst losses.

Random Early Drop (or RED) [39] was one such AQM proposal wherein the authors suggested to probabilistically drop packet when the queue size gets above a certain threshold. This probabilistic dropping distributed losses over time thus making them appear independent. It also introduced randomization at the bottleneck which in turn broke synchronization amongst flows and improved network performance. Also, by sending early congestion signal (by dropping a packet before the queue actually gets full) helped manage queues efficiently and also provided space to accommodate bursts. Thus RED avoids burst losses, synchronization, reduces the bias against long RTT flows and prevents timeouts.

However it's been almost a decade since the RED was proposed and the Internet continues to operates with Drop Tail queues. This can be explained by absence of concrete guidelines to set RED, requirement of significant network upgrade and

lack of deployment of ECN. Studies have shown that if not properly configured the performance of TCP with RED queues may be even worse than that with Drop Tail queues. Specifically, in [67, 22] the authors show that the probability of consecutive drops increases with RED queues. Though there have been some studies on how to configure RED these works attempt to configure only one or a set of parameters and as such have not found much favor with the network operators [20, 46]. Besides RED there have been other AQM proposals which have fewer parameters to configure and crisper guidelines for setting them [57, 7, 28, 91, 90, 29]. But in-spite of numerous AQM propositions the network still operates with Drop Tail queues and consequently the problems of TCP and Drop Tail queues exist to this day.

1.1.3 Summary

The policies outlined for preventing congestion collapse require either end-system support in form of congestion control scheme or router based schemes like AQMs. However there is a limit to the control which can be achieved by using the end-point congestion schemes. In absence of network control we have seen that the flows are subjected to burst losses and can get synchronized which in turn limits the effectiveness of TCP. Further, end-point congestion control scheme do not protect flows, on the contrary they allow some flows to monopolize the buffer space.

Network control for preventing congestion collapse was envisioned in form of Active Queue Management. RED was one such proposal which absorbed bursts by probabilistically enqueing packets. This introduced randomness at the bottleneck and helped avoid synchronization of flows. However RED's performance is highly sensitive to its parameter configuration, so much so that at times the performance of Drop Tail queues might be better than RED queues [13]. This problem is further compounded by the lack of guidelines for setting these parameters. Moreover, all AQM proposals including RED require deployment at all (bottleneck) routers in the network which not only require extensive network upgrade but is also expensive. As a result of these concerns, the network still operates with Drop Tail queues.

To summarize, due to configuration, implementation and deployment problems with AQM the Internet continues to operates with Drop Tail queues. As a result,

the problems of bursts losses, flow synchronization, bias against flows with longer RTT and manipulation of buffer space by selfish and unresponsive flows persist. These problems in turn limit the effectiveness of TCP and degrade the performance of the network.

1.2 A Microscopic View of Congestion Control: Fairness, Congestion Response Conformance and Management of Bottleneck Queues in the Internet

1.2.1 Fairness

Fairness can be defined in a number of ways but its essence in each of these definitions is that it is some measure of the distribution of the allocated rate amongst users. Fairness is related not only to the network but also to the end-system's congestion control scheme. Often the end-system's objective is to be fair to the other competing user's while the network's objective is that it does not arbitrarily penalize or differentiate amongst various competing user.

Traditionally, the Internet has relied on the "end-to-end" congestion control model like TCP (or alternate transport protocols) where end users choose a rate control scheme, and the network merely drops or marks packets during congestion as a method to convey the penalty or price [53, 56, 62]. One implication of this model is that end-systems are free to choose *any* rate control scheme (which in the Kelly's framework [53] means that they can pick any desired utility function). Kelly, Low etal have shown that a particular class of fairness is associated with every utility function. Thus, by choosing different utility functions varying equilibrium rate distributions or fairness criteria can be achieved. The end users, therefore might willfully seek to guard and increase their interests and choose the utility function which best suits their application, thus promoting selfish behavior in the network.

The network on the other hand seldom differentiates between flows. It drops (or marks) packets obliviously, i.e. drops packets whenever there is no space in the router queue or if the router queue length crosses a certain threshold. Drop Tail queuing, RED and many of RED's variants can be classified as oblivious queuing

disciplines. This oblivious dropping coupled with the flexibility of end system to chose a rate control scheme makes the problem of providing fair rate allocations to all users hard. The "fair" equilibrium allocations in an oblivious network therefore depend upon the utility functions chosen freely by users. These equilibrium allocations, though fair under Kelly's framework, might be unfair from network perspective.

Moreover the fair rate allocation problem is further compounded by the fact that there is no single definition of fairness. The two most common definitions of fairness are *max-min* [13, 69] and *proportional* fairness [53]. In *max-min* fairness criteria the objective is to maximize the minimum unsatisfied rate allocations. Thus given the same network conditions, two competing flows should get equal share of the bottleneck. On the other hand, in *proportional* fairness the rate allocations are in proportion to the network resources being used. But all the same, a more general definition of fairness, *(p,α) fairness* is defined in [69]. Thus irrespective of user's rate control schemes it is for the network provider to decide the criteria for allocating resources amongst users.

Over the years equal allocations and Max-Min fairness [69] have formed the network's view of fair allocations. As such, AQM schemes and schedulers deployed at every bottleneck have been used to enforce conformance with these definitions, by penalizing misbehaving users [64, 30, 77, 60, 87, 94]. For example, CHOKe [77] tries to enforce Max-Min fairness [69] across the network. Similarly fair queuing and it's variant have also been used to provide Max-Min fairness.

However, recent efforts in pricing based rate distribution infrastructure implies that the providers might differentiate amongst users on the basis of their willingness to pay. This is in direct contrast with equal and max-min allocation strategies. As a result of differentiation on willingness to pay, proportional fairness and it's variants are becoming popular with network providers. However, none of the current AQM proposals attempt to provide end-to-end proportionally fair rate distribution, at best they can provide max-min fair equilibrium rate allocation. Thus to summarize, any arbitrary fairness objective cannot be achieved by AQM schemes though they could be arrived at by use of schedulers throughout the network.

1.2.2 Congestion Response Conformance

The congestion control scheme in TCP has been the focus of numerous studies and consequently gone through lots of changes. These changes were also motivated by varying needs of the applications using the Internet. As such, even though TCP remains the most widely used protocol, we have now a spectrum of congestion control schemes. In [35] the authors show that absence of end-to-end congestion control schemes or presence of selfish users could not only lead to TCP being beaten down but also may even result in congestion collapse.

This thus represents the problem of congestion response conformance. In absence of compliance to a set of protocols, for example TCP, we might be faced with the problem of TCP flows being singled out and gets rates which are (significantly) less than their fair share. This problem is further highlighted in presence of unresponsive flows. (Flows which do not cut down their rates upon receipt of congestion indication are called unresponsive flows.) These unresponsive flows can shut-out TCP because on occurrence of congestion, TCP will cut its rate and the unresponsive flows will step in to take the available bandwidth. A similar problem is posed by responsive selfish flows (i.e. flows who react to congestion indication but are selfish as compared to TCP) in the network. Specifically, these flows could have a rate increase policy which is faster than TCP and some flows could have a rate decrease policy slower than that of TCP.

Given that TCP is the most widely used transport protocol, Floyd etal proposed the guidelines for managing and designing new congestion control schemes such that they were *friendly* to TCP. A flow is deemed TCP-Friendly if its sending rate does not exceed that of a conformant TCP flow in same circumstances. This TCP-Friendly definition can further be loosened to the following relationship between the sending rate, x, and loss rate, p: $x \propto \frac{1}{\sqrt{p}}$. This TCP Friendliness can also be understood as congestion response conformance, as all flows try to be conformant to TCP.

TCP-Friendliness is the criteria not only for safeguarding TCP flows but also for enforcing some kind of fairness in the network. (TCP-Friendliness ensures Minimum Potential Delay Fairness across the network [56].) Further, we could easily

expand TCP Friendliness definition to encompass a larger range of rate control scheme which can be done by relaxing the relationship between the sending and loss rates. However, enforcing TCP Friendliness on the network remains a challenging question. In [35] the authors argue that router-based mechanisms are needed to administer TCP-Friendliness. Other than these router based schemes, another way of enforcing TCP Friendliness could be design of end-point congestion control algorithms which are TCP-Friendly.

In [9, 99, 51, 84] authors have proposed a general class of TCP-Friendly congestion control schemes. Though these proposals are encouraging they solve only a part of the TCP-Friendliness problem because the end-users may willfully choose to ignore these TCP-Friendly guidelines. As such it becomes imperative to have router-based mechanisms for enforcing TCP-Friendliness. In [35] the authors point out using per-flow scheduling or pricing mechanisms for enforcing TCP-Friendliness. However, till date to the best of our knowledge, no such per-flow scheduling or pricing mechanisms have been proposed or deployed to achieve TCP-friendliness on the network.

1.2.3 Managing Bottleneck Queues

It is a widely held belief that buffers at bottleneck routers increase the network throughput and utilization. However, this is not entirely true, especially when the network operates with bursty and responsive congestion control schemes. The purpose of a bottleneck buffer is to absorb transient increases in input traffic or in other words bursts. A large buffer, delays congestion indication and falsely forces congestion control schemes to believe that network is not bottlenecked. As a result, the congestion control schemes keep increasing their windows. However, since the bottleneck buffers are finite, after certain size they overflow thus causing huge reductions in congestion windows. This forces big oscillations in congestion window which makes the transport protocol unsuitable for a variety of applications - especially the ones which have strict timing requirements like multi-media services. Sometimes these buffer overflows also cause multiple packets of a particular flow to be dropped thus forcing it into timeouts. Therefore, a delay in reporting in the congestion state

on account of large bottleneck queues can have harmful effects on network and more importantly end-to-end performance.

The discussion in the previous section has consistently illustrated that the current Internet operates with Drop Tail queues. Also, there have been many studies which show that TCP's performance degrades on a network of Drop Tail queues. This is because Drop Tail queues often maintain very large queues and this problem in networking is commonly referred to as *full queues*. In other words, Drop Tail queue operate will near full queues which results in burst losses, frequent timeouts and synchronization of congestion windows. Moreover, the synchronization of congestion windows (and by implication losses) causes the network to oscillate between very high and low network utilization. As a result, either the bottleneck queue is full or nearly empty. These queue fluctuations induce window oscillations and increase the delay jitter making TCP and Drop Tail queues unsuitable for a variety of applications.

Removing the problem of full-queues, which exists with Drop Tail queues, has been one of the main motivations for the AQM design. RED tries to proactively manage queues by dropping (or marking) a packet before the queue actually overflows. This proactive management of bottleneck queues helps RED avoid the problem of full queues and results in significant improvements in both network and end-to-end performances. However, as we mentioned above RED suffers from many configuration problems. One of the biggest drawbacks of RED is that when it is not properly configured it's performance becomes even worse than Drop Tail queues. This is because, under some circumstances, RED consistently operates with full-queues thus bringing back all the ills associated with full queues.

Newer AQM proposals have realized the significant disadvantages of operating will full queues and as a result focus on managing near zero steady state queues. The most popular proposals in this direction are Random Early Marking (REM) [7] and Adaptive Virtual Queue [57]. Both these schemes try to match the input demand to the output capacity (in this case the capacity of bottlenecked link) and in the process maintain almost no steady state queue. The key to the success of these algorithms is their radical approach to detect congestion. For example, both these

proposals rely on mis-match between input and output rate for detecting congestion as against using bottleneck queue lengths (which was used by RED and it's variants). As a result, these schemes are able to maintain near zero queues without sacrificing network utilization or throughput.

However, despite these interesting advances in AQM design and proactive management of bottleneck queues the network continues to operate with Drop Tail queues. Once again, the main reason against the deployment of AQM schemes has been the requirement of significant network upgrade. Moreover the current network design has led to the placement of very fast and expensive routers in the network core while simpler and slower routers are maintained at the network edge. As a result of this design, network providers do not want to add any new functionality to the network core. Therefore, on account of the lack of deployment of AQM schemes, the Drop Tail queues are ubiquitous in the network and the problem due to full queues persist.

1.2.4 Summary

From the above discussion it follows that congestion control functionalities such as fairness and congestion response conformance are very closely related and importantly are tightly coupled with network based architectures. Congestion response conformance guarantees a certain kind of fairness, for example TCP-Friendliness will result in a minimum potential delay fairness across the network [56]. Similarly any fairness definition can always be translated to another congestion response conformance.

Traditionally Max-Min fairness has formed the network's definition of fairness. This definition aims to provide equal allocations to different flows. However TCP allocates rate in proportion to the the round-trip times (RTT) of the flows and loss rate. This then stands in contradiction to the network's traditional fairness goals. Thus AQM and scheduling disciplines which enforce Max-Min fairness do nothing to enforce TCP-Friendliness.

Also these different schemes for providing network-wide fairness have their own drawbacks:

- We would need AQM or scheduler support throughout the network. This implies that we will have to make changes in the core.
- TCP-Friendly criteria constrains the choice of end-point congestion control schemes for users. This might also infringe with the requirements of different protocols as these needs might not always be satisfied with TCP.
- Both AQM/Schedulers and TCP-Friendliness cannot provide a broad range of fairness criteria in the network.

Management of bottleneck queues has also been a network based task and requires presence of active queue management module at the physical router. Moreover, Drop Tail queues do not manage the bottleneck queues resulting in consistent near full queues.

1.3 Contribution of this Thesis

The goal of this research is to *re-evaluate* the function placement of congestion control building blocks in the Internet. Over the years we have typically associated congestion avoidance tasks such as managing bottleneck queues, reducing congestion window synchronization, phase effects, burst losses etc with AQM design. Also, the AQM functionality scope has been broadened to include management of selfish flows in the Internet and to provide congestion response conformance (e.g. TCP Friendliness). However for a variety of reasons these AQM schemes are not deployed on the Internet.

In this thesis we show that the placement congestion avoidance tasks can be de-coupled from it's functionality. In other words, we show that we can *de-construct AQM* schemes and move all it's functionalities to end-system and network edges. This ability to *de-couple* congestion control *tasks from it's placement* gives us architectures which are easier to manage and more importantly are deployable.

This thesis proposes three end-and-edge based congestion avoidance architectures which together emulate almost all the beneficial properties of the AQM schemes. Specifically, we propose and end-system based architecture called *Randomized TCP* which disassociates the task of introducing randomization from AQM

schemes. This, then allows us to prevent synchronization of congestion windows, reduce phase effects and burst losses and improve network utilization from the end-system. We also propose an edge-based architecture called *Uncooperative Congestion Control* which can manage selfish flows and provide congestion response conformance. This framework shows that management of selfish flows need not be coupled with AQM design but can be viewed as an end-based policing question. Finally, the thesis proposes a framework called *virtual AQM* to manage bottleneck queues lengths from network edges. This functionality allows us to proactively manage bottleneck queues while letting bottleneck use simple Drop Tail queueing. In summary, this thesis shows, that on a network of Drop Tail queues, through some simple end-and-edge based architectures we can emulate almost all beneficial properties of AQM schemes.

1.3.1 Randomized TCP: An End-System Based Solution for Improving Fairness in a Network of Drop Tail Queues

In this thesis we look at an end-based solution to some of the problems of TCP and Drop Tail queues. Specifically, we propose to introduce randomization into the network by randomizing the sending times of packets in TCP and other similar window based transport protocols. For TCP we call this solution, *Randomized TCP*. In Randomized TCP, instead of sending back to back packets, the packet sending times are randomized. In particular, successive packets of a window are sent after an interval of $RTT(1 + x)/cwnd$, where $cwnd$ is the congestion window in packets and x is a random number drawn from an Uniform distribution on *[-1,1]*. This solution is distributed, can be implemented at the end systems and thus is very attractive from an implementation perspective.

Our results show that Randomized TCP reduces phase effects and synchronization. We also analytically show that Randomized TCP reduces the synchronization of flows which then results in overall performance improvements. Randomized TCP also substantially reduces burst losses and removes the bias against longer RTT flows. In addition, the benefits of randomization can be reaped even when it is partially deployed. Randomized TCP performs better than or as well as TCP Reno,

independent of the capacity and buffer size at the bottleneck and for both short and long flows. The performance improvements can be seen in throughput, fairness, loss rates, timeouts and latency of the flows. In summary *our proposal can emulate the beneficial effects of RED in a distributed manner* without the complexities and unfavorable aspects of parameter tuning of RED. However, unlike RED, Randomized TCP does not proactively manage queues or in other words do congestion avoidance. As a result it only emulates the beneficial properties of RED which accrue out of introduction of randomization at a bottleneck in the network. Finally, through Randomized TCP we show that we can *de-couple* the task of introducing randomization at the bottleneck to reduce synchronization, phase effects etc from AQM design and instead use end-to-end congestion control schemes.

Randomized TCP is an end-based solution for improving fairness in the network and reducing the phenomena which limit the effectiveness of TCP when operating with Drop Tail queues. However, it requires a presence of Randomized flow at every bottleneck to break the synchronization at that router and improve network performance. Since Randomized TCP is end-system based solution it might not proliferate the network well enough to improve network performance. Moreover end-based systems do not protect flows and neither perform congestion avoidance. To remedy these problems we would need to monitor flows inside the network and by implication need AQM. In the next section we outline a network based solution in order to improve network performance in presence of selfish flows.

1.3.2 Uncooperative Congestion Control: Edge-Based Re-marking for Providing Fairness and Congestion Response Conformance in the Internet

One of the aims of this research is to look at effects of selfishness of users in a network; specifically, to study in what ways and to what extent a selfish user can *deliberately* degrade the performance of other users (in the network) in order to improve his performance. First, we will look at ways to define mis-behavior of users and then follow up with analyzing the effect of selfishness on equilibrium rate distribution in a network. The next objective is to suggest scalable ways to

identify the mis-behaving users in the network. We will also evaluate the stateless architectures as a means of identifying and penalizing selfish users. Towards this end we suggest an optimization model for managing the selfish behavior in the Internet. Ultimately, the aim of this thesis is to come up with a *deployable* architecture which will enable network providers to restrict and manage the selfish behavior in their network.

From the earlier discussion it follows that end-system congestion control schemes are not sufficient to provide fairness and congestion response conformance in the network and need some form of network support. However, AQM schemes can not provide a broad range of fairness objective in the network and by implication cannot enforce congestion response conformance. This is because most of them are oblivious to the competing flows and thus impose same penalties on all sources. We illustrate this with the help of some simulation scenarios. Using these results we first classify flows according to their response to congestion indication. From this classification we then define conformance and selfish behavior. We then explore the selfish behavior of protocols, specifically we derive the conditions under which new selfish protocols can be obtained while keeping the network stable. Through these definitions of selfishness and conformance we show that rate allocations in the network can be unfair and more importantly do not always comply with TCP-Friendliness or any other congestion response conformance objective. This unfair sharing of the bottleneck is then our motivating factor for studying ways to achieve conformance and fairness in the network.

In this thesis we look at "fairness" from the network's perspective (rather than the end user's perspective) and focus on managing the distribution of rate allocations. We achieve this by transparently *managing the effective range* of user's utility functions. More specifically, users may choose arbitrary utility functions, but the edge of the network can *re-map* these utility functions into a target range of utility functions. Interestingly this re-mapping is a simple consequence of the duality framework of Low et. al. [62] and can be easily implemented at the edge of the network. Internal routers of the network function as usual, i.e. they may mark, or drop packets using any AQM scheme (including drop-tail policy). Broadly this

thesis also suggests that management of mis-behaving or non-conformant flows need *not be coupled* with AQM design, and can be simply viewed as an edge network based policing question. Our mechanisms may also be thought of as a new class of "traffic conditioning" techniques [12], where the "conditioning" is achieved by manipulation of the feedback stream rather than manipulation of the packet stream.

The re-marking framework presented in this thesis can also be extended to provide service differentiation. Rather than mapping utility function the utility function of the sources to a single objective utility function, we could instead map it to a range of target utility functions and thereby differentiate between flows. This solution is attractive because it can be achieved irrespective of the congestion control scheme employed by the user and works independently of AQM scheme deployed on the network. Moreover, it can work with Drop-Tail queues also.

1.3.3 virtual AQM (vAQM): An End-or-Edge Based Solution to Manage Bottleneck Queues

In this thesis we present an abstract framework for managing bottleneck queues from the network edge or end system. We conjecture that for any flow, through end-to-end probes, we can identify the capacity of the congested link and use it to control the rate of the flow. Further, we can group flows according to the path they take in the network, find the congested link on that path and run an AQM at the network edge (ingress) to limit the rate of these flows. Moreover, any AQM schemes can be run at the edge to limit the rate of the flows (to the corresponding bottleneck). In this thesis we refer to this framework as *virtual AQM* (*vAQM*) and in this thesis we outline a specific algorithm, *virtual AVQ* (*vAVQ*), which uses AVQ like properties to limit the rate of the flows at the network edge. The main advantage of this model is that the underlying network can still use Drop Tail queuing while allowing us to manage queues from network edges.

We define a stream as a group of flows where every flow (in the group) has the same ingress and egress router and has the same route (between the ingress and egress routers). Further, the route between the ingress and egress router is called a path. For each path we define the minimum link capacity as path capacity, C, and

the maximum demand (on the path) as path demand, D. End-to-end packet probing methods derived from packet pair and packet-train models are used to estimate the path capacity and available (or effective) capacity. The path demand can then be calculated as the difference between the path capacity and the effective capacity.

For a network we also define a desired utilization factor called γ, such that $0 < \gamma < 1$. The *vAVQ* algorithm uses this utilization factor to construct a virtual path capacity which is defined as γC. The aim of the *vAVQ* scheme is then to match the estimated demand to this virtual path capacity. The *vAVQ* algorithm maintains a shadow (or virtual) buffer similar to AVQ. For every incoming packet, the buffer length counter for the shadow buffer is increased and whenever the virtual buffer overflows a packet is marked in the actual router queue. Also, the virtual buffer is drained in a process similar to that of AVQ using the path capacity and the path demand estimates. Since, the network utilization factor, γ, is less than 1 at steady state the total input (on the path) will always be less than the bottleneck link capacity leading to near zero queues.

We have evaluated the *vAVQ* framework for both single and multi-bottleneck scenarios. Our initial results suggest that the proposed framework can significantly reduce bottleneck queue lengths without compromising on link utilization or fairness. However, the model presented in this thesis is sensitive to errors in estimation and the size of the virtual buffer. These errors becomes especially important in a multi-bottleneck scenario. Moreover, in a multi-bottleneck setup, the path capacity and demand estimates may not correspond to the same physical bottleneck. As a result, the gains with *vAVQ* in a multi-bottleneck scenarios is less than that in a single bottleneck setup. However, we believe that this is an area of research which has not been explored before and needs to be investigated further as it could lead to novel, interesting and deployable queue management algorithms.

1.4 Organization of the Thesis

This thesis looks at the question of improving fairness and congestion response conformance in the network through use of end-system and network based algorithms. In Chapter 2 we review the existing work on congestion control and

AQM. Chapter 3 first outlines the problems with Drop Tail queues and TCP and then presents an end-system based algorithm, called Randomized TCP, for emulating AQM behavior on a network of Drop Tail Gateways. In Chapters 4 and 5 we study the impact of selfish behavior on the network. We first define selfish behavior in Chapter 4 and outline ways in which selfish rate control schemes can be generated. Then, we use these selfish schemes use to show that Randomized TCP and other end system based schemes are insufficient to provide a fair service to all users in the network. In Chapter 5 we present an edge system based re-marking framework for managing selfish flows in the network and providing congestion response conformance. Chapter 6 introduces *virtual AQM*, an abstract framework for managing bottleneck queues from the network edge. Finally in Chapter 7 we present the conclusions and the future work.

CHAPTER 2
Background

In this chapter we review the need of congestion control in the Internet and its objectives. Thereupon we will discuss the end-system based proposals for congestion control (e.g. TCP and its variants) as well as network based proposals i.e. AQM. This discussion on congestion control brings us to the question of how resources are shared between users, i.e. the problem of fairness. For these purposes we first review the definitions of fairness and then the various schemes for achieving fairness in the network. Finally we discuss the question of protocol compliance and flow control optimization framework.

The rest of the chapter is organized as follows:

- We begin with a review of end point schemes in Section 2.1 for congestion avoidance and control. In particular we review sliding window based protocols like TCP and it's variants in Section 2.1.1 and it's performance on a network of Drop Tail queues in Section 2.1.2, rate based proposals in Section 2.1.3 and finally we discuss Paced TCP a flow control proposals which is hybrid of rate and window based schemes in Section 2.1.4.

- In Section 2.2 we survey the network based mechanisms for preventing congestion collapse.

- Section 2.3 reviews the distribution of rates in a network or in other words, *fairness*. Section 2.3.1 and 2.3.2 discuss the oblivious and non-oblivious network based proposals for achieving fair rate distribution in the network, respectively.

- Congestion Response Conformance and mechanisms for achieving it are discussed in Section 2.4 and 2.4.2 respectively.

- Finally in Section 2.5 we review the optimization framework proposed for flow control. Also within the context of flow optimization we also review the definition of fairness and protocol compliance.

2.1 End System Based Mechanisms for Preventing Congestion Collapse

In 1980s lack of attention to the dynamics of packet forwarding on the Internet resulted in severe service degradation or *congestion collapse*. Since this congestion collapse considerable research has been done on Internet dynamics and many solutions have been suggested to avoid it. The original fix for congestion collapse was provided by Van Jacobson in 1988 as some modifications to TCP [49]. Ever since, TCP has been the backbone of the modern Internet.

2.1.1 TCP and its Variants: Congestion Avoidance and Control

TCP is a sliding window based transport protocol where the window is increased upon successful reception of acknowledgments (Ack). This ensures that TCP gradually probes and takes all the available bandwidth. However, this probing will result in a situation where the sender's sending rate exceeds the network capacity and at that point the network will drop the excess packets. These packet losses are construed as sign of congestion by the TCP and it reacts to it by cutting its rate (or decreasing window).

In TCP, the sender maintains a congestion window, *cwnd* which represents the number of packets outstanding in the network, i.e. packets which have not been acknowledged. Upon setup of a connection the *cwnd* is set to 1 and TCP sends out one packet. Subsequently on receipt of every acknowledgments TCP sends an extra packet into the network. This window increase phase is called Slow Start and is characterized by the exponential increase in window size. However this window increase will soon exhaust the network's capacity and excess packet(s) will be dropped. When TCP detects this packet loss (in Slow Start) it is construed as the end of the Slow Start and the TCP re-transmits the lost packets and halves its congestion window. Thereafter it enters the congestion avoidance phase where the window increase is much slower (as compared to Slow Start). In congestion avoidance phase TCP puts an extra packet only when a window worth of packets have been acknowledged. Thus during congestion avoidance the window increase is linear.

Jacobson proposed that in the event of loss, a timer at TCP sender will timeout while waiting for the Ack. On the expiry of this timer TCP will retransmit the lost packet and halves the congestion window, *cwnd* and stores it in a variable called *ssthresh*. TCP then resets its *cwnd* to 2 and does a Slow Start till its *cwnd* is equal to *ssthresh*, thereupon it enters the Congestion Avoidance phase.

This reaction of TCP to congestion was found to be severe in most cases and as such there were proposals to remedy these significant window cuts. Fast Retransmit and Recovery (FRR) was proposed as a means to eliminate timeouts for retransmitting a packet [92]. In FRR, when the receiver gets out of order packets it sends out duplicate Acks for the first in sequence packet. When the sender receives three duplicate Acks, it detects that a packet is lost and it retransmits the first in sequence packet. It also sets the *ssthresh* to half of the *cwnd* value and then resets the *cwnd* to *ssthresh + 3*. There after for every duplicate Ack it receives it increments the congestion window by 1. Also, if the number of outstanding packets in the network is less than the congestion window, the sender sends new packets in to the network. Finally when the sender receives the Ack for the retransmitted packet it resets it *cwnd* to *ssthresh* and continues in the Congestion Avoidance phase. Thus FRR prevents TCP into timing out for every lost packet.

There have been other proposals to optimize the TCP and the most notable amongst those has been TCP SACK. Selective Acknowledgment (SACK) is a strategy wherein the receiver can inform the sender about all segments that have arrived successfully, so the sender needs to retransmit only the segments that have actually been lost.

Another proposal which merits mention is TCP Vegas [15]. Unlike other TCP proposals which use packet loss or marking as a congestion notification TCP Vegas uses queueing delays to decipher congestion. TCP Vegas relies on the fact that during congestion the queues will build up at the bottlenecks and as such the queueing delay will increase. This increase in queueing delay is construed as a sign of congestion and TCP Vegas decreases its window by one packet, otherwise it increases its window linearly. Since the window increase and decrease in TCP Vegas is small it is most likely to converge to the optimal bandwidth.

2.1.2 TCP and Drop Tail Queues

TCP and other similar congestion control schemes result in bursty traffic. This burstiness can be attributed to three main reasons. One, when the window is increased the last two packets are sent back to back. Two, in presence of congestion on the reverse path the Acks arrive back to back and as such the packets sent are back to back. And finally, when an Ack for a retransmitted packet arrives it might result in release of a previously stalled window which might in turn lead to back to back transmissions. In order to improve the performance of the network buffers are provided at the links to absorb these bursts.

The traditional technique for managing buffers at routers has been to set a maximum length (in terms of packets) for the buffer, accept packets for the buffer until the maximum length is reached, then reject (drop) subsequent incoming packets until the queue decreases (because a packet from the queue has been transmitted). This technique is known as "Drop Tail", since the packet that arrived most recently (i.e., the one on the tail of the queue) is dropped when the buffer is full.

However this simplistic buffer management has many problems which limit the effectiveness of end-to-end congestion control algorithms. Drop Tail queueing in some situations allows a single connection or a few flows to monopolize queue space, preventing other connections from getting room in the queue. This "lock-out" phenomenon is often the result of synchronization or other timing effects [39, 14].

The Drop Tail discipline allows queues to maintain a full (or, almost full) status for long periods of time, since Drop Tail signals congestion (via a packet drop) only when the queue has become full. It is important to reduce the steady-state queue size, and this is perhaps queue management's most important goal. However, this does not take into account the critical role that packet bursts play in Internet performance. If the queue is full or almost full, an arriving burst will cause multiple packets to be dropped. This can result in a global synchronization of flows throttling back, followed by a sustained period of lowered link utilization, reducing overall throughput [39]. Further these burst losses and phase effects also cause a bias against longer round trip time flows [39, 37].

Phase effects refer to conditions where in the bandwidth-delay product of the

path of a flow is not an integral multiple of the packet size [37]. Phase effects cause a specific section of competing flows to experience recurrent drops causing unfair distribution of bandwidth and increased latency. Phase effects are manifested in the network preferentially dropping packets from a specific subset of flows thereby reducing their throughput.

Another drawback of Drop Tail queues is that they don't protect flows, i.e. it allows for a section of flows to monopolize the entire bandwidth [14]. This is especially important in the current Internet where the end system has a flexibility of choosing its congestion control scheme. This then raises the question of unfair sharing of the bottlenecks as the aggressive flows might corner a larger share of the bandwidth.

2.1.3 Rate Based Proposals for End-System Based Congestion Control

Traditionally flow control algorithm were envisioned on sliding window protocol. However these sliding window protocols resulted in bursty traffic which brought with it host of other problems. Also there were lot of algorithms for which TCP's rate cut was considered too drastic and they needed smoother rate control protocol. For these purposes rate based flow control protocol were proposed [48, 82, 40, 83]. In this section we will review some of these proposals.

Jacobs [48] presents a scheme that uses the congestion control mechanisms of TCP, however, without retransmitting lost packets. In his scheme, the sender maintains a transmission window that is advanced based on the acknowledgments of the receiver, which are sent for each received packet. The sender then uses the window to calculate the appropriate transmission rate. Rejaie et al. present in [82] an adaptation scheme called Rate Adaptation Protocol (RAP). Just as with TCP, every packet sent is acknowledged by the receivers and these acknowledgment the sender estimates the round trip delay. If no losses are detected, the sender periodically increase its transmission rate additively as a function of the estimated round trip delay. Upon detection of a loss the rate is reduced by half in a similar manner to TCP. However, this approach as well as the one presented in [48] do not consider the cases of severe losses that might lead to long recovery periods for TCP

connections. Hence, the fairness of such an approach is not always guaranteed.

In [73] the authors proposed a analytical model for calculating the average goodput of a TCP connection. Using this model Padhye et al. [40] present a scheme in which the sender estimates the round trip delay and losses based on the receiver's acknowledgments. In the case of losses, the sender restricts its transmission rate to the equivalent TCP rate calculated using TCP's throughput formula proposed in [73] otherwise the transmission rate is doubled. While the scheme behaves in a TCP-friendly manner during loss phases, its increase behavior during underload situations is rather arbitrary. Specifically, it might result in severe unfairness as the adapting end system might increase its transmission rate much faster than a competing TCP connection.

TCP Emulation At Receivers (TEAR) [83] is a combination of window and rate based congestion control. It features a TCP-like window emulation algorithm at the receivers, but the window is not used to directly control transmission. Instead, the average window size is calculated and transformed into a smoothed sending rate, which is used by the sender to space out data packets.

2.1.4 TCP Pacing: Solution For Reducing Burstiness of TCP

Sliding window based protocols like TCP often send packets in burst. As such the performance of sliding window protocols suffers on a network of Drop Tail queues. On the contrary, rate based schemes send out packets at regular intervals thus avoiding burst transmissions. However, since rate based schemes loosely observe the packet conservation principle they at times can be less responsive to network congestion. TCP Pacing [101] is a hybrid approach between window based schemes and rate based schemes. In pacing, packets to be sent in a window are spaced by $\Delta = RTT/cwnd$. This spacing of packets avoids back to back transmissions and hence removes the burstiness of TCP.

Pacing was first suggested in [101] as a correction for the compression of acks due to cross traffic. Since then the concept of pacing has been applied to slow-start, after a packet loss and after an idling time in case of web traffic [8, 75, 43, 6, 96]. In order to speed up web connections the authors in [75] suggest using pacing during the

slow start as means for Ack clocking. Similar results have been reported in [6] where the authors show that performance of slow start can be improved by use of pacing. Pacing has also been suggested for improving TCP performance with asymmetry [8] and on high bandwidth delay product links [78]. In [78] the authors evaluate pacing over the entire lifetime of TCP as a means for reducing queueing bottlenecks in wireless, high bandwidth delay networks. In [43] the authors have proposed a fast web protocol, WebTP which uses pacing during congestion avoidance phase.

Rate Based Pacing has been suggested in [96] to improve startup after idling. Slow-start restart occurs when bursty data is periodically sent over a TCP connection. TCP depends on ACK clocking for flow control. Idle periods in the connection cause this clocking mechanism to break down. In [96] the authors propose a rate-based pacing (RBP), an intermediate approach to data transmission after an idle period. RBP paces outgoing packets at a certain rate until the ACK clock is restarted. Thus RBP attempts to provide a compromise between the extremes of sending back-to-back bursts and restarting with slow start.

In [4] the authors have done an exhaustive study of pacing with different operating characteristics. They show that with long flows pacing removes synchronization, improves fairness over TCP Reno and achieves the same throughput as TCP Reno. Through simulations they show that Pacing gets synchronized during the slow start, but in the congestion avoidance phase it has a de-synchronizing effects leading to slightly higher throughput. Even in presence of flows (sufficiently long) with different round-trip times pacing was shown to increase fairness with the same throughput as of TCP Reno. However, in presence of short flows the authors show that Pacing gets synchronized causing larger latencies. They contend that by evenly spacing the packets, pacing delays the congestion point, thus allowing the sources to ramp up rates, and finally on onset of congestion causing synchronized drops. This results in lower throughput and higher latencies. Also, the authors show that when Paced TCP is competing against TCP Reno it gets beaten down.

A modified version of pacing is also evaluated in [52]. In [52] the spacing interval is defined as $\frac{RTT}{cwnd+V}$, where V is the tunable parameter, which controls the aggressiveness of the pacing. However, the effect of this scheme on the synchro-

nization of flows, phase-effects, bias against long RTT flows etc is not investigated. They observe that with bulk data transfer the modified pacing shows results similar to TCP Reno. However, for a web-like load model , the modified paced TCP exhibits lower packet loss than TCP and also the average transfer latencies are lower. However the proposal [52] do not discuss the parameter setting for V and it's effect on the pacing scheme. Also, they do not consider the case where TCP Reno and Paced TCP are multiplexed on the same link.

2.2 Network Based Mechanisms for Congestion Avoidance

From the discussion in previous section it is clear that the TCP congestion avoidance mechanisms, while necessary and powerful, are not sufficient to provide good service in all circumstances. However, there is a limit to how much control can be accomplished from the edges of the network. Some mechanisms are needed in the routers to complement the endpoint congestion avoidance mechanisms.

Active Queue Management (AQM) was proposed to complement the end-system based congestion avoidance mechanism. The AQM proposal involved a proactive management of the bottleneck queue. Specifically it was proposed that by dropping some packets before the queue gets full is an early enough indication for the sources of impending congestion. As such these sources will react to these packet losses by cutting down their rates and as a result the queue build up at the routers won't be large. This in turn implied that burst losses and synchronization of loss events could be avoided.

By keeping the average queue size small AQM will provide greater capacity to absorb naturally occurring bursts without dropping packets. Also the small queue size reduces the delays seen by flows. This is particularly important for interactive applications such as short Web transfers, Telnet traffic, or interactive audio-video sessions. AQM can also prevent synchronization of loss events by ensuring that there will almost always be a buffer available for an incoming packet. For the same reason, active queue management can prevent a router bias against low bandwidth but highly bursty flows.

Random Early Drop (RED) [39] was the first significant AQM proposal which

advocated enqueing packets probabilistically. RED achieves this by comparing the time averaged value of queue length to a threshold and then probabilistically deciding whether to enque the packet or not. Further the probability of enqueing a packet decreases as the average queue length increases. Thus, if the queue has been mostly empty in the recent past, RED won't tend to drop packets. On the other hand, if the queue has recently been relatively full, indicating persistent congestion, newly arriving packets are more likely to be dropped.

RED operates with 5 control parameters and they are: the two thresholds, the minimum threshold min_{th} and the maximum threshold max_{th}, the queue averaging parameter w_q, the length of the buffer, B and the maximum dropping probability, max_p. When the average queue is in between the two thresholds packets are enqueued probabilistically, where the dropping probability increases linearly as a function of max_p and the average queue length (the dropping probability is 0 at min_{th} and max_p at max_{th}). However if the average queue crosses the maximum threshold all incoming packets are dropped.

Though RED solves many problems of drop tail queues it is not without its share of problems. The biggest concern with RED is it's configuration. RED has 5 operational control parameters and there are no fixed guidelines for tuning them. Further it has been shown that if RED is not properly configured can result in performance degradation, so much so that it is even worse than Drop Tail queues [67, 22]. Though recently there have been some proposals for configuring RED they are limited by the ability to configuring only a set of these control parameters [20, 46]. Thus in absence of strict guidelines RED has not found much favor with network operators.

Besides the problem of configurations, RED also does not protect flows [60]. When hit with a mixture of responsive and unresponsive sources, RED allows unfair bandwidth sharing. This is because RED enforces equal loss rates on each flow, irrespective of their bandwidth. As such if there are flows which do not respond to congestion then they will eventually corner bandwidth worth their sending rates and in the process beat down TCP or other responsive flows. A similar situation can be expected if there are flows who are more aggressive than TCP but are responsive.

Given these deficiencies with RED there have been several other AQM proposals which attempt to solve some of these problems [60, 57, 7, 28, 91, 90, 29]. The notable mentions amongst these schemes are Adaptive Virtual Queue (AVQ) [57] and Random Exponential Marking (REM) [7]. AVQ uses a virtual queue to enque packets in the network and the size of the virtual queue depends on the arrival rate of the traffic and the utilization desired at the bottleneck. If the virtual queue is full then the packet is not enqued. REM on the other hand tries to match the input rate to the bottleneck capacity. Though AVQ and REM have fewer parameters to configure and give crisper guidelines for deriving those parameters these schemes still do not protect flows. This is because these proposals do not differentiate between flows. However there are some schemes which take into account flow arrival rate to allocate marks (or losses) and thus protect flows [60, 71, 30, 64]. We will discuss these schemes in detail in the following section.

2.3 Fairness

Though there are many definitions of fairness but its meaning in all definitions is that it represents the distribution of rates between users. Fairness is one of the most important considerations before network providers. This is because it represents how a network distributes rates between users such that the network does not penalize any user and more importantly can also use it to provide service differentiation.

Kelly et al. in [53] showed that every rate control scheme is associated with a particular kind of fairness. Specifically they show that if all users use same congestion control scheme then the subsequent rate distribution in the network is associated with a unique fairness criteria. In Kelly's framework every rate control algorithm is associated with a *Utility function*, *U(x)*, which is a function of its rate allocation, x on the network. The end user's objective is to maximize its utility function with respect to rate. Kelly, Mo and Walrand showed in their work that the equilibrium distribution of rates or fairness with a utility function, U(x) is given as

$$\sum_i p_i U_i'(x_i - x_i^*)$$

where x_i is the rate allocated to the user i and x_i^* is the fair rate for the user i [53, 69]. The most interesting fact about this formulation is that it can used to represent any fairness criteria. For example *proportional fairness*, where the rates are allocated in proportion to the network resources being used in given by

$$\sum_i \frac{x_i - x_i^*}{x_i}.$$

Recent works in flow optimization have used this definition to relate different rate control schemes to corresponding fairness criteria. In [61, 56, 13] the authors show that TCP Reno is associated with *minimum potential delay* fairness, i.e. it tries to minimize the total delay in a file transfer. Similarly, TCP Vegas achieves proportional fairness [61]. Mo and Walrand have also proposed a range of congestion control algorithm which can achieve weighted proportional fairness [69].

Though a network can allocate bandwidth according to a range of fairness criteria, traditionally equal allocations and Max-Min fairness have formed network's criteria for providing fair service to all users. However both these criteria are significantly different from the inherent fairness provided by TCP, i.e. minimum potential delay fairness. Therefore any fairness objective in the network, other than minimum potential delay fairness, might penalize some TCP flows. As such, any fairness objective in the network also has to take into account that it does not penalize TCP flows in the network, especially when most of the network traffic is carried by TCP.

The fairness issue also assumes importance because of proliferation of different rate control schemes in the network. This is because there could exists schemes which do not react to congestion indication or their response is different from TCP's response. As such, in [35, 65, 14] the authors argue that these schemes pose twin problems of being unfair to TCP's flows and importantly congestion collapse.

Different network based mechanisms have been proposed to manage these selfish schemes to prevent congestion collapse and to provide fair service to TCP flows. These schemes can be broadly classified into two categories, oblivious and non-oblivious schemes. Oblivious schemes allocate equal marks (or loss rate) to all flows and therefore do not protect TCP flows from other selfish and non-responsive flows.

RED, REM, AVQ, CHOKe [57, 7, 29, 28] are some examples of such schemes. However CHOKe [77] is one notable exception amongst these oblivious schemes as it tries to punish aggressive flows. Non-Oblivious schemes on the other hand differentiate between flows mostly on their arrival rate and thus attempt to protect flows. Flow RED (FRED) [60], Stabilized RED (SRED) [71], Stochastic Fair BLUE (SFB) [30] , RED with Proportional Dropping (RED-PD) [64] are examples of AQM schemes which look at some flow characteristic before deciding to enque them. In this section we discuss some of these proposals.

2.3.1 Oblivious Schemes for Providing Fairness in the Network

End based techniques are insufficient to protect flows in the network and thereby provide fairness. Towards achieving these objective use of AQM at routers was proposed. RED [39] was the first significant AQM proposal. However as discussed in Section 2.2 RED cannot protect flows, especially when TCP flows in the cases where they compete with unresponsive flows [60]. Moreover since RED's control parameter are statically configured, i.e. the configuration does not change with time, RED's penalty function can be severe under low loads and insufficient with large multiplexing of flows [29, 28]. This further constrains the fairness objectives which RED can achieve.

Taking into the account these configuration issues Feng et al. proposed an AQM scheme, Adaptive RED (or ARED) [31]. ARED presents an on-line algorithm for dynamically changing the values of $\max_p$ or the maximum dropping probability, according to the observed traffic. Therefore depending on whether the queue has been full or empty the maximum dropping value is increased or decreased. BLUE is another fundamentally different AQM algorithm which uses packet loss and link idle events to manage congestion [28]. BLUE maintains a single probability, which it uses to mark (or drop) packets when they are required. If the queue is continually dropping packets due to buffer overflow, BLUE increments the marking probability, thus increasing the rate at which it sends at which it sends back congestion notification. Conversely, if the queue becomes empty or if the link is idle, BLUE decreases its marking probability.

Though both ARED and BLUE offer dynamic configuration of RED, they still do not protect TCP flows from misbehaving users. This is because these flows consider aggregate arrival rate to the bottleneck and thus do not differentiate between flows. Also these schemes allocate equal marks to all flows and thus misbehaving blows still corner a large share of the bottleneck. This is one of the drawbacks of the oblivious schemes.

However there is one significantly different oblivious AQM proposal which does protect flows from misbehaving users. This proposal called, CHOKe [77] takes into account the number of packets queued for a flow before deciding to enque them. When a packet arrives at the bottleneck, CHOKe randomly picks a packets already enqueued in the buffer and compares the flow identifier for both these packets. If a match is found then both the packets are dropped otherwise the incoming packet in enqued. This rule of deciding to enque a packet thus punishes aggressive flows as they are more likely to have more packets enqued and thus more probable to be dropped. The authors show that CHOKe tries to achieve Max-Min fair distribution across the network [77].

In summary barring CHOKe all oblivious schemes cannot protect TCP from misbehaving flows. However, all oblivious schemes are limited by the range of fairness criteria they can provide.

2.3.2 Non-Oblivious Schemes for Providing Fairness in the Network

From discussion in the previous section it is clear that in order to protect flows from misbehaving users we will need to assign marks not only on the basis of aggregate arrival rate to the bottleneck queue but also on individual flow arrival rates. Thus if we are monitoring individual flow rates to assign marks (or drops), the subsequent schemes are called *non-oblivious* schemes. Different ways have been suggested for monitoring individual flow's share in the bottleneck. One of these is explicit rate monitoring at every bottleneck, another method involves monitoring at one bottleneck and then sending the rate information through some means (either in packet header or through specific control packets) or deciphering the rate through number of packets enqueued in the bottleneck queue. In this section we will dis-

cuss some non-oblivious network based schemes which use one of these methods for providing fairness in the network.

The first significant non-oblivious AQM proposal was Flow RED (or FRED) [60]. FRED provides better protection than RED for responsive flows by isolating non-responsive greedy flows more effectively. Instead of indicating congestion to randomly chosen connections by dropping packets proportionally, FRED generates selective feedback to a filtered set of connections which have large number of packets queued. To achieve this, FRED estimates average per-flow buffer count, *avgcq*; flows with fewer than avgcq packets buffered are favored. Also FRED maintains a count, *strike* of number of times a flow has failed to respond to congestion notification. Any flow which has a higher strike value is more likely to be dropped. However one of the main drawbacks of FRED is that it has to maintain per-flow states, i.e. states for responsive as well as non-responsive flows, and might also increase average transfer delays.

A differential dropping scheme to manage fair bandwidth allocation at the router in presence of malicious users is presented in [76]. The scheme presented is similar to FRED in the sense that it maintains information about flows to decide which packet to drop. The authors propose the use of a shadow buffer where the count of packets is stored. A packet is dropped (or marked) if the packet count for the flow in the shadow buffer exceeds its fair rate (in terms of packets).

Probabilistic Aggregate Marking (or PAM) uses RED type thresholding on the token bucket contents to mark a packet (from a traffic aggregate) [23]. If the token bucket contents fall below min_{th} packets are marked with a lower priority. If the token bucket contents are in between min_{th} and max_{th} then packets are marked according to a linear function. The authors argue that PAM offer proportional marking though this argument is not backed by any analysis.

The authors in [23] also propose a scheme similar to CSFQ (Core Stateless Fair Queuing) and call it Stateless Aggregate Fair Marker (or SAFM). The edge marks packets based on the information present in the header. In SAFM the CSFQ header contents are replaced by token bucket size, token bucket rate and (1 - token allocation probability). The ingress calculates these values while the egress uses

them to construct the fair rate share vector and then using it to mark packets. The model involves per-flow calculations at the ingress router.

In [63] the authors present a model to control high bandwidth aggregates in the network which uses packet history to drop/mark aggressive flows. Also once these misbehaving sources have been identified and punished this information can be pushed back to the downstream routers which can further rate limit these flows.

Another per-flow differential dropping called RED-PD (RED- Preferential Dropping) is proposed in [64]. Therein a malicious user is detected using drop history, which implies maintaining state. However the authors claim that this state is minimal because they store information only about the aggressive flows. Nevertheless the scheme requires storing states at all the routers in the network. The authors propose the use of congestion epochs to detect aggressive users. The idea is that a flow ideally sees one drop in one congestion epoch and this information can be leveraged to detect malicious flows. A per-flow filter is applied to these malicious flows which either drops a packet probabilistically or admits the packet. The probabilistic dropping is based on ARED.

In [44] the authors present a strategy to detect unresponsive flows using Diff-Serv. They propose shaping the unresponsive flows at the edges using congestion information from the core. As such the core is required to maintain information about every dropped packet and sends this information periodically to edge routers. The scheme proposed by the authors cannot be deployed in the Internet as it requires modifying the core and also requires to maintain state inside the core which raises questions of scalability.[74, 24, 60].

Yet another network based scheme to achieve fairness in the network is Core Stateless Fair Queuing or CSFQ [95, 93]. In CSFQ a flow's arrival rate is monitored at the network edges and stored in the packet header for use by other routers. Thereafter each router updates this flow arrival rate and uses it to enque the packet probabilistically. The authors contend that CSFQ achieves fair queueing in the network.

Besides these AQM based proposals another popular way of providing fairness has been the use of schedulers in the network [25, 10, 88]. Fair queueing, Weighted

FQ, Deficit Round Robin (DRR) are some scheduling proposals which can be used to provide fair distribution of rates in the network. However, these schemes are constrained by the need of placement throughout the network and moreover need coordination between each scheduler.

2.4 Congestion Response Conformance in the Network

Over the years the Internet growth has been well supplemented by various applications who have varying needs, especially for transport protocols. Initially all these applications relied on TCP but as the requirements of the applications changed TCP was no longer the only favored transport protocols and therefore a variety of flow control algorithms were proposed. These flow control algorithms can be mainly divided into three main classes a) TCP-Compatible flows b) unresponsive flows, i.e., flows that do not slow down when congestion occurs, and (c) flows that are responsive but are not TCP-compatible [34]. Yet another class of flow control algorithm use a mix of responsiveness and unresponsiveness. Specifically these algorithms decrease their rate on receiving a congestion indication but they also have a lower limit on transmission rate, i.e. they do not react to congestion indications when the sending rate is below this limit.

As a result of this proliferation of different congestion control algorithms we may reach a stage where there is no congestion avoidance mechanisms in the network. This would bring us back to the congestion collapse problem of 1980s [34]. These different class of flow control algorithm either responsive or unresponsive also pose a problem of protocol compliance. Floyd et. al formally defined this problem of protocol conformance in [35] wherein a conformant flow was called TCP-Friendly or TCP-Compatible.

RFC 2309 defines TCP-compatibility as, "A TCP-compatible flow is responsive to congestion notification, and in steady-state uses no more bandwidth than a conformant TCP running under comparable conditions (drop rate, RTT, MTU, etc.) " [14]. Floyd et al. in [35] proposed mechanisms for protocol conformance. These mechanisms can be broadly classified into two categories a) End-System and b) Network Based mechanisms. They proposed guidelines for developing end-system

based flow control protocols which have the same reaction to loss as TCP. Since then there have been many proposals for TCP- Friendly flow control algorithms [9, 99, 51, 40, 84]. However, as we have previously seen there is a limit to how much control can be exerted from just the use of end-system. As such Floyd et al. also proposed network based mechanisms in form of schedulers and pricing mechanisms to ensure TCP-Friendliness on the network. We will now discuss these proposals briefly.

2.4.1 End-System Based Schemes for Congestion Response Conformance in the Network

Though conformance and fairness to TCP is significant it however should not constrain the choices of end-to-end congestion control algorithms. In [9] the authors propose a class of non-linear TCP compatible congestion control schemes called Binomial Congestion Control Schemes (BCCS). AIMD, can be considered as one of congestion control schemes in the subset of TCP Compatible BCCS. Formally, the Binomial Congestion Control scheme can be defined as:

$$W_{t+R} \leftarrow W_t + \alpha/W_t^k \quad if\ no\ loss \tag{2.1}$$

$$W_{t+\delta t} \leftarrow W_t - \beta W_t^l \quad if\ loss \tag{2.2}$$

where k and l are window scaling factors for increase and decrease respectively and α and β are increase the decrease proportionality constants. For any given values of α and β TCP Compatible BCCS can be defined by $k+l = 1 : k \geq 0,\ l \geq 0$.

Another interesting set of TCP Compatible congestion control algorithms has been presented in [84]. The proposal called Choose Your Response Function (or CYRF) has a general increase function f and a decrease function g which together constitute the congestion control policy. Formally the TCP-Compatibility is defined by the following constraints on the these two function $f,\ g$ as:

$$f(x)g(x) \propto x$$

There have also been other interesting proposals for TCP Compatible win-

dow based protocols [51, 99] but covering all of them is beyond the scope of the thesis. Besides these window based proposals there have been suggestions for TCP compatible rate control scheme. The most popular rate based scheme is called TCP-Friendly Rate Control (or TFRC) [40]. Since we have already discussed TFRC in Section 2.1.3 we do not elaborate on it any further.

Finally this section can be summarized as, "The concern expressed in [RFC2357] about fairness with TCP places a significant though not crippling constraint on the range of viable end-to-end congestion control mechanisms for best-effort traffic." [34].

2.4.2 Network Based Schemes for Congestion Response Conformance in the Network

Though there exists a range of end-system based TCP Compatible congestion control scheme they might still not meet the needs of various applications. Moreover there exists a possibility that end users may intentionally not use these algorithms. Therefore network based solutions are needed to enforce protocol compliance.

The network based support has been envisioned in two primary forms: a) schedulers and b) pricing mechanisms [35, 34]. Per-flow scheduling in form of Class Based Queueing, Priority Scheduling or Weighted Round Robin etc can be used to isolate flows, restrict bandwidth of misbehaving flows and thus provide TCP Compatibility. Similarly pricing mechanisms can also be used for differentiating against misbehaving flows by communicating them higher price and thus ensuring TCP Compatibility in the network. However in order to achieve TCP Compatibility for the current Internet environment where flows compete in a FIFO queue all these mechanisms require tight coordination between all routers.

2.5 Optimization: Flow Control, Fairness and TCP Compatibility

Recently congestion control schemes have been evaluated and proposed using optimization frameworks [53, 56, 62, 69, 16]. In these papers, the resource allocation problem is proposed as 1) individual users maximizing their utility functions

and 2) network maximizing every user's utility function given the network capacity constraints.

In these optimization models a user s, is described with the help of it's rate, x_s, a utility function U_s and the set of links which he uses, *L(s)*. It is further assumed that the rates bounded i.e., $m_s \leq x_s \leq M_s$. It is assumed that the utility functions are *increasing* with rates and *strictly concave*. The network is identified with links l of capacity C_l. The set of users using a link, l, is given by *S(l)*.

The optimization problem can then be defined as user's trying to maximize their individual utility functions and the network trying to maximize the resource allocation subject to link capacity constraints. Thus the primal problem can be defined as:

$$maximize \sum_{s \in S} U_s(x_s) \tag{2.3}$$

$$subject\ to \sum_{s \in S(l)} x_s \leq C_l, \ \forall l \tag{2.4}$$

for all $x_s \geq 0$. The dual formulation, D(p), for the above problem was defined by Low in [62] as:

$$D(p) = \underbrace{min}_{p \geq 0} \sum_{s \in S} (U_s(x_s) - \sum_l p_l x_s) + \sum_l p_l C_l \tag{2.5}$$

The authors in [62] show that using the Karush Kuhn Tucker (KKT) conditions and gradient projection algorithm the dual yields the following update algorithm

$$x_s(t) = U_s'^{-1}(\sum_l p_l) \tag{2.6}$$

$$p_l(t+1) = [p_l(t) + \gamma(\sum_{s \in S(l)} x_s - C_l)]^+ \tag{2.7}$$

Since the primal is strictly concave and the constraints are linear, there is no duality gap and hence dual optimal is also primal optimal. Further the strict concavity entails an unique global optimum, (x_s^*, p^{s*}) where $p^s = \sum_l p_l$. Also though the primal optimal, x_s^* is unique, we may not have a unique dual optimal p_l^* but instead we have a unique optimum end-to-end loss probability for every source, p^{s*}.

In [53] the authors analyze the stability and fairness of network under primal and dual formulation. The dual formulation has also been discussed in [62] where gradient projection method is used to solve the problem. A penalty function approach to solving the network problem has been suggested by the authors in [56]. Also, the current TCP implementations have been mapped to optimized rate control algorithm in [61] [56].

The user's rate control algorithm can be thought to be tightly coupled with a utility function. Also, if all the utility functions are of similar type then we can also associate a fairness criteria with it. Fairness is defined as the way the resources are distributed amongst competing users, eg. max-min fairness, where the goal is to maximize the minimum share. The fairness criteria for a set of users S, can be described in terms of utility function as follows:

$$\sum_{S} U'(x_i - x_i^*) \quad < \quad 0 \tag{2.8}$$

where x_i is the rate allocated to the user i and x_i^* is the fair rate for the user i. The max-min fair vector corresponds to $U(x) = lim_{\alpha \to \infty} \frac{-1}{x^\alpha}$. If the rate allocations are in "proportion" to the resources used by a user, then such a rate is said to be proportional fair and is defined by $U(x) = log(x)$.

Thus the equilibrium rate allocation is very closely tied with the utility function the user chooses to maximize. *This association of equilibrium rate allocation with the utility function might prompt sources to choose a utility function (and hence an aggressive congestion control scheme) which yields them higher rate allocations than other competing sources.* Such a choice of utility function will still optimize the network and keep it stable, though at the cost of unfair allocations amongst users.

Finally, in [5] the authors evaluate the existence and properties of Nash Equilibria for selfish TCP user. They define selfishness by allowing the user to choose (and modify) it's own increase and decrease parameters, α and β respectively. They pose the problem as game where all users try to maximize their goodputs and evaluate the Equilibria for TCP Reno, Tahoe and Sack with both Drop-Tail and RED queues. They show that efficient Nash Equilibria exists only for TCP-Reno and

Drop-Tail queues and the equilibrium can be defined by either $\alpha = 1$ or any β ¿ 0. However the equilibrium for TCP Sack and Tahoe is defined by an arbitrarily large value of α and $\beta \rightarrow 1$. Also the authors show that when these TCP flavors are evaluated with RED gateways the Nash equilibrium is inefficient. A similar result is also reported in [27]. The authors evaluate the Nash Equilibria for stateless AQM schemes and show that RED and Drop Tail do not impose Nash Equilibria on selfish users.

2.6 Managing Bottleneck Queues in the Network

The network traffic is bursty by nature. As a result, all routers in the Internet are configured with buffers to absorb packet bursts. However, configuring the buffer space has been an active area of research. It is a widely held belief that buffer can increase the network utilization or throughput. However, this is not always true and a large buffer can have adverse effect on network and end-to-end performance. A large buffer will cause consistent queueing in the network even though the link might be congested. This consistent queueing delays congestion signal causing all flows to *falsely* believe that the network is not congested. As a result, they keep increasing their rates and by implication congestion window. However, since bottleneck routers are finite, after certain size they overflow thus causing huge reductions in congestion windows. This forces big oscillations in congestion window which makes the transport protocol unsuitable for a variety of applications - especially the ones which have strict timing requirements like multi-media services. Sometimes these buffer overflows also cause multiple packets of a particular flow to be dropped thus forcing it into timeouts. Therefore, a delay in reporting in the congestion state on account of large bottleneck queues can have harmful effects on network and more importantly end-to-end performance.

The simplest buffer management scheme is Drop Tail, i.e. enque packets till there is space in the buffer and drop them when there is no space. This queue management policy is commonly referred to as passive queue management. Drop Tail queues often operate with near full queues which causes burst losses, timeouts and synchronization. Further, the synchronization of windows (and by implication

losses) causes sustained period of very high and low link utilization, reducing overall throughput [39]. Moreover, full queues can also be associated with the problem of phase effects and bias against flows with long propagation delay [37, 39]. However, despite of these issues what works in the favor of Drop Tail queues is it's simplicity. Drop Tail queues have no parameter to configure and therefore it's performance does not depend on the bottleneck link capacity or the number of flows. Because of their simplicity Drop Tail queues are very popular with network providers. In fact, the current Internet operates with only Drop Tail queues ! In summary, Drop Tail queues are passive or in other words do not manage bottleneck queues and this inability to manage queues often results in degradation of not only network but also end-to-end performance.

Active Queue Management (or AQM) was suggested as an alternative to Drop Tail or passive queueing [39]. Floyd et. al. have argued that actively managing queues can remove almost all deficiencies of Drop Tail queues. Random Early Drop was the first and the most popular AQM proposal [39]. Floyd et al. showed that by dropping packets before the queue over flows (or active queue management) sends an early congestion indication to some flows causing them to reduce rate. This rate reduction in turn reduces (or removes) impending congestion. Further, this active queue management ensures that there is always space in the queue to accommodate incoming bursts. As such, active queue management schemes remove almost all deficiencies of Drop Tail queueing.

Many AQM schemes based on RED were proposed [29, 28, 31, 64, 77, 60, 71]. All these schemes used some form of queue thresholds to decide when to start probabilistically dropping packets. However, all these schemes suffer from implementational complexities. All these variants of RED, including RED, have a large number of configurable parameter which usually depends on variety of factors including bottleneck link capacity, number of flows and buffer size. Recent studies have shown that if these AQM proposals are not configured properly or if some network operating conditions change then the worst case performance of these AQM schemes can be even worse than that of Drop Tail queues [67, 22]. This is because, these AQM schemes stop managing queues and instead operate with full queues.

However, newer AQM proposals have realized the pitfalls of using queue thresholds to actively manage queues [57, 7, 41]. These proposals, derived from network optimization framework, attempt to match the total input rate to some fraction of the bottleneck link capacity. This mis-match between the input and output rate is used to generate congestion indications. (This is in direct contrast with RED and it's variants which use some form of queue thresholding to generate congestion indications and thus manage queues.) The most popular schemes in which use rate mis-match to actively manage queues are Random Early Marking (REM) [7] and Adaptive Virtual Queue (AVQ) [57]. REM is derived from the duality model of network optimization (see equation 2.5) and tries to stabilize queue at a pre-specified desired value. AVQ on the other hand uses notions of virtual link and buffers to match the input and output rates. Specifically, it constructs a virtual link which has a capacity slightly less than that of actual bottlenecked link. For every incoming packet it updates both the virtual link capacity and virtual buffer and marks (or drops) packet in the actual queue whenever the virtual buffer overflows. Thus, AVQ tries to match the input rate to virtual link capacity. Since the virtual link capacity is less than the bottlenecked capacity, in the steady state the bottleneck queue size is almost zero.

It is interesting to note that all active queue management outlined above are network based, i.e. each bottleneck link has a module which actively manages the bottleneck queues. Thus, all AQM proposals require extensive deployment in the network. Moreover, all these AQM proposals have some configuration parameters which depend on link capacity, buffer size and network operating conditions like number of flows. Since, in most cases the guidelines to configure these parameters are not crisply defined and that network operating conditions might change network providers have not shown much enthusiasm to deploy these AQM proposals. As such, as of today, the network still operates with Drop Tail queues or in other words no queue management modules are deployed in the present Internet.

CHAPTER 3
Randomized TCP: End System Based Mechanism for Improving Fairness in a Network of Drop Tail Queues [1]

3.1 Introduction

As discussed in Chapter 2, Drop Tail queues substantially limit the effectiveness of end-to-end congestion control protocols. This is primarily due to failure to provide early congestion notification to the end users. To avoid this and for better queue management use of AQM has been suggested. However due to configuration problems these AQMs have not found their way to the Internet, which to this day operates with Drop Tail queues. As such the problems of congestion window and loss event synchronization, phase effects and bias against bursty and long RTT flows persist. In this chapter we look at a comprehensive solution to all these issues by randomizing the packet transmission times in TCP flows.

The rest of the chapter is organized as follows.

- We present an end system based scheme to introduce randomization in the network and thereby emulate AQM. The proposal is called Randomized TCP and is discussed in Section 3.2 and the algorithm is detailed in Section 3.3.
- In Section 3.4 we do a characterization of the increase parameter to enable Randomized TCP to compete fairly with TCP Reno. A queueing analysis is presented in Section 3.6 to show that the probability of burst losses decreases with Randomized TCP.
- We present the implementation of Randomized, simulation setup and define the performance metrics in Section 3.8.
- Parameter tuning Randomized TCP is presented in Section 3.10.
- In Section 3.11 we present the simulation results for comparative performance of TCP Reno, Paced TCP and Randomized TCP while Section 3.12 evaluates

[1] This work was done jointly with Prof. Biplab Sikdar

the bias against longer RTT flows, phase effects, synchronization and burst losses for Randomized TCP and TCP Reno.

- We extend the randomization of sending times to other window based schemes, specifically Binomial schemes and present its results in Section 3.13.

- Finally we present the conclusions in Section 3.14.

3.2 Randomized TCP

From the discussion in the Chapter 2 it is clear that introducing randomization into the network can break synchronization. Also by introducing the randomization, we avoid burst losses, thereby making the loss events "distributed". This then helps in solving the problem of phase effects. Though AQMs can introduce randomization in networks to some extent, it is not widely deployed due to variety of reasons [66, 67]. As such, we propose an end-system based mechanism for emulating AQM behavior. Specifically we propose a modification to TCP, called *Randomized TCP*, as a mechanism for introducing randomization into the network by randomizing the packet sending times. This solution is distributed, can be implemented at the end systems and therefore is very attractive from an implementation perspective.

Randomized TCP is similar to Paced TCP in that it "paces" packet transmissions but instead of spacing the transmissions equally, it adds or subtracts a random interval to the packet sending times at TCP sources. Packet transmissions are scheduled at intervals of $\frac{RTT}{cwnd}(1+x)$, where x follows the Uniform Distribution on *[-I, I]*. Evidently, I has to be between 0 and 1. A packet is transmitted at the expiry of the timer, if the window allows a packet to be sent. If not, upon reception of an ack, we schedule the packet transmission with a random delay of $\frac{RTT}{cwnd}y$, where y is *U(0,I)*. Setting I to 0 reduces Randomized TCP to Paced TCP. The Randomized TCP's sending time algorithm is stated in Section 3.3.

In Section 3.10, we investigate the optimal setting of the randomization interval and find that a Uniform distribution on *[-1,1]* is the best. This choice of Uniform distribution can be intuitively justified as; *a)* since the distribution is centered around *0*, on an average there is "no randomization" and Randomized TCP

behaves as Paced TCP, *b)* and a minimum value of *-1* of randomization implies TCP Reno implementation. This implies that sometimes we send back-to-back packets and sometimes we send paced packets. Thus with a randomization interval value of 1, Randomized TCP keeps moving forth between TCP Reno and TCP Paced. Intuitively, this entails an early detection of congestion (when the TCP behaves as Reno) and an even distribution of losses and throughput (when TCP behaves as Paced). Thus Randomized TCP takes the best of both Reno and Paced TCP and ensures lesser drops (because of early congestion detection) and fairer throughput.

In Paced TCP packets from each source reach the bottleneck at an uniform rate which can lead to near perfect interleaving. Such situations can cause all sources to lose packets thereby resulting in all the sources decreasing their windows together, resulting in synchronization. But with randomization, the rate is not uniform at the bottleneck and packets from flows are dropped after differing times due to the extra delay incurred due to randomization. This means that sources decrease their windows at different times and hence the periods of increase and decrease are not as synchronized as in Paced TCP. So the congestion epochs for different flows get out of sync and the network utilization is higher. Another nice property that comes because of randomization is that the source which has lost packets once is less likely to lose again (this may not be the case with deterministic TCP for some parameter settings [38]), thereby ensuring that over a larger time scale the rate distribution is fair.

Randomizing the sending times also results in extra delays causing the RTT to increase artificially. This causes Randomized TCP to get beaten down when competing with TCP Reno. It is well known that TCP's throughput is directly proportional to the square root of the window increase parameter and inversely proportional to RTT [73]. To allow Randomized TCP to compete fairly with TCP Reno, we analytically characterize the increased RTT (in Section 3.4) and make the increase factor in TCP proportional to the square of the ratio of the changed RTT to the real RTT.

We also note that the probability of two packets coming nearly back to back is significant only when the window size is large. This means that the probability of

multiple packet drops will be very low if the window size is small, thereby reducing timeouts. Using a simple M/M/1/K queueing analysis, similar to that in [67], in Section 3.6 we try to get a quantitative feel of the probability of a packet getting dropped with Randomized TCP.

The increased randomization increases the entropy of the system which correspondingly reduces the queue sizes thereby improving the stability of the system [79]. Our results show that Randomized TCP reduces phase effects and synchronization even when multiplexed with TCP Reno flows. Also it substantially reduces burst losses and removes the bias against longer RTT flows. In addition, *the benefits of randomization can be reaped even when it is partially deployed*. Randomized TCP performs better than or as well as Paced TCP and TCP Reno, independent of the capacity and buffer size at the bottleneck and for both short and long flows. The performance improvements can be seen in throughput, fairness, loss rates, timeouts and latency of the flows. We also investigate the impact of randomization on a class of slowly varying congestion control schemes called Binomial schemes [9] and show that by incorporating randomization in these schemes, the fairness increases dramatically when competing with TCP flows in drop tail queues.

In other words *our scheme can emulate the beneficial effects of RED in a distributed manner* without the complexities and unfavorable aspects of parameter tuning of RED. However, we wish to emphasize that unlike RED which is a congestion avoidance scheme, Randomized TCP is just a congestion control scheme. Thus Randomized TCP does not emulate the congestion avoidance features of RED, at best it provides the other beneficial features of RED which were achieved by introducing randomization in the network (by dropping packets probabilistically).

3.3 Randomized TCP Pseudo-code

Define by α the original increase parameter for the TCP Reno and by R the RTT. Then the Randomized TCP's algorithm can be stated as

- Send a packet. Schedule the next packet to be sent at time $t = \frac{RTT}{cwnd}(1 + x)$ where x is Uniformly distributed on [-1,1].

- Let $t^{'}$ be the arrival time of next ack. Then
  - If $t^{'} < t$ send the next packet at t.
  - Else send the next packet after $\frac{RTT}{cwnd}y$ where y is uniformly distributed on [0,1].
- At each RTT (estimate) update recalculate the new increase parameter as $\alpha_{new} = \alpha \left(\frac{RTT_{new}}{R}\right)^2$

3.4 Analytical Characterization of Increase Parameter for Randomized TCP

In this section we outline the methodology for setting the increase parameter, α for Randomized TCP so as to make it compete fairly with TCP Reno. This is required because randomizing the sending times results in extra delay and hence slows down the window growth. As such it is likely that Randomized TCP will lose to TCP Reno when competing on a single bottleneck.

Consider a Randomized TCP connection with a constant window size of w. Let the real RTT for the connection be a constant denoted by R. Each packet is sent after a time equal to $R(1+x)/w$ where x is a Uniform random variable between $[-I, I]$ (The optimal value of this interval is shown to be 1 in section 3.10, but presently we treat it more generally). Let the first packet be sent at time $t = 0$. Then the timer for the $w+1^{th}$ packet of the connection will be scheduled at time, say t_1, such that

$$t_1 = R(1 + \frac{1}{w}\sum_{i=1}^{w} x_i) \tag{3.1}$$

where x_i is the random value for the i^{th} packet in the window. The x_is are independent and identically distributed. The effective RTT of the flow is the given by the time when $(w+1)^{th}$ packet is sent. In the absence of random variations in real RTT, the ACK for the first packet comes exactly after time R. If $\sum_{i=1}^{w} x_i \geq 0$ then $t_1 > R$ and we will send the $(w+1)^{th}$ packet at time t_1. Else, the $(w+1)^{th}$ packet will be sent after a random time $\frac{RTT}{w}y$ after the ACK arrival, where y is drawn from an uniform distribution on [0,I].

Thus the effective RTT can be expressed as

$$RTT_{eff} = \begin{cases} R(1 + \frac{1}{w}\sum_{i=1}^{w} x_i) & \text{w.p. } P\{\sum_{i=1}^{w} x_i \geq 0\} \\ R(1 + \frac{y}{w}) & \text{w.p. } P\{\sum_{i=1}^{w} x_i \leq 0\} \end{cases} \tag{3.2}$$

where w. p. is short for "with probability". Then, the mean effective RTT, $\overline{RTT}_{eff}$, can be expressed as

$$\begin{aligned} \overline{RTT}_{eff} &= \{R(1 + \frac{1}{w}E[\sum_{i=1}^{w} x_i \mid (\sum_{i=1}^{w} x_i \geq 0)])\} \\ & P\{\sum_{i=1}^{w} x_i \geq 0\} + \{R(1 + \frac{\bar{y}}{w})\}\ P\{\sum_{i=1}^{w} x_i \leq 0\} \end{aligned} \tag{3.3}$$

where $\bar{y}$ is the mean of y equal to $I/2$. Since x_i follows an Uniform distribution around zero, its easy to see that

$$P\{\sum_{i=1}^{w} x_i \geq 0\ \} = P\{\sum_{i=1}^{w} x_i \leq 0\ \} = 0.5. \tag{3.4}$$

Assuming that the window size is sufficiently large to invoke the the Central Limit Theorem we get

$$\sum_{i=1}^{w} x_i \sim N(0, \sigma^2), \quad \sigma^2 = w * \frac{I^2}{3} \tag{3.5}$$

The pdf of $\sum_{i=1}^{w} x_i$ conditioned on $\sum_{i=1}^{w} x_i \geq 0$ can be found out to be twice that of the Gaussian pdf multiplied by the Unit step function. From this we can derive the conditional mean as

$$E[\sum_{i=1}^{w} x_i \mid (\sum_{i=1}^{w} x_i \geq 0)] = \sqrt{\frac{2wI^2}{3\pi}} \tag{3.6}$$

Plugging these back into the equation for $\overline{RTT}_{eff}$, we get

$$\overline{RTT}_{eff} = R + \frac{1}{2w}(\sqrt{\frac{2wI^2}{3\pi}} + \frac{I}{2}) \tag{3.7}$$

For Randomized TCP with increase parameter α and effective mean RTT, $\overline{RTT}_{eff}$, the throughput is proportional to $\frac{\sqrt{\alpha}}{\overline{RTT}_{eff}}$. To make the throughput same as that of TCP Reno (with $\alpha = 1$ and $RTT = R$), we set $\alpha = \frac{\overline{RTT}_{eff}^2}{R^2}$ for randomized

TCP. In the real implementation, since window value changes with time, $\overline{RTT}_{eff}$ changes with time and so we change the value of α also with time.

3.5 Analytical Characterization of Reduction of Synchronization with Randomized TCP

In this section we show that synchronization is reduced with Randomized TCP. To study the synchronization of flows we use the covariance between the congestion window of two competing flows. Flows would be synchronized if their windows increase and decrease simultaneously. In this case both flows' windows (say w_1 and w_2) would be above or below their mean values at any time t, i.e. $(w_1(t)-\bar{w_1})(w_2(t)-\bar{w_2}) > 0$. So the cross-covariance coefficient of synchronized flows would be positive. In the case where the flows are totally out of sync, $(w_1(t) - \bar{w_1})(w_2(t) - \bar{w_2}) < 0$, since when one flow has a large window, the other would have a smaller window and vice versa. So the cross-covariance coefficient of out of sync flows would be negative. This shows that the cross covariance coefficient of greater than 0 implies in-phase synchronization while less than 0 implies out-of phase synchronization. However, too large a negative value of cross-covariance denotes that synchronization effects still persist albeit in a negative sense which might lead to big window oscillations. Hence a value equal to or close to 0 for cross-covariance coefficient should be the optimal.

Consider a Randomized TCP connection with a constant window size of w_1. Let the real RTT for the connection be a constant denoted by R. Each packet is sent after a time equal to $R(1 + x)/w_1$ where x is a Uniform random variable between $[-I, I]$ (The optimal value of this interval is shown to be 1, but presently we treat it more generally). Let the first packet be sent at time $t = 0$ and the window size be $w_1(t)$. Let us further assume that the packet loss probability is nearly zero and therefore all packets are acked. Thus we are considering the time when a flow is about to increase it's window. Then the timer for sending the $w_1 + 1^{th}$ packet of the connection will be scheduled at time, say $t + 1$, such that

$$t + 1 = R(1 + \frac{1}{w_1}\sum_{i=1}^{w_1} x_i) \tag{3.8}$$

where x_i is the random value for the i^{th} packet in the window. The x_is are independent and identically distributed.

If we randomize the sending time then the window will be increased at $t{+}1 > R$ provided $\sum_{i=1}^{w_1} x_i \geq 0$. Otherwise, the $(w_1+1)^{th}$ packet will be sent after a random time $\frac{RTT}{w_1}y$ after the ACK arrival, where y is drawn from an uniform distribution on [0,I].

Since x_i follows an Uniform distribution around zero, we can calculate

$$P\{\sum_{i=1}^{w_1} x_i \geq 0\ \} = P\{\sum_{i=1}^{w_1} x_i \leq 0\ \} = 0.5. \tag{3.9}$$

Assuming that the window size is sufficiently large to invoke the the Central Limit Theorem we get

$$\sum_{i=1}^{w_1} x_i \sim N(0,\sigma^2), \quad \sigma^2 = w_1 * \frac{I^2}{3} \tag{3.10}$$

The pdf of $\sum_{i=1}^{w_1} x_i$ conditioned on $\sum_{i=1}^{w_1} x_i \geq 0$ can be found out to be twice that of the Gaussian pdf multiplied by the Unit step function. From this we can derive the conditional mean as

$$E[\sum_{i=1}^{w_1} x_i \mid (\sum_{i=1}^{w_1} x_i \geq 0)] = \sqrt{\frac{2w_1 I^2}{3\pi}} \tag{3.11}$$

Plugging these back into the equation for $w_1(t+1)$, we get

$$w_1(t+1) = w_1(t) + \frac{1}{2w}(\sqrt{\frac{2wI^2}{3\pi}}) \tag{3.12}$$

Now, let there be another Randomized TCP source whose congestion window is given as $w_2(t)$ and it's mean value is given as w_1. From the above equation we can calculate the coefficient of variation between $w_1(t+1)$ and $w_2(t)$ as

$$CoV_{Random} = E[(w_1(t+1) - \overline{w})(w_2(t+1) - w_2)] \tag{3.13}$$

$$= E[w_1(t+1)w_2(t+1)] - w_1\, w_2 \tag{3.14}$$

which can then be calculated as

$$CoV_{Random} = E[w_1(t+1)w_2(t+1)] + \sqrt{\frac{I^2}{6\pi}} E[w_2(t+1)\sqrt{\frac{1}{w_1(t+1)}}] - w_1 w_2 \quad (3.15)$$

Let's assume that $w_2(t)$ and $w_1(t)$ are identical processes. Then, we may write the above equation as

$$CoV_{Random} = E[w_1(t+1)w_2(t+1)] + \sqrt{\frac{I^2}{6\pi}} E[\sqrt{w_2(t+1)}] - w_1 w_2 \quad (3.16)$$

Let there be two TCP Reno sources, $w_1^{Reno}(t+1)$ and $w_2^{Reno}(t+1)$ such that there average window values, $\overline{w_1^{Reno}}$ and $\overline{w_2^{Reno}}$ are w_1 and w_2 respectively. Further at time $t+1$ let the window of one of TCP Reno flow increases. We may then after some simplification write the coefficient of variation of the two TCP Reno flows as

$$\begin{aligned} CoV_{Reno} &= E[(w_1^{Reno}(t+1) - w_1^{Reno})(w_2^{Reno}(t+1) - w_2^{Reno}(t))] & (3.17) \\ &= E[w_1^{Reno}(t) w_2^{Reno}(t)] + E[w_2^{Reno}(t)] - w_1\, w_2 & (3.18) \end{aligned}$$

Let us assume that $E[w_1(t)w_2(t)]$ is equal to $E[w_1^{Reno}(t)w_2^{Reno}(t)]$. Then, since $w(t)$ is always greater than 1, (irrespective of whether it is TCP Reno or Randomized TCP), comparing equations (9) and (11) we can see that CoV_{Random} will be less than CoV_{Reno}.

3.6 Queueing Analysis to Show Reduction in Burst Losses with Randomized TCP

Consider a *M/M/1/K* queueing system where the packets arrive according to a batch Poisson process; specifically, bursts (or batches) of B packets arrive according to a Poisson process of rate λ. Further, let us denote by $\pi(k)$ as the stationary distribution of k number of packets in the queue. Then using the PASTA (Poisson Arrival See Time Averages) property the probability of a packet drop in a Tail Drop

router with TCP as input can be calculated as [67]:

$$P_{TD} = \pi(K) + \pi(K-1)\frac{B-1}{B} + ... + \pi(K-B+1)\frac{1}{B}$$

Using the same model we will now calculate the probability of a packet being dropped for Randomized TCP. We first note that the size of burst, B, will now be changed because Randomized TCP paces the packets. Hence we first try to find the new burst size (given that the original burst size was B) and then calculate the packet drop probability. Figure 3.1 shows the epochs at which the packets are sent. Let us call the time instants at which the packets from a Paced TCP would have been sent as *centered epoch*. These *centered epochs* now represent the time instants around which we randomize the sending times of packets in Randomized TCP. Suppose a packet is sent at some time, x after the centered epoch (as shown in figure 3.1). Let us also define the length of the packet as L bits and the bottleneck link capacity as C bits/sec. Further, let the window size at steady state be W (B $\leq$ W) and let RTT denote the round-trip time. Then the probability, p, of packets from a burst of B, arriving back-to-back at the bottleneck router can be calculated as

$$p = \int_0^{\frac{RTT}{W}} \left(\frac{1}{2}\frac{x}{\frac{RTT}{W}}\right)\left(\frac{1}{2}\frac{\frac{L}{C}}{\frac{RTT}{W}}\right) dx \tag{3.19}$$

$$= \frac{L}{8C} \tag{3.20}$$

Note that the now, $B^{'} = min(B * \frac{L}{8C}, B)$, represents the upper bound on the number of back-to-back packets which can be received at a bottleneck with Randomized TCP and a burst of size B. Also note that the above analysis holds true iff $\frac{L}{8C} \leq \frac{RTT}{W}$ which holds true for WANs and MANs.

Using the above equation, the probability that a packet gets dropped with Randomized TCP and drop tail router can be calculated as

$$P_{TDR} = \pi(K) + \pi(K-1)\frac{B^{'}-1}{B^{'}} + ... + \pi(K-B^{'}+1)\frac{1}{B^{'}} \ .$$

Thus from the above observation we can conclude that the probability that a

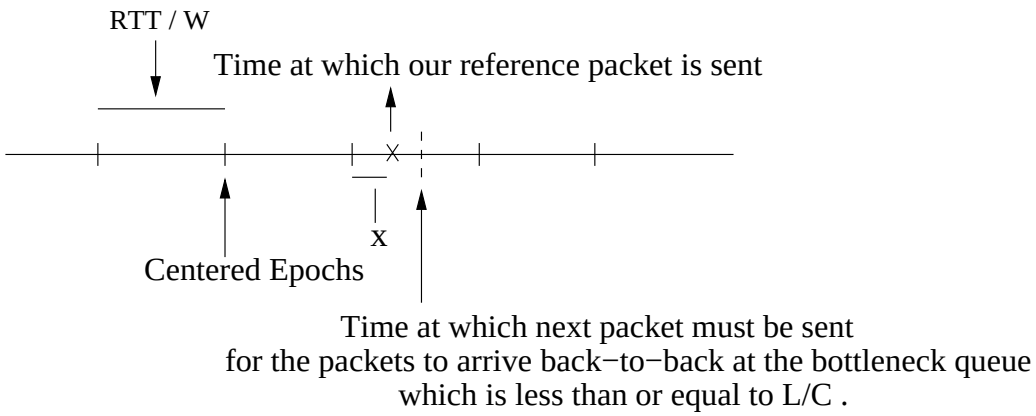


Figure 3.1: Packet Sent Times with Randomized TCP

packet gets dropped with Randomized TCP and drop tail queue decreases. However, it should be noted that Poisson arrivals do not capture the exact packet arrivals in the Internet. Nevertheless, this exercise is just intended to show that the probability of burst losses are reduced with Randomized TCP and have been validated by our simulation results in Section 3.12.4.

3.7 TCP-Friendliness of Randomized TCP

A key design objective for Randomized TCP is that it should compete fairly with TCP Reno. Previous studies have shown that Paced TCP [4] (which is Randomized TCP with $x = 0$) losses to TCP Reno. In this section we will show that our analytical characterization of Randomized TCP's increase parameter enables Randomized TCP to compete fairly with TCP Reno. Let R_R be the round-trip time of a Randomized TCP and R be the end-to-end propagation delay (which in other words is the round-trip time for the TCP Reno flow). Further let the increase parameter for Randomized TCP and TCP Reno be α_R and α respectively. Lastly, let the packet loss probability in the network be p. Under these assumptions, the throughput, x, of TCP Reno is given by [46] as

$$x = \sqrt{\frac{\alpha}{\beta p}} \frac{1}{R} \tag{3.21}$$

The window increase process for the Randomized TCP flow can be written as

$$W(t+R_R) = W(t) + \alpha_R(1-p)^{W(t)} - \beta W(t)(1-(1-p)^{W(t)}) \tag{3.22}$$

Assuming that the packet loss probability is close to 0, we may re-write the above equation as

$$W(t+R_R) = W(t) + \alpha - \beta W(t) p^{W(t)} \tag{3.23}$$

From the above equation, we may calculate the rate of change of congestion window as

$$\frac{W(t+R_R) - W(t)}{R_R} = \frac{\alpha_R}{R_R} - \beta \frac{W(t)}{R_R} pW(t) \tag{3.24}$$

or in other words

$$\frac{dW(t)}{dt} = \frac{\alpha_R}{R_R} - \beta \frac{W(t)}{R_R} pW(t) \tag{3.25}$$

Since at equilibrium the rate of increase of window will be equal to the rate of decrease of window we have $\frac{dW(t)}{dt} = 0$. We will also drop the time component from the window, W(t), and instead write it as W. Thus at equilibrium we get

$$\frac{\alpha_R}{R_R} = \beta \frac{W}{R_R} pW \tag{3.26}$$

which can be re-written as

$$p = \frac{\alpha_R}{\beta W^2} \tag{3.27}$$

We can also express the above equation in terms of the sending rate, x_R, of the source. Moreover, the sending rate is also a measure of the throughput. Thus in steady state the following equation gives the relationship between the throughput x and the end-to-end loss probability, p.

$$x_R = \sqrt{\frac{\alpha_R}{\beta p}} \frac{1}{R_R} \tag{3.28}$$

Now substituting the value of α_R which was shown to be $\alpha_R = \alpha \frac{{R_R}^2}{R}$ in Section 3.4 we get

$$x_R = \sqrt{\frac{\alpha \frac{{R_R}^2}{R}}{\beta p}} \frac{1}{R_R} \tag{3.29}$$

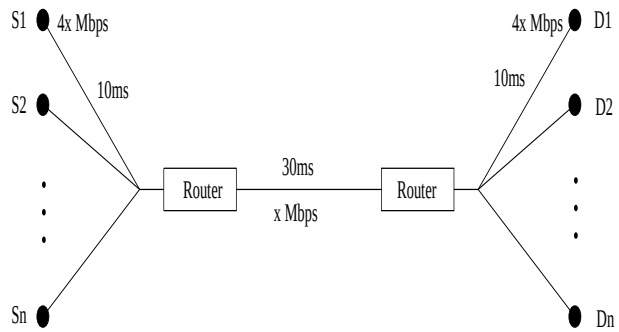


Figure 3.2: Topology used in the simulation.

or

$$x_R = \sqrt{\frac{\alpha}{\beta p}} \frac{1}{R} \tag{3.30}$$

which is the same as the throughput of TCP Reno (equation 3.21). Thus Randomized TCP is TCP-Friendly.

3.8 Implementation and Simulation Setup

We have implemented Randomized TCP in the Network Simulator *ns* [1]. For our implementation, we used the congestion control and loss recovery mechanisms of TCP Reno and thus Randomized TCP has the usual slow-start and fast recovery and retransmit mechanisms. For the simulations reported in this chapter, we disabled the delayed acknowledgments option. Also, we used the modified window increase parameter for Randomized TCP implementation.

Figure 3.2 shows the topology used in the simulations. The access links were configured at a rate 4 times greater than that of the bottleneck link and all the links use Drop Tail queues. The maximum advertised window is set sufficiently high so that it does not constrain the actual window. We use a Maximum Segment Size of 500 bytes.

We evaluate the performance of randomized TCP for the following set of metrics: average throughput, fairness, loss rates, timeouts, latency and synchronization. We characterize fairness using the modified Jain's fairness index, [21, 4]. Jain's fairness index is defined as

$$f = \frac{(\sum_{i=1}^{n} x_i.RTT_i)^2}{n(\sum_{i=1}^{n}(x_i.RTT_i)^2)} \tag{3.31}$$

where x_i is the throughput of the i^{th} flow, RTT_i is the round-trip time of flow i and n is the number of flows.

To study the synchronization of flows we use the covariance between the congestion window of two competing flows. Flows would be synchronized if their windows increase and decrease simultaneously. In this case both flows' windows (say w_1 and w_2) would be above or below their mean values at any time t, i.e. $(w_1(t) - \bar{w_1})(w_2(t) - \bar{w_2}) > 0$. So the cross-covariance coefficient of synchronized flows would be positive. In the case where the flows are totally out of sync, $(w_1(t) - \bar{w_1})(w_2(t) - \bar{w_2}) < 0$, since when one flow has a large window, the other would have a smaller window and vice versa. So the cross-covariance coefficient of out of sync flows would be negative. This shows that the cross covariance coefficient of greater than 0 implies in-phase synchronization while less than 0 implies out-of phase synchronization. However, too large a negative value of cross-covariance denotes that synchronization effects still persist albeit in a negative sense. In [101] the authors also argue that out-of-phase synchronization is not good. Hence a value equal to or close to 0 for cross-covariance coefficient should be the optimal.

In the following sections we present the simulation results. We first observe the effect of bottleneck bandwidth, buffer sizes and RTTs on the randomization interval I in section 3.10. Using these simulations we propose a value of the interval for optimal performance.

Section 3.12 shows the performance of Randomized TCP with respect to phase effects, synchronization amongst flows and burst losses. In Section 3.11 we present the result of comparative performance of Randomized, Paced and Reno TCP for the following set of metrics: throughput, losses, timeouts, fairness and latency for both bulk-data transfer and short-web like transfers. Finally in Section 3.13 we present the results of extension of Randomization to Binomial schemes.

3.9 Implementation on the Linux Kernel

We have implemented the Randomized TCP in Linux [68]. The following components were required to implement Randomized TCP 1) a microsecond resolution timer for Linux, 2) a random number generator and 3) a packet scheduling

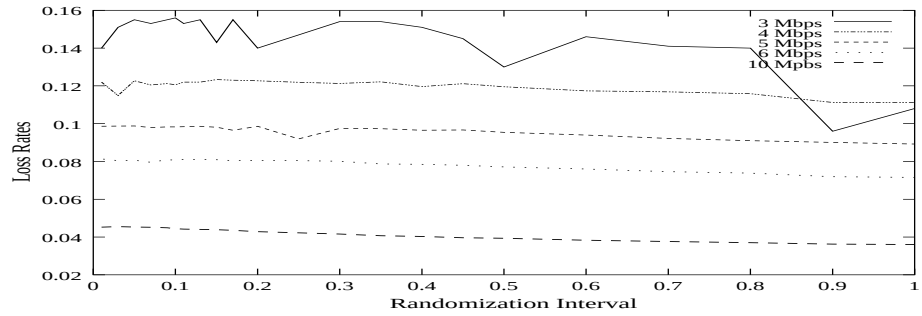
methodology to schedule packets in future. We used UTIME [2] extension to the Linux kernel to introduce microsecond resolution. Scheduling of packets is done on the expiry of this microsecond timer. The Linux kernel provides a random number generator that returns a requested number of random bytes to the module invoked within the kernel. However, our requirement needs the random number to be generated on the byte boundaries but rather or bit boundaries. We wrote functions to create such a random number and also to create both positive and negative random numbers. We tested the implementation with the simple dumb-bell topology.

3.10 Parameter Tuning

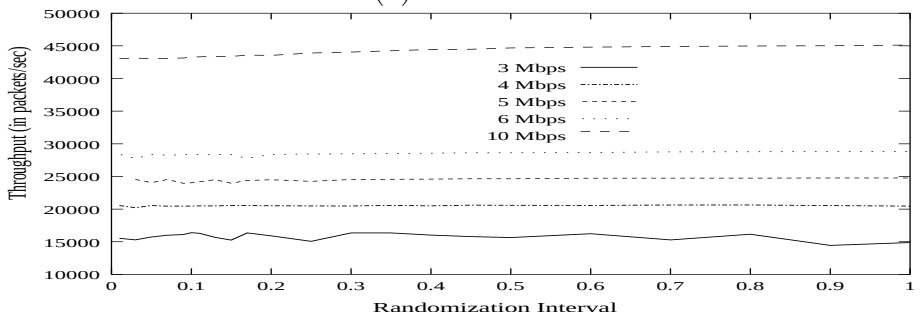
The randomization interval has a significant impact on the performance of Randomized TCP, and hence its characterization is of utmost importance. In this section we study the effect of change in bottleneck bandwidth, buffer size, number of flows and round-trip times on throughput, number of losses, timeouts as a function of the randomization interval. Through these simulations we obtain the optimal value of randomization interval. The default settings for this section are a bottleneck link of 1 Mbps, all the other links of bandwidth 4 Mbps, end-to-end propagation delay of 100ms and a Drop Tail queue of 25 packets at the bottleneck. Simulation settings are assumed to be default (as that mentioned in 3.8) unless specifically specified.

3.10.1 Different Bottleneck Bandwidth

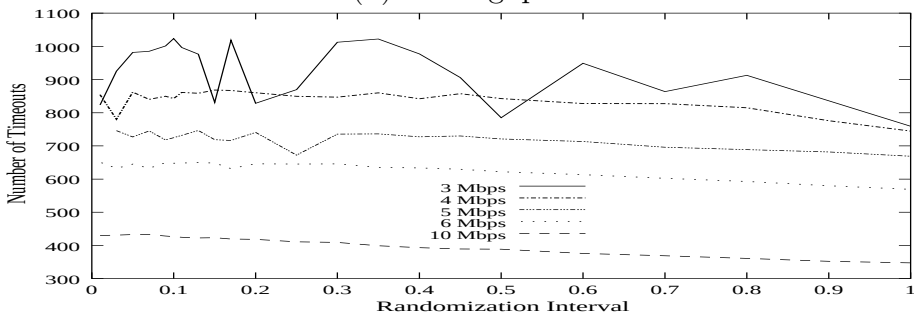
Figures 3.3 (a), (b) and (c) plot the loss rates, throughput and timeouts respectively, for a setup of 50 flows as a function of randomization interval on a single bottleneck setup (figure 3.2). The bottleneck bandwidth was varied in this case from 3Mbps to 10Mbps while the buffer size was held constant at 25 packets. The end-to-end propagation delay was 100 ms. It can be seen that as the randomization interval increases to 1, the loss rates and the timeouts reduce, while the throughput increases or remains almost the same. Similar results were obtained with a larger buffer size. The impact of buffer size on the randomization interval is detailed in the following section.



(a) Loss rate



(b) Throughput



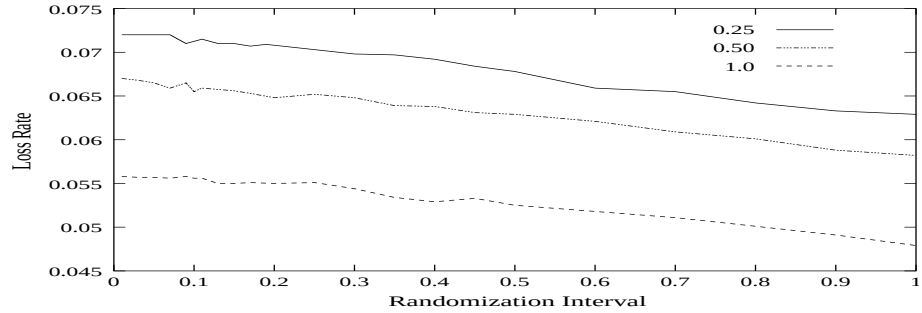
(c) Timeouts

Figure 3.3: Loss Rate, Throughput and Timeouts for 50 flows as a function of randomization interval, for different values of bottleneck bandwidth.

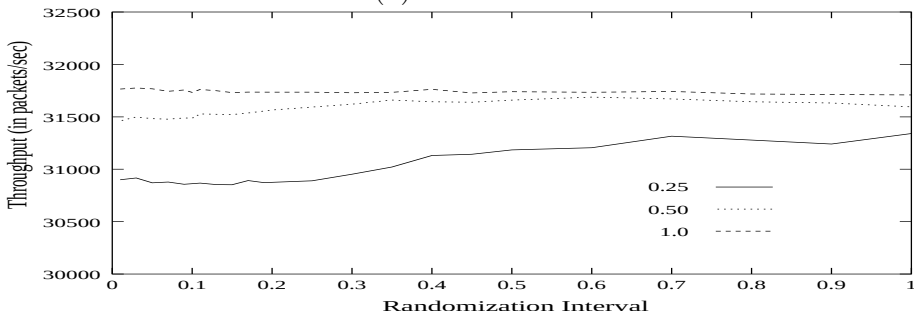
3.10.2 Different Buffer Sizes

In order to evaluate the effect of buffer sizes, we vary the buffer size at the bottleneck from one-fourth of bandwidth delay product to one bandwidth delay product. The bottleneck link is of 4 Mbps and the end-to-end propagation delay is 100ms. Thus we vary the buffer size from one-fourth bandwidth delay product to one bandwidth delay product. Again, we plot the losses, throughput and timeouts

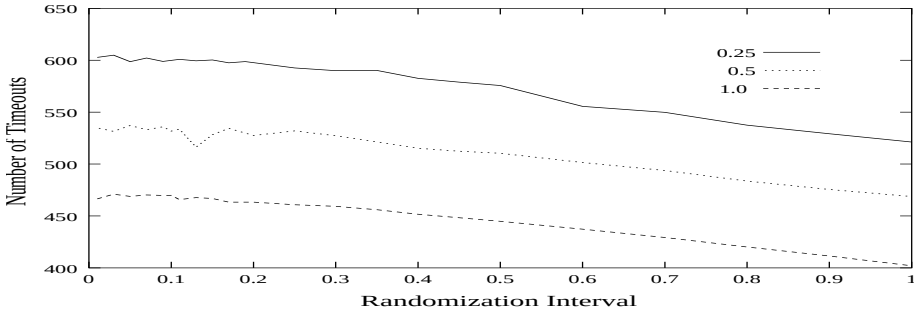
for 30 flows as a function of randomization interval. Figure 3.4 show the effect of buffer size. From the Figure 3.4 it can be inferred that a randomization interval value of 1 gives us the best results vis-a-vis throughput, loss rate and the number of timeouts.



(a) Loss rate



(b) Throughput

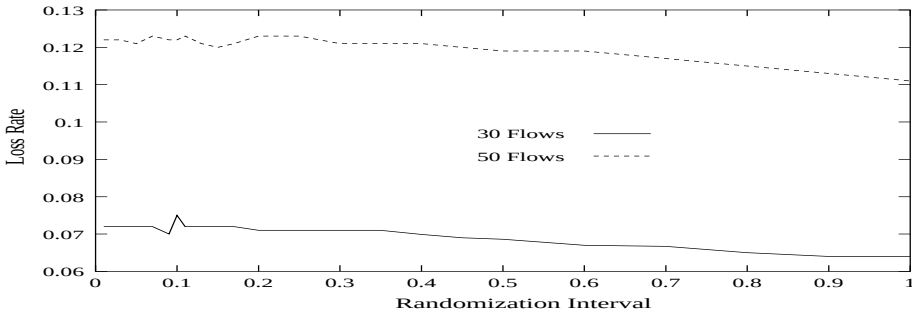


(c) Timeouts

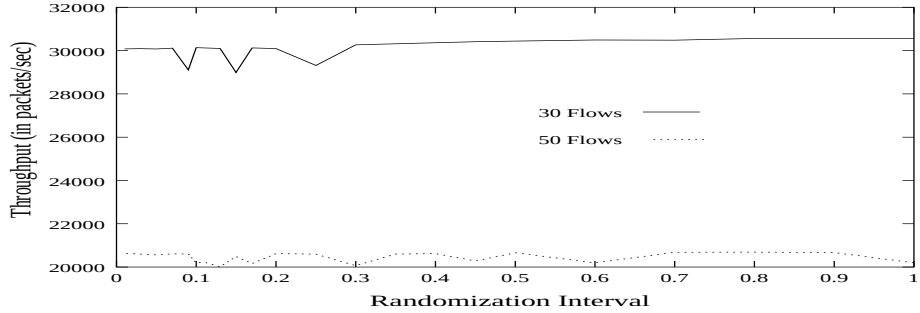
Figure 3.4: Loss Rate, Throughput and Timeouts for 30 flows as a function of randomization interval, for different values of bottleneck buffer size.

3.10.3 Different RTT

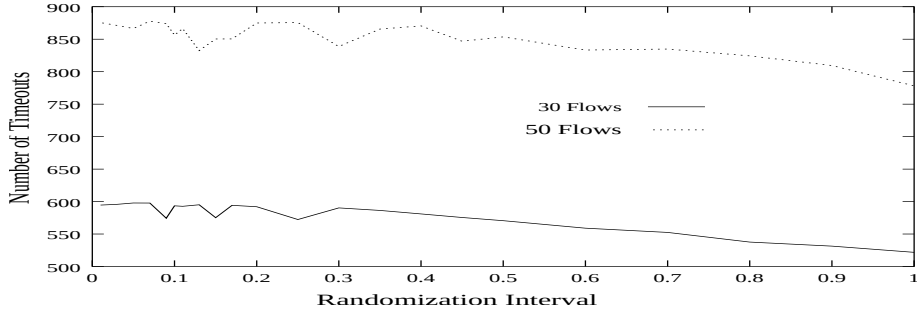
In this simulation setup every flow had a unique RTT in the range 80ms to 120ms. The RTT for the $i^{th}, i \in (0, \ldots, N-1)$ flow was $80+i*(120-80)/N$ where N is the total number of competing flows. In Figure 3.5 we plot the losses, throughput and timeouts for 30 and 50 flows as a function of randomization interval. From the Figure 3.5 we can conclude that a randomization interval value of 1 suits almost all the simulation metrics.



(a) Loss rate



(b) Throughput



(c) Timeouts

Figure 3.5: Loss Rate, Throughput and Timeouts for 30, 50 flows as a function of randomization interval, for varying RTT. The RTT varies from 80ms-120ms, the bottleneck bandwidth is 4Mbps and the buffer size is 25 packets.

From the above simulations it is evident that a higher value of randomization interval results in increased throughput and lower losses and timeouts. Randomization interval of 1 implies that inter-packet time intervals can lie anywhere between 0 and $2RTT/cwnd$. This means that packets are randomized the most and this results in increased scope for breaking the synchronization, thereby resulting in better performance.

This choice of randomization interval can be intuitively explained as following. With a randomized interval value of 1, randomized TCP keeps moving forth between TCP Reno and TCP Paced. (This is because, since randomization interval is Uniform on [-1,1] therefore when the randomized value is -1 then the packets are sent immediately after receiving an ACK akin to TCP Reno. Since the randomization interval is centered around 0, on an average Randomized TCP behaves as Paced TCP.) Intuitively, this entails an early detection of congestion (when the TCP behaves as Reno) and an even distribution of losses and throughput (when TCP behaves as Paced). Thus Randomized TCP takes the best of both Reno and Paced TCP and ensures lesser drops (because of early congestion detection) and fairer throughput.

3.11 Throughput, Loss, Timeouts, Fairness and Latency

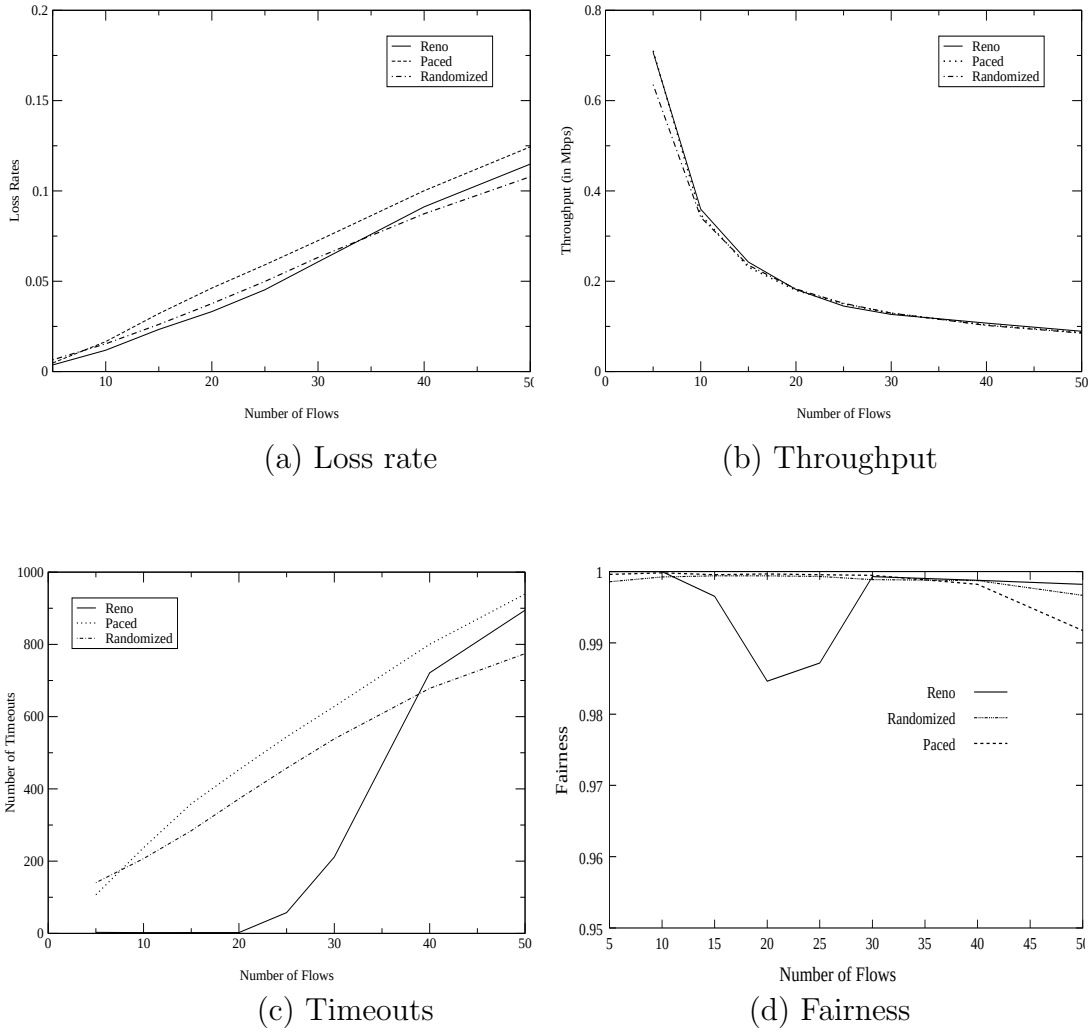
In this section we compare the performance of Randomized TCP with TCP Reno and Paced TCP. We evaluate all these three schemes for both Bulk data transfers and small Web like transfers. Specifically, we compare the following metrics: average throughput, loss rate, timeouts for bulk data transfers and latency for small web like transfers. We also assess the interaction of Randomized TCP and TCP Reno on a single bottleneck for the metrics throughput, loss rate and timeouts.

3.11.1 Bulk Data Transfer

3.11.1.1 Same RTT

Figure 3.6 plots the throughput, loss rate, number of timeouts and fairness for Reno, Paced and Randomized TCP. Though Reno, Paced and Randomized TCP have the same throughput the losses are more for Paced. This is because in slow

start, Pacing delays the congestion signal and hence looses a larger number of packets. As the number of flows increase Randomized TCP tends to do the best of the



(a) Loss rate (b) Throughput

(c) Timeouts (d) Fairness

Figure 3.6: Loss Rate, Throughput, Timeouts and fairness with Bulk Data transfer , each flow having same RTT.

3.11.1.2 Different RTT

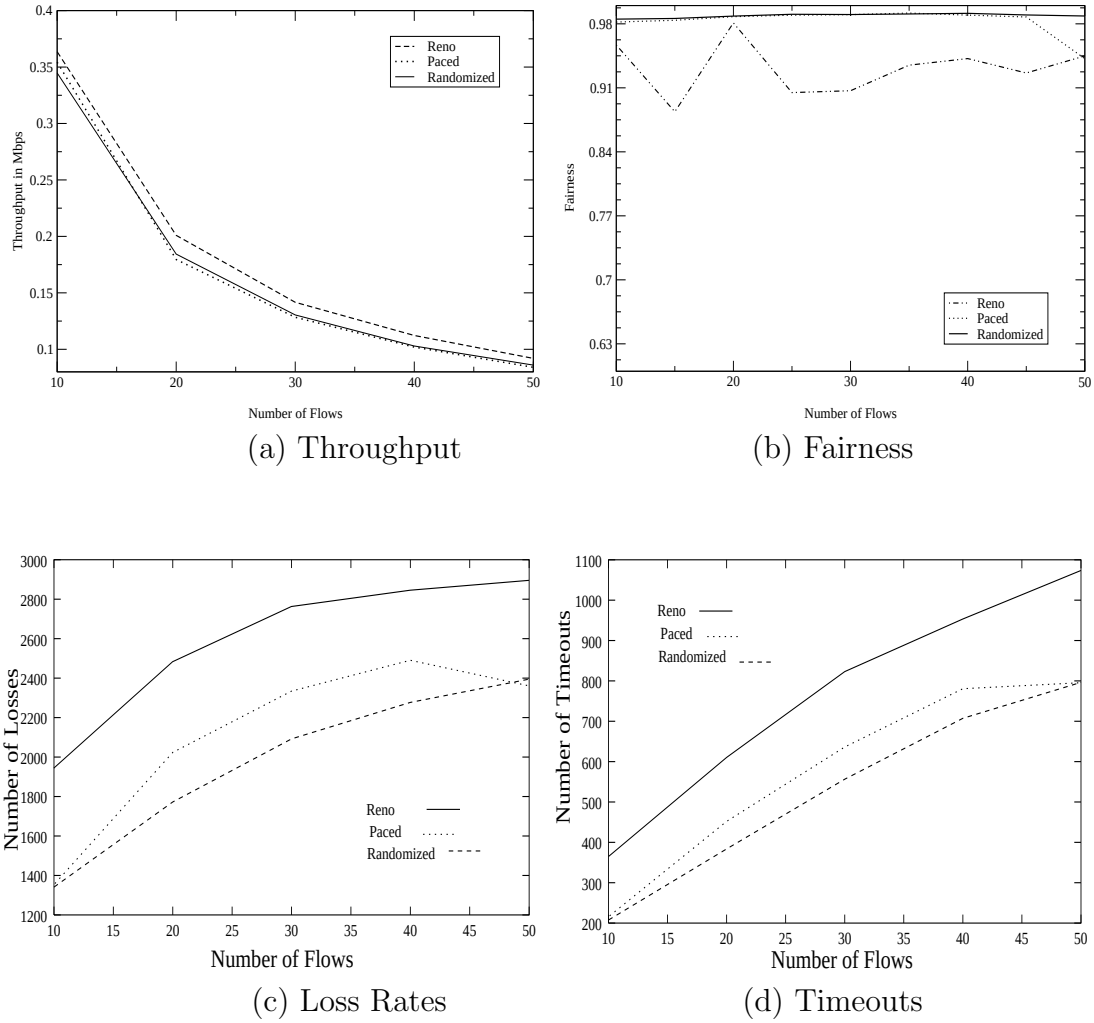
To study the performance of Randomized TCP with different RTT values for flows, we varied the RTT of each flow. The RTT of flows were in the range of 80ms to 120ms. The RTT for the $i^{th}, i \in (0, \ldots, N-1)$ flow was $80 + i * (120 - 80)/N$ where N is the total number of competing flows. Figure 3.7 shows the throughput,

fairness, loss rates and timeouts as the number of flows are increased from 10 to 50. Randomized TCP is the most fair and also the throughput achieved is marginally higher. However, it is interesting to note that Pacing also achieves almost the same performance as Randomized TCP. TCP Reno maintains its bias against flows with longer RTT (TCP throughput is inversely proportional to the RTT), which is shown by the fairness graph. Because of this bias, Reno's fairness curve is lowest. In [3], the authors contend that bias of TCP against longer RTT flows is considerably reduced with RED gateways due to uniform distribution of losses over time. The similarity of our simulation result to this indicates that randomization succeeds in distributing losses over time (to a certain extent), thereby decreasing TCP's bias towards long flows.

3.11.2 Short Web Like Transfers

In this section we present the performance of Randomized TCP for short flows. This is more representative of Web transfers. In this simulation we used a single bottleneck link of 4Mbps with a round-trip time of 100ms (figure 3.2). The buffer was fixed at 25 packets product. 25 flows were always maintained in the network. As soon as any flow finishes, a new flow initiates transfers. We varied the workload from 10 packets to 2500 packets.

Figure 3.8 (a and b) plots the latencies for Reno, Paced and Randomized TCP. For very short flows, i.e. for a workload of 10 packets to 200 packets, TCP Reno performs the best while Paced TCP performs the worst. Randomized TCP's performance though better than Paced TCP is not as good as Reno's. This can be attributed to the randomness which has been introduced in pacing intervals. Because of this randomization, Randomized TCP breaks ties and achieves better performance than Paced. Reno however, sends packets in bursts and is able to complete most of the transfers in the slow start. For workloads greater than 200 packets, Reno still performs the best, though the difference in the latencies for Reno and Randomized reduce as the workload increases. For Pacing, new flows starting in the slow start saturate the network. Due to late congestion signals in Pacing, many flows, even those who are in congestion avoidance, simultaneously drop packets thus



(a) Throughput (b) Fairness

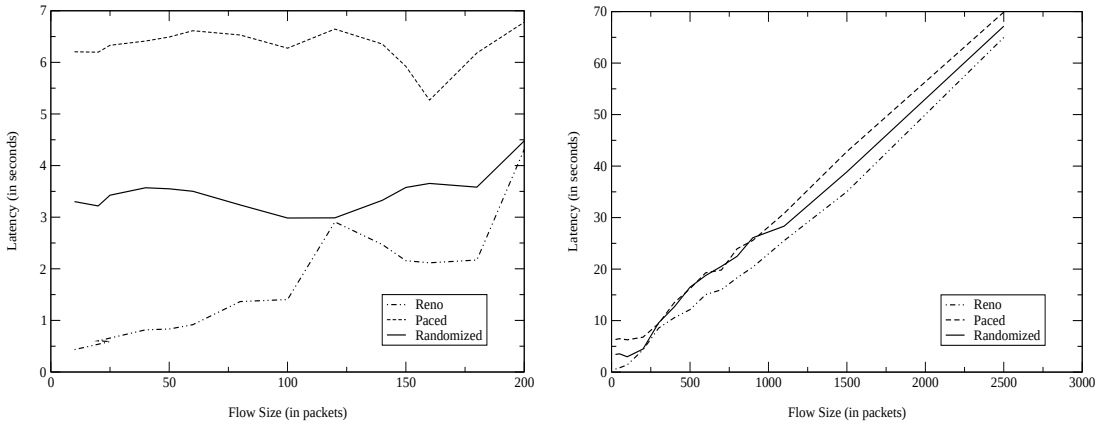
(c) Loss Rates (d) Timeouts

Figure 3.7: Throughput, Fairness, Loss Rates and Timeouts for a set of flows where each flow has a different RTT.

severely diminishing Paced TCP's performance [4]. Reno performs better because Reno flows send packets in clusters, a burst from a particular flow in slow start has only local effect; it does not effect all flows [4].

3.11.3 Interaction of Randomized TCP with TCP Reno

This section presents the result of multiplexing TCP Reno and Randomized TCP on the same link. In [4], the authors show that Paced TCP gets beaten down by TCP Reno, when multiplexed on the same link. This is because a single paced connection is more likely to have at least one of its packets encounter severe



(a) Short Workloads (b) Moderate Workloads

Figure 3.8: Latencies for Reno, Paced and Randomized for short and moderate Web like Workloads

TCP Type	Throughput	Losses (%)	Timeouts (%)
Reno	480.21	2.45	0.1
Paced	351.86	5.74	0.8
Reno	389.31	4.2	0.5
Random	408.92	5.1	0.8

Table 3.1: Comparison of Throughput (in pkts/sec), losses and timeouts for TCP Reno Vs (Paced, Random).

congestion when multiplexed with a bursty connection [4]. This problem is the same as a source's packets getting synchronized with the buffer overflow event. Hence that flow faces a disproportionate number of losses and a lower throughput [38]. This effect is reproduced in our simulations as shown in Table 3.1 where the throughput is considerably lesser for Paced TCP (351.86 Kbps) as against TCP Reno (480.21 Kbps). The RTT for this experiment was 100ms, the bottleneck link's capacity was 1 Mbps and it was configured with Drop Tail with 25 packets of buffer.

However, when Randomized TCP is multiplexed with TCP Reno, the fairness improves considerably. This is seen in Table 3.1 where the throughput for the Randomized TCP is 408.92 Kbps when compared to 389.31 Kbps for TCP Reno. This is primarily due to two reasons. Firstly, by modifying the increase parameter α of Randomized TCP we account for the extra delay being introduced by random-

ization. Secondly, by reduction of synchronization of the source to buffer overflow events, we ensure equitable distribution of drops.

3.11.4 Summary

To summarize the observations of this section:

- For bulk data transfer Randomized TCP performs as well as or better than TCP Reno and Paced TCP in almost all scenarios.
- Specifically, for bulk data transfer with same RTT amongst different flows, with higher multiplexing (of flows) Randomized TCP performs the best by increasing the throughput and fairness, reducing losses and timeouts.
- For bulk data transfers where every flow has different RTT, Randomized TCP clearly out-performs TCP Reno and Paced TCP. This is important because this is more representative of the Internet.
- In the scenario where all flows have different RTT and a Drop Tail queue at the bottleneck, randomization reduces the TCP bias against longer RTT flows and achieves a performance similar to RED gateways as mentioned in [3].
- With short web like transfers, Reno performs better than Randomized TCP. However as the workloads start to increase Randomized TCP catches up with TCP Reno. Small workload flows complete their transaction in slow-start (or with very small windows). As such, if we randomize the windows when they are small, randomization generally delays the sending times which results in increased latency. Moreover, our re-characterization of increase parameter (Section 3.4) does not come into play because it works for congestion avoidance phase. As such, we conjecture that one should not randomize the sending times when the windows are small (less than 4) and during the slow-start. However, these inferences are at best intuitive and need to be evaluated in detail. One could also calculate the adjustment factor for slow start (just like we did for steady state).

- Randomized TCP and TCP Reno can compete fairly at a bottleneck. This is primarily because of the modification of the increase parameter, α, of the congestion window growth in Randomized TCP. However, Paced TCP loses out to TCP Reno as already shown in [4].

3.12 Bias Against Long Flows, Phase Effects, Synchronization and Burst Losses

3.12.1 Bias Against Long Flows

It has been widely reported that Drop Tail gateways have a bias against long flows [39]. In this section we first demonstrate this bias and its reduction with the use of Randomized TCP. We present the results with single as well as multiple bottleneck topologies.

3.12.1.1 Single Bottleneck

We performed simulations with two flows, one shorter RTT source(60 ms) and another longer RTT source (80 ms) and for differing link capacities to demonstrate the bias against long flows. We varied the bottleneck capacity but kept the buffer size constant at 25 packets with Drop Tail queues. The simulation time was 500 seconds. Further, all the results reported in this section correspond to an average of 10 simulations. For the results corresponding to RED, the RED was configured to the recommendation in [33]. Specifically, the minimum threshold was set at one third of buffer length, the maximum threshold was set to four-fifth of buffer length, the queue weight was set at 0.002 and the maximum loss probability was set to 0.1 .

Consider the case when both the bottlenecks use simple Drop Tail queuing. If we assume that both flows see the same drop rate then the throughput for the two flows would be distributed as inversely proportional to the RTT (Throughput $\propto$ 1/RTT) [73]. Thus here the throughput should be distributed as 8/14 (0.57) and 6/14 (0.43) of the bottleneck capacity, amongst the 60ms and 80ms sources respectively. Now consider the case when the bottleneck bandwidth is 2 Mbps and

RTT	Type	Throughput (pkts/sec)	% Share of the Bottleneck	Loss (%)	Timeouts
Long	Reno	132.05	28	1.2	173
Short	Reno	333.58	72	0.3	35
Long	New Reno	113.15	25	1.6	107
Short	New Reno	333.63	75	0.2	17
Long	Reno	215.20	44	0.3	6
Short	Random	277.86	56	0.3	7
Long	Random	214.89	47	0.5	88
Short	Random	242.03	53	0.5	121
Long	Reno (RED)	216.05	46	0.3	2
Short	Reno (RED)	256.80	54	0.3	1
Long	New Reno (RED)	198.08	45	0.4	1
Short	New Reno (RED)	244.93	55	0.4	2
Long	Random (RED)	222.90	47	0.3	1
Short	Random (RED)	255.95	53	0.4	0

Table 3.2: Bias Against Large RTT Flows in a Single Bottleneck (2Mbps) Topology: The ideal % share of the bottleneck for the long flow is 43% and that for short flow is 57%. Drop Tail queues show a bias against large RTT flows with both TCP Reno and TCP New Reno. However, Randomized TCP removes this bias, moreover even a single Randomized TCP improves the fair sharing of the bottleneck. The results show that RED also removes the bias. Thus, Randomized TCP has similar performance gains as RED.

both the longer as well as the shorter flow use TCP Reno. The throughputs for the longer and the shorter flow in this case are 132.05 packets/sec (the standard deviation was 10 packets/sec) and 333.58 packets/sec (the standard deviation was 12 packets/sec) respectively (see Table 3.2). The share of the bottleneck for the two flows is 0.28 (long flow) and 0.72 (short flow) as against the theoretical values of 0.43 and 0.57 respectively. Therefore, we find that when both the sources use TCP Reno, bias against longer flow exist as expected. A similar result is obtained if we use TCP New Reno flows. Thus, the Drop Tail queues shows a sufficient bias against long RTT flows, irrespective of the congestion control scheme being used. However, with the same 2 Mbps bottleneck, if we randomize one source (in this case, the shorter source), we find that bias against longer flow is considerably reduced as seen in Table 3.2. In fact the throughput for the 80ms and 60ms flows are 215.20 packets/sec

(the standard deviation was 50 packets/sec) and 277.86 packets/sec (the standard deviation was 48 packets/sec) respectively. Also their share of the bottleneck are 0.44 (long flow) and 0.56 (short flow) as against the theoretical values of 0.43 and 0.57 respectively. This beneficial effect of Randomized TCP is preserved even if we randomize the longer flow or even if both the flows use Randomized TCP.

Thus Randomized TCP seems to remove the Drop Tail queue's bias against against large RTT flows. This behavior of Randomized TCP is similar to that of RED, as described in [39]. To further verify this argument we also evaluated the performance of TCP Reno, New Reno and Randomized TCP when RED queue was used at the bottleneck. As shown in Table 3.2 RED removes the bias against large RTT flows, further it results in a decrease in the number of timeouts. This can be attributed to the fact that RED manages the bottleneck queues proactively and thus rarely operates with full queues. This in turn translates into buffer space to accommodate any incoming burst of packets, thus preventing timeouts (which might have occurred due to a burst loss).

A similar statement about the bias against longer flow can be made for the other case where the bottleneck is of 3 Mbps (see Table 3.3). There too when both the flows use TCP Reno the bottleneck is shared as 0.30 for the long flow and 0.70 for the short flow instead of 0.43 and 0.57 respectively. Similarly, if both the flows use TCP New Reno the bandwidth is still shared disproportionately. But when one of flows uses Randomized TCP while the other uses TCP Reno, the bottleneck is shared as 0.41 for the long flow and 0.59 for the shorter flow. These two examples elicits that the bias against longer flows are present with TCP Reno and are removed with Randomized TCP. Once again, if RED is used at the bottleneck, the number of timeouts decrease and the bias against large RTT flows is also removed.

We investigated another simulation setup with a bottleneck of 1 Mbps, a Drop-Tail queue of 25 packets and 10 flows. In this experiment we had 5 sources each with RTTs of 60ms and 80ms.The results of this simulation are tabulated in Table 3.4. We first show the occurrence of bias against longer flows when all these sources used TCP Reno, and then we show the removal of this bias when all these sources used Randomized TCP. But more interestingly, *we demonstrate a reduction in bias*

RTT	Type	Throughput (pkts/sec)	% Share of the Bottleneck	Loss (%)	Timeouts
Long	Reno	195.29	30	0.6	143
Short	Reno	458.61	70	0.2	67
Long	New Reno	259.44	37	0.5	5
Short	New Reno	448.41	63	0.2	54
Long	Reno	276.42	41	0.2	9
Short	Random	395.68	59	0.2	13
Long	Random	273.89	42	2.7	59
Short	Random	374.14	58	2.3	70
Long	Reno (RED)	288.56	44	0.2	18
Short	Reno (RED)	371.35	56	0.2	28
Long	New Reno (RED)	302.66	43	0.3	2
Short	New Reno (RED)	394.99	57	0.2	7
Long	Random (RED)	308.80	46	0.2	2
Short	Random (RED)	368.00	54	0.2	4

Table 3.3: Bias Against Large RTT Flows in a Single Bottleneck (3Mbps) Topology: The ideal % share of the bottleneck for the long flow is 43% and that for short flow is 57%. Drop Tail queues show a bias against large RTT flows with both TCP Reno and TCP New Reno. However, Randomized TCP removes this bias, moreover even a single Randomized TCP improves the fair sharing of the bottleneck. The results show that RED also removes the bias. Thus, Randomized TCP has similar performance gains as RED.

even when any one source uses Randomized TCP and the rest use TCP Reno. This implies that a presence of even a ***single*** *Randomized TCP at a bottleneck might be helpful in reducing the bias against flows with larger RTT. Thus even an incremental deployment of Randomized TCPs would benefit the entire group of users.*

3.12.1.2 Multiple Bottleneck

In this section we evaluate the performance of TCP Reno and Randomized TCP with a multiple bottleneck topology. The topology is shown in Figure 3.9 consists of two bottleneck links of capacity 1 Mbps and delay of 20ms. All the other links in the figure have a capacity of 4 Mbps and delays as shown in Figure 3.9. The long flows have end-to-end propagation delay of 120ms while the short flows have

RTT	5 Short Reno 5 Long Reno	5 Short Random 5 Long Random	5 Short Reno 4 Long Reno + 1 Long Random	5 Long Reno 4 Short Reno + 1 Short Random
Short	62.6	41.50	45.51	50.02
Long	33.35	33.60	35.80	34.71

Table 3.4: Comparison of Throughput (in Kbps) for different configuration of competing 5 Long flows (RTT=80ms) and 5 Short Flows (RTT=60ms)

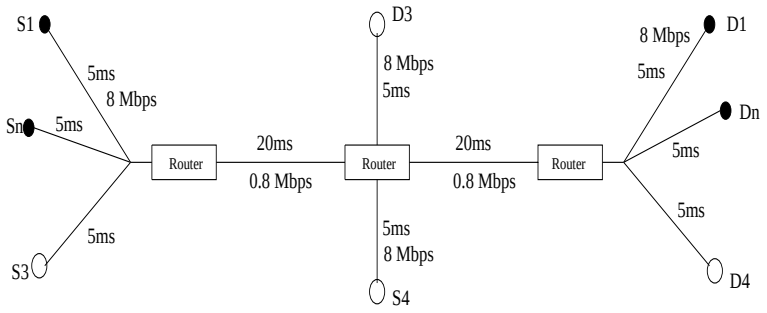


Figure 3.9: Multi Bottleneck Topology used in the simulation.

an end-to-end propagation delay of 60ms. Our simulation setup consist of 2 long flows denoted by (S1-D1) and (S2-D2) source-destination pairs and two small flows denoted by (S3-D3) and (S4-D4) source- destination pairs, as shown in figure 3.9. We investigate this topology when (S1,S2) and (S3,S4) use TCP Reno and Randomized TCP. Table 3.5 tabulates the results for different simulation setups. The results in this section correspond to an average of 10 simulations, with each simulation duration being 100 seconds. The standard deviations for results reported in this section were between 5-8 packets/sec for the long flows and 10-14 packets/sec for the short flows.

We can see from the Table 3.5 that there exists bias against flow(s) with longer RTT when all the flows use TCP Reno (or TCP New Reno), displayed by the considerable difference in their throughputs, and is subsequently removed when all the flows use Randomized TCP. However, an interesting observation again is that when the short flows use Randomized TCP while the long flows use TCP Reno, we see reduction in this bias. *This further supports our argument that a presence of*

Source	(S1,S2): Reno (S3,S4): Reno Drop Tail	(S1,S2): Random (S3,S4): Random Drop Tail	(S1,S2): Reno (S3,S4): Random Drop Tail	(S1:S2) Random (S3,S4) Reno Drop Tail
S1	33.12	39.39	41.63	36.72
S2	34.29	39.79	43.74	37.11
S3	170.12	138.14	146.60	152.04
S4	170.30	143.37	147.20	156.35

Source	(S1,S2): Random (S3,S4): Random RED	(S1,S2): Reno (S3,S4): Reno RED	(S1,S2): New Reno (S3,S4): New Reno Drop Tail	(S1,S2): New Reno (S3,S4): New Reno RED
S1	52.81	51.26	32.14	54.15
S2	53.19	51.74	34.61	54.66
S3	137.68	141.91	178.40	139.50
S4	139.75	141.56	183.20	149.29

Table 3.5: Bias Against Large RTT Flow in a Multi-Bottleneck Topology: Drop Tail queues' bias against large RTT flows with both TCP Reno and TCP New Reno persist. However, Randomized TCP removes this bias. Moreover presence of a single Randomized TCP flow at each bottleneck improves the fair sharing of the network. Once again, RED also removes the bias but Randomized TCP has similar performance gains.

even a single randomized flow **at every** *bottleneck is sufficient to reduce the bias against longer flow(s) and thus achieve a better fairness amongst flows.* In another simulation setup where the long flows use Randomized TCP and the short flows use TCP Reno, we see that the bias persists. This is intuitively true too. The long flows are the only sources of potential randomness at the bottleneck, which is visible at the first bottleneck. However, at the second bottleneck the streams arrive in phase because the randomness at the first bottleneck is broken by the "departure process" of the queue. Thus at the second bottleneck there is no randomization to break the bias against longer flows. Hence the long flows get beaten down and the bias persists.

However, when we use RED at the bottleneck queues, we can see that the bias against large RTT flows is removed. Moreover, if we look at Table 3.5 we can see that the not only has the fair share of the flows which go through both the bottleneck improves but also the over all link utilizations also increase. This

increase in link utilization with RED can be attributed to the reduction in timeouts. RED proactively manages the queues so as to avoid the impending congestion by dropping packets early. This in turn prevents the bottleneck queues to be full for a large duration thus allowing more space to accommodate packet bursts. Drop Tail queues unlike RED, do not attempt to avoid congestion rather they are configured only to absorb packet bursts and as a result often have full queues. This full queues with Drop Tail manifests itself with increase in timeouts (because of the inability to absorb the bursts).

In summary, the Drop Tail queues bias against large RTT flows persist irrespective of the TCP flavor (TCP Reno or New Reno) being used. However, presence of a single Randomized TCP flow at every bottleneck can improve the fairness in the network by reducing the bias against large RTT flows. This motivates the incremental deployment of Randomized TCP. RED also removes the bias against large RTT flows and also marginally improves the overall link and network utilization. But considering the fact that RED is not deployed on the network for a variety of reasons, the results in this section illustrate that one of the key benefits of using RED, i.e., removing bias against large RTT flows, can be emulated by using Randomized TCP.

3.12.2 Phase Effects

In [38] the authors show that phase effects with drop-tail queues can cause a source's loss events to get synchronized with the full queues. Consequently it loses a large number of packets and gets a very low throughput. The authors also note that an appropriate randomization included in the delay would reduce the phase effects. In this section we show the presence of phase effects in Drop Tail Gateways with TCP Reno as first shown in [38]. Subsequently, using the same simulation setup we show reduction in phase-effects with the use of Randomized TCP. We use the same simulation setup as discusses by the authors in [38]. Since phase-effects can be shown by either dis-proportionately high number of losses or low throughput in this work we chose losses to demonstrate phase-effects. Each point in these losses-time plot corresponds to the average losses for the last 50 seconds of the simulation.

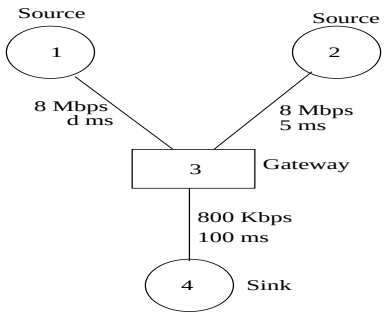
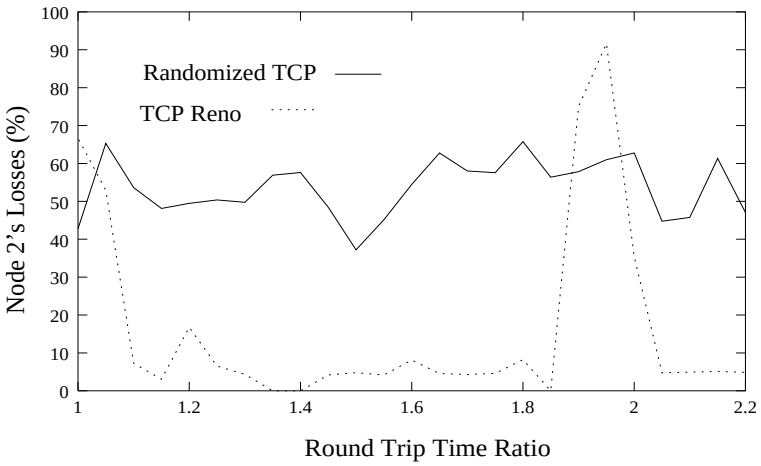


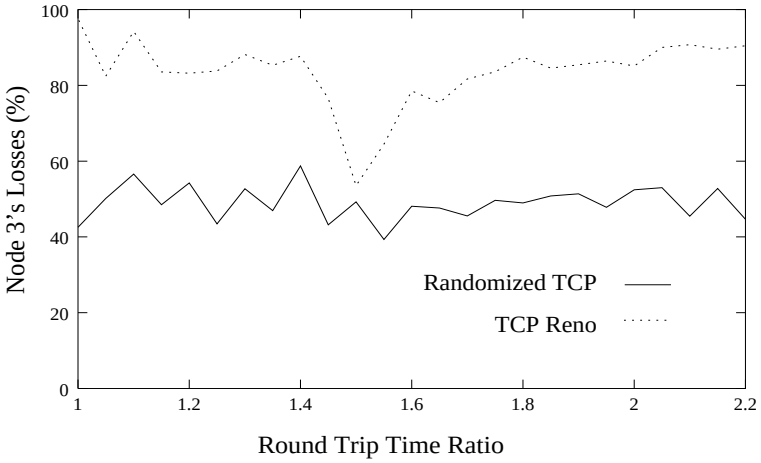
Figure 3.10: Single bottleneck Simulation Setup to show phase effects with Reno and Drop Tail Gateways.

Figure 3.10 shows the setup for a single bottleneck topology for a 100 ms simulation, a bottleneck buffer of 15 packets and the packet size of 1000B. In this simulation we vary the RTT of source 1 by varying the delay between source 1 and bottleneck. In figure 3.11(a) we plot the losses of Source 2 against the ratio of RTTs of the two sources. As can be seen from the figure 3.11(a) that for most of the data points, source 2 sees almost no loss (source 1 sees all the losses) while for some particular values of the RTT ratios (between 1.85-2.05) it sees most of the losses showing the presence of phase effects. However, we see that the phase effects are removed if Randomized TCP is used and the source 2 never sees disproportionately higher percentage of network losses.

We also evaluated Randomized TCP's performance vis-a-vis phase effects for a multiple bottleneck topology as shown in figure 3.12. In this simulation we varied the RTT of source 1 by varying the delay between Source 1 and bottleneck 1. The packet size used for the simulation was 1000B, the buffer length at each bottleneck was 15 packets (slightly more than 1 bandwidth delay product) and the simulation time was 100 ms. In Figure 3.11(b) we plot the percentage losses (of the total losses at the second bottleneck) as seen by Source 3 against the RTT ratios of source 1 and 2. Again it can be seen that Source 3 sees almost 80% losses with TCP Reno while the losses are considerably reduced (to about 40%) when Randomized TCP is used. This further verifies the presence of phase effects in Reno and Drop Tail gateways and removal of phase effects with the use of Randomized TCP.



(a) Single Bottleneck: Node 2 does not see disproportionate losses with Randomized TCP, Phase effects reduced.



(b) Multiple Bottleneck: Node 3 does not see disproportionate losses with Randomized TCP, Phase effects reduced.

Figure 3.11: Phase Effects

3.12.3 Synchronization

3.12.3.1 Synchronization in Bulk Data Transfer

We ran separate simulations with 2, 3, 10 and 25 flows of Reno, Paced and Randomized TCP and calculated pair-wise (between flows) covariance coefficients of congestion windows. We maintained the default simulation setup as described in Section 3.8 and the simulation time was 1000 seconds. The congestion window for each flow was sampled using a sample interval of 0.1 seconds, i.e., the congestion window was sampled approximately once every RTT. This sample set was then used

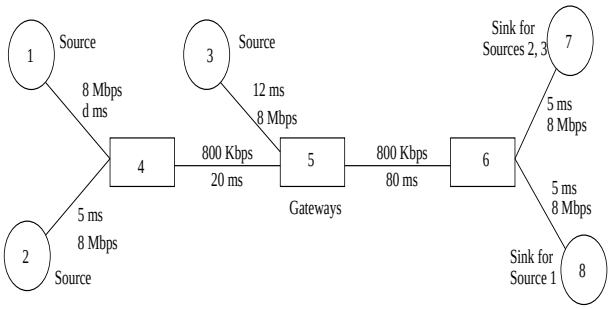


Figure 3.12: Multiple bottleneck Simulation Setup to show phase effects with Reno and Drop Tail Gateways.

Bandwidth	Reno	Paced	Randomized
3 Mbps	0.4254	-0.4124	0.1721
4 Mbps	0.3152	-0.1839	0.1604
5 Mbps	0.6700	-0.3302	0.0799

Table 3.6: Comparison of Covariance Coefficient of Congestion Window for two flows for TCP Reno, Paced and Randomized. (Value around 0 is Good.)

to calculate the pairwise covariance coefficients. product.

In our first simulation with 2 flows, we varied the bottleneck bandwidth from 3 Mbps to 5 Mbps while keeping the buffer fixed at 25 packets. Table 3.6 shows the covariance coefficients for each of the flows. It can be inferred that the synchronization in Reno increases as the bottleneck bandwidth increases. However Randomized TCP keeps the synchronization low while Paced TCP is out of phase synchronized. Also, it is interesting to note that while the synchronization increases in Reno with increase in bottleneck bandwidth, it decreases in Randomized.

In our second simulation with 3 flows, we kept the bottleneck bandwidth constant. Covariance coefficient values are tabulated in the table 3.7. Again, it is evident that Reno is the most synchronized and Paced TCP is out of phase synchronized. Also, it can be seen that both Paced and Randomized TCP lead to reduction in the synchronization.

Flow Pair	Reno	Paced	Randomized
(1,2)	0.5183	-0.1454	0.2525
(1,3)	0.5416	-0.1537	0.1422
(1,4)	0.3492	-0.1833	0.1535

Table 3.7: Comparison of Covariance Coefficient of Congestion Windows for 3 flows for TCP Reno, Paced and Randomized. (Value around 0 is Good.)

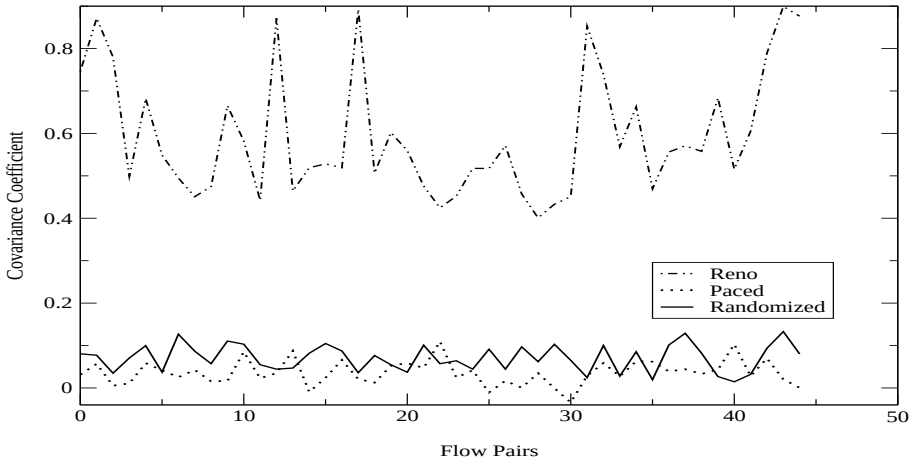
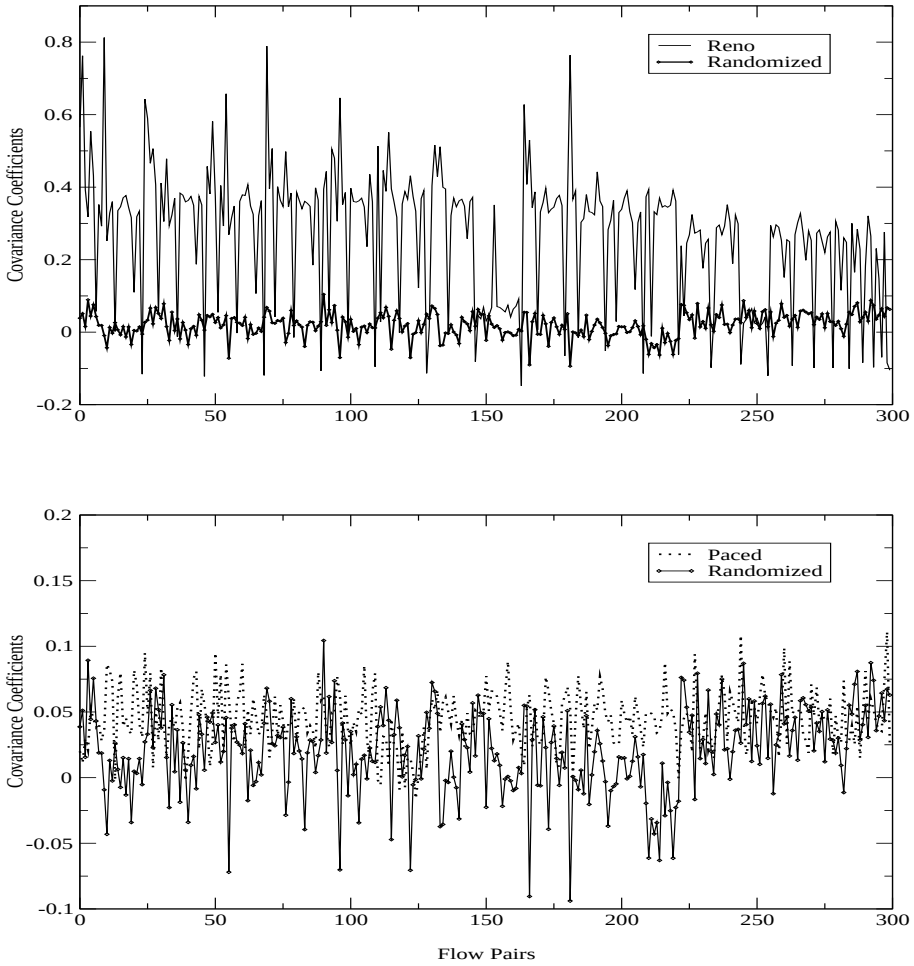


Figure 3.13: 10 flows Covar. coeff. of Congestion Window for (a) Reno, Paced & Randomized (b) Reno & Randomized, (c) Paced & Randomized

Figures 3.13 and 3.14(a and b) plot the pairwise covariance coefficients for 10 and 25 flows. The y axis of the graph plots the covariance coefficient against the pair of flows on x axis, i.e., each unit of x axis corresponds to a pair of flows, starting in the order (1,2), (1,3), ..., (2,3) .... Since the graphs for 25 flows are not visible on one graph we plot it in two. Fig 3.14(a) plots the covariance for Reno and Randomized TCP and 3.14(b) plots it for Randomized TCP and Paced TCP. Both Paced TCP and Randomized TCP break synchronization while Reno is highly synchronized. Also, as the number of flows start increasing, Randomized TCP starts to get better than Paced TCP.



(b) 25 flows

Figure 3.14: Covar. coeff. of Congestion Window for (a) Reno & Randomized, (b) Paced & Randomized

3.12.3.2 Synchronization with Short Web Transfers

In [4] the authors contend that one of the reasons for higher latency with Paced TCP in short web like transfers is that connections seem to get synchronized. In this simulation setup we have evaluated and verified their arguments. For the simulation we used a bottleneck link of 4Mbps, a RTT of 100 ms and a buffer of 25 packets. 25 flows were always maintained in the network. As soon as any flow finishes, a new flow initiates transfers. We varied the workload from 10 packets to 2500 packets.

Figure 3.15 plots the covariance coefficients of congestion windows for Paced

and Randomized TCP. A closer look shows that the covariance for Randomized TCP is *consistently* lower than that for Paced TCP. In Paced TCP packets reach the bottleneck at an uniform rate with near perfect interleaving. This causes all sources to lose packets, thereby resulting in all the sources cutting down their windows together, and hence higher covariance. But with randomization, the rate is not uniform at the bottleneck and packets from flows are dropped after differing times due to the extra delay incurred because of randomization. This means that sources decrease their windows at different times and hence the periods of increase and decrease are not as synchronized as in paced, resulting in a decreased covariance coefficient between the flows.

3.12.4 Burst Losses

In this section we investigate the proposition that Randomized TCP reduces the burst losses and also that the drops with Randomized TCP and Drop Tail queues are independent. For testing the first proposition, we varied the bottleneck bandwidth from 1-2 Mbps and the number of sources from 20 to 30. The end-to-end propagation delay was 200ms, the bottleneck buffer was set as 25 packets. We assumed that there is no reverse path congestion and the maximum number of back-to-back packets or burst at the bottleneck will be just 2. We in fact verified the also verified this argument by cross checking the burst loss size with the congestion window trace file for each flow at the bottleneck.

Table 3.8 shows the results average number of burst losses for TCP Reno and Randomized TCP as the bottleneck bandwidth and the flow multiplexing is increased. It can be inferred from the table that as the number of flows increase, with the bandwidth kept constant, the number of back-to-back losses increase in TCP Reno and decrease (or remain constant) in Randomized TCP. This supports our argument that Randomized TCP reduces burst losses.

It can also be conjectured here that Randomized TCP distributes the loss over time. This is because, TCP Reno and Randomized TCP have the same congestion control policy the total number of drops are likely to be same for both. Thus, by reducing the burst losses Randomized TCP makes the losses distributed. This

No. of Flows	1 Mbps		2 Mbps	
	Reno	RTCP	Reno	RTCP
20	87	23	1	27
25	119	18	100	31
30	141	15	168	28

Table 3.8: Comparison of average number of burst losses in Reno and Randomized TCP (RTCP)

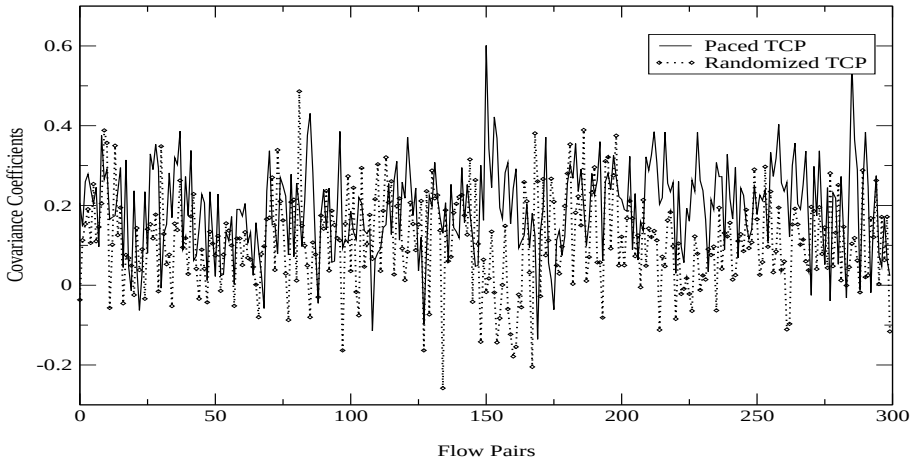


Figure 3.15: Covariance coefficients for Paced and Randomized TCP for a transfer of 2500 packets. (Value around 0 is good.)

argument is further supported by the results in Section 3.12.1. There it was shown that Randomized TCP is successful in removing the TCP bias against longer RTT flows with Drop Tail queues. In [3] the authors show that TCP bias against long flows can be reduced by Active Queue Management which distributes losses uniformly over time, specifically RED. The similarity of our simulation results in 3.12.1 suggest that Randomized TCP does succeed in making losses independent by distributing them over time.

3.12.5 Summary

The observations of this section can be summarized as:

- Randomized TCP increases the fairness amongst competing flows of different RTTs by removing the bias against the longer RTT flows (as found with TCP Reno) with Drop Tail queues.

- Presence of a "single" Randomized TCP flow at every bottleneck (Drop Tail Gateways) can reduce the bias against longer RTT flow at that bottleneck.
- Phase effects, which persist with TCP Reno with Drop Tail queues, are reduced if Randomized TCP is used.
- With bulk data transfers randomization reduces the synchronization of windows (thus loss events) as against TCP Reno. This should reduce the queue oscillations.
- Randomized TCP reduces synchronization with short web-transfers. This should lower average latency.
- Randomized TCP drastically reduces the number of burst losses. Specifically its performance increases as the number of flows increase.
- With Drop Tail queues Randomized TCP tries to distribute losses over time thus making them appear independent.

3.13 Binomial Congestion Control Algorithms

In [9] the authors propose a class of non-linear TCP compatible congestion control schemes called Binomial Congestion Control Schemes (BCCS) for audio and video applications. Formally, the Binomial Congestion Control scheme can be defined as:

$$W_{t+R} \leftarrow W_t + \alpha/W_t^k \quad if\ no\ loss \tag{3.32}$$

$$W_{t+\delta t} \leftarrow W_t - \beta W_t^l \quad if\ loss \tag{3.33}$$

where k and l are window scaling factors for increase and decrease respectively and α and β are increase the decrease proportionality constants. For any given values of α and β TCP Compatible BCCS can be defined by $k+l = 1 : k \geq 0,\ l \geq 0$. Inverse Increase Additive Decrease or IIAD is one such BCCS with $k=1,\ l=0$. Similarly Square Root Increase and Square Root Decrease or SQRT is defined as

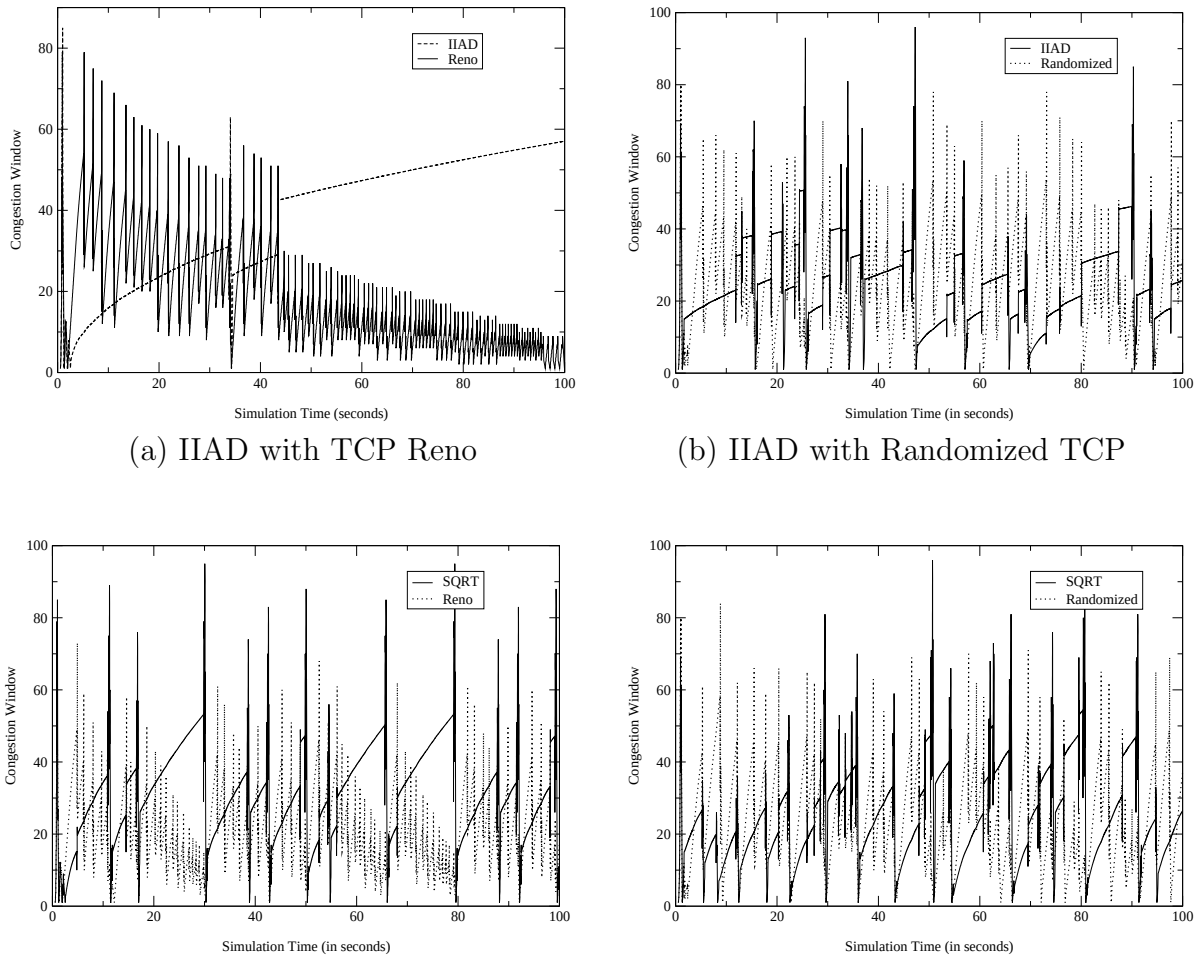
k=0.5, l=0.5. We refer the reader to [9] for a more detailed description of Binomial congestion control schemes.

In [9], the authors show that these algorithms, specifically IIAD and SQRT, beat down TCP when sharing a drop-tail gateway and hence suggest the use of RED gateways to maintain fairness. This unfairness is due to unequal distribution of drops amongst these flows. This behavior is seen in figure 3.16 a) and c). When we incorporate randomization into binomial schemes as well and make it compete against randomized TCP, we see a marked improvement in fairness as in figure 3.16 b) and d), due to the by now familiar reasons of de-synchronization and more uniform distribution of losses. The end-to-end propagation delay for this experiment was 100ms, the bottleneck link's capacity was 1 Mbps and it was configured with Drop Tail queue with 25 packets of buffer.

3.14 Conclusions

In this chapter we presented a methodology to introduce randomness in networks through end-to-end congestion control schemes. For the TCP case, we call it Randomized TCP. In this scheme, we space successive packet transmissions with a time interval $\Delta = RTT(1 + x)/cwnd$, where x is a zero mean random number drawn from an Uniform distribution. We showed that Randomized TCP, by introducing randomization in the network, reduces synchronization, phase effects and bias against bursty traffic, prevalent with current implementations of TCP and Drop Tail Gateways. We have also analytically characterized the new increase parameter for Randomized TCP to make it compete fairly with TCP. This was necessary because randomizing the sending times increases the RTT and as such the Randomized TCP losses to TCP Reno.

Randomized TCP reduces the bias against connections with larger RTTs with Drop Tail queues. The presence of a single Randomized flow at a bottleneck is sufficient to reduce the bias against longer RTT flows thereby motivating incremental deployment. Randomized TCP also reduces the burst losses and can also distribute losses over time thus emulating RED like properties. Multiplexing of Randomized TCP with TCP Reno helps in reducing synchronization and phase effects while



(a) IIAD with TCP Reno

(b) IIAD with Randomized TCP

(c) SQRT with TCP Reno

(d) SQRT with Randomized TCP

Figure 3.16: Performance of Binomial Congestion Control Algorithms with Randomization

increasing fairness. Additionally, when Randomized TCP is extended to Binomial congestion control schemes, there is a remarkable improvement in fairness, when competing with Reno. Consequently, it has high incentives for deployment.

Finally our results indicate that, *Randomized TCP can emulate the beneficial effects of RED in a distributed manner* without the complexities and unfavorable aspects of parameter tuning of RED. In addition, the benefits of randomization can be reaped even when it is partially deployed. However, we wish to emphasize that unlike RED which is a congestion avoidance scheme, Randomized TCP is just a congestion control scheme. Thus Randomized TCP does not emulate the congestion avoidance features of RED, at best it provides the other beneficial features of RED

which were achieved by introducing randomization in the network. We are currently working on implementation of Randomized TCP in the Linux Kernel.

CHAPTER 4
Selfish Flows: Characterization and Performance on Drop Tail Queues

4.1 Introduction

Randomized TCP is an end-system based solution which emulates some beneficial properties of AQM. Specifically Randomized TCP achieves fairness in the network by reducing burst losses. However, since Randomized TCP does not differentiate between flows and does not manage queues in the network and it can not protect flows under all circumstances. In this Chapter we illustrate this through some simulation setups. We show that in presence of selfish behavior in the network the end-system based techniques are insufficient to provide fair service to all users. However, to show this we first define ways in which selfish behavior can be defined. Specifically we show the existence of stable rate control schemes. This is then used to show the unfair distribution of bandwidth amongst competing flows.

The rest of the Chapter is organized as follows.

- In Section 4.2 we classify the selfish behavior in the network. Specifically we discuss the different type of flow control algorithms being used in the Internet.
- In Section 4.3 we show a class selfish schemes in the network which are obtained by using different increase and decrease policies. However, these policies do not change with time.
- We relax the assumption of constant increase decrease policy in Section 4.4 and introduce schemes which change some of their control parameters with time.
- In Section 4.5 we use these selfish schemes to highlight the problem that end-system based techniques are not sufficient to manage fair allocations in the network.
- Finally in Section 4.6 we summarize the arguments presented in this chapter.

4.2 Classes of Selfish Flows

In this section we will classify the selfish behavior of different rate control schemes. Since TCP is the most widely used transport protocol, for this classification we have used TCP as the benchmark flow, i.e. TCP flows are not considered to be selfish. Henceforth, we define selfish behavior as any rate control scheme which gets more share of the bottleneck bandwidth than TCP under same operating conditions. Though we have chosen TCP to identify selfish behavior, we would like to point out that TCP is just one special case, we could have as well chosen some other rate control scheme to recognize the selfish behavior.

At the outset we can classify selfish flows into two broad categories: a) responsive or adaptive flow and b) un-responsive or non-adaptive flow. A flow is called un-responsive flow if it does not react to the congestion indications being fed to it by the network. On the other hand responsive flows react to congestion indications by cutting down their rates.

Constant Bit Rate (or CBR) flows and UDP are the two main un-responsive rate control schemes. These rate control schemes are increasingly becoming popular in the network, especially as TCP introduces delays because of all its reliability mechanisms. Most of the multimedia and gaming applications use error protection through coding schemes and therefore are resistant to packet losses, though worrying about the end-to-end delays. As such these schemes use UDP. Real Audio, Internet telephony, and on-line games like Quake, Half life etc are some applications which use UDP [36]. Finally, flows which always increase their rate with total disregard to congestion indication would complete the definition of un-responsive schemes.

These un-responsive flows can be modeled by any linear utility function, i.e. $U(x) = ax$ and they are also characterized by constant marginal utility for any rate allocation. The utility function of such schemes is also given by a step function, i.e. till a particular rate these schemes have zero utility while after a particular rate these schemes have constant utility [85].

Responsive non-cooperative flows encompass a larger range of mis-behaving scenarios. However, their misbehavior can be differentiated on basis of their increase policy, (i.e. how they probe the network for available bandwidth) and their decrease

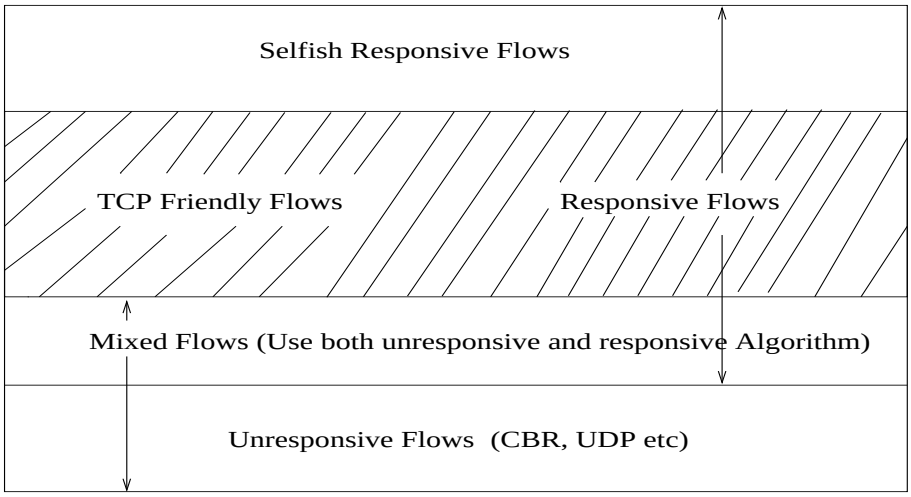


Figure 4.1: Classification of Selfish Behavior in the Network. Our region of interest is the Selfish Responsive Flows.

policy (or how they respond to congestion indication). These schemes could include strictly concave, concave or strictly increasing convex (with respect to rate) utility functions. Such a categorization of utility function leads to flows which are called greedy, i.e. they are always willing to consume any extra rate available (to them). This greediness enforces the strictly increasing condition on the utility functions.

Yet another class of non-cooperative functions manifest themselves as a mix of responsive and un-responsive flows. Some streaming application's rate control scheme falls in this category. These applications react to congestion indications till a certain limit (which could be rate or loss) and after that stops reacting to congestion indication and thus resorting to a CBR like transmission.

Finally, in figure 4.1 we show different rate control schemes. Though selfish behavior corresponds to all the sections other than TCP-Friendly flows, in this thesis *we will concentrate on managing selfish responsive flows*. Towards achieving this, in this chapter we will outline various techniques which can be used to generate selfish behavior in the network. Later in Chapter 5 we will present an edge system based solution for managing selfish behavior.

4.3 Selfish Rate Control Schemes and their Utility Functions

During the congestion avoidance phase TCP increases it's window by 1 packet every RTT and cuts it's window by half on receipt of congestion indication. Thus, the rate control from TCP can described as:

$$\begin{aligned} I : W(t+R) &\leftarrow W(t) + \alpha \\ D : W(t+R) &\leftarrow W(t) - \beta W(t) \end{aligned} \tag{4.1}$$

where R is the RTT, *I,D* represent the increase and decrease policy of TCP, respectively and α, β are the increase and decrease parameters. For TCP $\alpha = 1$ and $\beta = 0.5$.

Let us denote the instantaneous rate of a source by x and the packet loss probability as p. Further the relationship between window and rate is given as $W = x.R$. Then the window increase process can be written as

$$W(t+R) = W(t) + \alpha(1-p)^{W(t)} - \beta W(t)(1-(1-p)^{W(t)}) \tag{4.2}$$

Let us assume that the packet loss probability is close to 0. Then we may re-write the above equation as

$$W(t+R) = W(t) + \alpha - \beta W(t) p^{W(t)} \tag{4.3}$$

From the above equation, we may calculate the rate of change of congestion window as

$$\frac{W(t+R) - W(t)}{R} = \frac{\alpha}{R} - \beta \frac{W(t)}{R} p W(t) \tag{4.4}$$

or in other words

$$\frac{dW(t)}{dt} = \frac{\alpha}{R} - \beta \frac{W(t)}{R} p W(t) \tag{4.5}$$

Since at equilibrium the rate of increase of window will be equal to the rate of decrees of window we have $\frac{dW(t)}{dt} = 0$. We will also drop the time component from

the window, W(t), and instead write it as W. Thus at equilibrium we get

$$\frac{\alpha}{R} = \beta \frac{W}{R} pW \tag{4.6}$$

which can be re-written as

$$p = \frac{\alpha}{\beta W^2} \tag{4.7}$$

We can also express the above equation in terms of the sending rate, x, of the source. Moreover, the sending rate is also a measure of the throughput. Thus in steady state the following equation gives the relationship between the throughput x and the end-to-end loss probability, p.

$$p = \frac{\alpha}{\beta (xR)^2} \tag{4.8}$$

Then using the flow optimization analysis of Kelly, Low et al [53, 62, 56] and equilibrium properties of TCP [46] we can calculate the utility function of TCP as

$$U'(x) = p \tag{4.9}$$

However from the throughput analysis presented above, after some simplification at steady state we have

$$p = \frac{\alpha}{\beta (xR)^2} \tag{4.10}$$

Then using the equations (4.9) and (4.9) we have

$$U'(x) = \frac{\alpha}{(xR)^2 \beta} \tag{4.11}$$

Now, we can get the utility function of TCP by integrating the above equation. This gives us

$$U(x) = \frac{-\alpha}{R^2 x \beta} \tag{4.12}$$

The equilibrium rate allocations of TCP can be found using equation 4.8. It is clear that the throughput of TCP, x, increases with increase in a value of α and a decrease in the value of β. Thus, a straightforward way of generating selfish flows, with to

respect to TCP, would be to choose aggressive increase and decrease parameter. Therefore any choice $\alpha > 1$ or $0 < \beta < 0.5$ will result in aggressive rate control schemes. Akella et. al [5] use this definition of selfishness to evaluate the properties of Nash Equilibria with selfish flows and Drop Tail, RED queues.

Bansal et. al proposed non-linear increase decrease policies in form of Binomial Congestion Congestion Schemes. Formally, these schemes can be written as:

$$\begin{aligned} I : W(t+R) &\leftarrow W(t) + \frac{\alpha}{W(t)^k} \\ D : W(t+R) &\leftarrow W(t) - \beta W(t)^l \end{aligned} \tag{4.13}$$

where α, β, k, l define the Binomial Algorithm. TCP-Friendly Binomial schemes are defined as all the scheme satisfying $k + l = 1$. Using an analysis similar to the one stated above the utility function of the binomial schemes can be calculated as:

$$p = \frac{\alpha}{\beta\, W(t)^{k+l+1}} \tag{4.14}$$

$$= \frac{\alpha}{\beta\, (xR)^{k+l+1}} \tag{4.15}$$

$$U(x) = \frac{-\alpha}{\beta\ (k+l) R\, (xR)^{k+l}} \tag{4.16}$$

Again, selfish schemes can be generated by changing the increase and decrease parameters, α and β respectively. However, the throughput of Binomial schemes is given as

$$x = \frac{1}{R} \left(\frac{\alpha}{\beta d} \right)^{\frac{1}{k+l+1}}$$

Since, $p < 1$ it is easy to see that equilibrium allocations increase with decreasing value of $k + l$. Thus selfish rate control schemes can be generated by choosing $k + l < 1$. Further the utility function of Binomial schemes is strictly concave with respect to β, x and k, l.

Sastry et. al further relaxed the increase and decrease policy to come up with the following rate update rules:

$$I : W(t+R) \leftarrow W(t) + f(W(t)) \tag{4.17}$$

$$D : W(t+R) \quad \leftarrow \quad W(t) - g(W(t))$$

where *f, g* are some functions of window, W. The utility function of such schemes can be calculated as

$$U(x) = \int_x \frac{1}{Rxf(x)g(x)}dx \tag{4.18}$$

and the TCP-Friendly schemes are given by

$$f(x)g(x) \propto x \tag{4.19}$$

Further, Binomial congestion control schemes are special case of the above model.

From the discussions of utility function characterization of Binomial schemes and TCP, it is clear that selfish flows are given by

$$U(x) \propto \frac{-1}{x^n}, \quad n < 1 \tag{4.20}$$

Then using equation 4.20 selfish schemes can be generated by choosing *f, g* such that

$$f(x)g(x) < x \tag{4.21}$$

Also it can be easily shown that the strict concavity of the Utility function can be guaranteed by the following equation

$$\frac{-1}{x} > \frac{f'(x)}{f(x)} + \frac{g'(x)}{g(x)} \tag{4.22}$$

4.4 Aggressive Rate Control Scheme: Control Parameters are Time Dependent

In the previous section we looked at different selfish flow control schemes which were obtained by changing the increase and decrease rules. However, these rules do not change with time. In this section we consider rate control schemes where some parameters are allowed to change with time. Specifically, we will look at the time-varying Binomial schemes. We can get selfish rate control by either modifying the decrease parameter, β over time or by changing its binomial parameters k and l.

In the previous section we found the utility function for binomial schemes. It can be seen that the utility function is strictly concave with respect to x, β, k and l. In this section we will evaluate what update rules are allowed for decrease parameter and k, l.

4.4.1 Modifying the decrease parameter β

Let us assume that the source changes it's decrease parameter, β_s with time and lets represent it as $\beta_s(t)$. Assume that the other parameters, α, k and l are kept constant. Then the network optimization

$$\begin{aligned} maximize \quad & \textstyle\sum_{s\in S} \quad w_s U_s(x_s, \beta_s) && (4.23)\\ subject\ to \quad & \textstyle\sum_{s\in S(l)} \quad x_s \le C_l, \quad \forall l && (4.24) \end{aligned}$$

can be rewritten as

$$J(x_s, \beta_s) \;=\; \underbrace{max}_{x_s} \sum_s w_s U_s(x_s, \beta_s) - \gamma \sum_l \int_0^{\sum_j x_j} p_l(C_l, x)dx \qquad (4.25)$$

where w_s is the weight for the utility function, U_s, of source s, x_s is the rate of source s, γ is some constant greater than 0 and p_l represents the penalty function and is given as

$$p_l(C_l, \lambda) = \frac{(\lambda - C_l)^+}{\lambda}. \qquad (4.26)$$

where $y^+ = max(y, 0)$. Using the utility function of the binomial scheme we can rewrite equation 4.25 as

$$J(x_s, \beta_s) = \underbrace{max}_{x_s} \sum_s w_s \frac{-\alpha}{\beta_s(t)\ (k+l)d\,(x_s(t)d)^{k+l}} - \gamma \sum_l \int_0^{\sum_j x_j} p_l(C_l, x)dx \qquad (4.27)$$

We will assume that the function $J(x_s, \beta_s)$ is strictly concave. This constrains the range of possible choices for $\beta_s(t)$, but more importantly it guarantees us a unique optimum. Moreover, this constraint on $J(x_s, \beta_s)$ will also help us in proving that the optimum solution is also stable. Thus from the strict concavity assumption on

$J(x_s, \beta_s)$ the sufficient condition to reach the optimal point is given by

$$\frac{dJ(x_s, \beta_s)}{dt} > 0 \tag{4.28}$$

To achieve this consider the following differential equation

$$\frac{dJ(x_s, \beta_s)}{dt} = \frac{\partial J(x_s, \beta_s)}{\partial x_s}\frac{\partial x_s}{\partial t} + \frac{\partial J(x_s, \beta_s)}{\partial \beta_s}\frac{\partial \beta_s}{\partial t} \tag{4.29}$$

which can be calculated as

$$\frac{dJ(x_s, \beta_s)}{dt} = \left(a_s \frac{U_s^{'}(x_s)}{\beta_s} - \gamma \sum_l p_l(C_l, \sum_j x_j)\right)\dot{x_s} + \frac{b_s}{\beta_s(t)^2 x_s(t)^{k+l}}\dot{\beta_s} \tag{4.30}$$

where a_s, b_s are constant and can be calculated as

$$a_s = \frac{w_s \alpha}{d^{k+l+1}} \tag{4.31}$$

$$b_s = \frac{w_s \alpha}{(k+l)d^{k+l+1}} \tag{4.32}$$

Assume we choose the following rule for updating the rates

$$\dot{x_s} = \rho\left(a_s \frac{U_s^{'}(x_s)}{\beta_s} - \gamma \sum_l p_l(C_l, \sum_j x_j)\right) \tag{4.33}$$

where ρ is some constant greater than 0. Such a choice of update rule also satisfies our needs as this rule corresponds to the window dynamics of binomial congestion control schemes. Substituting this update rule into equation (4.30) we get

$$\frac{dJ(x_s, \beta_s)}{dt} = \rho\left(a_s \frac{U_s^{'}(x_s)}{\beta_s} - \gamma \sum_l p_l(C_l, \sum_j x_j)\right)^2 + \frac{b_s}{\beta_s(t)^2 x_s(t)^{k+l}}\dot{\beta_s} \tag{4.34}$$

From the above equation we can conclude that a **sufficient condition** for the game to reach its optimal point is that $\dot{\beta}(t) \geq 0$, i.e the decrease parameter increases with time. However in the previous section we have seen that the selfish behavior of a rate control algorithm increases with a decreasing value of β. But our present update rule for β, $\dot{\beta}(t) > 0$ will eventually make a selfish scheme TCP-friendly

because eventually we will reach a value of $\beta \geq 0.5$. Thus we are interested in the update rule where the end-system is allowed to decrease its value of β with time. Assume that we choose $\dot{\beta}(\mathrm{t}) \leq 0$ then the optimal point is reached if equation (4.28) is satisfied. This can be further expressed as

$$|\dot{\beta}(t)| < \frac{\dot{x_s}(t)^2}{\rho} \frac{\beta_s(t)^2 x_s(t)^{k+l}}{b_s} \tag{4.35}$$

$$< \frac{\dot{x_s}(t)^2}{\rho} \frac{\beta_s(t)^2 dW_s(t)^{k+l}}{w_s \alpha} \tag{4.36}$$

$$\dot{\beta}(t) < 0 \tag{4.37}$$

Assuming that the window size is always greater than or equal to 1, we can upper bound the decrease rate as

$$|\dot{\beta}(t)| \leq \frac{\dot{x_s}(t)^2}{\rho} \frac{\beta_s(t)^2 d}{w_s \alpha} \tag{4.38}$$

Since the Lagrangian relaxation, $J(x_s, \beta_s)$, is strictly increasing and concave in it's argument a unique and stable optimal solution exist. Also from previous analysis we have that at equilibrium rate of increase (of window) is equal to the rate of decrease. As such, we have that at equilibrium $\dot{x_s} = 0$. Thus the minimum value of $\dot{x_s}^2$ can be calculated as

$$inf\{\dot{x_s}^2\} = 0 \tag{4.39}$$

where *inf* is the infimum. Also from equations (4.38) and (4.37) we have

$$|\dot{\beta}(t)| \leq 0 \tag{4.40}$$

$$\dot{\beta}(t) < 0 \tag{4.41}$$

Thus from the above equations we can conclude that the system does not have a always have a unique optimal solution if $\dot{\beta_s}(t) < 0$. In other words, we cannot get a selfish scheme by consistently decreasing our decrease parameter, $\beta_s(t)$. However, we may generate selfish schemes by choosing a very small decrease parameter and then consistently increasing it. Moreover, in such cases we need to bound the final value of $\beta_s(t)$ to be less than 0.5. Thus we may generate time-variant selfish congestion

control scheme (when compared to TCP) if

$$0 < \beta_s(0) < 0.5 \tag{4.42}$$
$$\frac{\partial \beta_s(t)}{\partial t} > 0 \tag{4.43}$$
$$sup\{\beta_s(t)\} < 0.5 \tag{4.44}$$

where $sup(y)$ is the supremum of any series y.

4.4.1.1 Global Stability

Since the function $J(x_s, \beta_s)$ (equation 4.25) is strictly concave it has a unique maxima. Lets denote this maximum by J_{max}. Then the $J = J_{max} - J(x_s, \beta_s)$ can be thought of as Lyapunov function. It is easy to see that $J \geq 0$. Further it can be shown that J is Lipschitz continuous on $x_s \in (m_s, M_s), m_s > 0$ and $\beta > 0$. Then from equations (4.34, 4.43) we can conclude that

$$\frac{dJ}{dt} = -\frac{dJ(x_s, \beta_s)}{dt} < 0 \tag{4.45}$$

Thus from Lyapunov's stability theorem (Theorem 3.1 [55]) we have that the update rules for rate and the decrease parameter yield a stable system under equations (4.34, 4.42-4.42).

4.4.2 Modifying the Window Scaling Parameters, k, l

In the previous section we assumed that the window scaling parameters, k, l were held constant. In this section we relax this assumption however we add the assumption that the decrease parameter β is held constant. Further let us denote by $n = k + l$. Throughout this section we will use n for our analysis and use it to make observations about changing k, l with time. Then we may write the network optimization problem as

$$J(x_s, n_s) = \underbrace{max}_{x_s} \sum_s w_s \frac{-\alpha}{\beta\ nd\,(x_s(t)d)^n} - \gamma \sum_l \int_0^{\sum_j x_j} p_l(C_l, x)dx \tag{4.46}$$

where p_l is given by equation (4.26), and n_s represents the window scale parameter for source s. Since the above objective function is strictly concave in n and x the sufficient condition to reach the optimal point is

$$\frac{dJ(x_s, n_s)}{dt} > 0 \tag{4.47}$$

$$\frac{dJ(x_s, n_s)}{dt} = \frac{\partial J(x_s, n_s)}{\partial x_s}\frac{\partial x_s}{\partial t} + \frac{\partial J(x_s, n_s)}{\partial n_s}\frac{\partial n_s}{\partial t} \tag{4.48}$$

Assuming the update rule for rate is given by equation (4.33) the sufficient condition for above algorithm is met if

$$\frac{\partial J(x_s, n_s)}{\partial n_s}\frac{\partial n_s}{\partial t} \geq 0 \tag{4.49}$$

But

$$\frac{\partial J(x_s, n_s)}{\partial n_s}\frac{\partial n_s}{\partial t} = \frac{w_s \alpha}{\beta R}\left(n_s log(x_s R_s) + \frac{1}{n_s^2}\right)\dot{n_s}(t) \tag{4.50}$$

and since $\alpha, \beta, w_s, R, n_s$ are all positive we have that the optimal point will always be reached if

$$\dot{n_s}(t) \geq 0 \tag{4.51}$$

This update rule points that if the window scaling factors increase or stay constant with time, the optimal point will always be achieved. Further this also points to the following interesting update rule for the window scaling parameters

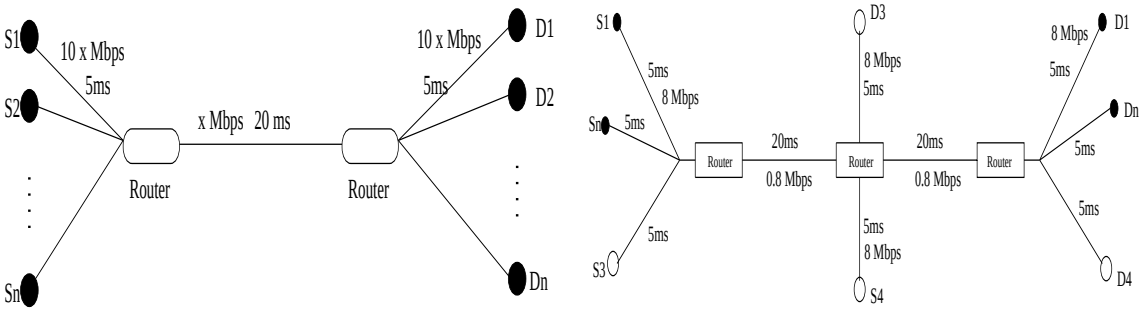
$$\dot{k_s}(t) = -\dot{l_s}(t) \tag{4.52}$$

4.4.2.1 Global Stability

Using an analysis similar to the one in Section 4.4.1.1 $J = J_{max} - J(x_s, n_s)$ can be thought of as the Lyapunov function. It's easy to see that J is again always positive and Lipschitz continuous on $x_s \in (m_s, M_s), m_s > 0$ and $n_s > 0$. Further from equations (4.47, 4.49) we have that $\dot{J}(t) \leq 0$. Thus J satisfies all the conditions for Lyapunov stability (Theorem 3.1 [55]) and the update rules of n_s, x_s are stable.

4.5 Selfish Flows and Drop Tail Queues

In the previous sections we showed different ways to generate mis-behaving flows in the network. In this section we will validate our claim that the Drop Tail queues do not protect TCP flows from mis-behaving flows. Further, Randomized TCP also is a marginal improvement over TCP and consequently does not protect TCP flows.



(a) Single Bottleneck (b) Multi-Bottleneck

Figure 4.2: Topologies used in the Simulations.

Because of the simplicity of implementation and understanding, for this work we used Binomial scheme to generate misbehaving flows. We fixed the values of α, β as 1 and 0.5 respectively. TCP flows are defined by $k = 0, l = 1$ and as discusses previously in this Chapter, misbehaving flows are defined by $k + l < 1$. This is because network allocates more resources to flows which have higher marginal utility, U_s'. Henceforth, we will use the k and l values to identify misbehaving flows.

In figures 4.3 and 4.4 we plot the throughputs for flows competing on a single and multi-bottleneck topologies respectively. We first present the result with a single bottleneck (4.2 a) of 0.8Mbps and access links of 8Mbps for 2 competing flows. We evaluated the single bottleneck topology for the two cases, one when we used TCP Reno flow and a misbehaving flow (k=0, l=0.5) and in the second case we replaced the TCP Reno flow with Randomized TCP flow. In both the cases flows have same RTT of 60ms. It can be seen from the figure 4.3 that in both the cases the misbehaving flow gets most of the bottleneck share. Moreover it beats the TCP

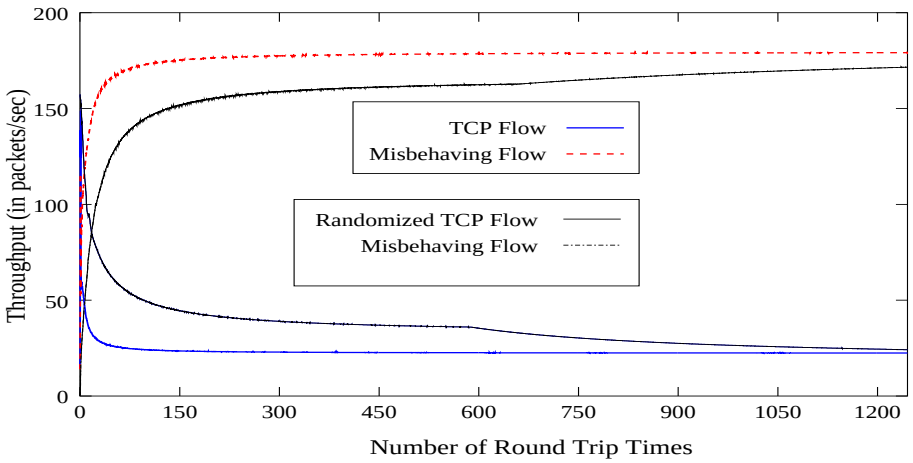


Figure 4.3: Single Bottleneck: Throughputs (in pkts/sec) for two competing flows, one is TCP while the other is Mis-behaving (k=0,l=0.5)

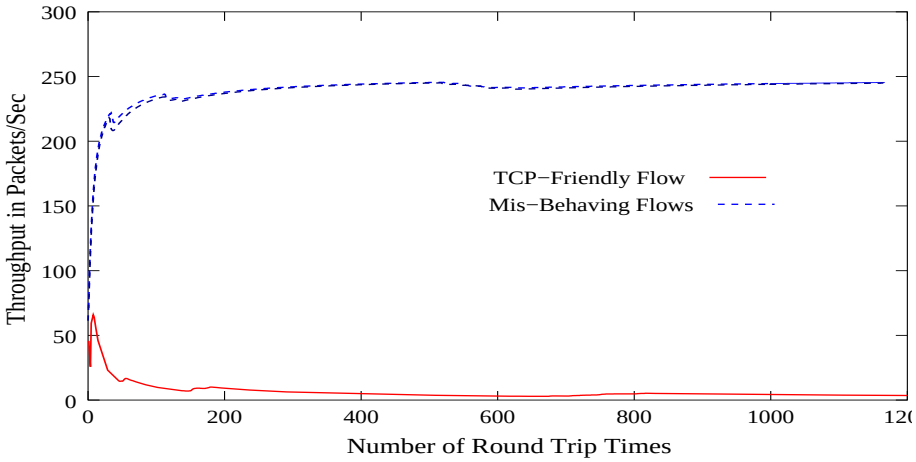


Figure 4.4: Multi Bottleneck: Throughputs (in pkts/sec) for 2 competing flows on a network of Drop Tail queues. One flow is TCP while the other is Misbehaving (k=0,l=0.5).

flows comprehensively. Further, Randomized TCP only marginally improves the performance. This can be explained by the fact that network allocates equal losses to both the flows and the misbehaving flow by cutting down its window slowly as compared to TCP flows always get a larger share of the bandwidth.

Figure 4.2 b) show a multi-bottleneck topology with a TCP flow traversing both the bottlenecks while one short mis-behaving flow (k=0, l=0.5), each going through one bottleneck. It can be seen from figure 4.4 that TCP flow is almost shut out by the mis-behaving flows, who now get all the bandwidth. Not only is the TCP

flow is forced into multiple timeouts (23 for this case) but these timeouts occur with very small windows and are often back to back. Similar results were obtained with Randomized TCP flow. In summary, with Drop Tail queues mis-behaving flows may get significant share of the bandwidth, almost to the extent of shutting out TCP flows.

4.6 Summary

In this chapter we first classified selfish flows. These flows were primarily characterized by their response to congestion indication. There are some flows which do not react to congestion indication. Instead either they keep sending their traffic at a specified rate on keep increasing their rates with total disregard to the state of congestion in the network. These flows are called non-responsive flows. However, there is another section of congestion control scheme, which reacts to congestion indication by cutting down the rate. But, in spite of this rate cut these flows may be selfish. This can be attributed to either their aggressive window increase (as compared to TCP) or by a smaller rate cut (than TCP). These flows are called responsive flows. The last category of the flows use both responsive and non-responsive policy. These flows, upon reception of congestion indication cut down their rates. However, beyond a certain threshold (which could either be a rate limit or a loss rate limit) these flows stop reacting to congestion signals, instead they keep sending data into the network at a constant rate.

After this classification of selfish behavior of rate control schemes we suggested ways in which these selfish responsive schemes could be implemented on the network. The simplest selfish schemes can be obtained by using aggressive increase and decrease parameter in TCP, i.e. $\alpha > 1$ or $0 < \beta < 0.5$. Another suggested method of generating selfish schemes was using Binomial schemes. These algorithms are characterized by window scaling parameters *k, l* besides the increase and decrease parameters. Selfish schemes can be created by using aggressive window scaling parameters, specifically by choosing k or l such that $k + l < 1$. These increase and decrease policy can be further generalized to include any function *f(x), g(x)* such that they are always positive. If *f(x), g(x)* are chosen such that $f(x)g(x)$ is sub-linear

then we can produce a variety of selfish rate control schemes. Besides these schemes, we also looked at schemes where the user is allowed to change his congestion control parameter with time. In this chapter we formulated guidelines under which these schemes might be stable.

Finally, in this chapter we evaluated the performance of Drop Tail queues in presence of selfish rate control schemes. Our results show that Drop Tail queues cannot protect TCP flows from selfish users. Moreover the performance of Drop Tail queues deteriorate with multiple bottleneck, so much so that selfish flows can almost shut out TCP flows. Further, Randomized TCP also does not protect flows. This can be explained by the fact that network allocates equal losses to both the flows and the misbehaving flow by cutting down its window slowly as compared to TCP flows always get a larger share of the bandwidth. Therefore it is imperative that we use some network based strategies to protect flows from selfish behavior in the network.

CHAPTER 5
Uncooperative Flow Control: An Edge-Based Re-marking Framework for Congestion Response Conformance in the Network

5.1 Introduction

Over the years as the Internet has evolved TCP has formed the backbone of its stability. TCP placed the trust of responsive behavior, i.e. decrease rate if there is congestion, at the end-user and as a result the core network could be kept simple. However as the application needs changed newer rate control schemes were proposed. Moreover, new software advancements have also placed users in a position where they can change their congestion control schemes. As such we now have an Internet which operates with a spectrum of transport protocols, some of which don't even react to congestion indications. Thus, over the years, the trust placed in the end-system to react to congestion indications has been sufficiently weakened. In this thesis, the flows which break the trust of the network by not reacting in appropriate or a standard way (e.g. TCP) will be called uncooperative flows. In this thesis, we will also refer to uncooperative flows as non-cooperative or non-conformant flow.

It has been widely reported that this breach of trust or absence of end-to-end congestion control schemes and presence of uncooperative users can lead to TCP unfriendliness and also cause congestion collapse [5, 35]. Moreover, as reported recently and further validated by our results, these uncooperative flows can also force a traffic volume based denial-of-service to their cooperative counterparts [58, 42]. Also as the network grows and the access pipes get bigger, uncooperative flows will pose a significant challenge before the network providers. This is because of uncooperative flows have the ability to monopolize bottleneck space and their disregard to appropriate congestion responses may cause congestion collapse thus effecting the stability of the Internet. Some architectural responses such as use of AQM schemes, schedulers and pricing mechanisms have been suggested to manage the uncooperative flows [35].

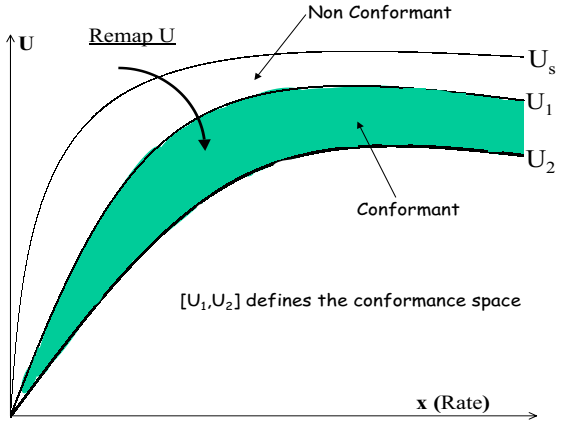


Figure 5.1: Mapping a uncooperative user to a conformant space.

However, use of AQM and schedulers require deployment at all (bottleneck) routers in the network, which is not only expensive but also requires significant network upgrade. These deployment considerations coupled with presence of simple Drop-Tail queueing schemes at all routers in the Internet present us with an interesting question - *What are the appropriate alternate architectural responses for managing a network of un-cooperative users, such that it requires minimal network support ?*

In this chapter we explore architectural responses for managing the entire spectrum of uncooperative sources at the *edge of the network*. The biggest advantages of the Uncooperative Congestion Control framework are that it is independent of buffer management scheme deployed on the network and works equally well in a dropping or a marking based network. The framework presented in this chapter can also be used to distribute rates amongst user's according to some a-priori fair rate allocation, while still allowing users to choose their rate control schemes. Thus, this proposal can be used to enforce congestion response conformance e.g. TCP-Friendliness. Moreover, our framework allows for a enforcement of a broader range of congestion response conformance criteria.

The framework presented in this chapter follows from the flow optimization model [53, 56, 62], specifically the duality framework of Low etal. [62]. The flow optimization framework is a network-based approach for modeling rate control schemes and computing average sending rates and end-to-end loss probabilities for users. In this work we describe a user with his rate, x and a utility function, $U(x)$, while

a network is identified with link capacities. Thereupon, the users try to maximize their utility functions subject to link capacity constraints and in the process we derive rate control schemes for the users and link price update mechanisms for the network.

In this chapter we call users cooperative if their utility functions fall within some a-priori specified target range of utility functions. For example in Fig 5.1 $[U_1, U_2]$ defines the cooperation boundaries or the target range. We show that through a transparent penalty function transformation the network provider can *re-map* the utility functions of the uncooperative users to a target range of utility functions, see Fig 5.1. Further, this *re-mapping* can be easily implemented at the edge of the network. Moreover, our framework allows users freedom to choose arbitrary concave utility functions or in other words they can pick any rate control scheme [53, 56, 62]. This solution presented in this chapter is attractive because it *does not require any upgrades in the routers* of the network, they function as usual, i.e. they may mark, or drop packets using any buffer management scheme (including Drop-Tail policy). Fig 5.2 shows the model for policing uncooperative users.

The problem of managing uncooperative users has been actively researched [5, 35, 28, 60, 64]. Router based mechanisms, such as Active Queue Management (AQM) schemes, schedulers and pricing mechanisms, have been suggested for managing uncooperative users in the network. However, use of AQM schemes, schedulers require deployment at all (bottleneck) routers in the network, which is not only expensive but also requires significant network upgrade. Further, almost all AQM proposals try to implement max-min fair rate distribution on the network which might not always be the desired fairness criteria for the network provider, especially if he wants to provide differentiated services. Moreover AQM schemes face configuration problems and also lack of deployment of ECN. As a result, they are not deployed and Internet works on simple Drop Tail queueing and the problems due to uncooperative flows persist.

The framework presented in this chapter also suggests that management of uncooperative flows need *not* be coupled with AQM design and can be simply viewed as an edge network based policing question. Our mechanisms may also be thought

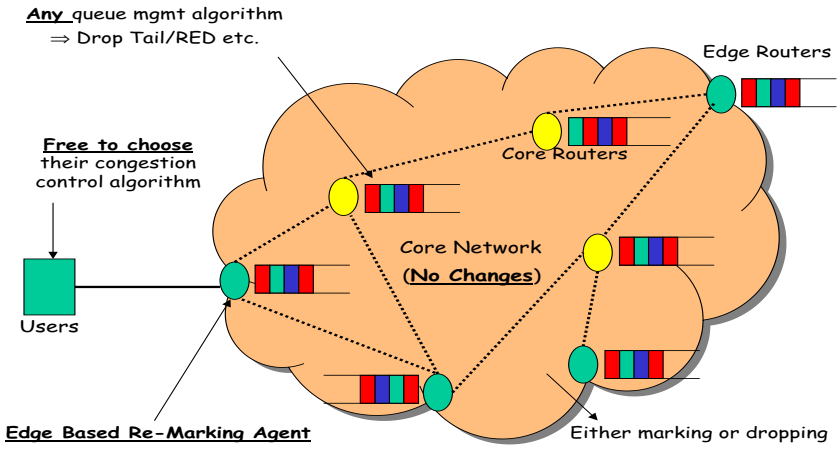


Figure 5.2: Model for managing Uncooperative users at Network Edge

of as a new class of "traffic conditioning" techniques, where the "conditioning" can be achieved by manipulating either the feedback or packet stream. Moreover, since the users cannot always be trusted with their rate control schemes, the network has to enforce this trust and the network edge is the first place this trust is enforced. Additionally, this function can combined with the other edge based functions like preventing spams, denial-of-service attacks etc.

We have implemented this framework in NS-2 and evaluated it for various single and multi-bottleneck topologies, for both marking and dropping congestion notification policies and also with and without AQM schemes. Our results show that the framework can "re-map" any uncooperative user to co-operative user for a broad range of network scenario. Further, the framework is robust and works well even in the presence of background web-traffic and reverse-path congestion. However, for our our scheme to perform well, we need to estimate user's utility function. Towards this end we also outline and evaluate Linear Least Squares Errors (LLSE) and Non-Linear LSE (Least Squared Error) methods. Our initial results show that these methods are easy to implement and work well, even with a small sample set or in other words they can quickly characterize sources. The chapter also presents results for simple differentiated services which can be derived from the model. Finally, we also compared the performance of CHOKe and BLUE in managing un-cooperative flows.

To summarize, the main contributions of this chapter is that it proposes an edge-based model for managing uncooperative users. The framework is independent of AQM schemes, i.e., it works with both RED or other AQM scheme and Drop Tail queues, with both marking and dropping as congestion notification policies. Further, it maintains state information only about mis-behaving users, at the edge. The framework can also be thought of as a new class of traffic conditioning where conditioning can be achieved by manipulating either the ack or packet stream. This chapter also suggests that *management of uncooperative flows need not be coupled with AQM design.* The model presented in this paper can also prevent traffic volume based denial of service attacks. Service differentiation can also be provided in this framework by mapping sets of users to different ranges of target utility functions. Finally, the paper also illustrates a simple estimation technique for characterizing user's according to their utility functions.

The rest of the chapter examines the policing of uncooperative sources in detail. The organization of the chapter is as follows:

- In Section 5.3 we present the network model, assumptions and motivation for protocol conformance.
- In Section 5.5 we present the edge based re-marking model.
- We present the implementation and simulation setup in Section 5.6 and estimation of utility function is described in Section 5.6.1.
- In Section 5.7 we present the results of the re-marking framework. The model is evaluated with both marking and dropping for single and multi-bottleneck topologies, background traffic and reverse path congestion.
- Finally we present the conclusions and limitations of the model in Section 5.9.

5.2 Network Model, Definitions and Assumptions

Consider a user s, who is described with the help of his rate, x_s, a utility function U_s and the set of links which he uses, *L(s)*. Let the network be identified with links l of capacity C_l and the set of users using a link, l, be given by *S(l)*.

Further, we will assume that the rates are bounded and that the utility functions are *increasing* with rates and *strictly concave*. Formally, the assumptions are stated as

- *A1:* The Utility functions are continuous, strictly concave and increasing in their arguments. Further the rates are bounded by *I: [m_s, M_s]*.
- *A2:* The curvature of U_s are bounded away from 0 on I, i.e. $-U_s^{''}(x_s) \geq 1/\alpha_s > 0$.

Then the flow optimization problem can be defined as users trying to maximize their individual utility functions and the network trying to maximize the resource allocation subject to link capacity constraints. The problem is formally defined as [62]:

$$maximize \sum_{s \in S} U_s(x_s) \tag{5.1}$$

$$subject\ to \sum_{s \in S(l)} x_s \leq C_l, \quad \forall l \tag{5.2}$$

for all $x_s \geq 0$. The solution to this problem is given by the following update rules

$$x_s(t) = U_s^{'-1}(\sum_l p_l) \tag{5.3}$$

$$p_l(t+1) = [p_l(t) + \gamma(\sum_{s \in S(l)} x_s - C_l)]^+ \tag{5.4}$$

where p_l are the dual variables of the problem and can be identified as penalties, price or link loss probability [62, 56, 53].

From the above update rules it follows that both the rate control algorithm and the equilibrium rate can be associated with the utility function user chooses to maximize (equation (5.3, 5.4)). However, given that the *same price* is being communicated by the network, the equilibrium rates can be different, but are still fair within Kelly's utility function framework. Thus even though the network doesn't desire to be perceived unfair, a bias in equilibrium rates can be created by choosing two different utility functions. We now illustrate this through an example.

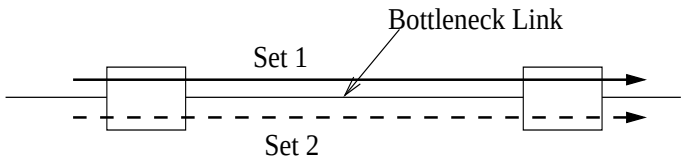


Figure 5.3: Example 1: Two competing set of flows through one bottleneck.

5.3 Motivation

In this section we will illustrate through examples what we mean by unequal bandwidth sharing. Specifically, we present two examples illustrating selfish behavior in a single bottleneck topology and a multi-bottleneck topology.

Example 1. *Consider a bottleneck link where two set of rate control schemes compete for the bandwidth as shown in figure 5.3. The utility function for Set 1 is given by* $U_s(x_s) = w_s log(x_s)$ *and that for Set 2 is given by* $U_s(x_s) = -w_s x_s^{-1}$*, where* w_s *represents the weight assigned to the flow. Let there be 50 sources each in Set 1 and Set 2. Assume that the link capacity to be 300, weights to be 1, the round-trip time (RTT) for all sources to be same. Then the throughput seen by each source can be obtained by solving the following optimization problem:*

$$max \qquad \sum_{i=1}^{50} logx_i \; - \; \sum_{j=51}^{100} \frac{1}{x_j} \tag{5.5}$$

$$subject\ to \quad \sum_{i=1}^{50} x_i + \sum_{j=51}^{100} x_j \leq 300 \tag{5.6}$$

and $x_i, x_j \geq 0 \ \forall i, j$*. Solving this problem yields* $x_i = 4.0, i \in \{1, ..., 50\}$ *and* $x_j = 2.0, j \in \{51, ..., 100\}$.

Thus even though the network is fair, the equilibrium rate depends on the rate control algorithm chosen by the sources. This differentiation in the rates is present because the network conveys the same congestion price to each competing user. (Henceforth we will call such a network as an oblivious network) and users respond differently to the congestion penalties. Another reason for rate differentiation can be attributed to how the users probe the network (or the increase policy). Thus with oblivious network, different final allocations can primarily be associated with

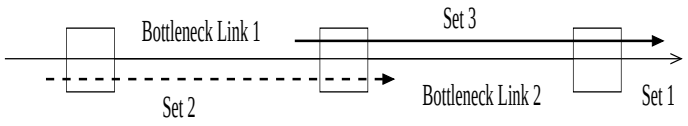


Figure 5.4: Example 2: Three competing set of flows through two bottlenecks.

users rate control schemes.

Example 2. *Figure 5.4 shows a scenario where there are 50 flows in Set 1 traversing both the bottlenecks. Let all of them have the same utility function of* $U_s(x_s) = -x_s^{-1}$. *Then there are two other sets of flows, Set 2 and 3 which go through bottleneck 1 and 2 respectively. Sets 2 and 3 have 50 flows each and their utility function is given as* $U_s(x_s) = log(x_s)$. *We will assume that all the flows have same RTT. Let the capacity of both the bottleneck links be 300 units. Then the final rates are the solution to the following optimization problem*

$$max \qquad \sum_{i=1}^{50} \frac{-1}{x_i} + \sum_{i=51}^{100} logx_i \; + \sum_{i=101}^{150} logx_i \tag{5.7}$$

$$subject\ to \quad \sum_{i=1}^{50} x_i + \sum_{i=51}^{100} x_i \leq 300 \tag{5.8}$$

$$\sum_{i=1}^{50} x_i + \sum_{i=101}^{150} x_i \leq 300 \tag{5.9}$$

Solving the above optimization problem we get the equilibrium rate allocation, $x = \{1.5, 4.5, 4.5\}$ *for the flows in Set 1, 2 and 3 respectively. However, if all the flows in Set 1 use a utility function,* $U_s(x_s) = log(x_s)$ *then on solving the corresponding optimization problem we would get the final rate allocations for Set 1, 2 and 3 as* $x = \{2, 4, 4\}$, *respectively.*

The above examples illustrate that if a subset of flows on the network change their utility function then the rate allocations at the bottleneck change. Thus with oblivious (i.e. which do not differentiate between flows) queue management schemes at the bottleneck e.g. RED the fairness (or the final rate allocation) in the network seems to be solely governed by its users rate control scheme. Also, it can be seen from

the example 2 that by using a slightly aggressive utility function, $log(x_s)$ instead of $\frac{-1}{x_s}$, the users of Set 1 can significantly alter their rates. In this case the users in Set 1 increased their equilibrium allocations by 33%.

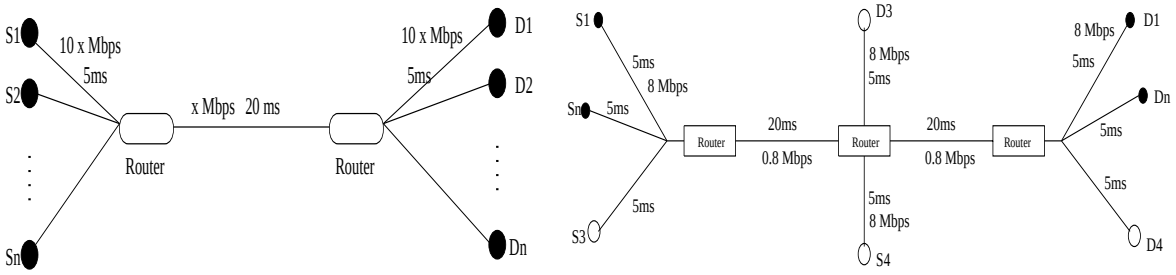
These examples thus illustrate that in an oblivious network, the fairness criteria is dependent upon the utility function chosen by the user. In other words the network does not control the rate distribution of the users and does not enforce any particular fairness criteria. For example, TCP Reno is associated with minimum potential delay fairness while TCP Vegas with proportional fairness [56], however when both TCP Reno and Vegas flows are competing for bandwidth, the final rate allocation is neither minimum potential delay fair nor proportionally fair. This is also illustrated in the above examples when the flows in Set 1 use the utility function $U_s(x_s) = -x_s^{-1}$ (TCP Reno) while the rest use $U_s(x_s) = log(x_s)$ (TCP Vegas), which can be verified to be neither minimum potential delay fair nor proportionally fair. But, if all the competing users deploy the same rate control scheme, e.g. use $U_s(x_s) = log(x_s)$, the final rate allocation as the bottleneck is indeed proportionally fair, as desired.

To summarize the arguments of this section, in presence of queue management schemes which do not differentiate between flows the fairness or the equilibrium rate allocations depend almost entirely on user's rate control schemes. Thus there are clear incentives for selfish behavior. Also the above arguments suggests that fairness might not entirely be network's prerogative, especially if the network does not differentiate between flows. Now we outline the re-marking framework, wherein the network by transforming the congestion penalties can make it appear as if all the users are maximizing the same utility function. Thus, the network by choosing a utility function can provide the fairness associated with that utility function throughout the network.

5.4 Impact of Uncooperative Flows on Existing Buffer Management Algorithms

Though many AQM schemes have been proposed to manage uncooperative flows their deployment on the Internet has been lacking because of variety of rea-

sons: configuration problem, lack of deployment of ECN and requirement of significant network upgrade. As a result of these deployment constraints, the present Internet works on simple Drop-Tail queueing. In this section we evaluate the effect on uncooperative flows on the buffer management schemes and motivate the need for our work.



(a) Single Bottleneck Topology (b) Multi-Bottleneck Topology

Figure 5.5: Topologies used in the Simulations.

5.4.1 Uncooperative Flows and AQM Schemes

Many AQM schemes have been proposed to limit the effect of uncooperative flows. These proposals can be broadly classified into two categories: state-full schemes like FRED [60] etc and stateless schemes like CHOKe [77], BLUE [28]. State-full schemes also include some partial state schemes like RED-PD [64] where states for only the mis-behaving sources are stored. Each of these proposals has it's own merits; stateless schemes are easy to manage while state-full schemes patrol uncooperative flows more efficiently but do not scale. However, given the number of AQM proposals it is beyond the scope of this paper to do an exhaustive performance evaluation across all schemes, hence we will only evaluate CHOKe and BLUE as they represent the stateless alternatives to this work.

We evaluated CHOKe and BLUE on NS-2 on various single and multi bottleneck topologies with different degrees of flow multiplexing. However, we will only present the results for multi-bottleneck scenario. The multi-bottleneck topology is shown in Fig 5.5 b) and the AQM settings are described in Section 5.6. For this setup, we define long flow as a flow which traverses both the bottleneck, whereas the short flows are defined as flows traversing only one bottleneck. Since limita-

tions of CHOKe with unresponsive flows has already been outlined in [64], for our simulations we will evaluate CHOKe (and BLUE) with responsive uncooperative flows. For our simulations the uncooperative flows were generated using BCCS with $k + l < 1$. There was one long and one short flow on each bottleneck and the short flows were mis-behaving, $k = 0, l = 0.5$.

Fig 5.6 (a)-(c) plots the throughput of each flow as well as the ideal share from each simulation while Fig 5.6 (e)-(g) shows the link utilization for the same simulation. Since we have chosen TCP-Friendliness as our definition of cooperation the ideal shares correspond the simulation where both the long and short flows were TCP flows. It can been seen from Fig 5.6 b) that CHOKe marginally improves the throughput of long flow as compared to that with RED, Fig 5.6 a). But more importantly this marginal improvement in performance of CHOKe comes at the expense of link utilization, i.e. the link utilization is almost 30% less with CHOKe (Fig 5.6 e, the thick curve in this plot is the average utilization). On the other hand, BLUE does even worse than RED and the long flow is further penalized as it's throughput goes down. Moreover BLUE also does not utilize the link efficiently, Fig 5.6 f), though it's better than CHOKe. Table 5.1 shows a similar results when the number of flows on each bottleneck was increased to 10 (5 long and 5 short flows), the bottleneck capacity increased to 10Mbps and a buffer of 150 packets. Again it can be seen that marginal improvement in performance of CHOKe comes at the expense of significantly low link utilization (of 70%). Figure 5.6 d) plots the results with our framework and shows that our framework can improve fair sharing of the bottleneck without compromising link utilization.

One of the reasons why CHOKe's performance suffers is because it has poor estimate for the aggressiveness of the uncooperative flow. For every incoming packet to the queue, CHOKe picks a random packet from the queue and matches it's header. If the headers match then CHOKe drops both the packets otherwise it probabilistically enques the incoming packet. Thus if the selfish behavior of the uncooperative flows can be classified properly then depending upon the aggressiveness CHOKe can pick n packets from the queue to match the header. Such a method will then greatly improve the fair sharing of the bottleneck. Our proposal does better precisely be-

Type	Ideal	RED	CHOKe	BLUE
Long Flow (S1-D1)	132	82	95	63
Short Flow (S3-D3)	340	390	300	430

Table 5.1: Performance of AQM Schemes: Comparison of throughput (packets/sec) of Different AQM Schemes on a multi-bottleneck topology with 10 flows on each bottleneck.

cause of this reason. At the edge of the network we measure the loss probability and rate of the uncooperative users and use it to decide the penalty transformation (Section 5.5 presents these arguments in detail).

We also ran simulation with partial network upgrade, i.e. setups where CHOKe was turned on only one bottleneck router while the other bottleneck had Drop Tail queueing. We found performance of CHOKe in partial upgrade to be similar to that of CHOKe on both bottlenecks. However, on a single bottleneck topology CHOKe does remarkably well and the all flows share bandwidth fairly though link utilization remains poor. In yet another set of simulations we enabled ECN on the network and also modified CHOKe to mark packets instead of dropping them. Since our sources were closed loop schemes we expected CHOKe to limit the rates of uncooperative sources. However, the results were most surprising as CHOKe performed even worse than RED.

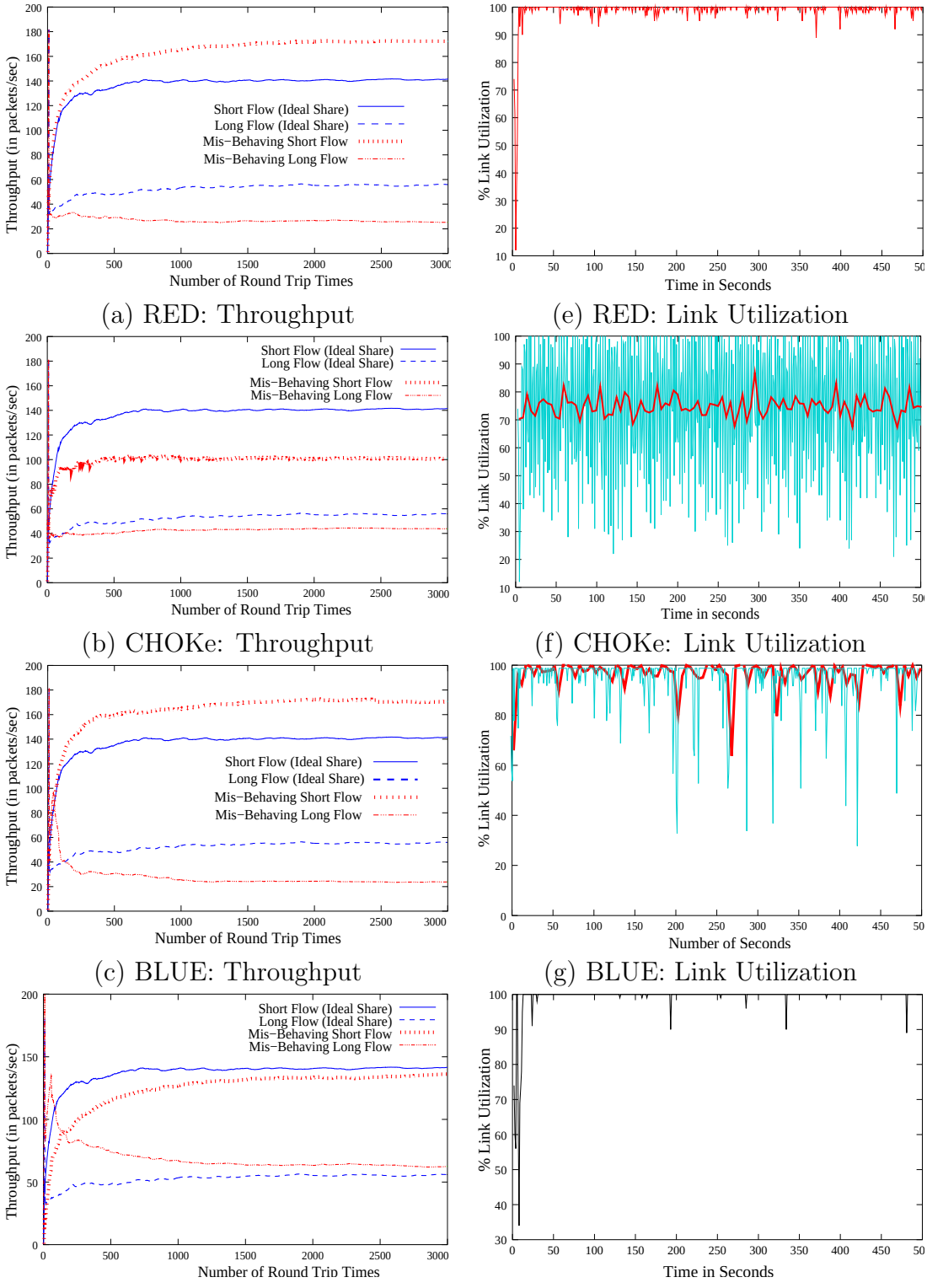
In summary, CHOKe performs remarkably well in patrolling uncooperative users over single bottleneck scenarios. However, it's performance is only marginally better than RED on multi-bottleneck scenarios and it also results in poor link utilization. These wide fluctuations in link utilization suggests oscillations in the bottleneck queue size which in turn cause window (or rate) oscillations. These oscillations are considered harmful as they increase jitter and make any kind of buffer or resource provisioning harder. Thus CHOKe and BLUE cannot always patrol uncooperative flows, especially under multi-bottleneck scenarios and also result in poor link utilization.

5.4.2 Uncooperative Flows and Drop-Tail Queues

Since AQM schemes require significant network upgrade, network providers have not turned on these proposals on the routers. As a result, the present Internet still works with simple FIFO queuing. In this section we will present the impact of uncooperative flows on a network of Drop-Tail queues.

Fig 5.7 shows the shares of a long and short flow on a multi-bottleneck topology. The simulation set-up is similar to the one described above with one long and one short flow. It can be seen from figure 5.7 a) TCP-Friendly is almost shut out by the mis-behaving flows, who now get all the bandwidth. Not only is the TCP-Friendly flow forced into multiple timeouts (23 for this case) but these timeouts occur with very small windows and are often back to back. This result is also indicative of traffic volume based denial-of-service attacks on cooperative users. Similar results were obtained with a higher multiplexing (of flows) and with single bottleneck scenarios and some of them are reported in [17].

To summarize, with DropTail queues uncooperative flows may get significant share of the bandwidth, almost to the extent of shutting out cooperative flows. This might also be construed as denial-of-service to the TCP flows [58, 42]. Thus given that AQM proposals are yet to be deployed on the network and presence of simple FIFO queueing uncooperative flows not only get more than their fair share but may also lead to denial-of-service to conformant flows. As such we are presented with the following question: *What are the appropriate alternate architectural responses for managing a network of un-cooperative users, such that it requires minimal network support ?* Moreover, *as ECN and AQMs are eventually deployed on the network, do these solutions still work ?* In the following section we present our framework which addresses these questions.



(a) RED: Throughput

(e) RED: Link Utilization

(b) CHOKe: Throughput

(f) CHOKe: Link Utilization

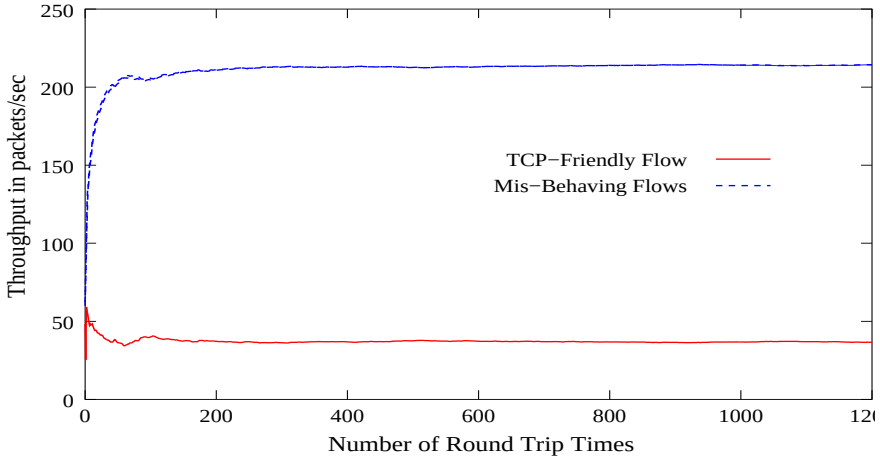
(c) BLUE: Throughput

(g) BLUE: Link Utilization

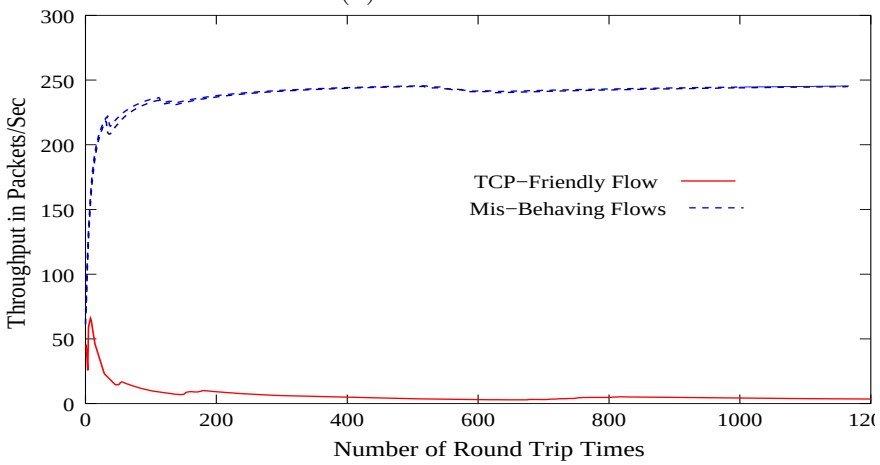
(c) Our Proposal: Throughput

(g) Our Proposal: Link Utilization

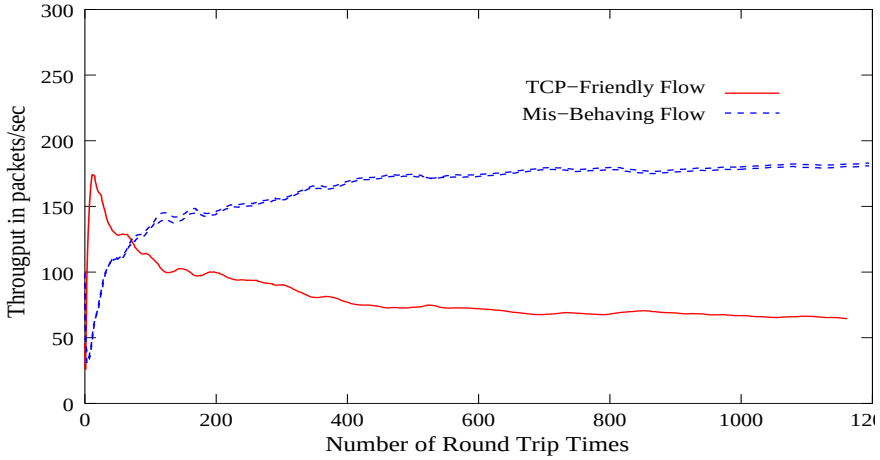
Figure 5.6: Multi Bottleneck: Throughput of long TCP-Friendly flows and short uncooperative flows (k=0,l=0.5) flows with different buffer management schemes.



(a) Ideal Share



(b) Impact of Uncooperative Flow



(c) With our Proposal

Figure 5.7: Performance of Drop-Tail Queueing: Throughputs (in pkts/sec) for two competing flows on a multi-bottleneck setup, long flow is TCP Friendly while the short flows are uncooperative.

5.5 Re-marking Framework for Managing Non-Conformant Users

From the discussion in the previous section it follows that sources can choose rate control schemes which yield higher rate allocations. Another important point to note is that even though the sources are cooperative (i.e. they react to congestion indication) they still can be unfair. As such it is imperative to decouple the fairness criteria from the user's rate control scheme and let the network decide the fairness. That way the network not only has the flexibility of being fair, but more importantly it can choose the fairness criteria it wants to provide. Now we shall describe the re-marking framework to manage non-conformant users.

Lets assume that the users are maximizing the utility function U_s and that the network decides that the final equilibrium rate allocation should be, as if every user chose to maximize a utility function of U_{obj}. Then the rate updation algorithm (and thus equilibrium rates) of the users is given by equations (5.3,5.4). Now, if we communicate a link price $f(p_l)$, instead of p_l, then the user-rate updation algorithm will be

$$x_s(t) = U_s^{'-1}(\sum_{l \in L(s)} f(p_l))$$

Further, if we choose $f(p_l) : f(p_l) \geq 0$, $\forall p_l$, $f(0) = 0$ and the following condition holds true

$$\sum_{l \in L(s)} f(p_l) = U_s^{'}(U_{obj}^{'-1}(\sum_{l \in L(s)} p_l)) = g(\sum_{l \in L(s)} p_l) \tag{5.10}$$

then the rate updation algorithm algorithm becomes

$$\begin{aligned} x_s &= U_s^{'-1}(\sum_{l \in L(s)} f(p_l)) && (5.11)\\ &= U_s^{'-1}[U_s^{'}(U_{obj}^{'-1}(p^s))] && (5.12)\\ &= U_{obj}^{'-1}(p^s) && (5.13) \end{aligned}$$

where $p^s = \sum_{l \in L(s)} p_l$. From the above equation it is easy to see that by communicating a different price we can transform the user's utility function from $U_s(x)$ to

$U_{obj}(x)$. This transformation can be explained by the following modified dual:

$$D(p) = \underbrace{min}_{p \geq 0} \sum_{s \in S} U_s(x_s) - \sum_l f(p_l)(\sum_{s \in S(l)} x_s - C_l) \tag{5.14}$$

where $f(p_l)$ is defined by equation 5.10. Next we will show that a unique solution exists for the modified dual, but before that we prove the following proposition.

Proposition 1. *Given the non-negativity constraint on x_s and p_l and strictly concave utility functions U_s and U_{obj}, the function $g(\sum_{l \in S(l)} p_l)$, $f(p_l)$ as defined in (5.10) are non-negative and strictly increasing in their argument.*

Proof. Define $p^s = \sum_{l \in S(l)} p_l$. Note $g(p^s) = U_s^{'}(U_{obj}^{'-1}(p^s))$. Recognizing that $U_{obj}^{'-1}(p^s)$ is just x_s from equation (4), we can rewrite $g(p^s)$ as $g(p^s) = U_s^{'}(x_s(p^s))$. Since $U_s(x_s)$ is increasing and strictly concave in its arguments hence $U_s^{'}(x_s) \geq 0$. Hence, $g(p^s)$ is greater than 0.

Let's define $F(p^s) = U_{obj}^{'}(p^s)$ and it's inverse as $H(p^s) = F^{-1}(p^s)$. Therefore, $H(F(p^s)) = p^s$.

Now differentiating both sides with respect to p^s we get,

$$\begin{aligned} H^{'}(F(p^s)) \cdot F^{'}(p^s) &= 1 & (5.15) \\ or \qquad (U_{obj}^{'-1}())^{'} &= \frac{1}{U_{obj}^{''}(p^s)}. & (5.16) \end{aligned}$$

Now, differentiating $g(p^s)$ with respect to p^s we get

$$\begin{aligned} g^{'}(p^s) &= U_s^{''}(U_{obj}^{'-1}(p^s)).(U_{obj}^{'-1}(p^s))^{'} \\ &= U_s^{''}(\cdot)(U_{obj}^{'-1}(\cdot))^{'}. & (5.17) \end{aligned}$$

Since U_s and U_{obj} are strictly concave therefore $U_s^{''}(), U_{obj}^{''}() < 0$ and from equation (5.16) we conclude that $g^{'}(p^s)$ is greater than 0. Combining $g^{'}(p^s) > 0$ and the definition of $f(p_l)$ (equation 5.10) we conclude $f^{'}(p_l) > 0$.

□

Theorem 1. *The modified dual represents a non-linear optimization problem where the objective function is as if every user is maximizing a utility function of U_{obj}*

subject to the capacity constraints. Moreover, if the objective utility function is strictly concave then a unique maximizer exists.

Proof. The transformation or the re-mapping function, $U_s'(U_{obj}^{'-1}(p))$, can also be explained as the solution to the following set of equations:

$$\sum_{s \in S(l)} x_s \leq C_l, \quad \forall l \tag{5.18}$$

$$p_l(\sum_{s \in S(l)} x_s - C_l) = 0 \tag{5.19}$$

$$U_s^{'}(x_s) = g(\sum_{l \in L(s)} p_l) \tag{5.20}$$

$$p, x \geq 0 \tag{5.21}$$

Then using equation 5.10 we can rewrite equation 5.20 as

$$U_{obj}^{'}(x_s) = \sum_{l \in L(s)} p_l \tag{5.22}$$

Then equations (5.18-5.22) are the KKT conditions for the following strictly concave maximization problem

$$\underbrace{max}_{x} \quad \sum_{s \in S} U_{obj}(x_s) \tag{5.23}$$

$$\sum_{s \in S(l)} x_s \leq C_l, \quad \forall l \tag{5.24}$$

$$x \geq 0 \tag{5.25}$$

Then using assumption A1 we conclude that the objective function (equation 5.23) is strictly concave and hence an unique solution exists.

□

Then, using the KKT conditions and the gradient projection method we get the following rate and price updation rules

$$x_s = U_s^{'-1}(\sum_l f(p_l)) \tag{5.26}$$

$$p_l(t+1) = [p_l(t) + \gamma \frac{\partial}{\partial p_l} f(p_l)(\sum_{s \in S(l)} x_s - C_l)]^+ \tag{5.27}$$

The above formulation is however difficult to implement because it requires per-flow queuing and that too inside the network. Since upgrades in the network are hard to achieve consider the following update rule.

$$p_l(t+1) = [p_l(t) + \gamma(\sum_{s \in S(l)} x_s - C_l)]^+ \tag{5.28}$$

$$x_s(t+1) = U_s^{'-1}(\sum_l f(p_l(t))) \tag{5.29}$$

Next we establish that the update rule presented above converges to the optimal point. But before that we prove that $D(p)$ is lower bounded, continuously differentiable and convex and $\nabla D(p)$ is Lipschitz continuous.

Proposition 2. *Under Assumptions A1 $\nabla D(p)$ is Lipschitz.*

Proof. Define by A the incidence matrix where A_{ls} is 1 if source s uses link l and 0 otherwise. Further let the total number of links used by any source be bounded by L and the total number of sources by S. Then after some simplification we have

$$\frac{\partial x(p)}{\partial p} = diag\left(\frac{1}{U''_{obj}(x_s(p))}\right) A^T \tag{5.30}$$

Also from equation (5.14) we get $\nabla D = f^{'}(p)(C - Ax)$. Differentiating it again with respect to p_l we get

$$\nabla^2 D = f^{''}(p)(C - Ax) + f^{'}(p)(-A\frac{\partial x(p)}{\partial p}) \tag{5.31}$$

After some simplification $f^{''}(p)$ can be calculated as

$$f^{''}(p) = \frac{U_s^{3'}(x(p))}{\alpha_s^2} + \frac{\alpha_s}{U_{obj}^{3'}(x(p))} \tag{5.32}$$

Since the utility functions are strictly increasing in their arguments hence they will be rightly skewed, i.e. $U_s^{3'}$ is bounded away from 0. Further since the rate are bounded by I (Assumption A1) the second derivative of *f(p)* will be bounded, let

us say that this bound is F. After some simplification the bound on $f^{'}(p)(-A\frac{\partial x(p)}{\partial p})$ can be calculated as βLS (for some $\beta > 0$ and β function of $\alpha_s(> 0)$). Then using the capacity constraint we conclude that ∇D will be Lipschitz with the following bound

$$\|\nabla D(q) - \nabla D(p)\| \leq (FC + \beta LS)\,\|q - p\|$$

□

Proposition 3. *Under assumption A1 D(p) is lower bounded, continuously differentiable and strictly convex.*

Proof. By Assumption (A1), U_s is bounded and continuously differentiable thus $U_s^{'-1}$ (and f(p)) exist and is also bounded and continuously differentiable. Therefore, D(p), as defined in equation (5.14) is also lower bounded and continuously differentiable.

Further from the assumption A1 (strictly increasing and strictly concave utility function) and equation (5) $f^{''}(p)$ (as defined in equation (5.32)) will be greater than 0. Using this knowledge and the capacity constraint the first term in equation (5.31) is always greater than or equal to 0. The second term of equation (5.31) is

$$f^{'}(p)(-A\ diag\left(\frac{1}{U_{obj}^{''}(x_s(p))}\right)A^T)$$

also strictly positive because from Proposition 1, $f^{'}(p)$ is always greater than 0, the incidence matrix is a 0-1 matrix and the utility functions are strictly concave. Thus we can say that equation (5.31) is greater than 0. Or $\nabla^2 D(p) \geq 0$ and *D(p)* is strictly convex.

□

Proposition 4. *Given the non-negativity constraint on x_s and p_l and strictly concave utility functions U_s and U_{obj}, the new update algorithm as defined in equations (5.28, 5.29) converges to the optimal point.*

Proof. Using the equation 5.14 and differentiating it with respect to time we get

$$\frac{d}{dt}D(p) \;=\; \frac{d}{dp_l}\left(f(p_l)[C_l - \sum_{s\in S(l)} x_s]\right)\frac{d}{dt}p_l$$

$$\begin{aligned} \frac{d}{dt}p_l &= \gamma(\sum_{s\in S(l)} x_s - C_l),\ \gamma > 0 \\ Thus,\ \frac{d}{dt}D(p) &= -\gamma\frac{d}{dp_l}f(p_l)[C_l - \sum_{s\in S(l)} x_s]^2 \end{aligned}$$

Since, $f^{'}(p_l) > 0$ and $\gamma > 0$ we can establish that $D(p_l(t))$ is a decreasing function in t. Also since D is strictly convex (see Proposition 3), there exists a minima, and $\frac{d}{dt}D(p(t)) \leq 0$ implies convergence to the optimal point.

□

Theorem 2. *Assume that utility functions, U_s, are increasing, strictly concave and continuously differentiable, and their curvature is bounded away from 0. Then starting from any initial rates in the interior of* X *and prices* p(0) ≥ 0*, every accumulation point (x^*, p^*) of the sequence (*x(t), p(t)*) generated by the above algorithm and equations (5.28,5.29) is primal dual optimal.*

Proof. By Propositions 3 and 4 the dual objective function $D(p)$ is convex, lower bounded and $\nabla D(p)$ is Lipschitz, then any accumulation point p^* of the sequence $\{p(t)\}$ generated by the gradient projection algorithm is dual optimal [11]. Moreover, the constraints are linear and the primal problem is strictly concave hence there is no duality gap. Therefore dual optimal is also primal optimal.

□

Thus this update rule minimizes the dual function and converges asymptotically. The above update rule also does not change the core network, as we retain the price update rule as proposed in [62]. Further, the price being communicated to the user can be updated at the edge. We now state the algorithm for the edge re-marker as

Edge Marker's Algorithm:

- For each source, receive from the network the total price for the source's traffic as $p^s(t) = \sum_{l\in S(l)} p_l(t)$.

- Recalculate (or Re-mark) the new price for the source as

$$p^s_{new} = g(\sum_{l \in S(l)} p_l(t)).$$

- Communicate this *re-marked* price to the source.

The update algorithm for the network and the source are given by equation (5.28) and (5.29) respectively.

Finally, we end this section with the following remark on convergence of the algorithm.

Proposition 5. *The rate of convergence of the algorithm is given by the smallest eigen vector of ABA^t where A is the routing matrix and B is $diag(U'^{-1}_{obj}(p^*))'$ and p^* is equilibrium price.*

Proof. In Theorem 1 we showed that by our penalty transformation the original optimization problem was translated into as if all the users were maximizing a utility function of U_{obj} and the update algorithm can be written as

$$\frac{\partial p(t)}{\partial t} = -\gamma(\mathbf{Ax} - \mathbf{C}) \tag{5.33}$$

$$\mathbf{x} = \mathbf{U_{obj}}'^{-1}(\mathbf{p^t A}) \tag{5.34}$$

where $\mathbf{p}$ is the vector of dual variables or price in our framework. Since the objective function is strictly concave a unique maximizer exists, let this maximizer be called p^*. Let $p(t) = p^* + \xi(t)$. Then linearizing the system of equations (5.33,5.34) about p^* and after some simplification we get

$$\frac{\partial(p^* + \xi(t))}{\partial t} = -\gamma\left(\mathbf{A}\mathbf{U_{obj}}'^{-1}((\mathbf{p}^* + \xi(\mathbf{t}))^{\mathbf{t}}\mathbf{A}) - \mathbf{C}\right) \tag{5.35}$$

$$\frac{\partial \xi(t)}{\partial t} = -\gamma \mathbf{A} B \mathbf{A}^t \tag{5.36}$$

$$B = diag(\mathbf{U_{obj}}'^{-1}(p^*))' \tag{5.37}$$

Thus the rate of convergence of the algorithm is given by the smallest eigen vector

of ABA^t. □

5.6 Implementation

We implemented the edge based re-marker in the NS (Network Simulator). The edge based re-marker was tested for two scenarios, one when network marks packets and second when it drops packets. For the case where the network marks packets, the edge based re-marker was placed on the reverse path (i.e. on the reverse access link of the user) and re-marked the ACKs. However when the network is dropping packets, the edge based re-marker was placed on the forward path at the network egress and dropped packet. Also, in either cases the edge re-marker estimated the loss rate for each flow and subsequently used it to re-mark or drop acks or packets. We also assumed that we know the utility functions of all the flows. However, later we detail a procedure to estimate the utility functions of users at network edges.

For our simulation we used the congestion control and loss recovery mechanisms of TCP New Reno. Also, we disabled the delayed acknowledgments option. In our simulations we have assumed TCP Friendliness as the conformance criteria. Thus all rate control schemes whose utility function is given by $U_s = \frac{-1}{x}$ are called conformant (or compliant or TCP Friendly). For simulating mis-behaving (or selfish) flows we used the Binomial Congestion Control scheme (BCCS) proposed in [9]. As explained in Chapter 4 the BCCS is described by the window increase and decrease parameters: α and β respectively and the window increase and decrease scaling factors, k and l respectively. For our simulations, we fixed the values of α, β as 1 and 0.5 respectively. Also, as shown in Chapter 4 the utility function for Binomial schemes is defined as $U(x) \propto \frac{-1}{x^n}$, where $k+l=n$. Thus conformant or TCP Friendly flows are described by $k+l = 1$. Since the network allocates more resources to user's whose marginal utility are higher, non-conformant or misbehaving flows can be generated by choosing $k + l < 1$. Henceforth, we will use the k and l values to identify mis-behaving flows.

Figure 5.5(a) shows the single bottleneck topology used in the simulations. The access links were configured at a rate 10 times greater than that of the bottleneck link. All the links use Random Early Drop (RED) queues with min thresh and max

thresh set as buffer/3 and 0.8*buffer respectively, where buffer is the total bottleneck buffer length. Further, the weight was set as 0.002 and the marking probability for RED was set to 0.1. The RTT was 60ms and the packet size 500B.

Figure 5.5(b) shows a multi-bottleneck topology used in the simulation. The bottleneck buffer was set to 25 packets. We also evaluated our framework for another multi-bottleneck setup of bottleneck link of 10 Mbps, access link of 100 Mbps and a buffer of 250 packets. The link delays were kept the same. RED minimum and maximum threshold settings were similar to those of single bottleneck. Also for all the simulation setups (single or multi-bottleneck) the access link rate are always 10 times greater than that of the bottleneck link.

The maximum advertised window is set sufficiently high so that it does not constrain the actual window. The simulation time for each setup was 1500 seconds. We plot the throughput of competing flows in packets/sec, averaged over 20 round-trip times. We assumed that all the flows have infinite data to transfer.

5.6.1 Estimating the Utility Function

The uncooperative congestion control framework works well if the network knows the utility function of the non-cooperative flows. Thus estimating the utility function parameters is of paramount importance. In this section we briefly describe ways to estimate the utility function.

For the results presented in this chapter we have chosen BCCS schemes as non-cooperative users. These schemes can generally be described by their exponent, n, by the following relationship

$$U(x) \propto \frac{-1}{x^n} \tag{5.38}$$

Thus, for describing these class of selfish flows we just need to estimate the parameter, n. For that purposes consider the following relationship between the rate and the loss probability

$$U_s^{'}(x_s) \propto p \tag{5.39}$$

$$p \propto \frac{n}{x^{n+1}} \tag{5.40}$$

$$log(p) = log(nK) - (n+1)log(x) \tag{5.41}$$

where K is some constant. It is interesting to note that estimating the parameter n is nothing but a regression analysis on the equation (5.41). But for those purposes we would need to have a measure of the throughput x and the loss probability p. These can be calculated by either sampling the packet stream (at the egress) or the ack-stream. If processing the packet-stream we could just count the number of packets sent and lost in a specified time. Using this time-series a Linear-Least Squared Errors (LLSE) method could be applied to estimate n. In this thesis we employed LLSE to estimate the parameter n and present the results in Section results.

For a more general utility function as defined in equation (4.18) we could employ the Non-Linear Least Squared techniques to detect a power-series in x *and* n. We are currently working on this estimation problem.

However, for all these estimation to work we would need to identify the non-cooperative users first. We leave this problem as that of future work, but would like to point out to some schemes described in SRED, Stochastic Fair Blue and RED-PD. But irrespective of how we detect these selfish users, we will have to store information (or state) about them at the routers. However, since these will be malicious users the amount of state would not be large. Moreover, it is further constrained by the fact that this state information is kept *only* at the edge routers and *nothing* needs to be stored in the core. Also, since we are trying to police selfish flows we are inadvertently doing per-flow management where per-flow is the total number of selfish flows in the Internet. Stateless schemes like CHOKe which police the selfish flows enforce a max-min fairness criteria and therefore may end up differentiating flows on RTTs and also in a multi-bottleneck scenario do not enforce the protocol compliance criteria (like TCP Friendliness).

5.6.2 Estimating the Loss Rate

The scheme proposed in this thesis is sensitive to loss-estimation. If we under-estimate the loss rate, then we would be not penalizing the selfish user enough and consequently there will be unfair rate allocation in the network. On the other hand over-estimation of losses leads to the scenario where selfish user's are over penalized and get a lesser share of the bandwidth, even less than the conformant users.

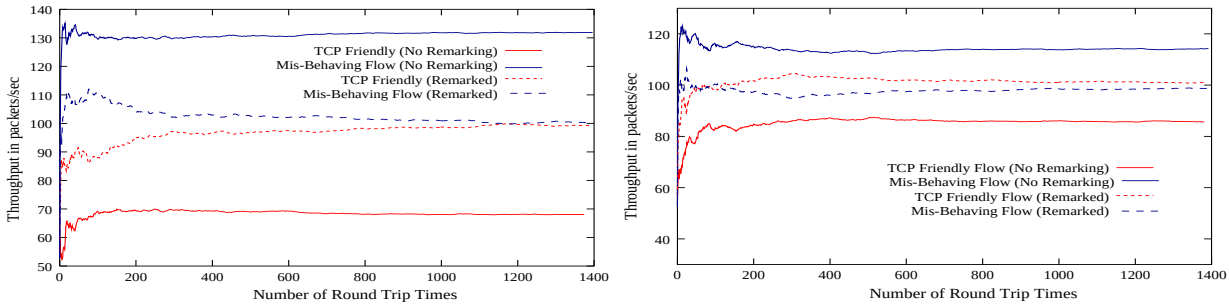
In [40] the authors propose two methods for estimating losses: Exponential Weighted Moving Average (EWMA) and the Weighted Average Loss Indication (WALI). In the EWMA proposal the losses are averaged infinitely over time while in WALI the authors average a fixed window of loss event with higher weights to the current loss events. However, there are no clear guidelines on how to configure EWMA and WALI weights; the values assigned are obtained by trials and some intuition regarding the importance of recent loss events. End-to-end losses in the Internet have been examined using real traffic traces in [98]. Again, the modeling aspect is limited to comparing a fixed window averaging scheme to an EWMA scheme, no attempt is made to find weights for past samples or adapt these weights. The traffic process (not the loss process) has been modeled using an AR model in [100]. The authors in [100] make no attempt to analyze the loss process and restrict the paper to fitting an AR model to traffic traces. In [81] the propose a linear prediction formulation to predict these loss rates. In order to adapt the loss rates to the samples as they arrive, the author has suggested to use the Recursive Least Squares algorithm.

For estimating the losses we have used EWMA and the WALI methods of Equation-Based Rate Control algorithm [40]. We updated these loss indications every RTT and we have assumed that the network knows the RTT of the flows. We present the results for EWMA based loss-estimator. Similar results were obtained with WALI based estimator. For EWMA based system we gave 60% weight to the history, while with the WALI based estimator we measured samples over 8 windows to estimate losses.

5.7 Simulation Results

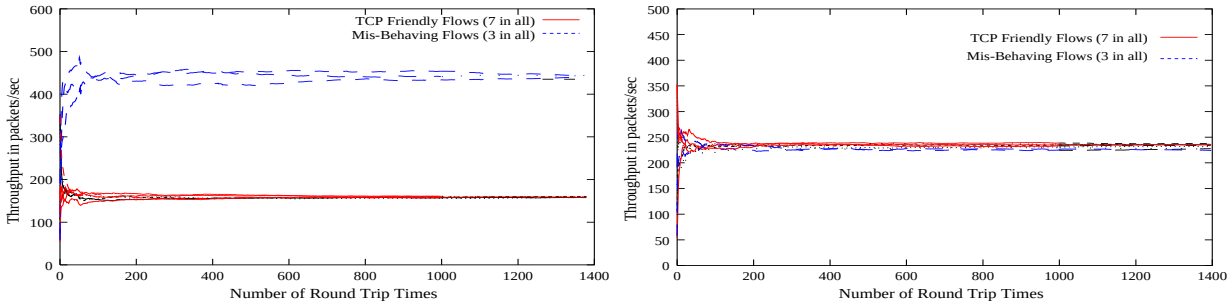
In the following sections we present our simulation results. Our simulation objectives can be stated as

- Validate the model with both single as well as multi-bottleneck topologies with varying degrees of (flow) multiplexing.
- Examine the robustness of the model in presence of background (web) traffic



(a) k=0, l=0.5 (b) k=0, l=0.8

Figure 5.8: Single Bottleneck (Marking): Throughputs (in pkts/sec) for two competing flows, one is TCP Friendly while the other is non-conformant with and without Re-Marking.



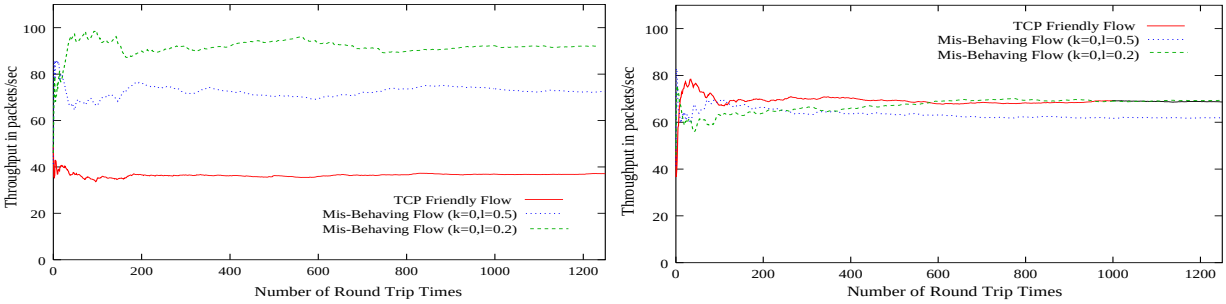
(a) No Re-Marking (b) Re-Marking the non-conformant Flow

Figure 5.9: Single Bottleneck (Marking): Throughputs (in pkts/sec) for ten competing flows, where seven flows are TCP Friendly while three are non-conformant (k=0, l=0.5) with and without Re-Marking.

and reverse path congestion.

- Verify if the model works with dropping as a congestion notification mechanism. Specifically, if it can work with a network of Droptail queues only.
- Substantiate and test how to estimate utility functions.
- Test the sensitivity of the model with respect to inaccurate RTT and utility function estimates.

The result section is organized into two separate sub-sections to evaluate the framework with both marking and dropping. In Section 5.7.1 we present the results of edge-based re-marking framework. For the results in this section, the penalty



(a) No Re-Marking (b) Re-Marking the non-conformant Flows

Figure 5.10: Single Bottleneck (Marking): Throughputs (in pkts/sec) for three competing flows, where one flow is TCP Friendly while the other two are non-conformant with (k=0, l=0.5) and (k=0, l=0.2) resp., with and without Re-Marking.

transformation agents were placed on ingress node in the reverse path and re-marked the Acks. In order to evaluate the network where ECN is not enabled and dropping is used to convey congestion indication, we placed the penalty transformation at the egress nodes on the forward path. These agents conveyed appropriate penalties by dropping packets from the malicious flows. The results with dropping are reported in Section 5.7.2.

5.7.1 Evaluation of Edge Based Re-Marking Framework on an ECN Enabled Network

In this section we present the results of managing selfish behavior on an ECN enabled network. We assume that the network operates with RED queues and their parameter setting are detailed in Section 5.6. We evaluate the framework for both single and multi-bottleneck scenarios, , background traffic and with reverse path congestion.

5.7.1.1 Single Bottleneck

In figure 5.8 a) we present the throughputs of two competing flows on a single bottleneck of 0.8 Mbps with a buffer of 25 packets. Here, one of the flows is TCP, while the other is non-conformant and is defined by $k = 0$ and $l = 0.5$. As the figure 5.8 a) shows, when we do not re-mark the non-conformant flow, it garners more

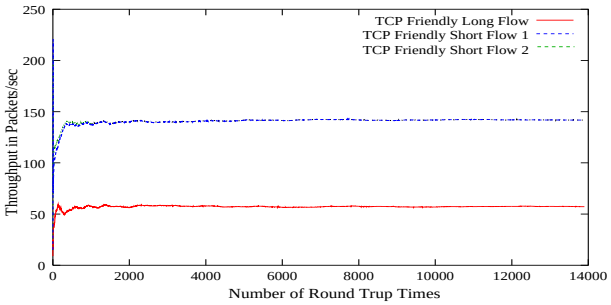
bandwidth than the TCP friendly flow. However, re-marking the non-conformant flow makes the two flows to share the bandwidth equitably. Figure 5.8 b) show similar results where the non-conformant user is defined by $k = 0$, $l = 0.8$.

Figure 5.9 shows the results for a set of 10 competing flows on a 10Mbps bottleneck and 150 packet buffer. The flow set comprises of 7 TCP Friendly flows while the remaining 3 flows are non-conformant and are defined by $k = 0$ and $l = 0.5$. The bandwidth is shared equitably in presence of re-marking, however in absence of re-marking mis-behaving flows easily beat the TCP Friendly flows.

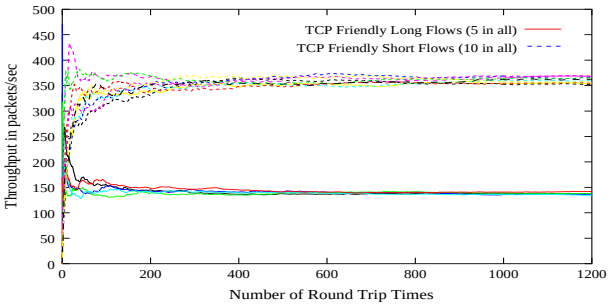
We also evaluated our scheme for a scenarios where every flow has a different utility function. Figure 5.10 shows the result for one such setup for a bottleneck bandwidth of 0.8 Mbps. In the first simulation setup we have three flows, one TCP-Friendly flow and the others are defined as (k=0, l=0.5) and (k=0, l=0.2). We can see from the figure 5.10 that in the absence of re-marking, non-conformant flows beat the TCP friendly flow; however when we re-mark the non-conformant flows the bandwidth is shared fairly. These simulation results also illustrates that the edge based re-marking framework can map the utility functions of the selfish flows to that of TCP, thus making them appear TCP Friendly.

5.7.1.2 Multi Bottleneck Topology

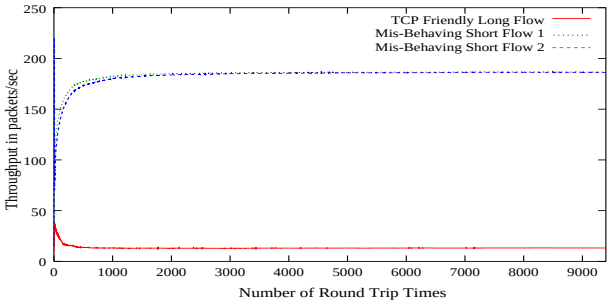
In this section we present the results for multi-bottleneck topology (figure 5.5 b)). We define long flow as a flow which traverses both the bottleneck, whereas the short flows are defined as flows traversing only one bottleneck. In the first simulation setup, where there were 2 flows on each bottleneck, (0.8Mbps, 25 packet buffer), we first measured the optimal rate allocations when all the flows (long and short) are TCP friendly and plot them in 5.11 a). As expected, the short flows grab more share of the bottleneck because they have smaller RTTs and go through a single bottleneck as compared to the long flow. We then changed the short flows to be non-conformant (k=0, l=0.5) and plot the result in figure 5.11 b). The effect of mis-behavior is more pronounced in this case as the non-conformant flows are trying to shut out the TCP friendly flow. However, when we used our model to re-mark the non-conformant flows we see that (figure 5.11 c)) the flows now share the bandwidth



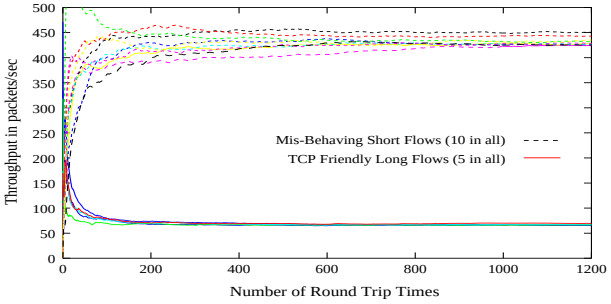
(a) 2 Flows: Every Flow is TCP Friendly Ideal bottleneck Shares



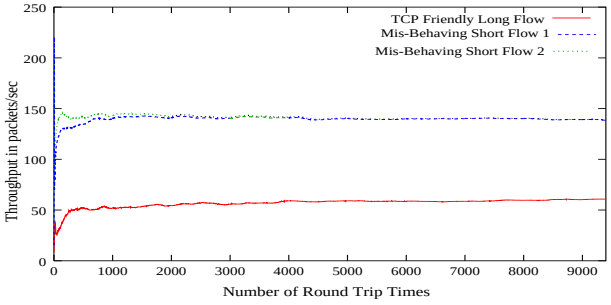
(d) 10 Flows: Every Flow is TCP Friendly Ideal bottleneck Shares



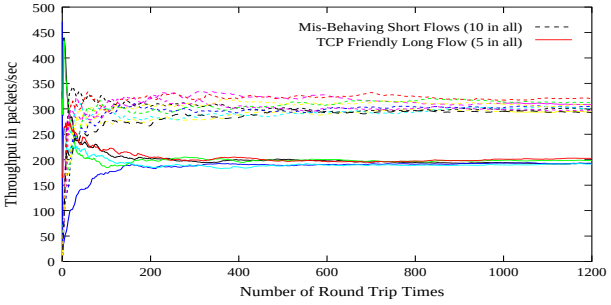
(b) Short Flows are Non-Conformant: No Re-Marking



(e) Short Flows are Non-Conformant: No Re-Marking



(c) Short Flows are Non-Conformant, Re-Marking



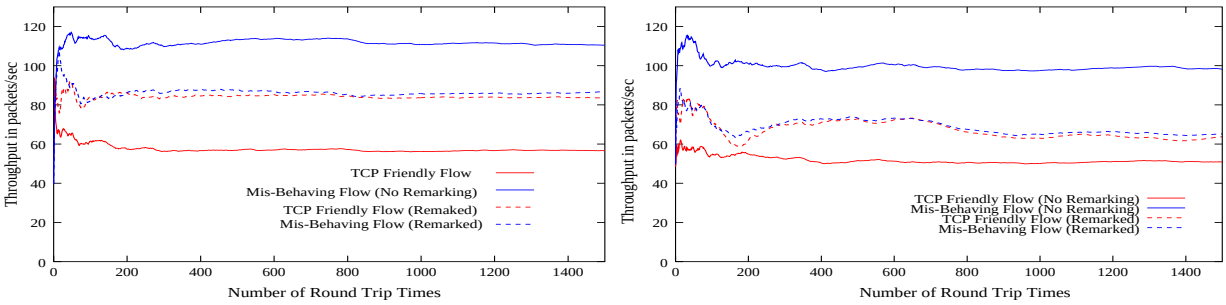
(f) Short Flows are Non-Conformant, Re-Marking

Figure 5.11: Multi Bottleneck (Marking): Throughputs for 2, 10 competing flows.

fairly. More importantly, we see that the result in figure 5.11 c) is very similar to 5.11 a), i.e., *we have successfully mapped the utility function of the non-cooperative flows*. We also simulated the scenario where the long flows were non-conformant and short flows TCP-Friendly and similar results were obtained.

In figures 5.11 d), e) and f) we plot the results for a multi-bottleneck topology (10Mbps, 250 packets buffer) where on each bottleneck there are 5 TCP Friendly

flows and 5 non-conformant flows (k=0, l=0.5). Figure 5.11 (d) plots the throughput of long and short flows, if all of them were TCP Friendly. As expected the longer flows get a smaller share of the bottleneck than the shorter flows. In Figure 5.11 (e), we changed the shorter flows to act as non-conformant flows and plot the throughput, and it can be seen that the non-conformant shorter flows conveniently beat down the TCP friendly flows. However, in presence of re-marking, (Figure 5.11 (f)) the non-conformant flows are conveyed higher price by the edge-re-marker and thereupon share the bottleneck more favorably with the longer flows. Once again, we see that *re-marking tends to achieve the same performance as those as if all the flows were TCP Friendly.*



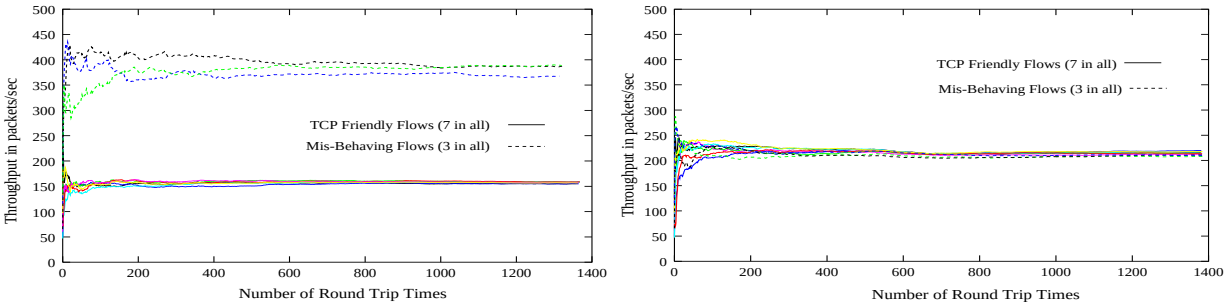
(a) 15% noise (b) 25% noise

Figure 5.12: Background Traffic (Marking): Throughputs (in pkts/sec) for two competing flows in a single bottleneck topology, where one flow is TCP Friendly while the other is Non-Conformant with (k=0, l=0.5). Also there is web-traffic in the background.

5.7.1.3 Background Traffic

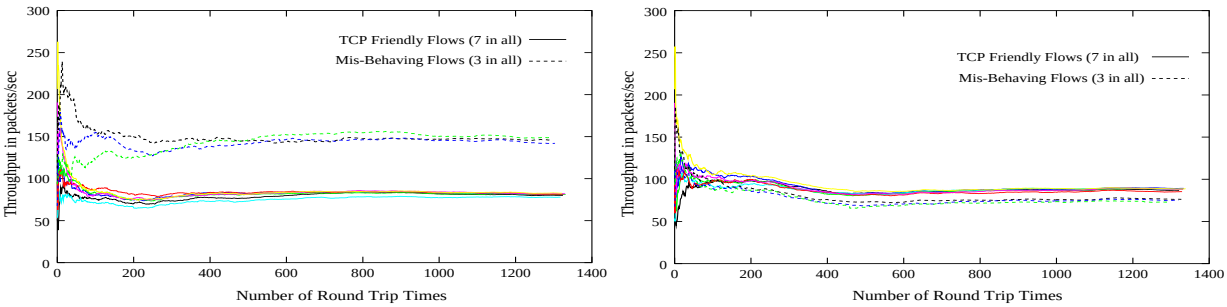
In this section we evaluate the framework in presence of noise-like mice traffic. HTTP sources were added to the persistent non-conformant and conformant sources. Each http page sends a single packet request to the destination, which then replies with a file of size which was exponentially distributed with 12 1Kb packets. After a source completes this transfer it waits for a random time, which was exponentially distributed with a mean of 1 second and then repeats the process.

Two sets of simulations were conducted for the single bottleneck case. In the first simulation, there were two persistent flows (one non-conformant and the other



(a) No Re-Marking (b) Re-Marking

Figure 5.13: Background Traffic (Marking): Throughputs (in pkts/sec) for 10 competing flows in a single bottleneck topology, where 7 flows are TCP Friendly while the other 3 are Non-Conformant with (k=0, l=0.5) with 20% noise.



(a) No Re-Marking (b) Re-Marking

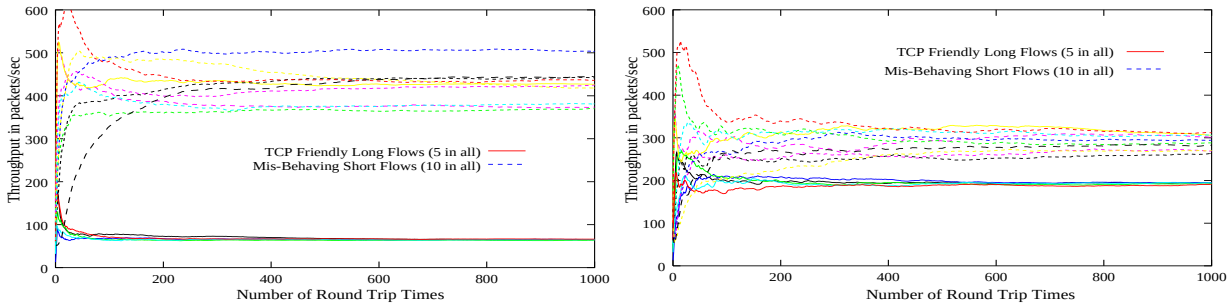
Figure 5.14: Background Traffic (Marking):Throughputs (in pkts/sec) for 10 competing flows in a single bottleneck topology, where 7 flows are TCP Friendly while the other 3 are Non-Conformant with (k=0, l=0.5) with 65% noise.

TCP Friendly) competing for a bottleneck of 0.8 Mbps. 2 and 4 http sources were added to generate 15% and 25% noise (i.e. the http sources occupied 15% and 25% of the bottleneck bandwidth). The results for this simulation are plotted in figure 5.12. Again, it can be seen that the re-marking works well in the presence of noise and the bottleneck is shared equitably. In another simulation we increased the number of competing persistent flows to 10 and of these, 7 flows were TCP Friendly while the remaining 3 where non-conformant (k=0,l=0.5). The bottleneck bandwidth for this simulation was 10Mbps and a buffer of size 150 packets. Also in this setup we increased the noise sufficiently high to validate the robustness of

the scheme in presence of many flows and noise. Figures 5.13 and 5.14 plot the results for the cases where the noise traffic is 20% (25 http sources) and 65% (80 http sources) respectively. Figure 5.14 also shows the robustness of the scheme. The re-marker manages to efficiently patrol non-conformant users even when the noise is the network is sufficiently high, (65% noise).

5.7.1.4 Cross Traffic

In this section we present the results for our penalty function transformer when two way traffic is present. We evaluate this scenario with the multi-bottleneck topology, where we have 5 TCP Friendly long flows and 5 non-conformant (k=0, l=0.5) short flows on each bottleneck. Additionally, on the reverse path, there are 5 TCP Reno flows on each bottleneck. The bottleneck bandwidth for this simulation was 10Mbps and a buffer of size 250 packets. Re-Marking, once again achieves equitable sharing of the bottleneck (as shown in Figures 5.15 (a) and (b)).



(a) No Re-Marking (b) With Re-Marking

Figure 5.15: Cross Traffic (Marking): Throughputs (in pkts/sec) for 10 competing flows in a multi-bottleneck topology, where on each bottleneck there are 5 TCP Friendly flows and 5 Non-Conformant with (k=0, l=0.5), with two-way traffic.

5.7.2 Evaluation of Edge Based Re-Marking Framework on a Non ECN Capable Network

Up till now we have discussed the non-conformant framework with re-marking, i.e., we have assumed that ECN support is available in the network. In this section, we look at the alternative scenario, when drops are used to convey congestion penal-

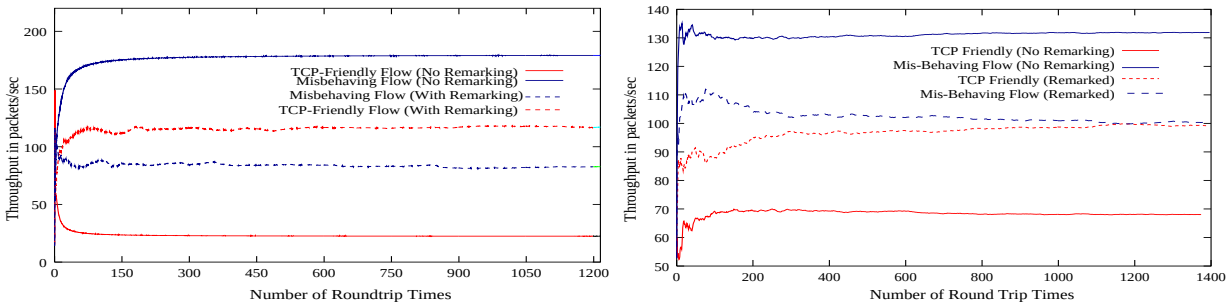
ties. We present the results for two cases, when one the network operates with Drop Tail queues only and then the second where the network works with RED queues. For RED parameter settings the reader is referred to Section 5.6. Again, we test the framework for single and multi-bottleneck scenarios and cross traffic and reverse path congestion.

5.7.2.1 Single Bottleneck

We present the result with a single bottleneck of 0.8Mbps and access links of 8Mbps for 2 competing flows. One of the flows is TCP-Friendly while the other is misbehaving flow (with k=0, l=0.5). Both the flows have same RTT of 60ms. For such a scenario we sampled the packet-stream at the egress router and also placed the re-marker there. The re-marker in this case conveyed the transformed penalties to the mis-behaving flows by dropping its packets. Figure 5.16 a) and b) shows the results of with and without the re-marking framework with Drop Tail and RED queues respectively. It can be seen from the figure 5.16 a) that in a network of Drop Tail queues and absence of re-marking the non-conformant flow gets most of the bottleneck share. Moreover it beats the TCP-Friendly flow comprehensively as against the same simulation setup with RED queues (as shown in figure 5.16 b). However, when we start re-marking the misbehaving flows this bias against the TCP-Friendly is reversed. But, it can be seen from the figure 5.16 a) that now TCP-Friendly flow gets a better share of the bottleneck. This is because unlike marking, dropping is a stricter means to convey congestion notification as it can lead to timeouts. As such the misbehaving flow suffers.

5.7.2.2 Multi-Bottleneck

Figure 5.5 b) show a multi-bottleneck topology with a TCP-Friendly flow traversing both the bottlenecks while one short mis-behaving flow (k=0, l=0.5), each going through one bottleneck. It can be seen from figure 5.17 a) TCP-Friendly is almost shut out by the mis-behaving flows, who now get all the bandwidth. Not only is the TCP-Friendly flow is forced into multiple timeouts (23 for this case) but these timeouts occur with very small windows and are often back to back. Similar results were obtained with a higher multiplexing (of flows) but are not reported here.



(a): DropTail Queues (b): RED Gateways

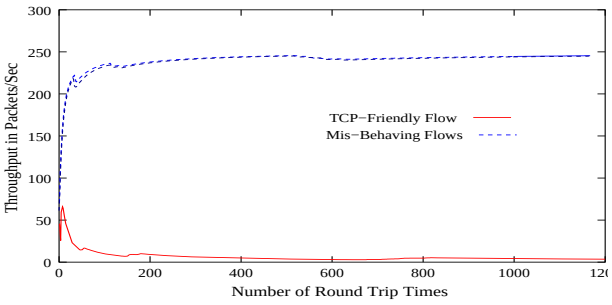
Figure 5.16: Single Bottleneck (Dropping): Throughputs (in pkts/sec) for 2 competing flows on a network of DropTail and RED queues with and without Re-Marking. One flow is TCP Friendly while the other is Non-Conformant (k=0,l=0.5).

In summary, with DropTail queues mis-behaving flows may get significant share of the bandwidth, almost to the extent of shutting out conformant flows.

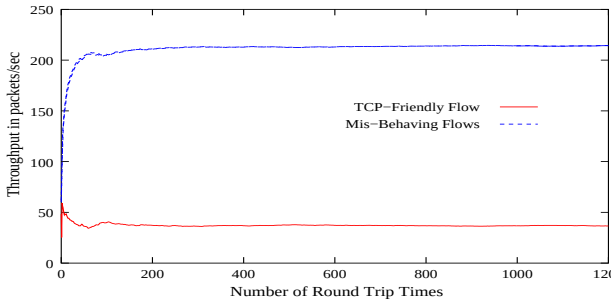
Figure 5.17 b) plots the throughput when instead of DropTail queues we used RED queues at the bottleneck. (The reader is referred to Section 5.6 for RED settings.) It can be concluded from the figures that though RED improves the shares of TCP-Friendly flow, the unfair rate allocations because of mis-behavior of flows persist. This is because RED is an oblivious AQM scheme and therefore allocates equal marks to all users. Then as outlined earlier, the final rate allocations are dependent on the utility function used by the user's and as such an different choices of utility function can cause unfair sharing of the bottleneck. In figure 5.17 c) and d) we plot the results with re-marking enabled in the network, with DropTail and RED queues respectively. Our results suggests that when re-marking is enabled on a network of DropTail queues we can significantly improve the sharing of the bottleneck. On a network of RED queues with re-marking enabled the results are even more appealing thus pointing to virtues of deploying RED in the network.

5.7.2.3 Background Traffic

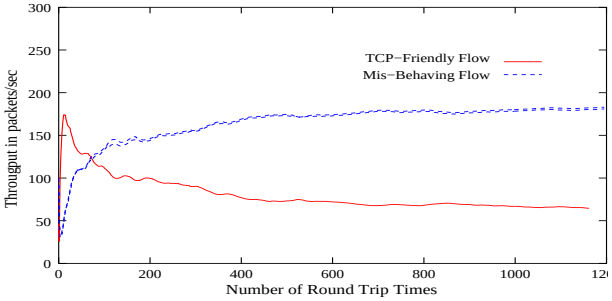
In this section we evaluate the framework in presence of noise-like mice traffic. HTTP sources were added to the persistent non-conformant and conformant sources. The details of these HTTP sources have already been outlined in Section 5.7.1.3 and



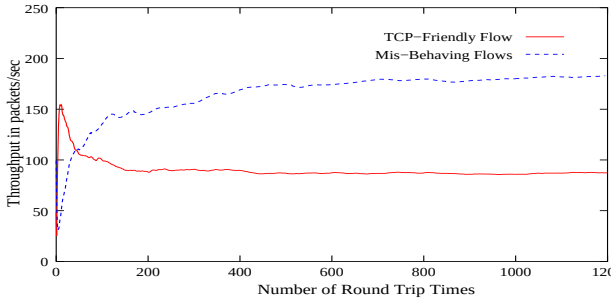
(a): DropTail Queues (No Remarking)



(b): RED Gateways: (No Remarking)

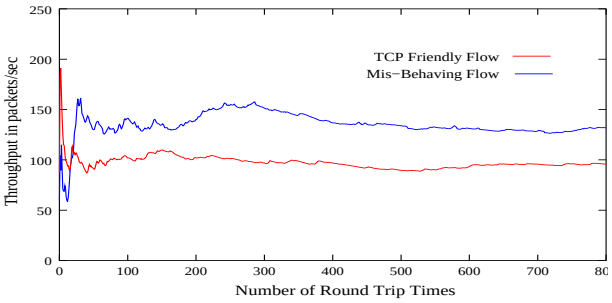


(c): DropTail Queues (Remarking)

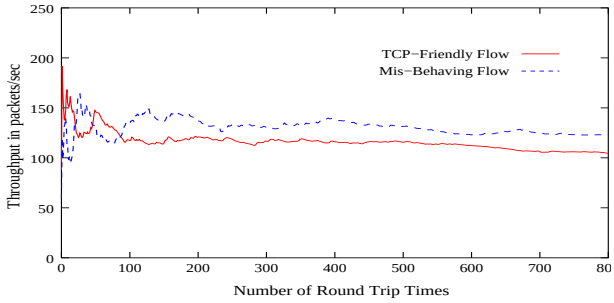


(d): RED Gateways (Remarking)

Figure 5.17: Multi Bottleneck (Dropping): Throughputs (in pkts/sec) for 2 competing flows on a network of DropTail and RED queues with and without Re-Marking. One flow is TCP Friendly while the other is Non-Conformant (k=0,l=0.5).



(a): No Re-Marking



(b): Re-Marking

Figure 5.18: Background Traffic (Dropping):Throughputs (in pkts/sec) for 10 competing flows in a single bottleneck topology, where 7 flows are TCP Friendly while the other 3 are Non-Conformant with (k=0, l=0.5) with 65% noise.

Flow Type	DropTail		RED	
	No-Rem	Rem	No-Rem	Rem
TCP-Friendly	55	85	100	200
Non-Conformant	450	400	400	325

Table 5.2: Cross Traffic (Dropping): Comparison of throughput (packets/sec) for network with DropTail and RED queues with and without re-marking.

as such are not reported here.

We used a single bottleneck topology and different level of flow multiplexing to evaluate the effect of background traffic on the performance of a droptail queue network with and without re-marking. However we report results for one case where there were 10 persistent and of these, 7 flows were TCP Friendly while the remaining 3 where non-conformant (k=0,l=0.5). The bottleneck bandwidth for this simulation was 10Mbps and a buffer of size 150 packets. Also in this setup we increased the noise sufficiently high to validate the robustness of the scheme in presence of many flows and noise. Figures 5.18 a) and 5.18 b) plot the results for the cases where the noise traffic is 65% (or 80 http sources), i.e. mice traffic occupied 65% of the bandwidth. Figure 5.18 b)shows the robustness of the scheme when sufficiently high (65%) noise is present in the network and the re-marker still manages to efficiently patrol non-conformant users.

5.7.2.4 Cross Traffic

Finally we present the results for the edge based re-marker when two way traffic is present in the network. We evaluate this scenario with the multi-bottleneck topology, where we have 5 TCP Friendly long flows and 5 non-conformant (k=0, l=0.5) short flows on each bottleneck. Additionally, on the reverse path, there are 5 TCP Reno flows on each bottleneck. The bottleneck bandwidth for this simulation was 10Mbps and a buffer of size 250 packets. Re-Marking, once again achieves fair sharing of the bottleneck (as shown in Table 5.2). However, it can be seen from the results that DropTail queues perform poorly in comparison to RED queues. This further suggests that deployment of RED will help in improving overall network

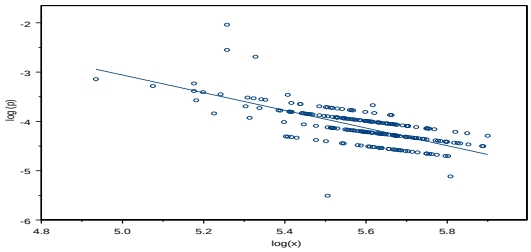
performance, especially in presence of non-conformant flows.

5.7.3 Estimating the Utility Function

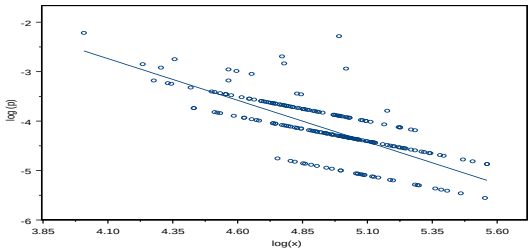
The uncooperative congestion control framework, presented in this chapter, works well if the network knows the utility function of the non-conformant flows. Thus estimating the utility function parameters is of paramount importance. In section we outlined a LMMSE method to estimate the utility function. In this section we present the results of estimating utility function.

We have assumed that the identity of misbehaving user is revealed to us. Thereafter, we sample its packet stream at the egress node counting the number of packets sent as well as packets lost. This data set is then separated into bins of 0.5, 1.0 and 2 seconds where in each bin we measure the number of packets sent and the loss rate for that bin. Once we have constructed such a series we used LMMSE method detailed in Section 5.6.1. The results of a simulation of 2 flows, one TCP and the other a non-conformant flow with k=0, l=0.5, competing on a single bottleneck (see figure 5.5 a) is showed in figure 5.19 with the bin size being 2 seconds. Further, the bottleneck is 0.8Mb, the access links of 8Mb and the bottleneck employs RED with a buffer size of 25 packets. Also, RTT of the flows is 60ms. Figure 5.19 a) and b) show the estimation results for the non-conformant and the TCP flow respectively. The slope of the graph in each case measures $n+1$, where n is the exponent (see Section 5.6.1). The exponents theoretical values for our simulation are 0.5 and 1.0 for the non-conformant and TCP flow respectively. For the non-conformant flow we estimate the exponent to be approximately 0.6 (the slope of the graph is 1.5). Similarly for the TCP flow we estimate the exponent to be approximately 0.8.

Figure 5.22 a) shows the throughput of the two competing flows when we used these exponent values to remark the flows. Figure 5.8 a) shows the throughput when we didn't mark any flow. Because of estimation errors the re-mapping of utility functions was not exact and as such we see that the non-conformant flow still gets more bandwidth. However, there is a significant improvement in the TCP-friendly flow's allocation thus suggesting that the model improves the fair share of



(a) Estimation of Utility Function: Uncooperative Flow



(b) Estimation of Utility Function: TCP Flow

Figure 5.19: Estimation of Utility Function for 2 competing flows in a single bottleneck topology, where one flow is TCP Friendly flow while other is Non-Conformant with (k=0, l=0.5).

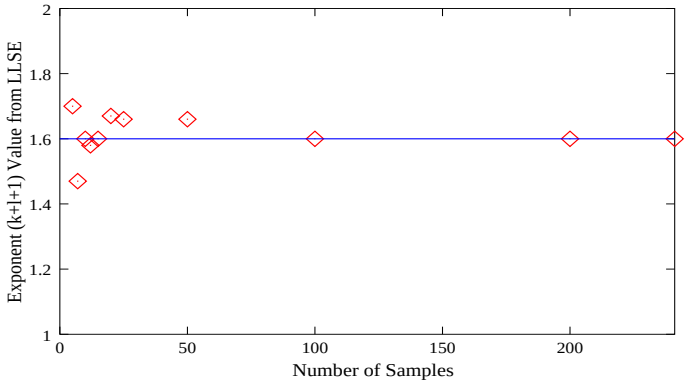


Figure 5.20: Exponent Value Vs Sample Size: As the Sample Size increases estimation gets better. Even Smaller samples give good estimates. Motivates use of RLS.

the conformant flows at the bottleneck by penalizing misbehaving flows.

Although LLSE is simple and has faster convergence it suffers from implementational complexities. Specifically its time complexity is $O(M^2)$, where M is the order of the filter [80]. Moreover, it needs the entire data set a-posteriori to estimate the parameters. However, there exist LLSE schemes which compromise the implementation complexity with convergence. Recursive Least Squares (RLS) [80] is one such scheme. It has a time complexity of $O(M)$ and it can recursively use new data with some incremental work. We will now motivate the need for using RLS and show that good estimates can be gathered with small sample set and then the estimates can be improved by further measurements. Moreover, with RLS the

new measurements can be incrementally consumed.

A point of concern in estimation is - *how many samples are needed to characterize a source ?* We will address this concern using the example presented above. We took the time-series used in previous examples and broke it into smaller series. This gives us the results for estimation with smaller sample space; the new sample sets corresponded to 5, 7, 10, .., 250 samples. In Fig 5.20 we have plotted the exponent value versus number of samples for the uncooperative user. As the figure shows even with 5 samples the exponent, n, is detected to be 0.7 and as the sample size increases the exponent value fast approached the true value. This suggests that using estimation schemes like RLS will make the estimation task easier.

5.7.4 Sensitivity Analysis of Framework

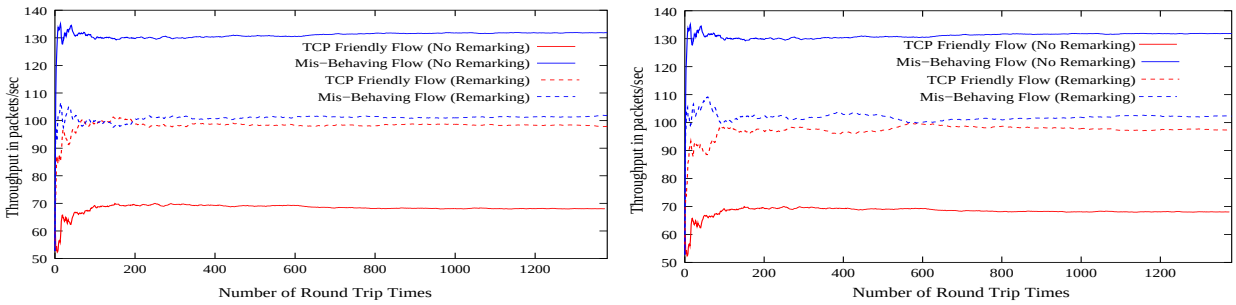
In this section we investigate the effect of inaccurate estimation. Specifically we test the validity of the model in presence of inaccurate utility function and RTT estimates. RTT-estimation is needed for updating our congestion indication estimations (which is similar to the one presented in [40]) while utility function estimation is needed for re-mapping. Our simulation results suggests that the inaccurate RTT estimates don't have a pronounced effect on the re-mapping, at most they might slow the convergence (to the objective utility function). However, large errors in estimation of utility function may over-penalize the non-conformant sources. For the results reported in this section, we assumed that the network was ECN capable and therefore marked packets.

5.7.4.1 Effect of Inaccurate RTT Estimate

In all our previous simulations we assumed that the network knows the RTT of the flows. We used these RTT estimates to update our congestion indication estimations. For the results presented in this section we looked at two cases, one when we under-estimated the RTT and the other when we over-estimated it. We present the results with a single-bottleneck of 0.8Mbps, 25 packet buffer and 2 competing flows.

Figure 5.21 a) shows the results when the RTT was under-estimated as 0.05 (instead of 0.06). Figure 5.21 b) shows similar results when we over-estimated the

RTT as 0.07. The figures suggest that inaccuracy in RTT estimates alters the convergence speed to the optimal point; a larger value of RTT will slow down the convergence while a smaller value will increase the convergence. However, from both the results its easy to see that the effect of inaccurate RTT estimation is not pronounced and the model works well. We ran simulations with higher degree of multiplexing and came to a similar conclusion. However, we do not present those results here.



(a) Under-estimation of RTT (rtt=0.05s) (b) Over-Estimation of RTT (rtt=0.07s)

Figure 5.21: Inaccurate RTT Estimates: Throughputs (in pkts/sec) for 2 competing flows in a single-bottleneck topology, where one flow is TCP Friendly flow while other is Non-Conformant with (k=0, l=0.5), when network has inaccurate RTT estimates.

5.7.4.2 Effect of Inaccurate Utility Function Estimate

Till now we have assumed that the network knows the utility function of the flows. Since utility functions are not being explicit conveyed to the network therefore we will need to estimate them. Thus we need to explore the effect of inaccurate utility function estimates. In this section we evaluate the model's sensitivity to utility functions; when the utility functions are under-estimated and second when they are over-estimated. Under-estimation here refers to the case when we estimate the utility function to be less aggressive than it really is, i.e. when $k + l$ values are reported to be larger than the actual values. Over-estimation refers to the case where we report the flow to be more aggressive than it really is, i.e. $k + l$ values are reported to be smaller than the actual values. We present the results with a single-bottleneck topology (figure 5.5 a)) for 2 flows.

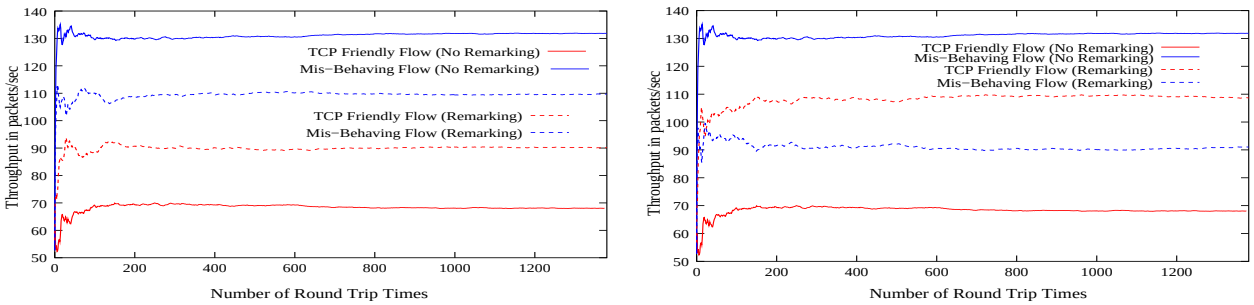
Figure 5.22 a) shows the results when the utility function was under-estimated as 0.6 (instead of 0.5). Figure 5.22 b) shows similar results when we over-estimated it as 0.4. It can be seen from the results that the model is sensitive to inaccurate estimate of utility functions. When we under-estimated the utility function ($k + l = 0.6$) the model didn't penalize the mis-behaving flow much, and as such it still garners more bandwidth than the TCP flow. In the case of over-estimation ($k + l = 0.4$) we see that the network penalizes the mis-behaving flow more and consequently brings it share down below the TCP Friendly flow.

However, the estimation errors pointed out in the simulation are large (the error is 20% since we estimate the $k+l$ values as 0.5 ± 0.1). We evaluated the model for two other error estimates, 10% and 5% and report the result for the 5% error case in figure 5.23. As expected, as the estimation error decreases the model starts to get better. Further we found that for estimation errors of more than 5% the model does not penalize (or over penalizes) the mis-behaving flow much and it consequently has a larger (or smaller) share at the bottleneck. However in spite of these estimation errors, these shares are still more fair than the case when there was no re-marking present. For estimation errors of less than or equal to 5% the model worked well (figure 5.23). We evaluated the model for different simulation setting where we had 10 flows (5 mis-behaving, 5 friendly) and came to a similar conclusion.

5.8 Differentiated Services

In this section we will briefly present how simple differentiated services can be obtained from our framework. As shown in Fig 5.1 any uncooperative user can be mapped to a conformant utility space. Exploit this mapping simple differentiated services can be obtained by re-mapping the utility function to a higher utility function curve, for example map U_2 to U_1 (Fig 5.1).

Though theoretically it is possible to map a utility function to a higher utility function, e.g. map U_2 to U_1, but in practice it implies reducing the end-to-end price for U_2. This clearly cannot work in a dropping based network. Moreover this line of direction is also flawed when applied to a marking based network. This is because a mark always represents a congestion state and by removing a mark would just result



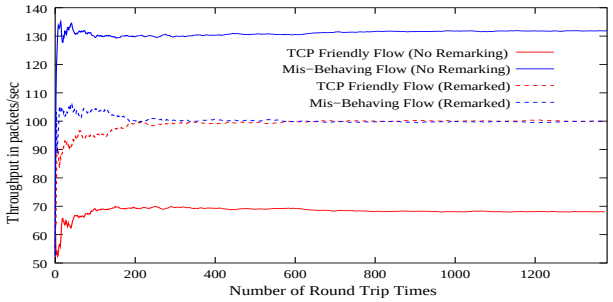
(a) Under-estimation of Utility Function: In-sufficient Re-marking, Misbehaving Flow Still Wins

(b) Over-Estimation of Utility Function: Excessive Re-marking, Misbehaving Flow Over Penalized (Loses to TCP)

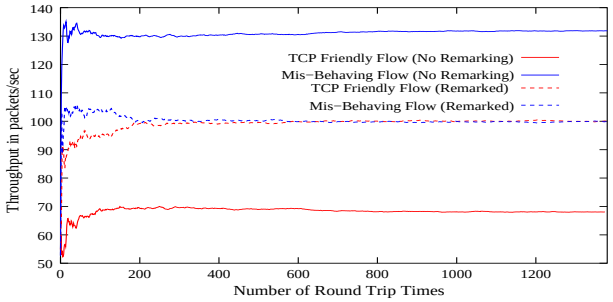
Figure 5.22: Inaccurate Utility Function Estimates, 20% Estimation Errors: Throughputs for 2 competing flows in a single-bottleneck topology, where one flow is TCP Friendly flow while other is Non-Conformant with (k=0, l=0.5), when network has inaccurate estimates of source's utility function.

in delaying the congestion indication, which is in turn more harmful for the source. Hence we need to take a slightly different approach. Suppose that there are two flows, F_1, F_2, in the network, and the utility function of both the flows is U_1. Further assume we need to provide differentiated services to F_1 such that it always receives 10% more bandwidth than F_2. This can be implemented in our framework by simply re-mapping the utility function of F_2 to U_2, such that $U_1\overrightarrow{f(p)}U_2 \Rightarrow x_1\overrightarrow{f(p)}x_2$ where $f(p)$ represents the re-marking function and x_1, x_2 represents the steady state rates of F_1, F_2 respectively.

In Fig 5.24 a) we plot one such result for a single bottleneck topology, where both the flows use TCP and go over a bottleneck link of 0.8Mbps, the buffer size is 25 packets and the RTT is 60 ms. The aim of the simulation was to given one flow 10% more bandwidth than the other flow. As shown in the figure, by re-mapping one of the flows to a lower utility function we can achieve simple differentiated service. A similar result is plotted for a multi-bottleneck scenario in Fig 5.24 b) where the aim was to increase the share of the long flow by 10%. In this simulation the bottleneck capacity was again 0.8Mbps, buffer size of 25 packets, there was one long and one short flow on each bottleneck and all the flows used TCP.

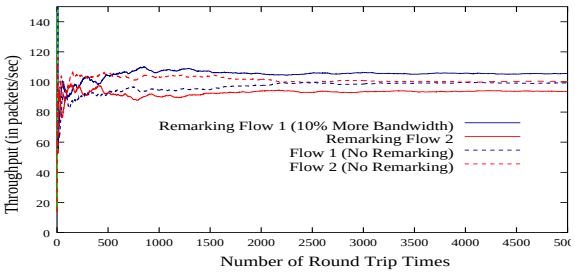


(a) Under-estimation of Utility Function: Bottleneck is still shared equitably because estimation errors are small

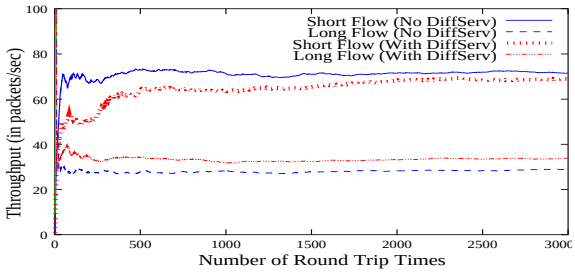


(b) Over-Estimation of Utility Function: Bottleneck is still shared fairly because estimation errors are small.

Figure 5.23: Inaccurate Utility Function Estimates, 5% Estimation Errors: Throughputs (in pkts/sec) for 2 competing flows in a single-bottleneck topology, where one flow is TCP Friendly flow while other is Non-Conformant with (k=0, l=0.5), when network has inaccurate estimates of source's utility function.



(a) Single Bottleneck



(b) Multi-Bottleneck

Figure 5.24: Differentiated Services

5.9 Summary

This chapter addresses the question of protocol conformance in the Internet. In presence of different rate control schemes in the Internet, we consider a rate control protocol to be conformant if they are obtained by maximizing same utility function. Towards understanding the effect of protocol non-conformance, we looked at the impact of non-conformant flows (or mis-behaving flows) on a network of Droptail and RED queues. Our results show that on a network of DropTail queues non-conformant flows get a large (unfair) share of the bandwidth. Further in a multi-bottleneck scenario non-conformant flows can almost shut out the conformant

flows by pushing them into timeouts. However, on a network of RED queues though the non-conformant still share the bottleneck unfairly but the conformant flows are not shut out. In other words the mis-behavior has a significant impact on a network of Droptail queues than RED queues thus motivating for use of RED.

In this chapter we have proposed an abstract model for modeling and managing non-conformant flows. The primary objective of this framework are to look at ways to achieve protocol conformance. However, in this chapter we also look at the fairness at network's perspective, i.e. how a network might allocate resources amongst different users. Towards addressing these issues we have proposed a framework to map a user's utility function, U_s, to any objective utility function, U_{obj}, by manipulating congestion penalties. These penalty transformation agents can be completely implemented at network edges. Further we have a flexibility of choosing either to re-mark the packets or acks. In cases where we do not have access to the packet-stream we can re-mark the ack-stream and achieve the goals of the model. Packeteer boxes, deployed widely on the Internet, already do a similar work by accessing the ack-stream and pacing the acks [72] and work well with even 20,000 flows.

This proposed utility function transformation can decouple the fairness criteria from the user's rate control scheme. This allows the network provider to allocate bandwidth amongst user's according to a broad range of fairness criteria. This framework could also find application in pricing especially those of usage based or flat rate pricing. By having all users conform to a particular rate control scheme, the network ensures that it is fair to all users and hence makes usage based or flat rate billing more meaningful. Broadly, this chapter also suggests that management of mis-behaving or non-conformant flows need *not* be coupled with AQM design and can be simply viewed as an edge network based policing question. This framework may also be thought of as a new class of "traffic conditioning" technique, where the "conditioning" is achieved by manipulation of the feedback stream rather than manipulation of the packet stream.

We have analyzed the framework and evaluated it for various single and multi-bottleneck scenarios with marking and dropping policies being used for congestion

notification. Further we showed model is robust and works well even in presence of high background (web) traffic and reverse path congestion. In this chapter we have also presented a scheme to estimate the utility function of the non-conformant user. We also tested the sensitivity of the framework to estimation errors (for RTT and utility function). Our results how that inaccurate RTT estimates do not have a very profound effect on the model's correctness. However, in presence of large utility function estimation errors the model does not fully correct the non-conformant flows, but still considerably improves the fairness at the bottleneck (as compared to the scenario when there was no re-marking).

However, a limitation of the proposed framework is that it only considers the mapping of selfish responsive schemes and might not work well if path asymmetry exists. In such a situation we would have to place the penalty transformation at every exit routers. Further, path asymmetry will also result in erroneous values for network losses which might make the framework either over penalize or under penalize the selfish flow. Thus when a flow may take different paths, we would need coordination between all the penalty transformation agents. However, if a single ingress (or egress) router is used by the flow then the model is immune to path-asymmetry problems of the network. Both unique ingress or egress routers is generally true in the present Internet.

CHAPTER 6
virtual AQM: Managing Bottleneck Queues from Network Edge

The proposals previously discussed in this thesis improve fair sharing of the network, can efficiently manage selfish flows and remove some other deficiencies of Drop Tail queues like phase effects and synchronization. Consequently, we have seen an improvement in performance of TCP with reduction in number of timeouts and burst losses, removal of bias against flows with large round-trip time and improvement in fairness. However neither Randomized TCP nor Uncooperative Congestion Control framework can proactively manage bottleneck queues. This implies that the task of managing bottleneck queues in the network is still associated with the requirement that an active queue management module be present at every router in the Internet.

The Internet uses only Drop Tail queues. Drop Tail queueing is also often referred to as passive queueing or in other words it does not manage queues. As a result the Internet operates with near full queues which causes increase in end-to-end latencies. Moreover, this inability to manage bottleneck queues also results in delayed congestion response from the sources, and possibly big congestion window oscillations. As such, we need to direct efforts to manage bottleneck Drop-Tail queues from network edge or end-systems. Towards achieving this target in this Chapter we will outline an edge-and-end-system proposal called *virtual AQM (vAQM)*, to manage almost all bottleneck queues in the network.

We show through simulations that an edge based *virtual AQM* module can reduce the steady state queues in the network. As such, the framework presented in this chapter, *virtual AQM*, shows that we can de-couple the task of management of bottleneck queues and it's placement. In other words, for managing queues in a network, we do not require an active queue management component to be present at every bottleneck.

6.1 Introduction

The primary task of any AQM is to proactively manage bottleneck queue length so as to provide early congestion indication and thus keep the network utilization healthy. Proactive management of queues also generally results in small bottleneck queues which in turn not only reduce end-to-end latency but also provide space to enque any incoming bursts and thus prevent burst losses and timeouts. However, a majority of AQM proposals, especially RED and it's derivatives, do not always operate with small queues and consequently their worst case performance is worse than simple Drop-Tail queues. The primary reason for such poor worst case performance is lack of proper guidelines for configuring such AQMs.

Recently a few AQM schemes schemes based on virtual or shadow queueing have been proposed and initial results have shown that these proposals have a fairly large operating area. The most popular of such schemes is AVQ (Adaptive Virtual Queue) by Kunniyur et. al. [57]. AVQ at a router emulates a virtual link that has a capacity less than the real link. In order to achieve this, AVQ marks/drops packet to match the capacity of this virtual link. This virtual link capacity can also be thought of as the actual desired link utilization. In order to do this, AVQ maintains a virtual buffer which together with actual router buffer simultaneously enques any incoming packet. Upon the receipt of an incoming packet, the virtual queue length is updated. If the virtual queue length overflows, the packet is subsequently marked (or dropped) in the actual router queue. Thus, AVQ allows network operators to deploy any AQM scheme in the network, including Drop Tail queue, to take any corrective action on any packet which resulted in overflow of virtual queue. To summarize, since the incoming packet rate equals the virtual link capacity at the steady state, the router queue can thus be kept to zero (or, in fact, a very small value due to bursty traffic).

Though AVQ allows the providers to continue operating with Drop Tail queues, all the routers must be upgraded to perform the virtual queueing tasks. This is not only expensive but requires significant upgrades. As such, for some foreseeable future, the network will continue to operate with Drop Tail queues. In this thesis, we present an end-or-edge based framework which emulates AVQ properties over a

network of Drop Tail queues.

The framework presented in this chapter, *virtual AQM* shows that we can de-couple the task of management of bottleneck queues and it's placement. In other words, for managing queues in a network, we do not require an active queue management component to be present at every bottleneck. We show that using end-to-end packet probes and a *virtual AQM* module, we can manage the bottleneck queues at the network edge.

Our initial results suggest that the proposed framework can significantly reduce bottleneck queue lengths without compromising on link utilization or fairness. The rest of the Chapter presents the arguments in detail. In Section 6.2 we present our framework, *virtual AQM* and outline a special case called *vAVQ* or virtual AVQ which emulates virtual queueing properties of AVQ in Section 6.2.1. Section 6.3 presents some results with *vAVQ* and the comparison of its performance with AVQ and simple Drop Tail queueing. Finally, in Section 6.4 we debate on the merits and limitations of *vAVQ* and present the conclusions and future work in Section 6.5.

6.2 vAQM: virtual AQM

In the Internet a flow might traverse multiple bottlenecks. However, of these multiple bottlenecks there is only one congested link which is dominant. In other words, the dominant bottleneck is the most congested link or is referred to the link with highest marking (or dropping) probability. Recent studies have shown that even though the end-to-end latency distribution might be multi-modal (i.e. there are multiple bottlenecks) the tail distribution of end-to-end latency is dominated by a single link [97]. As such, if for every flow, we could abstract out this dominant congested link then running an AQM only on this router will also ease out congestion on other bottlenecks. However, the congestion at other bottlenecks is not only because of traffic stream going to the dominant router, but also of cross traffic which might not always come to the dominant router.

Figure 6.1 shows one such scenario. In this example, for the flow, F, the router, R_3 is the dominant congested router while routers R_1 and R_2 are other congested routers on its path. If we were to run just one AQM on the router R_3 we will certainly

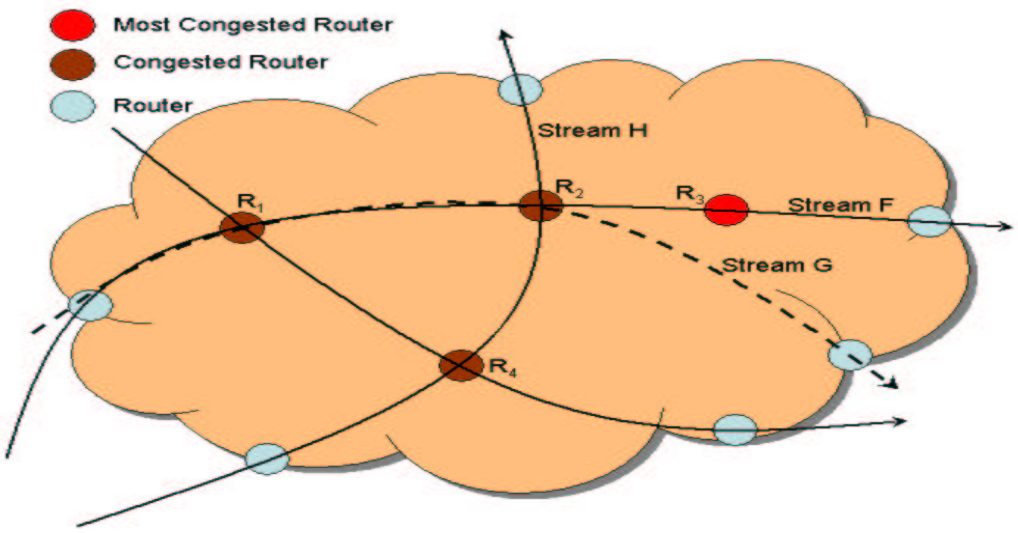


Figure 6.1: Most Congested Router

ease the congestion on that router. As a result of this active queue management at router R_3, the flow group F will also be policed, thus easing the congestion on routers R_1 and R_2. However, since the total traffic at routers R_1 and R_2 is made up of other traffic streams such as G and H, the congestion at these bottlenecks may not be alleviated fully. None the less, such a method will also reduce the queue length at the bottleneck queues and thus reduce end-to-end latency.

This ability to identify the most congested router on a path and run just one AQM on that particular router is the stepping stone for our proposal *virtual AQM* or *vAQM*. However, we further extend the notion of running one AQM to network edge, i.e. from the network edge for every path we will try to identify and manage the most congested router by running an AQM for that *particular path* at the network edge. In this proposal we refer this to as *virtual AQM*.

The virtual AQM model can be explained within the utility function based network optimization framework. Consider a source, s, which is described by its rate x_s and lets assume it's maximizing a utility function $U_s(x_s)$. We will further assume that the utility function is strictly increasing and concave in it's arguments. Then we can explain running an AQM only for the dominant congested router by

the following set of equations

$$U'(x_s) = \underbrace{max}_{l} p_l \tag{6.1}$$

$$p_l(\sum_s x_s - C_l) = 0 \tag{6.2}$$

$$\sum_s x_s - C_l \leq 0 \tag{6.3}$$

$$p_l \geq 0 \tag{6.4}$$

6.2.1 vAVQ: virtual AVQ

We will now outline one specific proposal under *vAQM* which uses packet probes and AVQ to identify and manage the most congested router on any path. However, before we outline the model we will define a few terms used in our framework.

Definition 1. *Stream: The group of flows which have the same ingress and egress routers. Moreover, these flows have the same routing path between the ingress and egress routers.*

Definition 2. *Path: The route, between ingress and egress routers, taken by a particular stream.*

Definition 3. *Path Capacity: The minimum link capacity along a path. Thus, for a stream, f, which traverses n bottlenecks, $R_1, R_2, .., R_n$, we define the path capacity, C_f, as $C_f = min(C_{R_1}, C_{R_2}, .., C_{R_n})$*

Definition 4. *Path Demand: The maximum demand along a path. Thus, for a stream, f, which traverses n bottlenecks, $R_1, R_2, .., R_n$, we define the path demand, D_f, as $D_f = min(D_{R_1}, D_{R_2}, .., D_{R_n})$*

Consider a scenario where a network edge knows the path capacity and demand for every stream entering the network. Further assume that the provider has specified a desired target utilization for the network. Let the desired network utilization be denoted by γ, such that $0 < \gamma < 1$. Further, for any stream, f, let the path capacity be denoted by C_f, demand by D_f and the virtual path capacity of γC_f. Thereupon we conjecture that if we try to match the demand, D_f, to the

virtual path capacity, γC_f then by the virtue of utilization factor being less than 1, the steady state queue due to the stream f will be very small or close to 0. Moreover, if we are able to match the demand with respective virtual path capacity for all streams then the queues at all the bottleneck routers will be small or close to 0. Further, if we know the path capacity and demand for each stream, we can do this at the network. This is the premise of our proposal *virtual AQM*, which use packet probes to estimate path capacity and demand and tries to manage bottleneck queues from the network edge. Figure 6.2 shows the model of *vAVQ*.

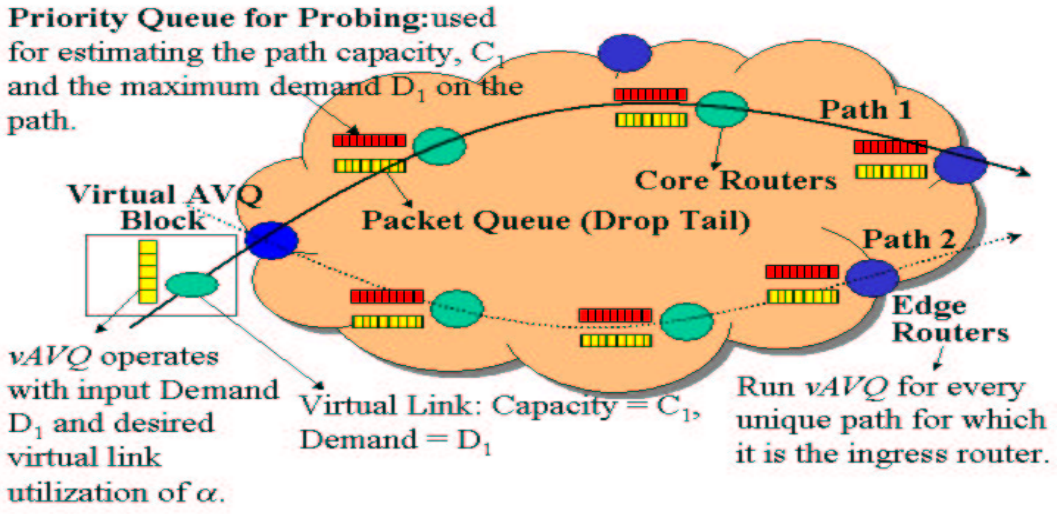


Figure 6.2: The model for *vAVQ*

Further, since the vAVQ consistently attempts to keep the network utilization below 100% (or at the specified network utilization factor) it will usually signal congestion indications to flows before the network starts dropping packets. Such a behavior in turn results in proactive queue management and thus might help us in managing the bottleneck queue length from the network edge.

6.2.2 Estimating Path Capacity and Demand

In this section we will outline end-or-edge system based methods for estimating path capacity and demand. For the purposes of estimation we envision use of priority

Algorithm 1 *virtual AVQ* (vAVQ)

t: Current time
s: Arrival time of previous packet
T: Time when last demand estimate was taken
$D(T)$: Demand Estimate at time t in packets
$C(T)$: Capacity Estimate at time t in packets
ξ: Time after which $D(T)$ is updated
γ: Network Utilization Factor
B: Buffer Size in packets
VQ: Size of Virtual Queue in packets
$\tilde{C}$: Virtual Path Capacity in packets.

for each packet arrival in $(t, t+\xi)$ **do**
  VQ $\leftarrow$ max(VQ, $\tilde{C}$(t-s), 0) /* Update virtual queue size */
  if $VQ + 1 > B$ **then**
    Mark (or drop) the packet in real queue.
  else
    VQ $\leftarrow$ VQ + 1
    Enque the packet in the real queue.
  end if
  $\tilde{C} = \tilde{C} + \alpha * (\gamma C(T) - D(T)) * (t - s)$ /* Update virtual path capacity */
  $\tilde{C} = min(max(\tilde{C}, 0), \gamma C(T))$
  s $\leftarrow$ t /* Update last packet arrival time */
end for

queues in the network. Specifically, we propose two queues: a high and low priority queues. The packets enqued in the high priority non pre-emptive queue are serviced before those in the low priority queues.

To calculate the path capacity, C_f, we send probe packets in pairs and these packets are enqueues in the priority queue in the network. These probe packets are sent back to back and their inter-arrival time at the receiver is used to measure the available path capacity. Specifically, if the inter-arrival time of these probe packets, at the receiver, is δ, then the path capacity is given as

$$C_f = \frac{8 * S}{\delta} \tag{6.5}$$

where S is the size of probe packets in bytes.

To estimate the path demand, D_f, a packet train is sent through the low priority queue. However, this packet train is not sent back-to-back but with some

inter-packet gap. Then, at the receiver, the difference between the inter-arrival time and inter-packet gap represents the estimate of effective capacity on that path. An estimate of demand can then be calculated by taking the difference of the actual path capacity and the effective capacity. Thus, if the inter-arrival gap at receiver is t_a and the inter-packet gap is t_g, the demand can be estimated as

$$D_f = C_f - \frac{8 * S}{t_a - t_g} \quad t_a > t_g \tag{6.6}$$

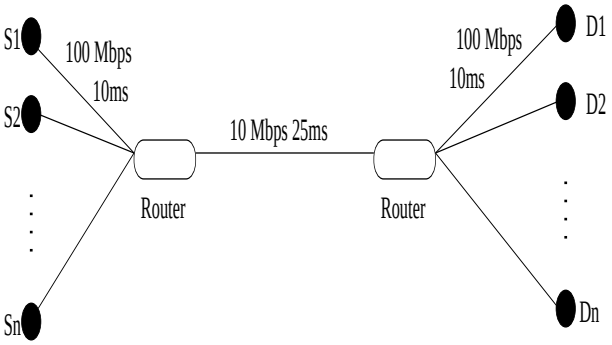
$$= 0 \quad otherwise \tag{6.7}$$

However, to estimate demand the most important configuration parameter is the inter-packet gap. If the inter-packet gap is very large then our simulation results show that inter-arrival time at receiver is equal to the inter-packet gap. Thus, in such cases we do not have an estimate of demand. Similarly, if the inter-packet gap is very small then packets may go back to back and in such cases the estimate for demand is often equal to the path capacity. A more through discussion on the inter-packet gap is presented in [47].

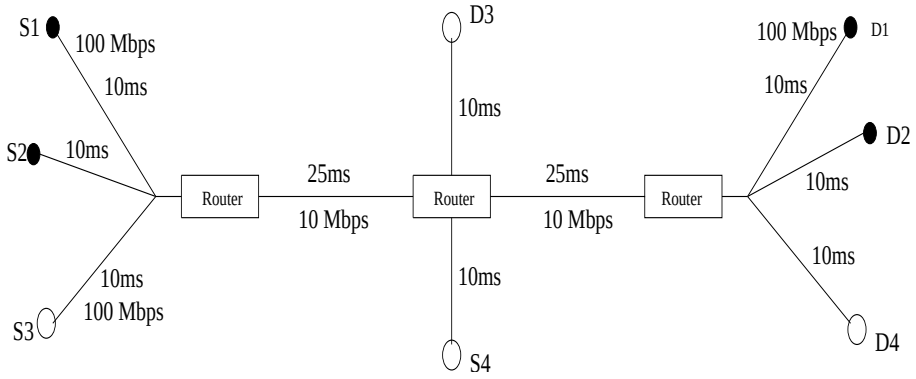
Besides using packet probes for estimating demand we could use the data packets themselves for purposes of estimation. If the sender stamps the sending of every packet (or some packets) and the receiver echoes the time when these packets were received then the difference between sending and receiving timestamp will give us a measure of the demand on the path. However, such a method will not work with very small window sizes because then the inter-packet gap is huge and we often get no demand estimates.

6.3 Results

We implemented the *vAVQ* in NS. The bottleneck routers had two queues, a priority queue for path capacity probe packets and a Drop Tail queue for data and path demand probe packets. Packet pairs were used for estimating both path capacity and demand. For the results presented in this section, the path capacity was always estimated as the minimum link capacity in the path. As such, we will not present the results for path capacity estimation. Figure 6.3 (a) and (b) show



(a) Single Bottleneck Topology

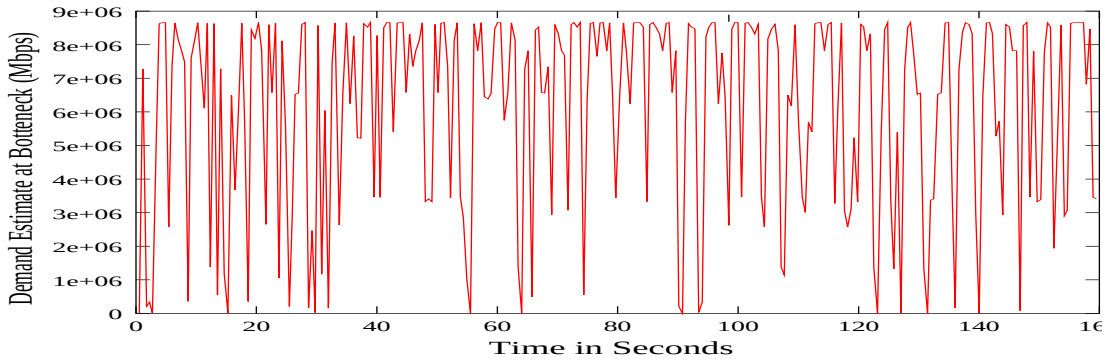


(b) Multi-Bottleneck Topology

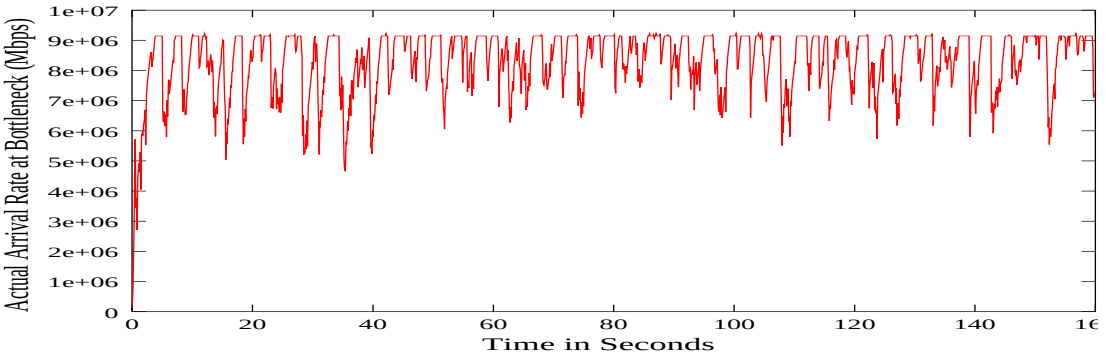
Figure 6.3: Topologies used in the Simulations.

the single and multi-bottleneck topologies used in the simulation. The access links have 10 times more capacity than the bottleneck link. Further, all the bottleneck links had Drop Tail queueing. For our implementation, we used the congestion control and loss recovery mechanisms of TCP New Reno and disabled the delayed acknowledgments option.

For the results presented in the following sections, the performance metrics of interests are evolution of instantaneous bottleneck queue length, average bottleneck queue length, fairness and bottleneck link utilization. In this section we will evaluate *vAVQ* for these metrics and compare it's performance with Drop Tail and AVQ buffer management policies. For all the simulation results presented in this section, unless specifically specified, the virtual buffer length for AVQ was set to be equal to the actual bottleneck buffer length. Further, the desired link utilization was set to be at 90% of link capacity and AVQ was set to mark packets in the actual queue.



(a) Demand Estimate at the Bottleneck



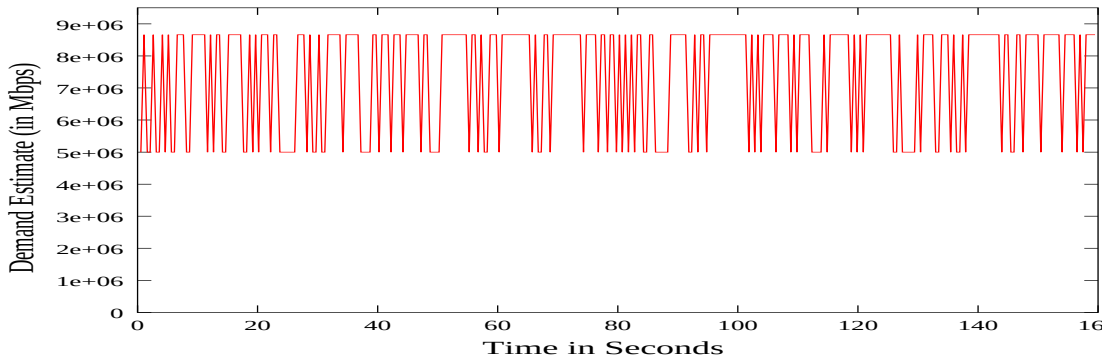
(b) Actual Arrival Rate at Bottleneck

Figure 6.4: Demand Estimation in a Single Bottleneck scenario.

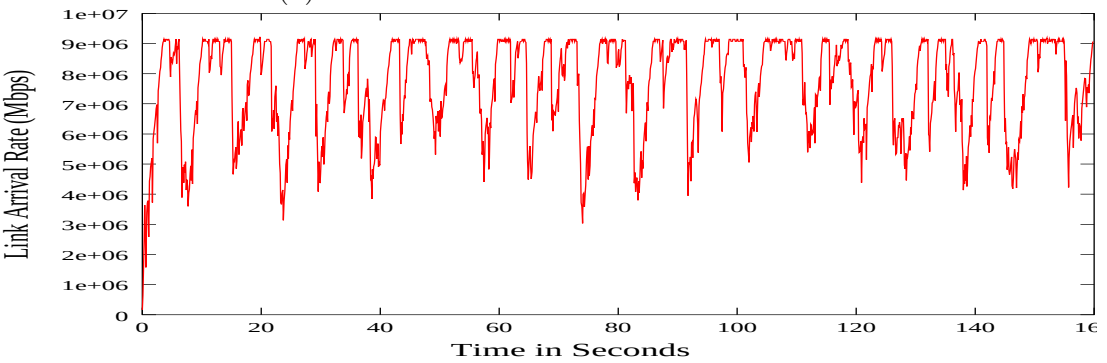
6.3.1 Single Bottleneck Topology

We will first present the results for demand estimation. For estimating the demand we used packet pairs. A packet pair with an inter-packet gap of 0.00037 seconds was sent every 0.1 seconds. The sender time stamped the packets receiver used these timestamps to calculate an estimate of the demand. Further, we filtered this demand estimate using an Exponentially Weighted Moving Average (EWMA). The EWMA filter gave 40% weight to the current estimate and 60% to the smoothed average. This smoothed demand estimate was then communicated to the sender for running the *vAVQ* algorithm. Figure 6.4 (a) and (b) show the results of estimation using packet probes and actual packet arrival rate at the bottleneck.

As the results show, the demand estimate are often close to zero and as such the estimates are not entirely accurate. The main reason behind this inaccuracy is the burstiness of the cross-traffic (TCP in this case). In some cases, a burst of packet arrives before the first packet of the probe and as a result the first probe



(a) Demand Estimate at the Bottleneck



(b) Actual Arrival Rate at Bottleneck

Figure 6.5: Demand Estimation in a Single Bottleneck scenario when the estimate's lower bound is 50% of Link Capacity. This bounding then gives us a better estimate of Demand.

packet sees a large queue in front of it. Now if there is an idle period of the burst, and the second probe packet arrives in this period, it is enqued behind the first probe packet and we thus under-estimate demand. A similar case can be sighted for the over-estimation of demand, wherein the first probe packet sees almost no burst while the second period sees a huge burst. Estimation of available capacity, path capacity and demand has received considerable attention, starting from the first work on packet trains [50] and packet-pair [54] to the current efforts [59, 47, 26]. However, all these efforts report that the estimation suffers and is at best within 10% of the actual value.

Now we will present a result which will show that *vAVQ* will perform significantly better is the demand estimations improved. Our previous demand estimation results showed that we often estimate the demand to be close to 0, while the actual

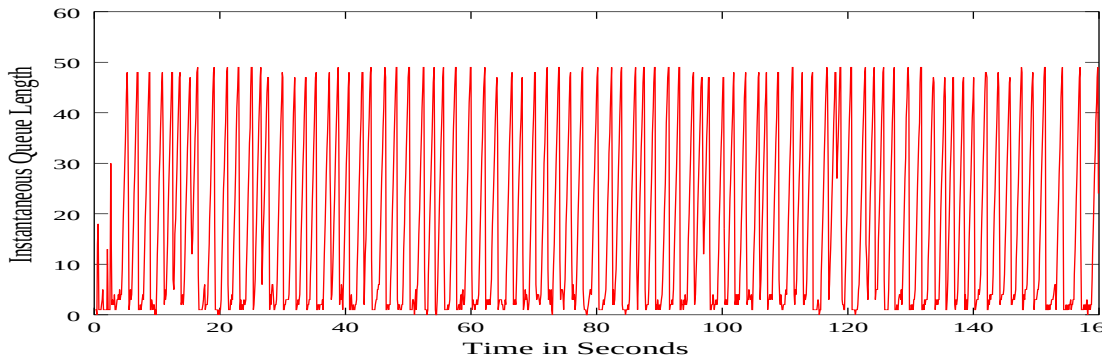
	Drop Tail	AVQ	vAVQ	vAVQ (good estimate)	AVQ (6 packet buffer)
Avg. Queue Size	18.69	13.62	12.22	10.94	4.30
Avg. Throughput (Mbps)	1.6	1.5	1.6	1.5	1.45
Fairness	0.067	0.30	0.06	0.05	0.012

Table 6.1: vAVQ: Performance on a Single Bottleneck Topology

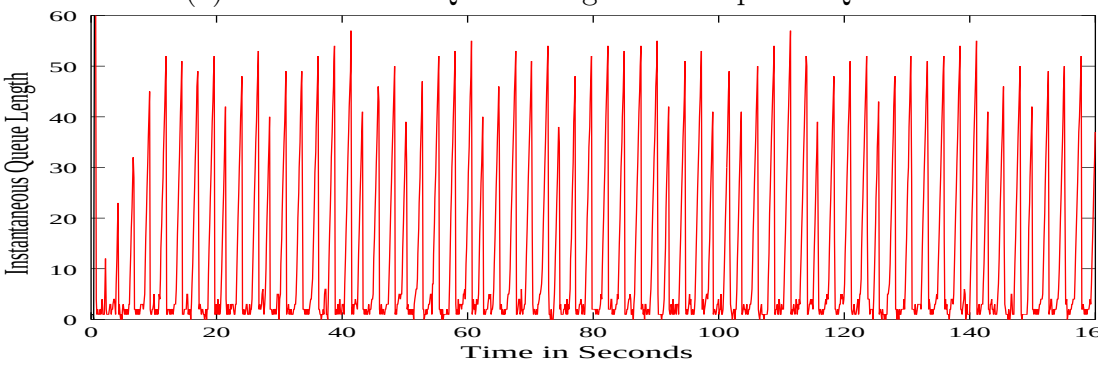
demand at the bottleneck is seldom less than half of the bottleneck capacity. We used this insight to engineer a new demand estimate. Specifically, we decided to round-off all demand estimates to be at least half of the path capacity. We conjecture that in absence of window synchronization and presence of many persistent flows the bottleneck link will always be almost fully utilized. Our results as shown in Figure 6.4 (b) and 6.5 (b) further validate our arguments. Coming back to our engineered demand estimates, Figure 6.5 shows the results of one such experiment. As we can see the rounding off of demand estimate significantly improves the results and now the estimate as shown in Figure 6.5 (a) almost resembles Figure 6.5 (b). Later in this section we will present the result with this tailored estimate and show that it significantly improves the performance of *vAVQ*.

Now we will present the results for *vAVQ* and compare it's performance with simple Drop Tail queueing and AVQ. Our performance metrics of interests are evolution of instantaneous bottleneck queue length, average bottleneck queue length, fairness and bottleneck link utilization. For this simulation the bottleneck link capacity was 10 Mbps, the bottleneck queue size was 50 packets and the round-trip time was 90ms.

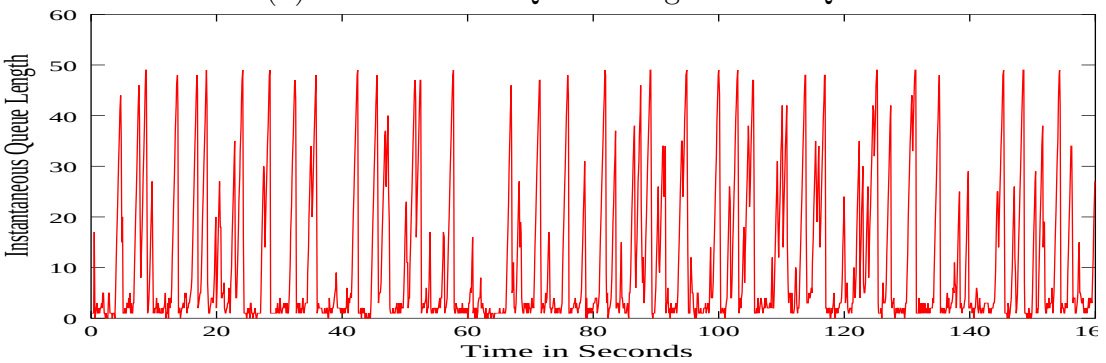
Figure 6.6 presents the instantaneous queue length evolution for Drop Tail, AVQ and *vAVQ* buffer management policies. Drop Tail queues hardly manage the bottleneck queues and are more likely to operate with near full queues, see Figure 6.6 (a). This huge and consistent variations in queue not only increase the average queue size (as shown in Table 6.1) but also result in oscillations in window size thereby making any form of provisioning (for multi-media services) difficult. AVQ on the other hand does much better than Drop Tail queue, as it's average queue size is less



(a) Instantaneous Queue Length for Drop Tail Queue



(b) Instantaneous Queue Length for AVQ



(c) Instantaneous Queue Length for vAVQ

Figure 6.6: Instantaneous Queue Length Evolution in a Single Bottleneck

than that of Drop Tail queues by almost 30%. However, the oscillations to full and zero queues are still present, though they are not as consistent as Drop Tail queues. One of the reasons, why AVQ still oscillates is the large virtual buffer length which ensures that there is almost always space to accommodate any incoming packet. As such, the virtual buffer rarely overflows and not many packets are not marked thus allowing the actual router queues to grow. However, AVQ still manages to

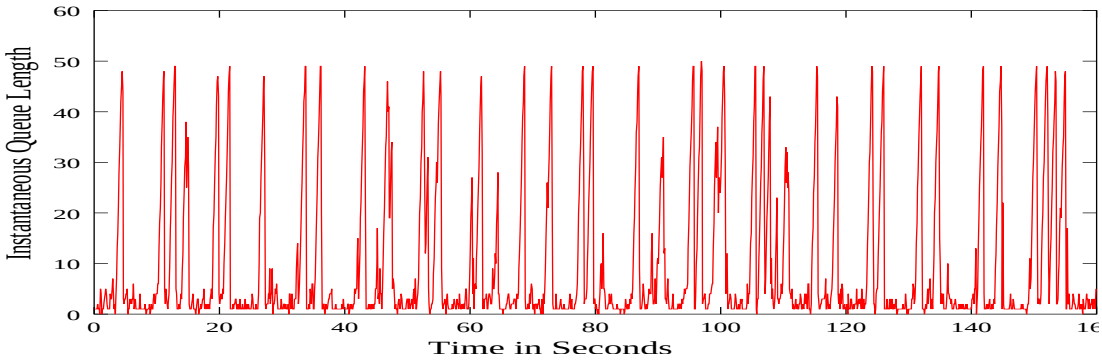


Figure 6.7: Instantaneous Queue Length for *vAVQ* with almost perfect Demand estimate

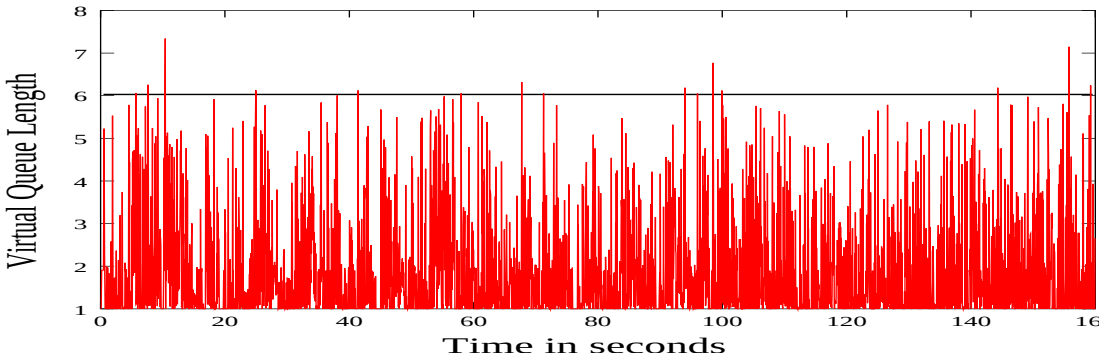


Figure 6.8: *vAVQ*: Evolution of Virtual Queue at Network Edge

match the arrival rate to the bottleneck link capacity and thus keeps the network utilization pegged at 90% (as shown in Table 6.1). In Figure 6.9 we plot the results of the simulation when the virtual buffer length for AVQ is 6 packets (this is the value used in our *vAVQ* simulations). As discussed previously in this section, with small and moderate values of virtual buffer length AVQ performance significantly improve: not only is the average value of queue length small in this case but also there are fewer excursion to full queue.

We configured *vAVQ* such that one instance of *vAVQ* was being run for every source. However, there was only one demand estimation module which estimate demand every 0.1 seconds, on an average. All the *vAVQ* instances used this demand estimate to update their virtual capacity and buffer estimates. We evaluated *vAVQ* with a range of virtual buffer length values. A very large value of *vAVQ* buffer length will not result in performance improvements. This is because when we run a *vAVQ* for every flow then, given the burstiness of TCP (which in absence of reverse

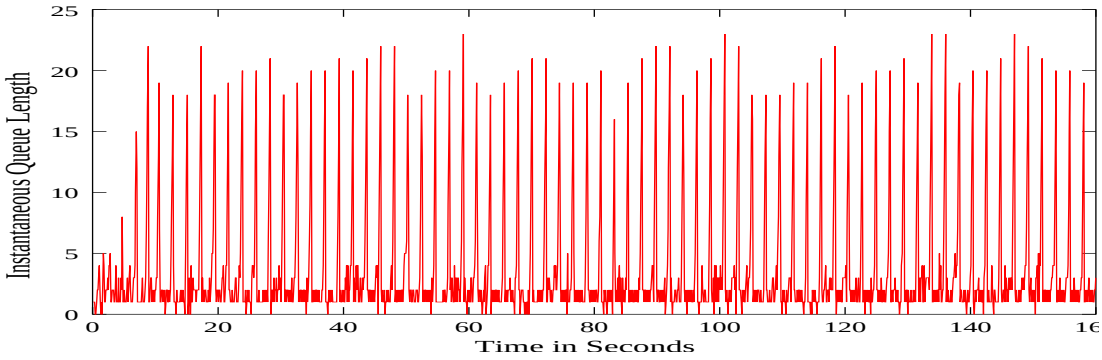


Figure 6.9: AVQ: Instantaneous Queue Length evolution when the virtual buffer length is 25% of actual router queue.

path congestion sends only two or three back-to-back packets) a very large virtual queue will always have space to enque the incoming packets. As such, the virtual queue hardly overflows and there is almost no change in the queueing at the actual router queue. Similarly, we can argue that if we run *vAVQ* for every flow, then the virtual buffer length at the router should have at least two or three packets. For the result presented here, we chose the virtual buffer length to be 6 packets. Further, the demand estimation for this simulation setup has already been explained earlier in this Section and is shown in Figure 6.4.

Figure 6.6 (c) shows the instantaneous queue length plot while the Figure 6.8 shows the evolution of virtual queue length. Further, the Table 6.1 shows the average value of instantaneous queue. It can be seen from these values that *vAVQ* can substantially reduce the average queue size and by implication average queueing delay and thus end-to-end latency. In this particular case, the reduction is almost 30% as compared to queueing with Drop Tail queues. Moreover, these are the results when we do not have a good measure of the demand on the bottleneck. To further evaluate the *vAVQ* algorithm we used the demand estimate where the lower bound of estimate was fixed at half of path capacity, see Figure 6.5. The instantaneous queue length for this simulation is plotted in Figure 6.7. Our results show that with good demand estimates the performance of *vAVQ* improves and in this case the average value of instantaneous queue length is almost half of it corresponding value with just Drop Tail queues.

Table 6.1 shows the average queue size, average throughput and coefficient of

variation of throughput for Drop Tail, AVQ and *vAVQ* schemes. The AVQ and *vAVQ* algorithms were set to operate at 90% of bottleneck capacity and this is reflected in their throughputs. When *vAVQ* is operated with good demand estimates and for either case of AVQ, we can see that the average throughput is about 90% of it's corresponding value for Drop Tail queues. However, there is a substantial reduction in the queue size, so much so that for the *vAVQ* with good demand estimates the queue size is almost half of that for Drop Tail queues. Further, we can see from the coefficient of variation of throughput that both AVQ and *vAVQ* are fair.

However, the performance of *vAVQ* is still inferior to that of AVQ (with same virtual buffer queue lengths). This is because, AVQ operates at the bottleneck, as such at every instant it has precise estimates of demand. *vAVQ* on the other hand relies on packet probes to estimate demand which are not only inaccurate but often stale. *vAVQ* sends probe every 0.1 seconds to estimate demand and this estimate is used for all subsequent calculations. Thus the demand estimate is often stale. However, we could correct this situation by taking demand estimates more frequently. Any such method will however considerably increase the control traffic in the network.

6.3.2 Multi-Bottleneck Topology

The multi-bottleneck topology used in this simulation is shown in Figure 6.3. There were two long flows, i.e. flows which traversed both the bottlenecks. Also, on each bottleneck there were three short flows, i.e. flows which traverse only one bottleneck. The queue length at each bottleneck router was fixed at 50 packets. Once again, we compared the performance of *vAVQ* with both Drop Tail and AVQ buffer management policies. Figure 6.10 and 6.11 show the instantaneous queue length for both these cases for both the bottlenecks. Table 6.2 shows the corresponding average queue size and fairness for this simulation.

For *vAVQ* we ran three separate packet probes for estimating demand. We ran one probe for estimating demand for the long flows, this is because these flows go through two bottlenecks and as such their estimate of path capacity and demand will be very different from that for short flows. Accordingly, we ran two more demand

First Bottleneck

	Drop Tail	AVQ	vAVQ	vAVQ (good estimate)
Avg. Queue Size	18.00	11.46	15.48	14.02

Second Bottleneck

	Drop Tail	AVQ	vAVQ	vAVQ (good estimate)
Avg. Queue Size	17.97	10.86	15.72	14.66

Long Flow	Drop Tail	AVQ	vAVQ	vAVQ (good estimate)
Avg. Throughput (Mbps)	0.66	0.53	0.89	1.11
Fairness	0.05	0.02	0.03	0.05

Short Flow	Drop Tail	AVQ	vAVQ	vAVQ (good estimate)
Avg. Throughput (Mbps)	2.20	2.17	2.20	2.21
Fairness	0.09	0.03	0.03	0.05

Table 6.2: vAVQ: Performance on a Multi-Bottleneck Topology

estimates for the short flows: one for the flows going through only one bottleneck and one for the flows going through the second bottleneck. Once again our demand estimates are inaccurate for all these cases and this will reflect in the performance of *vAVQ*. Finally, we ran one instance of *vAVQ* for each flow. For the long flows the *vAVQ* was configured to be on the access link to first bottleneck router while for the short flows the *vAVQ* was configured on their respective upstream access links.

Figure 6.10 (c) and 6.11 (c) show the results for instantaneous queue evolution at both the bottlenecks with *vAVQ*. The virtual buffer length for this simulation was set to 6 packets. We experimented with other values of virtual buffer length and will comment on them later in the section. Table 6.2 shows the average value of instantaneous bottleneck queues. Once again, *vAVQ* improves on the performance of Drop Tail queues. However, the gains are less than those obtained with the single bottleneck simulations. One of the reasons for this is the increase in errors in estimation of demand. Another reason is that with long flows going over both the

bottlenecks it is not possible to perfectly pin-point to the bottleneck. For example the demand estimate we get may largely correspond to the first bottleneck but the second bottleneck is congested more (as compared to the first bottleneck). As such, the rate cut might not be sufficient to alleviate the congestion at the second bottleneck. Another reason for reduced performance gains is that once again we take demand estimates every 0.1 seconds. As such, in addition to the estimates being stale the probability that they are wrong is also high.

Table 6.2 shows the average throughput and fairness for the multi-bottleneck scenario. The results show that the bias against large RTT flows is present with Drop Tail queues (which can be seen by low average throughput for long flows). However, surprisingly AVQ also shows this bias. This is because the virtual buffer for AVQ is large so even though it matches the input rate to output capacity, the queue management is similar to drop tail queues. As a result of this AVQ instead of marking packet also ends up dropping packets. Consequently both AVQ and Drop Tail show a comparatively large number of retransmissions as compared to *vAVQ*. The *vAVQ* algorithm has shorter virtual buffer (6 packets) and thereby frequently marks packets. Moreover, since the arrival rate for the short flows is more (than long flows) they get a proportionately larger share of marks and this allows long flows to get more share of the bottlenecked link.

Table 6.2 also shows the results when we lower bounded the demand estimates to half of path capacity. We can see that as the demand estimates improve the performance of the proposed algorithm also improves. In another experiment, we varied the virtual buffer lengths. Specifically, we chose different virtual buffer lengths for the long and the short flows. Our initial results show that a smaller virtual buffer length for short flows (as compared to that for long flows) reduces the average queue size further. This is because, the short flows go through only one bottleneck, as such they are the biggest contributor to the demand at any bottleneck. As such, a large value of virtual buffer length for them ensures that the virtual buffer length does not over flow often and as a result fewer packets are marked. As such, the biggest contributor to traffic generally passes unchecked and average queue size continues to be high.

6.4 Discussion

In this section we will discuss the merits and limitations of *vAVQ*. As the results show, the edge-based queue management scheme can reduce the average queueing delays and does not require extensive network upgrades. We believe that this is an area of research which needs to be investigated further as it could lead to interesting and deployable queue management algorithms.

In this thesis we have presented an abstract framework for managing bottleneck queues from the network edge. Specifically, we conjecture that if we run an AQM, at network edge, per path then we can reduce the average queueing in the network. For this purpose, in this Chapter we propose use of packet probes to identify the bottleneck router. However, we are not interested in finding exactly which router is back logged rather we just try to find the bottlenecked capacity and the peak demand on that path. Thereupon, we propose that through an edge based AQM we can match this demand to some desired fraction of bottlenecked capacity (which in turn in the desired network utilization). Now, if this network utilization factor is less than 100 then at steady state the total input (on the path) will always be less than the bottleneck link capacity leading to near zero queues.

The edge-based AQM scheme presented in this thesis is a novel concept which to the best of our knowledge has not been explored before. Moreover, our results suggests that this line of work entails very interesting possibilities and needs to be evaluated further. However, the evaluation of the *vAVQ* is by no means thorough and many of it's building blocks needs to be thoroughly investigated. In this section we will look at all the building blocks of the *vAVQ* and discuss how they can be improved further.

The key to a successful operation of the proposed scheme is the accuracy of the estimation component. It is through estimation that we try to find the bottlenecked link capacity and demand estimates. Inaccuracies in either estimates can severely constrain the performance of the proposed algorithm. As shown in Section 6.3.2 the performance of the algorithms severely suffers because of errors in estimation. Moreover, the scheme presented in this paper entails use of priority queues to estimate the path capacity. In absence of priority queues, we will have to

take up more frequent probing to estimate the path capacity. In such a scenario, we propose that we send N pairs of back-to-packets and the maximum estimated value from these values might be taken as the bottleneck link capacity. Further, this value of N can be calculated depending upon the estimate of the loss rate in the network. For example, if the loss estimate for the network is p then by sending at least $1/p$ probe packet pairs we can argue that at least one pair will be go through the network without getting dropped and we will have an estimate of the path capacity. However, if p is large then we will need to send lot of probe packets and thereby will increase the control traffic in the network.

Estimating effective end-to-end capacity is an active area of research [59, 47, 54, 50, 26]. However, all the current proposals incur an error of at least 10%. We believe we can leverage the current work in this area. However, to estimate demand we not only have to rely on estimates of effective capacity but also estimates of path capacity. As such, any errors in either estimates will effect our calculations about the path demand. Besides errors in estimation the proposed model may also suffer because the demand estimate and path capacity might not always correspond to the same physical bottleneck. However we can argue that our path capacity estimate will mostly likely identify the most constrained link in the network. As such, in multi-bottleneck scenarios we might be trying to match the demand of some bottleneck link to that of most constrained bottleneck link. As a result of this mis-match the most constrained link might continue to be a bottleneck. However, if we run a *vAVQ* for every path, then since its likely that with time we will match the demand to the respective bottleneck link capacities. This also follows from the premise that at any time we are trying to match the demand to some percentage (less than 100) of network capacity. As such, in steady state the total demand will be less than the network capacity and the queueing will be there to accommodate the burst arrivals. None the less, this is needs to investigated further.

Another configuration parameter on which the performance of *vAVQ* depends is the length of the virtual buffer. Our results have shown that a large value of virtual buffer queue will result in a marginal improvement in performance. This is because, if the virtual buffer is large then most likely there is always space to

enque incoming packets or in other words the queue will drain fast. As such, the virtual queue will rarely overflow and the network will operate with near full queues. On the other hand, very small values of virtual buffer length will result in frequent marking (or dropping) which might reduce the throughput and also cause considerable oscillations in the queue size. Thus, configuration of virtual buffer length is critical to good performance of *vAVQ*. In absence of reverse path congestion TCP will send two back-to-back packets and thus its likely that the burst size will be 2 packets. The results presented in Chapter 3 Section 3.12.4 further substantiates our hypothesis that the burst size in the network is 2 and rarely 3. As such taking into account statistical multiplexing and burst size, two packets buffer per flow per path will probably be an ideal virtual buffer length.

The proposal presented in this chapter conveys a modified price to the end-user, i.e., instead of communicating the end-to-end price we are now communicating a price which accrues out of matching path demand with path capacity. We conjecture that this modified price is representative of the maximum price on any link, on the source's path, in the network. This modified price will change the utility function which a source will be modifying. Since every utility function is associated with a certain kind of fairness, this modification of utility function results in a different fair distribution of equilibrium rates. We need to analytically analyze the *vAVQ* scheme to characterize the equilibrium rate allocation in the network. Further, through an analytical characterization of the model we can possibly find ways to configure the virtual buffer length and comment on the steady state behavior of the algorithm, specifically answer whether the proposed scheme is stable or not.

Current AQM proposals require deployment at all routers in the network, which is not only requires significant network upgrade but is also expensive. Moreover, the core routers in the network are very fast and requirement to perform queue management may slow them down beyond the line speed. Further, these routers were designed to only store and forward packets and this also fits well with the end-to-end design principle in the network. Also, this high speed of core routers makes them ill equipped to do per-flow or per-stream based calculations. On the other hand, edge routers are typically slow, have fewer flows (or streams) passing through them and

thus can easily manage flows. As such, the ability of *vAVQ* to operate at network edge and still match the input rate to the network capacity (and thus intrinsically perform queue management) makes it suitable for deployment. Further, as this thesis shows despite the limitations presented above, the initial results with *vAVQ* are very encouraging. We believe this is a line of research which has not been explored before and needs to be evaluated further.

6.5 Conclusions and Future Work

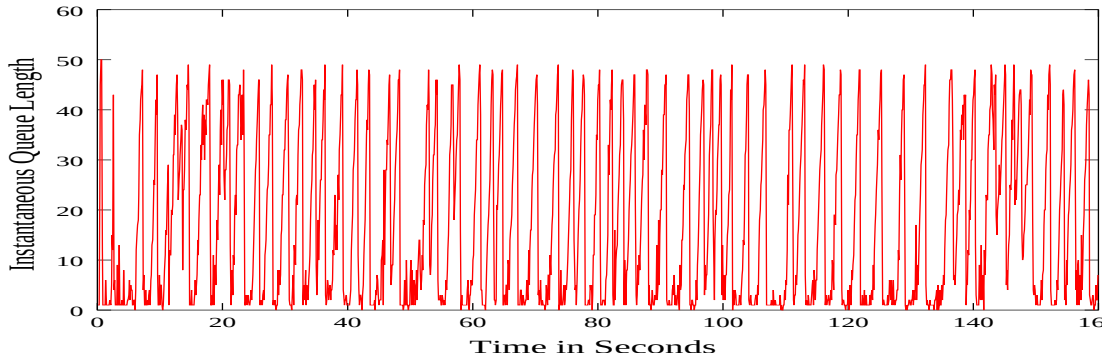
The Internet operates with TCP and Drop Tail queueing. Several studies have shown that TCP's and network's performance degrades on a network of Drop Tail queues. One of the main reasons behind this poor performance of Drop Tail queues is that it does not manage queues and thus often operates with near full queues.

In this chapter we have proposed a framework called *virtual AQM* to manage bottleneck queue from the network edge. The most interesting aspect of this proposal is that it does not require the placement of active queue management component at the router. Thus it allows the management of bottleneck queues on a network of Drop Tail queues. The framework uses packet probes to estimate the maximum demand and minimum capacity of a path. Thereupon, it uses an edge based AQM, based on AVQ, to match the demand estimates to some desired target network utilization. Since the desired network utilization is often (slightly) less than one, at steady state the input traffic in the network is less than the network capacity and as such the steady state queue sizes should be close to zero.

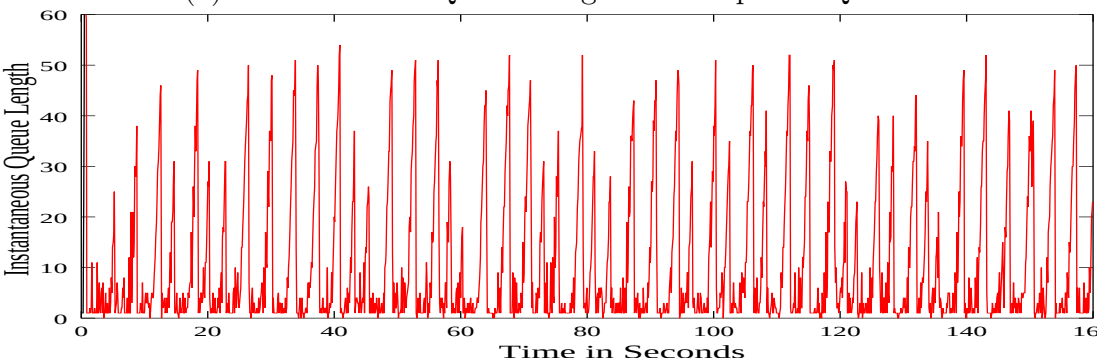
We implemented *virtual AQM* framework in NS and evaluated it for both single and multi-bottleneck topologies. Our initial results show that this framework can reduce steady state queue size on a network. However, the model proposed in this chapter is sensitive to errors in estimation of path capacity and demand. These errors are especially important in a multi-bottleneck scenario. Further, in a multi-bottleneck scenario the demand and path capacity estimate need not correspond to the same physical bottleneck. As such, in these situations the gains in reducing steady state queue size might not be as significant as those with single bottleneck. The evaluation of the model, as presented in this chapter, is by no means thorough.

However, this line of work entails very interesting possibilities and needs to be evaluated further.

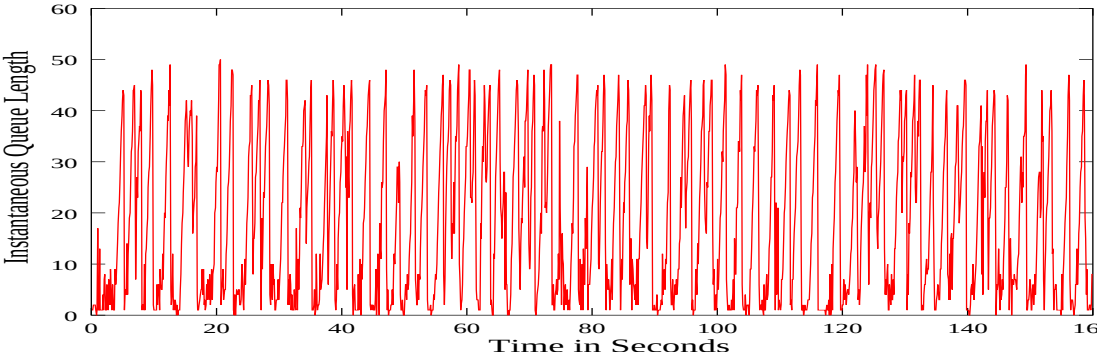
The framework presented in this chapter, *virtual AQM* shows that we can de-couple the task of management of bottleneck queues and it's placement. In other words, for managing queues in a network, we do not require an active queue management component to be present at every bottleneck. We show that using end-to-end packet probes and a *virtual AQM* module, we can manage the bottleneck queues at the network edge.



(a) Instantaneous Queue Length for Drop Tail Queue

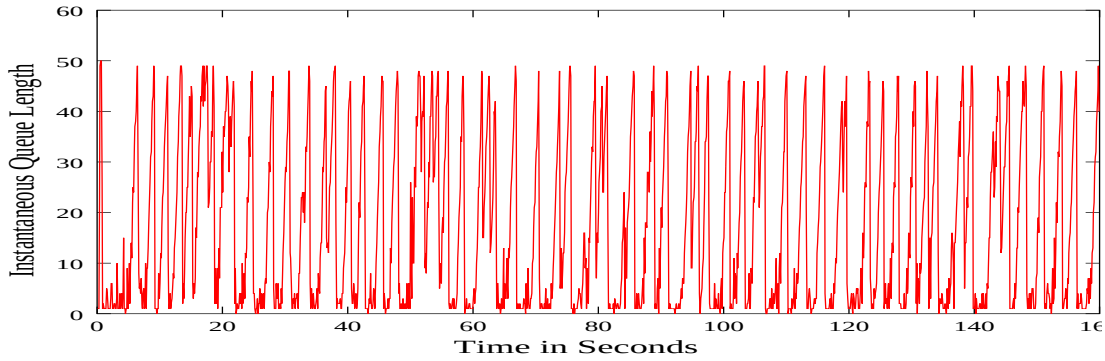


(b) Instantaneous Queue Length for AVQ

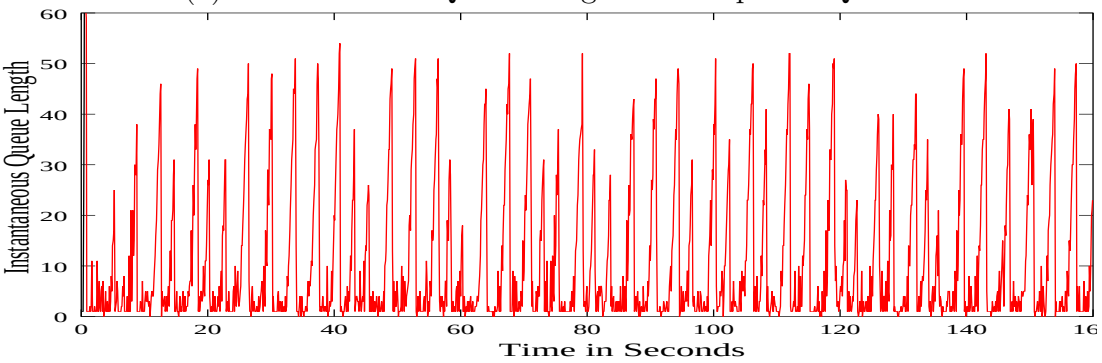


(c) Instantaneous Queue Length for vAVQ

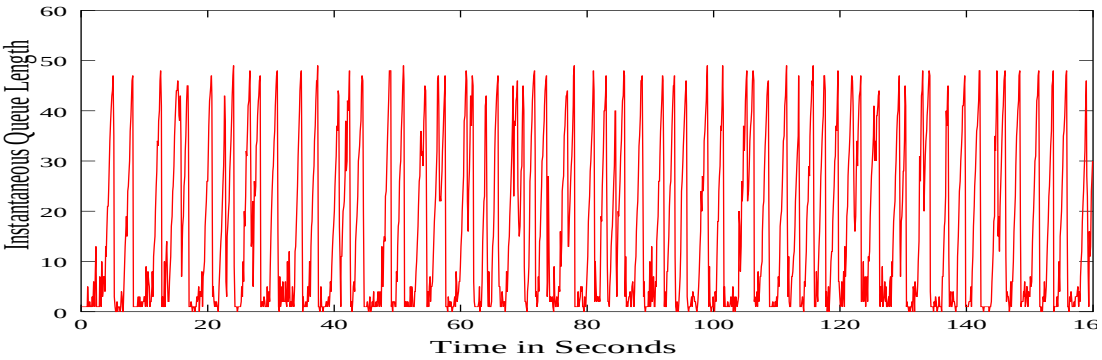
Figure 6.10: Multi-Bottleneck: Instantaneous Queue Length evolution at the first Bottleneck



(a) Instantaneous Queue Length for Drop Tail Queue



(b) Instantaneous Queue Length for AVQ



(c) Instantaneous Queue Length for vAVQ

Figure 6.11: Multi-Bottleneck: Instantaneous Queue Length evolution at the second Bottleneck

CHAPTER 7
Conclusions and Future Work

Congestion control has been the mainstay of the stability and robustness of the Internet. In 1980s, Internet suffered a series of failures which resulted in congestion collapse and the event is commonly referred to as Internet meltdown. These failure prompted increased efforts in understanding and solving the congestion control problem. As a result of this research, several end and network based solutions for congestion avoidance and control were proposed. TCP's congestion avoidance and control algorithm is one such end system based proposal and has fast become the most popular and widely used transport protocol. However, the other network based proposals for congestion avoidance and control have not been so favored and consequently lacked deployment on the Internet.

These network based proposals mainly include router based Active Queue Management (AQM) schemes for proactively managing bottleneck queues (and link). However these AQM schemes are beset with parameter configuration problem. This leads to implementational complexities and is one of the major reason why the proposed solutions are not implemented in real networks. In this thesis we have outlined *deployable* end-system and edge based architectures for congestion service which can bridge the gap between deployment concerns and desirability of congestion control functionality. In order to develop deployable solution it is *essential* to *separate* the congestion control tasks from their *placement* in the network. For example, management of selfish behavior is a congestion control task which is presently coupled with router based Active Queue Management (AQM) design. In contrast,we show that this task can be achieved at the end or the network edge by transparently engineering congestion penalties.

In this thesis we propose *Randomized TCP*, an end-system based solution, to emulate AQM behavior on a network of Drop Tail queues. Specifically, we propose to randomize the packet sending times in TCP. In Randomized TCP successive packets of a window are sent after an interval of $RTT(1+x)/cwnd$, where $cwnd$ is

the congestion window in packets and x is a random number drawn from an Uniform distribution on *[-1,1]*. This solution is distributed, can be implemented at the end systems and therefore is very attractive from an implementation perspective.

Randomized TCP introduces randomization in the network which helps break flow synchronization. Loss of synchronization thereupon results in lesser burst losses, reduction of phase effects, removal of Drop Tail's bias against flows with longer RTT flows and improvement in fairness in the network. We evaluated Randomized TCP for a variety of single and multi-bottleneck topologies. Our results show that a presence of even a *single* Randomized TCP at a bottleneck is helpful in improving performance of the network. Thus even an incremental deployment of Randomized TCPs would benefit the entire group of users. Finally, we extended the randomization of sending times to other window based protocols, primarily Binomial congestion control schemes. Again, randomization of sending times, improves the fairness in the network and allows different TCP Friendly Binomial schemes to share bandwidth equitably. Finally, through our proposal, Randomized TCP we show that we can *de-couple the need for introducing randomization in the network from AQM schemes*. Instead, we can do it through simple end system based modifications.

Though Randomized TCP improves performance of TCP and network there is a limit to how much control that can be achieved by end-system schemes, especially in a network which operates with disparate congestion control schemes. These different rate control schemes present us with the problem of congestion response conformance which manifests itself as smaller problems of fairness and management of selfish behavior in the network. As a first step towards addressing these issues, we first define selfish behavior and conformance. In this thesis, different rate control schemes are called conformant if they are maximizing the same utility function. In this thesis, we define TCP Friendly schemes as the conformant schemes. Thereafter we use this definition of conformance to define selfish end-system rate control schemes.

In this thesis we propose, *Uncooperative Congestion Control*, an edge system based re-marking framework to enforce congestion response conformance on the Internet. We achieve this by transparently *managing the effective range* of user's

utility functions. More specifically, users may choose arbitrary utility functions, but the edge of the network can *re-map* these utility functions into a target range of utility functions. This framework thus lets the network choose the target utility functions and thereby allows it to distribute resources amongst users according to some specified fairness criteria. Alternatively, this framework also considers providing fairness from a network's point of view and thus *effectively decouples the fairness from user's utility functions.*

The proposed framework can be implemented on the network edges and can work with either dropping or marking enabled network. Also the edge based re-marking is independent of the buffer management policies in the network and therefore works even with a network of Drop Tail queues. Moreover, the flexibility of the framework to map any utility function to any target utility function helps it provide broad range of fairness criteria. This edge based re-marking model also suggests that the *management of selfish flows in the network need not be necessarily coupled with AQM design*, instead it can be achieved by simple edge based modules. Finally the re-marking architecture proposed in this thesis can be thought of as a new class of traffic conditioning scheme and can be leveraged to provide service differentiation.

We have evaluated the edge based re-marking framework for a variety of single and multi bottleneck scenarios with both background web traffic and reverse path congestion. Our results show that the framework can map utility functions in all the cases, with either dropping or marking being used to convey penalties. However, a limitation of the model is that it may not work well in cases of path asymmetry. We believe that the uncooperative congestion control model, specifically the utility function transformation, can be used to manage malicious behavior of sources on the Internet; in particular, those constituting collusion of many sources (e.g., distributed denial-of-service). Flash attacks like Slashdot effect, coordinated attacks to bring down peer-to-peer or overlay networks and selfish behavior in online games are some examples of users colluding to exploit network to their advantage. Modeling the network and users as a multi-player cooperative game can capture such a behavior.

In this thesis we present an abstract framework for managing bottleneck queues from the network edge or end system. We conjecture that for any flow, through end-

to-end probes, we can identify the capacity of the congested link and use it to control the rate of the flow. Further, we can group flows according to the path they take in the network, find the congested link on that path and run an AQM at the network edge (ingress) to limit the rate of these flows. Moreover, any AQM schemes can be run at the edge to limit the rate of the flows (to the corresponding bottleneck). In this thesis we refer to this framework as *virtual AQM* (*vAQM*) and in this thesis we outline a specific algorithm, *virtual AVQ* (*vAVQ*), which uses AVQ to limit the rate of the flows at the network edge. The main advantage of this model is that the underlying network can still use Drop Tail queuing while allowing us to manage queues from network edges. The most interesting aspect of this proposal is that it *de-couples the placement of active queue management component at the router from the task of managing bottleneck queues.*

We have evaluated the *vAVQ* framework for both single and multi-bottleneck scenarios. Our initial results suggest that the proposed framework can significantly reduce bottleneck queue lengths without compromising on link utilization or fairness. However, the model presented in this thesis is sensitive to errors in estimation and the size of the virtual buffer. These errors becomes especially important in a multi-bottleneck scenario. Moreover, in a multi-bottleneck setup, the path capacity and demand estimates may not correspond to the same physical bottleneck. As a result, the gains with *vAVQ* in a multi-bottleneck scenarios is less than that in a single bottleneck setup. However, we believe that this is an area of research which has not been explored before and needs to be investigated further as it could lead to novel, interesting and deployable queue management algorithms.

[1] "Network simulator," http://www.isi.edu/nsnam.

[2] "Utime: Microsecond resolution timers for linux," http://www.ittc.ku.edu/utime/.

[3] A. A. Abouzeid and S. Roy, "Analytic understanding of red gateways with multiple competing tcp flows," in *IEEE Globecom*, 2000.

[4] A. Aggarwal, S. Savage, and T. Anderson, "Understanding the performance of tcp pacing," in *IEEE Infocom*, 2000.

[5] A. Akella, S. Seshan, R. Karp, S. Shenker, and C. Papadimitriou, "Selfish behavior and stability of the internet: A game theoretic analysis of tcp," in *ACM Sigcomm*, 2002.

[6] M. Aron and P. Druschel, "Tcp: Improving startup dynamics by adaptive times and congestion control," 1999, tR98-318, Rice Univ. Techreport.

[7] S. Athuraliya, S. H. Low, and D. E. Lapsley, "Random early marking," in *QofIS*, 2000.

[8] H. Balakrishnan, V. Padmanabhan, and R. Katz, "The effects of asymmetry in tcp performance," in *ACM/IEEE Mobicom*, 1997.

[9] D. Bansal and H. Balakrishnan, "Binomial congestion control scheme," in *IEEE Infocom*, 2000.

[10] J. C. R. Bennett and H. Zhang, "Worst case fair weighted fair queueing," in *IEEE Infocom*, 1996.

[11] D. P. Bertsekas and J. N. Tsitsiklis, "Parallel and distributed computation: Numerical methods," 1989, prentice-Hall.

[12] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An architecture for differentiated services," December 1998, iETF RFC 2475.

[13] T. Bonald and L. Massoulie, "Impact of fairness on internet performance," in *ACM Sigmetrics*, 2001.

[14] B. Braden and et. al., "Recommendations on queue management and congestion avoidance in the internet," April 1998, iETF RFC 2309.

[15] L. Brakmo and L. Peterson, "Tcp vegas: End to end congestion avoidance on a global internet," *IEEE Journal on Selected Areas in Communication*, 1995.

[16] K. Chandrayana, K. Avrachenkov, E. Altman, and C. Barakat, "Complementarity formulation for tcp/ip networks: Uniqueness of solution and relation with utility optimization," 2002.

[17] K. Chandrayana and S. Kalyanaraman, "On impact of non-conformant flows on a network of droptail gateways," in *IEEE Globecom*, 2003.

[18] ——, "Uncooperative congestion control," in *ACM Sigmetrics*, 2003.

[19] K. Chandrayana, S. Ramakrishnan, B. Sikdar, S. Kalyanaraman, A. Balan, and O. Tickoo, "On randomizing the sending times in tcp and other window based algorithm," *Conditional Accept for Journal of Computer Networks*, 2003.

[20] K. Chandrayana, B. Sikdar, and S. Kalyanaraman, "Scalable configuration of red queue parameters," in *IEEE High Speed ¡witching and Routing*, 2000.

[21] D. Chiu and R. Jain, "Analysis of the increase/decrease algorithms for congestion avoidance in computer networks," *Journal of Computer Networks and ISDN*, vol. 17, issue = 1, pp. 1–14, 1989.

[22] M. Christiansen, K. Jaffay, D. Ott, and F. D. Smith, "Tuning RED for web traffic," in *ACM Sigcomm*, 2000.

[23] A. Das, D. Dutta, and A. Helmy, "Fair stateless aggregate marking techniques using aqm techniques," in *IEEE/IFIP MMNS*, 2002.

[24] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queuing algorithms," in *ACM Sigcomm*, 1989.

[25] ——, "Analysis and simulation of fair queueing algorithm," in *ACM Sigcomm*, 1989.

[26] C. Dovrolis, P. Ramnathan, and D. Moore, "Packet dispersion techniques and capacity estimation," in *IEEE Infocom*, 2001.

[27] D. Dutta, A. Goel, and J. Heidemann, "Oblivious aqm and nash equilibria," in *IEEE Infocom*, 2003.

[28] W. Feng, D. D. Kandlur, D. Saha, and K. G. Shin, "Blue: A new class of active queue management algorithms," 1999, uM CSE-TR-387-99, Univ. of Michigan Techreport.

[29] W. C. Feng, D. D. Kandlur, D. Saha, and K. G. Shin, "A self-configuring red gateway," in *IEEE Infocom*, 1999.

[30] ——, "Stochastic fair blue: A queue management algorithm for enforcing fairness," in *IEEE Infocom*, 2001.

[31] W.-C. Feng, D. D. Kandlur, D. Saha, and K. G. Shin, "A self-configuring RED gateway," in *IEEE Infocom*, 1999.

[32] S. Floyd, "Connections with multiple congested gateways in packet-switched networks part 1: One-way traffic," *ACM Computer Communication Review*, vol. 21, pp. 30–47, 1991.

[33] ——, "Red: Discussions of setting parameters,," 1997, rED Homepage, http://www-nrg.ee.lbl.gov/floyd/REDparameters.txt.

[34] ——, "Congestion control principles," September 2000, iETF RFC 2914.

[35] S. Floyd and K. Fall, "Promoting the use of end-to-end congestion control in the internet," *IEEE/ACM Transactions on Networking*, vol. 7, pp. 458–472, 1999.

[36] S. Floyd, M. Handley, and E. Kohler, "Problem statement of dcp," 2002.

[37] S. Floyd and V. Jacobson, "On traffic phase effects in packet-switched gateways," *ACM Computer Communication Review*, vol. 21, 1992.

[38] ——, "On traffic phase effects in packet-switched gateways," *Internetworking: Research and Experience*, vol. 3, pp. 115–156, 1992.

[39] ——, "Random early drop gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, pp. 397–413, 1993.

[40] S. Floyd, J. P. M. Handley, and J. Widmer, "Equation-based congestion control for unicast applications," in *ACM Sigcomm*, 2000.

[41] R. Gibbens and F. Kelly, "Distributed connection acceptance control for a connectionless network," in *16th International Teletraffic Congress*, 1999.

[42] S. Gorinsky, S. Jain, H. Vin, and Y. Zhang, "Robustness to inflated subscription in multicast congestion control," in *ACM Sigcomm*, 2003.

[43] R. Gupta, M. Chen, S. McCanne, and J. Walrand, "Webtp: A receiver driven transport protocol," 1999, check it out.

[44] A. Habib and B. Bhargava, "Unresponsive flow detection and control using the differentiated services framework," 2002.

[45] E. Hashem, "Analysis of random drop for gateway congestion control," 1989, report LCS TR-465, MIT.

[46] C. V. Hollot, V. Misra, D. F. Towsley, and W. Gong, "A control theoretic analysis of RED," in *IEEE Infocom*, 2001.

[47] N. Hu and P. Steenkiste, "Evaluation and characterization of available bandwidth probing techniques," *IEEE Journal on Selected Areas in Communication*, vol. 21, 2003.

[48] S. Jacobs and A. Eleftheriadis, "Providing video services over networks without quality of service guarantees," in *RTMW*, 1996.

[49] V. Jacobson, "Congestion avoidance and control," in *ACM Sigcomm*, 1988.

[50] R. Jain and S. Routhier, "Packet trains: Measurements and a new model for computer network traffic," *IEEE Journal on Selected Areas in Communication*, vol. 4, pp. 1162–1167, 1986.

[51] S. Jin, L. Guo, I. Matta, and A. Bestavros, "TCP-friendly SIMD congestion control and its convergence behavior," Tech. Rep., 2001.

[52] J. Ke and C. Williamson, "Towards a rate based tcp protocol for the web," in *MASCOTS*, 2000.

[53] F. Kelly, A. Maulloo, and D. Tan, "Rate control in communication networks: Shadow prices, proportional fairness and stability," *Journal of the Operational Research Society*, vol. 49, pp. 237–252, 1998.

[54] S. Keshav, "A control-theoretic approach to flow control," in *ACM Sigcomm*, 1991.

[55] H. K. Khalil, "Non linear systems," pp. 100–101, 1996.

[56] S. Kunniyur and R. Srikant, "End-to-end congestion control: Utility functions, random losses and ecn marks," in *IEEE Infocom*, 2000.

[57] ——, "Analysis and design of an adaptive virtual queue (avq) algorithm for active queue management," in *ACM Sigcomm*, 2001.

[58] A. Kuzmanovic and E. Knightly, "Low-rate tcp-targeted denial of service attacks (the shrew vs. the mice and elephants)," in *ACM Sigcomm*, 2003.

[59] K. Lai and M. Baker, "Measuring link bandwidths using deterministic model of packet delay," in *ACM Sigcomm*, 2000.

[60] D. Lin and R. Morris, "Dynamics of random early detection," in *ACM Sigcomm*, 1997.

[61] S. Low, "A duality model of tcp and queue management algorithms," in *ITC Specialist Seminar on IP Traffic Measurement, Modeling and Management*, 2000.

[62] S. Low and D. Lapsley, "Optimization flow control, i: Basic algorithm and convergence," *IEEE/ACM Transactions on Networking*, vol. 7, pp. 861–975, 1999.

[63] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker, "Controlling high bandwidth aggregates in the network," *ACM Computer Communication Review*, 2002.

[64] R. Mahajan and S. Floyd, "Red-pd: Controlling high bandwidth flows at the congested router," 2001.

[65] A. Mankin, A. Romanow, S. Bradner, and V. Paxson, "Ietf criteria for evaluating reliable multicast transport and application protocols," June 1998, iETF RFC 2357.

[66] M. May, J. Bolot, C. Diot, and B. Lyles, "Reasons not to deploy red," in *ACM IWQoS*, 1999.

[67] M. May, T. Bonald, and J.-C. Bolot, "Analytic evaluation of red performance," in *IEEE Infocom*, 2000.

[68] M. Mehta, "On randomizing the sending times in tcp: An implementation," 2001, masters Thesis, ECSE, R.P.I.

[69] J. Mo and J. Walrand, "Fair end-to-end window-based congestion," *IEEE/ACM Transactions on Networking*, vol. 8, pp. 556–567, 2000.

[70] J. Moghul, "Observing tcp dynamics in real networks," in *ACM Sigcomm*, 1992.

[71] T. J. Ott, T. V. Lakshman, and L. Wong, "Sred: Stabilized red," in *IEEE Infocom*, 1999.

[72] Packeteer, "http://www.packeteer.com."

[73] J. Padhye, V. Firoiu, D. F. Towsley, and J. Kurose, "Modeling tcp reno performance: A simple model and its empirical validation," *IEEE/ACM Transactions on Networking*, vol. 8, pp. 133–145, 2000.

[74] J. Padhye and S. Floyd, "On inferring tcp behavior," in *ACM Sigcomm*, 2001.

[75] V. Padmanabhan and R. Katz, "Tcp fast start: A technique for speeding up web transfers," in *IEEE Globecom*, 1998.

[76] R. Pan, L. Breslau, B. Prabhakar, and S. Shenker, "Approximate fairness through differential dropping," 2001, http://www.research.att.com/*sim*breslau/papers/afd-techreport.ps.gz.

[77] R. Pan, B. Prabhakar, and K. Psounis, "Choke - a stateless queue management scheme for approximating fair bandwidth allocation," in *IEEE Infocom*, 2000.

[78] C. Partridge, "Ack spacing for high bandwidth-delay paths with insufficient buffering," 1998.

[79] N. Plotkin and P. Varaiya, "The entropy of traffic streams in atm virtual circuits," in *IEEE Infocom*, 1994.

[80] J. G. Proakis, C. M. Rader, F. Ling, M. Moonen, I. K. Proudler, and C. L. Nikias, "Algorithms for statistical signal processing," 2002, prentice-Hall.

[81] S. Raghunath, "Modeling the tcp loss rate process for efficient congestion control," Tech. Rep., 2002.

[82] R. Rejaie, M. Handley, and D. Estrin, "RAP: An end-to-end rate-based congestion control mechanism for real-time streams in the internet," in *IEEE Infocom*, 1999.

[83] I. Rhee, V. Ozdemir, and Y. Yi, "Tear: Tcp emulation at receivers – flow control for multimedia streaming," 2000, nCSU Technical Report.

[84] N. R. Sastry and S. S. Lam, "CYRF: A framework for window-based unicast congestion control," in *ICNP*, 2002.

[85] S. Shenker, "Fundamental design issues for the future internet," *IEEE Journal on Selected Areas in Communication*, vol. 13, 1995.

[86] S. Shenker, L. Zhang, and D. Clark, "Some observations on the dynamics of a congestion control algorithm," *ACM Computer Communication Review*, vol. 20, pp. 30–39, 1990.

[87] M. Shreedhar and G. Varghese, "Efficient fair queueing using deficit round robin," in *ACM Sigcomm*, 1995.

[88] M. Shreedhar and G. Verghese, "Efficient fair queueing using deficit round robin," in *ACM Sigcomm*, 1994.

[89] B. Sikdar, "Network traffic modeling and transmission control protocol," Ph.D. dissertation, R. P. I., 2001.

[90] B. Sikdar, K. Chandrayana, K. Vastola, and S. Kalyanaraman, "On reducing the degree of second order scaling in network traffic," in *IEEE Globecom*, 2002.

[91] ——, "Queue management algorithm and traffic self similarity," in *IEEE High Speed Switching and Routing*, 2002.

[92] W. Stevens, "Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms," 1997, iETF RFC 2001.

[93] I. Stoica, S. Shenker, and H. Zhang, "Core stateless fair queueing: Achieving approximately fair bandwidth allocations in a high speed networks," in *ACM Sigcomm*, 1998.

[94] ——, "Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks," in *ACM Sigcomm*, 1998.

[95] I. Stoica and H. Zhang, "Providing guaranteed services without per flow management," in *ACM Sigcomm*, 1999.

[96] V. Visweswaraiah and J. Heidemann, "Improving restart of idle tcp connections," 1997, tech Report 97-661, Univ. of South California.

[97] C. H. Xia and Z. Liu, "Queueing systems with long-range dependent input process and subexponential service times," in *ACM Sigmetrics*, 2003.

[98] M. Yajnik, S. Moon, J. Kurose, and D. Towsley, "Measurement and modeling of the temporal dependence in packet loss," in *IEEE Infocom*, 1999.

[99] Y. R. Yang, M. S. Kim, and S. S. Lam, "Transient behaviors of TCP-friendly congestion control protocols," in *IEEE Infocom*, 2001.

[100] C. You and K. Chandra, "Time series models for internet data traffic," in *24th Conf. on Local Computer Networks*, 1999.

[101] L. Zhang, S. Shenker, and D. Clark, "Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic," in *ACM Sigcomm*, 1991.