WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

IMPROVING THE PERFORMANCE OF INTERNET DATA TRANSPORT

by

Anshul Kantawala, B.S. M.S.

Prepared under the direction of Professor Jonathan S. Turner

A dissertation presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Doctor of Science

May, 2005

Saint Louis, Missouri

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

IMPROVING THE PERFORMANCE OF INTERNET DATA TRANSPORT

by Anshul Kantawala

ADVISOR: Professor Jonathan S. Turner

May, 2005

Saint Louis, Missouri

With the explosion of the World Wide Web, the Internet infrastructure faces new challenges in providing high performance for data traffic. First, it must be able to provide a fair-share of congested link bandwidth to every flow. Second, since web traffic is inherently interactive, it must minimize the delay for data transfer. Recent studies have shown that queue management algorithms such as Tail Drop, RED and Blue are deficient in providing high throughput, low delay paths for a data flow. Two major shortcomings of the current algorithms are: they allow TCP flows to get synchronized and thus require large buffers during congestion to enable high throughput; and they allow unfair bandwidth usage for shorter round-trip time TCP flows. We propose algorithms using multiple queues and discard policies with hysteresis at bottleneck routers to address both these issues. Using ns-2 simulations, we show that these algorithms can significantly outperform RED and Blue, especially at smaller buffer sizes.

Using multiple queues raises two new concerns: scalability and excess memory bandwidth usage caused by dropping packets which have been queued. We propose and evaluate an architecture using Bloom filters to evenly distribute flows among queues to improve scalability. We have also developed new intelligent packet discard algorithms that discard packets on arrival and are able to achieve performance close to that of policies that may discard packets that have already been queued.

Finally, we propose better methods for evaluating the performance of fair-queueing methods. In the current literature, fair-queueing methods are evaluated based on their worst-case performance. This can exaggerate the differences among algorithms, since the worst-case behavior is dependent on the the precise timing of packet arrivals. This work seeks to understand what happens under more typical circumstances.

to my parents, my wife and my daughter

Contents

List of Tables . . . . **vii**

List of Figures . . . . **ix**

Acknowledgments . . . . **xiii**

1 Introduction **1**

1.1 TCP Overview 2

1.1.1 Reliability 2

1.1.2 Congestion Control 3

1.2 High queueing delay due to large buffers 5

1.3 Unfairness due to differences in round-trip times 6

1.4 Evaluating Fair Queueing Algorithms 7

2 Interaction of TCP and Router Buffer Management **8**

2.1 TCP Flow Control and Tail Discard 8

2.1.1 Simple Buffer Analysis for TCP-like Flows 8

2.1.2 RED 14

2.2 TCP Fairness and Round-Trip Times (RTT) 16

3 Per-Flow Queueing **20**

3.1 Algorithms 20

3.2 Evaluation 22

3.3 Continuous TCP Flows 24

3.3.1 Single Bottleneck Link 24

3.3.2 Results 25

3.3.3 Multiple Round-Trip Time Configuration 28

3.3.4 Results 29

3.3.5 Multi-Hop Path Configuration 31
3.3.6 Results 31
3.4 Connections with Multiple Short Transfers 32
3.4.1 Single Bottleneck Link 33
3.4.2 Results 33
3.4.3 Multiple Roundtrip-time Configuration 34
3.4.4 Results 35
3.4.5 Multi-Hop Path Configuration 36
3.4.6 Results 36

4 A Closer Look at Queue State DRR 38
4.1 Simple Buffer Analysis for TCP Flows using QSDRR 38
4.2 Desychronizing effects of QSDRR 41
4.3 QSDRR with a Small Number of TCP Flows 47
4.4 Experimental Results 51
4.5 Usability over Large Networks 54
4.5.1 Simulation Setup 54
4.5.2 Results 55

5 Scalability Issues 58
5.1 Flow Distribution Algorithm 60
5.1.1 Overview of Bloom filters 60
5.1.2 Bloom filter architecture 61
5.1.3 Distribution Policy 62
5.1.4 Bloom filter queue ratios and memory usage 63
5.2 Dynamic rebalancing 65
5.2.1 Periodic Balancing (PB) 67
5.2.2 Balancing at Dequeue (DB) 68
5.3 Results 69
5.3.1 Flows arriving and leaving the system 70
5.3.2 Static number of flows 75

6 Packet Discard on Arrival 80
6.1 Memory Bandwidth Issues 80
6.2 Algorithms 81
6.3 Simulation Environment 83

6.3.1 Single Bottleneck Link . . . 84
6.3.2 Multiple Roundtrip-time Configuration . . . 84
6.3.3 Multi-Hop Path Configuration . . . 85
6.4 Results . . . 86
6.4.1 Single-Bottleneck Link . . . 86
6.4.2 Multiple Round-Trip Time Configuration . . . 89
6.4.3 Multi-Hop Path Configuration . . . 91
6.4.4 Scalability Issues . . . 94
6.4.5 Short-Term Fairness . . . 95

7 Metrics for Evaluating Fair-Queueing Algorithms . . . 97
7.1 Delay Performance . . . 98
7.2 Throughput Performance . . . 99
7.3 Single Target Flow Model . . . 100
7.3.1 Delay Performance . . . 101
7.3.2 Throughput Performance . . . 102

8 Related Work . . . 106
8.1 Other Buffer Management Policies . . . 106
8.1.1 FRED . . . 106
8.1.2 Self-Configuring RED . . . 107
8.1.3 TCP with per-flow queueing . . . 107
8.2 TCP/Active Queue Management (AQM) Analysis . . . 107
8.3 Bloom Filters . . . 108
8.3.1 Queue Management: Stochastic Fair Blue (SFB) . . . 109
8.3.2 Traffic Measurement and Accounting . . . 109
8.4 Shared-Memory Buffer Management . . . 110
8.5 Fair Queueing (FQ) . . . 111

9 Conclusions and Future Work . . . 112

References . . . 114

Vita . . . 120

List of Tables

2.1 Definitions of terms used in analysis 9
2.2 Comparing queue drop and cycle time obtained using the analytical model vs. simulation data 14
2.3 Throughput variations when changing RED's dropping probability 16
2.4 Average Throughputs for 40 ms RTT and 200 ms RTT flows under Tail Discard and RED 19

3.1 RED parameters 23
3.2 Blue parameters 23

4.1 Definitions of terms used in analysis 39
4.2 Number of flows affected during the first drop interval, comparing the analytical and simulation values 41

5.1 SRAM memory needed for per-flow queueing vs. Bloom filter architecture for 100,000 flows 62
5.2 Max/min flows queue ratios for Bloom architecture vs. simple hashing . . . 63
5.3 Max/min flows queue ratios for Bloom architecture with Periodic, Dequeue and both dynamic rebalancing vs. simple hashing for dynamic number of flows 74
5.4 Max/min flows queue ratios for Bloom architecture with Periodic, Dequeue and both dynamic rebalancing vs. simple hashing for dynamic number of flows 74
5.5 Max/min flows queue ratios for Bloom architecture with Periodic, Dequeue and both dynamic rebalancing vs. simple hashing for dynamic number of flows 75
5.6 Max/min flows queue ratios for Bloom architecture with Periodic, Dequeue and both dynamic rebalancing vs. simple hashing for static number of flows 78

5.7 Max/min flows queue ratios for Bloom architecture with Periodic, Dequeue and both dynamic rebalancing vs. simple hashing for static number of flows 78
5.8 Max/min flows queue ratios for Bloom architecture with Periodic, Dequeue and both dynamic rebalancing vs. simple hashing for static number of flows 79
6.1 Discard queue time statistics 96

List of Figures

2.1 Bottleneck queue behaviour for Tail Discard with buffer size 4167 packets (round-trip delay times bandwidth) 9
2.2 Single Bottleneck Link Network Configuration 12
2.3 Comparison of analytical queueing behaviour with simulation results 13
2.4 Comparison of analytical queueing behaviour with simulation results when queue underflows 13
2.5 Buffer Occupancy Time History for RED with changing Drop Probability . 15
2.6 Average TCP Congestion Window Sizes for TCP sources with different RTT using Tail Discard 17
2.7 Average TCP Congestion Window Sizes for TCP sources with different RTT using RED 18

3.1 Algorithm for QSDRR 22
3.2 Single Bottleneck Link Network Configuration 24
3.3 TCP Reno Goodput distribution over single-bottleneck link with 200 packet buffer 26
3.4 Standard deviation relative to fair-share for TCP Reno flows over a single-bottleneck link 27
3.5 Fair share performance over a single bottleneck link 27
3.6 Fair share performance of TDRR over a single bottleneck link with varying exponential weights 29
3.7 Multiple Round-Trip Time Network Configuration 29
3.8 Fair share performance of different RTT flows over a single bottleneck link 30
3.9 Multi-Hop Path Network Configuration 31
3.10 Fair Share performance of end-to-end and cross traffic flows over a multi-hop path configuration 32
3.11 Single Bottleneck Link Network Configuration 33

3.12 Performance of short burst TCP flows over a single bottleneck link 34
3.13 Multiple Roundtrip-time Network Configuration 34
3.14 Performance of short burst TCP flows over a multiple round-trip time configuration 35
3.15 Multi-Hop Path Network Configuration 36
3.16 Performance of short burst TCP flows over a multi-hop path configuration . 37

4.1 Queue lengths for 10 flows with buffer size = 417 pkts 41
4.2 Queue lengths for 10 flows with buffer size = 50 pkts 42
4.3 Queue lengths for 10 flows with buffer size = 100 pkts 43
4.4 Packet drop trace for QSDRR compared to Tail Drop for a fair share window size of 5 packets over a single bottleneck link 44
4.5 Packet drop trace for QSDRR compared to Tail Drop for a window size of 25 packets over a single bottleneck link 45
4.6 Packet drop trace for QSDRR compared to Tail Drop for a window size of 5 packets over a multiple-RTT configuration 46
4.7 Packet drop trace for QSDRR compared to Tail Drop for a window size of 25 packets over a multiple-RTT configuration 47
4.8 Fair share performance of a small number of TCP flows over a single bottleneck link 48
4.9 Fair share performance of 2 TCP flows with different RTTs over a single bottleneck link 49
4.10 Fair share performance of 4 TCP flows with different RTTs over a single bottleneck link 49
4.11 Fair share performance of 8 TCP flows with different RTTs over a single bottleneck link 50
4.12 Time history of congestion window sizes for 4 TCP flows using Tail Drop . 50
4.13 Experimental configuration using two Multi-Service Routers and four NetBSD hosts 51
4.14 Experimental fair share performance and standard deviation for 32 TCP flows over a single bottleneck link 52
4.15 Experimental fair share performance and standard deviation for 48 TCP flows over a single bottleneck link 53
4.16 Experimental fair share performance and standard deviation for 60 TCP flows over a single bottleneck link 53

4.17 Single Bottleneck Link Network Configuration 54
4.18 Performance of TCP flows over single bottleneck link while varying the bottleneck link 55
4.19 Performance of TCP flows over single bottleneck link while varying the RTT 56
4.20 Performance of TCP flows over single bottleneck link while varying the number of sources 56

5.1 Performance of TDRR and QSDRR for a buffer size of 1000 packets, with varying number of buckets 59
5.2 Flow distribution using Bloom Filters 60
5.3 Algorithm for distributing Bloom filters among queues 62
5.4 Max/min queue ratios for Bloom filter architecture compared with simple hashing for flow to queue ratios from 1 to 1,000 64
5.5 Memory usage for Bloom filter policies compared with per-flow queueing for flow to queue ratios from 1 to 1,000 65
5.6 Max/min queue ratios for Bloom filter architecture without dynamic rebalancing compared with simple hashing 66
5.7 Policy for updating Bloom filter counts 67
5.8 Algorithm for periodically balancing flows across queues 68
5.9 Algorithm for balancing flows during dequeue 69
5.10 Flows arriving and departing at a rate of 10 flows/s 70
5.11 Flows arriving and departing at a rate of 20 flows/s 71
5.12 Flows arriving and departing at a rate of 50 flows/s 72
5.13 Flows arriving and departing at a rate of 100 flows/s 73
5.14 Mean destination hold time = 2s 76
5.15 Mean destination hold time = 5s 77

6.1 Algorithm for DSDRR 82
6.2 Single Bottleneck Link Network Configuration 84
6.3 Multiple Roundtrip-time Network Configuration 85
6.4 Multi-Hop Path Network Configuration 85
6.5 Standard deviation relative to fair-share for long-lived TCP Reno flows over a single-bottleneck link 86
6.6 Fair share performance for long-lived TCP Reno flows over a single bottleneck link 87
6.7 Ratio of maximum to minimum flow throughputs 88

6.8 Performance of short burst TCP flows over a single bottleneck link 89
6.9 Fair share performance of different RTT long-lived TCP flows over a single bottleneck link 90
6.10 Performance of short burst TCP flows over a multiple round-trip time configuration 91
6.11 Fair Share performance of long-lived TCP flows over a multi-hop path configuration 92
6.12 Performance of short burst TCP flows over a multi-hop path configuration . 93
6.13 Performance of DTDRR and DSDRR for a buffer size of 1000 packets, with varying number of buckets 94
6.14 Distribution of queue discard times for DSDRR and QSDRR 96

7.1 Average and Maximum Delays for small (1 Mb/s) flows 98
7.2 Average and Maximum Delays for the large (10 Mb/s) flow 99
7.3 Time taken for flows to reach within 10% of their fair-share bandwidth . . . 100
7.4 Average delay experienced by the 5 Mb/s target CBR flow 101
7.5 Maximum delay experienced by the 5 Mb/s target CBR flow 103
7.6 Time taken for 5 Mb/s target flow to reach within 10% of its fair-share bandwidth 104

Acknowledgments

First, I would like to acknowledge my advisor, Prof. Jonathan Turner for his extraordinary guidance and patience which enabled me to achieve this goal. I would also like to thank Dr. Guru Parulkar, who was my advisor during my undergraduate years and my initial thesis advisor and has been gracious enough to be on my D.Sc. committee. He was instrumental in guiding me during the earlier years and his encouragement and help have gone a long way towards my success. I also thank the rest of my committee for their help and insight in my research.

Finally, I would like to thank my parents and my wife for their undying support, encouragement, love and patience that propelled me to this goal. I also thank all my friends and colleagues in the department who made my life so much more interesting and enjoyable during my tenure here as a graduate student!

Anshul Kantawala

Washington University in Saint Louis
May 2005

Chapter 1

Introduction

Given the current explosive growth of the Internet, one of the main challenges is to improve performance of data transport in the Internet in the presence of both transient and sustained congestion. The key issues pertaining to data transport performance in the Internet are:

- Ensuring goodput when bandwidth is limited.
- Minimizing queueing delay at a router without underutilizing the link.
- Providing a fair-share of the bandwidth resources amongst competing flows.
- Adapting to changing traffic behaviour and available bandwidth.

Millions of flows may traverse an Internet router at any given time. If each source sent at its fair share, the router would not need extensive buffers or complex packet scheduling policies to manage the traffic. In practice, sources tend to be greedy, causing the load on the links to fluctuate. The traditional role of buffer space in an Internet router is to absorb the transient imbalances between offered load and capacity. Choosing the correct buffer size is something of an art: too small risks high loss rates during transient congestion and low utilization, too large risks long queueing delays. In this thesis, we concentrate on buffer management in the network data path for improving the data transport performance. The first part of this work proposes buffer management algorithms that can provide high throughput for TCP flows using very small buffers while providing a fair-share of the bandwidth resources to each flow regardless of differences in round-trip times and hop counts. The second part of this work proposes new methods for evaluating fair-queueing algorithms.

1.1 TCP Overview

Before we discuss the problems in further detail, we present a brief overview of the TCP protocol. TCP is a reliable transport protocol that operates on top of IP. Along with providing an acknowledgment (ACK) based reliable data transfer, it implements a window based congestion control algorithm. The subsections below provide a brief overview of the TCP protocol.

1.1.1 Reliability

TCP provides reliability by doing the following:

- The application data is broken up into chunks called *segments*. The maximum sized segment (MSS) a TCP connection can send is negotiated during TCP connection setup.
- When TCP sends a segment it maintains a timer, waiting for the other end to acknowledge reception of the segment. If an acknowledgement is not received, the segment is retransmitted.
- When TCP receives a segment from the other end, it sends an acknowledgement (ACK). This ACK may be delayed for a fraction of a second in the hope that there is some data to send in the same direction as the ACK. This allows the receiver to *piggyback* the ACK with the data.
- TCP maintains an end-to-end checksum on its header and data in order to detect any modification of the data in transit.
- Since TCP segments are transmitted as IP datagrams, they can arrive out of order. A TCP receiver must resequence the data if necessary. A TCP receiver is also responsible for discarding duplicate segments.
- TCP also provides flow control. At connection set-up, each TCP receiver advertises a maximum window size it can handle. This window size corresponds to the amount of buffer space it has for the connection.

1.1.2 Congestion Control

In this section, we briefly present mechanisms used by TCP for congestion control. TCP congestion control consists of two main components: **slow start** and **congestion avoidance**. A TCP sender uses the size of the sending window to provide flow control as well as congestion control.

Slow Start

Slow start is a mechanism used by the TCP sender to gauge the amount of network bandwidth available for the connection. The TCP sender maintains a window, called the *congestion window (cwnd)*, for slow start and other congestion control mechanisms. When a new connection is established, *cwnd* is initialized to one segment. Each time an ACK is received (for a packet of any size), *cwnd* is increased by one segment. Although *cwnd* is maintained in bytes, slow start always increments it by the segment size. The sender starts by transmitting one segment and waiting for its ACK. When the ACK is received, *cwnd* is incremented from one to two, and two segments can be sent. When both of those segments are acknowledged, *cwnd* is increased to four. Thus, slow start provides an exponential increase in the sending rate.

At some point, the capacity of the connection path is reached and the intermediate router will start discarding packets. This tells the sender that the window has grown too large and the sender enters *congestion avoidance*. The sender can transmit up to the minimum of *cwnd* and the advertised window. Thus, *cwnd* serves as the sender imposed flow control while the advertised window is the receiver imposed flow control.

Congestion Avoidance

Slow start is used by TCP to initiate data flow in a connection. However, once the connection exceeds the capacity of an intervening router, there will be dropped packets. TCP uses a congestion avoidance algorithm to deal with lost packets. We briefly describe the congestion avoidance algorithm and point out the differences between TCP Tahoe and TCP Reno implementations. This algorithm is described in detail in [35].

Congestion avoidance and slow start require two variables to be maintained for each connection: a congestion window, *cwnd*, and a slow start threshold, *ssthresh*. The algorithm works as follows:

- Initially, *cwnd* is set to one segment and *ssthresh* to 65535 bytes.

- Congestion is detected via a timeout or duplicate ACKs. Since TCP sends out a cumulative ACK for the segments received in order, if a packet is dropped by an intermediate router, packets following the dropped packet will cause the receiver to generate duplicate ACKs.

- When congestion is detected (either via a timeout or duplicate ACKs), *ssthresh* is set to one-half of the current window size (the minimum of *cwnd* and the receiver's advertised window). If congestion is detected due to duplicate ACKs, *cwnd* is set to *ssthresh* in TCP Reno and set to one segment (slow start) in TCP Tahoe. If congestion is detected via a timeout, *cwnd* is set to one segment (slow start) in both TCP Reno and TCP Tahoe.

- When new data is acknowledged, the sender increases *cwnd*, but the way it increases depends on whether the source is performing slow start or congestion avoidance. If *cwnd* is less than *ssthresh*, the source performs slow start, otherwise it does congestion avoidance.

  On entering slow start, the source sets *cwnd* to one segment and while in slow start, it increases *cwnd* by one segment every time an ACK is received. In congestion avoidance, the source increases *cwnd* by (MSS*MSS)/*cwnd* every time an ACK is received. Thus, *cwnd* will increase by at most one segment every round-trip time. This is an additive increase compared to slow start's exponential increase.

- If three or more duplicate ACKs are received in a row, the sender concludes that there was a packet loss and immediately retransmits what appears to be the missing segment without waiting for a timeout. This is called the *fast retransmit* algorithm. The retransmission is followed by slow start in TCP Tahoe and congestion avoidance TCP Reno (this is referred to as the fast recovery feature of TCP Reno). The *fast recovery* algorithm is briefly outlined below:

  1. When the third duplicate ACK is received, *ssthresh* is set to one-half the current value of *cwnd*. The missing segment is retransmitted and *cwnd* is set to *ssthresh* plus 3 times the segment size.
  2. Each time another duplicate ACK is received, *cwnd* is incremented by one segment and a new packet is transmitted if allowed by the new value of *cwnd*.
  3. When the next ACK arrives acknowledging new data, *cwnd* is set to *ssthresh* and the sender enters congestion avoidance as described above.

1.2 High queueing delay due to large buffers

Routers today primarily use First-In-First-Out (FIFO) queues with a Tail Drop packet drop policy. A Tail Drop policy means that when the queue is full, a new incoming packet is dropped on arrival. With such a drop policy, TCP flows tend to get synchronized during congestion and need a buffer size comparable to the link bandwidth-delay product to ensure high throughput and minimize the chance of underutilization. Thus, backbone routers in the Internet are typically configured with buffers that are several times larger than the product of the link bandwidth and the typical round-trip delay on long network paths. Such buffers can delay packets for as much as half a second during congestion periods. When such large queues carry heavy traffic loads, and are serviced using the Tail Drop policy, the large queues remain close to full much of the time. Thus, even if each flow is able to obtain its share of the link bandwidth, the end-to-end delay remains very high. This is exacerbated for flows with multiple hops, since packets may experience high queueing delays at each hop. This phenomenon is well-known and has been discussed by Hashem [34] and Morris [51], among others. This prevents us from realizing one of the key benefits of high bandwidth links, which is lower queueing delays.

To address this issue, researchers have developed alternative queueing algorithms which try to keep average queue sizes low, while still providing high throughput and link utilization. The most popular of these is *Random Early Discard* or RED [30]. RED maintains an exponentially-weighted moving average of the queue length which is used to detect congestion. When the average crosses a minimum threshold (min_{th}), packets are randomly dropped or marked with an explicit congestion notification (ECN) bit. When the queue length exceeds the maximum threshold (max_{th}), all packets are dropped or marked. RED includes several parameters which must be carefully selected to get good performance. To make it operate robustly under widely varying conditions, one must either dynamically adjust the parameters or operate using relatively large buffer sizes [23, 64]. Another queueing algorithm called Blue [27], was proposed to improve upon RED. Blue adjusts its parameters automatically in response to queue overflow and underflow events. When the buffer overflows, the packet dropping probability is increased by a fixed increment ($d1$) and when the buffer empties (underflows), the dropping probability is decreased by a fixed increment ($d2$). The update frequency is limited by a *freeze_time* parameter. Incoming packets are then randomly dropped or marked with an ECN bit. Although Blue does improve over RED in certain scenarios, its parameters are also sensitive to different congestion conditions and network topologies.

Although RED and Blue try to alleviate the synchronization problem by using a random drop policy, they do not perform well with buffers that are smaller than the bandwidth-delay product. When buffers are very small, even with a random drop policy, there is a high probability that all flows suffer a packet loss.

1.3 Unfairness due to differences in round-trip times

Since TCP uses a window-based flow and congestion control algorithm, whereby each flow can only have a window-size number of outstanding bytes, flows with larger window sizes achieve higher throughput. Also, this window size is incremented when packets are successfully acknowledged. Thus, flows with a shorter round-trip time (RTT) can increase their window size faster and thus achieve a higher throughput compared to flows with a longer RTT which are sharing the same bottleneck link.

Current buffer management policies such as Tail Discard, RED and Blue use random dropping and thus cannot differentiate flows based on their RTTs. This leads to shorter RTT flows grabbing an unfair share of the bottleneck bandwidth.

To tackle the above issues, we investigate queueing algorithms that use multiple queues, to isolate flows from one another. While algorithms using multiple queues have historically been considered too complex, continuing advances in technology have made the incremental cost negligible, and well worth the investment if these methods can reduce the required buffer sizes and resulting packet delays. We show, using ns-2 simulations, that the proposed queueing algorithms represent major improvements over existing methods for both the issues described above.

Although per-flow queueing helps in desynchronizing TCP flows and provides fair-share throughput to flows with different RTTs, it raises two new issues:

1. **Scalability**
   Per-flow queueing policies require a certain amount of memory to store queue pointers and flow filters. These fields are typically stored in SRAM, since the memory access speeds are critical for high performance routers. Although this memory may be small in comparison to the total buffer memory, there is some legitimate concern about the cost associated with using a large number of queues.

2. **Memory bandwidth wastage**
   Standard per-flow fair-queueing algorithms usually drop packets that have already been queued which can require significantly more memory bandwidth than policies

that drop packets on arrival. In high performance systems, memory bandwidth can become a key limiting factor.

1.4 Evaluating Fair Queueing Algorithms

In the Internet, along with TCP flows, UDP flows also share the link bandwidth. These flows may be reserved flows for applications requiring some Quality of Service (QoS) or unreserved datagram traffic. There has been a lot of work recently developing work-conserving fair-queueing (FQ) algorithms for reserved flows (UDP traffic). We would like to evaluate how our buffer management algorithms compare against other FQ algorithms. The usual evaluation method used for FQ algorithms is worst-case delay analysis. Although the analytical methods provide worst-case delay bounds, they tend to exaggerate the differences since they rely on precise timings of individual packet arrivals.

Until now, there has not been a concerted effort to develop a simulation framework for evaluating different FQ algorithms over realistic network configurations and traffic patterns. In one previous study [37], the authors present a simulation study of hierarchical packet fair queueing algorithms. The simulation study bolsters our claim by showing that there is a significant gap between the worst delays obtained via simulation for non-WF^{2}Q and what we would expect from the analytical bounds. Although this study is a step in the right direction, the simulation scenarios are fairly limited and concentrate only on the delay metric. We propose to pursue a more systematic and complete study of different metrics for evaluating FQ algorithms.

The rest of the thesis is organized as follows. Chapter 2 illustrates the interaction between TCP flow control and the Tail Discard dropping policy. We present an approximate analysis of the queueing behaviour for a congested bottleneck-link buffer with TCP-like flows. We also describe the effect of different RTTs on a TCP flow's congestion window size and its throughput. Chapter 3 discusses the per-flow queueing algorithms we propose for improving performance over a congested link along with simulation results. Chapter 4 presents a detailed study of QSDRR's properties and experimental results. In Chapter 5, we discuss scalability issues and present an architecture for distributing flows among queues using Bloom filters. Chapter 6 presents algorithms for intelligent packet discard on arrival to minimize excess memory bandwidth usage. Chapter 7 outlines the current issues with evaluating fair-queueing algorithms and presents results derived from a new set of simulation configurations. Chapter 8 presents a brief summary of related work and Chapter 9 has the conclusions and future work.

Chapter 2

Interaction of TCP and Router Buffer Management

2.1 TCP Flow Control and Tail Discard

When a congested link drops packets, TCP flow control reacts by cutting its congestion window size in half. For congested links using a Tail Drop packet drop policy, what we observe is that TCP sources tend to get synchronized and the buffer occupancy exhibits a high degree of oscillation. This happens due to the TCP sources cutting their congestion window to half at the same time due to near simultaneous packet drops, caused by the full buffer. Figure 2.1 illustrates the buffer occupancy oscillation for TCP flows with a Tail Drop packet discard policy. For additional insight into this issue, we present an analysis of a simplified protocol similar to TCP in its congestion control behaviour in the following section.

2.1.1 Simple Buffer Analysis for TCP-like Flows

In this section, we present a simplified analysis of buffer occupancy of a congested bottleneck link using the Tail Drop packet discard policy in the presence of a large number of TCP Reno flows. The motivation for this analysis is not to provide a precise model for TCP, but to help develop some elementary insights into its behaviour. This insight is useful in understanding the much more detailed simulation results presented in later sections. The assumptions we make for the analysis are:

- The analysis is a discrete time analysis considering one RTT as a unit.

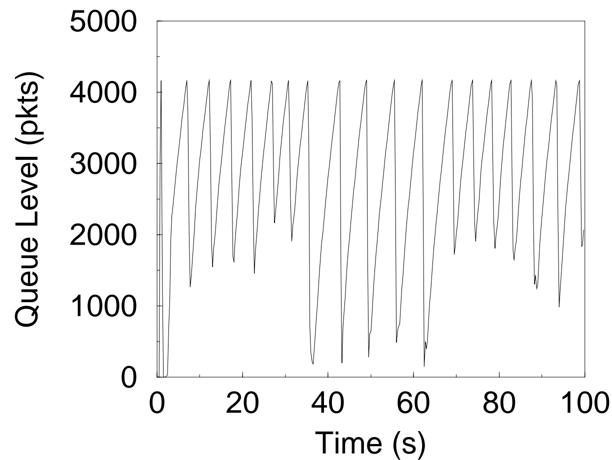


Figure 2.1: Bottleneck queue behaviour for Tail Discard with buffer size 4167 packets (round-trip delay times bandwidth)

Table 2.1: Definitions of terms used in analysis

N	Number of sources
R	Link rate in packets/second
B	Buffer size
T_0	Round-trip time when queue is empty
$T(b)$	Round-trip time when queue level is b
W_0	Fair-share window-size for each TCP flow when queue is empty
W^*	Fair-share window-size for each TCP flow when queue is full
α	Probability that a source experiences a packet drop in RTT when buffer becomes full

- The Tail drop policy is used in the bottleneck link buffer.
- All TCP flows are in congestion avoidance mode.
- All packet drops are indicated by duplicate ACKs.
- All TCP flows are synchronized initially (i.e. they have the same *cwnd* value) at the time of buffer overflow.
- In each RTT period, we assume a fluid model for the TCP data traffic. During each RTT period, all TCP flows transmit their entire *cwnd* and receive ACKs for all successfully received packets.

Table 2.1 describes the notation we use in the analysis. We present an approximate calculation of the drop in queue level after a buffer overflow and the time between consecutive buffer overflow events. We note that given our assumptions, the queue overflows when each source sends $W^* + 1$ packets (i.e. *cwnd* for each source is $W^* + 1$).

From our assumptions, we have:

$$W_0 = \frac{RT_0}{N} \tag{2.1}$$

$$W^* = W_0 + \frac{B}{N} = \frac{(RT_0 + B)}{N} \tag{2.2}$$

$$T(b) = T_0 + \frac{b}{R} \tag{2.3}$$

$$T(B) = T_0 + \frac{B}{R} = \frac{NW^*}{R} \tag{2.4}$$

In the first RTT after overflow, let the expected number of sources experiencing a packet drop equal αN. We expect α to be between 1/2 and 1. For these αN sources, the new value of *cwnd* is equal to $\frac{W^*+1}{2}$, since these sources experience a drop and will reduce their *cwnd* by half. For $(1-\alpha)N$ sources, the new value of *cwnd* is equal to $W^* + 2$, since these sources do not experience any drops and thus will increase their *cwnd* by one each RTT. Now, in this RTT, the number of packets drained from the buffer is $T(B) \cdot R$ and the number of packets sent by the sources is $\alpha N \left(\frac{W^*+1}{2}\right) + (1-\alpha)N(W^* + 2)$. Hence, queue level after first RTT is:

$$\begin{aligned} Q_1 &= B - T(B)R + \alpha N \left(\frac{W^* + 1}{2}\right) + (1-\alpha)N(W^* + 2) \\ &= B - NW^* + \frac{\alpha}{2}NW^* + \frac{\alpha}{2}N + NW^* + 2N - \alpha NW^* - 2\alpha N \\ &= B - N\left[\frac{\alpha}{2}(W^* + 3) - 2\right] \end{aligned} \tag{2.5}$$

Since $RT_0 + B = NW^*$, we can substitute $NW^* - RT_0$ for B giving

$$Q_1 = N\left[\left(1 - \frac{\alpha}{2}\right)W^* + \left(2 - \frac{3\alpha}{2}\right)\right] - RT_0 \tag{2.6}$$

In the second RTT after overflow, the number of packets drained from the buffer is $T(Q_1) \cdot R$ and the number of packets sent by the sources is $\alpha N \left(\frac{W^*+1}{2} + 1\right) + (1 - \alpha)N(W^* + 3)$. Hence, the queue level after the second RTT is:

$$Q_2 = Q_1 - T(Q_1) \cdot R + \alpha N \left(\frac{W^* + 1}{2} + 1\right) + (1 - \alpha)N(W^* + 3) \tag{2.7}$$

Note that

$$\begin{aligned} T(Q_1) &= T_0 + \frac{Q_1}{R} \\ &= T_0 + \frac{N\left[\left(1 - \frac{\alpha}{2}\right)W^* + \left(2 - \frac{3\alpha}{2}\right)\right] - RT_0}{R} \end{aligned} \tag{2.8}$$

Substituting Equation 2.8 into Equation 2.7 we get,

$$Q_2 = Q_1 + N \tag{2.9}$$

From the second RTT onward, each RTT, the queue level increases by N packets until the queue is full. Thus, the total number of steps (RTTs) from the second RTT until the queue is full again is given by:

$$S = \frac{B - Q_1}{N} \tag{2.10}$$

$$= \frac{\alpha}{2}(W^* + 3) - 2 \tag{2.11}$$

The time taken for the first step (RTT) = $T(B)$ and for the second step = $T(Q_1)$. After the second RTT, for each step, the RTT increases by a fixed constant = N/R. Also, the time for the last step (RTT) = $T(B)$. Thus, the average time for each step from the second RTT to the last = $\frac{T(Q_1)+T(B)}{2}$

The total time taken from the time the queue overflows to the next time the queue is full is:

$$\begin{aligned} T_{total} &= T(B) + S \cdot \left[\frac{T(Q_1) + T(B)}{2}\right] \\ T_{total} &= T(B) \cdot \\ & \left[\left(\frac{\alpha}{2} - \frac{\alpha^2}{8}\right)W^* - \left(2 - 3\alpha + \frac{9\alpha^2}{8}\right)\frac{1}{W^*} - \left(1 - \frac{5\alpha}{2} + \frac{3\alpha^2}{4}\right)\right] \end{aligned} \tag{2.12}$$

The main thing to take away from this analysis is that the time between the periods when the buffer overflows is roughly proportional to the product of $T(B)$ and W^* when W^* is not too small and α is between 1/2 and 1.

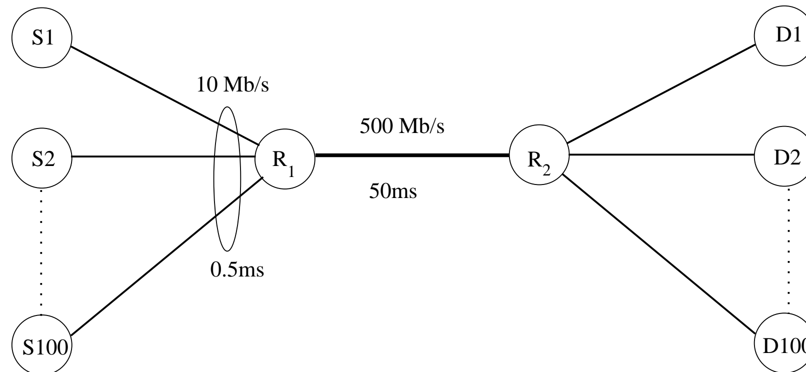
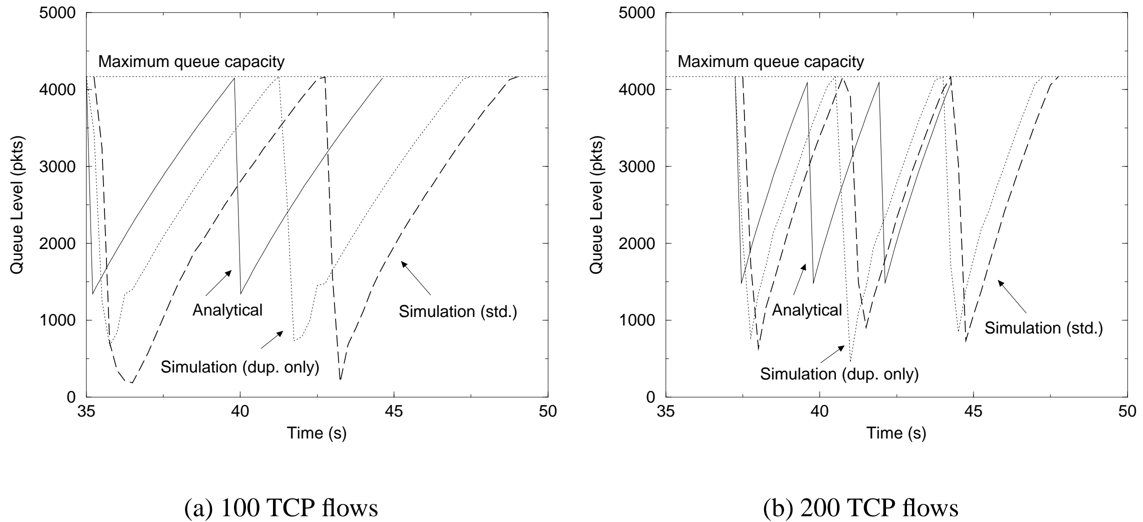


Figure 2.2: Single Bottleneck Link Network Configuration

To verify the accuracy of our analysis, we compare the evolution of the bottleneck-link queue occupancy using our analytical model against ns-2 simulations. The configuration used for the simulation is shown in Figure 2.2. Figure 2.3 shows the queue occupancy curves as predicted by the analytical model compared with actual levels obtained using two ns-2 simulation models. Figure 2.3(a) shows the queue occupancy curves for 100 TCP Reno sources and Figure 2.3(b) shows the queue occupancy curves for 200 TCP Reno sources. As we would intuitively expect, when we increase the number of sources, the buffer fills up faster, and thus we have a shorter time period for each cycle. One of our analytical model assumptions is that all packet losses are detected via duplicate ACKs and the TCP source enters *fast recovery* mode after retransmission of the packet instead of *slow start*. To try and simulate this behaviour, we changed the TCP Reno implementation in ns-2 to enter *fast recovery* mode even after a timeout. The curve labeled "Simulation (dup. only)" represents a simulation run using such modified TCP Reno sources. The curve labeled "Simulation (std.)" represents the queue occupancy curve obtained from a simulation using regular TCP Reno sources.

From Figure 2.3 we conclude that although our analytical model does not exactly match the simulation results, it captures the first-order behaviour of the system and its relative simplicity provides useful insight into the behaviour of real TCP implementations. In particular, it clarifies what factors would improve performance by reducing the drops in queue occupancy after packet drops and reducing total queue *cycle* times. One of the reasons for the discrepancy between our model and the simulation results is due to timeouts. Although we modified the TCP Reno sources to enter *fast recovery* after a timeout, the time



(a) 100 TCP flows

(b) 200 TCP flows

Figure 2.3: Comparison of analytical queueing behaviour with simulation results

taken by the source to react to (detect) a packet loss is much longer than one RTT, since a TCP timeout is usually two or three times the actual RTT value. During the RTT periods that sources are waiting for a timeout, they do not transmit **any** packets. However, in our analytical model, we assume that every source will react to a packet loss within one RTT and continue to send packets. Thus, the curves obtained via simulation show a slightly higher drop in queue occupancy after packet drops. Also, given that the queues level drops further, the time taken for the queue to fill up again is longer thus causing the analytical curves to complete each cycle in less time than the simulation.

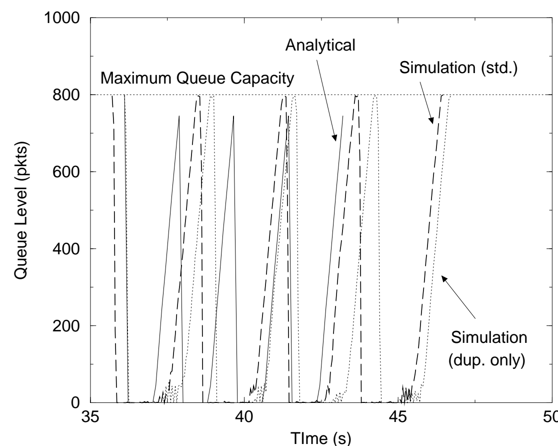


Figure 2.4: Comparison of analytical queueing behaviour with simulation results when queue underflows

Table 2.2: Comparing queue drop and cycle time obtained using the analytical model vs. simulation data

Buffer Size	Analysis		Simulation	
	Queue Drop (pkts)	Cycle time (s)	Queue Drop (pkts)	Cycle time (s)
4167	2822	4.89	3400	6.25
8333	4280	10.94	5250	13
20833	8655	43.54	12900	60

Figure 2.4 shows the queue occupancy curves for 100 TCP Reno sources with a maximum buffer size of 800 packets. This graph illustrates that our analytical model is also able to handle scenarios when the buffer size is small and the queue underflows.

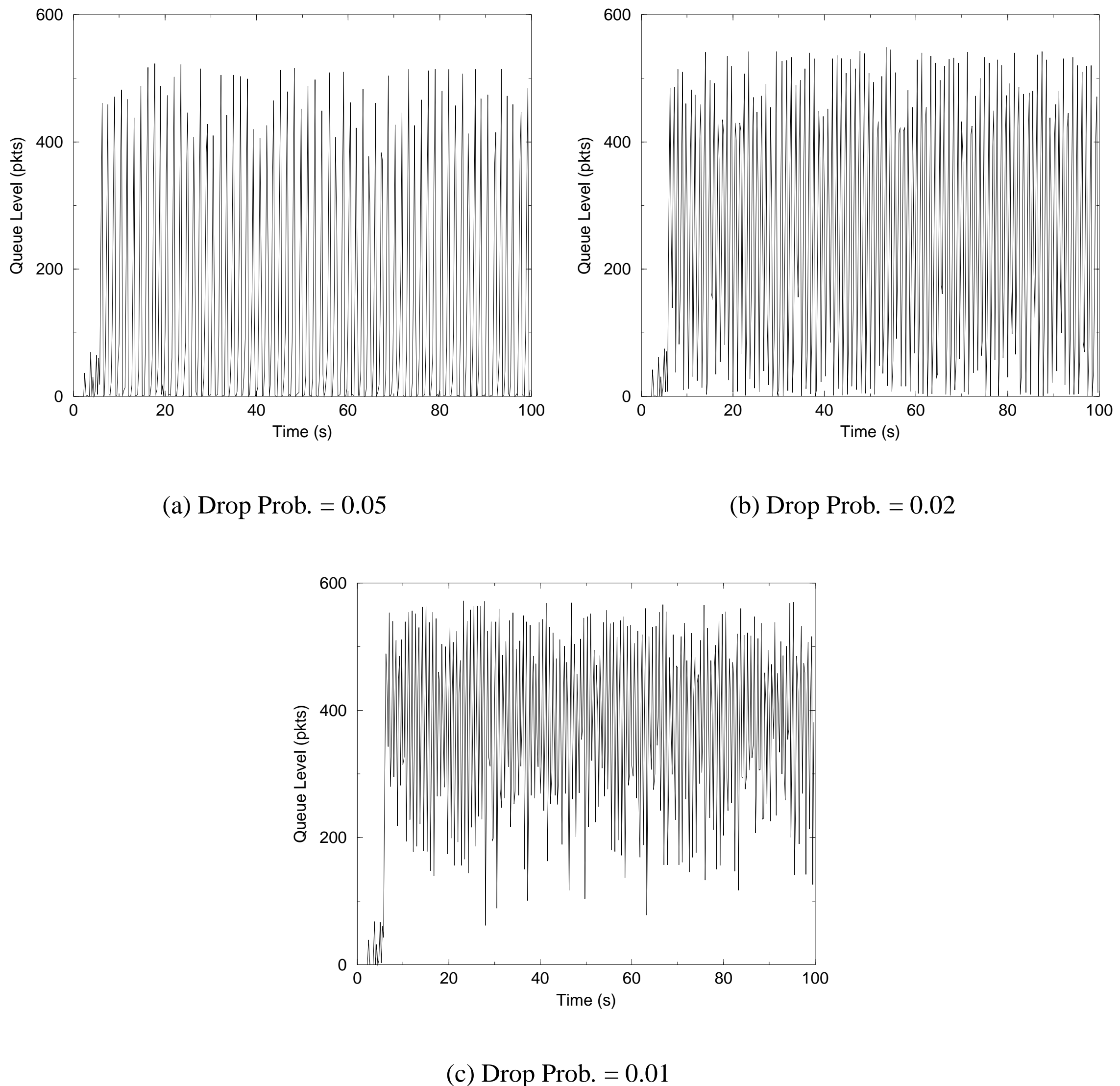
Finally, Table 2.2 compares the queue drop and total cycle (T_{total}) values obtained using the analytical model and the simulation (dup. only) for different buffer sizes. For our analytical model numbers, we chose α to be 0.7. When TCP sources are synchronized, the probability that a source will experience a packet drop can be approximated by $(1 - e^{-1}) = 0.63$ even for fairly small values of W^*. Although, in practice, all TCP sources do not remain synchronized every RTT period, we found that this value of α provides a good approximation in our analysis.

Overall, we emphasize that the exercise of developing the rudimentary analytical model was to get a handle on what parameters predominantly affect length of the period of buffer oscillation. The simulation comparison was done only to judge the rough accuracy of our model and observe that the slopes obtained via the analytical model matched the slopes obtained via simulation. From our analytical model, we observe that W^* (the fair-share window size of each TCP flow when the buffer is full) is a predominant factor determining the length of the period of buffer oscillation. Thus, if we have very small TCP flows (fair-share window size less than 5), the oscillation periods will be smaller.

2.1.2 RED

To address the issue of TCP flow synchronization due to Tail Drop packet discard policies, researchers have developed alternative queueing algorithms which try to keep average queue sizes low, while still providing high throughput and link utilization. The most popular of these is *Random Early Discard* or RED [30]. RED maintains an exponentially-weighted moving average of the queue length which is used to detect congestion. When the average crosses a minimum threshold (min_{th}), packets are randomly dropped or marked

with an explicit congestion notification (ECN) bit. When the queue length exceeds the maximum threshold (max_{th}), all packets are dropped or marked. RED includes several parameters which must be carefully selected to get good performance. To make it operate robustly under widely varying conditions, one must either dynamically adjust the parameters or operate using relatively large buffer sizes [23, 64].



(a) Drop Prob. = 0.05

(b) Drop Prob. = 0.02

(c) Drop Prob. = 0.01

Figure 2.5: Buffer Occupancy Time History for RED with changing Drop Probability

Figure 2.5 further illustrates this issue by showing the effect of different packet drop probabilities used for RED on buffer occupancy over a single bottleneck link. The minimum and maximum thresholds were fixed at 400 and 800 packets respectively for

the simulation runs. Table 2.3 shows how the parameter settings for RED affect average throughputs achieved by TCP flows. As expected from the buffer occupancy graphs, as we reduce the dropping probability, the average throughput increases.

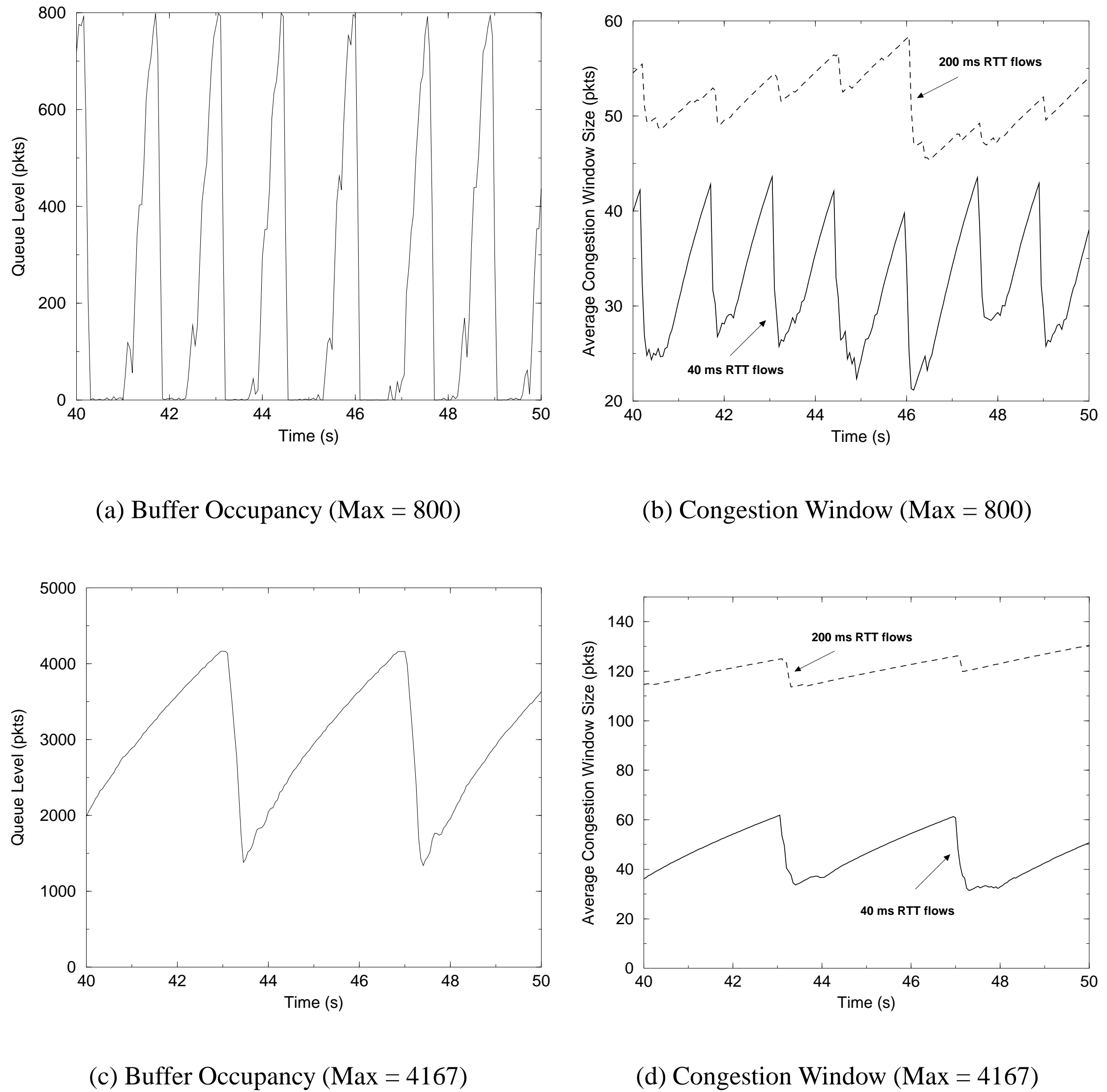
Table 2.3: Throughput variations when changing RED's dropping probability

RED	Drop Probability		
	0.05	0.02	0.01
Mean Throughput (Mb/s)	4.67	4.82	4.84

2.2 TCP Fairness and Round-Trip Times (RTT)

Given TCP's window-based flow and congestion control algorithms, flows with a shorter RTT will be able to increase their congestion windows faster and thus are able to grab an unfair share of the bottleneck bandwidth. Also, flows with a longer RTT need a larger window size to achieve comparable throughput to flows with a short RTT. This is due to the fact that a TCP connection can only have a window's worth of bytes in transit at any time. Thus, the maximum throughput a flow can achieve is its window size divided by the RTT.

Figure 2.6 illustrates the dynamic behaviour of the congestion window sizes of two sets of TCP flows using the Tail Discard packet drop policy. The short RTT flows have a RTT of 40 ms (in the absence of queueing delays), while the long RTT flows have a RTT of 200 ms. The buffer size of 4167 packets is equal to the bandwidth-delay product for a flow with an RTT of 100 ms. To achieve ideal fairness, the long RTT flows should have a congestion window size of about 5 times that of the short RTT flows. With the Tail Discard packet drop policy, all incoming packets are dropped at random when the buffer overflows, thus affecting both short and long RTT flows equally. This effect is somewhat offset for long RTT flows by the fact that since they take longer to recover after packet drops, during subsequent overflow periods, the short RTT flows will have a greater proportion of packets and hence a higher probability of getting dropped. From Figure 2.6(a), we observe that for small buffers, the congestion window sizes for the 200 ms RTT flows are only about one and a half times that of the 40 ms RTT flows. Thus, the 40 ms RTT flows can achieve more than *2 times* the throughput compared to the 200 ms RTT flows. For the larger buffer size, Figure 2.6(c), Tail discard does allow the 200 ms RTT flows to reach a larger window size,



(a) Buffer Occupancy (Max = 800)

(b) Congestion Window (Max = 800)

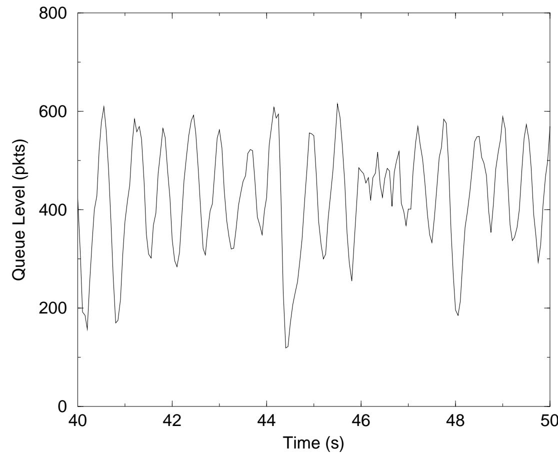
(c) Buffer Occupancy (Max = 4167)

(d) Congestion Window (Max = 4167)

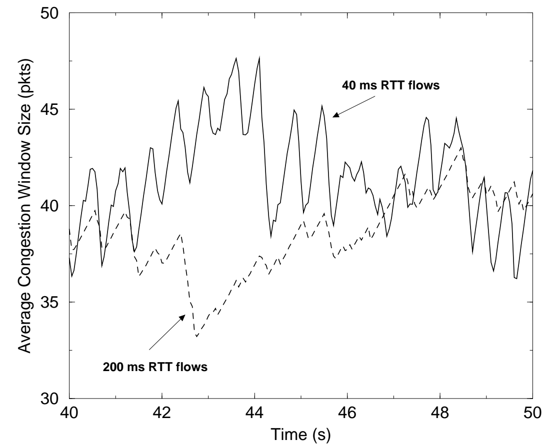
Figure 2.6: Average TCP Congestion Window Sizes for TCP sources with different RTT using Tail Discard

but it is only thrice that of the 40 ms RTT flows. Thus, the 40 ms RTT flows will still get higher throughput compared to the 200 ms RTT flows.

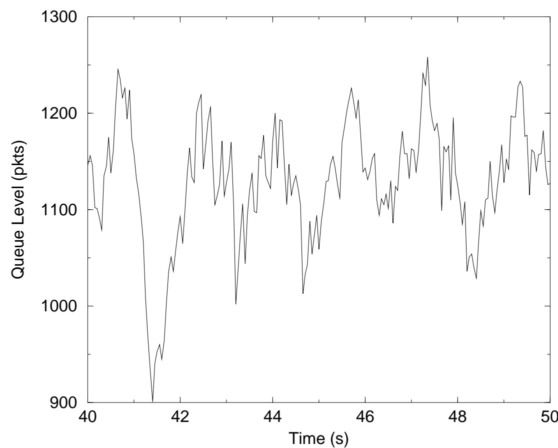
Figure 2.7 illustrates the dynamic behaviour of the congestion window sizes of two sets of TCP flows using RED. We observe that the congestion window sizes of both the 40 ms RTT and 200 ms RTT flows are approximately equal for small and large buffer sizes. Thus, 40 ms RTT flows can achieve about *5 times* the throughput of the 200 ms RTT flows. This shows that RED's discarding policy is unfair to higher RTT flows regardless of buffer size.



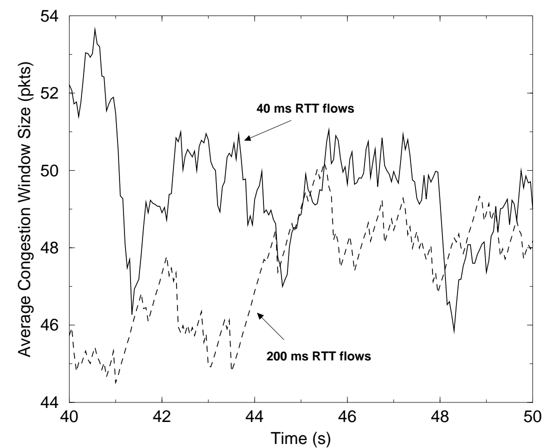
(a) Buffer Occupancy (Max = 800)



(b) Congestion Window (Max = 800)



(c) Buffer Occupancy (Max = 4167)



(d) Congestion Window (Max = 4167)

Figure 2.7: Average TCP Congestion Window Sizes for TCP sources with different RTT using RED

Finally, Table 2.4 summarizes the throughputs achieved by the 40 ms RTT and 200 ms RTT flows under both Tail Discard and RED. As we observed with the congestion window size graphs, regardless of buffer size, RED discriminates against the higher RTT flows whereas Tail discard is less unfair to the longer RTT flows when buffer sizes are sufficiently large.

Table 2.4: Average Throughputs for 40 ms RTT and 200 ms RTT flows under Tail Discard and RED

Buffer Size (pkts)	Average Throughput Mb/s			
	Tail Drop		RED	
	40 ms RTT	200 ms RTT	40 ms RTT	200 ms RTT
800	6.31	2.87	7.74	2.07
4167	5.10	4.70	7.44	2.41

Chapter 3

Per-Flow Queueing

The results presented in this Chapter show that both RED and Blue are deficient in meeting the objectives of a packet scheduler as described in Chapter 1. Both perform fairly poorly when buffer space is limited to a small fraction of the round-trip delay. Although Blue is less sensitive to parameter choices than RED, it still exhibits significant parameter sensitivity. Both RED and Blue exhibit a fairly high *variance* among individual TCP flow goodputs even over a single-bottleneck link.

The TCP analysis presented in Chapter 2 gives us valuable insight into the bottleneck queue behaviour. We notice that if more sources are synchronized (αN is high), the drop in queue level ($B - Q_1$) will be higher and the cycle time (T_{total}) will increase. In the algorithms presented below, we use multiple queues to explicitly control the number of flows that suffer a packet loss. This significantly reduces the synchronization among flows and allows us to get very good performance over buffers which are a fraction of the bandwidth-delay product.

3.1 Algorithms

Given the problems with existing congestion buffer management algorithms, we decided to evaluate a fair queueing discipline for managing TCP flows. We started by using Deficit Round Robin (DRR) [59]. DRR is an approximate fair-queueing algorithm that requires only $O(1)$ work to process a packet and thus it is simple enough to be implemented in hardware. Also, since there are no parameters to set or fine tune, it makes it usable across varying traffic patterns. We evaluated three different packet-discard policies. It is worth noting that although we have chosen DRR as the packet scheduler, our discard policies are not specific to DRR and can be used with other packet scheduling algorithms.

1. **DRR with Longest Queue Drop**
   Our first policy combined DRR with packet-discard from the longest active queue. For the rest of the thesis, we refer to this policy as plain DRR or DRR, since this packet-discard policy is part of the original DRR algorithm [59] and was first proposed by McKenney in [50]. Through our simulation study, we found that plain DRR was not very effective in utilizing link bandwidth or providing fair sharing among competing TCP flows over a single-bottleneck link. DRR did perform significantly better than RED and Blue when there were TCP flows with different RTTs or the flows were sent through a multi-bottleneck link topology. However its performance was roughly comparable to RED over a single-bottleneck link using large buffers, and worse for small buffer sizes. Thus, we investigated two different enhancements to the packet-discard policy which are outlined below.

2. **Throughput DRR (TDRR)**
   In this algorithm, we maintain a throughput value for each DRR queue. The throughput parameter is maintained as an exponentially weighted average and is used in choosing the drop queue. The exponential weight used in our simulations is 0.03125. We found that TDRR is not very sensitive to the weight parameter and performed equally well for weights ranging from 0.5 to $1.0e-6$. The discard policy for a new packet arrival when the link buffer is full, is to choose the queue with the highest throughput (amongst the currently active DRR queues) to drop a packet. Intuitively, this algorithm should penalize higher throughput TCP flows more and thus achieve better fairness and our simulation results do confirm this. The main drawback of this policy is that we need to store and update an extra parameter for each DRR queue. A second minor drawback is the time averaging parameter, which might require tuning under some circumstances (although our experience to date shows no significant sensitivity to this parameter).

3. **Queue State DRR (QSDRR)**
   Since TDRR has an overhead associated with computing and storing a weighted throughput value for each DRR queue, we investigate another packet-discard policy which adds some hysteresis to plain DRR's longest queue drop policy. The idea is that once we drop a packet from one queue, we keep dropping from the same queue when faced with congestion until that queue is the smallest amongst all active queues. This policy reduces the number of flows that are affected when a link becomes congested. This reduces the TCP synchronization effect and reduces the magnitude of

the resulting queue length variations. A detailed description of this algorithm is presented in Figure 3.1.

```
Let Q be a state variable which is
  undefined initially.
When a packet arrives and there is no
  memory space left:

if Q is not defined
  Let Q be the longest queue in the
    system;
  Discard one or more packets from
    the front of Q to make room
    for the new packet;
else // Q is defined
  if Q is shorter then all
    other non-empty queues
    Let Q be the longest queue in the
      system now;
    Discard one or more packets
      from the front of Q to make
      room for the new packet;
  else
    Discard one or more packets
      from the front of Q to make
      room for the new packet;
```

Figure 3.1: Algorithm for QSDRR

3.2 Evaluation

In order to evaluate the performance of DRR, TDRR and QSDRR, we ran a number of experiments using ns-2. We compared the performance over a varied set of network configurations and traffic mixes which are described below. In all our experiments, we used TCP sources with 1500 byte packets and the data collected is over a 100 second simulation interval. We ran experiments using TCP Reno and TCP Tahoe and obtained similar results for both; hence, we only show the results using TCP Reno sources.

For each of the configurations, we varied the bottleneck queue size from a 100 packets to 20,000 packets. 20,000 packets corresponds to a half-second bandwidth-delay

Table 3.1: RED parameters

RED		
max_p	Max. drop probability	0.01
w_q	Queue weight	0.001
min_{th}	Min. threshold	20% of buffer
max_{th}	Max. threshold	Buffer size

Table 3.2: Blue parameters

Blue		
$d1$	Increment	0.0025
$d2$	Decrement	0.00025
$freeze_time$	Hold-time	0.1s

product buffer which is a common buffer size deployed in current commercial routers. We ran several simulations to determine values of max_p and w_q for RED that worked best for our simulation environment, to ensure a fair comparison against our multi-queue based algorithms. The RED parameters we used in our simulations are in Table 3.1. For Blue, we ran simulations over our different configurations to compare the four sets of parameters used by the authors in their paper while evaluating Blue [27]. The Blue parameters we used are in Table 3.2 and are the ones that gave the best performance.

We now present the evaluation of our multi-queue policies in comparison with Blue, RED and Tail-Drop. We compare the queue management policies using the average goodput of all TCP flows as a percentage of its fair-share as the metric. We also show the goodput distribution of all TCP sources over a single-bottleneck link and the variance in goodput. The *variance* in goodputs is a metric of the fairness of the algorithm; lower variance implies better fairness. For all our graphs, we concentrate on the goodputs obtained while varying the buffer size from 100 packets to 5000 packets. Note, for the multi-queue algorithms, the stated buffer size is shared over all the queues, while with the single queue algorithms, the stated buffer size is for that single queue. Since our bottleneck link speed is 500 Mb/s, this translates to a variation of buffer *time* from 2.4 ms to 120 ms. In all our simulations, we noticed that all the policies behaved in a similar fashion past the 5000 packet buffer size.

3.3 Continuous TCP Flows

For the first set of experiments, we use TCP sources that remain *on* (always have data to send) throughout the length of the simulation. We compare our algorithms against RED, Blue and Tail Discard using three different network configurations. The parameters and comparison results for each configuration are detailed below.

3.3.1 Single Bottleneck Link

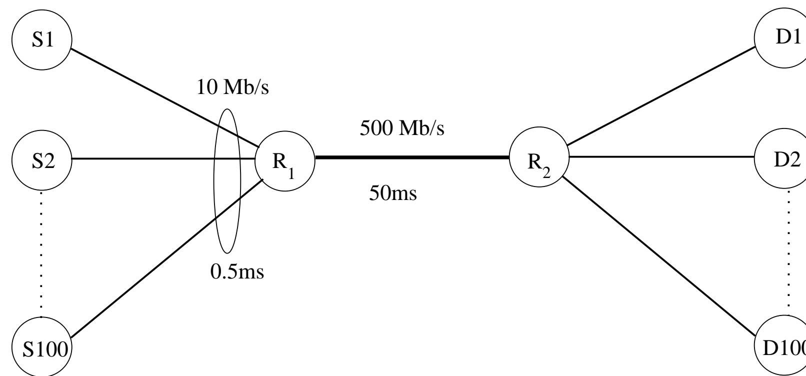


Figure 3.2: Single Bottleneck Link Network Configuration

The network configuration for this set of experiments is shown in Figure 3.2. The TCP sources, $\{S_1, S_2, ...S_{100}\}$, are each connected by 10 Mb/s links to the bottleneck link. Since the bottleneck link capacity is 500 Mb/s, if all TCP sources send at the maximum rate, the overload ratio is 2:1. The destinations, named $\{D_1, D_2, ...D_{100}\}$, are directly connected to the router R_2. All 100 TCP sources are started simultaneously to simulate a worst-case scenario whereby TCP sources are synchronized in the network. In each of the configurations, the delay shown is the one-way link delay. Thus, round-trip time (RTT) over a link is twice the link delay value. For this configuration, the fair-share bandwidth for each TCP flow is 5 Mb/s. With an RTT of 100 ms, this translates to each flow sending 500 Kb per RTT. Since the packet size (MSS) is 1500 bytes, the fair-share window size for each TCP flow when the queue is empty, $W_0 = \frac{500Kb}{1500bytes} = 42$.

Recalling our simple buffer analysis in Section 2.1.1, we have N = 100, R = 500 Mb/s, W_0 = 42 and T_0 = 100 ms for this configuration. The bandwidth-delay product is 50 Mb or 4167 packets. Thus, $W^* = W_0 + \frac{B}{N}$ = 84 packets, for a buffer size equal to the bandwidth-delay product of 4167 packets. Also, for this buffer size, $T(B)$ (RTT when the queue is full) = 200 ms.

Finally, we note that with $N = 100$, we expect 100 packets that need to be dropped during a discard interval when all TCP flows are synchronized. Also, when a flow experiences a packet discard, it will halve its window size. With $W_0 = 42$ packets, if 5 flows halve their window size (and thus halve their sending rate), the number of packets sent reduces by more than 100. Thus, we can effectively reduce the incoming rate by enough to prevent buffer overflow by affecting only 5-10 flows.

3.3.2 Results

The first set of graphs, shown in Figure 3.3, compares the distribution of goodputs for all 100 TCP Reno flows over the simulation run. For this experiment, the single-bottleneck link configuration is used and the buffer size is set to 200 packets. The closer the goodputs are to each other, the lower the variance, which implies better fairness. We notice that under TDRR and QSDRR (Figures 3.3(b), 3.3(c)), all TCP flows had goodputs very close to the mean and the mean goodput is very near the fair-share threshold. We notice that the average goodput under DRR 3.3(a) is not as good as TDRR and QSDRR and it is even slightly lower than RED, so simple DRR is not sufficient to prevent under-utilization of the link. In the case of Blue (Figure 3.3(d)), although the goodputs of different TCP flows are close to each other, the mean goodput achieved is far below the fair-share threshold which leads to under-utilization of the link. The mean goodput achieved using RED (Figure 3.3(e)) is close to the fair-share threshold, but the variance is high. Also, a significant number of sources are able to get more than their fair-share of the bandwidth. As expected, Tail Drop (Figure 3.3(f)), performs most poorly, with the highest variance in goodputs and a very low average goodput.

Figure 3.4(a) shows the ratio of the goodput standard deviation of the TCP Reno flows to the fair share bandwidth for all algorithms while varying the buffer size. Even at higher buffer sizes, the goodput standard deviation under DRR and QSDRR is very small and the ratio to the fair share bandwidth is less than 0.025. TDRR exhibits a higher goodput standard deviation, but it is still significantly below Blue, RED and Tail Drop. RED exhibits about 10 times the variance compared to QSDRR and DRR, while Blue exhibits about 5 times the variance. Overall, we observe that the goodput standard deviation is between 2%-4% of the fair share bandwidth for the multi-queue policies compared to 6% for Blue, 10% for RED and 12% for TailDrop.

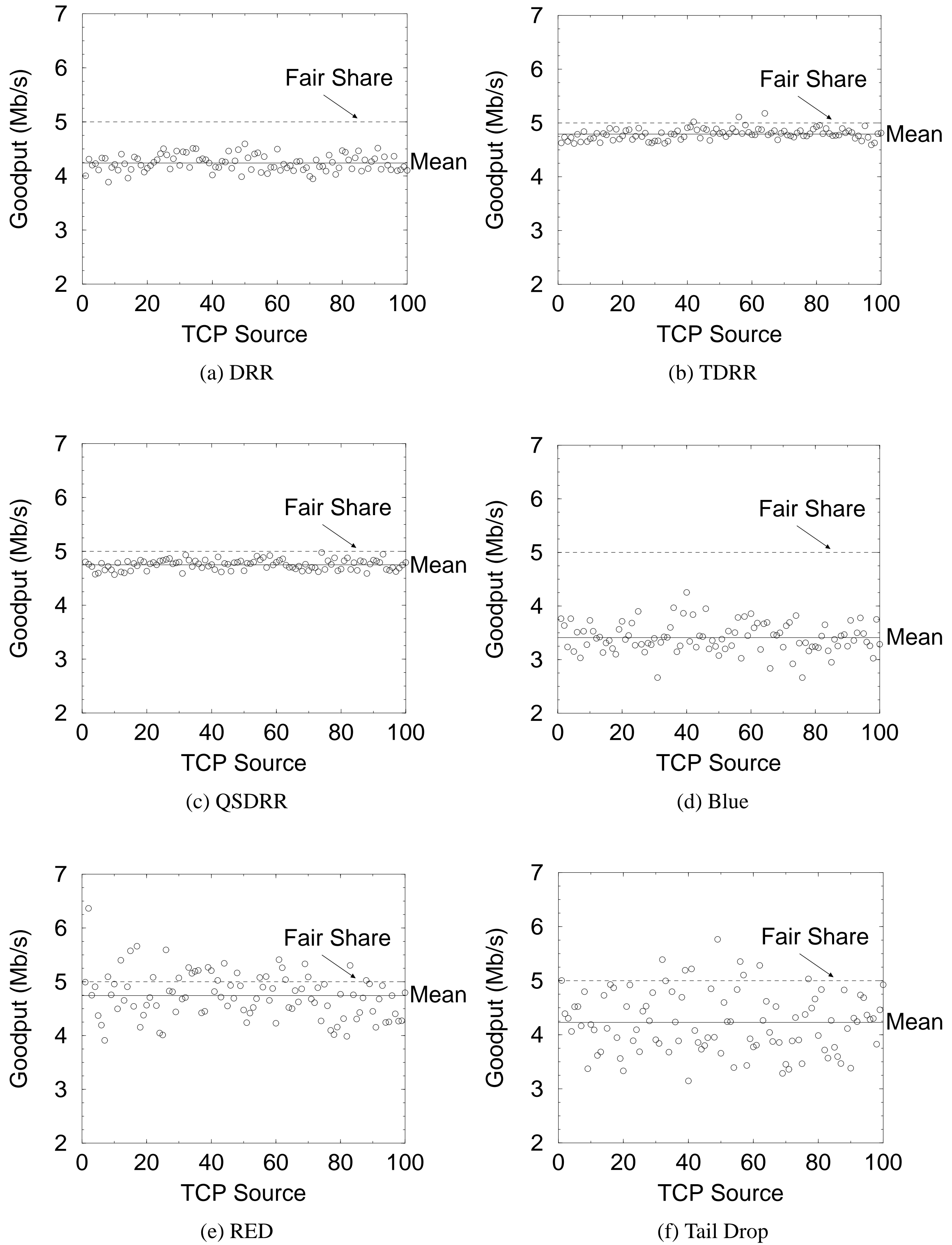
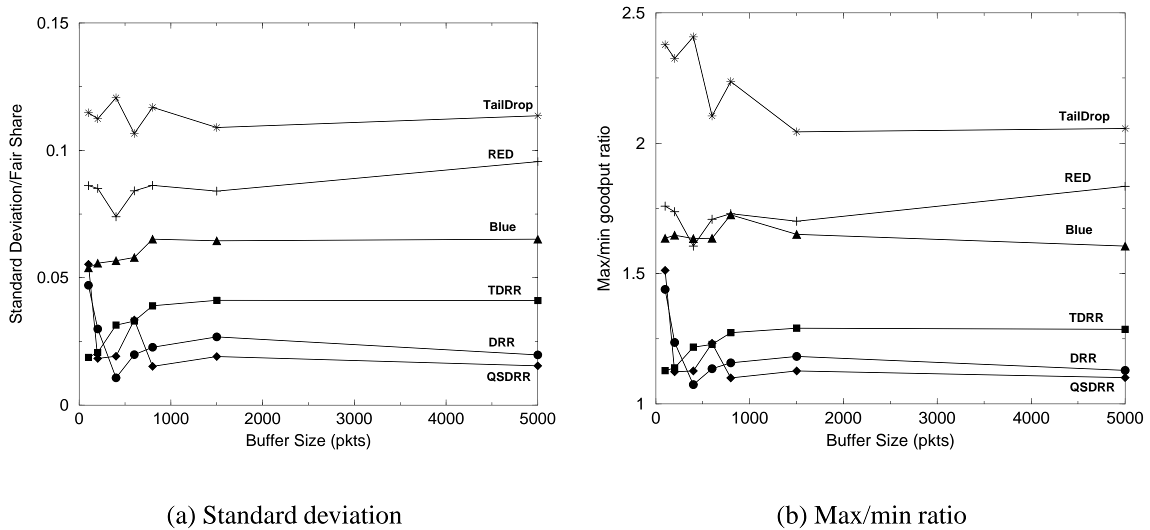


Figure 3.3: TCP Reno Goodput distribution over single-bottleneck link with 200 packet buffer



(a) Standard deviation

(b) Max/min ratio

Figure 3.4: Standard deviation relative to fair-share for TCP Reno flows over a single-bottleneck link

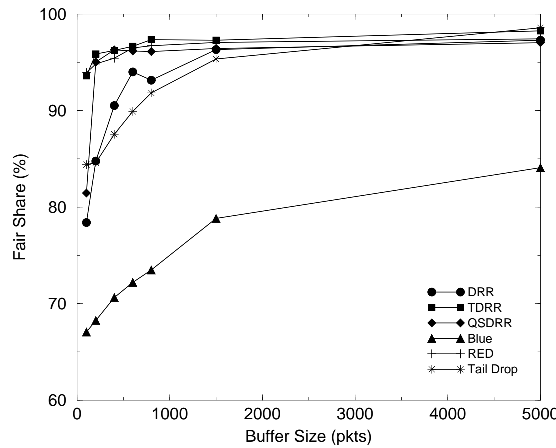


Figure 3.5: Fair share performance over a single bottleneck link

A link with thousands of flows will have some flows that have a goodput which is greater than three times the standard deviation over the average and some flows that have a goodput greater than three times the standard deviation below the average. Thus, although the differences in standard deviation in Figure 3.4(a) appear small, they could result in a significantly higher difference in the ratios of the maximum goodput over minimum goodput flows. Figure 3.4(b) shows the ratios of maximum over minimum goodputs for all the algorithms. To generate this graph, we add three times the standard deviation to the average goodput to get the maximum goodput and subtract three times the standard deviation to the

average goodput to get the minimum goodput. From Figure 3.4(b), we observe that QS-DRR and DRR have a max/min ratio very close to 1, while Blue and RED have a max/min ratio close to 1.8 which is almost twice that for QSDRR. Also, Tail Drop has the worst max/min ratio that remains above 2 even for bandwidth-delay sized buffers. Thus, even for a single-bottleneck link, we observe that the multi-queue policies offer much better fairness to a set of TCP flows.

Figure 3.5 illustrates the average fair-share bandwidth percentage received by the TCP Reno flows using different buffer sizes. We observe that over a single bottleneck link, QSDRR, TDRR and RED deliver comparable average fair-share goodput. DRR performs slightly worse and is comparable to Tail Drop. Although QSDRR and TDRR are comparable to RED for the *average* fair-share goodput, both QSDRR and TDRR have a significantly lower standard deviation between goodputs of different flows compared to RED. We also observe that both QSDRR and TDRR are able to perform well for buffer sizes that are 5% of the bandwidth-delay product. This means that the TCP flows can achieve a high throughput using QSDRR and TDRR at **half** the end-to-end delay. It is interesting to note that even at a large buffer size of 5000 packets, all policies significantly outperform Blue, including Tail Drop. For Blue with very small buffers, the buffer will overflow very frequently. This causes the drop probability to increase very rapidly leading to unnecessary extra drops during the congestion avoidance phase, which further reduces the goodputs achievable by the flows. Also, the parameters used for Blue are the ones recommended by the authors of Blue. Since these parameters are sensitive to traffic mixes and network configurations, they may not be ideal for our experiments, thus further contributing to the poor performance under Blue.

Finally, Figure 3.6 illustrates the fact that the TDRR algorithm is insensitive to the value of the exponential weight parameter, which is set to 0.03125 for our experiments. We varied the exponential weight from 0.0078125 ($\frac{1}{4}$ times 0.03125) to 0.125 (4 times 0.03125). From Figure 3.6, we observe that at the lowest buffer size of 100 packets, there is a small difference in the fair-share performance for the three different weights, but for all buffer sizes above that, the fair-share performance of TDRR is nearly identical for the three different weights.

3.3.3 Multiple Round-Trip Time Configuration

The network configuration for this set of experiments is shown in Figure 3.7. This configuration is used to evaluate the performance of the different queue management policies

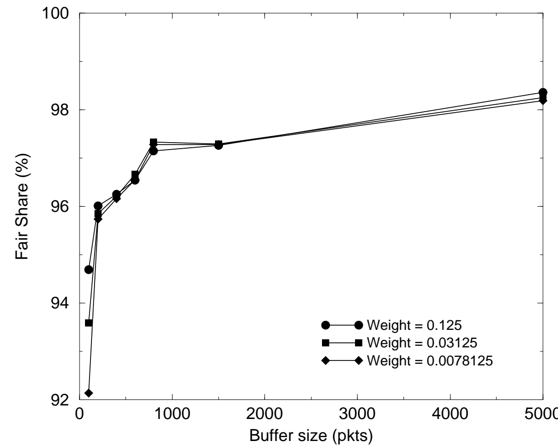


Figure 3.6: Fair share performance of TDRR over a single bottleneck link with varying exponential weights

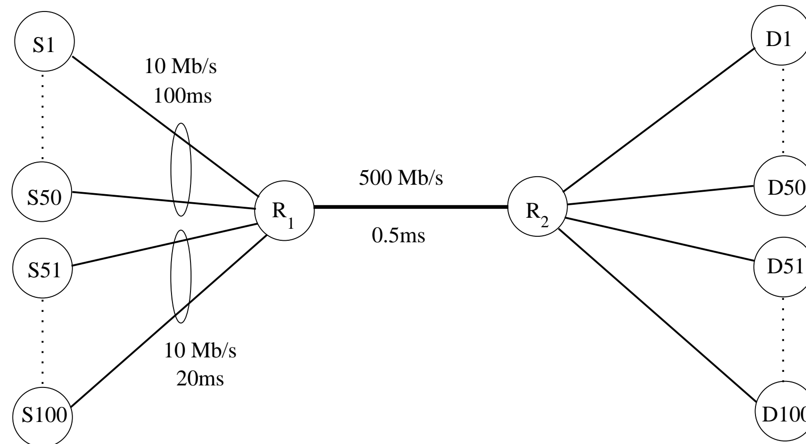
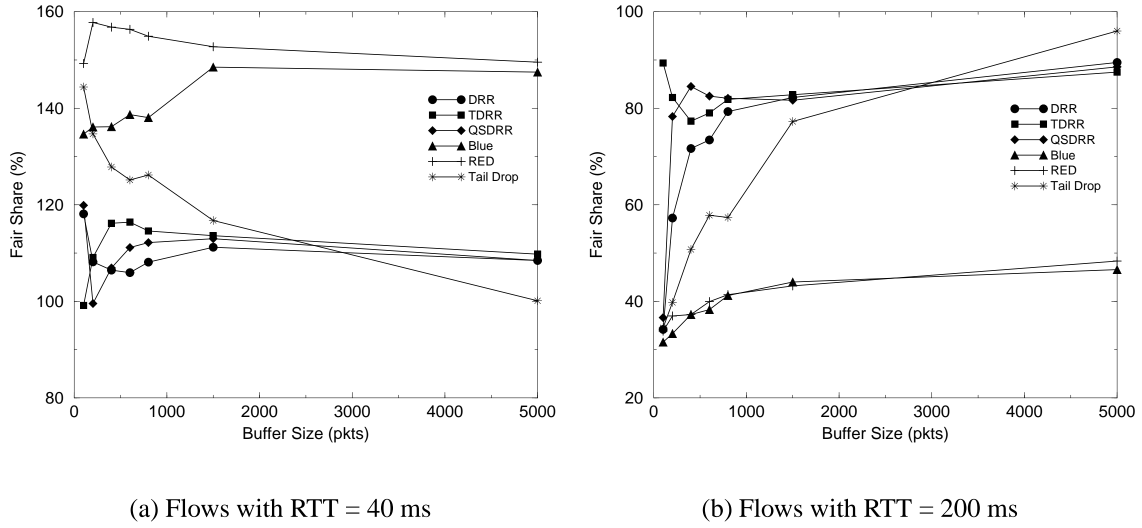


Figure 3.7: Multiple Round-Trip Time Network Configuration

given two sets of TCP flows with widely varying round-trip times over the same bottleneck link. The source connection setup is similar to the single-bottleneck configuration, except for the access link delays for each source. For 50 sources, the link delay is set to 20 ms, while it is set to 100 ms for the other 50 sources.

3.3.4 Results

For this configuration, we use 100 TCP Reno flows over a single bottleneck link. 50 flows have a 40 ms RTT and 50 flows have a 200 ms RTT. Figure 3.8 shows the average fair-share goodput received by each set. As shown in Figure 3.8(a), both RED and Blue allow the 40 ms RTT flows to use almost 50% more bandwidth than their fair share. Tail Drop



(a) Flows with RTT = 40 ms

(b) Flows with RTT = 200 ms

Figure 3.8: Fair share performance of different RTT flows over a single bottleneck link

also allows the 40 ms RTT flows to use more than their fair share of the bandwidth for buffer sizes smaller than 1000 packets. All the DRR-based policies exhibit much better performance allowing only 10% extra bandwidth to be used by the 40 ms RTT flows. Both RED and Blue discriminate against longer RTT flows, as we observe in Figure 3.8(b), the 200 ms RTT flows achieve only about 40% of their fair-share bandwidth whereas using the DRR-based policies, 200 ms RTT flows are able to achieve almost 90% of their fair-share.

At a very small buffer size of 100 packets, 200 ms RTT flows using DRR and QS-DRR get about 40% of their fair-share. However, at this buffer size, when all the flows are active, there is only one packet per flow that can be buffered. This causes the poor performance of DRR and QSDRR, since it becomes very difficult to single out flows that are using more bandwidth. Even with this limitation, when we move to 200 packets, both DRR and QSDRR significantly improve their performance and 200 ms RTT flows achieve about 80% of their fair-share bandwidth on the average. In QSDRR, longer queues are preferentially selected for discard. Since shorter RTT TCP flows will be able to increase their window sizes faster than longer RTT TCP flows, the shorter RTT TCP flows will send at a higher rate. Thus, the queues for the shorter RTT flows will build up faster and be longer than the queues for the longer RTT TCP flows, since we use DRR as our packet scheduler which is a fair queueing scheduler. This will lead to QSDRR choosing the shorter RTT flow queues for discard when the buffer overflows. Only when the longer RTT flows have achieved a rate equal or higher than the shorter RTT flows, their queues will become long and be considered for packet discard. Also, since TDRR maintains an exponentially

weighted throughput average for each flow, even at the smallest buffer size of 100 packets, it is able to deliver almost 90% of the fair-share bandwidth to the 200 ms RTT flows.

3.3.5 Multi-Hop Path Configuration

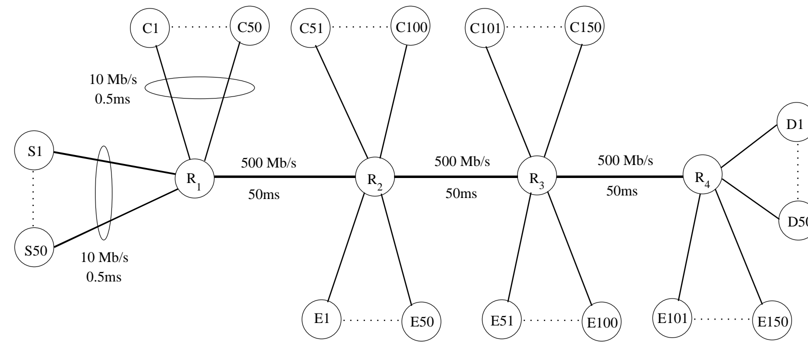
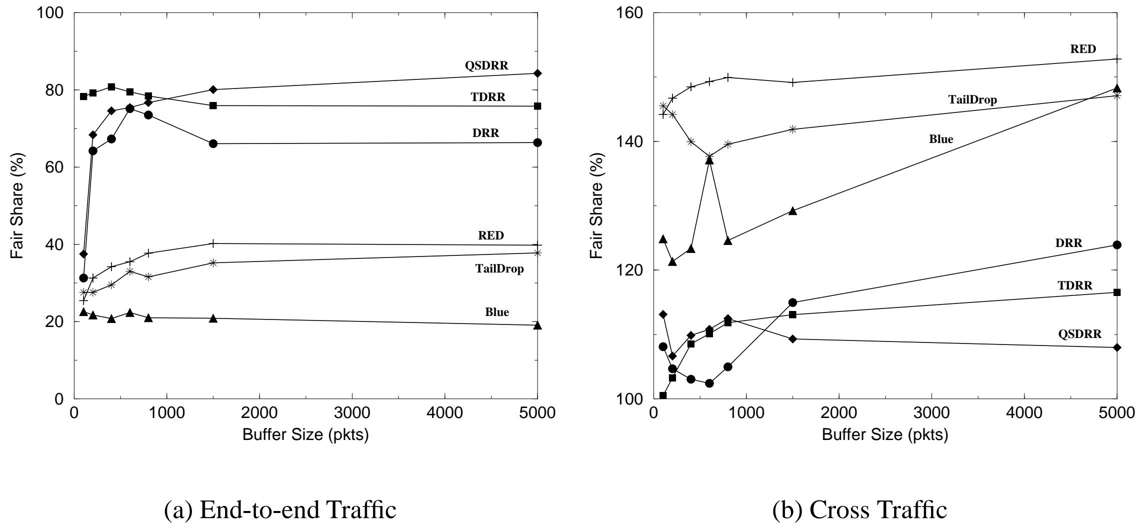


Figure 3.9: Multi-Hop Path Network Configuration

The network configuration for this set of experiments is shown in Figure 3.9. In this configuration, we have 50 TCP sources traversing three bottleneck links and terminating at R_3. In addition, on each link, there are 50 TCP sources acting as cross-traffic. We use this configuration to evaluate the performance of the different queue management policies for multi-hop TCP flows competing with shorter one-hop cross-traffic flows.

3.3.6 Results

In this configuration, 50 end-to-end TCP Reno flows go over three hops and have an overall round-trip time of 300 ms. The cross-traffic on each hop consists of 50 TCP Reno flows with a round-trip time of 100 ms (one hop). Figure 3.10 illustrates the average fair-share goodput received by each set of flows. For this configuration, TDRR and QSDRR provide almost *twice* the goodput of RED and Tail Drop and *four* times the goodput provided by Blue for end-to-end flows. As shown in Figure 3.10(a), end-to-end flows achieve nearly 80% of their fair-share under TDRR and QSDRR and 60% under DRR. Under RED and Tail Drop, they can achieve only 40% of their fair share. For even the smallest buffer size of a 100 packets, end-to-end TCP flows under TDRR are able to achieve 80% of their fair-share. Using QSDRR and DRR, for the smallest buffer size, their fair-share is the same as RED, but once the buffer size increases to 200 packets, their performance improves significantly and they allow the end-to-end flows to achieve close to 80% and 60% respectively.



(a) End-to-end Traffic

(b) Cross Traffic

Figure 3.10: Fair Share performance of end-to-end and cross traffic flows over a multi-hop path configuration

For this multi-hop configuration, the end-to-end flows face a probability of packet loss at each hop under RED and Blue. Due to congestion caused by the cross-traffic, RED and Blue will randomly drop packets at each hop. Although the cross-traffic flows will have a greater probability of being picked for a drop, the end-to-end flows also experience random dropping and thus achieve very poor goodput. For Blue, this is further exacerbated, since due to the high load from the cross-traffic flows, the discard probability remains high at each hop. This increases the probability of an end-to-end flow facing packet drops at each hop and thus further reducing the goodput.

Figure 3.10(b) shows the average goodput for the cross-traffic flows attached to router R_1. For DRR, TDRR and QSDRR, the cross-traffic takes up the slack in the link and consumes about 115-120% of its fair-share bandwidth. For both RED and Tail Drop, the link utilization is lower and although the end-to-end flows consume only about 40% of their fair-share, the cross-traffic flows consume 150% of their fair-share and thus leave about 5% unutilized. Cross-traffic flows under Blue consume about 120-140% of their fair-share, leaving 20-30% unutilized.

3.4 Connections with Multiple Short Transfers

The above set of results are for long-lived TCP flows. However, since a large percentage of the Internet traffic is currently web traffic, we investigate the performance of our algorithms

for short-lived TCP connections. To simulate web traffic, each TCP flow sends a burst of 256 packets (384 KB) and then is idle. We use a fixed burst size so that we can accurately compare the times taken to service each burst under the different algorithms. The idle time between bursts is exponentially distributed with a mean of 2 seconds. Also, each source now has a maximum link bandwidth of 100 Mb/s connection to the bottleneck link. We use the same network configurations as outlined in the previous section, but with a different number of TCP connections. The parameters and comparison results for each configuration are detailed below.

3.4.1 Single Bottleneck Link

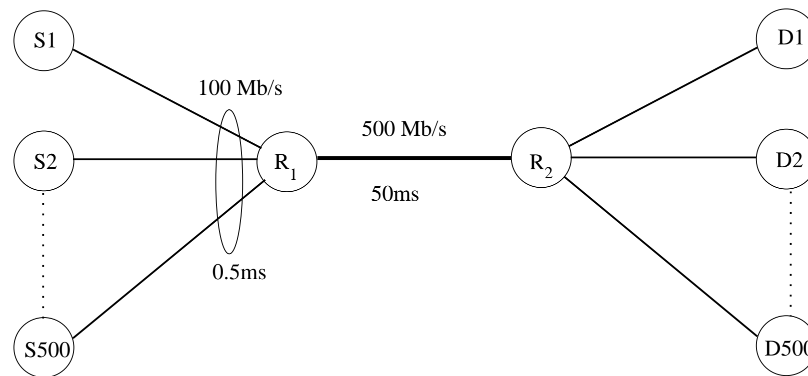
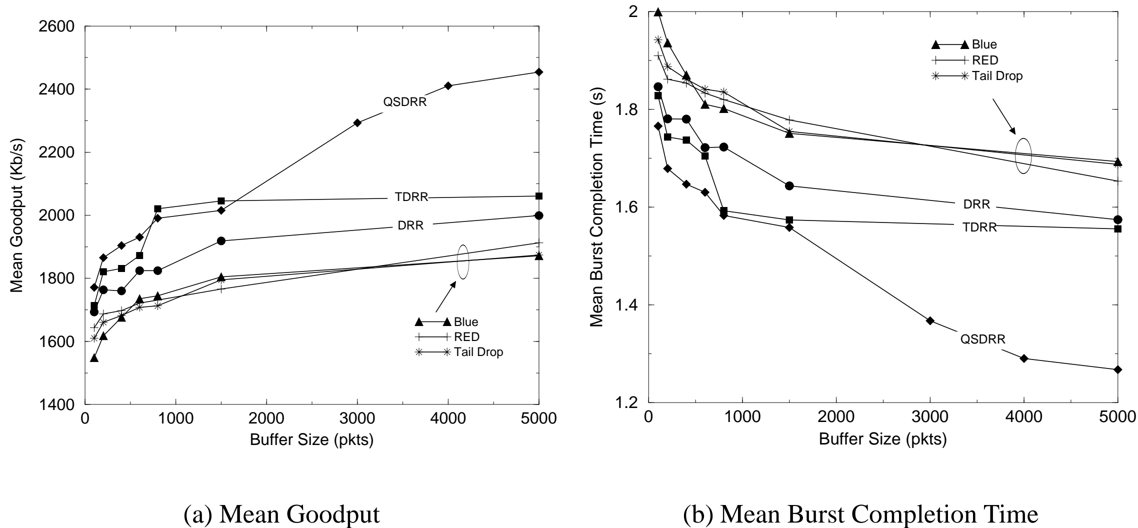


Figure 3.11: Single Bottleneck Link Network Configuration

The network configuration for this set of experiments is shown in Figure 3.11. The TCP sources, $\{S_1, S_2, ...S_{500}\}$, are each connected by 100 Mb/s links to the bottleneck link. The destinations, named $\{D_1, D_2, ...D_{500}\}$, are directly connected to the router R_2. All 500 TCP sources are started simultaneously to simulate a worst-case scenario whereby TCP sources are synchronized in the network.

3.4.2 Results

Figure 3.12(a) shows the mean goodput achieved by the TCP flows and Figure 3.12(b) shows the mean burst completion times for the flows over a **single bottleneck link** configuration. *Goodput* is the amount of actual data transmitted excluding retransmissions and duplicates. We notice that Blue, RED and Tail Drop have almost exactly the same performance in terms of mean goodput achieved and burst completion times for all buffer sizes, whereas the DRR schemes are uniformly better. For buffer sizes less than 2000 packets,



(a) Mean Goodput (b) Mean Burst Completion Time

Figure 3.12: Performance of short burst TCP flows over a single bottleneck link

TDRR and QSDRR exhibit about 10% better goodput performance over Blue, RED and Tail Drop. However, it is interesting to note that QSDRR is almost 30% better than the non-DRR policies at a buffer size of 5000 packets. The results are similar for the burst completion times.

3.4.3 Multiple Roundtrip-time Configuration

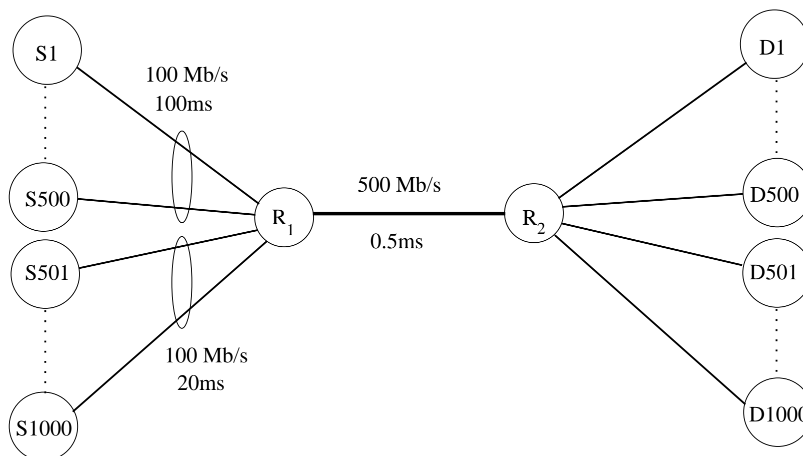
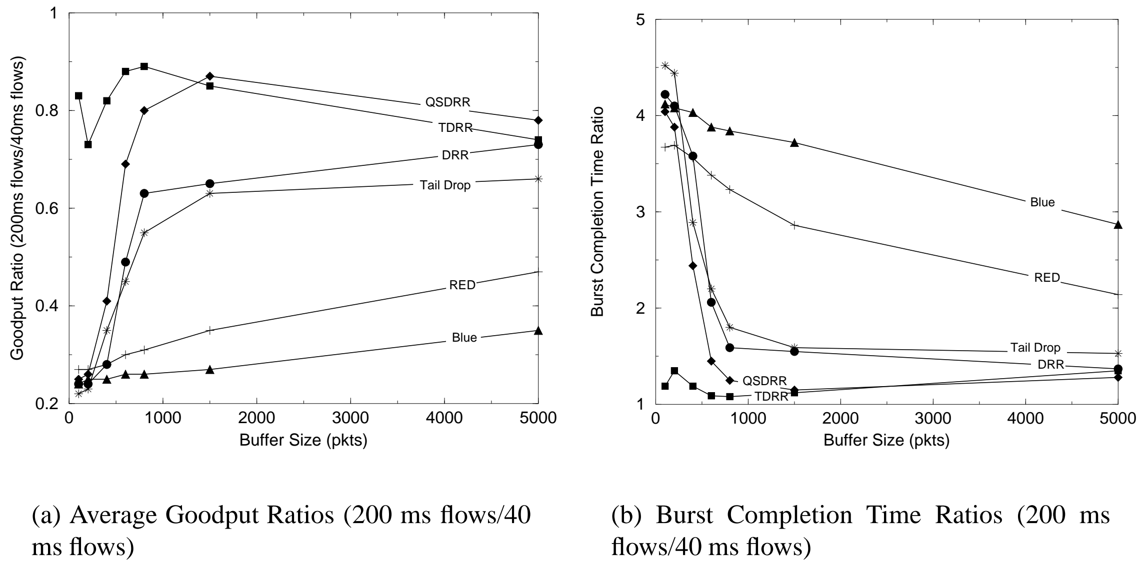


Figure 3.13: Multiple Roundtrip-time Network Configuration



(a) Average Goodput Ratios (200 ms flows/40 ms flows)

(b) Burst Completion Time Ratios (200 ms flows/40 ms flows)

Figure 3.14: Performance of short burst TCP flows over a multiple round-trip time configuration

The network configuration for this set of experiments is shown in Figure 3.13. This configuration is used to evaluate the performance of the different queue management policies given two sets of TCP flows with widely varying roundtrip-times over the same bottleneck link. The source connection setup is similar to the single-bottleneck configuration, except for the access link delays for each source and the total number of sources. We simulated 1000 TCP sources, 500 sources with link delay set to 20 ms, and 500 sources with link delay set to 100 ms.

3.4.4 Results

Figure 3.14(a) shows the ratios of the average goodputs obtained by 200 ms round-trip time flows over the average goodputs of the 40 ms round-trip time flows for the **multiple RTT** configuration. In this configuration, for buffer sizes less than a 1000 packets, QSDRR and TDRR outperform Blue and RED by more than 100%. The ratio of goodputs is used to illustrate the fairness of each algorithm. The closer the ratio is to one, the better the algorithm is in delivering fair-share to different round-trip time flows. In this case, even Tail Drop performs significantly better than Blue and RED, showing that for short-lived flows with different round-trip times, Blue and RED cannot deliver good fair-sharing of the bottleneck bandwidth. Figure 3.14(b) shows the ratios of burst completion times of the 200 ms round-trip time flows over the 40 ms round-trip time flows. In this case, QSDRR

and TDRR remain close to one (which is the ideal fairness), whereas Blue has the worst performance, with the 200 ms round-trip time flows taking almost *three times* the time to complete a burst compared to the 40 ms round-trip time flows, even for 5000 packet buffers.

In QSDRR, longer queues are preferentially selected for discard. Since shorter RTT TCP flows will be able to increase their window sizes faster than longer RTT TCP flows, the shorter RTT TCP flows will send at a higher rate. Thus, the queues for the shorter RTT flows will build up faster and be longer than the queues for the longer RTT TCP flows, since we use DRR as our packet scheduler which is a fair queueing scheduler. This will lead to QSDRR choosing the shorter RTT flow queues for discard when the buffer overflows. Only when the longer RTT flows have achieved a rate equal or higher than the shorter RTT flows, their queues will become long and be considered for packet discard.

3.4.5 Multi-Hop Path Configuration

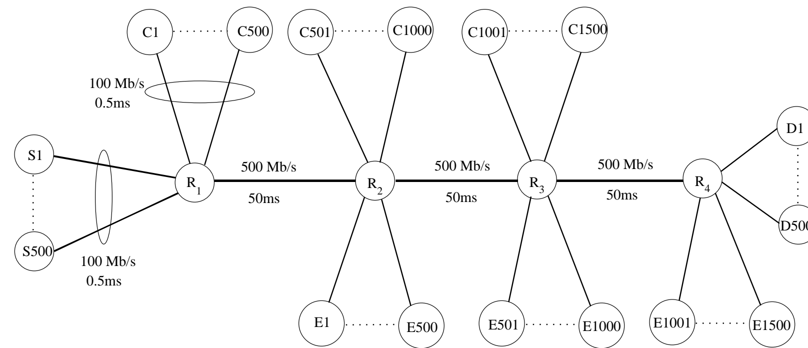
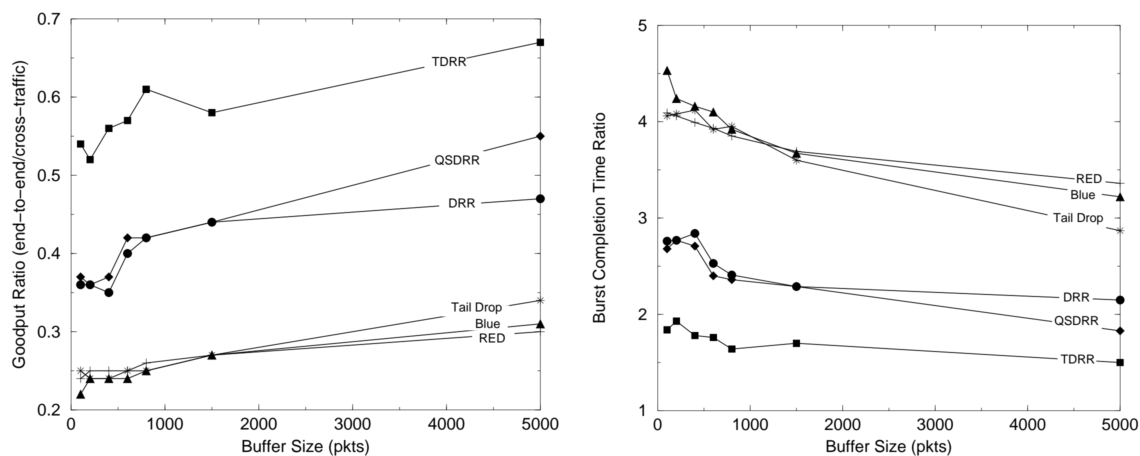


Figure 3.15: Multi-Hop Path Network Configuration

The network configuration for this set of experiments is shown in Figure 3.15. In this configuration, we have 500 TCP sources traversing three bottleneck links and terminating at R_3. In addition, on each link, there are 500 TCP sources acting as cross-traffic. We use this configuration to evaluate the performance of the different queue management policies for multi-hop TCP flows competing with shorter one-hop cross-traffic flows.

3.4.6 Results

Figure 3.16(a) shows the ratios of the goodputs achieved by the end-to-end flows over the cross-traffic flows for the **multi-hop path** configuration. In this configuration, we see that the non-DRR policies perform very poorly, allowing the end-to-end flows a mere 30% of the goodput achieved by the cross-traffic flows. On the other hand, QSDRR and even DRR



(a) Goodput Ratios (end-to-end flows/cross-traffic flows)

(b) Burst Completion Time Ratios (end-to-end flows/cross-traffic flows)

Figure 3.16: Performance of short burst TCP flows over a multi-hop path configuration

outperform the non-DRR schemes by 40% for buffer sizes less than 2000 packets. QSDRR is almost *2 times* better than the non-DRR policies for a buffer size of 5000 packets. Since TDRR maintains an exponentially weighted throughput average for a fairly long period, it outperforms all policies. For short-lived TCP flows, DRR and QSDRR cannot deliver the same fairness as TDRR since they do not maintain long-term state.

Figure 3.16(b) shows the ratios of burst completion times of the end-to-end flows over the cross-traffic flows. Only TDRR can achieve a ratio close to one, since it maintains long-term state. However, QSDRR and DRR perform reasonably well and beat the non-DRR policies by at least a factor of two. Even though the end-to-end traffic flows over three bottleneck links compared to just one bottleneck-link for the cross-traffic flows, QSDRR and DRR are able to achieve a burst completion time ratio of under two for a buffer size of 5000 packets. At the same buffer size, the non-DRR policies achieve fairly poor ratios ranging from 3.5 to 4.0.

Chapter 4

A Closer Look at Queue State DRR

4.1 Simple Buffer Analysis for TCP Flows using QSDRR

In this section, we present a simplified analysis of buffer occupancy of a congested bottleneck link using the QSDRR packet discard policy in the presence of a large number of TCP Reno flows. The assumptions we make for the analysis are:

- The analysis is a discrete time analysis considering one RTT as a unit.
- QSDRR policy is used in the bottleneck link buffer.
- All TCP flows are in congestion avoidance mode.
- All packet drops are indicated by duplicate ACKs.
- All TCP flows are synchronized initially (i.e. they have the same *cwnd* value) at the time of buffer overflow.
- In each RTT period, we assume a fluid model for the TCP data traffic. During each RTT period, all TCP flows transmit their entire *cwnd* and receive ACKs for all successfully received packets.

Table 4.1 describes the notation we use in the analysis. We present an approximate calculation of the drop in queue level after a buffer overflow and the time between consecutive buffer overflow events. We note that given our assumptions, the queue overflows when each source sends $W^* + 1$ packets (i.e. *cwnd* for each source is $W^* + 1$).

In the first RTT after overflow, let the expected number of sources experiencing a packet drop = X. For these X sources, the new value of *cwnd* = $\frac{W^*+1}{2}$, since these sources

Table 4.1: Definitions of terms used in analysis

N	Number of sources
R	Link rate in packets/second
B	Buffer size
T_0	Round-trip time when queue is empty
$T(b)$	Round-trip time when queue level is b
W_0	Fair-share window-size for each TCP flow when queue is empty
W^*	Fair-share window-size for each TCP flow when queue is full

experience a drop and will reduce their *cwnd* by half. For $(N - X)$ sources, the new value of *cwnd* = $W^* + 2$, since these sources do not experience any drops and thus will increase their *cwnd* by one each RTT. Now, in this RTT, packets drained from the buffer = $T(B) \cdot R$ and packets sent by sources = $X * N\left(\frac{W^*+1}{2}\right) + (1 - X)N(W^* + 2)$. Hence, queue level after the first RTT is:

$$\begin{aligned} Q_1 &= B - T(B)R + X\left(\frac{W^*+1}{2}\right) + (N - X)(W^* + 2) \\ &= B - \left[\frac{X}{2}(W^* + 3) - 2N\right] \end{aligned} \tag{4.1}$$

To compute X, which is the number of flows experiencing a drop, we first make the assumption that since all the sources are synchronized (sending at the same rate), their individual queues are the same size. In QSDRR, the drop queue picked is used to discard packets until it becomes the shortest queue. Thus, in the scenario where all queues start out having equal lengths, the first drop queue picked discards one packet before it becomes the shortest queue. The second drop queue picked discards two packets before it becomes the shortest queue. The third drop queue picked discards three packets before it becomes the shortest queue and so on. Following this line of reasoning, the number of queues experiencing packet discards, X, required to discard a total of N packets is given by:

$$\frac{X(X+1)}{2} \geq N \tag{4.2}$$

Solving this equation for X, we get

$$X \geq \frac{\sqrt{8N+1} - 1}{2} \tag{4.3}$$

Since X must be an integer, we round up and get

$$X = \left\lceil \frac{\sqrt{8N+1}-1}{2} \right\rceil \tag{4.4}$$

When all flows are synchronized (or sending at the same rate), each flow will have $\frac{B}{N}$ packets queued on the average when the buffer is full. Equation 4.2 is a valid formula for X (the number of queues experiencing discard), as long as $X \leq \frac{B}{N}$. If $X > \frac{B}{N}$, each additional queue selected for discard will only be able to discard $\frac{B}{N}$ packets. In this case, the total number of drop queues, X is given by:

$$X = X_1 + X_2 \tag{4.5}$$

$$X_1 = \frac{B}{N} \tag{4.6}$$

$$X_2 = \frac{\left[N - \frac{X_1(X_1+1)}{2}\right]}{\frac{B}{N}}$$

$$= \frac{N}{X_1} - \frac{X_1+1}{2} \tag{4.7}$$

$$= \frac{N^2}{B} - \frac{B+N}{2N^2} \tag{4.8}$$

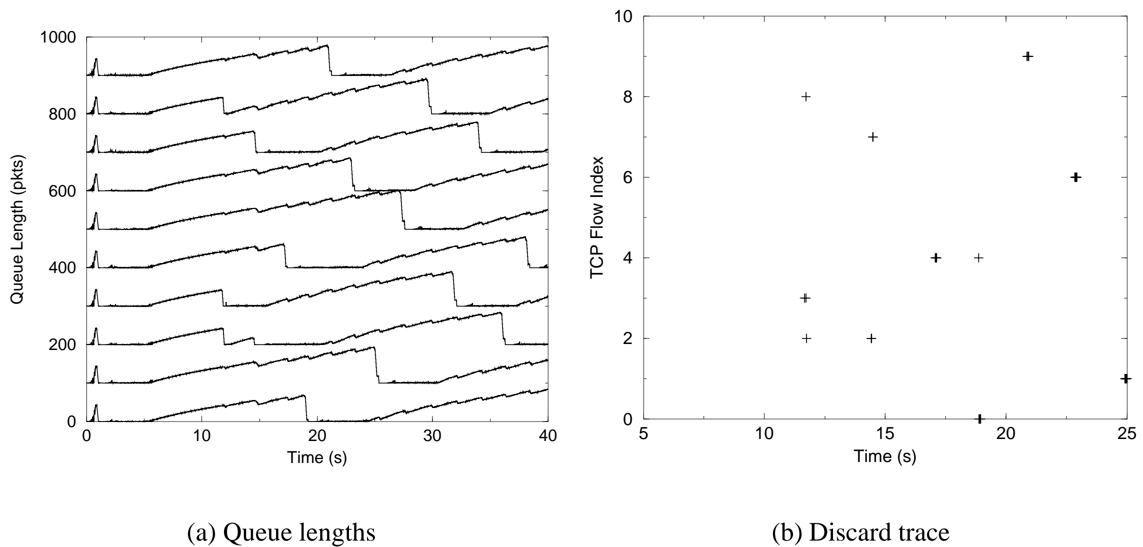
After the first discard interval, the flows which have experienced a drop will be sending at half their previous rate (since TCP will cut the congestion window size by half) and thus their queues will be nearly empty. The flows which have not experienced a drop will continue to increment their congestion window sizes and thus their corresponding queues will keep growing since they are sending a higher rate and we are using DRR which is a FQ packet scheduler. At the next packet discard interval, the discard queue picked will be forced to discard nearly all its packets before it becomes the shortest queue, since the flows that experienced discard earlier will have reduced their rates and their corresponding queues will be nearly empty. Thus, the number of flows affected during this interval will be significantly fewer. For subsequent drop intervals, the number of flows affected keeps reducing and quickly converges to one. From simulation results, we observe that after only two to three discard intervals, all subsequent drop intervals affect only one flow. This keeps the flows from getting re-synchronized while maintaining high throughput and link utilization.

Table 4.2: Number of flows affected during the first drop interval, comparing the analytical and simulation values

Number of flows	Buffer size (pkts)	Number of affected flows	
		Analytical	Simulation
100	4000	14	18
10	400	4	3

Table 4.2 compares the analytical model's prediction of the number of drop queues for QSDRR to the simulation model for a single bottleneck link configuration. We investigate two sets of input traffic: 100 flows with a 4,000 packet buffer and 10 flows with a 400 packet buffer. In both instances, the buffer size is equal to the bandwidth-delay product. From Table 4.2, we observe that the analytical prediction is very close to what we observe in the simulation.

4.2 Desychronizing effects of QSDRR



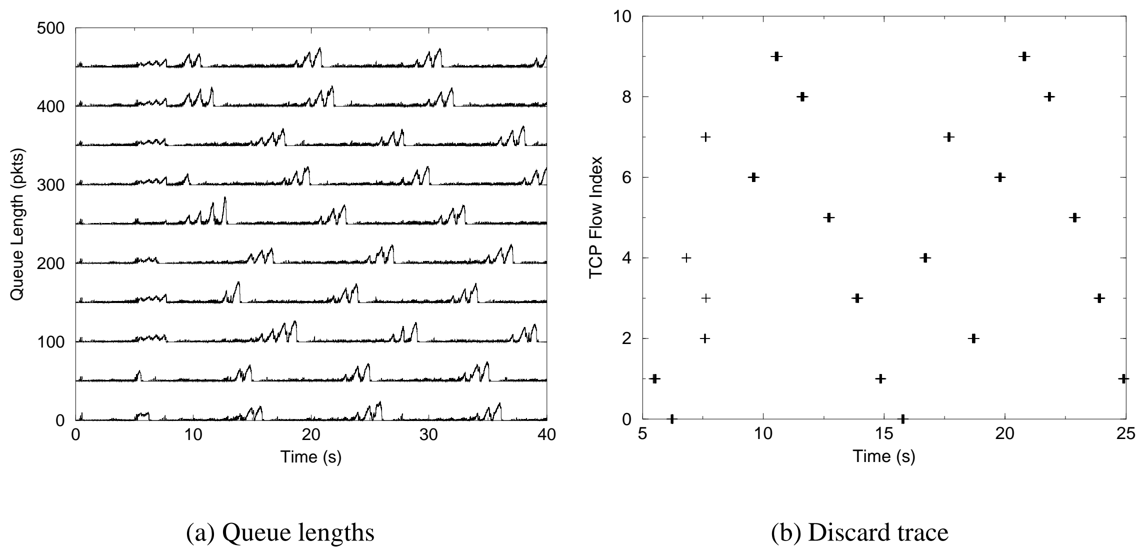
(a) Queue lengths

(b) Discard trace

Figure 4.1: Queue lengths for 10 flows with buffer size = 417 pkts

In this section, we take a closer look at the packet discard characteristics of QSDRR and how they help desynchronize TCP flows. To illustrate the TCP flow desynchronization property of QSDRR, we ran a simple ns-2 simulation with 10 TCP Reno flows over a 50

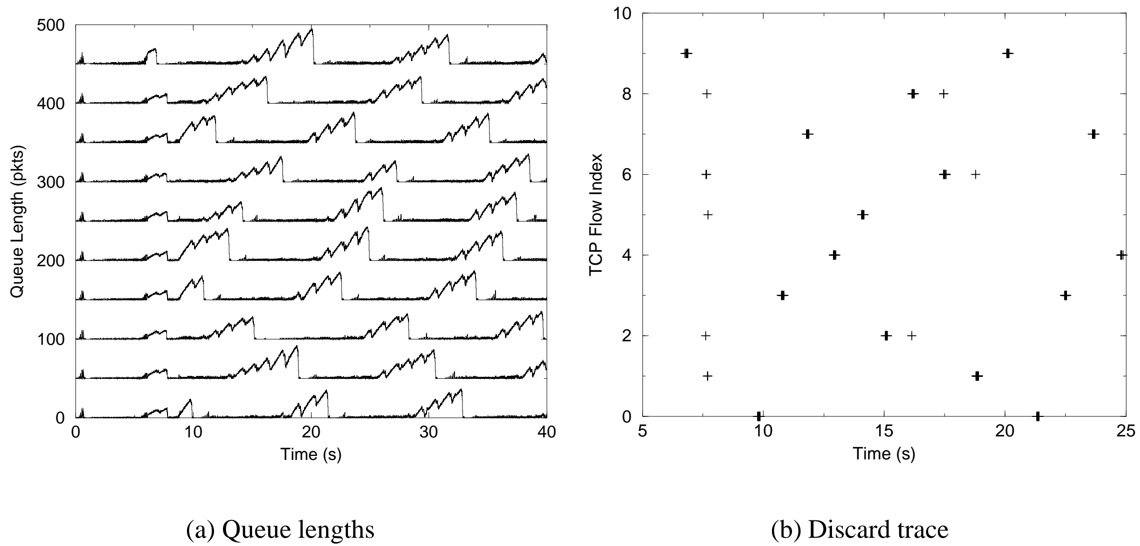
Mbps bottleneck link with a 100 ms round-trip time (RTT). Each TCP flow is connected by a 10 Mbps link to the bottleneck link and thus if all TCP sources send at the maximum rate, the overload ratio is 2:1. The bottleneck buffer was set to 10%, 20% and 100% of the bandwidth-delay product size of 417 packets. Figures 4.1, 4.2 and 4.3 show the time history of individual TCP queues at the bottleneck buffer. To easily view the data, each queue is offset (raised higher on the Y-axis) by a factor of 50 multiplied by the flow number starting from 0. Thus, flow 0's queue is raised by 0, flow 1's queue is raised by 50 and so on. For a buffer size of 417 packets, each queue is offset by a factor of 100.



(a) Queue lengths (b) Discard trace

Figure 4.2: Queue lengths for 10 flows with buffer size = 50 pkts

From Figures 4.1, 4.2 and 4.3, we notice that after the initial synchronous drop at 1 second (when all the sources are still in the slow-start, exponential increase phase), QS-DRR is able quickly to desynchronize the flows. In our analysis, we assume that all the TCP flows are in congestion avoidance phase and thus we ignore the initial slow-start, exponential increase phase. For the rest of this section, the *first* drop phase according to our analytic model is in reality the **second** drop period in the queue length graphs shown in the above figures. For a buffer size of 417 packets, shown in Figure 4.1, at the second drop period (11 seconds), only three flows (flows 2, 3 and 8) are affected. At the third drop period (14 seconds), only two flows (flows 2 and 7) are affected. Then, for all subsequent packet drop periods, only one flow is affected. Thus, at this point, all TCP flows are completely desynchronized and the queue remains near full occupancy enabling the TCP flows to achieve high throughput.



(a) Queue lengths

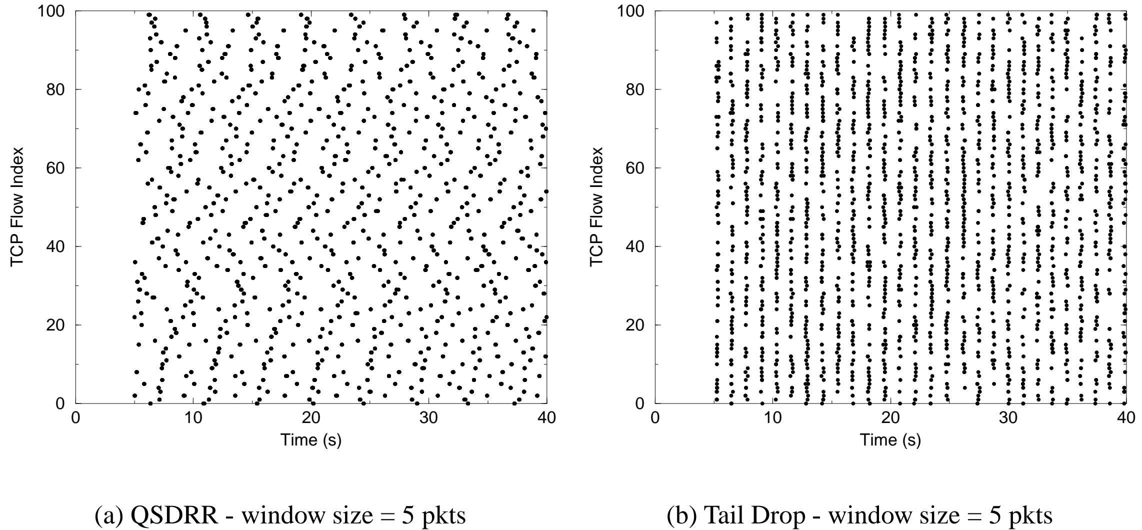
(b) Discard trace

Figure 4.3: Queue lengths for 10 flows with buffer size = 100 pkts

Figure 4.2 shows the individual queue lengths for a buffer size of 50 packets which is approximately 10% of the bandwidth-delay product. Even for such a small buffer, QSDRR only affects three flows (flows 0, 1 and 4) at the second drop period (5 seconds) and three flows (flows 2, 3 and 7) at the third drop period (7 seconds). Then, for all subsequent packet drop periods, only one flow is affected. Thus, QSDRR keeps all the TCP flows desynchronized allowing the link to be fully utilized and the TCP flows to achieve high throughput. Figure 4.3 shows the individual queue lengths for a buffer size of 100 packets which is slightly more than 20% of the bandwidth-delay product. The discard behaviour of QSDRR for this buffer size is very similar to that for the buffer size of 417 packets (100% of the bandwidth-delay product). Although six flows (flows 1, 2, 5, 6, 8 and 9) are affected in the second drop period (7 seconds), which is higher than the number affected for a buffer size of 417 packets, QSDRR quickly desynchronizes the flows and after the third drop period, only one flow is affected for all subsequent drop periods.

Overall, we observe that QSDRR is able to effectively desynchronize the TCP flows even for buffer sizes which are a fraction of the bandwidth-delay product. Another interesting thing to note is that the number of queues affected in the first drop period (during congestion avoidance phase) is very close to the number predicted by the analytical model. We recall from the previous section, that the analytical model predicts that QSDRR will affect four flows, whereas we observe that between three and six flows are affected from the simulation.

In the earlier experiments we described, each TCP flow's fair-share window size, W_0, is approximately 42 packets. For the next two experiments, we investigate the packet discard behaviour for QSDRR compared to Tail Drop for two different (and smaller) W_0 values of 5 and 25 packets.



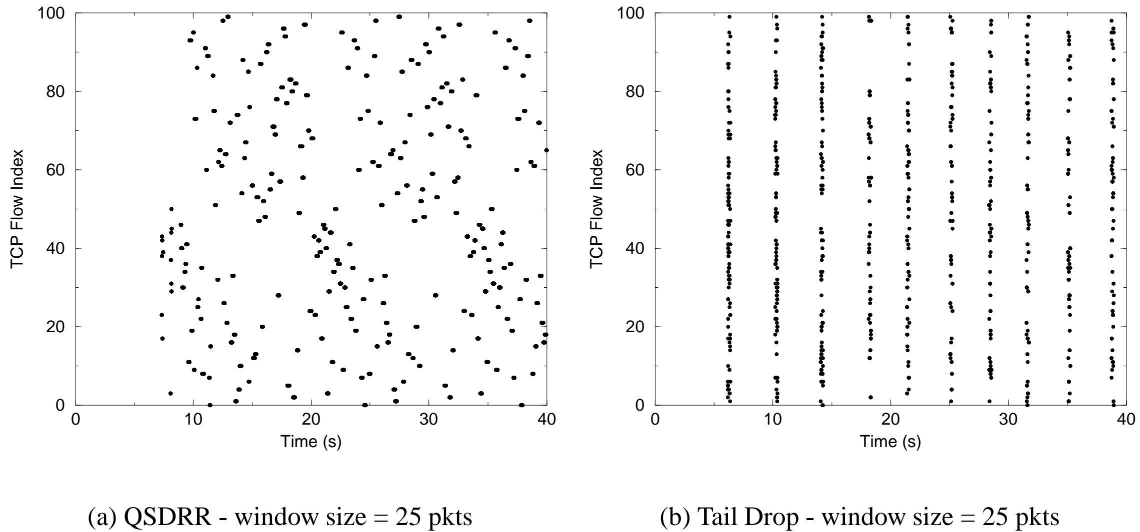
(a) QSDRR - window size = 5 pkts

(b) Tail Drop - window size = 5 pkts

Figure 4.4: Packet drop trace for QSDRR compared to Tail Drop for a fair share window size of 5 packets over a single bottleneck link

Figure 4.4 shows the time history of packet discards for QSDRR and Tail Drop for a W_0 of 5 packets. For this experiment, we used the single bottleneck link configuration with 100 TCP flows and a 100 ms RTT. The bottleneck link bandwidth was set to 60 Mb/s with packet size of 1500 bytes and a buffer size of 500 packets. In Figure 4.4, the Y-axis is the flow number of the TCP flow experiencing a packet drop. A straight line in this graph would indicate a series of flows experiencing a packet drop at the same time which leads to synchronization. We observe that the packet discard pattern for QSDRR is random for any particular packet discard time. However, for Tail Drop, during every packet discard interval a large percentage of flows experience drops, keeping those flows synchronized. Thus, even for small fair share window sizes per TCP flow, QSDRR is fairly effective at keeping the flows desynchronized.

Figure 4.5 shows the time history of packet discards for QSDRR and Tail Drop for a W_0 of 25 packets. For this experiment, we used the single bottleneck link configuration with 100 TCP flows and a 100 ms RTT. The bottleneck link bandwidth was set to 300 Mb/s with packet size of 1500 bytes and a buffer size of 2500 packets. In Figure 4.5, the Y-axis is the flow number of the TCP flow experiencing a packet drop. We observe that at higher fair



(a) QSDRR - window size = 25 pkts

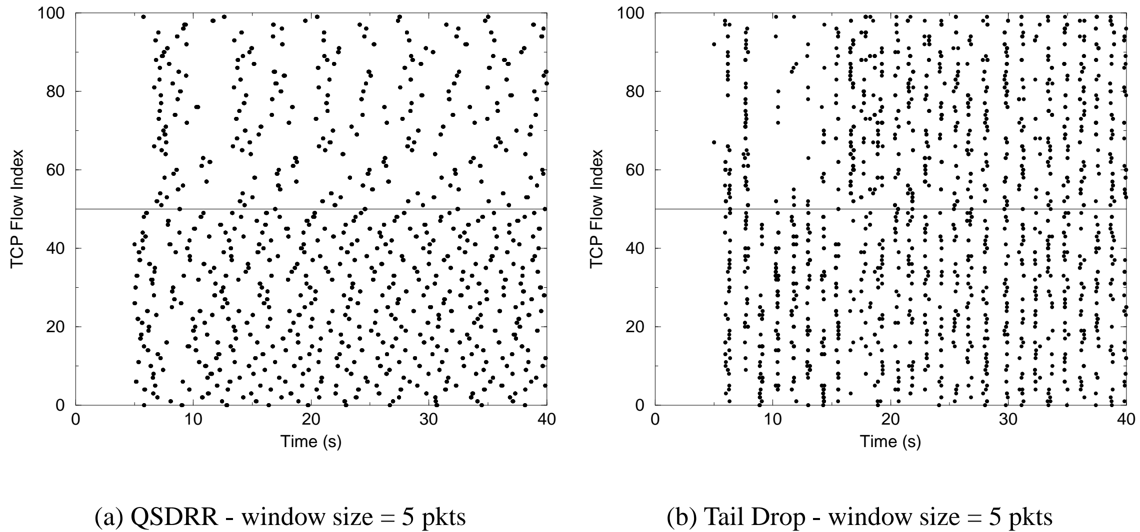
(b) Tail Drop - window size = 25 pkts

Figure 4.5: Packet drop trace for QSDRR compared to Tail Drop for a window size of 25 packets over a single bottleneck link

share window sizes per TCP flow, QSDRR is remarkably better at desynchronizing flows and the graph shows a random scattering of flows which experience packet discards. On the other hand, for Tail Drop, we again observe a majority of the flows experiencing packet discards simultaneously, thus causing them to remain synchronized.

In the next experiment, we used the multiple-RTT configuration to observe the packet discard characteristics of QSDRR compared to Tail Drop for TCP flows with different RTTs competing over a single bottleneck link. Figure 4.6 shows the time history of packet discards for QSDRR and Tail Drop for a W_0 of 5 packets. For this experiment, we used the multiple-RTT configuration with 100 TCP flows. Flows numbered 0-49 had an RTT of 20 ms and flows numbered 50-99 had an RTT of 400 ms. The bottleneck link bandwidth was set to 60 Mb/s with packet size of 1500 bytes and a buffer size of 500 packets. In Figure 4.6, the Y-axis is the flow number of the TCP flow experiencing a packet drop.

We observe that for the different RTT flows, QSDRR maintains its random discard pattern indicating little synchronization of the flows. On top of that, QSDRR also preferentially discards packets from the smaller RTT flows (which are also the higher rate flows), thus allowing the longer RTT flows (flows 50 to 100) to achieve nearly equal rates. In QSDRR, longer queues are preferentially selected for discard. Since shorter RTT TCP flows will be able to increase their window sizes faster than longer RTT TCP flows, the shorter RTT TCP flows will send at a higher rate. Thus, the queues for the shorter RTT flows will build up faster and be longer than the queues for the longer RTT TCP flows, since we use



(a) QSDRR - window size = 5 pkts

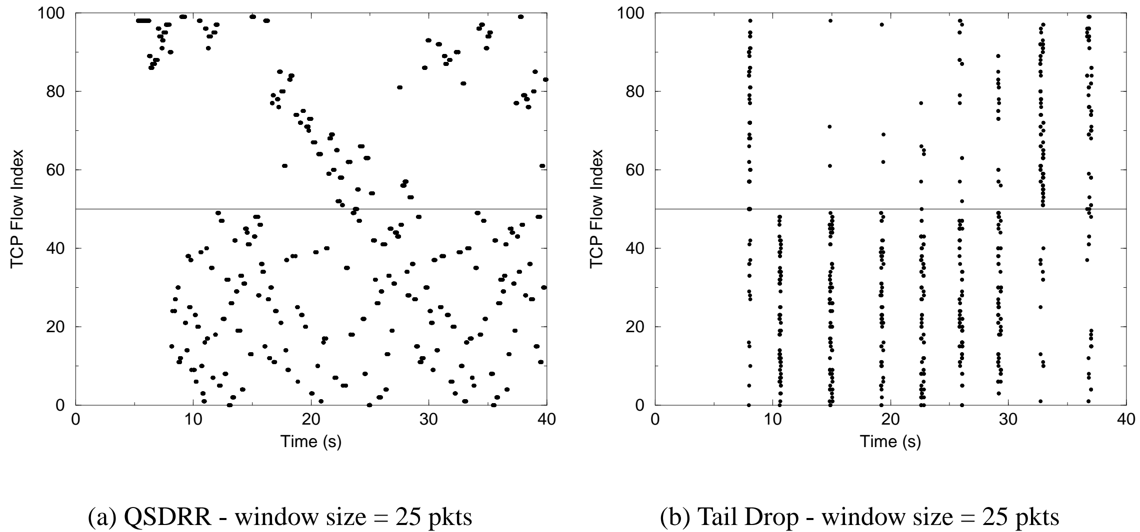
(b) Tail Drop - window size = 5 pkts

Figure 4.6: Packet drop trace for QSDRR compared to Tail Drop for a window size of 5 packets over a multiple-RTT configuration

DRR as our packet scheduler which is a fair queueing scheduler. This will lead to QSDRR choosing the shorter RTT flow queues for discard when the buffer overflows. Only when the longer RTT flows have achieved a rate equal or higher than the shorter RTT flows, their queues will become long and be considered for packet discard.

Under Tail Drop, we again observe that a large number of flows are affected simultaneously during a packet discard event and thus the flows remain synchronized. Also, although there is a slight decrease in the probability of discard for the longer RTT flows, it is not significant and does not allow them to achieve the same rates as shorter RTT flows.

Figure 4.7 shows the time history of packet discards for QSDRR and Tail Drop for a W_0 of 5 packets. For this experiment, we used the multiple-RTT configuration with 100 TCP flows. Flows numbered 0-49 had an RTT of 20 ms and flows numbered 50-99 had an RTT of 400 ms. The bottleneck link bandwidth was set to 300 Mb/s with packet size of 1500 bytes and a buffer size of 2500 packets. In Figure 4.7, the Y-axis is the flow number of the TCP flow experiencing a packet drop. For this higher value of W_0, QSDRR does a remarkable job of preferentially dropping the shorter RTT flows (flows 0 to 49) and allowing the longer RTT flows (flows 50 to 100) to achieve nearly equal bandwidth. For Tail Drop, we do observe a lower percentage of discards for the longer RTT flows (flows 50 to 99), but the straight lines in the graph again indicate a large number of flows being affected simultaneously leading to synchronization. Overall, we observe that even for lower W_0



(a) QSDRR - window size = 25 pkts

(b) Tail Drop - window size = 25 pkts

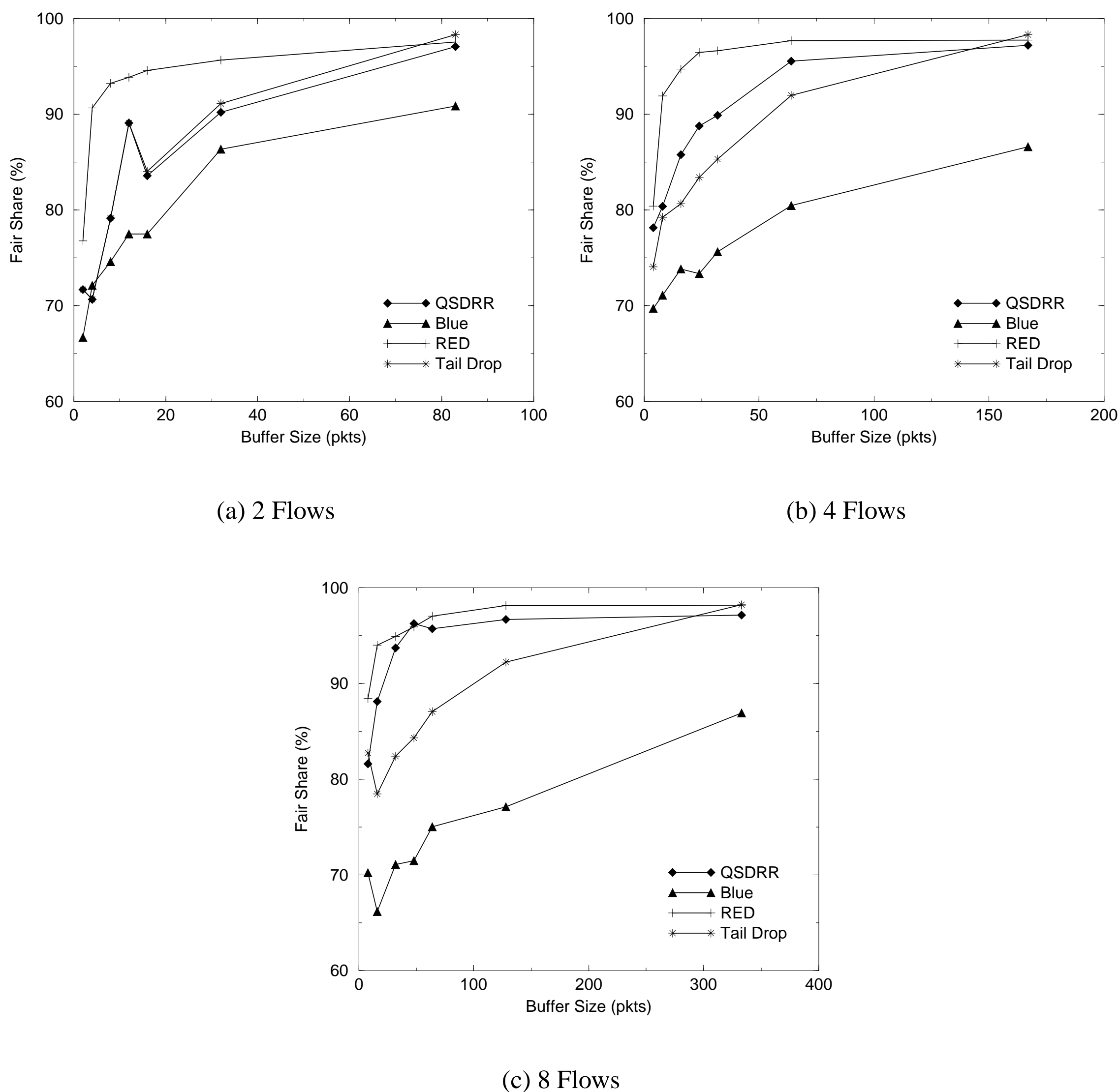
Figure 4.7: Packet drop trace for QSDRR compared to Tail Drop for a window size of 25 packets over a multiple-RTT configuration

values, QSDRR is able to effectively desynchronize the TCP flows for different network configurations.

4.3 QSDRR with a Small Number of TCP Flows

The majority of experiments we have conducted are for a reasonably large number of TCP flows. In this section, we evaluate QSDRR for a very small number of TCP flows. We vary the number of flows from 2 to 8 and run experiments over two different network configurations: single bottleneck link configuration and multiple-RTT configuration.

Figure 4.8 compares the fair share performance of a small number of TCP flows over a single bottleneck link. The bottleneck link bandwidth is set to 10 Mb/s for 2 flows, 20 Mb/s for 4 flows and 40 Mb/s for 8 flows. Since each incoming flow is limited to a maximum bandwidth of 10 Mb/s, the bottleneck link ensures a 2:1 overload. We observe from the graphs that QSDRR is not as effective for a small number of flows. In fact, RED has the best performance for a small number of flows over a single bottleneck link. One reason why QSDRR is not able to perform as well is that it depends on packet discard hysteresis to achieve fairness and with a very small set of flows to choose from, the hysteresis effect is not significant which results in poor performance.



(a) 2 Flows

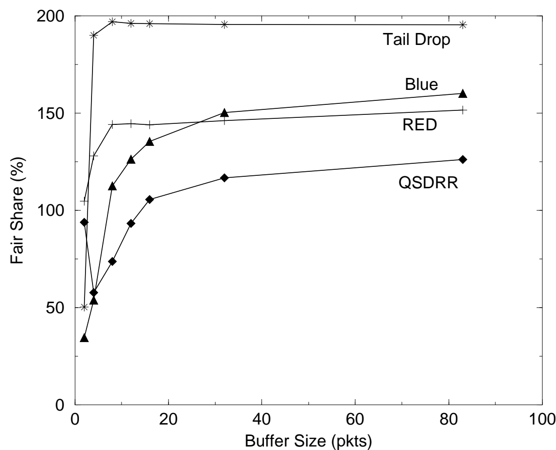
(b) 4 Flows

(c) 8 Flows

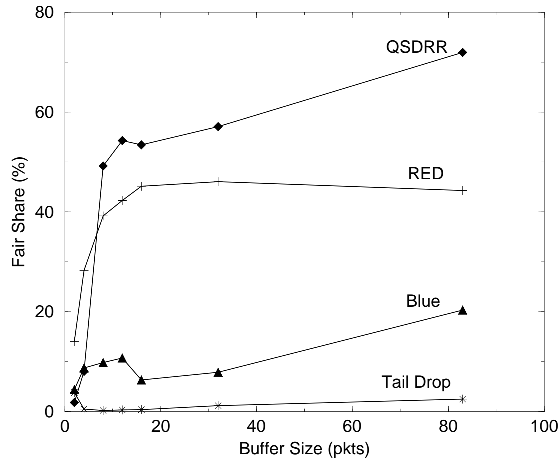
Figure 4.8: Fair share performance of a small number of TCP flows over a single bottleneck link

Figures 4.9, 4.10 and 4.11 compare the fair share performance of a small number of TCP flows over a multiple-RTT configuration. The bottleneck link bandwidth is set to 10 Mb/s for 2 flows, 20 Mb/s for 4 flows and 40 Mb/s for 8 flows. Since each incoming flow is limited to a maximum bandwidth of 10 Mb/s, the bottleneck link ensures a 2:1 overload. For each set of experiments, half the flows have an RTT of 20 ms while the other half have an RTT of 200 ms. From the graphs, we observe that even for a small number of flows, QSDRR's packet discard algorithm performs significantly better than RED and Blue since

it will preferentially be able to discard packets from higher rate flows (shorter RTT TCP flows).

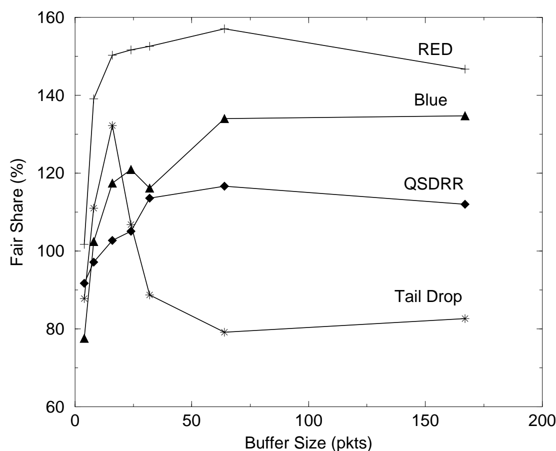


(a) Flows with RTT = 40 ms

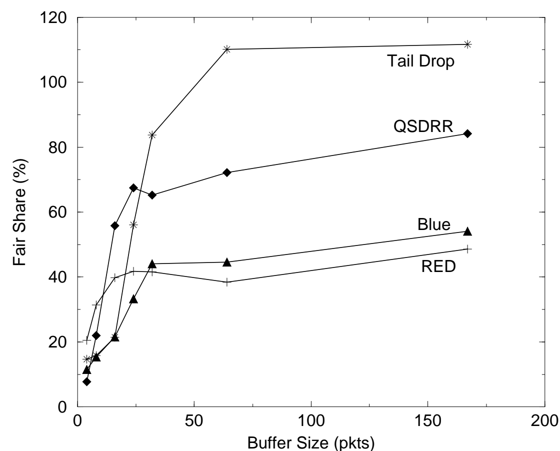


(b) Flows with RTT = 200 ms

Figure 4.9: Fair share performance of 2 TCP flows with different RTTs over a single bottleneck link



(a) Flows with RTT = 40 ms

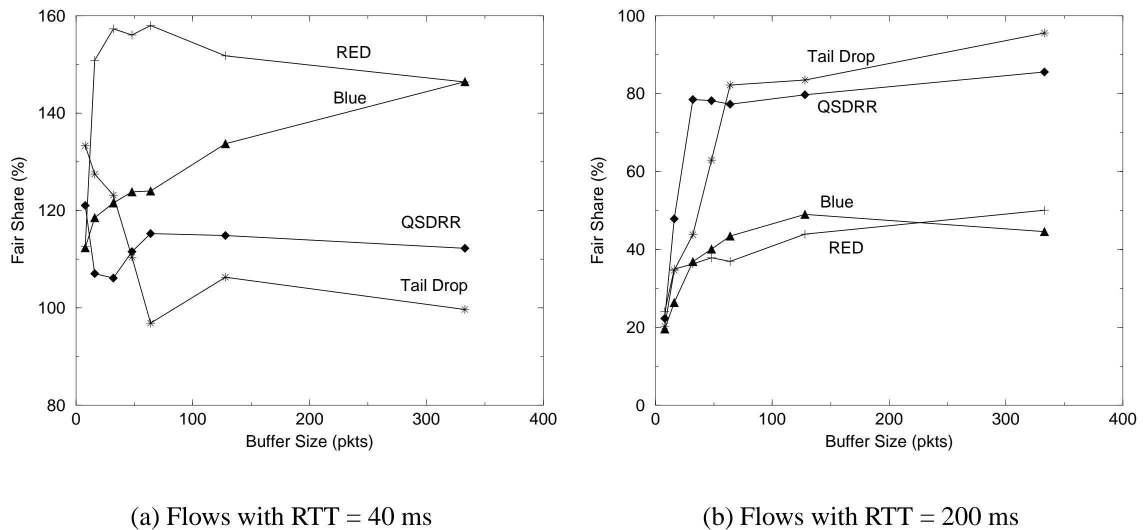


(b) Flows with RTT = 200 ms

Figure 4.10: Fair share performance of 4 TCP flows with different RTTs over a single bottleneck link

One interesting thing to note is that for a very small number of flows, the average fair share performance for the 200 ms RTT flows under Tail Drop is comparable to QSDRR

and significantly outperforms RED and Blue. One reason why Tail Drop is able to deliver good average performance is that for a small number of flows, if one 200 ms RTT flow escapes being discarded initially, it can significantly increase its congestion window size before the next discard interval. This allows the flow to reach and maintain a very high throughput and significantly increase the average throughput for all 200 ms RTT flows.



(a) Flows with RTT = 40 ms (b) Flows with RTT = 200 ms

Figure 4.11: Fair share performance of 8 TCP flows with different RTTs over a single bottleneck link

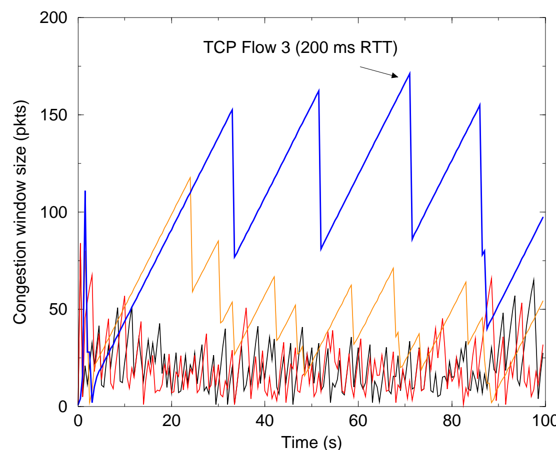


Figure 4.12: Time history of congestion window sizes for 4 TCP flows using Tail Drop

Figure 4.12 illustrates this point by showing the time history of the congestion window sizes for four TCP flows. The configuration used for this experiment is the multiple-RTT configuration with four TCP flows with a buffer size of 32 packets. Two flows have an

RTT of 40 ms while the other two flows have an RTT of 200 ms. In Figure 4.12, the lower two lines are the congestion window sizes for the 40 ms RTT flows. The middle line is the congestion window size of one 200 ms RTT flow. This flow does get affected early and is not able to substantially increase its window size. The top line represents the other 200 ms RTT flow. This flow escapes packet discards early and is able to significantly increase its congestion window size and achieve almost twice the throughput compared to the 40 ms RTT flows. Thus, the average throughput of the 200 ms RTT flows is appreciably increased due to the throughput of the last flow.

4.4 Experimental Results

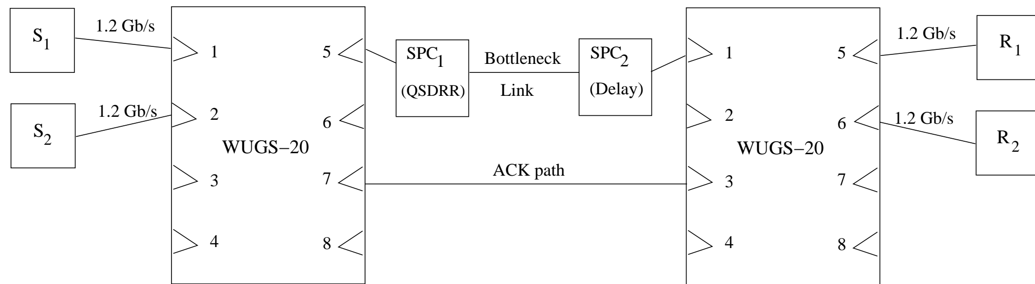
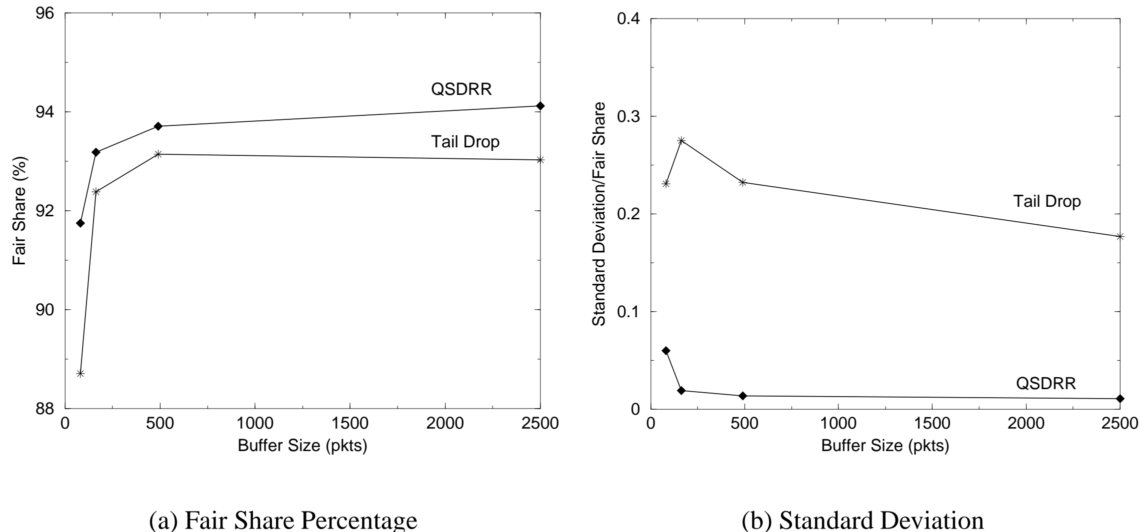


Figure 4.13: Experimental configuration using two Multi-Service Routers and four NetBSD hosts

We have implemented the QSDRR algorithm in the Smart Port Card (SPC) [18] which resides on the ports of the Multi-Services Router (MSR) [12]. The Smart Port Card (SPC) consists of an embedded Intel processor module, 64 MBytes of DRAM, an FPGA (Field Programmable Gate Array) that provides south bridge functionality, and a Washington University APIC ATM hostnetwork interface [20]. The SPC runs a version of the NetBSD operating system [1] that has been substantially modified to support fast packet forwarding, active network processing and network management. See [18] for additional details. On the SPC, ATM cells are handled by the APIC [21, 22]. Each of the ATM ports of the APIC can be independently operated at full duplex rates ranging from 155 Mb/s to 1.2 Gb/s. The APIC supports AAL-5 and is capable of performing segmentation and reassembly at maximum bus rate (1.05 Gb/s peak for PCI-32). The APIC directly transfers ATM frames to and from host memory and can be programmed so that cells of selected channels pass directly from one ATM port to another. We have customized NetBSD to use a disk image stored in main memory, a serial console, a self configuring APIC device driver



(a) Fair Share Percentage

(b) Standard Deviation

Figure 4.14: Experimental fair share performance and standard deviation for 32 TCP flows over a single bottleneck link

and a fake BIOS. The fake BIOS program acts like a boot loader: it performs some of the actions which are normally done by a Pentium BIOS and the NetBSD boot loader during power-up.

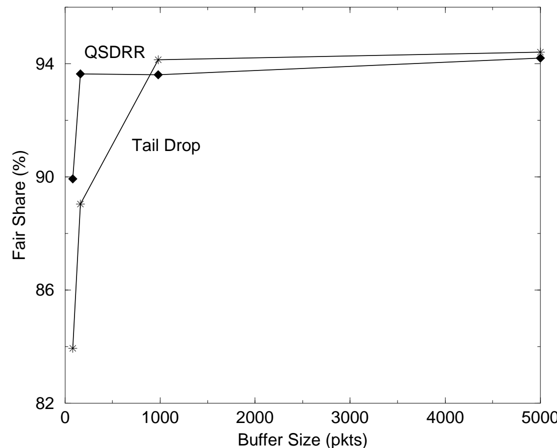
Figure 4.13 illustrates the experimental setup that we used to test QSDRR using the MSR. We conducted experiments with the number of sources ranging from 32 to 60. The bottleneck bandwidth was limited by the maximum throughput capacity of the SPC card which is about 70 Mb/s. For our experiments we varied the bottleneck bandwidths from 20 Mb/s to 55 Mb/s. We used 576 byte packets which equates to a TCP MSS of 536 bytes. The buffer size was varied from 10% of the bandwidth-delay product to a maximum of a half-second buffer.

Using our experimental setup, we wanted to investigate two issues:

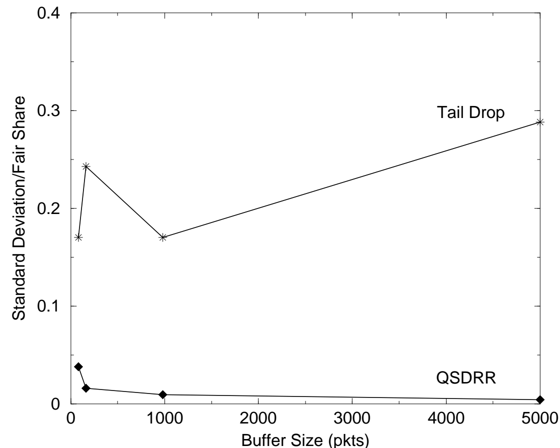
- Whether using QSDRR increases the overhead of packet processing compared to Tail Drop such that we experience a reduction in the total throughput capacity of the router port.

- Do the simulation results obtained accurately predict the performance we would observe for TCP flows over an actual network.

From our experiments, we noticed no change in the total throughput achieved under QSDRR compared to Tail Drop for the SPC card. We observed that the per-packet IP processing time in software was greater than the overhead imposed by QSDRR.

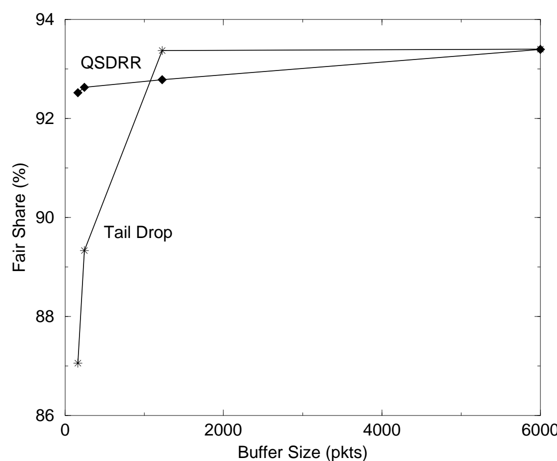


(a) Fair Share Percentage

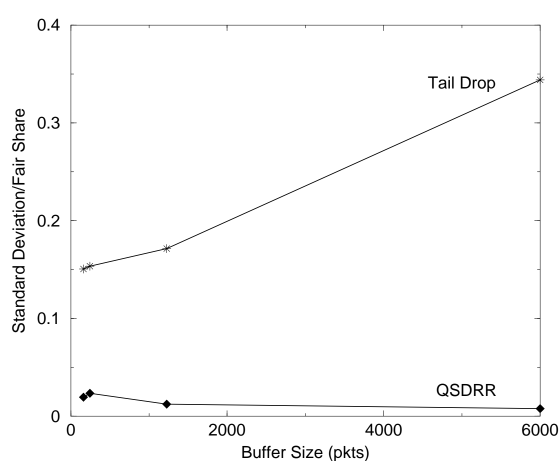


(b) Standard Deviation

Figure 4.15: Experimental fair share performance and standard deviation for 48 TCP flows over a single bottleneck link



(a) Fair Share Percentage



(b) Standard Deviation

Figure 4.16: Experimental fair share performance and standard deviation for 60 TCP flows over a single bottleneck link

Figures 4.14, 4.15 and 4.16 show the fair share percentage and standard deviation for the TCP flows under QSDRR compared to Tail Drop. The results observed in these graphs agree extremely well with what we observed via simulation. Even at very small buffer sizes (10% of the bandwidth-delay product), QSDRR is able to ensure higher than 90% of the fair share throughput on average to the TCP flows. Also, the standard deviation

obtained under QSDRR is extremely low compared to Tail Drop, remaining below 0.05 for all buffer sizes compared to 0.2 and higher for Tail Drop. Due to throughput restrictions for the SPC, we were not able to exactly mimic the simulation environment of a 100 TCP flow and a 500 Mb/s link, but the above experiments are reasonably comparable. Also, the results show that QSDRR is able to perform as well as anticipated by the simulation results and does not add any significant overhead which would lead to loss of throughput capacity in the router.

4.5 Usability over Large Networks

A natural question is that though the simulation results show good performance for our selected simulation configuration (1000 sources and 500 Mb/s bottleneck link), will we still achieve the same performance when we have 20,000 sources over a 10 Gb/s link? Realistically, it is impractical to simulate such large network configurations. However, we attempt to address this issue by showing that, if the ratio of the buffer size to the link bandwidth-delay product is held invariant, the performance is fairly insensitive to a wide range of changes in parameters such as number of sources, RTT and bottleneck link capacities. In our study, we varied the number of sources from 10 to a 1000, RTT times from 6 ms to 1.5 s and bottleneck link bandwidths from 20 Mb/s to 3 Gb/s.

4.5.1 Simulation Setup

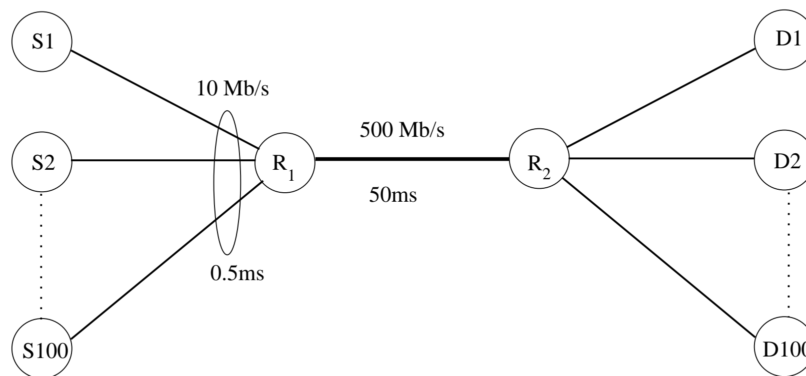


Figure 4.17: Single Bottleneck Link Network Configuration

We use the single bottleneck link configuration as shown in Figure 4.17 as our base topology. For each set of experiments (graphs), we evaluate four different *fair-share window sizes* (2, 10, 50, 100) for a TCP source. We define *fair-share window size* as the

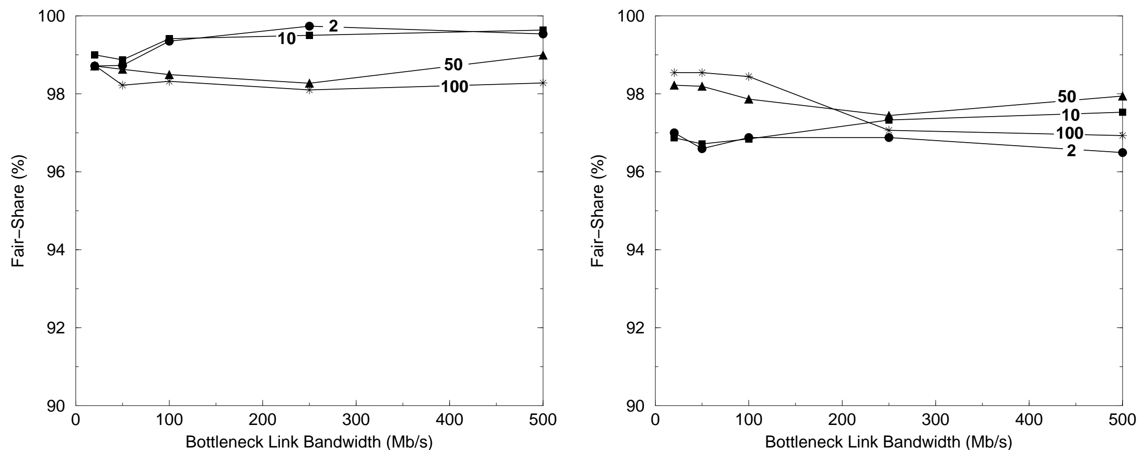
fair-share bandwidth per TCP flow times the RTT. The bottleneck link buffer is set to the bandwidth-delay product of the network configuration. The three different simulation scenarios we study are outlined below.

1. **Varying bottleneck link bandwidth**
   For this experiment, we set the number of sources to 100 and vary the bottleneck link bandwidth from 20Mb/s to 500Mb/s. The RTT is scaled along with the bottleneck bandwidth to maintain the constant fair-share window size.

2. **Varying number of sources**
   For this experiment, we set the RTT to 100ms and vary the number of sources from 20 to 500. The bottleneck link bandwidth is scaled along with the number of sources to maintain the constant fair-share window size.

3. **Varying RTT**
   For this experiment, we set the bottleneck link bandwidth to 500Mb/s and vary the RTT from 6ms to 120ms. The number of sources are scaled along with the RTT to maintain the constant fair-share window size.

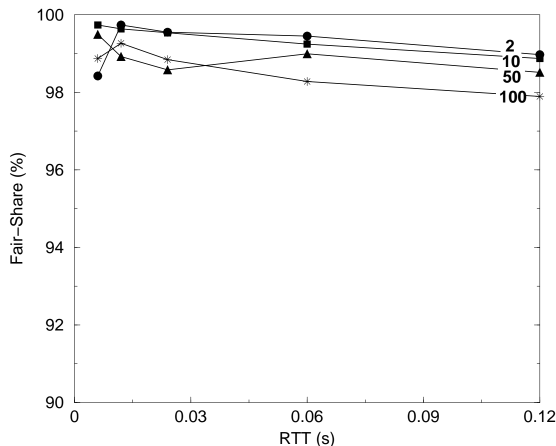
4.5.2 Results



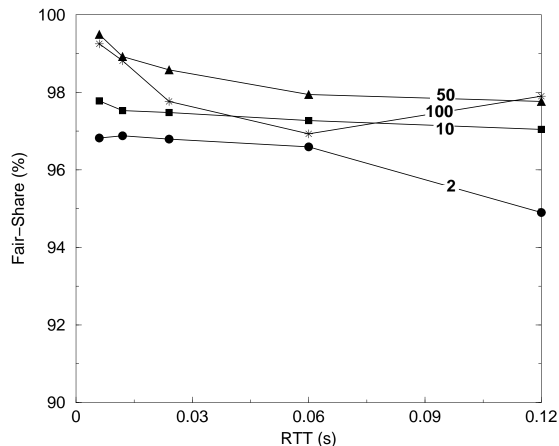
(a) Tail Drop: Changing Bottleneck Bandwidth, 100 sources

(b) QSDRR: Changing Bottleneck Bandwidth, 100 sources

Figure 4.18: Performance of TCP flows over single bottleneck link while varying the bottleneck link



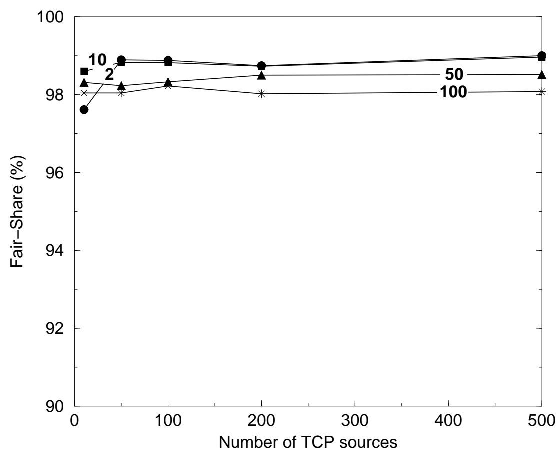
(a) Tail Drop: Changing RTT, 500 Mb/s bottleneck



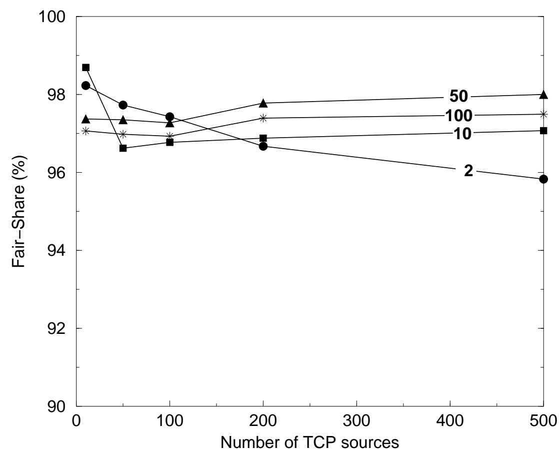
(b) QSDRR: Changing RTT, 500 Mb/s bottleneck

Figure 4.19: Performance of TCP flows over single bottleneck link while varying the RTT

Figure 4.18, 4.19 and 4.20 show the effect on the mean fair-share goodput percentage achieved by the TCP flows using Tail Drop policy compared to using QSDRR policy. From the above graphs, it is clear that changing bottleneck-link bandwidths, RTTs, number of sources or fair-share window sizes has a negligible impact on TCP goodputs over a



(a) Tail Drop: Changing number of sources, 100 ms RTT



(b) QSDRR: Changing number of sources, 100 ms RTT

Figure 4.20: Performance of TCP flows over single bottleneck link while varying the number of sources

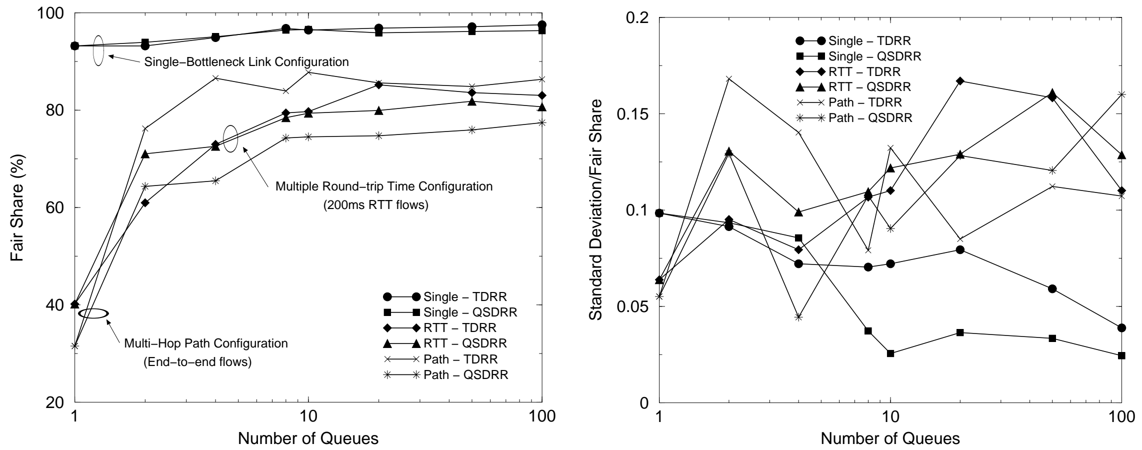
congested link. This tells us that we can reliably apply results from a smaller scale simulation to a larger scale scenario, assuming that the simulations are for similar fair-share bandwidths.

Chapter 5

Scalability Issues

One drawback with a fair-queueing policy such as DRR is that we need to maintain a separate queue for each active flow. Since each queue requires a certain amount of memory for the linked list header, used to implement the queue, there is a limit on the number of queues that a router can support. In the worst-case, there might be as many as one queue for every packet stored. Since list headers are generally much smaller than the packets themselves, the severity of the memory impact of multiple queues is intrinsically limited. On the other hand, since list headers are typically stored in more expensive SRAM, while the packets are stored in DRAM, there is some legitimate concern about the cost associated with using large numbers of queues. One way to reduce the impact of this issue is to allow multiple flows to share a single queue. While this can reduce the performance benefits observed in the previous sections, it may be appropriate to trade off performance against cost, at least to some extent. To address this issue, we used simulations to study the effects of merging multiple flows into a single queue. Figure 5.1 illustrates the effects of varying the number of queues. The sources are TCP Reno and the total buffer space is fixed at 1000 packets.

Figure 5.1(a) illustrates the effect on the goodput received by each flow under different numbers of queues. For the multiple round-trip time configuration and the multi-hop path configuration, we show the goodput for the 200 ms RTT (longer RTT) flows and the end-to-end (multi-hop) flows respectively. In both these configurations, the above mentioned flows are the ones which receive a much lower goodput compared to their fair share under existing policies such as RED, Blue and Tail Drop. We observe that the effect of increasing the number of buckets produces diminishing returns once we go past 10 buckets. In fact, there is only a marginal increase in the goodput received when we go from 10 buckets to 100 buckets. Since at each bottleneck link there are a 100 TCP flows, this



(a) Fair Share Percentage

(b) Standard Deviation in goodput over Fair Share Bandwidth

Figure 5.1: Performance of TDRR and QSDRR for a buffer size of 1000 packets, with varying number of buckets

implies that our algorithms are scalable and can perform very well even with *one-tenth* the number of queues as flows.

We also present the standard deviation in goodput received by each flow for different numbers of queues in Figure 5.1(b). The results are presented as a ratio of the standard deviation to the fair share bandwidth to better illustrate the measure of the standard deviation. We notice that changing the number of queues does not have a significant impact on the standard deviation of the goodputs, and thus we do not lose any fairness by using much fewer queues, relative to the number of flows. Also, the overall standard deviation is below 15% of the fair share goodput for all our multi-queue policies, regardless of the number of queues.

The results presented above assume that the flows are evenly distributed among all the available queues. This a best-case scenario and impractical to implement in a real router. The main problem with using a simple hash function to randomly distribute flows among queues is that the distribution will be unbalanced. This is especially true for small flow to queue ratios. Consider an example where we want to randomly distribute 1024 flows over 64 queues. Using the binomial distribution, the mean number of flows per queue is 16 and the standard deviation is approximately 4. Thus the ratio of the maximum to the minimum number of flows per queue can be as high as 4 or 5. This has a significant impact on fairness, as the flows in the high occupancy queues will obtain a much smaller share of the

link bandwidth compared to flows in low occupancy queues. When we have a large number of flows sharing a few queues, this problem is much less severe. For example, consider a million flows sharing 64 queues. The mean number of flows per queue is 16,000 and the standard deviation is approximately 125. Thus the ratio of the maximum to minimum number of flows per queue is fairly close to 1, limiting the impact of unfair link bandwidth distribution among flows.

In this chapter, we address the issue of evenly distributing flows among queues for reasonably small flows to queue ratios, using Bloom filters [7] acting as flow counters.

5.1 Flow Distribution Algorithm

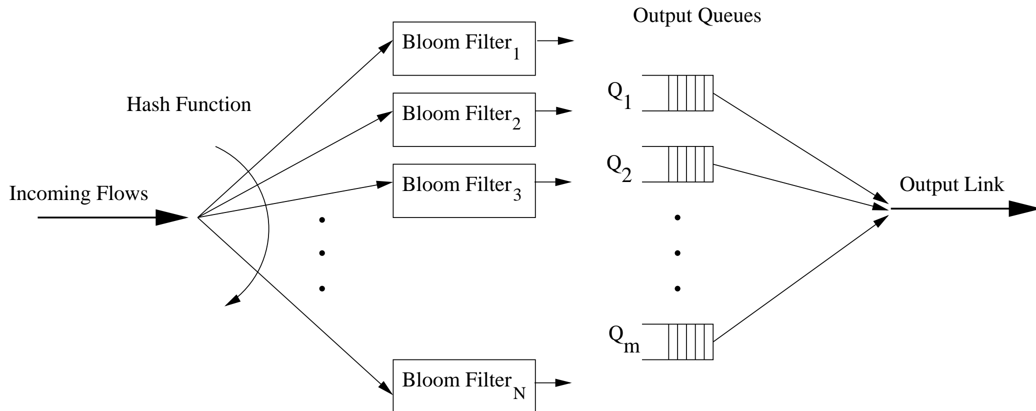


Figure 5.2: Flow distribution using Bloom Filters

As shown in Figure 5.1(a), QSDRR performs well even with multiple flows sharing a single queue if the flows are evenly distributed among the queues. Thus, in this section we present an algorithm for evenly distributing flows among queues. The algorithm consists of two parts:

1. Bloom filters [7] which are used to implement approximate flow counters.

2. Policy for distributing flows to queues, based on the approximate flow counts.

5.1.1 Overview of Bloom filters

A Bloom filter for representing a set $S = \{x_1, x_2, ..., x_n\}$ of n elements is described by an array of b bits, initially all set to 0. A Bloom filter uses k independent hash functions

$h_1, ..., h_k$ with range $\{1, ..., b\}$. We assume that the hash functions map each element in the universe to a random number uniform over the range $\{1, ..., b\}$. To check if an element y is in S, we check to see whether all $h_i(y)$ are set to 1. If they are not, then y is clearly not in S. If they are all set to 1, we assume that y is in S, though we may be wrong with some probability. Hence, a Bloom filter may yield a *false positive*, where it suggests that an element y is in S even though it is not. Thus, a Bloom filter is a simple space-efficient randomized data structure for representing a set in order to support approximate membership queries. The space efficiency is achieved at a cost of a small probability of false positives [11].

5.1.2 Bloom filter architecture

Figure 5.2 shows our proposed Bloom filter architecture. First, we hash incoming flows into N Bloom filters, where N is larger than the number of queues. Each flow sets k bits in its Bloom filter array, where k is the number of hash functions in the Bloom filter. Using this Bloom filter array, we can maintain an approximate count of the number of flows mapped to each Bloom filter and we can use this to evenly distribute flows across the output queues. To maintain a count of the number of flows, we associate a counter with each Bloom filter. This counter is incremented when a flow sets at least one bit in the Bloom filter array from a 0 to a 1. If the number of false positives is very low, each new flow will set at least one unique bit in the array from a 0 to a 1. Thus, using the above counter, we can get a fairly accurate estimate of the number of flows mapped to each Bloom filter.

We illustrate the SRAM memory savings obtained using a Bloom filter architecture compared to per-flow queueing using a simple example with 100,000 flows.

- **Per-flow Queueing**
  For classification of TCP flows, we need to store the following 5-tuple per flow: source and destination addresses (32 bits each), source and destination ports (16 bits each) and the protocol field (8 bits). Thus, we need to store 13 bytes per flow for flow classification. Since there are 100,000 queues, we need 3 bytes each for the queue head and tail. Hence, we need a total of 19 bytes per flow and 1.9 MB for 100,000 flows.

- **Bloom filter architecture**
  We hash the incoming flows into 20,000 Bloom filters and use 2,000 outgoing queues. Each filter will handle 5 flows on the average. From [11], the false positive rate

for a filter is $(0.6185)^{m/n}$, where m is the number of bits in the Bloom filter array representing a set of n elements. For our example, n is 5 and thus we need m to be 50 bits to achieve a false positive probability of less than 1%. We also need 4 bits for the counter. Hence, we need 7 bytes per Bloom filter. Since we only have 2,000 output queues, we need 2 bytes each for the queue head and tail. Thus the total memory required for the Bloom filter architecture is 148 KB, which results in a 13:1 reduction in SRAM usage.

Table 5.1: SRAM memory needed for per-flow queueing vs. Bloom filter architecture for 100,000 flows

	Per-flow Queueing	Bloom filter	
	Queues	Bloom Filters	Queues
Number	100,000	20,000	2,000
SRAM needed	1.9 MB	140 KB	8 KB
SRAM reduction	-	13:1	

Table 5.1 summarizes the SRAM memory savings obtained using the Bloom filter architecture compared to per-flow queueing for an example scenario of 100,000 flows. Note that this is intended just as an illustrative example. A systematic examination of alternative parameter choices would likely produce greater savings.

5.1.3 Distribution Policy

```
Sort all Bloom filters in descending
  order based on their flow counts.
Assume B[] is the resulting sorted
  array of Bloom filters.
Set Q to be the current minimum queue
for (i = 0; i < number of filters; i++)
  Assign B[i] to Q
  Set Q.count <- Q.count + B[i].count
  Set Q to current minimum queue
end
```

Figure 5.3: Algorithm for distributing Bloom filters among queues

We use a fairly simple algorithm to distribute Bloom filters among the available queues. First, we sort the Bloom filters in descending order according to their flow *counts*. Then, we simply assign each Bloom filter (in order) to the queue with the current minimum flow count. A detailed description of the algorithm is presented in Figure 5.3.

Table 5.2: Max/min flows queue ratios for Bloom architecture vs. simple hashing

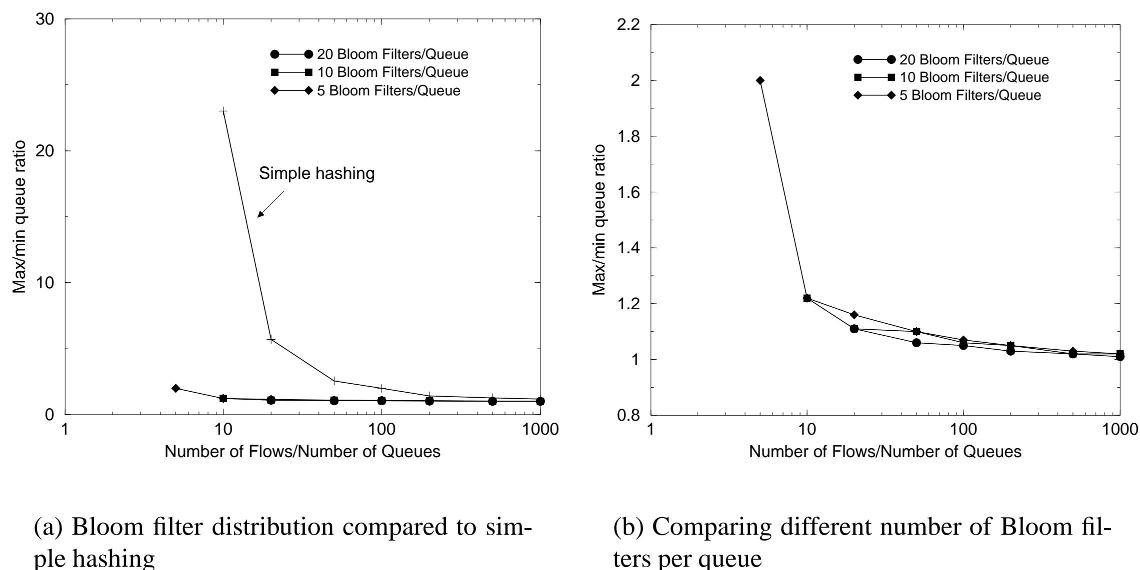
Mean Address Hold Time (s)	Average max/min queue ratio	
(Flows,Filters,Queues)	Bloom	Hash
(100000,20000,2000)	1.10	2.55
(10000,2000,200)	1.06	2.52
(1000,200,20)	1.04	1.62

Table 5.2 shows the ratios of the queue with the most flows over the queue with the least flows. From now on, we refer to this ratio as the **max/min queue ratio**. Ideally, for a perfectly even distribution, this ratio should be one. From Table 5.2, we observe that by using our Bloom filter architecture and distribution algorithm, we can achieve a near optimal distribution of flows. There will always remain a slight deviation from the exact optimum ratio of one since we cannot split flows that belong to a single Bloom filter across multiple queues. In comparison, simple hashing achieves a much worse ratio of 1.62 for 1,000 flows to 2.55 for 100,000 flows.

5.1.4 Bloom filter queue ratios and memory usage

In this section, we investigate in more detail the max/min queue ratios obtained using our Bloom filter distribution policy compared to simple hashing over a wide range of flow to queue ratios. We also investigate the memory requirements for the Bloom filters over this range of flow to queue ratios. For the simulation setup, we use 100,000 flows and vary the number of queues from a 100 to 100,000.

Figure 5.4(a) shows the max/min queue ratio obtained using three Bloom filter policies (different number of Bloom filters per queue) compared to simple hashing. Using simple hashing, for flow to queue ratios less than 5, there remain some queues that are empty (no flows mapped) and thus the max/min queue ratio is infinite. From Figure 5.4(a), we observe that the max/min queue ratio stays very close to 1 (which is the ideal) for all three Bloom filter policies. Under simple hashing, the max/min queue ratio remains significantly higher (above 3) for flow to queue ratios lower than 100. We observe from Figure 5.4(b)



(a) Bloom filter distribution compared to simple hashing

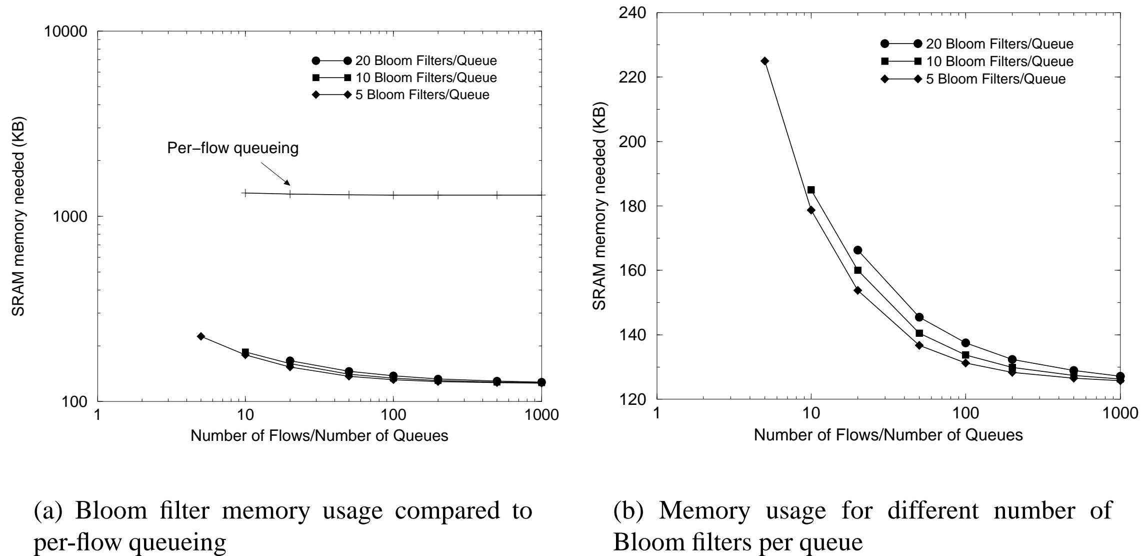
(b) Comparing different number of Bloom filters per queue

Figure 5.4: Max/min queue ratios for Bloom filter architecture compared with simple hashing for flow to queue ratios from 1 to 1,000

that even when using very few Bloom filters per queue (5 Bloom filters per queue), we can achieve near ideal max/min queue ratios. Also, for all three Bloom filter policies, the max/min queue ratio remains below 1.2 for flow to queue ratios greater than 10.

Figure 5.5(a) compares the SRAM memory requirements for per-flow queueing to that for the three Bloom filter policies. To calculate memory requirements for the Bloom filter methods, we allocated a sufficient number of bits to each Bloom filter array such that the false positive probability remained below 1%. Also, the per-Bloom filter counter was allocated enough bits to allow it to count upto three times the flows to Bloom filter ratio without overflowing.

From Figure 5.5(a), we observe that for all flow to queue ratios, the Bloom filter policies require and order of magnitude *less* memory compared to per-flow queueing. Also, from Figure 5.5(b), we observe that the difference in memory requirements for the three different Bloom filter to queue ratio policies is fairly small. Since we have already observed (from Figure 5.4(b)) that the max/min queue ratios are nearly identical for all three policies, we can safely choose the Bloom filter policy with the least memory requirement without any tradeoff in performance.



(a) Bloom filter memory usage compared to per-flow queueing

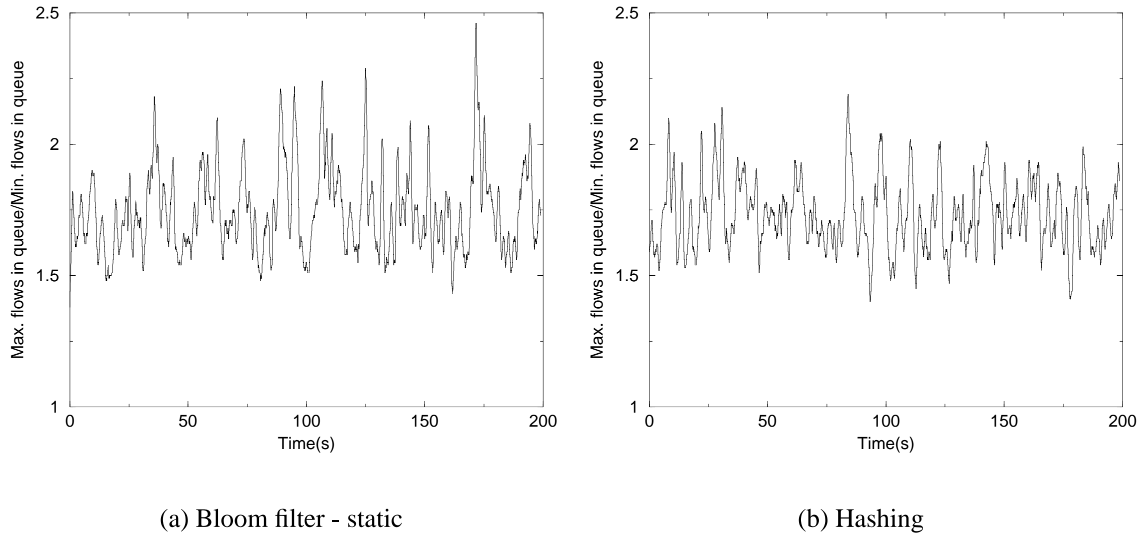
(b) Memory usage for different number of Bloom filters per queue

Figure 5.5: Memory usage for Bloom filter policies compared with per-flow queueing for flow to queue ratios from 1 to 1,000

5.2 Dynamic rebalancing

In the previous section, we showed that by using our Bloom filter architecture and a simple static distribution algorithm, we can achieve near optimal flow distribution for a fixed set of flows. However, in a real router, the set of flows is constantly changing and thus we may need to dynamically rebalance the Bloom filters across queues to maintain a near optimal flow distribution.

To study the effect of changing flows, we ran a simulation with 1,000 flows, 200 Bloom filters and 20 queues. We use a simple hash of the flow's destination address to assign it to a Bloom filter. To simulate flows leaving and arriving, each flow changes its destination address (independently) after an exponentially distributed time with an average of 2 seconds. Thus, the average time between changes is 2 ms. The data is collected from a simulation run of 200 seconds. Figure 5.6(a) shows the time history of the max/min queue ratios using a static assignment of Bloom filters to queues. In this scenario, the Bloom filters are mapped to queues using the distribution algorithm at the start of the simulation and then this mapping is held constant for the remainder of the simulation run. Figure 5.6(b) displays the time history of the max/min queue ratio using simple hashing. From these two graphs, we observe that although the initial mapping of Bloom filters to queues generates a



(a) Bloom filter - static

(b) Hashing

Figure 5.6: Max/min queue ratios for Bloom filter architecture without dynamic rebalancing compared with simple hashing

near optimal distribution of flows across queues, when we have dynamic flows, this static assignment degrades to the same uneven distribution we obtain using a simple hash policy.

Thus, we need a policy to dynamically move Bloom filters between queues in order to maintain an even flow distribution, in response to changing flows. One issue with moving a Bloom filter from one queue to another is the additional overhead of moving already queued packets of the Bloom filter's flows between queues. This overhead can be fairly large depending on the number of packets that need to be moved and hence, we work on minimizing such moves while trying to maintain a near optimal flow distribution. Moving packets is not strictly necessary, since the Internet Protocol (IP) does not require in-order packet delivery. However, since out-of-order delivery has a significant negative impact on end-to-end performance, we take it as a requirement that the load balancing mechanism preserve packet order.

Before we describe the dynamic re-balancing algorithms, we address the issue of how we maintain and update the flow counts associated with each Bloom filter. With new flow arrivals, there is a possibility that we may get a false positive match in a Bloom filter and thus not increment the counter, leading to an underestimate of the flows mapped to that Bloom filter. However, with a careful selection of the number of bits used in a Bloom filter, we can keep the false positive probability low (in the order of 1% or less). A bigger concern is dealing with flows that have departed the system. Bloom filters cannot account for these flows and thus, over time, we could have an overestimate of the number of flows

in every Bloom filter. To help alleviate this problem, we periodically clear all Bloom filters (and flow counts) and wait for a short period for incoming packets to recalibrate our Bloom filters and the flow counts associated with each of them. After we have updated our counts, we also update a *threshold* value, which is the total number of flows divided by the number of queues. The *threshold* reflects an ideal number of flows each queue should handle. We choose a 200 ms period for clearing our Bloom filters and counts and wait for 50 ms after clearing for packets from flows to update the Bloom filters and counters. During this update phase (50 ms), we cannot use the Bloom filter counts to re-balance flows, since the counts are inaccurate. Thus, we need to choose an update phase period that is long enough to enable us to track almost all the flows, but not so long that we do not have a sufficient window between each clearing period to be able to rebalance the flows. We feel that a 50 ms period is a good compromise and should be able to track flows with round-trip times as high as 100-200 ms. Figure 5.7 presents our count updating policy in detail.

```
Every time period T:
  Clear all Bloom filters and counts
  Wait for time δ for flows to update Bloom filters
  For every new 1 set by a packet in a Bloom filter
    increment Bloom filter count by 1
  At time T + δ:
    Update all queue flow counts
    Set threshold <- (total number of flows)/(number of queues)
  Flows counts and threshold values are held constant
    until the next clearing period (2T)
```

Figure 5.7: Policy for updating Bloom filter counts

5.2.1 Periodic Balancing (PB)

Our first approach is to periodically rebalance the flows across queues. In this policy, every time period T, we rebalance flows by moving Bloom filters with no packets queued from queues with flow counts that exceed the *threshold* to queues whose flow counts are currently below the *threshold*. A detailed description of this algorithm in presented in Figure 5.8.

```
Every time period T:
  Wait for time δ for flows to update Bloom filters
    and queue flow counts
  Redistribute flows across queues by redistributing
    Bloom filters across queues
Redistributing policy:
  Sort all N queues in descending order of flow counts
  Let L ≤ N be the number of queues with
    flow count > threshold
  for (i = 1 to L) do
    Let S be the set of Bloom filters mapped to Q_i
      which have no packets queued
    Let Q_min be the queue with the minimum number of flows
    while [(Q_min < threshold) and
      (Q_i > threshold) and (S is not empty)] do
      Move Bloom filter B ∈ S from Q_i to Q_min
      Remove B from S
      Update flow counts for Q_i and Q_min
    endwhile
  endfor
```

Figure 5.8: Algorithm for periodically balancing flows across queues

5.2.2 Balancing at Dequeue (DB)

Since periodically rebalancing flows incurs significant overhead of sorting queues and processing all the queues in order, we investigate another rebalancing policy which seeks to amortize the rebalancing cost. In this algorithm, at every packet dequeue operation, we check to see if the Bloom filter matching this packet has no more packets enqueued and whether the queue's flow count is over the *threshold*. If both these conditions are true, we move this Bloom filter to Q_{min}, which is currently the queue with the minimum number of flows. For this policy, we need maintain only the queue with the minimum number of flows and can save significant overhead of sorting and processing every queue, which is required for the periodic rebalancing algorithm. A detailed description of this policy is presented in Figure 5.9.

```
After dequeue of packet P_{B,Q}:
  where B is the Bloom filter matching P
    Q is the queue for packet P
  if [(number of packets enqueued for B = 0) and
    (Q > threshold)] then
    Move B to Q_{min}
      where Q_{min} is the queue with minimum number of flows
```

Figure 5.9: Algorithm for balancing flows during dequeue

5.3 Results

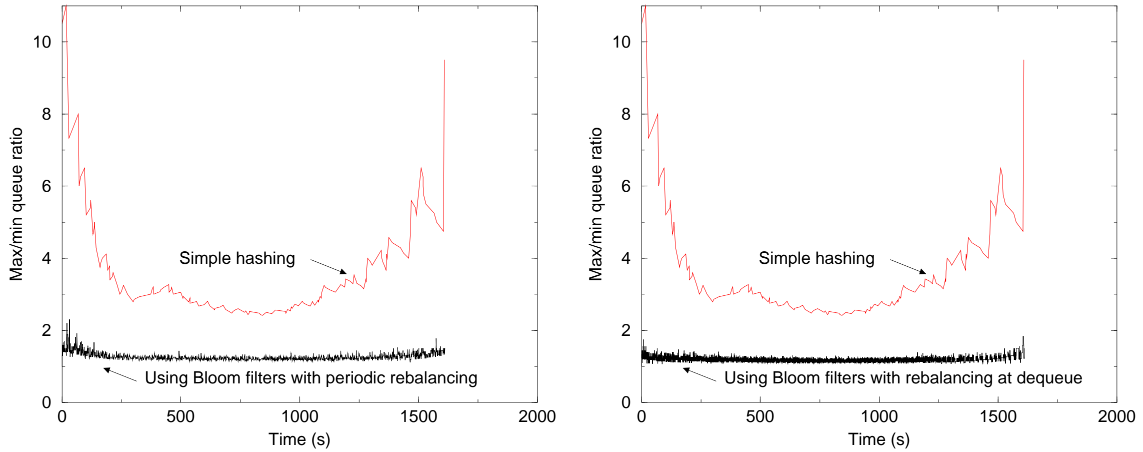
To evaluate our flow distribution algorithms, we ran experiments for two different scenarios. In the first set of experiments, we varied the number of sources from 2,000 sources to 10,000 sources and then back down to 2,000 sources. This scenario studies the effectiveness of our algorithms under changing traffic conditions. Since the flows increase and then decrease, we can evaluate our algorithms for both an increase in network traffic and a reduction in network traffic.

In the second set of experiments, we keep the total number of flows fixed, but flows pick a new (randomly generated) destination address after an exponential holding time. This scenario studies the effectiveness of our algorithms under a constant network load, but with flows arriving and leaving the system.

We use the ratio of the maximum number of flows in a queue to the minimum number of flows in a queue as the metric for comparison. From now on we refer to this ratio as the **max/min queue ratio**. A ratio of 1 indicates an ideal distribution of flows across queues and thus, we would like our algorithms to maintain this ratio very close to 1.

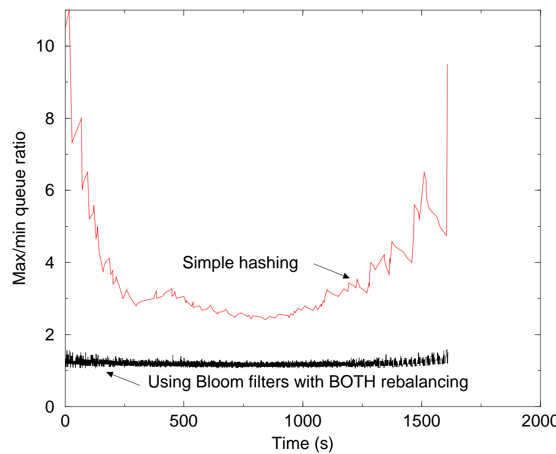
In each of the graphs, we show the time history of the max/min queue ratio using our algorithms compared to the one obtained using simple hashing. We note that although what we plot is the current *actual* ratio traced in our simulation, our algorithms do not use this information. They rely on the approximate counts obtained from the Bloom filters. Also, the graphs labeled “Both” are obtained by using both the PB and DB algorithms to rebalance flows. For simple hashing, the hash function is a uniform random number generator in the range [0 to N-1] (where N is the number of queues), with the destination address as the seed.

5.3.1 Flows arriving and leaving the system



(a) Periodic, 10 flows/s

(b) At dequeue, 10 flows/s

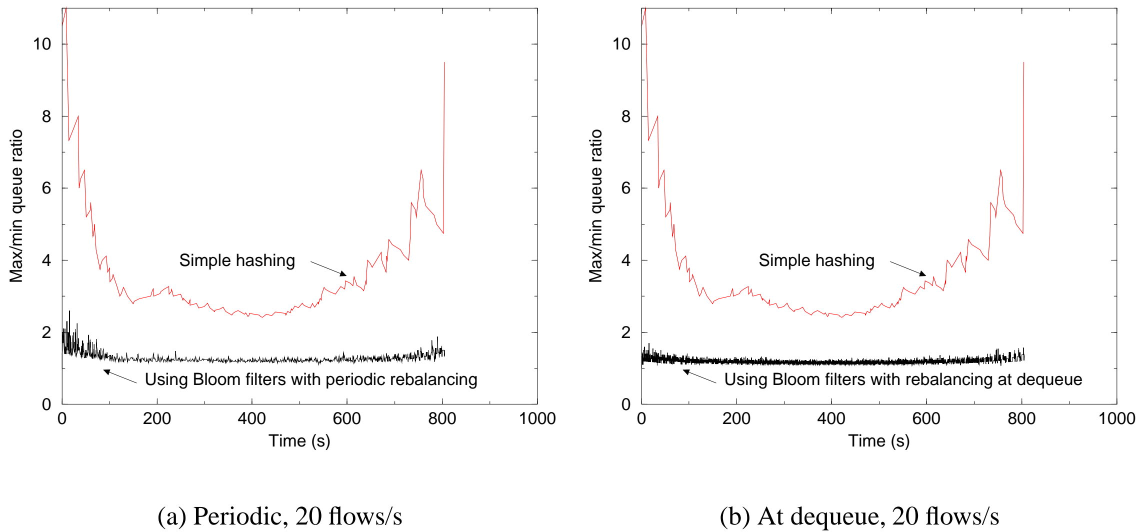


(c) Both, 10 flows/s

Figure 5.10: Flows arriving and departing at a rate of 10 flows/s

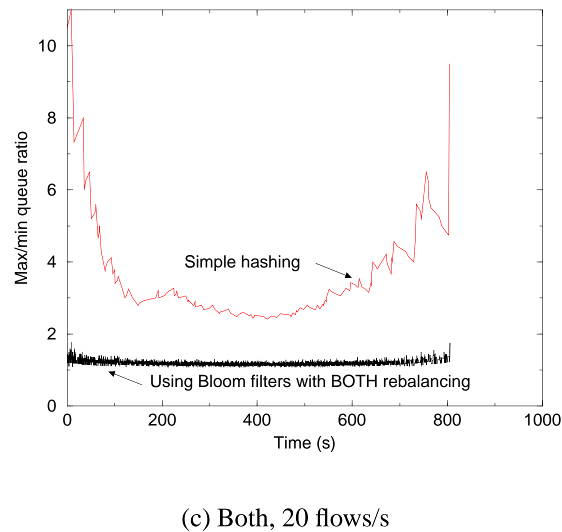
For the first set of experiments, we study our algorithms under varying traffic conditions, with the number of flows increasing from 2,000 to 10,000 and then decreasing back down to 2,000. We use Poisson traffic sources with a mean rate of 1 Mb/s and packet size of 1,500 bytes. The flow arrival/departure rate is exponentially distributed with the mean rate ranging from 10 flows/s to 100 flows/s. We use 2,000 Bloom filters and 200 queues, thus achieving a 50:1 reduction of queues for 10,000 flows. The link rate is set to 1.2 Gb/s and we use QSDRR as the packet scheduling and discard algorithm with a buffer size of

4,000 packets. Figures 5.10, 5.11, 5.12 and 5.13 compare the performance of our algorithms against simple hashing for flow arrival/departure rates of 10 flows/s, 20 flows/s, 50 flows/s and 100 flows/s respectively.



(a) Periodic, 20 flows/s

(b) At dequeue, 20 flows/s

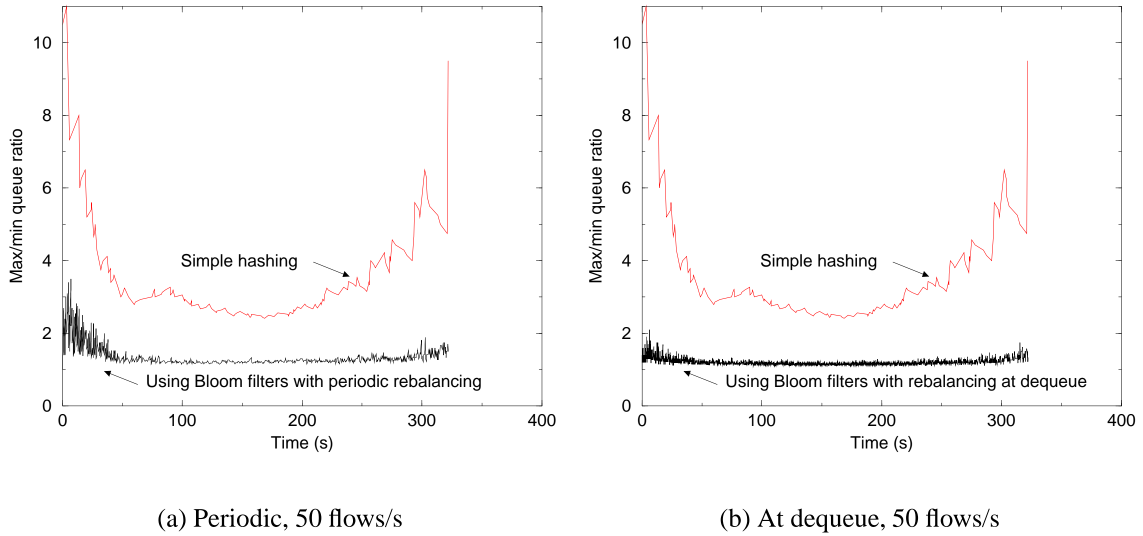


(c) Both, 20 flows/s

Figure 5.11: Flows arriving and departing at a rate of 20 flows/s

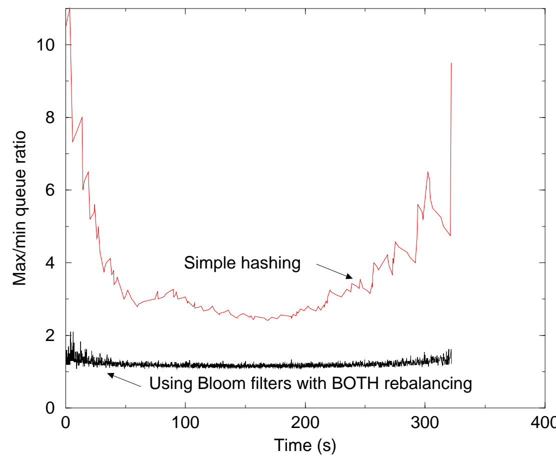
From Figures 5.10 and 5.11, we observe that for flow arrival/departure rates of 10 and 20 flows/s, both the periodic (PB) and the dequeue (DB) algorithms perform very well and maintain a max/min queue ratio very close to 1. They also significantly outperform simple hashing. The performance of our algorithms is an order of magnitude better (10 times lower ratio) than simple hashing for smaller number of flows (2,000 to 4,000 flows).

For higher number of flows (10,000 flows), our algorithms maintain a ratio that is three times lower than simple hashing.



(a) Periodic, 50 flows/s

(b) At dequeue, 50 flows/s

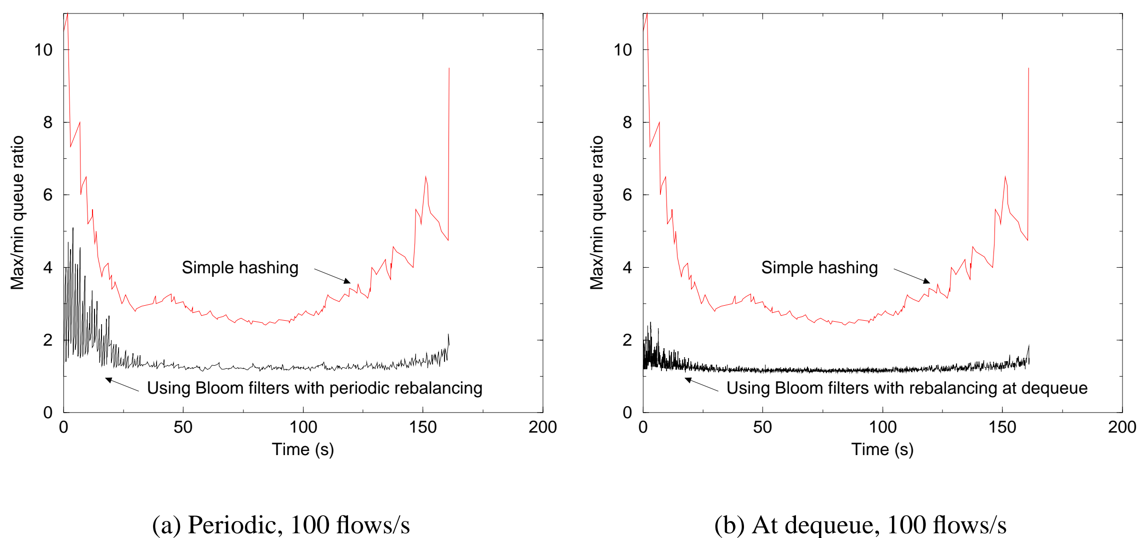


(c) Both, 50 flows/s

Figure 5.12: Flows arriving and departing at a rate of 50 flows/s

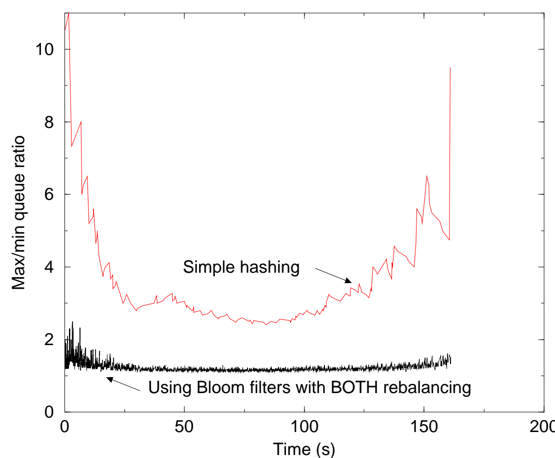
Figures 5.12 and 5.13 compare the performance of our algorithms for flow arrival/departure rates of 50 and 100 flows/s. We observe that for these higher rates, there is a slight degradation in the performance of the PB algorithm (especially for smaller number of flows), since it only rebalances every 200 ms. Flow arrival/departure rates of 50 and 100 flows/s translate to an average of 10 and 20 flows arriving/departing every 200 ms period. When the total number of flows is small, i.e. in the 2,000 to 4,000 range, the max/min

queue ratio will increase noticeably between rebalancing periods due to arriving/departing flows having a larger impact on the ratio. However, the DB algorithm performs very well even for the higher flow arrival rates. It maintains the max/min queue ratio close to the ideal even when the flow arrival rate is high and the total number of flows are in the 2,000 to 4,000 range. As in the previous case, the DB algorithm maintains a ratio that is 10 times smaller than simple hashing when there are between 2,000 and 4,000 flows and three times smaller than simple hashing when the number of flows is in the 10,000 range.



(a) Periodic, 100 flows/s

(b) At dequeue, 100 flows/s



(c) Both, 100 flows/s

Figure 5.13: Flows arriving and departing at a rate of 100 flows/s

Table 5.3: Max/min flows queue ratios for Bloom architecture with Periodic, Dequeue and both dynamic rebalancing vs. simple hashing for dynamic number of flows

Source Arrival Rate Flows/s	Average max/min queue ratio			
	Periodic	Dequeue	Both	Hash
100	1.98	1.28	1.28	3.72
50	1.55	1.21	1.21	3.72
20	1.34	1.18	1.18	3.72
10	1.28	1.17	1.18	3.72

Table 5.3 shows the average max/min queue ratios using PB, DB and both PB and DB combined compared to simple hashing over the entire simulation run. As observed from the graphs, PB is not able to maintain a very low average for high flow arrival/departure rates (50 to 100 flows/s), but it performs well for the 10 and 20 flows/s arrival rates. DB performs very well for all flow arrival rates, keeping the average max/min queue at 1.28 even for flow rates of 100 flows/s. Another thing to note is that using both approaches does not improve upon the DB algorithm significantly for the average max/min queue ratio. For lower flow arrival rates, DB manages to keep the average max/min queue ratio at 1.17, which is just 17% off the ideal. On the other hand, simple hashing is three times worse, maintaining an average max/min queue ratio at 3.72.

Table 5.4: Max/min flows queue ratios for Bloom architecture with Periodic, Dequeue and both dynamic rebalancing vs. simple hashing for dynamic number of flows

Source Arrival Rate Flows/s	Maximum max/min queue ratio			
	Periodic	Dequeue	Both	Hash
100	5.10	2.50	2.50	11.00
50	3.50	2.10	2.10	11.00
20	2.60	1.70	1.78	11.00
10	2.30	1.83	1.57	11.00

Table 5.4 shows the maximum max/min queue ratios using PB, DB and both PB and DB combined compared to simple hashing over the entire simulation run. Here again, PB's performance degrades slightly for higher flow rates of 50 and 100 flows/s, but it still outperforms simple hashing by a factor of 2. At all rates, DB keeps the maximum max/min

queue ratio below 2.5 and outperforms simple hashing by a factor of 4. Combining both approaches helps in keeping the maximum max/min queue ratio slightly lower.

Table 5.5: Max/min flows queue ratios for Bloom architecture with Periodic, Dequeue and both dynamic rebalancing vs. simple hashing for dynamic number of flows

Source Arrival Rate Flows/s	Minimum max/min queue ratio			
	Periodic	Dequeue	Both	Hash
100	1.00	1.00	1.00	2.41
50	1.00	1.00	1.00	2.41
20	1.00	1.00	1.00	2.41
10	1.00	1.00	1.00	2.41

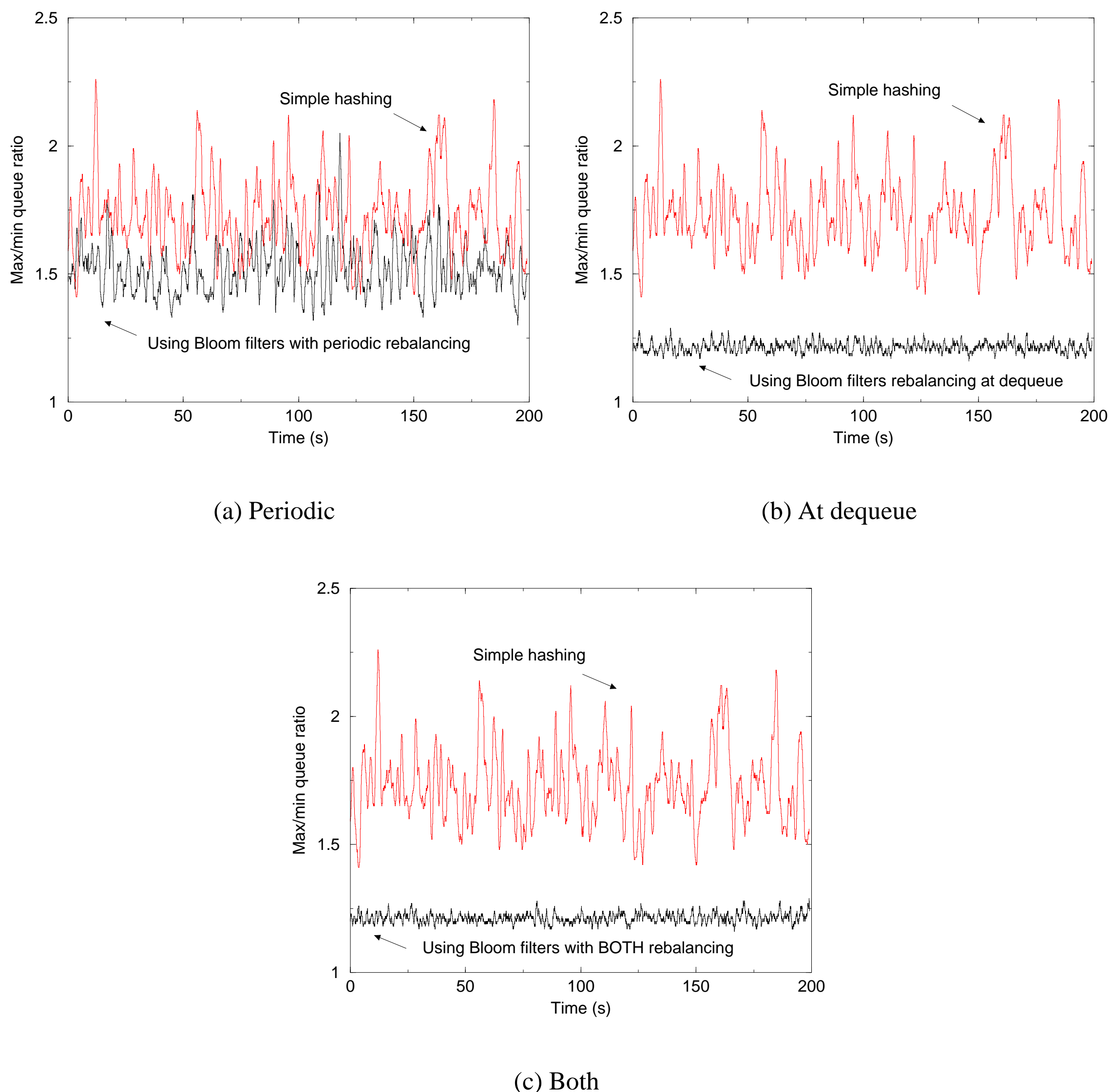
Table 5.5 shows the minimum max/min queue ratios using PB, DB and both PB and DB combined compared to simple hashing over the entire simulation run. All three of our algorithms, PB, DB and combined, are able to achieve optimal flow distribution at one point during the simulation run. However, simple hashing can only achieve a minimum max/min queue ratio of 2.41.

5.3.2 Static number of flows

For this set of experiments, we study our algorithms under constant network load achieved by maintaining the total number of flows constant. To model arrival/departures of flows, each flow changes its destination address after an exponential holding time. This traffic scenario is close to a realistic **worst-case** scenario for our algorithms. Since the total number of flows stays constant, the Bloom filter counts will deviate appreciably from the actual count due to flow departures from some Bloom filters corresponding to flow arrivals in other Bloom filters. For example, say a Bloom filter loses one flow and gains two flows. The Bloom filter count will be incremented by two (it cannot detect flow departures) and thus start deviating from an accurate count.

To evaluate our dynamic rebalancing policies, PB and DB, we ran a packet level simulation with 1,000 Poisson traffic sources, 200 Bloom filters and 20 queues. Each traffic source had a mean rate of 1 Mbps and a packet size of 1500 bytes. We periodically cleared all Bloom filters every 200 ms and waited 50 ms to allow incoming packets to update the filter counts again. We do not perform any rebalancing of filters during this 50 ms interval. We use QSDRR as our packet scheduler with a total buffer size of 4,000 packets. The link

bandwidth was set to 1.2 Gb/s which translated to an 85% load and the results shown were collected over a 200 second simulation interval.



(a) Periodic

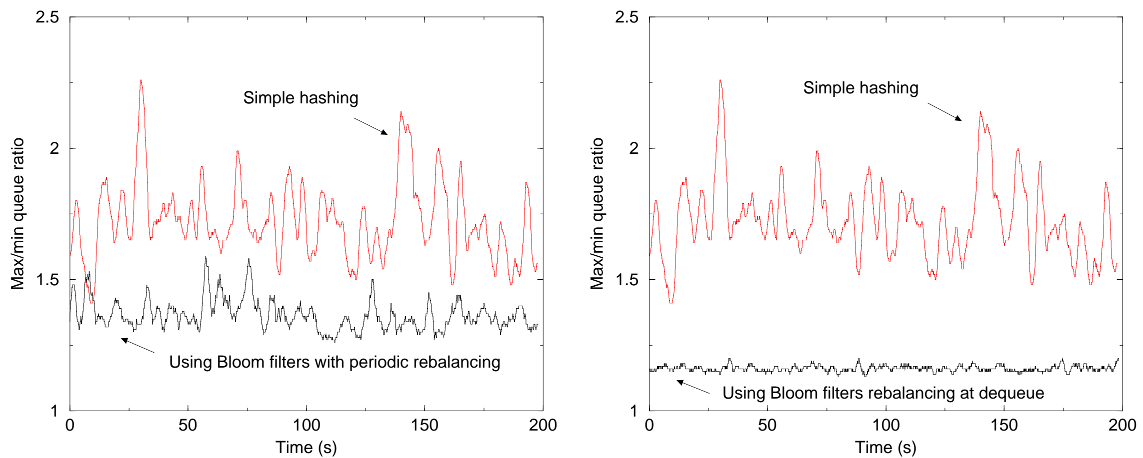
(b) At dequeue

(c) Both

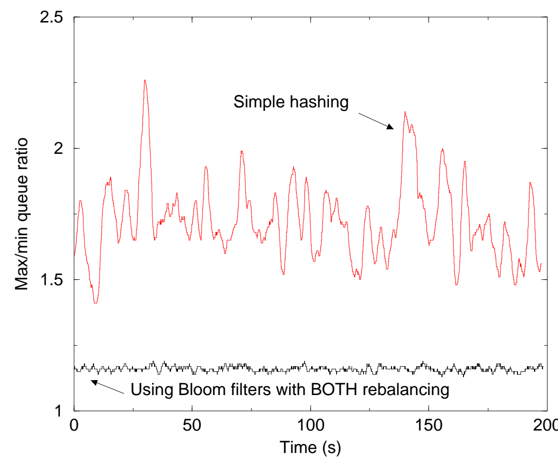
Figure 5.14: Mean destination hold time = 2s

Figure 5.14 shows a time history of the max/min queue ratio for the PB and DB algorithms compared to simple hashing for exponential address holding time with mean of 2 seconds. Given a 1,000 flows, an exponential holding time of 2 seconds translates to a destination change every 2 ms on average. Since we periodically clear and update our Bloom filters and counts every 200 ms, the PB algorithm faces an average of 100 destination changes between balancing intervals. Thus, its performance is only slightly

better than simple hashing when the mean destination hold time is 2 seconds. However, the DB algorithm performs significantly better than simple hashing, keeping the max/min ratio below 1.25 for the entire simulation run compared to simple hashing which maintains a max/min queue ratio above 1.7 on the average.



(a) Periodic (b) At dequeue



(c) Both

Figure 5.15: Mean destination hold time = 5s

Figure 5.15 shows a time history of the max/min queue ratio for the PB and DB algorithms compared to simple hashing for exponential address holding time with mean of 5 seconds. For the mean holding time of 5 seconds, both PB and DB significantly outperform simple hashing. PB keeps the max/min queue ratio below 1.5, while DB manages to

maintain the max/min queue ratio below 1.2 compared to simple hashing which maintains an average max/min queue ratio over 1.7

Table 5.6: Max/min flows queue ratios for Bloom architecture with Periodic, Dequeue and both dynamic rebalancing vs. simple hashing for static number of flows

Mean Address Hold Time (s)	Average max/min queue ratio			
	Periodic	Dequeue	Both	Hash
0.5	1.78	1.40	1.39	1.71
1.0	1.65	1.29	1.29	1.73
2.0	1.52	1.22	1.21	1.73
5.0	1.37	1.16	1.16	1.72

Table 5.6 shows the average max/min queue ratios using PB, DB and both PB and DB combined compared to simple hashing over the entire simulation run of 200 seconds. For lower mean address hold times of 0.5 and 1 seconds, PB's performance degrades to being almost comparable to simple hashing. However, at higher mean address hold times of 2 and 5 seconds, it performs 15% and 30% better than simple hashing. The DB algorithm performs significantly better for all mean address hold times, performing 25% better for mean address hold time of 0.5 seconds and 50% better for mean address hold time of 5 seconds. Using the combined approach marginally improves the performance for all mean address hold times.

Table 5.7: Max/min flows queue ratios for Bloom architecture with Periodic, Dequeue and both dynamic rebalancing vs. simple hashing for static number of flows

Mean Address Hold Time (s)	Maximum max/min queue ratio			
	Periodic	Dequeue	Both	Hash
0.5	3.16	2.38	2.21	2.78
1.0	2.79	1.85	2.06	2.78
2.0	2.56	1.57	1.62	2.71
5.0	1.91	1.43	1.39	2.71

Table 5.7 shows the maximum max/min queue ratios using PB, DB and both PB and DB combined compared to simple hashing over the entire simulation run of 200 seconds. The PB algorithm is again comparable to simple hashing, but the DB algorithm manages

to keep the maximum max/min queue ratio significantly lower than simple hashing. Using the combined approach helps in reducing the maximum max/min queue ratio, but does not result in a substantial improvement.

Table 5.8: Max/min flows queue ratios for Bloom architecture with Periodic, Dequeue and both dynamic rebalancing vs. simple hashing for static number of flows

Mean Address Hold Time (s)	Minimum max/min queue ratio			
	Periodic	Dequeue	Both	Hash
0.5	1.04	1.04	1.04	1.23
1.0	1.04	1.04	1.04	1.23
2.0	1.04	1.04	1.04	1.27
5.0	1.04	1.04	1.04	1.27

Table 5.8 shows the minimum max/min queue ratios using PB, DB and both PB and DB combined compared to simple hashing over the entire simulation run. All three of our algorithms, PB, DB and combined, are able to achieve a near optimal flow distribution of 1.04 at one point during the simulation run. However, simple hashing can only achieve a minimum max/min queue ratio of 1.23 and 1.27.

From our simulation results, we observe that just by using the DB algorithm, we can achieve and maintain near optimal max/min queue ratios for both the dynamically changing traffic model and the static traffic model. Although combining the PB and DB approaches does help in slightly lowering the maximum and average max/min queue ratios, the overhead in performing periodic rebalancing is significantly higher.

Chapter 6

Packet Discard on Arrival

In Chapter 3, we proposed and evaluated two different packet dropping algorithms: Throughput DRR (TDRR) and Queue State DRR (QSDRR). We found that these algorithms significantly outperform RED, Blue and Tail-Drop for both long-lived and short burst TCP traffic. They also perform reasonably well when multiple flows share a single queue. However, both of these approaches need the queues to be ordered by throughput or length. Also, policies that drop packets that have already been queued can require significantly more memory bandwidth than policies that drop packets on arrival. In high performance systems, memory bandwidth can become a key limiting factor. Thus, in this chapter, we investigate buffer management algorithms that can *intelligently* drop incoming packets during congestion without maintaining an ordered list of queues. Using ns-2 simulations, we show that they deliver significant performance improvements over the existing methods. We also show that the results obtained are comparable to what we can achieve using QSDRR, without wasting memory bandwidth and the need to sort queues based on their length.

6.1 Memory Bandwidth Issues

Buffer management policies such as QSDRR and TDRR have some drawbacks for hardware implementation. Two significant issues that affect hardware performance are:

1. **Memory bandwidth wastage**
   When buffers are full, QSDRR drops a packet from the current *drop* queue (the method for choosing the *drop* queue is elaborated in [41]). Similarly, TDRR picks the queue with the current highest exponentially weighted throughput. In most cases, this will lead to a packet already in memory being chosen to be dropped. This leads to

higher memory bandwidth requirements, since the bandwidth used to write packets that are later dropped is wasted.

2. **Queue length sorting**
   All the previously studied DRR algorithms in [41] need to find the *longest queue* (the definition of the *longest queue* varies according to the packet dropping policy) for discarding a packet during congestion. This results in a large overhead during congestion, since each incoming packet would potentially trigger a new search for the current longest queue. One way to reduce this overhead is to use more complex data structures which reduce the time to find the longest queue. However, this adds complexity and cost to any hardware implementation.

6.2 Algorithms

Given the above issues regarding implementation of packet drop policies such as DRR, TDRR and QSDRR, we propose a new packet drop policy based on a dynamic threshold. The original idea for this algorithm is presented in [13]. In [13], the authors propose a memory bandwidth efficient buffer sharing policy among different output ports in a shared memory packet switch. This algorithm makes packet drop decisions based only on the length of the incoming packet's destination queue and the total amount of free buffer space. An incoming packet, destined for queue i is discarded if

$$Q_i(t) \geq \alpha \times F(t) \tag{6.1}$$

where $Q_i(t)$ is the current length of queue i, $F(t)$ is the current free buffer space and α is a multiplicative parameter.

1. **Dynamic Threshold DRR (DTDRR)**
   In our first policy, we adapted the above buffer management policy for use as a packet discard policy for DRR packet scheduling. Thus, an incoming packet destined for queue i is dropped if the current queue length exceeds α times the free buffer space. In all our simulation results, we set α to 2 for evaluating this policy. Although this algorithm performed very well for short burst TCP flows and reasonably sized buffers (1000 packets or more), we found that it did not perform as well as QSDRR for long-lived TCP traffic and very small buffers (200 to 400 packets).

2. **Discard State DRR (DSDRR)**

```
$W$ <- 10% of number of queues
$W_{max}$ <- 50% of number of queues

$Enqueue$:
Discard packet destined for queue i
  if any of the following conditions is true
    1.  $Q_i(t)$ is marked for discard
    2.  $Q_i(t) \geq \alpha \times F(t)$ and
        (number of queues with discard bit set < $W$)
         Then mark $Q_i(t)$ for discard
    3.  $F(t) = 0$
         Then set overflow bit
Else
  Enqueue packet

$Dequeue$:
If $Q_i(t)$ becomes empty, discard bit is cleared

Every time period $T$
If overflow bit is set
  If $W$ < $W_{max}$
    $W$ <- $W + 2$
Else
  If number of queues in discard < $W$
    $W$ <- number of discard queues + 1
```

Figure 6.1: Algorithm for DSDRR

Taking a cue from QSDRR, we added some hysteresis to the basic DTDRR policy which leads to DSDRR. The idea is similar to QSDRR. In DSDRR, once we start discarding from a particular queue, we mark it with a discard bit. Subsequent packets destined for a queue marked with a discard bit are discarded regardless of the queue length. The discard bit is cleared when the queue becomes empty. We found that, although this policy helped in desynchronizing the TCP flows, it marked too many queues for discard and thus suffered from poor throughput. To alleviate this problem, we added another parameter, W. This is an adaptive parameter that limits the number of queues marked for discard. Every time period T, if the buffer overflows, W is increased by 2. If there is no overflow in the last time period and the number

of queues marked for discard is less than W, W is set to one more than the current number of discard queues. Thus, when a packet arrives for an unmarked queue and the queue exceeds the threshold as described in equation 6.1, it is marked for discard only if the total number of discard queues is less than W. In addition, a queue is marked for discard if there is no free space for the arriving packet. Incoming packets are dropped if the queue is marked for discard. We found that the policy is not sensitive to the initial value of W and we initially set W to 10% of the number of queues (flows) for all our simulation experiments and we limit W to a maximum value of 50% of the number of queues. Also, α is set to 0.1 and T is set to 1 second for our simulation runs. A detailed description of this algorithm is presented in Figure 6.1.

6.3 Simulation Environment

In order to evaluate the performance of DRR, TDRR and QSDRR, we ran a number of experiments using ns-2. In this paper, we investigate the performance of our algorithms for both long-lived and short-lived TCP connections. Long-lived TCP flows stay active for the entire duration of the simulation. We emulate short-lived TCP flows using on-off TCP sources. The *on-phase* models an active TCP flow sending data, while the *off-phase* models the inter-arrival time between connections. To effectively compare the times taken to service each burst under different algorithms, we fix the data transferred per connection (during the *on-phase*) to 256 packets (384 KB). The idle time between bursts is exponentially distributed with a mean of 2 seconds.

We compared the performance over a varied set of network configurations and traffic mixes which are described below. In all our experiments, we used TCP sources with 1500 byte packets and the data collected is over a 100 second simulation interval. We ran experiments using TCP Reno and TCP Tahoe and obtained similar results for both; hence, we only show the results using TCP Reno sources. For each of the configurations, we varied the bottleneck queue size from 100 packets to 20,000 packets. 20,000 packets represents a half-second buffer which is a common buffer size deployed in current commercial routers. We ran several simulations to determine the best parameter values for RED and Blue for our simulation environment, to ensure a fair comparison against our multi-queue based algorithms. In all our configurations below, the access links are 10 Mb/s for long-lived TCP flows and 100 Mb/s for short-lived (on-off) TCP flows. Since the bottleneck-link bandwidth is 500 Mb/s, if all long-lived TCP flows send at the maximum rate, the overload

ratio is 2:1. For the short-lived TCP sources, a maximum rate of 100 Mb/s is needed to congest the bottleneck link.

6.3.1 Single Bottleneck Link

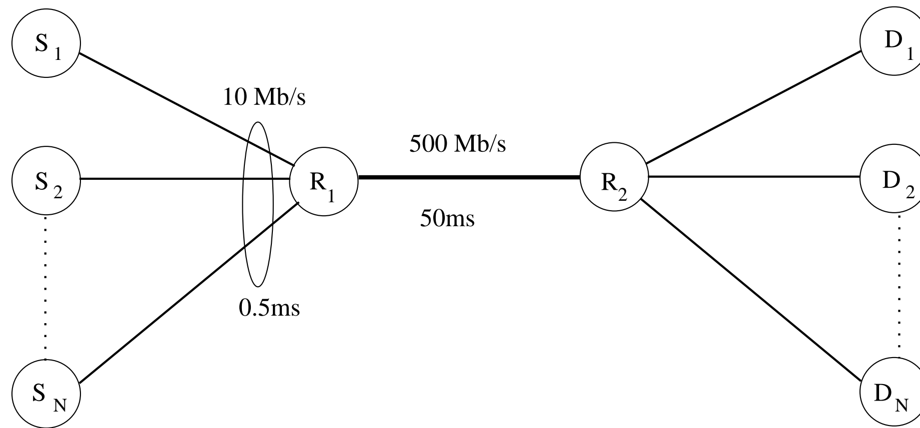


Figure 6.2: Single Bottleneck Link Network Configuration

The network configuration for this set of experiments is shown in Figure 6.2. The TCP sources, $\{S_1, S_2, ...S_N\}$ are connected to the bottleneck link. The destinations, $\{D_1, D_2, ...D_N\}$, are directly connected to the router R_2. N is 100 for long-lived TCP flows and 500 for short-lived TCP flows. All the TCP sources are started simultaneously to simulate a worst-case scenario whereby TCP sources are synchronized in the network. In each of the configurations, the delay shown is the one-way link delay. Thus, round-trip time (RTT) over a link is twice the link delay value.

6.3.2 Multiple Roundtrip-time Configuration

The network configuration for this set of experiments is shown in Figure 6.3. This configuration is used to evaluate the performance of the different queue management policies given two sets of TCP flows with widely varying round-trip times over the same bottleneck link. The source connection setup is similar to the single-bottleneck configuration, except for the access link delays for each source and the total number of sources. Half of the TCP sources have their link delay set to 20 ms, and the other half have their link delay to 100 ms. For this configuration, N is 50 for long-lived flows and 500 for short-lived flows.

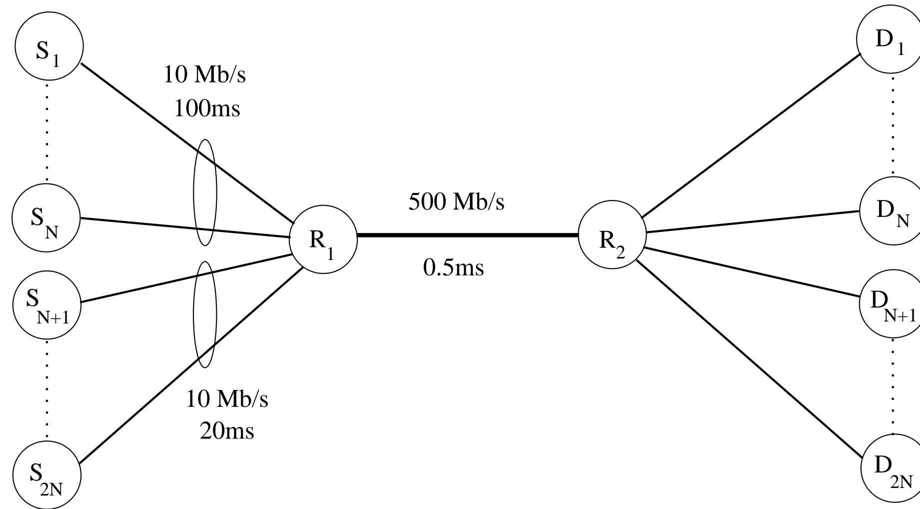


Figure 6.3: Multiple Roundtrip-time Network Configuration

6.3.3 Multi-Hop Path Configuration

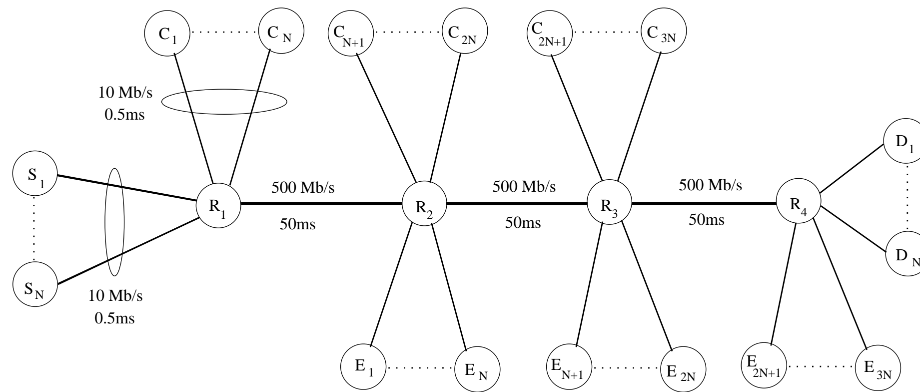


Figure 6.4: Multi-Hop Path Network Configuration

The network configuration for this set of experiments is shown in Figure 6.4. In this configuration, we have N TCP sources traversing three bottleneck links and terminating at R_3. In addition, on each link, there are another N TCP sources acting as cross-traffic. We use this configuration to evaluate the performance of the different queue management policies for multi-hop TCP flows competing with shorter one-hop cross-traffic flows. N is 50 for long-lived flows and 500 for short-lived flows.

6.4 Results

We now present the evaluation of our DTDRR and DSDRR policies in comparison with QSDRR, Blue, RED and Tail-Drop. We compare the queue management policies using the average goodput of all TCP flows as a percentage of its fair-share as the metric. We also show the variance in goodput for a single-bottleneck link under the different policies. The *variance* in goodputs is a metric of the fairness of the algorithm; lower variance implies better fairness. For all our graphs, we concentrate on the goodputs obtained while varying the buffer size from 100 packets to 5000 packets. Since our bottleneck link speed is 500 Mb/s, this translates to a variation of buffer *time* from 2.4 ms to 120 ms. In all our simulations, we noticed that all the policies behaved in a similar fashion past the 5000 packet buffer size.

6.4.1 Single-Bottleneck Link

For this experiment, the single bottleneck link configuration is used. For the long-lived TCP flow case, we use 100 TCP Reno sources, and for the short burst TCP scenario, we use 500 on-off TCP Reno sources.

Long-lived TCP flows

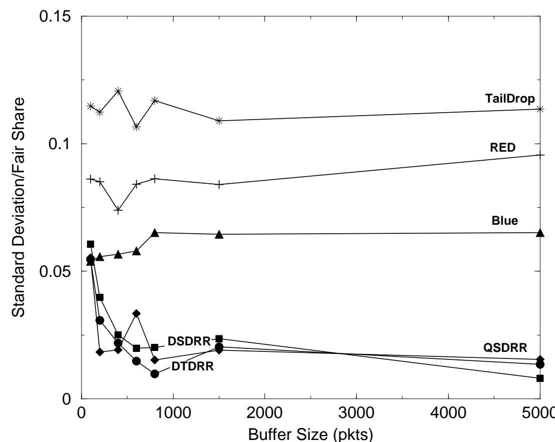


Figure 6.5: Standard deviation relative to fair-share for long-lived TCP Reno flows over a single-bottleneck link

Figure 6.5 shows the ratio of the goodput standard deviation of the TCP Reno flows to the fair share bandwidth for all algorithms while varying the buffer size. Even at higher buffer sizes, the goodput standard deviation under DTDRR and DSDRR is very small and the ratio to the fair share bandwidth is less than 0.025 which is equivalent to the standard

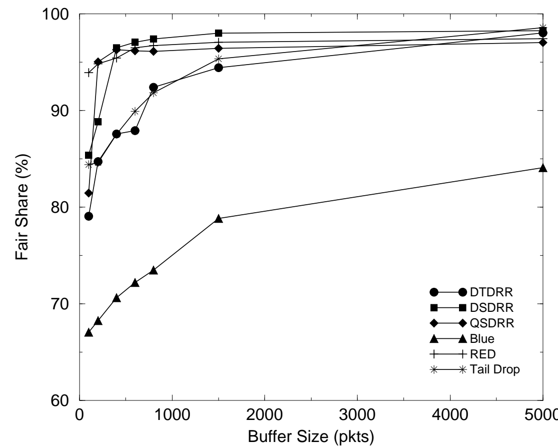
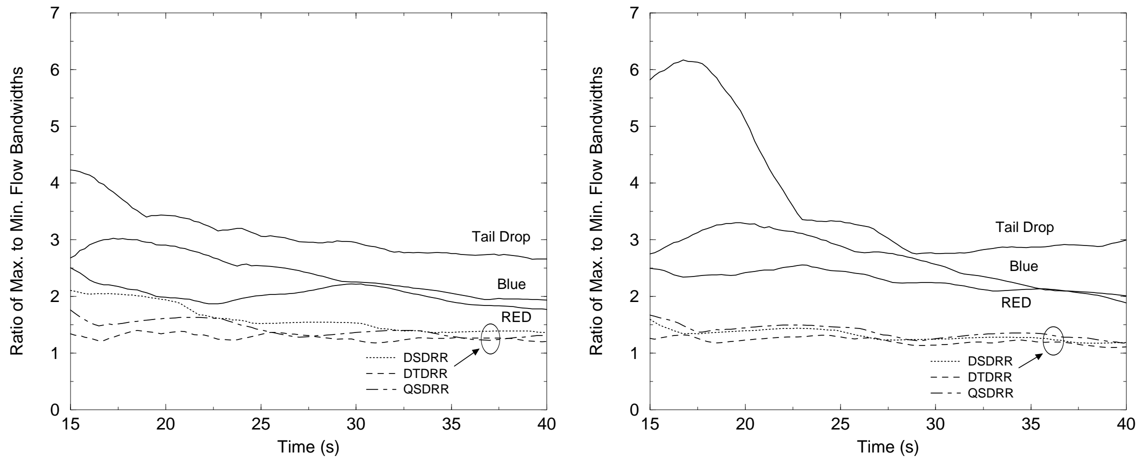


Figure 6.6: Fair share performance for long-lived TCP Reno flows over a single bottleneck link

deviation ratio of QSDRR. RED exhibits about 10 times the variance compared to DSDRR and DTDRR, while Blue exhibits about 5 times the variance. Overall, we observe that the goodput standard deviation is between 2%-4% of the fair share bandwidth for the DSDRR and DTDRR policies compared to 6% for Blue, 10% for RED and 12% for Tail-Drop. Thus, even for a single-bottleneck link, we observe that the DSDRR and DTDRR policies offer much better fairness to a set of TCP flows and are equivalent in fairness to QSDRR.

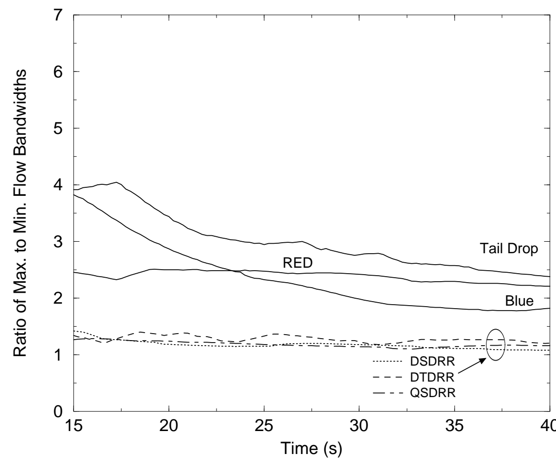
Figure 6.6 illustrates the average fair-share bandwidth percentage received by the TCP Reno flows using different buffer sizes. For this configuration, we notice that the performance under DTDRR is comparable to Tail-Drop for all buffer sizes. However, DSDRR delivers performance which is very close to QSDRR and outperforms RED and Tail-Drop, especially for small buffer sizes, i.e. under 500 packets.It is interesting to note that even at a large buffer size of 5000 packets, all policies significantly outperform Blue, including Tail-Drop.

Figure 6.7 shows the ratios of the maximum to minimum flow throughputs over time for different buffer sizes. These graphs are useful in illustrating the variation in throughputs experienced by the TCP flows under the different scheduling algorithms. For all buffer sizes, the ratio for the DSDRR and DTDRR algorithms quickly converges to 1 which implies that the variance in the TCP flows' throughputs is very small. Also, both of the algorithms are able to match QSDRR's performance. In the case of both RED and Blue, the best they can achieve is a ratio of around 2, which means that the maximum TCP flow receives *twice* the throughput of the minimum flow. As expected, Tail Drop is the worst and allows the maximum TCP flow to receive *thrice* the throughput of the minimum flow.



(a) 400 packet buffer

(b) 800 packet buffer



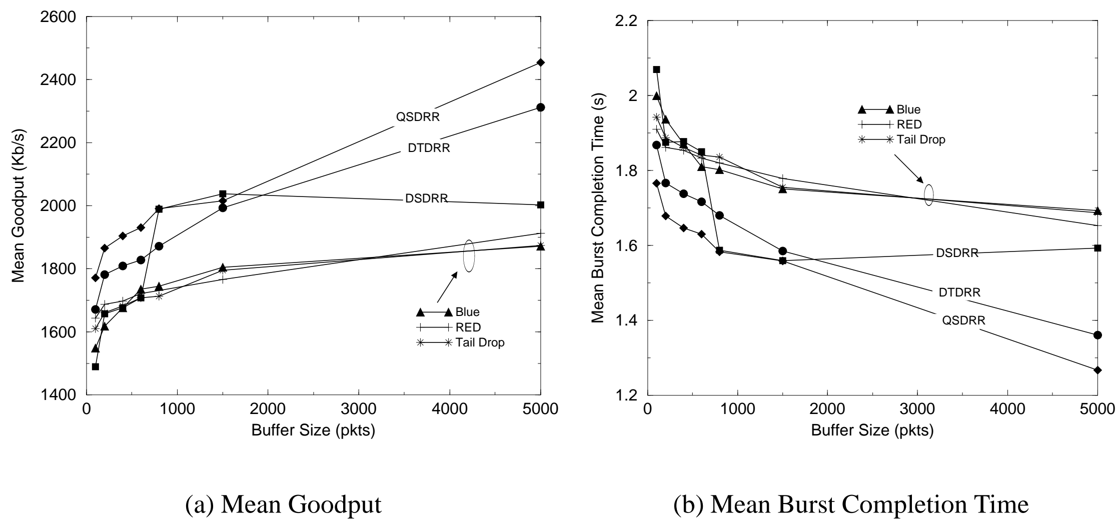
(c) 4167 packet buffer

Figure 6.7: Ratio of maximum to minimum flow throughputs

Short burst TCP flows

Figure 6.8(a) shows the mean goodput achieved by the TCP flows and Figure 6.8(b) shows the mean burst completion times for the flows over a single bottleneck link configuration. *Goodput* is the amount of actual data transmitted excluding retransmissions and duplicates. We notice that Blue, RED and Tail-Drop have almost exactly the same performance in terms of mean goodput achieved and burst completion times for all buffer sizes, whereas the DTDRR and DSDRR policies are uniformly better. For buffer sizes less than 2000 packets, DTDRR and DSDRR exhibit about 10% better goodput performance over Blue,

RED and Tail-Drop. However, it is interesting to note that DTDRR is almost 30% better than the non-DRR policies at a buffer size of 5000 packets and is very close to QSDRR. DSDRR does not perform as well at higher buffer sizes due to its aggressive dropping threshold and keeping queues in discard state. At smaller buffer sizes (2000 packets or less), DSDRR performs very well and almost exactly matches the performance of QSDRR. The results are similar for the burst completion times.



(a) Mean Goodput

(b) Mean Burst Completion Time

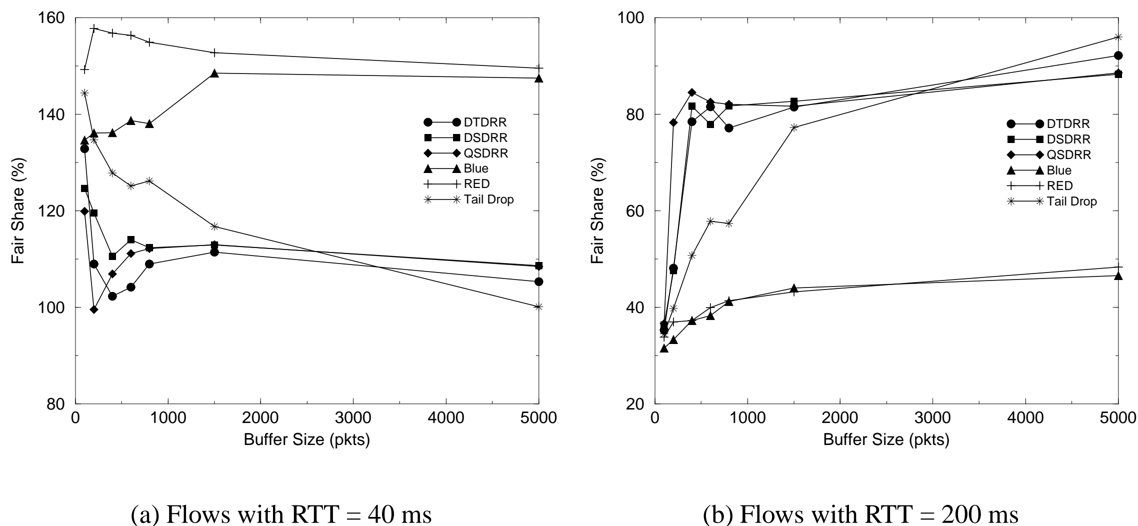
Figure 6.8: Performance of short burst TCP flows over a single bottleneck link

6.4.2 Multiple Round-Trip Time Configuration

In this configuration, we again use a single bottleneck link, but half the TCP sources have a 40 ms RTT whereas the other half have a 200 ms RTT. For long-lived TCP flows, we use 100 TCP Reno sources and for short burst TCP flows, we use 1000 on-off TCP Reno sources.

Long-lived TCP flows

Figure 6.9 shows the average fair-share goodput received by TCP flows using the different algorithms. As shown in Figure 6.9(a), both RED and Blue allow the 40 ms RTT flows to use almost 50% more bandwidth than their fair share. Tail-Drop also allows the 40 ms RTT flows to use more than their fair share of the bandwidth for buffer sizes smaller than 1000 packets. Both the DTDRR and DSDRR policies exhibit much better performance allowing only 10% extra bandwidth to be used by the 40 ms RTT flows. Both RED and Blue discriminate against longer RTT flows, as we observe in Figure 6.9(b), the 200 ms



(a) Flows with RTT = 40 ms

(b) Flows with RTT = 200 ms

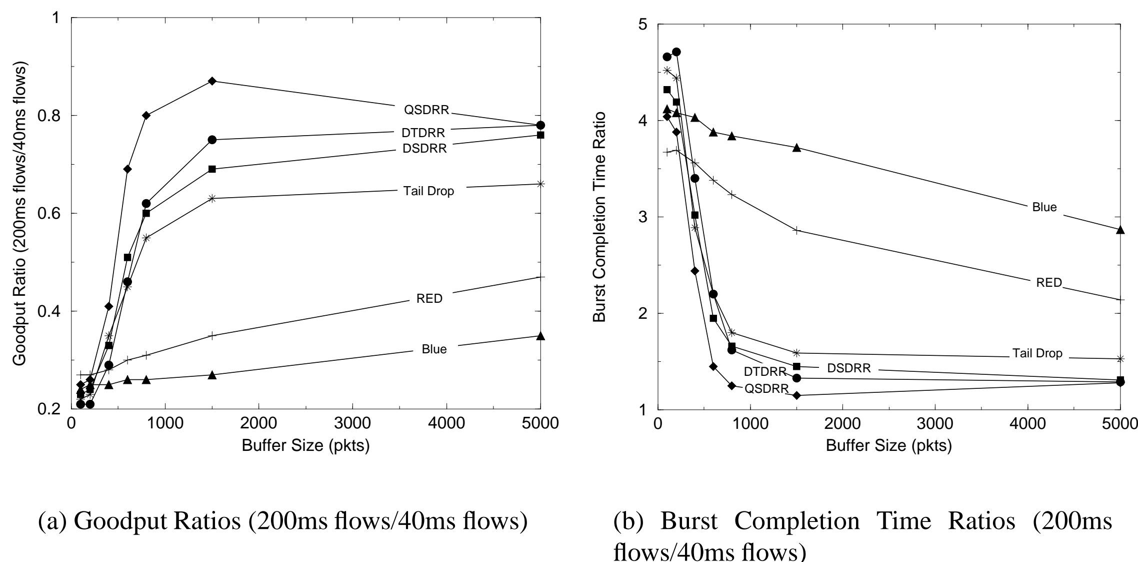
Figure 6.9: Fair share performance of different RTT long-lived TCP flows over a single bottleneck link

RTT flows achieve only about 40% of their fair-share bandwidth whereas using the DTDRR and DSDRR policies, 200 ms RTT flows are able to achieve almost 90% of their fair-share.

At a very small buffer size of 100 packets, 200 ms RTT flows using DTDRR and DSDRR get about 40% of their fair-share. However, at this buffer size, when all the flows are active, there is only one packet per flow that can be buffered. This causes the poor performance of DTDRR and DSDRR, since it becomes very difficult to single out flows that are using more bandwidth. Even with this limitation, when we move to 400 packets, both DTDRR and DSDRR significantly improve their performance and 200 ms RTT flows achieve about 80% of their fair-share bandwidth on the average. Although QSDRR is better at a buffer size of 200 packets, at all buffer sizes greater than that, both DTDRR and DSDRR are able to match the performance of QSDRR.

Short burst TCP flows

Figure 6.10(a) shows the ratios of the goodputs obtained by 200 ms round-trip time flows over the goodputs of the 40 ms round-trip time flows for the multiple RTT configuration. In this configuration, for buffer sizes greater than a 800 packets, DTDRR and DSDRR outperform Blue and RED by more than 100%. Although the performance improvement at smaller buffer sizes is not as dramatic, DTDRR and DSDRR still outperform RED and Blue significantly. The ratio of goodputs is used to illustrate the fairness of each algorithm. The closer the ratio is to one, the better the algorithm is in delivering fair-share to different round-trip time flows. In this case, even Tail-Drop performs significantly better than Blue



(a) Goodput Ratios (200ms flows/40ms flows)

(b) Burst Completion Time Ratios (200ms flows/40ms flows)

Figure 6.10: Performance of short burst TCP flows over a multiple round-trip time configuration

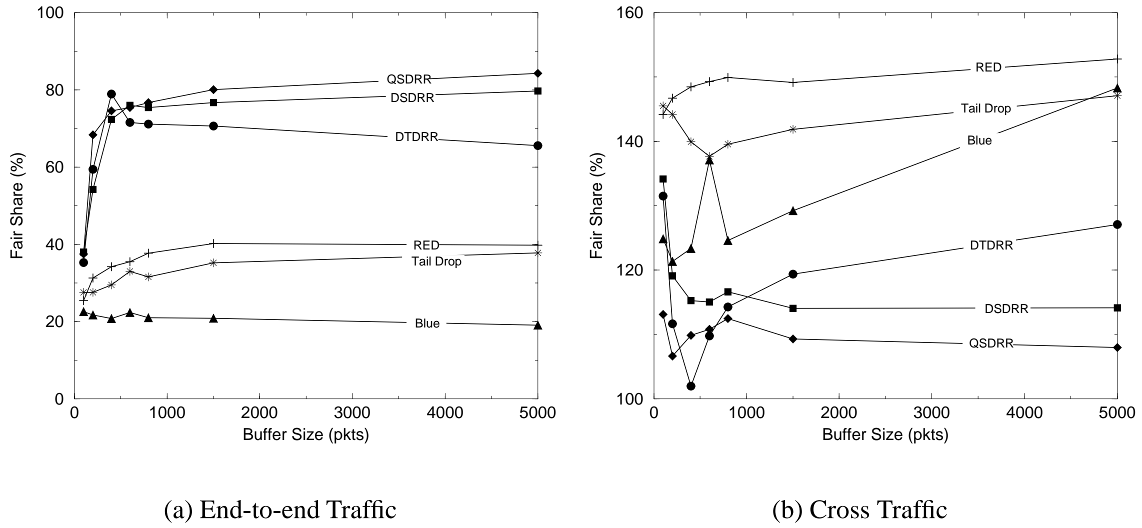
and RED, showing that for short-lived flows with different round-trip times, Blue and RED cannot deliver good fair-sharing of the bottleneck bandwidth. Figure 6.10(b) shows the ratios of burst completion times of the 200 ms round-trip time flows over the 40 ms round-trip time flows. In this case, DTDRR and DSDRR remain close to one for buffer sizes greater than 1000 (which is the ideal fairness), whereas Blue has the worst performance, with the 200 ms round-trip time flows taking almost *three times* the time to complete a burst compared to the 40 ms round-trip time flows, even for 5000 packet buffers. Also, their performance is only 10-20% worse than QSDRR for small buffer sizes. At a buffer size of 5000, DTDRR and DSDRR match the performance of QSDRR.

6.4.3 Multi-Hop Path Configuration

In this configuration, end-to-end TCP Reno flows go over three hops and have an overall round-trip time of 300 ms. The cross-traffic on each hop consists of TCP Reno flows with a round-trip time of 100 ms (one hop). For long-lived TCP flows, we use 50 end-to-end and 50 cross-traffic TCP Reno sources on each link and for short burst TCP flows, we use 500 end-to-end and 500 cross-traffic on-off TCP Reno sources on each link.

Long-lived TCP flows

Figure 6.11 illustrates the average fair-share goodput received by each set of flows. For this configuration, DTDRR and DSDRR provide almost *twice* the goodput of RED and



(a) End-to-end Traffic

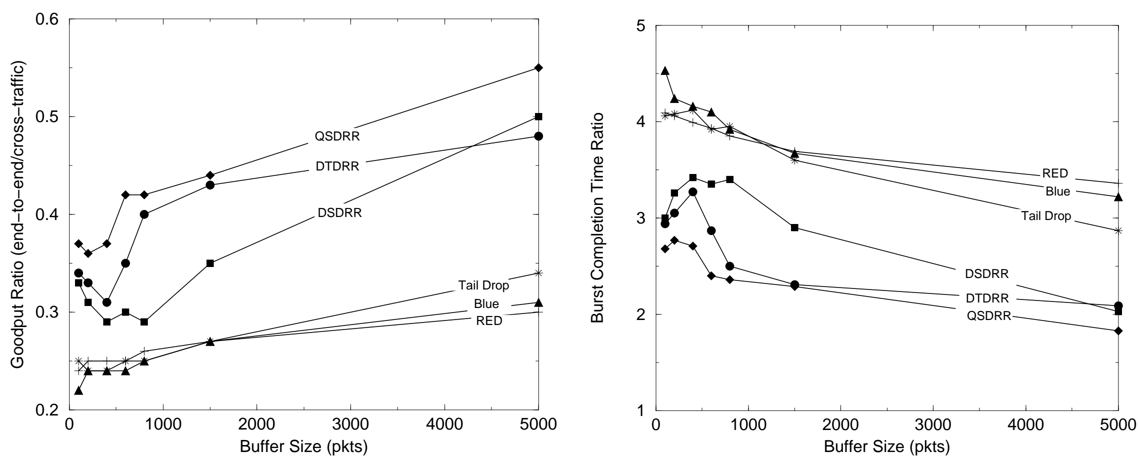
(b) Cross Traffic

Figure 6.11: Fair Share performance of long-lived TCP flows over a multi-hop path configuration

Tail Drop and *four* times the goodput provided by Blue for end-to-end flows. As shown in Figure 6.11(a), end-to-end flows achieve nearly 80% of their fair-share under DSDRR and 70% under DTDRR. Under RED and Tail Drop, they can achieve only 40% of their fair share even at a buffer size of 5000 packets. Using DTDRR and DSDRR, even for the smallest buffer size, their fair-share is better than RED, but once the buffer size increases to 400 packets, their performance improves significantly and they allow the end-to-end flows to achieve close to 80% of their fair share. We notice that in this configuration, DSDRR's performance is very close to QSDRR. Although DTDRR's performance is slightly worse than DSDRR and QSDRR (about 10%) for buffer sizes greater than a 1000 packets, it is still *1.5* times the performance provided by RED.

For this multi-hop configuration, the end-to-end flows face a probability of packet loss at each hop under RED and Blue. Due to congestion caused by the cross-traffic, RED and Blue will randomly drop packets at each hop. Although the cross-traffic flows will have a greater probability of being picked for a drop, the end-to-end flows also experience random dropping and thus achieve very poor goodput. For Blue, this is further exacerbated, since due to the high load from the cross-traffic flows, the discard probability remains high at each hop. This increases the probability of an end-to-end flow facing packet drops at each hop and thus further reducing the goodput.

Figure 6.11(b) shows the average goodput for the cross-traffic flows attached to router R_1. For DTDRR and DSDRR, the cross-traffic takes up the slack in the link and



(a) Goodput Ratios (end-to-end flows/cross-traffic flows)

(b) Burst Completion Time Ratios (end-to-end flows/cross-traffic flows)

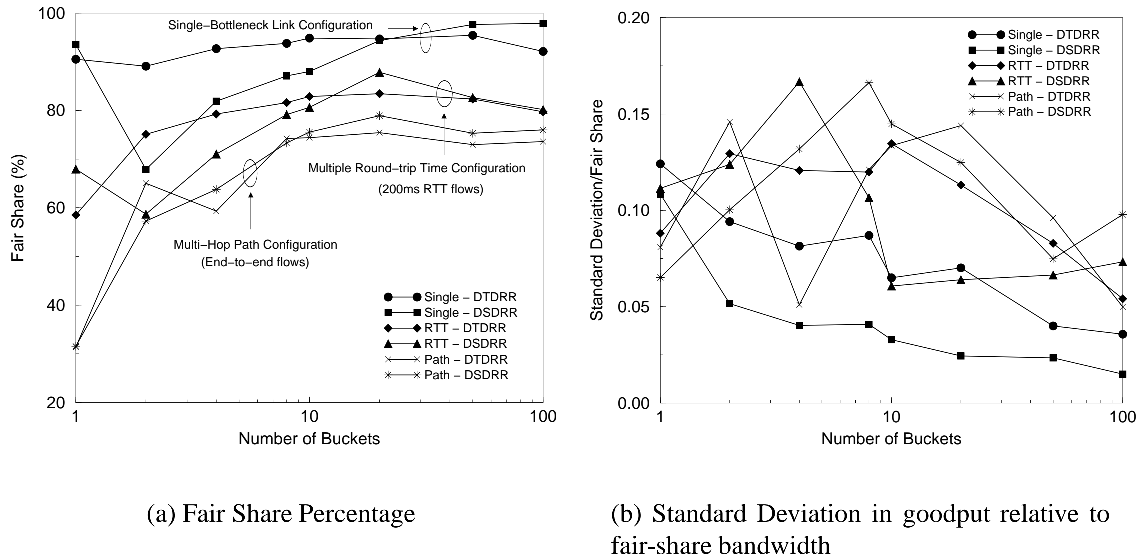
Figure 6.12: Performance of short burst TCP flows over a multi-hop path configuration

consumes about 115-120% of its fair-share bandwidth. For both RED and Tail Drop, the link utilization is lower and although the end-to-end flows consume only about 40% of their fair-share, the cross-traffic flows consume 150% of their fair-share and thus leave about 5% unutilized. Cross-traffic flows under Blue consume about 120-140% of their fair-share, leaving 20-30% unutilized.

Short burst TCP flows

Figure 6.12(a) shows the ratios of the goodputs achieved by the end-to-end flows over the cross-traffic flows for the multi-hop path configuration. In this configuration, we see that the non-DRR policies perform very poorly, allowing the end-to-end flows a mere 30% of the goodput achieved by the cross-traffic flows. On the other hand, DTDRR and DS-DRR outperform the non-DRR policies by 20-30% for buffer sizes less than 600 packets. For buffer sizes between 600 and 5000 packets, DTDRR outperforms non-DRR policies by about 50% and closely matches the performance of QSDRR. We notice that DSDRR underperforms DTDRR and QSDRR for buffer sizes below 5000 packets, but still outperforms non-DRR policies by 20-50%. DTDRR and DSDRR are almost *2 times* better than the non-DRR policies for a buffer size of 5000 packets.

Figure 6.12(b) shows the ratios of burst completion times of the end-to-end flows over the cross-traffic flows. DTDRR performs almost as well as QSDRR and beats the non-DRR policies by at least a factor of two. DSDRR also performs reasonably well achieving burst completion time ratios of about a factor of 1.5 better than the non-DRR policies.



(a) Fair Share Percentage

(b) Standard Deviation in goodput relative to fair-share bandwidth

Figure 6.13: Performance of DTDRR and DSDRR for a buffer size of 1000 packets, with varying number of buckets

Even though the end-to-end traffic flows over three bottleneck links compared to just one bottleneck-link for the cross-traffic flows, DTDRR and DSDRR are able to achieve a burst completion time ratio near two for a buffer size of 5000 packets. At the same buffer size, the non-DRR policies achieve fairly poor ratios ranging from 3.5 to 4.0.

Overall, we notice that DTDRR matches the performance of QSDRR for short burst TCP traffic while DSDRR matches the performance of QSDRR for long-lived TCP traffic. Although, DSDRR is not as good as DTDRR for short burst TCP flows, it still significantly outperforms RED, Blue and Tail-Drop for all configurations and traffic mixes.

6.4.4 Scalability Issues

One drawback with a fair-queueing policy such as DTDRR or DSDRR is that we need to maintain a separate queue for each active flow. Since each queue requires a certain amount of memory for the linked list header, used to implement the queue, there is a limit on the number of queues that a router can support. In the worst-case, there might be as many as one queue for every packet stored. Since list headers are generally much smaller than the packets themselves, the severity of the memory impact of multiple queues is intrinsically limited. On the other hand, since list headers are typically stored in more expensive SRAM, while the packets are stored in DRAM, there is some legitimate concern about the cost associated with using large numbers of queues. One way to reduce the impact of this issue

is to allow multiple flows to share a single queue. While this can reduce the performance benefits observed in the previous sections, it may be appropriate to trade off performance against cost, at least to some extent. To address this issue, we ran several simulations evaluating the effects of merging multiple flows into a single queue. Figure 6.13 illustrates the effects of varying the number of queues. The sources are long-lived TCP Reno flows and the total buffer space is fixed at 1000 packets.

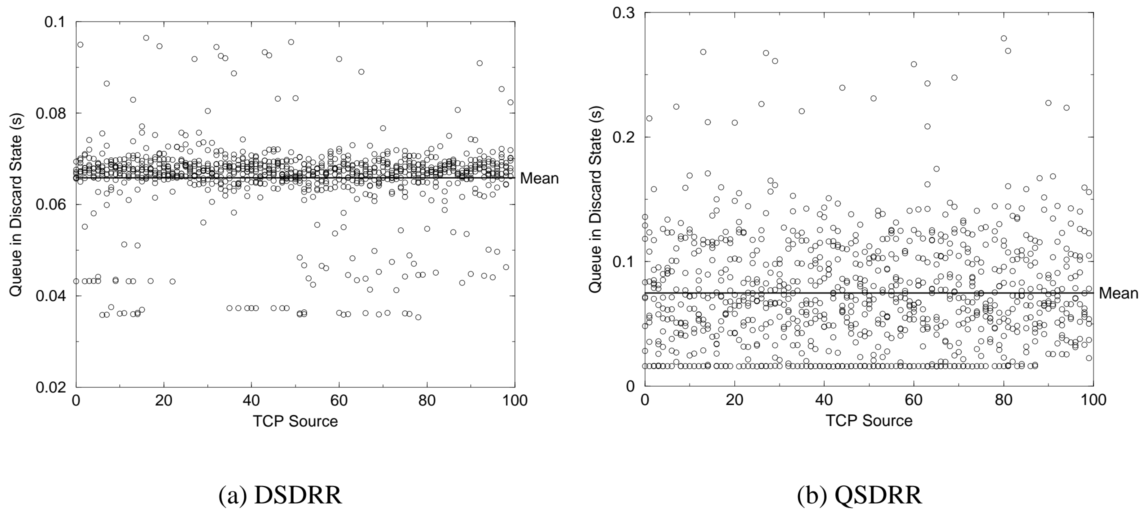
Figure 6.13(a) illustrates the effect on the goodput received by each flow under different numbers of queues. For the multiple round-trip time configuration and the multi-hop path configuration, we show the goodput for the 200 ms RTT (longer RTT) flows and the end-to-end (multi-hop) flows respectively. In both these configurations, the above mentioned flows are the ones which receive a much lower goodput compared to their fair share under existing policies such as RED, Blue and Tail Drop. We observe that the effect of increasing the number of buckets produces diminishing returns once we go past 10 buckets. In fact, there is only a marginal increase in the goodput received when we go from 10 buckets to 100 buckets. Since at each bottleneck link there are a 100 TCP flows, this implies that our algorithms are scalable and can perform very well even with *one-tenth* the number of queues as flows.

We also present the standard deviation in goodput received by each flow for different numbers of queues in Figure 6.13(b). The results are presented as a ratio of the standard deviation to the fair share bandwidth to better illustrate the measure of the standard deviation. We notice that changing the number of queues does not have a significant impact on the standard deviation of the goodputs, and thus we do not lose any fairness by using fewer queues, relative to the number of flows. Also, the overall standard deviation is below 15% of the fair share goodput for all our multi-queue policies, regardless of the number of queues.

6.4.5 Short-Term Fairness

One concern regarding policies such as DSDRR and QSDRR is that since they mark certain queues for discard, TCP flows mapped to those queues would suffer from short-term unfairness due to loss of throughput. In this section, we address this concern by quantifying this unfairness, using the time spent by a queue in discard state as a metric.

For our evaluation, we use the single-bottleneck link configuration with 100 long-lived TCP Reno flows and a buffer size of a 1000 packets. Figure 6.14 illustrates the distribution of the time in discard state for each queue under DSDRR and QSDRR for the



(a) DSDRR

(b) QSDRR

Figure 6.14: Distribution of queue discard times for DSDRR and QSDRR

Table 6.1: Discard queue time statistics

	DSDRR (s)	QSDRR (s)
Maximum	0.0964	0.2792
Minimum	0.0353	0.0160
Average	0.0658	0.0749
Std. Dev.	0.0085	0.0449

simulation run. For a queue i, each point in the graph denotes the time in seconds that it was in *discard-mode* during the simulation run. We note that this is not the cumulative time the queue is in discard mode during the simulation, but the individual durations when it is marked for discard. In the case of DSDRR, this implies that during each of these time durations, queue i's discard bit was set and all received packets destined for queue i were dropped. For QSDRR, this means that during each of these time durations, queue i was the *drop-queue*. Table 6.1 summarizes the statistics of the queue discard times.

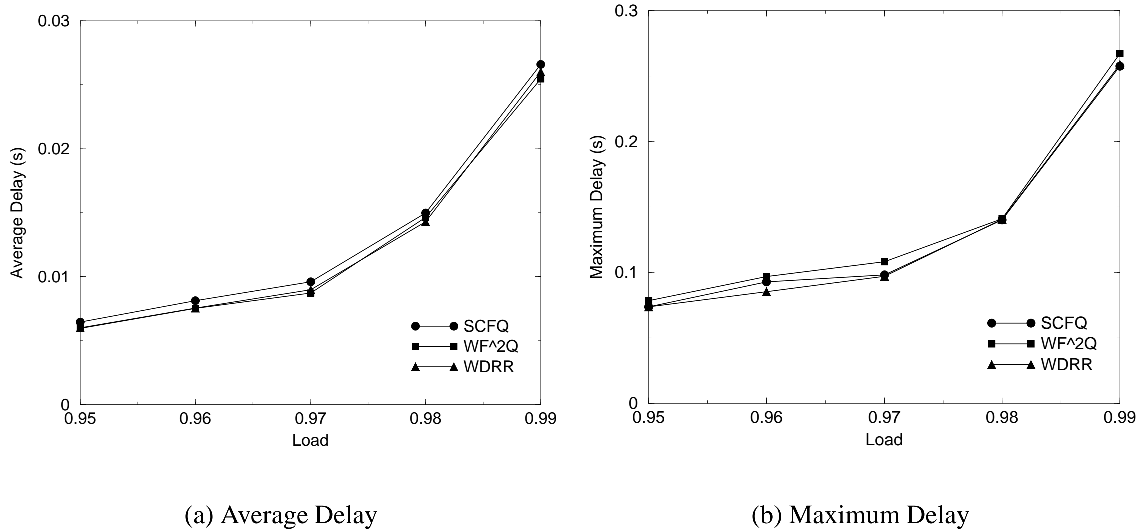
From the graphs and the table, we notice that under DSDRR, queues remain in discard modes for only about 66 ms on the average and 96 ms in the worst case. Since the RTT for the flows is 100 ms, the unfair treatment of TCP flows lasts for a very short time (less than one RTT period). Also, we note that DSDRR is actually better than QSDRR in terms of short-term fairness to individual TCP flows.

Chapter 7

Metrics for Evaluating Fair-Queueing Algorithms

Recently there have been a large number of work-conserving fair-queueing algorithms developed such as SCFQ [32], WF^{2}Q [3] and DRR [59]. Although each of them have been analytically evaluated for their worst-case delay and fairness index values, no metrics have been developed to evaluate their performance in real-world scenarios, where the assumption that the queues are backlogged all the time is unrealistic. Also, the actual differences in fairness and delay values between these three algorithms over large links (10 Gb/s) is negligible.

We use a simple example to illustrate the point that the share of the link given to individual flows is indistinguishable for reasonably sized time periods (which are still small relative to the RTT). Also, the actual delay value differences are negligible in comparison to the RTT. Consider a 10 Gb/s link with a maximum packet size of 1 KB (8 Kb) shared by a 1,000 flows, where 1 flow has a weight of 10 and the rest have a weight of 1 each. Thus, the transmission time of a packet is 0.8 μs. For periods of 10 ms (which are fairly small relative to typical RTTs of 100 ms), more than 10,000 packets will be transmitted. For this scenario, even under FQ policies such as Weighted DRR (WDRR), every flow will have had a chance to transmit on multiple occasions during the 10 ms interval. Thus, the share of the link given to each flow will be indistinguishable under WDRR, SCFQ and WF^{2}Q. Now, the analytical worst-case delay bound for the flow with weight 10 under WDRR is 10 times that for a similar weighted flow under WF^{2}Q. However, at link rates of 10 Gb/s, the actual delays (even if 10 times higher) are insignificant when compared to a typical RTT of 100 ms. For our example, the analytical worst-case delay for a flow with weight 10 under WDRR is 0.8 ms compared to 0.08 ms under WF^{2}Q. Also, the analytical worst-case delay



(a) Average Delay

(b) Maximum Delay

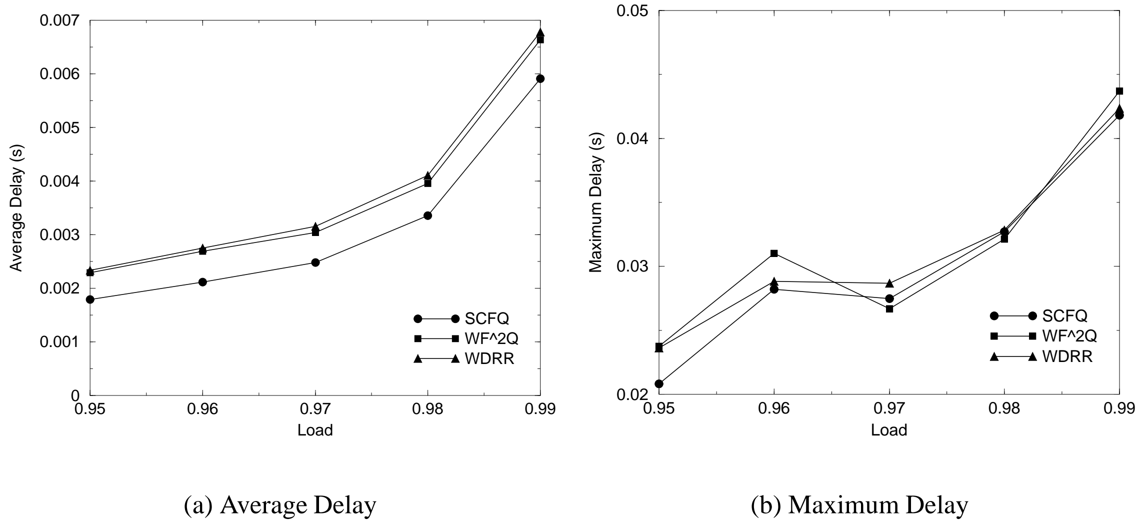
Figure 7.1: Average and Maximum Delays for small (1 Mb/s) flows

for WDRR assumes a case where all other flows would have to have a maximum sized packet at the head of each of their queues at precisely the same instant after the higher weighted flow had been served in its round and exhausted its quantum. In a real network, due to varying queueing delays along different paths, this situation is unlikely to arise. Thus, in practice, observed delays for flows using WDRR are much smaller.

In this chapter, we present our investigations comparing WDRR, SCFQ and WF^{2}Q under varying traffic conditions. For our simulation setup, we use a single bottleneck link configuration as shown in Figure 6.2, but with different bottleneck link bandwidths. The sources are UDP sources and the bottleneck link buffer is made large enough so that there are no packet drops during the simulation run.

7.1 Delay Performance

For our first test, we evaluate the three algorithms using the *delay* metric. We measure the average and maximum delays a packet suffers on the bottleneck link. Figures 7.1 and 7.2 show the packet delays for a simulation setup of 10 UDP sources sending Poisson traffic over a bottleneck link with a capacity of 20 Mb/s. 9 sources have a weight of 1 and one source has a weight of 10. The mean sending rate of each source is adjusted according to the distribution of weights and the desired load on the bottleneck link. For example, for a link load of 0.95, 9 sources send at a mean rate of 1 Mb/s and one source sends at 10 Mb/s.



(a) Average Delay

(b) Maximum Delay

Figure 7.2: Average and Maximum Delays for the large (10 Mb/s) flow

We measured the average and maximum delays for all flows, but we plot the delays for one small (1 Mb/s) flow and the large (10 Mb/s) flow.

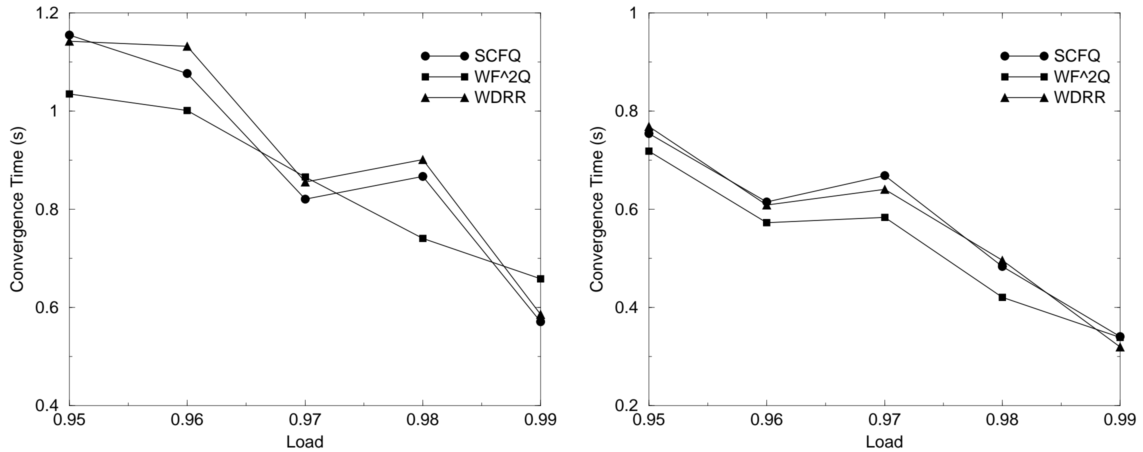
Although worst-case analysis of the three algorithms show that WF^{2}Q has the tightest delay bound and WDRR the weakest, we notice for Figures 7.1 and 7.2 that for a realistic scenario, there is negligible difference in the delay characteristics of all the three algorithms. It is interesting to note that for the higher weighted flows (10 Mb/s), SCFQ actually has the smallest average delay per packet for all link loads amongst the three FQ algorithms.

7.2 Throughput Performance

In our second study, we evaluate the FQ algorithms using a *throughput* metric. For this metric, we measure the time it takes for all sources to achieve throughputs which are within 10% of their fair-share (reserved) bandwidth. We define this time as *convergence time*.

Figure 7.3 shows the convergence times for two different input traffic scenarios. The simulation setup for Figure 7.3(a) uses 9 UDP Poisson flows with weight 1 and one UDP Poisson flow with weight 10. For Figure 7.3(b), the simulation uses 10 identical UDP Poisson sources with weight 1. The convergence time shown is an average over 51 samples taken during a 250 second simulation run.

As with the delay metric results, Figure 7.3 shows that there is almost no difference between the convergence times of all three FQ algorithms. One issue with this simulation



(a) Convergence times for 9 1 Mb/s and 1 10 Mb/s flows

(b) Convergence times for 10 1.9 Mb/s flows

Figure 7.3: Time taken for flows to reach within 10% of their fair-share bandwidth

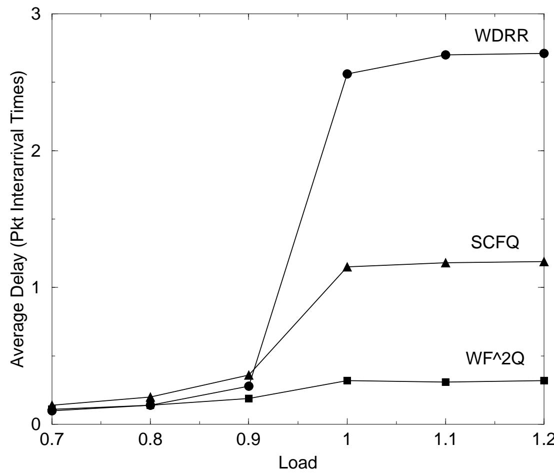
setup is that we cannot easily isolate the variations in performance due to dynamics of the Poisson source and that due to the FQ algorithm's scheduling policy.

7.3 Single Target Flow Model

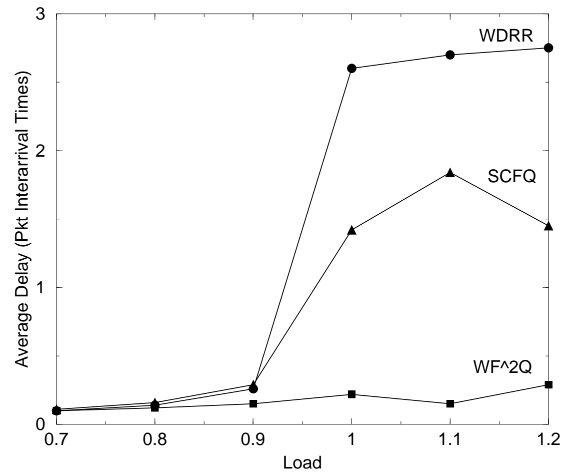
We observed in the previous simulation studies using Poisson traffic sources that it was difficult to differentiate between various FQ algorithms. One of the main issues was that since the traffic sources themselves had some randomness, we were not able to determine whether the delay and throughput convergence times obtained were due to the characteristics of the FQ algorithm or due to vagaries in the source traffic.

Given this observation, we develop a new simulation model, where we study the performance of a *single* target Constant-Bit Rate (CBR) flow under different background loads for each of the FQ algorithms. Since the target flow is a CBR, we can precisely control its traffic pattern and observe its behaviour under the different FQ algorithms. In this model, we use UDP Poisson sources as background load. We vary the total number of sources from 10 sources to 50 sources. The bottleneck bandwidth is 100 Mb/s and the target source data rate is 5 Mb/s. Packet size for all flows is set to 1,000 bytes (8,000 bits), which corresponds to the target source sending 1 packet every 1.6 ms. The network load is varied by varying the mean source rate of the background flows.

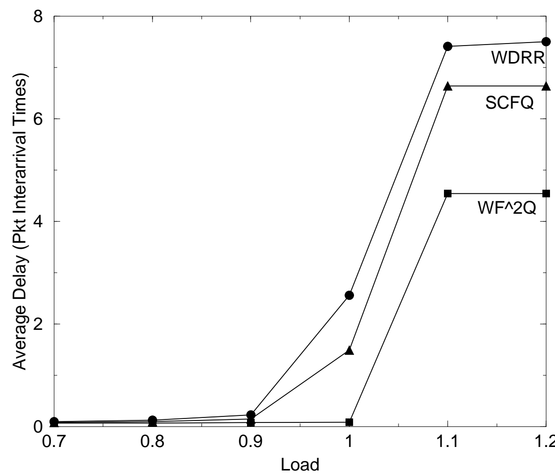
7.3.1 Delay Performance



(a) Average delay for a 5 Mb/s CBR flow with 9 Poisson flows as background traffic



(b) Average delay for a 5 Mb/s CBR flow with 19 Poisson flows as background traffic



(c) Average delay for a 5 Mb/s CBR flow with 49 Poisson flows as background traffic

Figure 7.4: Average delay experienced by the 5 Mb/s target CBR flow

Figure 7.4 shows the average delays experienced by the target 5 Mb/s CBR flow with 9, 19 and 49 background traffic flows. To get a more intuitive understanding of the delay times, we plot the delay in multiples of **packet interarrival time** of the target CBR flow. Since the target flow is sending at 5 Mb/s with a packet size of 1,000 bytes, the packet interarrival time is 1.6 ms. Viewing the delay in terms of packet interarrival times enables

us to quickly gauge how many *packets* will be enqueued for the target source on average due to queueing and scheduling delays caused by the FQ algorithm.

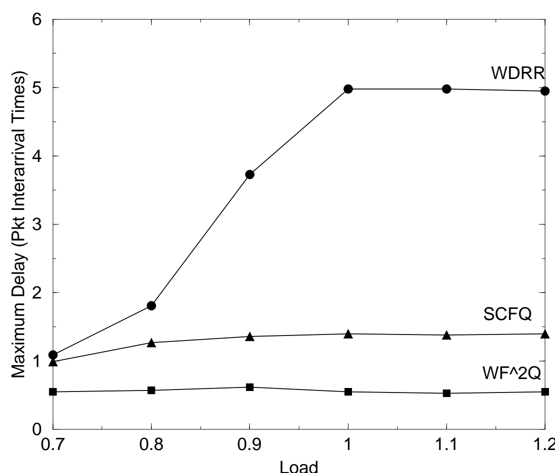
From Figure 7.4, we notice that as expected from the worst-case analysis, WDRR has the longest average delays, while WF^2Q has the shortest average delays. However, although the worst-case analysis predicts a linear increase in delays with increase in number of flows for WDRR and SCFQ, we notice that with flows increasing from 10 to 20, there is a negligible increase in average delays for the target flow. Also, the delays experienced by the target flow under WDRR and SCFQ are not an order of magnitude higher than those experienced under WF^2Q as predicted by worst-case analysis.

Another interesting observation is the delay performance of the target source with 49 background Poisson sources. In this scenario, the delay performance degrades for WDRR and SCFQ as expected (but it is still sub-linear), but so does the delay performance under WF^2Q. The reason we see the higher average delays under WF^2Q for loads greater than 1.0 is due to queueing delays caused by packets getting queued in the CBR flow queue. Under this scenario, the delays experienced by the target source are dominated by queueing delays. Although the source is a CBR source, when it is operating at 100% of its allocation and the background load is greater than 1.0, a small amount of packets will get queued. We observe that if we reduce the target CBR's rate to 95% of its allocated rate, the average delay observed under WF^2Q drops to 0.12 ms and the maximum delay is 0.16 ms. Finally, as long as the load on the network is under 0.9, there is a negligible difference in the delay performance between all three FQ algorithms.

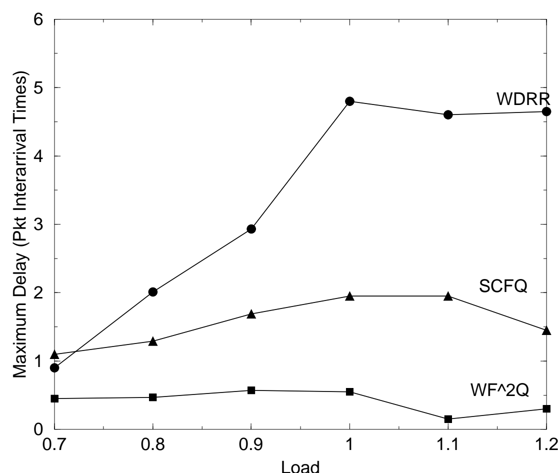
Figure 7.5 shows the maximum delays experienced by the target 5 Mb/s CBR flow with 9, 19 and 49 background traffic flows. For WDRR, the maximum delay experienced is about twice the average delay. However, it is interesting to note that for SCFQ, the maximum delay experienced is not much higher than the average delay for 9 and 19 background traffic flows. For 49 background traffic flows, the maximum delay experienced for all three FQ algorithms increases similarly to the average delay. Again for loads under 0.9, there is negligible difference between the three FQ algorithms and even at higher loads, WDRR and SCFQ are within a factor of 2 in performance compared to WF^2Q.

7.3.2 Throughput Performance

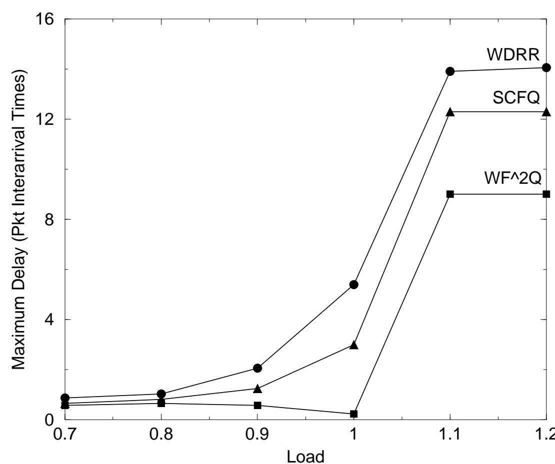
In our second study, we evaluate the FQ algorithms using a *throughput* metric. For this metric, we measure the time it takes for the target source to achieve a throughput which is within 10% of its fair-share (reserved) bandwidth (5 Mb/s). We define this time as



(a) Maximum delay for a 5 Mb/s CBR flow with 9 Poisson flows as background traffic



(b) Maximum delay for a 5 Mb/s CBR flow with 19 Poisson flows as background traffic

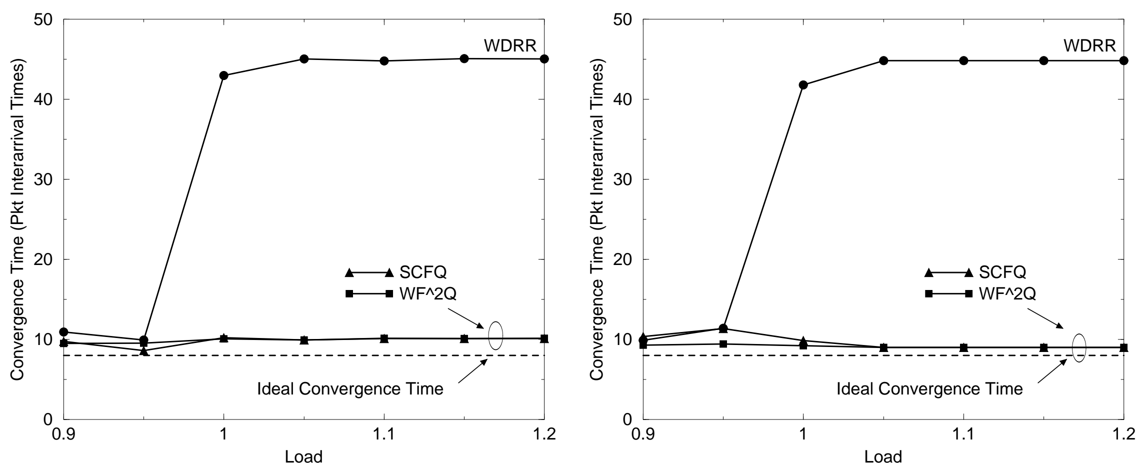


(c) Maximum delay for a 5 Mb/s CBR flow with 49 Poisson flows as background traffic

Figure 7.5: Maximum delay experienced by the 5 Mb/s target CBR flow

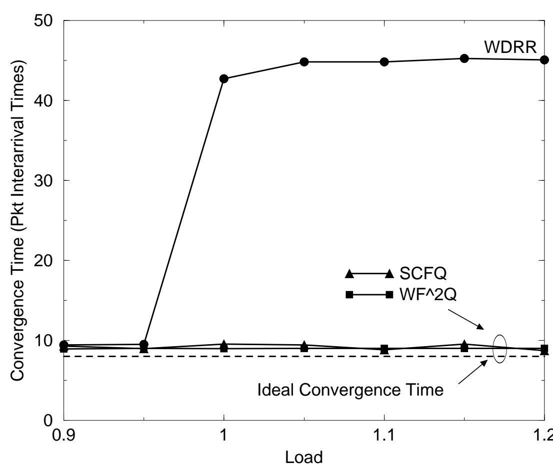
convergence time. The convergence time we use is the *first* time after which the throughput of the flow *never* goes below 10% of its fair-share (reserved) bandwidth.

Figure 7.6 shows the convergence times for two different input traffic scenarios over a 100 Mb/s bottleneck link. The simulation setup for Figure 7.6(a) uses 9 UDP Poisson flows as background traffic and one UDP CBR flow sending at 5 Mb/s which is our



(a) Convergence time for a 5 Mb/s CBR flow with 9 Poisson flows as background traffic

(b) Convergence time for a 5 Mb/s CBR flow with 19 Poisson flows as background traffic



(c) Convergence time for a 5 Mb/s CBR flow with 49 Poisson flows as background traffic

Figure 7.6: Time taken for 5 Mb/s target flow to reach within 10% of its fair-share bandwidth

target flow. For Figure 7.6(c), we have 19 UDP Poisson flows as background traffic competing with the same target flow and for Figure 7.6(c), we have 49 UDP Poisson flows as background traffic competing with the same target flow. To generate different loads, the background traffic's sending rate is scaled appropriately, whereas the target flow's rate is kept constant at 5 Mb/s. For both scenarios, we only show the convergence time for the throughput of the target CBR flow. Instead of showing convergence times in seconds, we

show the times as a multiple of **interarrival time between packets**. Since our target flow is a CBR flow, this unit provides a quick measure of how many packets are sent by the flow before it achieves its reserved bandwidth.

Since the target flow is a CBR flow, we can analytically derive the ideal convergence time. We start measuring its rate just after it transmits its first packet. Assuming ideal behaviour for the scheduler, it would send its next packet after one packet interarrival time. At this time, its rate would be 1. However, during the next packet interarrival time, its throughput would reduce from 1 to 1/2 (just before arrival of the next packet). During the third interarrival time, its throughput would reduce to 1/3. Using a similar analysis, during the eight interarrival time, its throughput would reduce to 1/8. After this, during the ninth interarrival time, its throughput would reduce to 1/9 which is within the 10% tolerance. Thus, ideally a CBR flow would converge to within 10% of its fair-share throughput in 8 packet interarrival times.

Figure 7.6 shows that, for loads under 0.95, there is almost no difference between the convergence times of all three FQ algorithms. For loads greater than 0.95, the target flow under WDRR takes about four times as much time to converge compared to its convergence time under SCFQ and WF^2Q. This is mainly due to all flows being backlogged at the higher load values, causing the target flow's packets to be delayed since it is served in round robin order. It is interesting to note that the convergence times for WDRR do not change when we increase the number of background traffic flows while keeping the target flow rate and bottleneck bandwidth constant.

From our simulation results we observe that although WF^2Q performs the best under both the delay and throughput metrics, SCFQ has a similar performance under the throughput metric and performs only slightly worse under the delay metric. Thus, although using worst-case analysis, SCFQ does not provide a tight delay bound compared to WF^2Q, we observe in practice it is comparable in performance and has much lower complexity for implementation. WDRR has the lowest implementation complexity and for loads less than 1.0, performs comparably to both SCFQ and WF^2Q under the throughput and delay metrics.

Chapter 8

Related Work

In this chapter, we discuss prior work in buffer management algorithms, applications for Bloom filters in buffer management and traffic measurement, shared-memory buffer management for packet switches and in the area of fair queueing algorithms and analysis.

8.1 Other Buffer Management Policies

Our DRR-based policies, TDRR and QSDRR, which combine fair queueing and packet discard policies, provide one particular solution for managing very small buffers while maintaining very high link utilization and goodput. In this section, we compare our approach with other related approaches. One thing to note about all the related work is that none of the approaches have been tested on multiple network configurations or with heterogeneous traffic. Also, all of the RED variants presented below fail to address the issue of unfairness to longer RTT TCP flows.

8.1.1 FRED

One proposal for using RED mechanisms to provide fairness is Flow-RED (FRED) [47]. The idea behind FRED is to keep state based on the instantaneous queue occupancy of a given flow. It defines a threshold, min_q, which is the minimum number of packets each source is allowed to queue. When a new packet arrives and the queue size is greater than min_{th}, FRED will apply RED to sources whose buffer occupancy exceeds min_q. Although this algorithm provides rough fairness in many situations, since it maintains a min_q threshold for all sources, it needs a large buffer space to work well. We have shown that TDRR and QSDRR are able to provide fair-sharing for very small buffers even with a large

number of flows. Without sufficient buffer space, it becomes hard for FRED to detect non-responsive flows since they may not have enough packets continually queued to trigger the detection mechanism. In addition, non-responsive flows are immediately re-classified as being responsive as soon as they clear their packets from the congested queue. For small queue sizes, it is quite easy to construct a transmission pattern which circumvents FRED's protection mechanisms. Also, since FRED does not maintain long-term statistics on a flow's queue occupancy, it cannot protect against misbehaving flows. On the other hand, TDRR maintains an exponentially-weighted throughput average for each flow, allowing it to "remember" events much longer in the past than the queue time constant; this allows it to enforce fairness, even for small buffer sizes.

8.1.2 Self-Configuring RED

Self-configuring RED [26] is a proposal for an adaptive RED policy that can self-parameterize given different congestion types. This policy is similar to Blue, where the RED's dropping probability, max_p is decreased when the average queue size falls below min_{th} and increased when the average queue size exceeds max_{th}. This improves over RED in reducing the queue size variations, but does not help provide better fair-sharing between flows, suffering from the same weaknesses present in RED.

8.1.3 TCP with per-flow queueing

Another proposal for managing TCP buffers is using frame-based fair-queueing [60] with longest queue or random discard policy [61]. This policy is similar to plain DRR. However, it has a disadvantage in that the frame-based fair-queueing uses the rate allocated to each flow in its scheduling policy. This implies that it needs to know the number of flows apriori, which is a difficult requirement to meet. We have shown that our multiqueue policies can adapt to any number of flows, even if the ratio of flows to queues is 10:1. We have also shown that a fair queueing scheduler with longest queue discard (plain DRR) does not perform very well over a single-bottleneck configuration for small buffers.

8.2 TCP/Active Queue Management (AQM) Analysis

In [48], Steven Low has proposed a duality model of end-to-end congestion control and uses it to study the equilibrium behaviour of TCP and AQM policies. The basic idea is to regard source rates as primal variables and congestion measures as dual variables and

congestion control as a distributed primal-dual algorithm over the Internet to maximize aggregate utility subject to capacity constraints. The primal iteration is carried out by TCP algorithms such as Reno or Vegas and the dual iteration is carried out by queue management algorithms such as Tail Drop, RED and REM. Using this theory, the authors have proposed a new version of TCP that can perform very well in high bandwidth-delay environments.

This theory works well for AQM policies such as RED or REM where it is feasible to obtain the packet drop probability. The authors have also formulated a series of equations for TCP Vegas over Tail Drop, where they assume that the Tail Drop queue is large enough such that there are no packet drops using TCP Vegas. However, it is not feasible to apply this theory in general to AQM policies such as QSDRR, TDRR and Tail Drop using Reno, where packet discard decisions are made using local queue lengths.

Another issue with the model is that it can be used to study only the *equilibrium* behaviour of TCP flows. Given the increase in network bandwidth and that the majority of the TCP traffic is Web traffic (which is inherently short and bursty), most TCP flows will never reach an equilibrium state. Thus, the model does not well represent the majority of the TCP traffic. Another way to look at this issue is by comparing *mice* (short-lived flows) to *elephants* (long-lived, high-bandwidth flows). Since Web traffic (short-lived flows) dominates all other TCP traffic, it is the *mice* that make up most of the TCP traffic. Thus, network congestion is more likely to be caused by a large number of *mice* than a few *elephants*. In [48], the author admits that the proposed model is useful for studying and controlling the behaviour of *elephants* allowing the *mice* to use the remaining bandwidth. However, in the current Internet, a large number of *mice* are more likely to be the cause of congestion, this model is not very useful in understanding or controlling the congestion.

8.3 Bloom Filters

A Bloom filter is a space-efficient representation of a set or a list that handles membership queries. There are numerous examples where one would like to use a list in a network. Especially when space is an issue, a Bloom filter may be an excellent alternative to keeping an explicit list. The drawback of using a Bloom filter is that it introduces false positives. The effect of a false positive must be carefully considered for each specific application to determine whether the impact of false positives is acceptable. In this section we describe two earlier approaches that use Bloom filters for queue management and traffic monitoring.

8.3.1 Queue Management: Stochastic Fair Blue (SFB)

In [28], the authors propose and evaluate SFB, which combines Blue and Bloom filters to produce a highly scalable means to enforce fairness amongst flows using a small amount of state and a small amount of buffer space. SFB is a FIFO queueing algorithm that identifies and rate-limits non-responsive flows based on accounting mechanisms similar to those used with Blue. SFB maintains N x L accounting bins. The bins are organized in L levels with N bins in each level. SFB also maintains (L) independent hash functions, each associated with one level of the accounting bins. Each hash function maps a flow into one of the N accounting bins in that level. The accounting bins are used to keep track of queue occupancy statistics of packets belonging to a particular bin. Thus, SFB uses a Bloom filter with N x L bits with L hash functions. This is in contrast to Stochastic Fair Queueing [50] (SFQ where the hash function maps flows into separate queues. Each bin in SFB keeps a marking/dropping probability p_m as in Blue, which is updated based on bin occupancy. As a packet arrives at the queue, it is hashed into one of the N bins in each of the L levels. If the number of packets mapped to a bin goes above a certain threshold (i.e., the size of the bin), p_m for the bin is increased. If the number of packets drops to zero, p_m is decreased. The observation which drives SFB is that a nonresponsive flow quickly drives p_m to 1 in all of the L bins it is hashed into. Responsive flows may share one or two bins with non-responsive flows, however, unless the number of non-responsive flows is extremely large compared to the number of bins, a responsive flow is likely to be hashed into at least one bin that is not polluted with nonresponsive flows and thus has a normal p_m value. The decision to mark a packet is based on p_{min}, the minimum p_m value of all bins to which the flow is mapped into. If p_{min} is 1, the packet is identified as belonging to a non-responsive flow and is then rate-limited. Note that this approach is akin to applying a Bloom filter on the incoming flows.

Although this approach works well for identifying large non-responsive flows and preferentially discarding packets from the non-responsive flows, it does not perform any better than Blue for fair-sharing of the link bandwidth. Also, this approach uses a counter per bit of the Bloom filter, which leads to a greatly increased memory requirement.

8.3.2 Traffic Measurement and Accounting

In [24] Estan and Varghese present an application of Bloom filters to traffic measurement problems inside of a router, which is similar to the technique used in the Stochastic Fair Blue algorithm [28]. The goal in this work was to easily determine heavy flows in a router.

Each entering packet is hashed k times into a Bloom filter. Associated with each location in the Bloom filter is a counter that records the number of packet bytes that have passed through the router associated with that location. The counter is incremented by the number of bytes in the packet. If the minimum counter associated with a packet is above a certain threshold, the corresponding flow is placed on a list of heavy flows. Heavy flows can thereby be detected with a small amount of space and a small number of operations per packet. A false positive in this situation corresponds to a light flow that happens to hash into k locations that are also hashed into by heavy flows, or to a light flow that happens to hash into locations hit by several other light flows. All heavy flows, however, are detected.

The authors also introduce the idea of a conservative update, an interesting variation that reduces the false positive rate significantly for real data. When updating a counter upon a packet arrival, it is clear that the number of previous bytes associated with the flow of that packet is at most the minimum over its k counters. Let this be M_k. If the new packet has B bytes, the number of bytes associated with this flow is at most $M_k + B$. So the updated value for each of the k counters should be the maximum of its current value and $M_k + B$. Thus, instead of adding B to each counter, conservative update only changes the values of counter to reflect the most possible bytes associated with the flow. This reduces the probability that several light flows hashing to the same location can raise the counter value over the threshold.

8.4 Shared-Memory Buffer Management

In [13], Choudhury and Hahne propose a Dynamic Threshold (DT) policy for managing buffers in a shared-memory packet switch. The approach is conceptually similar to bottleneck flow control [36] and to the bandwidth balancing mechanism in distributed queue dual bus (DQDB) networks [33]. The key idea is that the output queue length threshold, at any instant of time, is proportional to the current amount of unused buffering in the switch. Cell arrivals for an output port are blocked whenever the output port's queue length equals or exceeds the current threshold value. The DT method deliberately holds a small amount of buffer space in reserve, but distributes the remaining buffer space equally among the active output queues.

The DT policy adapts to changes in traffic conditions. Whenever the load changes, the system will go through a transient. For example, when a lightly loaded output port suddenly becomes very active, its queue will grow, the total buffer occupancy will increase, the control threshold will decrease, and queues exceeding the threshold will have their

arrivals blocked temporarily while they drain, freeing up more cell buffers for the newly active queue. Eventually all the very active queues, both old and new, will stabilize at equal lengths. The DT policy also deliberately wastes a small amount of buffer space. This *wastage* actually serves two useful functions. The first advantage of maintaining some spare space at all times is that it provides a cushion during transient periods when an output queue first becomes active. This cushion reduces cell loss during such transients. Secondly, when an output queue has such a load increase and begins taking over some of the spare buffer space, this action signals the allocation mechanism that the load conditions have changed and that a threshold adjustment is now required.

We adapted this shared-memory buffer management policy to implement a new discard policy for fair-sharing of the link bandwidth among TCP flows. The proposed policies and their evaluation is described in Chapter 6.

8.5 Fair Queueing (FQ)

Several scheduling algorithms are known in the literature for bandwidth allocation and transmission scheduling. These include the packet-by-packet version of Generalized Processor Sharing [55] (also known as Weighted Fair Queueing [19]), VirtualClock [67], Stochastic Fairness Queueing [50], Self-Clocked Fair Queueing [32], Weighted Round Robin [44], Deficit Round Robin (DRR) [59], Frame-based Fair Queueing [60], Stratified Round Robin [57]. We chose DRR due to its simplicity and ease of implementation in hardware.

As discussed in Chapter 7, analytical worst-case bounds for FQ algorithms are overly pessimistic. Until now, there has not been a concerted effort to develop a simulation framework for evaluating different FQ algorithms over realistic network configurations and traffic patterns. In one previous study [37], the authors present a simulation study of hierarchical packet fair queueing algorithms. The simulation study bolsters our claim by showing that there is a significant gap between the worst delays obtained via simulation for non-WF^2Q and what we would expect from the analytical bounds. Although this study is a step in the right direction, the simulation scenarios are fairly limited and concentrate only on the delay metric. We propose to pursue a more systematic and complete study of different metrics for evaluating FQ algorithms.

Chapter 9

Conclusions and Future Work

This thesis has demonstrated the inherent weaknesses in current queue management policies commonly used in Internet routers. These weaknesses include limited ability to perform well under a variety of network configurations and traffic conditions, inability to provide a fair-sharing among competing TCP connections with different RTTs and relatively low link utilization and goodput in routers that have small buffers. In order to address these issues, we presented TDRR and QSDRR, two different packet-discard policies used in conjunction with a simple, fair-queueing scheduler, DRR. Through extensive simulations, we showed that TDRR and QSDRR significantly outperform RED and Blue for various configurations and traffic mixes in both the average goodput for each flow and the variance in goodputs. For very small buffer sizes, on the order of 5-10% of the bandwidth-delay product, we showed not just that our policies significantly outperformed RED, Blue and Tail Drop, but were able to achieve near optimal goodput and fairness. Thus, using QSDRR and TDRR, we can operate effectively with *half* the end-to-end delay.

When we started this work, most of the operating systems were using either TCP Tahoe or TCP Reno. Thus, we concentrated on these two variants of TCP for all our simulations. Recently, with the increasing popularity of Linux, there is a new variant of TCP implemented in Linux which is similar to TCP Vegas. Thus, an interesting future study would be to investigate the effects on fairness using QSDRR for competing TCP Reno and TCP Vegas flows.

Using multiple queues raised two new concerns: scalability and excess memory bandwidth usage caused by dropping packets which have been queued. In this thesis, we developed and evaluated a flow distribution algorithm using a Bloom filter architecture with dynamic rebalancing to evenly distribute flows among queues to improve scalability. We proposed and evaluated two dynamic rebalancing policies, PB and DB which achieve near

optimal flow distribution for varied load conditions while reducing the overhead of moving packets across queues. Thus, using our algorithms significantly reduces the memory requirement compared to maintaining per-flow state while achieving near optimal flow distribution and using them in conjunction with QSDRR, we can achieve the performance of per-flow queueing at a significantly reduced cost.

In this thesis, we presented DTDRR and DSDRR as alternatives to QSDRR that provide comparable performance, while allowing packets to be discarded on arrival, saving memory bandwidth. Through extensive simulations, we showed that DTDRR and DSDRR significantly outperform RED, Blue and Tail-Drop for various configurations and traffic mixes in both the average goodput for each flow and the variance in goodputs and the performance for both long-lived and short burst TCP flows is very close to that of QSDRR. We also showed that these algorithms can provide good performance, when each queue is shared among multiple flows, and we show that the hysteresis in the packet discard policy for DSDRR has little effect on short-term fairness.

Finally, we proposed better methods for evaluating the performance of fair-queueing methods. In the current literature, fair-queueing methods are evaluated based on their worst-case performance. This can exaggerate the differences among algorithms, since the worst-case behavior is dependent on the the precise timing of packet arrivals. We developed two metrics: *delay* and *throughput*, along with a simulation setup to better study and compare the average behaviour of different FQ algorithms. We showed that although WF^2Q has the best delay bounds and fairness index according to analytical studies, less complex FQ algorithms such as WDRR and SCFQ provide comparable performance under both the *delay* and *throughput* metrics.

One useful avenue for future work here is developing a complete performance comparison of all the mainstream FQ algorithms under the delay and throughput metrics. Another interesting extension of this study would be to use real traffic traces in the simulation model to compare the delay and throughput performances of different FQ algorithms. Using such a traffic model would enable network administrators to tailor their routers to use the least complex FQ algorithm that would meet the desired delay and throughput guarantees.

References

[1] NetBSD. http://www.netbsd.org.

[2] M.L. Bailey, B. Gopal, P. Sarkar, M.A. Pagels, and L.L. Peterson. Pathfinder: A pattern-based packet classifier. In *Proceedings of the* 1st *Symposium on Operating System Design and Implementation*, November 1994.

[3] Jon C. R. Bennett and Hui Zhang. WF^2Q: Worst-Case Fair Weighted Fair Queueing. In *IEEE INFOCOM 1996*, pages 120–128, 1996.

[4] Jon C. R. Bennett and Hui Zhang. Hierarchical Packet Fair Queueing Algorithms. *IEEE/ACM Transactions on Networking*, 5(5):675–689, October 1997.

[5] Y. Bernet, R. Yavatkar, P. Ford, F. Baker, and L. Zhang. A Framework for End-to-End QoS Combining RSVP/IntServ and Differentiated Services. Internet Draft, March 1998.

[6] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. Internet Draft, August 1998.

[7] B. Bloom. Space/time tradeoffs in hash coding with allowable errors. *Communications of ACM*, 13(7):422–426, July 1970.

[8] R. T. Braden. Extending TCP for Transactions - Concepts. RFC-1379, November 1992.

[9] R. T. Braden. TCP Extensions for Transactions Functional Specification. RFC-1644, July 1994.

[10] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *ACM SIGCOMM '94*, August 1994.

[11] Andrei Broder and Michael Mitzenmacher. Network Applications of Bloom Filters: A Survey. In *Allerton Conference*, 2002.

[12] Sumi Choi, John Dehart, Anshul Kantawala, Ralph Keller, Fred Kuhns, John Lockwood, Prashanth Pappu, Jyoti Parwatikar, W. David Richard, Ed Spitznagel, David Taylor, Jonathan Turner, and Ken Wong. Design of a High Performance Dynamically Extensible Router. In *In DARPA Active Networks Conference and Exposition (DANCE)*, May 2002.

[13] A. Choudhury and E. Hahne. Dynamic Queue Length Thresholds for Shared-Memory Packet Switches. *IEEE/ACM Transactions on Networking*, 6(2):130–140, April 1998.

[14] Inc. CiscoWorks, CISCO Sytems.

[15] David Clark and John Wroclawski. An Approach to Service Allocation in the Internet. Internet Draft, July 1997.

[16] Mark E. Crovella and Azer Bestavros. Self-Similarity in World Wide Web Traffic Evidence and Possible Causes. In *Proceedings of ACM SIGMETRICS '96*, May 1996.

[17] Laurence Crutcher and Aurel A. Lazar. Management and Control for Giant Gigabit Networks - are we ready for B-ISDN. Technical Report 341-93-21, Center for Telecommunications Research, Columbia University, 1993.

[18] John D. DeHart, William D. Richard, Edward W. Spitznagel, and David E. Taylor. The Smart Port Card: An Embedded Unix Processor Architecture for Network Management and Active Networking. Technical Report WUCS-01-18, Washington University, August 2001.

[19] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *Internetworking: Research and Experience*, 1(1):3–26, 1990.

[20] Zubin D. Dittia. ATM Port Interconnect Chip. htp://www.arl.wustl.edu/apic.html.

[21] Zubin D. Dittia, J.R. Cox Jr., and Guru M. Parulkar. Design of the APIC: A High Performance ATM Host-Network Interface Chip. In *Proceedings of IEEE INFOCOM '95*, pages 179–187, Boston, USA, April 1995. IEEE Computer Society Press.

[22] Zubin D. Dittia, Guru M. Parulkar, and J.R. Cox Jr. The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques. In

Proceedings of IEEE INFOCOM '97, Kobe, Japan, April 1997. IEEE Computer Society Press.

[23] S. Doran. RED Experience and Differential Queueing. Nanog Meeting, June 1998.

[24] Cristian Estan and George Varghese. New Directions in Traffic Measurement and Accounting. In *ACM SIGCOMM '02*, October 2002.

[25] W. Feng, D. Kandlur, D. Saha, and K. Shin. Techniques for Eliminating Packet Loss in Congested TCP/IP Networks. Technical Report 97-661, University of Southern California, November 1997.

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

[27] W. Feng, D. Kandlur, D. Saha, and K. Shin. Blue: A New Class of Active Queue Management Algorithms. Technical Report CSE-TR-387-99, University of Michigan, April 1999.

[28] W. Feng, D. Kandlur, D. Saha, and K. Shin. Stochastic Fair Blue: A Queue Management Algorithm for Enforcing Fairness. In *IEEE INFOCOM 2001*, April 2001.

[29] S. Floyd. TCP and Explicit Congestion Notification. *ACM Computer Communications Review*, 24(5):10–23, October 1994.

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

[31] S. Floyd and V. Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4):365–386, August 1995.

[32] S. Golestani. A self-clocked fair queueing scheme for broadband applications. In *IEEE INFOCOM 1994*, April 1994.

[33] E. L. Hahne, A. K. Choudhury, and N. F. Maxemchuk. Distributed-Queue-Dual-Bus Networks With and Without Bandwidth Balancing. *IEEE Transactions on Communications*, 40:1192–1204, July 1992.

[34] E. Hashem. Analysis of random drop for gateway congestion control. Technical Report LCS TR-465, Laboratory for Computer Science, MIT, 1989.

[35] V. Jacobson. Congestion Avoidance and Control. In *ACM SIGCOMM '88*, August 1988.

[36] J. M. Jaffe. Bottleneck Flow Control. *IEEE Transactions on Communications*, pages 954–962, July 1981.

[37] S. Jamin, P. Danzig, S. Shenker, and L. Zhang. A measurement-based admission control algorithm for integrated services packet networks. *IEEE/ACM Transactions on Networking*, 5(1):56–70, February 1997.

[38] Sugih Jamin, Scott J. Shenker, and Peter B. Danzig. Comparison of Measurement-based Admission Control Algorithms for Controlled-Load Service. In *Proceedings of IEEE INFOCOM '97*, April 1997.

[39] Anshul Kantawala, Samphel Norden, Ken Wong, and Guru M. Parulkar. DiSp: An Architecture for supporting Differentiated Services in the Internet. In *INET'99 Proceedings*, June 1999.

[40] Anshul Kantawala and Jonathan Turner. Efficient Queue Management of TCP Flows. Technical Report WUCS-01-22, Washington University, August 2001.

[41] Anshul Kantawala and Jonathan Turner. Efficient Queue Management of TCP Flows. In *SPECTS 2002*, July 2002.

[42] Anshul Kantawala and Jonathan Turner. Queue Management for Short-Lived TCP Flows in Backbone Routers. In *High-Speed Networking Symposium, IEEE Globecom '02*, November 2002.

[43] Anshul Kantawala and Jonathan Turner. Intelligent Packet Discard Policies for Improved TCP Queue Management. Technical Report WUCSE-2003-41, Washington University, May 2003.

[44] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis. Weighted round-robin cell multiplexing in a general-purpose ATM switch chip. *IEEE Journal on Selected Areas in Communications*, 9:1265–1279, October 1991.

[45] Edward Knightly and Jingyu Qiu. Measurement Based Admission Control with Aggregate Traffic Envelopes. In *Proceedings of the* 10th *IEEE International Tyrrhenian Workshop on Digital Communications*, September 1998.

[46] Eileen T. Kraemer. *A Framework, Tools, and Methodology for the Visualization of Parallel and Distributed Systems*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, August 1995.

[47] Dong Lin and Robert Morris. Dynamics of Random Early Detection. In *ACM SIGCOMM '97*, September 1997.

[48] Steven H. Low. A Duality Model of TCP and Queue Management Algorithms. *IEEE/ACM Transactions on Networking*, 11(4):525–536, August 2003.

[49] B. A. Mah. An Empirical Model of HTTP Network Traffic. In *IEEE INFOCOM 1997*, April 1997.

[50] P. McKenney. Stochastic Fairness Queueing. *Internetworking: Research and Experience*, 2:113–131, January 1991.

[51] Robert Morris. Scalable TCP Congestion Control. In *IEEE INFOCOM 2000*, March 2000.

[52] K. Nichols, V. Jacobson, and L. Zhang. A Two-Bit Differentiated Services Architecture for the Internet. Internet Draft, November 1997.

[53] Borje Ohlman. Receiver Control in Differentiated Services. Internet Draft, March 1998.

[54] Venkata N. Padmanabhan and Randy H. Katz. TCP Fast Start: A Technique For Speeding Up Web Transfers. In *IEEE Globecom '98 Internet Mini-Conference*, November 1998.

[55] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control - the single node case. In *IEEE INFOCOM 1992*, May 1992.

[56] G. Parulkar, D. Schmidt, E. Kramer, J. Turner, and A. Kantawala. An Architecture for Monitoring, Visualization and Control of Gigabit Networks. *IEEE Network*, 11(5):34–43, October 1997.

[57] Sriram Ramabhadran and Joseph Pasquale. Stratified Round Robin: A Low Complexity Packet Scheduler with Bandwidth Fairness and Bounded Delay. In *ACM SIGCOMM '03*, August 2003.

[58] F. Reichmeyer, K. Chan, D. Durham, R. Yavatkar, S. Gai, K. McCloghrie, and S. Herzog. COPS Usage for Differentiated Services. Internet Draft, August 1998.

[59] M. Shreedhar and George Varghese. Efficient Fair Queueing using Deficit Round Robin. In *ACM SIGCOMM '95*, August 1995.

[60] D. Stiliadis and A. Varma. Design and analysis of Frame-based Fair Queueing: A New Traffic Scheduling Algorithm for Packet-Switched Networks. In *ACM SIGMETRICS '96*, May 1996.

[61] B. Suter, T. V. Lakshman, D. Stiliadis, and A. Choudhury. Design Considerations for Supporting TCP with Per-flow Queueing. In *IEEE INFOCOM 1998*, March 1998.

[62] J. Touch. TCP Control Block Interdependence. RFC-2140, April 1997.

[63] D. Tse and M. Grossglauser. Measurement-based call admission control: Analysis and simulation. In *Proceedings of INFOCOM '97*, April 1997.

[64] C. Villamizar and C. Song. High Performance TCP in ANSNET. *Computer Communication Review*, 24(5):45–60, October 1994.

[65] V. Visweswaraiah and J. Heidemann. Improving restart of idle TCP connections. Technical Report CSE-TR-349-97, University of Michigan, 1997.

[66] W. Willinger, M. S. Taqqu, R. Sherman, and D. V. Sherman. Self-Similarity Through High-Variability: Statistical Analysis of Ethernet LAN Traffic at the Source Level. In *Proceedings of ACM SIGCOMM '95*, pages 100–113, Aug 1995.

[67] L. Zhang. VirtualClock: a new traffic control algorithm for packet switching networks. *ACM Transactions on Computer Systems*, 9:101–124, May 1991.

Vita

Anshul Kantawala

Date of Birth October 16, 1968

Place of Birth Ahmedabad, India

Degrees
B.S. Summa Cum Laude, Computer Science, May 1991
B.S. Summa Cum Laude, Electrical Engineering, May 1991
M.S. Computer Science, May 1995
D.Sc. Computer Science, May 2005

Professional Societies Institute of Electrical and Electronics Engineers

Publications

Anshul Kantawala and Jonathan Turner (2004). Intelligent Packet Discard Policies for Improved TCP Queue Management, *CCN 2004 Proceedings*.

Anshul Kantawala and Jonathan Turner (2002). Queue Management for Short-Lived TCP Flows in Backbone Routers, *High-Speed Networking Symposium, IEEE GLOBECOM '02*.

Anshul Kantawala and Jonathan Turner (2002). Efficient Queue Management for TCP Flows, *SPECTS 2002 Proceedings*.

Sumi Choi, John Dehart, Anshul Kantawala, Ralph Keller, Fred Kuhns, John Lockwood, Prashanth Pappu, Jyoti Parwatikar, W. David Richard, Ed Spitznagel, David Taylor, Jonathan Turner and Ken Wong (2002). Design of a High Performance Dynamically Extensible Router, *DARPA Active Networks Conference and Exposition (DANCE)*.

Anshul Kantawala, Samphel Norden, Ken Wong and Guru M. Parulkar (1999). DiSp: An Architecture for supporting Differentiated Services in the Internet, *INET'99 Proceedings*.

Guru M. Parulkar, Douglas Schmidt, Eileen Kramer, Jonathan Turner and Anshul Kantawala (1997). An Architecture for Monitoring, Visualization and Control of Gigabit Networks, *IEEE Network*.

Anshul Kantawala, Guru M. Parulkar, John DeHart and Ted Marz (1997). Supporting DIS Applications using ATM Multipoint Connection Caching, *IEEE INFOCOM '97*.

James P.G. Sterbenz, Anshul Kantawala, Milind Buddhikot and Guru M. Parulkar (1992). Hardware Based Error and Flow Control in the AXON Gigabit Host-Network Interface, *IEEE INFOCOM '92*.

May 2005