Efficient Queue Management for TCP Flows

Anshul Kantawala and Jonathan Turner *
Department of Computer Science
Washington University
St. Louis, MO 63130
{*anshul,jst*}*@arl.wustl.edu*

Keywords: Backbone router, queueing delay, buffer management, per-flow queueing, fair-share, goodput.

Abstract

Packets in the Internet can experience large queueing delays during busy periods. Backbone routers are generally engineered to have large buffers, in which packets may wait as long as half a second (assuming FIFO service, longer otherwise). During congestion periods, these buffers may stay close to full, subjecting packets to long delays, even when the intrinsic latency of the path is relatively small. This paper studies the performance improvements that can be obtained by using more sophisticated packet schedulers, than are typical of Internet routers. The results show that the large buffers found in WAN routers contribute only marginally to improving router throughput, and the higher delays that come with large buffers makes them a dubious investment. The results also show that better packet scheduling algorithms can produce dramatic improvements in fairness. Using ns-2 simulations, we show that algorithms using multiple queues can significantly outperform RED and Blue, especially at smaller buffer sizes. Over a single-bottleneck link, the variance in TCP goodput using the proposed multiqueue packet schedulers is *one-tenth* that obtained with RED and *one-fifth* that obtained with Blue. Given a traffic mix of TCP flows with different round-trip times, longer round-trip time flows achieve 80% of their fair-share using multiqueue schedulers, compared to 40% under RED and Blue. We observe a similar performance improvement for multi-hop paths.

1 INTRODUCTION

Backbone routers in the Internet are typically configured with buffers that are several times 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 TCP traffic loads, and are serviced using the Tail Drop policy, the large queues remain close to full most of the time. Thus, even if each TCP 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 [1] and Morris [2], among others.

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 [3]. 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 [4, 5].

Recently another queueing algorithm called Blue [6], 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.

In this paper, we investigate how packet schedulers using multiple queues can improve performance over existing methods. Our goal is to find schedulers that satisfy the following objectives:

- *High throughput when buffers are very small (a fraction of the bandwidth-delay product)*. This allows queueing delays to be kept low.
- *Insensitivity to operating conditions and traffic*. This reduces the need to tune parameters, or compromise on

*This work is supported in part by NSF Grant ANI-9714698

performance.

- *Fair treatment of different flows*. This should hold regardless of differences in round-trip delay or number of hops traversed.

The results presented here show that both RED and Blue are deficient in these respects. 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.

Another regularly observed phenomenon for queues with Tail Drop is big swings in the occupancy of the bottleneck link queue. One of the main causes for this is the synchronization of TCP sources going through the bottleneck link. Although RED and Blue try to alleviate the synchronization problem by using a random drop policy, they do not perform well with buffers which are a fraction of 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. However, with per-flow queueing, we can explicity control the number of flows that suffer a packet loss and thus significantly reduce synchronization among flows.

We investigate queueing algorithms that use multiple queues, to isolate flows from one another. Most of the results reported use per-flow queues, but we also show that comparable performance can be obtained when queues are shared by multiple flows. 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, with respect to all three of the objectives listed above.

The rest of the paper is organized as follows. Section 2 describes the new multi-queue methods investigated here. Section 3 documents the configurations used for the simulations and the parameters used for RED and Blue. Section 4 compares the performance results of the proposed multi-queue methods against RED, Blue and Tail Drop. Section 5 presents a brief summary of related work and Section 6 concludes the paper.

2 ALGORITHMS

As stated in the previous section, when we evaluated current queueing disciplines such as RED and Blue, we found that they did not work well with small buffer sizes. Another big disadvantage with both these algorithms (more so in the case of RED), is the problem of finding the right parameters for a given queue. Given our experiments and prior work in this area, it has been shown that no single set of RED parameters work for different bottleneck bandwidths, different flow combinations and different queue lengths. Thus, to be able to effectively use RED, the parameters must be fine tuned given link bandwidth, buffer size and traffic mix. In practice, this is very difficult since the input traffic mix is continuously varying.

Given these problems with existing congestion buffer management algorithms, we decided to evaluate a fair queueing discipline for managing TCP flows. We started with using Deficit Round Robin (DRR) [7]. 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.

1. **DRR with Longest Queue Drop**
   Our first policy combined DRR with packet-discard from the longest active queue. For the rest of the paper, we refer to this policy as plain DRR or DRR, since this packet-discard policy is part of the original DRR algorithm [7] and was first proposed by McKenney in [8]. 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 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 store a throughput value associated with 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 drawback of this policy is that we need to store and update an extra parameter for each DRR queue, 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 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 1: Algorithm for QSDRR

3 SIMULATION ENVIRONMENT

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.

Table 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 2: Blue parameters

Blue		
$d1$	Increment	0.0025
$d2$	Decrement	0.00025
$freeze\_time$	Hold-time	0.1s

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 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 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 [6]. The Blue parameters we used are in Table 2 and are the ones that gave the best performance.

3.1 Single Bottleneck Link

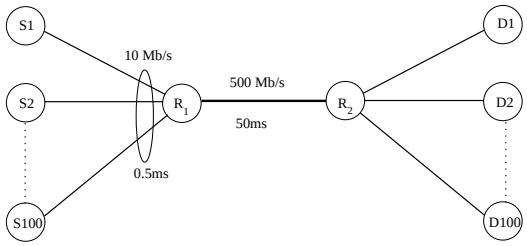


Figure 2: Single Bottleneck Link Network Configuration

The network configuration for this set of experiments is shown in Figure 2. $\{S_1, S_2, ...S_{100}\}$ are the TCP sources, each connected by 10Mb/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.

3.2 Multiple Round-Trip Time Configuration

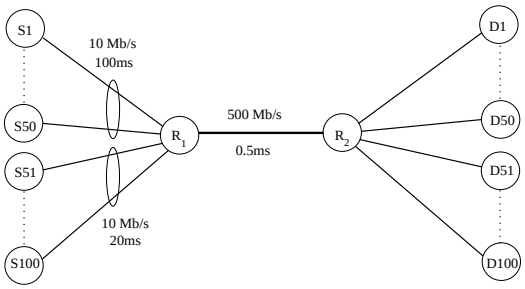


Figure 3: Multiple Round-Trip Time Network Configuration

The network configuration for this set of experiments is shown in Figure 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. For 50 sources, the link delay is set to 20ms, while it is set to 100ms for the other 50 sources.

3.3 Multi-Hop Path Configuration

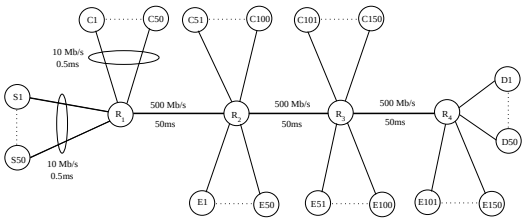


Figure 4: Multi-Hop Path Network Configuration

The network configuration for this set of experiments is shown in Figure 4. 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.

4 RESULTS

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 2.4ms to 120ms. In all our simulations, we noticed that all the policies behaved in a similar fashion past the 5000 packet buffer size.

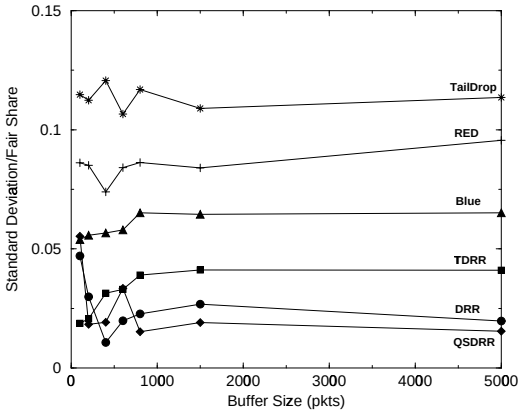
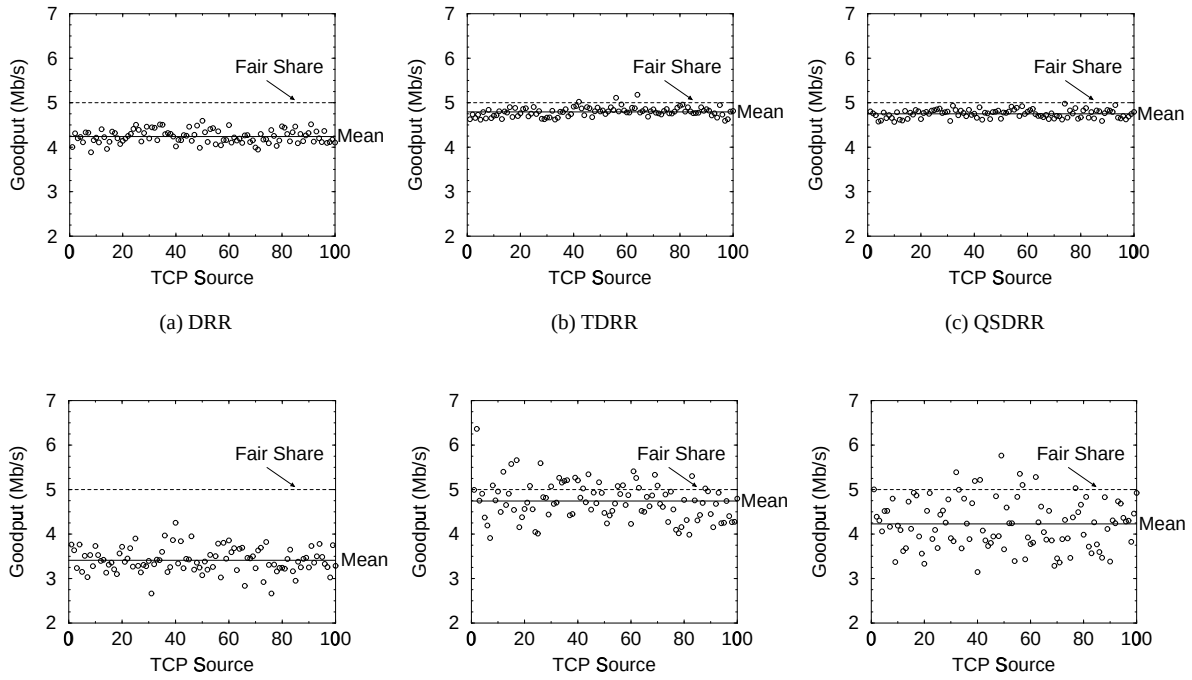


Figure 6: Standard deviation relative to fair-share for TCP Reno flows over a single-bottleneck link

4.1 Single-Bottleneck Link

The first set of graphs, shown in Figure 5, 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 5(b), 5(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 5(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 5(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 5(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 5(f)), performs most poorly, with the highest variance in goodputs and a very low average goodput.

Figure 6 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



(a) DRR (b) TDRR (c) QSDRR

(d) Blue (e) RED (f) Tail Drop

Figure 5: TCP Reno Goodput distribution over single-bottleneck link with 200 pkt 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. Thus, even for a single-bottleneck link, we observe that the multi-queue policies offer much better fairness to a set of TCP flows.

Finally, figure 7 illustrates the average fair-share bandwidth percentage received by the TCP Reno flows using different buffer sizes. For small buffer sizes, i.e. under 500 packets, TDRR and QSDRR outperform RED significantly and DRR is comparable to RED. It is interesting to note that even at a large buffer size of 5000 packets, all policies significantly outperform Blue, including Tail Drop.

4.2 Multiple Round-Trip Time Configuration

For this configuration, we use 100 TCP Reno flows over a single bottleneck link. 50 flows have a 40ms RTT and 50 flows have a 200ms RTT. Figure 8 shows the average fair-share goodput received by each set. As shown in Figure 8(a), both RED and Blue allow the 40ms RTT flows to use almost 50% more bandwidth than their fair share. Tail Drop also

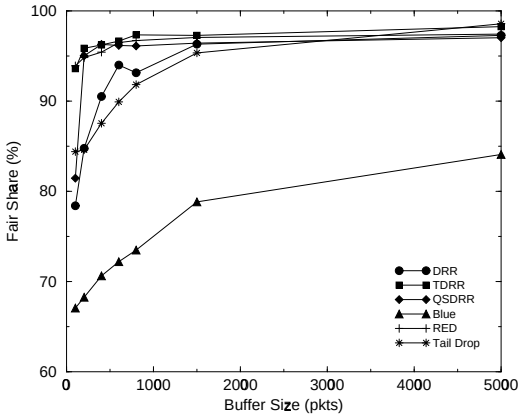
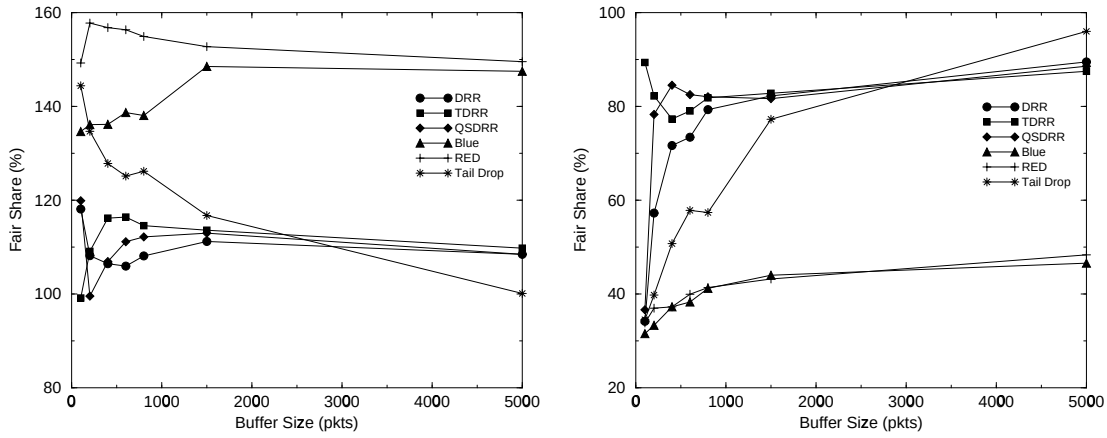


Figure 7: Fair share performance over a single bottleneck link

allows the 40ms 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 40ms RTT flows. Both RED and Blue discriminate against longer RTT flows, as we observe in Figure 8(b), the 200ms RTT flows achieve only about 40% of their fair-share bandwidth whereas using the DRR-based policies, 200ms RTT flows are able to achieve almost 90% of their fair-share.

At a very small buffer size of 100 packets, 200ms RTT flows using DRR and QSDRR get about 40% of their fair-



(a) Flows with RTT = 40ms

(b) Flows with RTT = 200ms

Figure 8: Fair share performance of different RTT flows over a single bottleneck link

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 200ms RTT flows achieve about 80% of their fair-share bandwidth on the average. 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 200ms RTT flows.

4.3 Multi-Hop Path Configuration

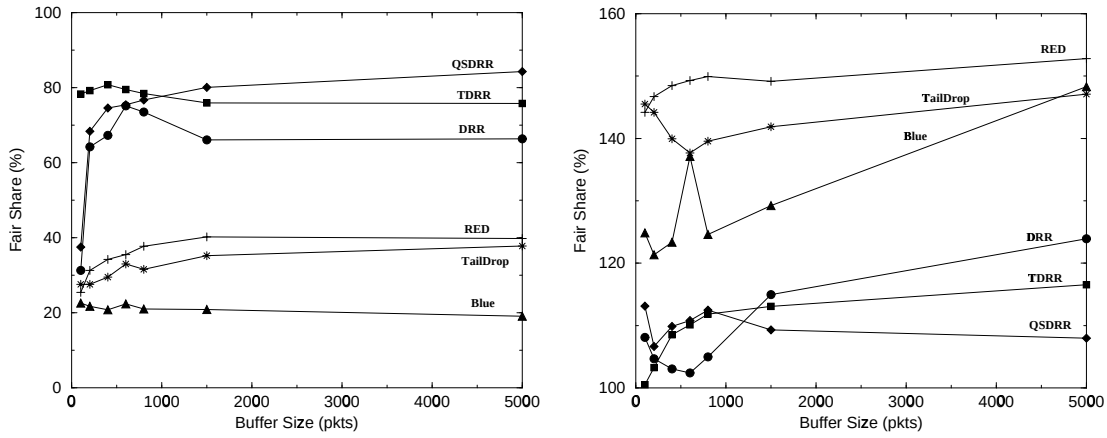
In this configuration, 50 end-to-end TCP Reno flows go over three hops and have an overall round-trip time of 300ms. The cross-traffic on each hop consists of 50 TCP Reno flows with a round-trip time of 100ms (one hop). Figure 9 illustrates the average fair-share goodput received by each set of flows. 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 9(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.

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 9(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.

4.4 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 is-



(a) End-to-end Traffic

(b) Cross Traffic

Figure 9: Fair Share performance of end-to-end and cross traffic flows over a multi-hop path configuration

sue 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 10 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 10(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 200ms 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 10(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.

5 RELATED WORK

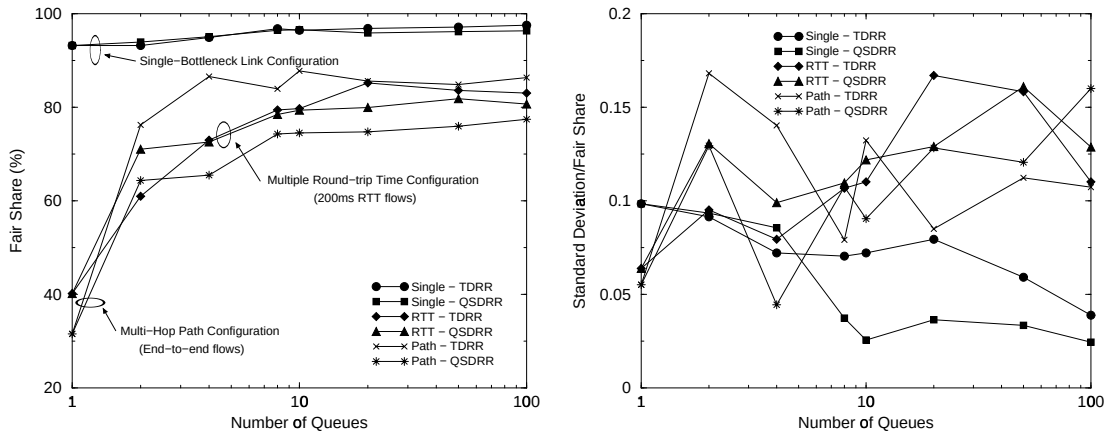
Our DRR-based policies, TDRR and QSDRR, which combine fair queueing and packet discard policies, provide one particular solution for managing very small buffers while maintaining very high link utilization and goodput. In this section, we compare our approach with other related approaches. One thing 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, our studies are also among the first to simulate fairly large bottleneck links with very small buffers (a fraction of the bandwidth-delay product) and different network configurations for TCP flows. Earlier studies were limited to small bottleneck link capacities with buffers equal to the bandwidth-delay product.

5.1 Fair Queueing Algorithms

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 [9] (also known as Weighted Fair Queueing [10]), VirtualClock [11], Stochastic Fairness Queueing [8], Self-Clocked Fair Queueing [12], Weighted Round Robin [13], Deficit Round Robin (DRR) [7] and Frame-based Fair Queueing [14]. We chose DRR due to its simplicity and ease of implementation in hardware.

5.2 FRED

One proposal for using RED mechanisms to provide fairness is Flow-RED (FRED) [15]. 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



(a) Fair Share Percentage

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

Figure 10: Performance of TDRR and QSDRR for a buffer size of 1000 packets, with varying number of buckets

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. 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.

5.3 Self-Configuring RED

Self-configuring RED [16] is a proposal for an adaptive RED policy that can self-parameterize given different congestion types. This policy is similar to Blue, where 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.

5.4 TCP with per-flow queueing

Another proposal for managing TCP buffers is using frame-based fair-queueing [14] with longest queue or random discard policy [17]. 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.

6 CONCLUSION

This paper 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. We also showed that our algorithms perform well even when memory is limited and we have to aggregate multiple sources into one queue.

References

[1] E. Hashem, "Analysis of random drop for gateway congestion control", Tech. Rep. LCS TR-465, Laboratory for Computer Science, MIT, 1989.

[2] Robert Morris, "Scalable TCP Congestion Control", in *IEEE INFOCOM 2000*, March 2000.

[3] S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance", *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, Aug. 1993.

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

[5] C. Villamizar and C. Song, "High Performance TCP in ANSNET", *Computer Communication Review*, vol. 24, no. 5, pp. 45–60, Oct. 1994.

[6] W. Feng, D. Kandlur, D. Saha, and K. Shin, "Blue: A New Class of Active Queue Management Algorithms", Tech. Rep. CSE-TR-387-99, University of Michigan, Apr. 1999.

[7] M. Shreedhar and George Varghese, "Efficient Fair Queueing using Deficit Round Robin", in *ACM SIGCOMM '95*, Aug. 1995.

[8] P. McKenney, "Stochastic Fairness Queueing", *Internetworking: Research and Experience*, vol. 2, pp. 113–131, Jan. 1991.

[9] 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.

[10] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm", *Internetworking: Research and Experience*, vol. 1, no. 1, pp. 3–26, 1990.

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

[12] S. Golestani, "A self-clocked fair queueing scheme for broadband applications", in *IEEE INFOCOM 1994*, Apr. 1994.

[13] 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*, vol. 9, pp. 1265–1279, Oct. 1991.

[14] 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.

[15] Dong Lin and Robert Morris, "Dynamics of Random Early Detection", in *ACM SIGCOMM '97*, Sept. 1997.

[16] W. Feng, D. Kandlur, D. Saha, and K. Shin, "A Self-Configuring RED Gateway", in *IEEE INFOCOM 1999*, Mar. 1999.

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