Report Number: WUCSE-2003-41

2003-05-19

Intelligent Packet Discard Policies for Improved TCP Queue Management

Anshul Kantawala and Jonathan S. Turner

Recent studies have shown that suitably-designed packet discard policies can dramatically improve the performance of fair queueing mechanisms in internet routers. The Queue State Deficit Round Robin algorithm (QSDRR) preferentially discards from long queues, but introduces hysteresis into the discard policy to minimize synchronization among TCP flows. QSDRR provides higher throughput and much better fairness than simpler queueing mechanisms, such as Tail-Drop, RED and Blue. However, because QSDRR discards packets that have previously been queued, it can signficantly increase the memory bandwidth require-ments of high performance routers. In this paper, we explore alternatives to QSDRR that provide comparable performance,... **Read complete abstract on page 2.**

Intelligent Packet Discard Policies for Improved TCP Queue Management

Anshul Kantawala and Jonathan S. Turner

Complete Abstract:

Recent studies have shown that suitably-designed packet discard policies can dramatically improve the performance of fair queueing mechanisms in internet routers. The Queue State Deficit Round Robin algorithm (QSDRR) preferentially discards from long queues, but in-troduces hysteresis into the discard policy to minimize synchronization among TCP flows. QSDRR provides higher throughput and much better fairness than simpler queueing mech-anisms, such as Tail-Drop, RED and Blue. However, because QSDRR discards packets that have previously been queued, it can signficantly increase the memory bandwidth require-ments of high performance routers. In this paper, we explore alternatives to QSDRR that provide comparable performance, while allowing packets to be discarded on arrival, saving memory bandwidth. Using ns-2 simulations, we show that the revised algorithms can come close to matching the performance of QSDRR and substantially outperform RED and 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 the revised algorithms, compared to 40% under RED and Blue. We observe a similar improvement in fairness for long multi-hop paths competing against short cross-traffic paths. We also show that these algorithms can provide good performance, when each queue is shared among multiple flows.

2003-41

Intelligent Packet Discard Policies for Improved TCP Queue Management

Authors: Kantawala, Anshul; Turner, Jonathan

May 19, 2003

Abstract: Recent studies have shown that suitably-designed packet discard policies can dramatically improve the performance of fair queueing mechanisms in internet routers. The Queue State Deficit Round Robin algorithm (QSDRR) preferentially discards from long queues, but introduces hysteresis into the discard policy to minimize synchronization among TCP flows. QSDRR provides higher throughput and much better fairness than simpler queueing mechanisms, such as Tail-Drop, RED and Blue. However, because QSDRR discards packets that have previously been queued, it can signficantly increase the memory bandwidth requirements of high performance routers. In this paper, we explore alternatives to QSDRR that provide comparable performance, while allowing packets to be discarded on arrival, saving memory bandwidth. Using ns-2 simulations, we show that the revised algorithms can come close to matching the performance of QSDRR and substantially outperform RED and Blue. Given a traffic mix of TCP flows with different round-trip times, longer round-trip time flows achieve 80%

Intelligent Packet Discard Policies for Improved TCP Queue Management

Anshul Kantawala
anshul@arl.wustl.edu
Jonathan Turner
jst@arl.wustl.edu

WUCSE-2003-41

May 19, 2003

Department of Computer Science and Engineering
Campus Box 1045
Washington University
One Brookings Drive
St. Louis, MO 63130-4899

Abstract

Recent studies have shown that suitably-designed packet discard policies can dramatically improve the performance of fair queueing mechanisms in internet routers. The Queue State Deficit Round Robin algorithm (QSDRR) preferentially discards from long queues, but introduces hysteresis into the discard policy to minimize synchronization among TCP flows. QSDRR provides higher throughput and much better fairness than simpler queueing mechanisms, such as Tail-Drop, RED and Blue. However, because QSDRR discards packets that have previously been queued, it can signficantly increase the memory bandwidth requirements of high performance routers. In this paper, we explore alternatives to QSDRR that provide comparable performance, while allowing packets to be discarded on arrival, saving memory bandwidth. Using ns-2 simulations, we show that the revised algorithms can come close to matching the performance of QSDRR and substantially outperform RED and 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 the revised algorithms, compared to 40% under RED and Blue. We observe a similar improvement in fairness for long multi-hop paths competing against short cross-traffic paths. We also show that these algorithms can provide good performance, when each queue is shared among multiple flows.

This work is supported in part by DARPA Grant N660001-01-1-8930

Intelligent Packet Discard Policies for Improved TCP Queue Management

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

Abstract

Recent studies have shown that suitably-designed packet discard policies can dramatically improve the performance of fair queueing mechanisms in internet routers. The Queue State Deficit Round Robin algorithm (QSDRR) preferentially discards from long queues, but introduces hysteresis into the discard policy to minimize synchronization among TCP flows. QSDRR provides higher throughput and much better fairness than simpler queueing mechanisms, such as Tail-Drop, RED and Blue. However, because QSDRR discards packets that have previously been queued, it can signficantly increase the memory bandwidth requirements of high performance routers. In this paper, we explore alternatives to QSDRR that provide comparable performance, while allowing packets to be discarded on arrival, saving memory bandwidth. Using ns-2 simulations, we show that the revised algorithms can come close to matching the performance of QSDRR and substantially outperform RED and 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 the revised algorithms, compared to 40% under RED and Blue. We observe a similar improvement in fairness for long multi-hop paths competing against short cross-traffic paths. We also show that these algorithms can provide good performance, when each queue is shared among multiple flows.

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. 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. Although Blue does improve over RED in certain scenarios, its parameters are also sensitive to different congestion conditions and network topologies.

In our previous study, we investigated 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 small.* This allows queueing delays to be kept low.
- *Insensitivity to operating conditions and traffic.* This reduces the need to tune parameters, or compromise on performance.
- *Fair treatment of different flows.* This should hold regardless of differences in round-trip delay or number of hops traversed.

In [7, 8] we 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.

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. While per-flow queues have been historically viewed as too expensive to implement, continuing technology advances have cut the costs to negligible levels. Indeed, by enabling the use of smaller memory sizes for buffering packets, per-flow queues can actually reduce costs and at the same time cut network queueing delays.

In our prior work [7, 8], we proposed and evaluated two different packet dropping algorithms: Throughput DRR (TDRR) and 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, the focus of this paper is to investigate buffer management algorithms that can *intelligently* drop incoming packets during congestion without maintaining an ordered list of queues. Our new algorithms meet all of the objectives outlined above and using ns-2 simulations, we show that they deliver significant performance improvements over the existing methods. We also show that the resuls obtained are comparable to what we can achieve using QSDRR, without wasting memory bandwidth and the need to sort queues based on their length.

The rest of the paper is organized as follows. Section 2 discusses the implementation drawbacks of QSDRR and TDRR. Section 3 describes the new packet drop methods investigated here. Section 4 documents the configurations used for the simulations and the parameters used for evaluating our algorithms. Section 5 compares the performance results of the proposed dynamic threshold multi-queue algorithms against QSDRR, RED, Blue and Tail-Drop for both long-lived and short burst TCP traffic and Section 6 concludes the paper.

2. 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 [7]). 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 [7] 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.

3. 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 [9]. In [9], 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

$$Q_i(t) \geq \alpha \times F(t) \tag{1}$$

where $F(t)$ is the current free buffer space.

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, dicard 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 1: Algorithm for DSDRR

Taking a cue from QSDRR, we add 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 particular queue exceeds the threshold as described in equation 1, it is marked for discard only if the total number of discard queues is less than W. Also, incoming packets are only dropped if the queue is already marked for discard or if the queue exceeds the threshold and the total number of discard queues is less than W. 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 1.

4. 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 a 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.

4.1. Single Bottleneck Link

The network configuration for this set of experiments is shown in Figure 2. $\{S_1, S_2, ...S_N\}$ are the TCP sources connected to the bottleneck link. The destinations, named $\{D_1, D_2, ...D_N\}$, are

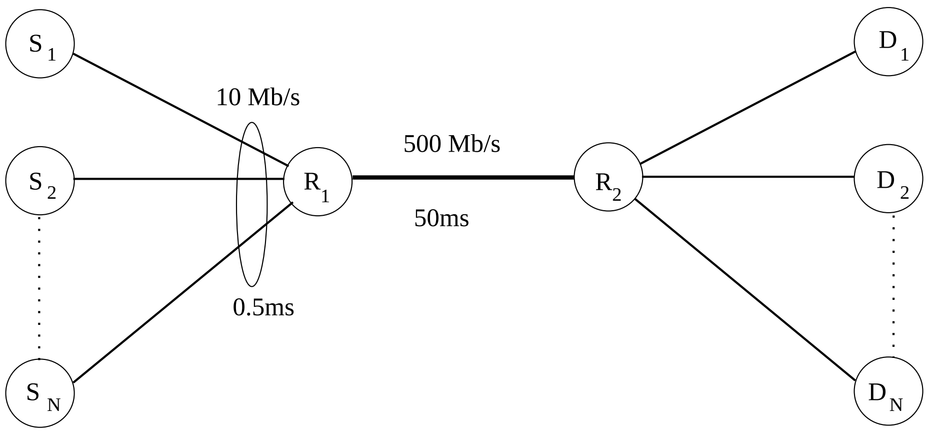


Figure 2: Single Bottleneck Link Network Configuration

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.

4.2. Multiple Roundtrip-time Configuration

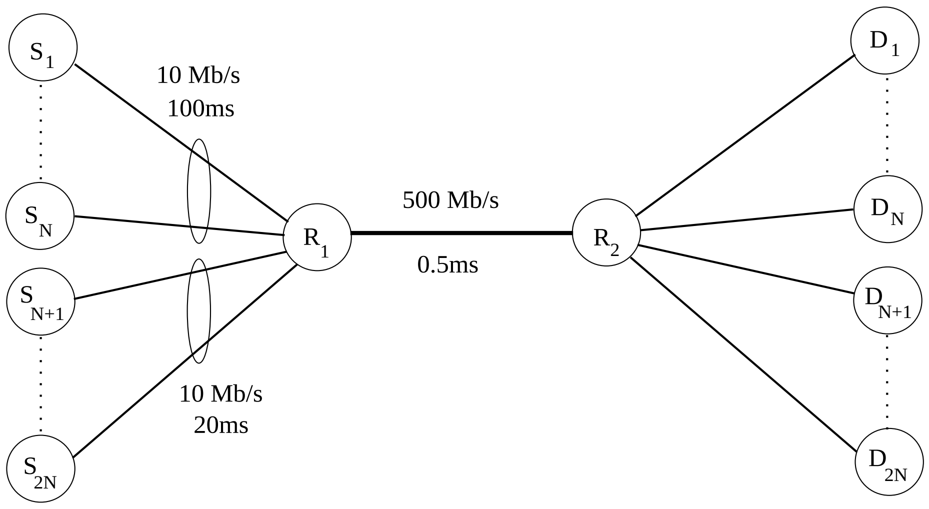


Figure 3: Multiple Roundtrip-time Network Configuration

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 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.

4.3. Multi-Hop Path Configuration

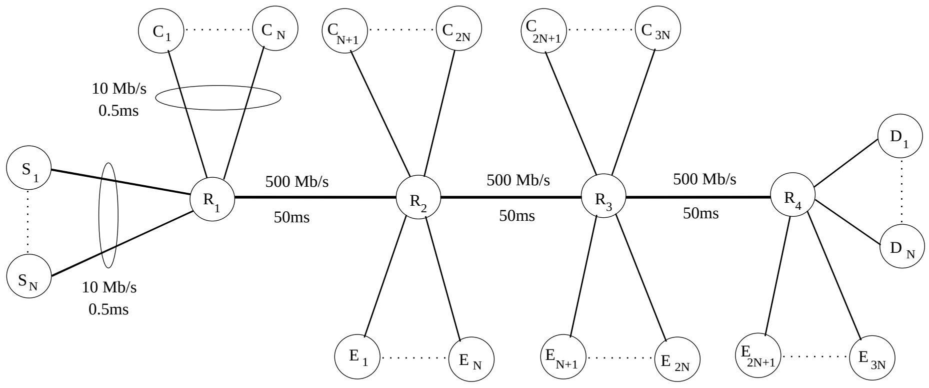


Figure 4: Multi-Hop Path Network Configuration

The network configuration for this set of experiments is shown in Figure 4. In this configuration, we have 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.

5. 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.

5.1. 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

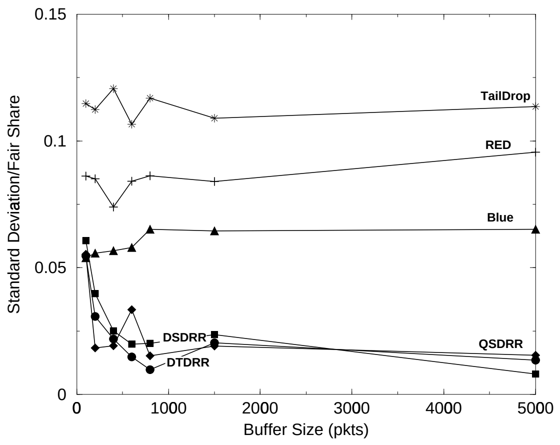


Figure 5: Standard deviation relative to fair-share for long-lived TCP Reno flows over a single-bottleneck link

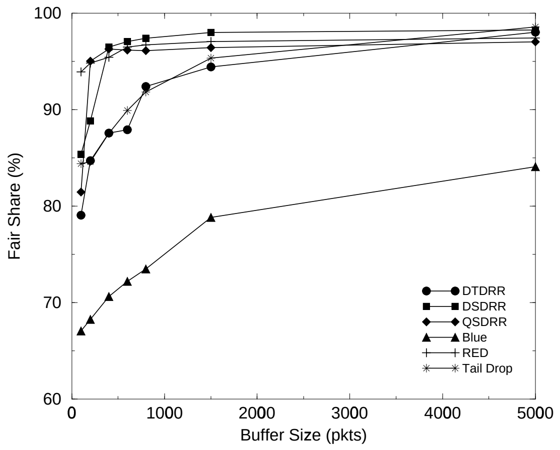
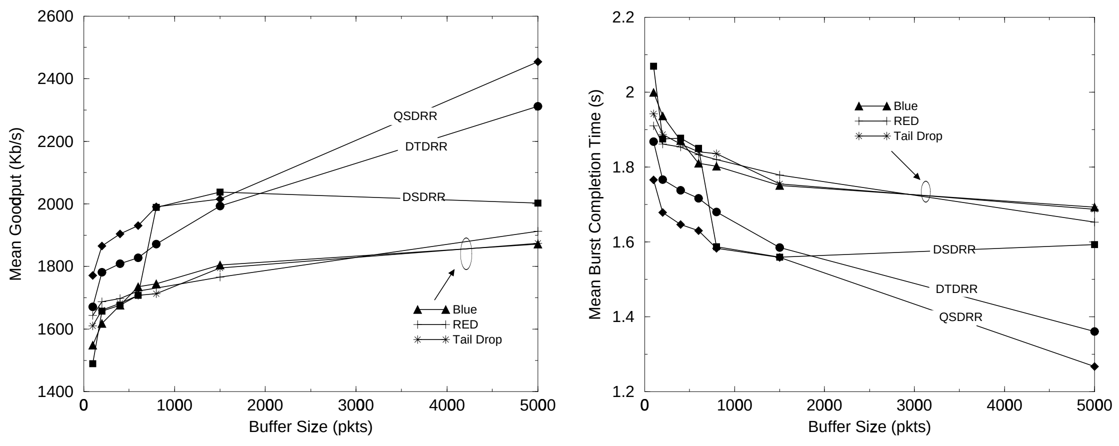


Figure 6: Fair share performance for long-lived TCP Reno flows over a single bottleneck link

Figure 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 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 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



(a) Mean Goodput

(b) Mean Burst Completion Time

Figure 7: Performance of short burst TCP flows over a single bottleneck link

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.

Short burst TCP flows

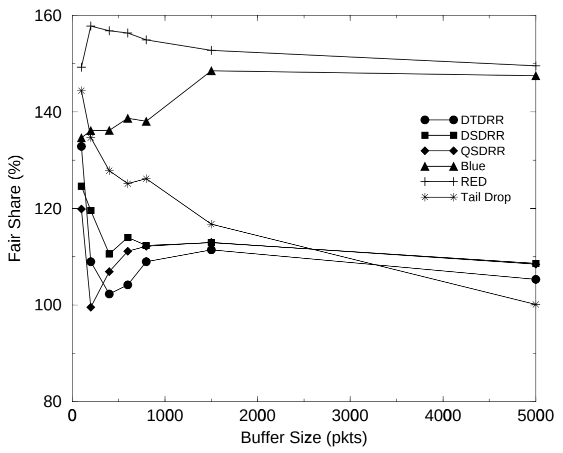
Figure 7(a) shows the mean goodput achieved by the TCP flows and Figure 7(b) shows the mean burst completion times for the flows over a single bottleneck link configuration. *Goodput* is the amount of actual data transmitted excluding retransmissions and duplicates. We notice that Blue, RED and Tail-Drop 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.

5.2. Multiple Round-Trip Time Configuration

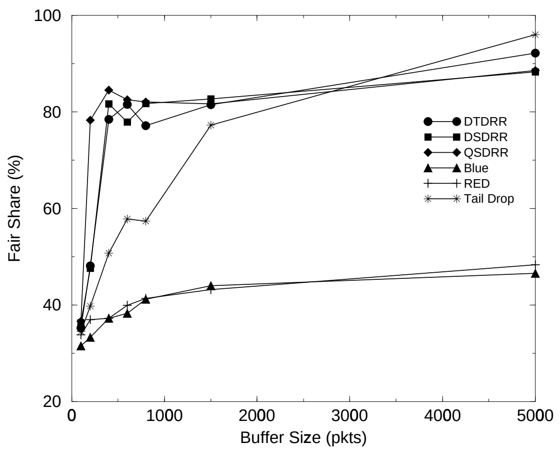
In this configuration, we again use a single bottleneck link, but half the TCP sources have a 40 ms RTT whereas the other half have a 200 ms RTT. For long-lived TCP flows, we use 100 TCP Reno sources and for short burst TCP flows, we use 1000 on-off TCP Reno sources.

Long-lived TCP flows

Figure 8 shows the average fair-share goodput received by TCP flows using the different algorithms. As shown in Figure 8(a), both RED and Blue allow the 40 ms RTT flows to use almost 50% more



(a) Flows with RTT = 40 ms



(b) Flows with RTT = 200 ms

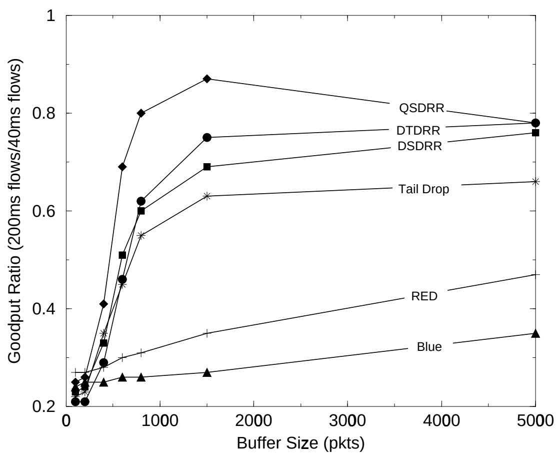
Figure 8: Fair share performance of different RTT long-lived TCP flows over a single bottleneck link

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 8(b), the 200 ms 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.

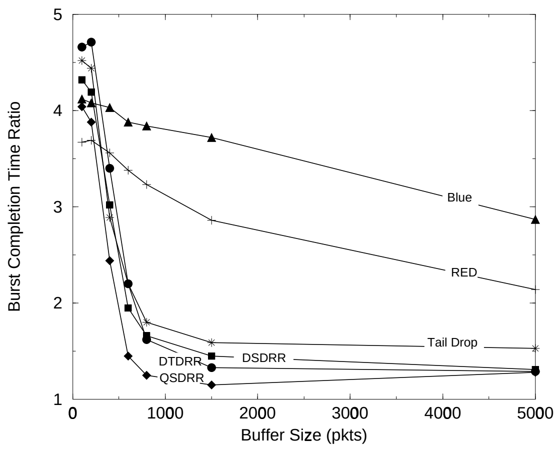
At a very small buffer size of 100 packets, 200 ms RTT flows using DTDRR and DSDRR get about 40% of their fair-share. However, at this buffer size, when all the flows are active, there is only one packet per flow that can be buffered. This causes the poor performance of DTDRR and DSDRR, since it becomes 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 9(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 and RED, showing that for short-lived flows with different round-trip times,



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



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

Figure 9: Performance of short burst TCP flows over a multiple round-trip time configuration

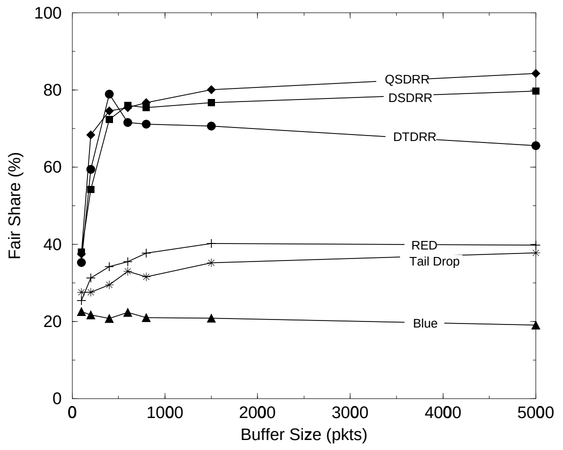
Blue and RED cannot deliver good fair-sharing of the bottleneck bandwidth. Figure 9(b) shows the ratios of burst completion times of the 200 ms round-trip time flows over the 40 ms round-trip time flows. In this case, DTDRR and DSDRR remain close to one for buffer sizes greater than 1000 (which is the 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 QS-DRR for small buffer sizes. At a buffer size of 5000, DTDRR and DSDRR match the performance of QSDRR.

5.3. Multi-Hop Path Configuration

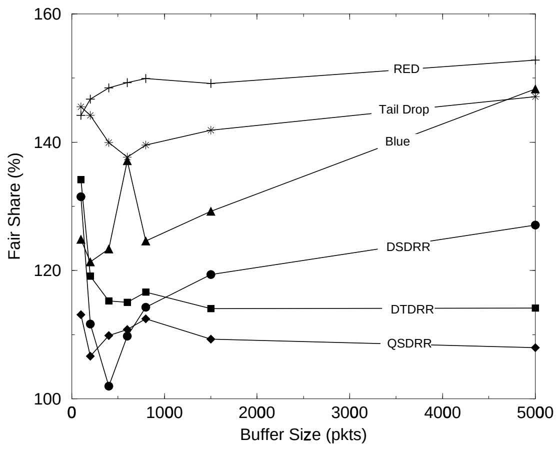
In this configuration, end-to-end TCP Reno flows go over three hops and have an overall round-trip time of 300 ms. The cross-traffic on each hop consists of TCP Reno flows with a round-trip time of 100 ms (one hop). For long-lived TCP flows, we use 50 end-to-end and 50 cross-traffic TCP Reno sources 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 10 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 Tail Drop and *four* times the goodput provided by Blue for end-to-end flows. As shown in Figure 10(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



(a) End-to-end Traffic



(b) Cross Traffic

Figure 10: Fair Share performance of long-lived TCP flows over a multi-hop path configuration

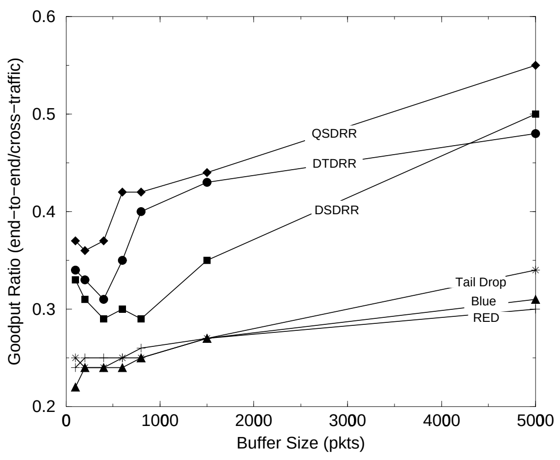
end-to-end flows to achieve close to 80% of their fair share. We notice that in this configuration, DSDRR's performance is very close to QSDRR. Although DTDRR's performance is slightly worse than DSDRR and QSDRR (about 10%) for buffer sizes greater than a 1000 packets, it is still *1.5* times the performance 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.

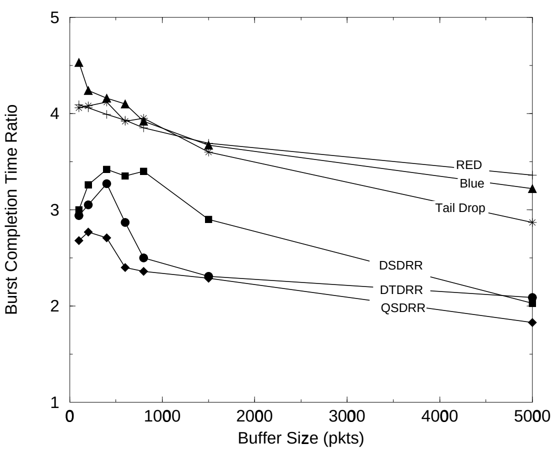
Figure 10(b) shows the average goodput for the cross-traffic flows attached to router R_1. For DTDRR and DSDRR, the cross-traffic takes up the slack in the link and consumes about $115-120\%$ of its fair-share bandwidth. For both RED and Tail Drop, the link utilization is lower and although the end-to-end 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 11(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 DSDRR 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



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



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

Figure 11: Performance of short burst TCP flows over a multi-hop path configuration

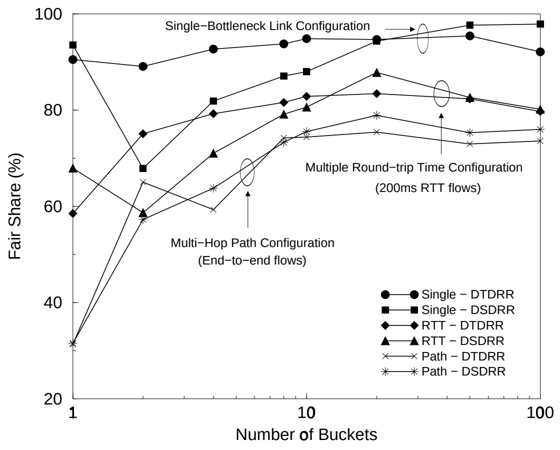
packets, but still outperforms non-DRR policies by $20 - 50\%$. DTDRR and DSDRR are almost *2 times* better than the non-DRR policies for a buffer size of 5000 packets.

Figure 11(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. 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.

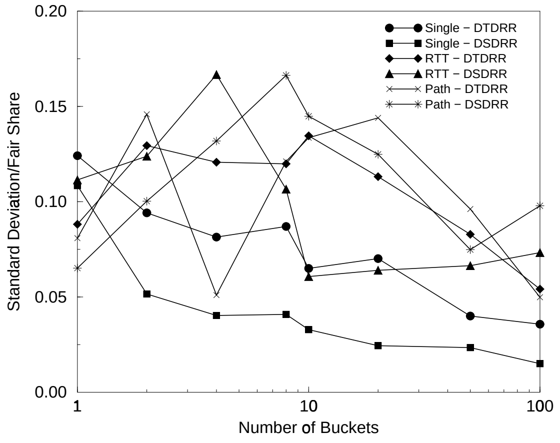
Overall, we notice that DTDRR matches the performance of QSDRR for short burst TCP traffic while DSDRR matches the performance of QSDRR for long-lived TCP traffic. Although, DSDRR is not as good as DTDRR for short burst TCP flows, it still significantly outperforms RED, Blue and Tail-Drop for all configurations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

5.4. 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



(a) Fair Share Percentage



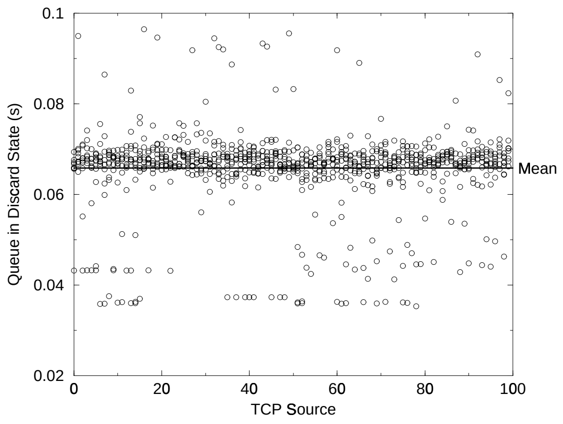
(b) Standard Deviation in goodput relative to fair-share bandwidth

Figure 12: Performance of DTDRR and DSDRR for a buffer size of 1000 packets, with varying number of buckets

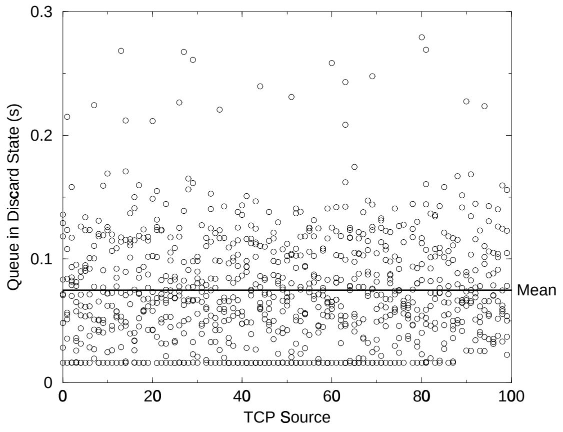
some legitimate concern about the cost associated with using large numbers of queues. One way to reduce the impact of this issue is to allow multiple flows to share a single queue. While this can reduce the performance benefits observed in the previous sections, it may be appropriate to trade off performance 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 12 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 12(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 12(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.



(a) DSDRR



(b) QSDRR

Figure 13: Distribution of queue discard times for DSDRR and QSDRR

5.5. 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.

Table 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

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 13 illustrates the distribution of the time in discard state for each queue under DSDRR and QSDRR for the 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 1 summarizes the statics 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.

6. Conclusion

This paper has demonstrated techniques that can be used to intelligently drop packets on arrival during congestion periods. In previous work, we showed that QSDRR provides higher throughput and much better fairness than simpler queueing mechanisms, such as Tail-Drop, RED and Blue. Because it provides excellent performance, even when buffers are much smaller than the bandwidth-delay product, it also can substantially reduce delays along congested paths. However, because QSDRR discards packets that have previously been queued, it can signficantly increase the memory bandwidth requirements of high performance routers. In this paper, 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 show 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.

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] Anshul Kantawala and Jonathan Turner, “Efficient Queue Management of TCP Flows”, in *SPECTS 2002*, July 2002.

[8] Anshul Kantawala and Jonathan Turner, “Queue Management for Short-Lived TCP Flows in Backbone Routers”, in *High-Speed Networking Symposium, IEEE Globecom ’02*, Nov. 2002.

[9] A. Choudhury and E. Hahne, “Dynamic Queue Length Thresholds for Shared-Memory Packet Switches”, *IEEE/ACM Transactions on Networking*, vol. 6, no. 2, pp. 130–140, Apr. 1998.