Achieving per-flow Queueing Performance without a per-flow Queue

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

WUCSE-2004-44

July 6, 2004

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, since QSDRR needs to maintain a separate queue for each active flow, there is a legitimate concern that it may be too costly for the highest speed links. In previous studies, we have shown that QSDRR can deliver almost the same performance with one-tenth the number queues as flows, if the flows are evenly distributed across the queues. In this paper, we develop and evaluate a flow distribution algorithm using a Bloom filter architecture with dynamic rebalancing. We show that our algorithm significantly reduces the memory requirement compared to maintaining per-flow state and can achieve near optimal flow distribution. Thus, using this algorithm in conjunction with QSDRR, we can achieve the performance of per-flow queueing at a significantly reduced cost.

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

Achieving per-flow Queueing Performance without a per-flow Queue

Anshul Kantawala and Jonathan Turner
Department of Computer Science and Engineering
Washington University in St. Louis
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, since QSDRR needs to maintain a separate queue for each active flow, there is a legitimate concern that it may be too costly for the highest speed links. In previous studies, we have shown that QSDRR can deliver almost the same performance with one-tenth the number queues as flows, if the flows are evenly distributed across the queues. In this paper, we develop and evaluate a flow distribution algorithm using a Bloom filter architecture with dynamic rebalancing. We show that our algorithm significantly reduces the memory requirement compared to maintaining per-flow state and can achieve near optimal flow distribution. Thus, using this algorithm in conjunction with QSDRR, we can achieve the performance of per-flow queueing at a significantly reduced cost.

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]. 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 [7, 8], we investigated how packet schedulers using multiple queues can improve performance over existing methods. Namely, we proposed and evaluated a new packet dropping algorithm called Queue State Deficit Round Robin (QSDRR). The QSDRR algorithm uses Deficit Round Robin (DRR) [9] as the packet scheduler, but introduces hysteresis into the packet discard policy to minimize synchronization among TCP flows. Over a single-bottleneck link, the variance in TCP goodput using QSDRR 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.

One drawback with a fair-queueing policy such as QSDRR 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, it may be appropriate to trade off performance against cost, at least to some extent.

In [7], we investigated the effects on the performance of QSDRR of merging multiple flows into a single queue and observed that QSDRR can deliver almost the same performance with *one-tenth* the number of queues as flows. However, these results assumed that the flows were evenly distributed among all the available queues. This is a best-case scenario and not practical to implement in a real router, since it requires maintaining per-flow state which is approximately just as expensive as per-flow queues. We can avoid maintaining per-flow state by using a hash function computed over a packet's header and map it to a queue. Unfortunately, this may not distribute flows evenly enough to meet fairness objectives. 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.

Thus, the focus of this paper is to develop an algorithm for evenly distributing flows among queues while using much less memory than per-flow queues. We have developed a Bloom filter architecture for counting active flows and dynamically balancing the number of flows per queue to achieve near optimal flow distribution. The rest of the paper is organized as follows. Section 2 briefly describes the QSDRR algorithm and illustrates its property for desynchronizing TCP flows. Section 3 outlines our Bloom filter architecture. Section 4 describes enhancements to the flow distribution algorithm to achieve better dynamic load balancing. Section 5 presents our simulation results and Section 6 concludes the paper.

2. Queue State DRR

Queue State DRR is a packet-scheduling policy that introduces hysteresis into DRR's packet discard 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.
if Q is not defined
  Set Q to current longest queue
else (Q is defined)
  if Q is shorter than all active queues
    Set Q to current longest queue
Drop packets from front of Q
```

Figure 1: Algorithm for QSDRR

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 the bandwidth-delay product size of 417 packets. Figure 2 shows 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 100 multiplied by the flow number starting from 0. Thus, flow 0's queue is raised by 0, flow 1's queue is raised by 100 and so on.

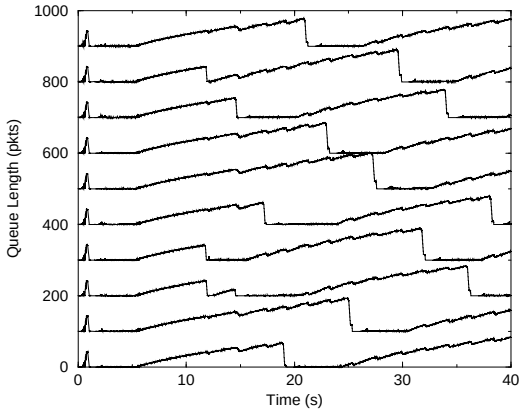


Figure 2: Queue lengths of 10 TCP flows under QSDRR

From Figure 2, we notice that after the initial synchronous drop at 1 second (when all the sources are still in the slow-start, exponential increase phase), QSDRR is able quickly to desynchronize the flows. 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.

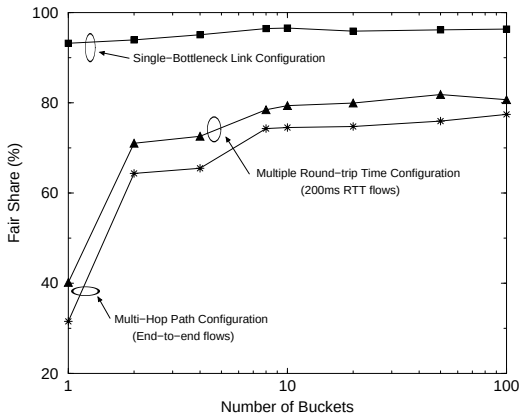


Figure 3: Performance of QSDRR for a buffer size of 1000 packets, with varying number of buckets

Figure 3 illustrates the effect on the goodput received by each flow under different numbers of queues under the assumption that all flows are evenly distributed among the queues. The sources are TCP Reno and the total buffer space is fixed at 1000 packets. In the multiple round-trip time configuration, 50 sources have an RTT of 40 ms and the other 50 sources have an RTT of 200 ms. In the multi-hop path configuration, we have 50 end-to-end sources that travel three hops (three bottleneck links), while there are 50 competing one-hop cross-traffic flows at each bottleneck link. In Figure 3 for the multiple round-trip time configuration and the multi-hop path configuration, we only 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. By *fair share*, we mean the goodput equal to the bottleneck link divided by the number of flows. 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. More information on the network configurations and simulation parameters are in [7]. Table 1 shows the fair share percentages achieved by the longer RTT and multi-hop flows under RED, Blue and Tail Drop.

Table 1: Performance of RED, Blue and Tail Drop for a buffer size of 1000 packets

Configuration	Fair Share Percentage		
	RED	Blue	Tail Drop
Single bottleneck	97	74	92
Multi-RTT (200 ms flows)	41	41	57
Multi-hop (end-to-end flows)	38	21	32

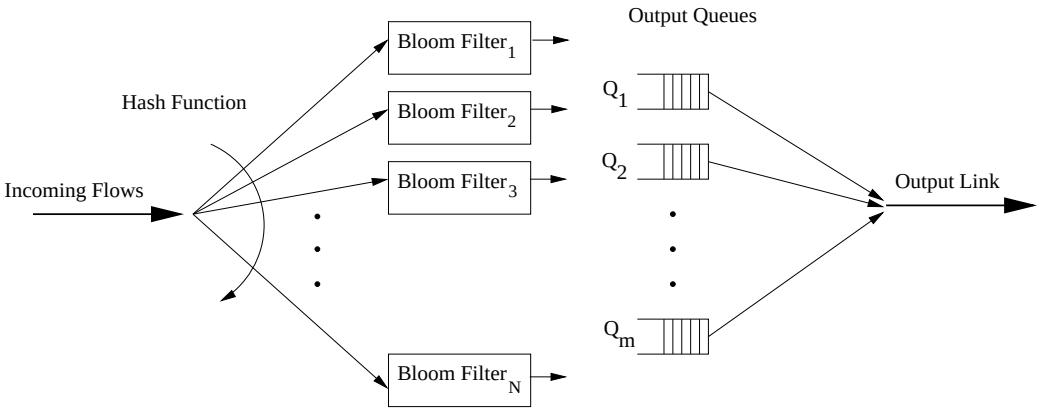


Figure 4: Flow distribution using Bloom Filters

3. Flow Distribution Algorithm

As shown in the previous section, 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 to two parts:

1. Bloom filters [10] which are used to implement approximate flow counters.
2. Policy for distributing flows to queues, based on the approximate flow counts.

3.1. 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].

3.2. Bloom filter architecture

Figure 4 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.

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

Table 2 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.

3.3. 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: 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.

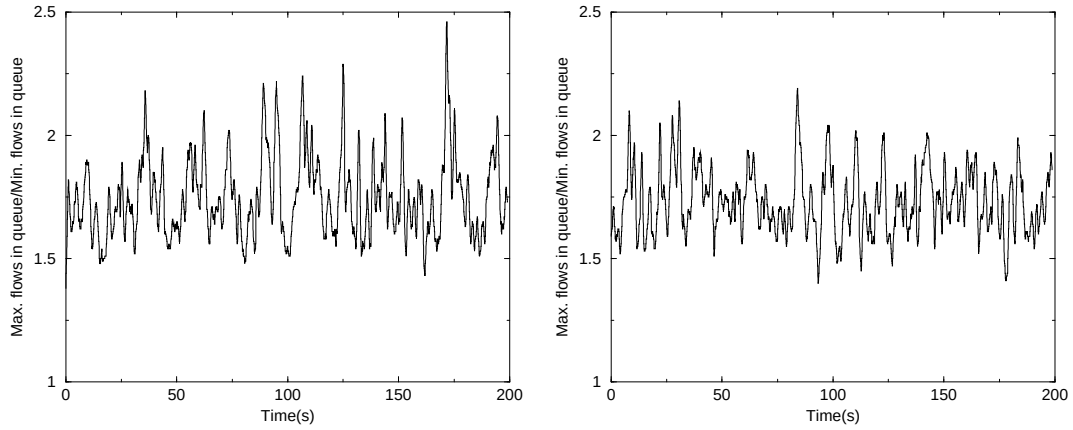
Table 3 shows the ratios of the queue with the most flows over the queue with the least flows. From now on, we refer to this ratio as the **max/min queue ratio**. Ideally, for a perfectly even distribution, this ratio should be one. From Table 3, we observe that by using our Bloom filter architecture 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.

Table 3: 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

4. 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.



(a) Bloom filter - static

(b) Hashing

Figure 6: Max/min queue ratios for Bloom filter architecture without dynamic rebalancing compared with simple hashing

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 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 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 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.

4.1. Single Filter (SF)

```
Set max_count to the flow count of
  the queue with most flows.
Let B[i] be the current Bloom
  filter that is being updated.
Assume B[i] is mapped to Q[i]
Increment Q[i].count
if (Q[i].count > max_count)
  Set Q_min to queue with smallest
    number of flows
  Set B_max to the Bloom filter
    with the most flows mapped
    to Q[i] such that:
  B_max.count < (Q[i].count - Q_min.count)
  Move B_max to Q_min
  Update max_count
end
```

Figure 7: Algorithm for Single Filter

In our first dynamic rebalancing policy, we move only *one* Bloom filter when rebalancing. We maintain an extra state variable, **max_count**, which stores the flow count of the queue that currently holds the largest number of flows. Note that this *count* and all subsequent *counts* mentioned in the paper refer to *Bloom filter counts*. Since we do not keep per-flow state, we cannot use the actual flow count. Whenever a Bloom filter's count is updated, if the new flow count for the queue exceeds **max_count**, we move one Bloom filter from the current queue to the queue with the minimum number of flows. We choose to move the Bloom filter with the maximum number of flows which satisfies the constraint that its flow count is less than the difference in flow counts between the queue

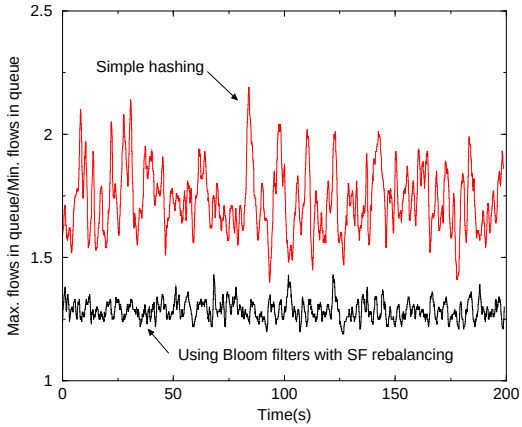


Figure 8: Max/min queue ratios for Bloom filter architecture with dynamic rebalancing compared with simple hashing

with the most flows and the queue with the least flows. A detailed description of this algorithm is presented in Figure 7.

One issue with using Bloom filters as flow counters is updating counts when flows leave. When new flows arrive, they set new Bloom filter bits and thus can be easily detected and the flow count incremented. However, when flows leave, there is no way to detect that event and decrement the Bloom filter's flow count. Thus, as time progresses, with dynamic flows, a Bloom filter's count will start significantly deviating from the real flow count. To compensate for this effect, we periodically clear all the Bloom filter bits and flow counts. New packets arriving from current sources are then used to update the Bloom filters and again obtain an accurate flow count.

Figure 8 compares the time history of the max/min queue ratio using the SF algorithm against the one obtained using simple hashing. We used the same simulation parameters as described above (1,000 flows, 200 Bloom filters, and 20 queues). The destination address hold time was exponentially distributed with a mean of 2 seconds and the Bloom filters were periodically cleared and updated every 1 second. We observe that even though we restrict the number of Bloom filters moved to just *one*, we can maintain a near optimal max/min queue ratio which is significantly lower than the one obtained using simple hashing.

Table 4 compares the average max/min queue ratios obtained using SF, static Bloom filter assignments and simple hashing for mean destination address hold times ranging from 0.5 seconds to 5 seconds. The period for clearing and updating Bloom filters was maintained at 1 second for all the simulation runs. We observe that even when sources change their destination addresses very rapidly (twice a second), SF is able to maintain a fairly low average max/min queue ratio of 1.46 compared to 1.71 for simple hashing. For mean address hold time of 5 seconds, it achieves an excellent average max/min queue ratio of 1.21 compared to 1.72 for simple hashing.

Although we try to minimize the Bloom filter moves under SF, Table 5 shows that we still do end up moving a fairly large number of filters during a simulation run of 200 seconds. In our SF algorithm, we move Bloom filters from the queue with the maximum number of flows. Since this queue holds the most flows, it is likely to be the longest queue and will be selected as the *drop*

Table 4: Max/min queue ratios for Bloom filters with static and dynamic rebalancing vs. simple hashing

Mean Address Hold Time (s)	Average max/min queue ratio		
	SF	Static	Hash
0.5	1.46	1.77	1.71
1.0	1.37	1.77	1.73
2.0	1.29	1.75	1.73
5.0	1.21	1.72	1.72

Table 5: Bloom filter moves using SF during a 200 second simulation run

Mean Address Hold Time (s)	Bloom filter moves
0.5	28791
1.0	20952
2.0	13590
5.0	6789

queue by QSDRR during congestion. Given that QSDRR keeps dropping packets from the drop queue until it is the smallest, there is a good likelihood of finding Bloom filters attached to the maximum queue that have **no** packets queued. Moving these filters which have no packets queued would not incur any overhead during rebalancing. Thus, we use this property of QSDRR in a new rebalancing algorithm that attempts to maintain a near optimal max/min queue ratio while incurring no overhead for moving Bloom filters.

4.2. Multiple Filters (MF)

This algorithm is similar to SF, except that, during rebalancing, we only move Bloom filters that have *no* packets queued, from the queue with the maximum number of flows to the queue with the minimum number of flows. A detailed description of the MF algorithm is presented in Figure 9.

4.3. Single or Multiple Filters (SMF)

During periods of severe congestion on a link, we may not be able to find any Bloom filters with no packets queued during rebalancing. Thus, we propose a hybrid approach that combines the SF and MF policies. In this approach, we first try to find and move Bloom filters with no packets queued which are mapped to the queue with the most flows. If we do not find *any* Bloom filters to move, we fall back to the SF policy and move the largest Bloom filter whose flow count is less than the difference in flow counts between the maximum flows queue and the minimum flows queue. A detailed description of SMF is presented in Figure 10.

```
Set max_count to the flow count of
  the queue with most flows.
Let B[i] be the current Bloom
  filter that is being updated.
Assume B[i] is mapped to Q[i]
Increment Q[i].count
if (Q[i].count > max_count)
  Set Q_min to queue with smallest
    number of flows
  Set S to the set of Bloom filters
    mapped to Q[i] which have
    no packets queued
  Move N Bloom filters from set S
    to Q_min with the constraint:
  sum_{i=0}^{N} B[i].count < (Q[i].count - Q_min.count)
  Update max_count
end
```

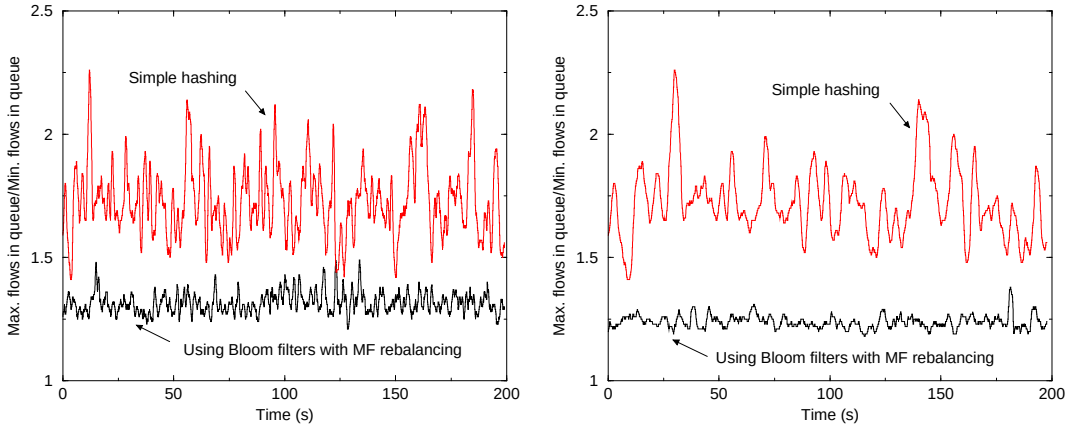
Figure 9: Algorithm for Multiple Filters

```
Set max_count to the flow count of
  the queue with most flows.
Let B[i] be the current Bloom
  filter that is being updated.
Assume B[i] is mapped to Q[i]
Increment Q[i].count
if (Q[i].count > max_count)
  Set Q_min to queue with smallest
    number of flows
  Set S to the set of Bloom filters
    mapped to Q[i] which have
    no packets queued
  if (S is non-empty)
    Use MF policy
  else
    Use SF policy
end
```

Figure 10: Algorithm for Single or Multiple Filters

5. Results

To evaluate our dynamic rebalancing policies, MF and SMF, 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 1 second 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. We changed the link bandwidths to achieve the desired load for the results presented below.



(a) Mean destination address hold time = 2s

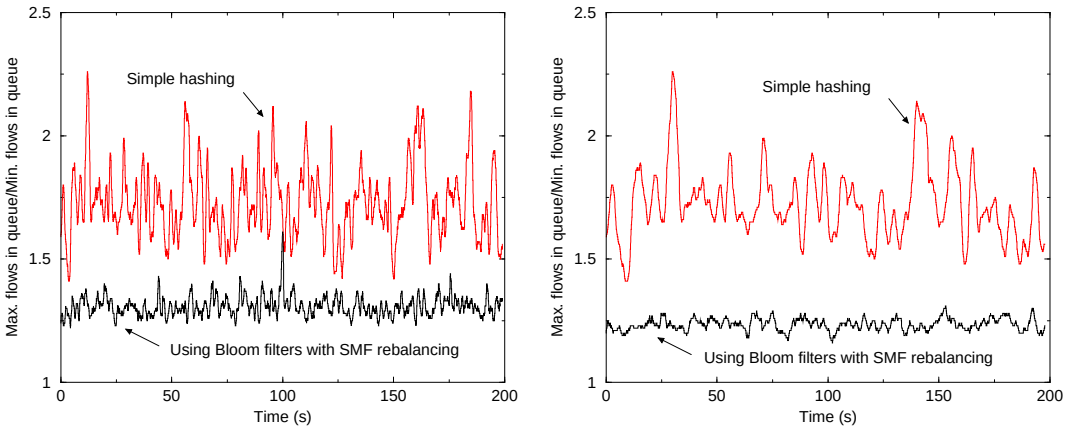
(b) Mean destination address hold time = 5s

Figure 11: Max/min queue ratios for MF dynamic rebalancing compared with simple hashing with 85% load

Figure 11 compares the time history of the max/min queue ratio using the MF algorithm against the one obtained using simple hashing for mean destination address hold times of 2 and 5 seconds and a load of 85%. From these graphs, we observe that by using MF dynamic rebalancing, even at relatively high loads of 85%, we can achieve very low max/min queue ratios while incurring *no* overhead for moving Bloom filters.

Table 6 compares the average max/min queue ratios obtained using MF and SMF dynamic rebalancing policies and simple hashing for mean destination address hold times ranging from 0.5 seconds to 5 seconds for a 85% load. We observe that MF is able to deliver the same performance as SMF without incurring the additional overhead of moving packets between queues. Table 7 illustrates that at a load of 85%, the SMF policy moves very few Bloom filters which have packets queued and thus incurs a negligible overhead.

Figure 12 compares the time history of the max/min queue ratio using the SMF algorithm against the one obtained using simple hashing for mean destination address hold times of 2 and 5 seconds and a 2:1 overload. From these graphs, we observe that even at a 2:1 overload, SMF is able to effectively rebalance the flows while minimizing the overhead of moving packets.



(a) Mean destination address hold time = 2s

(b) Mean destination address hold time = 5s

Figure 12: Max/min queue ratios for SMF dynamic rebalancing compared with simple hashing with 2:1 overload

Table 6: Max/min flows queue ratios for Bloom architecture with MF and SMF dynamic rebalancing vs. simple hashing with a 85% load

Mean Address Hold Time (s)	Average max/min queue ratio		
	MF	SMF	Hash
0.5	1.47	1.47	1.71
1.0	1.40	1.39	1.73
2.0	1.32	1.31	1.73
5.0	1.24	1.24	1.72

Table 8 compares the average max/min queue ratios obtained using MF and SMF dynamic rebalancing policies and simple hashing for mean destination address hold times ranging from 0.5 seconds to 5 seconds for a 2:1 overload. At such a high overload, MF is not able to find enough Bloom filters with no packets queued and thus cannot perform an effective rebalancing of flows. However, SMF is able to deliver the same low max/min ratios as in the 85% load case. Also, we notice from Table 9, that by using the hybrid approach, SMF is able to reduce the Bloom filter moves by more than 20% compared to the standard SF policy.

From the above results we can conclude that the Bloom filter architecture with the SMF policy is able to deliver near optimal flow distribution for different load distributions while minimizing the overhead incurred due to moving packets between queues.

Table 7: Bloom filter moves using SMF during a 200 second simulation run with a 85% load

Mean Address Hold Time (s)	Bloom filters moved	Packets moved
0.5	75	75
1.0	38	39
2.0	13	13
5.0	9	9

Table 8: Max/min queue ratios for Bloom architecture with MF and SMF dynamic rebalancing vs. simple hashing with a 2:1 overload

Mean Address Hold Time (s)	Average max/min queue ratio		
	MF	SMF	Hash
0.5	1.76	1.47	1.71
1.0	1.75	1.39	1.71
2.0	1.77	1.31	1.73
5.0	1.75	1.23	1.72

Table 9: Bloom filter moves using SMF during a 200 second simulation run with a 2:1 overload

Mean Address Hold Time (s)	Bloom filters moved	Packets moved
0.5	25076	328224
1.0	17875	240829
2.0	11477	161074
5.0	5834	90117

6. Conclusion

In this paper, we developed and evaluated a flow distribution algorithm using a Bloom filter architecture with dynamic rebalancing. We proposed and evaluated two dynamic rebalancing policies, MF and SMF which achieve near optimal flow distribution for varied load conditions while reducing the overhead of moving packets across queues. Thus, using our algorithm significantly reduces the memory requirement compared to maintaining per-flow state while achieving near optimal flow distribution. Thus, using this algorithm in conjunction with QSDRR, we can achieve the performance of per-flow queueing at a significantly reduced cost.

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

[10] B. Bloom, "Space/time tradeoffs in hash coding with allowable errors," *Communications of ACM*, vol. 13, no. 7, pp. 422–426, July 1970.

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