Weighted Fair Queuing with Differential Dropping

Feng Lu, Geoffrey M. Voelker, and Alex C. Snoeren
UC San Diego

***Abstract*—Weighted fair queuing (WFQ) allows Internet operators to define traffic classes and then assign different bandwidth proportions to these classes. Unfortunately, the complexity of efficiently allocating the buffer space to each traffic class turns out to be overwhelming, leading most operators to vastly overprovision buffering—resulting in a large resource footprint. A single buffer for all traffic classes would be preferred due to its simplicity and ease of management. Our work is inspired by the approximate differential dropping scheme but differs substantially in the flow identification and packet dropping strategies. Augmented with our novel differential dropping scheme, a shared buffer WFQ performs as well or better than the original WFQ implementation under varied traffic loads with a vastly reduced resource footprint.**

I. Introduction

Internet traffic demand is increasing at a staggering rate, forcing providers to carefully consider how they apportion capacity across competing users. This resource allocation challenge is complicated by the diversity of traffic classes and their vastly different importance and profitability. Indeed, many Internet Service Providers have taken steps to curb the bandwidth dedicated to traffic perceived as low value (i.e., peer-to-peer file sharing, over-the-top multimedia streaming) while attempting to improve performance for high-value traffic classes such as provider-supported streaming media, VoIP, and games. These demands are not unique to the consumer Internet, and similar requirements have begun appearing in multi-tenant datacenter [1] and other enterprise environments.

In many instances, network operators do not wish to set explicit bandwidth guarantees, but instead express relative importance among traffic classes. Such proportional resource allocation is a well-studied problem, and a variety of techniques are available in the literature—and many are even deployed in commercial routers and switches. Indeed, algorithms like Weighted Fair Queuing (WFQ) [2] and Deficit Round Robin [3] have been proposed, analyzed, and implemented in many vendors' standard access router and switch offerings. Unfortunately, sophisticated traffic management mechanisms like WFQ are not available in all switch and router types, largely because of their complexity and significant resource footprint at scale (both in terms of CPU and buffer space). In an effort to combat this complexity, researchers have proposed packet dropping schemes built around the idea that packets from different flows should be dropped differently to achieve the fairness goal. Packet dropping schemes are generally much cheaper to implement and a single FIFO queue is often sufficient. However, they are only able to provide coarse-grained fairness in the long term, which may not be sufficient for many applications.

Perhaps most importantly, however, even those proportional bandwidth allocation mechanisms that are available and appropriate are not always employed, due in part to their configuration complexity as well as the potential for router forwarding performance degradation. This is especially true when links and switches are operating near capacity: parameter tuning becomes increasingly critical when buffer space is tight, and poorly configured traffic management schemes may perform worse than a simple FIFO queue under heavy load. Hence, we argue that there is a need for a robust, easy-to-configure mechanism that scales to a large number of flows and traffic classes without prohibitively large buffering requirements.

In this paper, we propose an Enhanced Weighted Fair Queuing (EWFQ) scheme, which combines the accuracy of scheduling algorithms like WFQ with the decreased resource footprint of dropping-based active queue management schemes. Critically, our system is largely self-tuning, and does not require the demand-specific buffer configuration that WFQ does, nor the parameter adjustment that traditional active queue management schemes require.

II. Preliminaries and Related Work

Weighted Fair Queuing [2] provides weighted fair sharing at the packet level, as does Deficit Round Robin (DRR) [3]. Both of these algorithms have been deployed in a wide range of commercially available hardware. However, they are often criticized for their intricate packet scheduling which makes it difficult to implement them at high line rates. Moreover, in their pure forms, these schemes require per-flow queues to ensure perfect isolation in the face of ill-behaved flows. Properly sizing these per-flow queues—which requires users to partition the total buffer space—is error prone and cumbersome.

To reduce implementation complexity and the necessity of keeping per-flow queues, a large body of work has explored so-called active queue management (AQM) [4], [5], [6], [7], [8]. By relaxing the packet-by-packet fairness requirement, these solutions utilize cheap flow-rate estimation methods and employ a single FIFO queue. As a result, they achieve approximate fairness in the long run, but provide no guarantees about the bandwidth sharing among short-lived flows.

For example, CHOKe [4] makes the observation that flow rate is proportional to the number of packets in the queue. Upon the arrival of a new packet, it compares this new packet with a randomly chosen packet from the queue, and both packets are dropped if they are from the same flow. SRED [6] employed a packet list to keep a counter for each flow, and packets are dropped according to their respective flow counts. However, the flow estimation methods employed by

CHOKe and SRED turn out to be too rough in many instances. Theoretically, flow rate could be accurately estimated if all the incoming packets are recorded, but keeping state for all flows proves to be overly complex. Both AFD [5] and AF-QCN [7] use random packet sampling to reduce the overhead in the hope that large flows can still be identified and accurately estimated. Unfortunately, it is not easy to find a sampling probability where rate estimation is sufficiently accurate while sampling costs remains low. Therefore, Kim *et al.* [8] propose to use sample-and-hold techniques to accurately estimate flow rates. However, their queue management scheme and packet drop methodology require accurate per-flow RTT estimates.

III. Overall System Design

In this work, we aim to incorporate the benefits of active queue management into weighted fair queuing while preserving the accuracy of WFQ. In particular, we propose a *shared-buffer* WFQ implementation, removing the principle limitation to supporting large numbers of traffic classes [7], as well as freeing the network operator from configuring per-flow queue sizes. In our design, the shared queue is managed by a probabilistic packet dropper in a fashion similar to AFD, but packet scheduling is still based on WFQ—as opposed to FIFO in AFD—maintaining packet-by-packet fairness for both short and long-lived flows.

A. Weighted Fair Dropping

The most basic issue with AFD that we must address is to provide support for *weighted* fair dropping, where each flow has a weight w_i associated with it. We observe that this modification is trivial if one follows a particular weight assignment scheme.

Suppose there are n flows in the system with weights $w_1, w_2, \cdots, w_n$; we desire that flow i will on average achieve a data rate of $Cw_i/\sum w_j$ when all flows are backlogged and C is the link rate. If we normalize the weights such that $\min(w_1, \cdots, w_n) = 1$ then a flow with weight w_i can be treated as w_i concurrent flows in the standard fair queuing system [3], where the effective data rate of a "transformed flow" from flow i is $\frac{r_i}{w_i}$. The per-flow fair sharing in WFQ can be expressed as $r^i_{fair} = min(r_i, r_{fair}w_i)$, where r_{fair} is the solution to $C = \sum_i min(r_i, r_{fair}w_i)$. Once r_{fair} and r_i are known, the differential dropping probability applied to flow i can be expressed as $d_i = (1 - \frac{r_{fair}}{r_i})_+$,[1] where the expression x_+ denotes $\max(0, x)$.

We now revisit the original AFD scheme [5] using this normalization. Instead of uniformly sampling each packet with probability $\frac{1}{s}$, packets belonging to flow i are sampled with probability $\frac{1}{s \cdot w_i}$. The estimated rate for each flow is then $\frac{r_i}{w_i}$, and the rest of the AFD system does not need to be altered: m_{fair} is estimated in the same manner except it now has a different meaning, and the dropping probability is based on the normalized m_i and m_{fair} (see Table I for the definition of each parameter).

[1]This applies to non-responsive flows only.

B. Identification of High-bandwidth Flows

The first step in active queue management is determining the offered load. In particular, we must detect high-bandwidth flows and estimate their sending rates. In this work, we apply the sample-and-hold technique proposed by Estan and Varghese [9]. The advantage of sample-and-hold is that it decouples the flow identification from flow estimation. In basic sample-and-hold, the probability of flow detection is linearly proportional to the normalized flow size (which is equivalent to flow rate since the measurement interval is fixed). Hence, in order to reliably track large bandwidth flows, many small flows are likely to be unnecessarily included. A number of variants [10], [11] have been proposed to reduce this identification probability for small flows. The basic idea underlying them all is to maintain a randomly sampled packet cache of the arriving packets. A flow is considered identified only when a packet from it is sampled *and* the packet matches a packet randomly drawn from the packet cache. Considering the fact that the packet cache is randomly updated, this match-hit strategy effectively makes the flow detection probability proportional to the square of the flow size. We generalize the match-hit approach by extending it to multiple hit stages so that the detection probability for small flows can be arbitrarily scaled down. In our multi-stage match-hit scheme, k packet matches must happen before an entry for a flow is created.

C. Differentiated Drop Treatment

Once the flows have been characterized, it remains to drop packets that exceed a flow's fair share. Our solution is based on an observation made by Jacobson *et al.* [12]: Consider a bottleneck ingress router with a single long-lived TCP flow. When the size of the input queue is small, the average packet drop rate of this TCP flow can be approximated by $1/(\frac{3}{8}P^2)$, where P is the bandwidth(R) $\cdot$ delay(RTT) product in packets and the effective data rate is $0.75 * R$. We find that when the packet drop rate is decreased to $1/(\frac{2}{3}P^2)$, the TCP flow achieves the full bandwidth R (proof elided due to space constraints). Happily, our result also matches the well-known relationship between window size and drop rate, $w = \sqrt{\frac{2}{3q}}$, where w is the window size and q is the average packet drop rate [13]. Based on our earlier description, it can be seen that $r_{fair} \cdot t_i$ is roughly the number of bytes transmitted by a fair-bandwidth shared flow over a measurement interval t_i. Given a specific value of RTT, then the packet drop rate for a fair bandwidth flow would be: $1/(\frac{2}{3}(\frac{r_{fair} * RTT}{size_p})^2)$. Therefore, we propose the following packet drop rate heuristic:

$$d_i = \frac{1}{0.66 \cdot \left(\frac{m_{fair}}{size_p} \cdot \frac{RTT}{t_i}\right)^2}, \quad \text{if } m_{fair} < m_i \qquad (1)$$

where $size_p$ is the average packet size of a TCP flow and m_i is the observed actual byte count over a measurement interval. Often, the value of RTT is unknown and may not be static over the lifetime of a TCP connection. We defer our discussion of how to handle an unknown RTT to the next section. Finally, for a flow of weight w_i, its corresponding bytes sent during

the measurement interval t_i will be $m_{fair}.w_i$. Hence, if the observed bytes sent is substituted in, the actual drop rate will be $d_i' = d_i/(w_i)^2$.

IV. Design and Implementation Details

In this section, we discuss our prototype implementation and justify our design choices for various system parameters.

A. Aggregated Flows

Class-based WFQ extends basic WFQ by aggregating multiple responsive or non-responsive flows into a single user-defined class. It is not hard to see that classes comprised of non-responsive flows should be treated the same as a single non-responsive flow. Unfortunately, this does not hold for classes that are made up of responsive flows. For the single TCP flow case, when a packet is dropped, it immediately responds by halving its sending window. While for the aggregated TCP flow case, a single packet drop only causes one of the flows to halve its sending window, and the aggregated window size is not reduced by half. Obviously, the aggregated case would acquire more bandwidth under the same packet drop rate. We find that with n aggregated TCP flows, the packet drop rate needs to scale quadratically so that both cases enjoy the same bandwidth share. (Proof omitted due to space constraints.) For example, if there are n TCP flows in a single aggregated class, the corresponding drop rate would be $n^2 d_i$.

B. Unknown RTTs

As developed in Section III-C, the packet drop rate depends on the RTT of the flow. However, in practice the RTT of a flow is generally not known *a priori* and could vary from time to time depending on the path connecting the two end hosts. Hence, we seek to remove the need to include RTT in the dropping equation. Suppose the actual RTT for a TCP flow i is RTT_i, but RTT_o is used in the packet drop probability in Equation 1 instead. During the next measurement interval, rather than m_{fair} bytes, $m_{fair}\frac{RTT_o}{RTT_i}$ bytes are transmitted. Therefore, if the actual number of bytes sent is known, the RTT ratio $\frac{RTT_o}{RTT_i}$ can be estimated and tracked. At a first glance, it might appear that additional state variables are needed to estimate the RTT ratio. In fact, if a flow is sending near m_{fair} or more bytes in an interval, this flow will most likely be identified by the sampling and holding hashmaps, which suggests that the actual bytes sent is already kept in the system. To send more than m_{fair} bytes and be identified, RTT_o has to be larger than RTT_i. Fortunately, in reality, a maximum RTT value can generally be safely assumed.

Thus, for each identified flow, an additional variable $\gamma_i = \frac{RTT_o}{RTT_i}$ is kept. At the end of each interval, the value of γ_i is updated as follows:

$$\gamma_i^{new} = \max(0.5\gamma_i^{old} + 0.5\gamma_i^{old}\frac{m_i}{\bar{m}_{fair}}, 1)$$

where m_i is the actual bytes sent and $\bar{m}_{fair}$ is the average of m_{fair} during the measurement interval. If the flow is not found in the holding hashmap, we reset the value of γ_i to 1. Another advantage of setting RTT_o to be the maximum RTT is that γ_i is always lower bounded by 1. Finally, the drop probability d_i is written as:

$$d_i = \frac{\gamma_i^2 n_i^2}{w_i^2} \cdot \frac{1}{0.66 \cdot \left(\frac{m_{fair}}{size_p} \cdot \frac{RTT_o}{t_i}\right)^2} \quad (2)$$

where n_i is the number of active TCP flows in a responsive flow class. Note that if n_i is unknown and set to 1 in Equation 2, then γ_i could be used to directly estimate the value of $n_i \cdot \frac{RTT_o}{RTT_i}$.

Parameter	Meaning
p	byte sampling probability
w_i	assigned weight for flow i
n	total number of flows
m_i	bytes countered for flow i during an interval
L	packet length
d_i	packet dropping parameter
m_{fair}	bytes counted assumed a fair flow
T	flow identification threshold
p_c	packet cache insertion probability
α, β	m_{fair} update parameters
k	number of match-hit stages
R	aggregated traffic load
f_s	queue length sampling frequency
RTT_o	estimated maximum RTT value

TABLE I: System parameters.

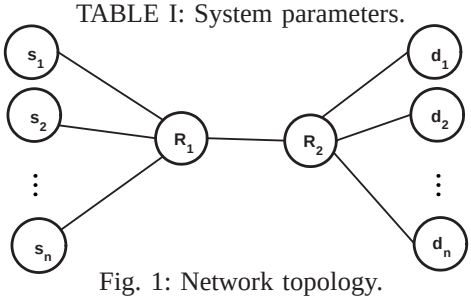


Fig. 1: Network topology.

V. Experimental Results

We use the $ns2$ simulator to evaluate our proposed design. Table I summarizes all the system parameters used in our implementation.The following configuration parameters have worked well for our simulation experiments: $\alpha = 0.85$, $\beta = 0.9$, $f_s = 100$, $p_c = 0.1$, $k = 1$, $q_{target} = 0.5 \cdot q_{size}$, and measurement interval = $1s$. The parameters p and T are chosen in such a way that the probability of miss identification for a flow whose rate exceeds the fair share is less than 10%.

We first demonstrate that no single optimal buffer allocation decision matches all offered traffic load under WFQ. In contrast EWFQ shares the same physical buffer among all flow classes, dynamically repartitioning the buffer to consistently achieve excellent bandwidth sharing under a variety of traffic loads without operator intervention. Further, even if the offered traffic load is known in advance, it is still challenging to determine the optimal buffer allocation decision under WFQ. Any sub-optimal allocation would inevitably result in degraded buffer utilization. The second part of this section then compares the relative performance of both WFQ and EWFQ under

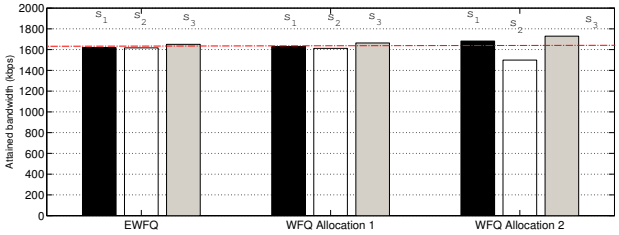


Fig. 2: Per flow bandwidth for the "Uniform" network.

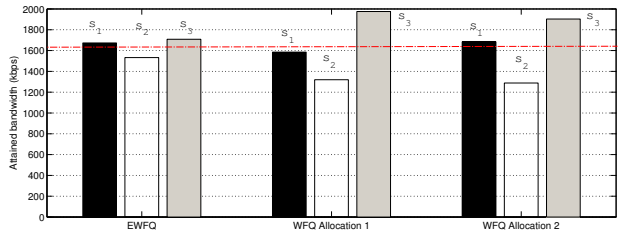


Fig. 3: Per flow bandwidth for the "Varied" network.

a range of buffer sizes, showing that EWFQ can achieve more with much less buffer memory.

We use the network topology in Figure 1 with $n = 3$ nodes. Under the first scenario "Uniform", links s_1, s_2 to R_1 have a delay of $10ms$. In the second scenario "Varied", the links have delays 50 and $100ms$, respectively. Delays for all other links in both scenarios are $1ms$. In addition, the bottleneck connection between R_1 and R_2 is $5Mbps$ while all other links have a bandwidth of $10Mbps$. Both s_1 and s_2 send TCP traffic, and s_3 sends UDP traffic. All traffic flows have equal weight and the total queue size is $30KB$. For WFQ, we use two buffer allocations, $(10, 10, 10)KB$ ("Uniform allocation") and $(15, 5, 10)KB$ ("Varied allocation"). EWFQ dynamically partitions a single $30KB$ buffer and does not require a specific allocation to be pre-configured.

Figure 2 summarizes the results for the "Uniform" network scenario. It shows the attained bandwidth for each flow under EWFQ and under the two different WFQ buffer allocations. EWFQ achieves a fair bandwidth allocation (fairness index 0.9999), as does the "Uniform allocation" for WFQ (fairness 0.9998) which matches the network scenario. The "Varied allocation" for WFQ also performs well (fairness 0.9963) even on this scenario. Figure 3 shows similar results but for the "Varied" network scenario. EWFQ requires no operator configuration, yet still achieves excellent bandwidth fairness (0.9978). The WFQ buffer allocations, however, perform less well. The fairness among flows using the "Varied allocation" for WFQ, even though it is tuned for this scenario, suffers in comparison (0.9765). And the "Uniform allocation" for WFQ performs slightly worse yet (0.9733).

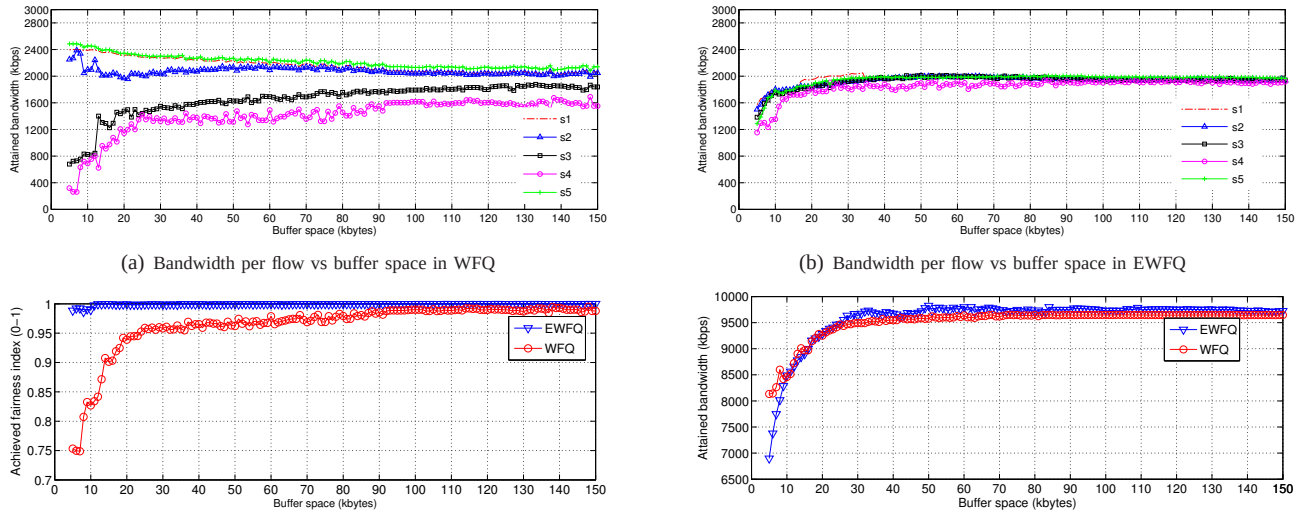
With a shared buffer implementation, EWFQ performs consistently well under both scenarios, performs slightly better than WFQ in the "Uniform" scenario, and significantly outperforms WFQ in the "Varied" scenario. The purpose here is not to show how much EWFQ could outperform WFQ. Rather, we want to illustrate that there does not exist a single choice of static WFQ buffer allocations which match all different scenarios. Instead, dynamically adjusting allocations as with EWFQ can effectively adapt to changes in network conditions without reconfiguration by the operator.

We then evaluate the impact of buffer size on the relative behavior of EWFQ and WFQ, and, at least in one traffic instance, the amount of buffer space required by WFQ to achieve similar performance as EWFQ. In this experiment the topology and workload remain fixed, while we vary the amount of buffer memory available to balance per-flow bandwidth, total bandwidth, and fairness among the flows. We configure the topology in Figure 1 with five source nodes, four nodes sending TCP traffic and one sending UDP traffic. The bottleneck link between R_1 and R_2 has a bandwidth of $10Mbps$, and the links connecting s_i to R_i are $2.5Mbps$. The bandwidth for all other links is $10Mbps$. The link delays are $2, 20, 100, 200, 50ms$, respectively, for links connecting $s_1, \cdots, s_5$ to R_1. We create 5 equally-weighted flow classes with each flow from s_i belonging to one class. Since all flows have the same weight, we allocate the buffer space equally for each flow under WFQ (one-fifth of the buffer size per flow). We then vary the total buffer memory from $5KB$ to $150KB$ in increments of $1KB$.

Figures 4(a) and 4(b) show the attained bandwidth per flow over the range of buffer sizes, and Figure 4(c) shows the Jain fairness index across the flows. Particularly for small to medium buffer sizes, WFQ struggles to distribute bandwidth equally across flows. Small queue sizes induce frequent packet drops, placing long-RTT TCP flows (s_3, s_4) at a disadvantage as they recover from loss. The attained bandwidth for these flows are well below their fair share (at very small queue sizes, small RTT flows achieve $10\times$ the bandwidth of the long RTT flows). Since WFQ is a work-conserving scheduling algorithm, the small RTT flows (s_1, s_2) are able to take more than their fair share. As more buffer space is available, flows gradually converge towards their fair share. However, the fairness index does not start to approach 1.0 until the buffer size is larger than $90KB$ and, even with a buffer size of $150KB$, there is still considerable variation among the flows.

In contrast EWFQ compensates for differences in RTT among flows and drops packets from small RTT flows more frequently, resulting in an extremely fair bandwidth share among competing flows with different RTTs. As buffer sizes increase the attained bandwidth curves consistently overlap each other, showing that EWFQ is able to attain higher bandwidths in a fair manner. Even with a buffer size as low as $10KB$ EWFQ achieves a fairness index close to 1.0.

Figure 4(d) shows the total aggregate bandwidth across flows as a function of buffer size. Total bandwidth increases as buffer size increases because fewer packets are dropped from the queues. When the buffer space is very small ($\leq 10kbytes$), the total attained bandwidth under EWFQ is smaller than WFQ. There are two reasons for this difference. The first is that, when buffer space is small, the AQM mechanism becomes less stable and may lead to oscillations where queue occupancy goes back and forth between 0–100% [14]. When oscillation occurs, bandwidth performance degrades. The other



(a) Bandwidth per flow vs buffer space in WFQ

(b) Bandwidth per flow vs buffer space in EWFQ

(c) Fairness vs buffer space

(d) Total bandwidth vs buffer space

Fig. 4: Comparison between WFQ and EWFQ on fairness and attained bandwidth with respect to buffer space.

reason is that EWFQ by design avoids the situation where the queue is completely full. Once the queue is full, packets are forced to drop rather than be shaped by the drop probability. An extremely shallow buffer results in a very small targeted queue length. Therefore, although EWFQ achieves over 0.98 fairness at very small buffer sizes, it does so at the penalty of lower aggregated bandwidth. As buffer space gradually increases, the AQM mechanisms becomes more stable and EWFQ matches and then exceeds WFQ in aggregated bandwidth attained at $30KB$. At this buffer size, EWFQ can apportion buffer space across flows according to their RTTs such that all flows obtain their desired bandwidth share. WFQ partitions the buffer into separate per-flow queues, though, and cannot reallocate excess capacity for use by other flows.

In summary, considering that WFQ approaches the same degree of fairness as EWFQ at $150KB$, while they both approach maximum total bandwidth at $30KB$, for this particular traffic instance WFQ needs $5\times$ the buffer space of EWFQ to achieve the same aggregated bandwidth and fairness.

VI. Conclusion

Weighted fair queueing, being a close approximation of generalized processor sharing, can provide near-optimal fair bandwidth sharing, bounded delay, and traffic isolation. Unfortunately, it is also very difficult to allocate the appropriate amount of buffer space for each flow, particularly when the offered traffic load is unknown *a priori*. Further, no single optimal configuration works well for all types of traffic workloads and, even when the offered load is known, the optimal buffer allocation decision remains challenging.

In this paper we have introduced EWFQ, an AQM mechanism that is able to drop packets differentially based on its corresponding flow weight and type. For workloads consisting of a variety of traffic flows, simulation results indicate that EWFQ can attain near perfect fairness among competing flows while requiring much less buffer space than a WFQ that uses separate queues. In addition, EWFQ frees operators from having to specify buffer allocations for traffic classes by dynamically sharing the same buffer among all flows.

Acknowledgment

We are indebted to Barath Raghavan and George Varghese who first exposed us to the buffer-management problem in WFQ implementations.

References

[1] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha, "Sharing the Data Center Network," in *Proc. USENIX NSDI*, 2011.

[2] J. C. R. Bennett and H. Zhang, "WF2Q: worst-case fair weighted fair queueing," in *Proc. IEEE INFOCOM*, 1996, pp. 120–128.

[3] M. Shreedhar and G. Varghese, "Efficient fair queueing using deficit round robin," in *Proc. ACM SIGCOMM*, 1995, pp. 231–242.

[4] R. Pan, B. Prabhakar, and K. Psounis, "CHOKe - a stateless active queue management scheme for approximating fair bandwidth allocation," in *Proc. IEEE INFOCOM*, vol. 2, 2000, pp. 942–951.

[5] R. Pan, L. Breslau, B. Prabhakar, and S. Shenker, "Approximate fairness through differential dropping," *SIGCOMM Comput. Commun. Rev.*, vol. 33, pp. 23–39, April 2003.

[6] T. Ott, T. Lakshman, and L. Wong, "SRED: Stabilized RED," in *Proc. IEEE INFOCOM*, vol. 3, Mar. 1999, pp. 1346–1355.

[7] A. Kabbani, M. Alizadeh, M. Yasuda, R. Pan, and B. Prabhakar, "AF-QCN: Approximate fairness with quantized congestion notification for multi-tenanted data centers," in *Proc. High Performance Interconnects (HOTI)*, Aug. 2010, pp. 58–65.

[8] J. Kim, H. Yoon, and I. Yeom, "Active queue management for flow fairness and stable queue length," *IEEE Trans. Parallel and Distributed Systems*, vol. 22, no. 4, pp. 571–579, Apr. 2011.

[9] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *Proc. ACM SIGCOMM*, 2002, pp. 323–336.

[10] F. Hao, M. Kodialam, T. Lakshman, and H. Zhang, "Fast, memory-efficient traffic estimation by coincidence counting," in *Proc. IEEE INFOCOM*, vol. 3, Mar. 2005, pp. 2080–2090.

[11] M. Kodialam, T. V. Lakshman, and S. Mohanty, "Runs bAsed Traffic Estimator (RATE): A simple, memory efficient scheme for per-flow rate estimation," in *Proc. IEEE INFOCOM*, 2004.

[12] V. Jacobson, K. Nichols, and K. Poduri, "RED in a different light," Cisco, Tech. Rep., 1999.

[13] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modeling TCP throughput: a simple model and its empirical validation," in *Proc. ACM SIGCOMM*, 1998, pp. 303–314.

[14] V. Firoiu and M. Borden, "A study of active queue management for congestion control," in *Proc. IEEE INFOCOM*, vol. 3, Mar. 2000, pp. 1435–1444.