Improving Individual Flow Performance with Multiple Queue Fair Queuing

Manfred Georg, Christopher Jechlitschek, and Sergey Gorinsky

Fair Queuing (FQ) algorithms provide isolation between packet flows, allowing max-min fair sharing of a link even when flows misbehave. However, fairness comes at the expense of per-flow state. To keep the memory requirement independent of the flow count, the router can isolate aggregates of flows, rather than individual flows. We investigate the feasibility of protecting individual flows under such aggregate isolation in the context of Multiple Queue Fair Queuing (MQFQ), where the router maintains a fixed number of queues and associates multiple queues with each flow. MQFQ places packets in the shortest queue associated with their flow. The... **Read complete abstract on page 2.**

Improving Individual Flow Performance with Multiple Queue Fair Queuing

Manfred Georg, Christopher Jechlitschek, and Sergey Gorinsky

Complete Abstract:

Fair Queuing (FQ) algorithms provide isolation between packet flows, allowing max-min fair sharing of a link even when flows misbehave. However, fairness comes at the expense of per-flow state. To keep the memory requirement independent of the flow count, the router can isolate aggregates of flows, rather than individual flows. We investigate the feasibility of protecting individual flows under such aggregate isolation in the context of Multiple Queue Fair Queuing (MQFQ), where the router maintains a fixed number of queues and associates multiple queues with each flow. MQFQ places packets in the shortest queue associated with their flow. The redundancy of multiple queues allows a flow to transmit at a fair rate even when one of its queues is congested. However, a misbehaving flow is able to acquire a larger than fair share of the bottleneck link capacity. We also discuss important implementation issues such as avoidance of packet reordering.

Improving Individual Flow Performance with Multiple Queue Fair Queuing

Authors: Manfred Georg, Christoph Jechlitschek, and Sergey Gorinsky

Corresponding Author: mgeorg@cse.wustl.edu

Web Page: http://www.cse.wustl.edu/~mgeorg/mqfq/

Abstract: Fair Queuing (FQ) algorithms provide isolation between packet flows, allowing max-min fair sharing of a link even when flows misbehave. However, fairness comes at the expense of per-flow state. To keep the memory requirement independent of the flow count, the router can isolate aggregates of flows, rather than individual flows. We investigate the feasibility of protecting individual flows under such aggregate isolation in the context of Multiple Queue Fair Queuing (MQFQ), where the router maintains a fixed number of queues and associates multiple queues with each flow. MQFQ places packets in the shortest queue associated with their flow. The redundancy of multiple queues allows a flow to transmit at a fair rate even when one of its queues is congested. However, a misbehaving flow is able to acquire a larger than fair share of the bottleneck link capacity. We also discuss important implementation issues such as avoidance of packet reordering.

Improving Individual Flow Performance with Multiple Queue Fair Queuing

Manfred Georg, Christoph Jechlitschek, and Sergey Gorinsky
Applied Research Laboratory
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri 63130–4899, USA
Email: {mgeorg, chrisj, gorinsky}@arl.wustl.edu

***Abstract*— Fair Queuing (FQ) algorithms provide isolation between packet flows, allowing max-min fair sharing of a link even when flows misbehave. However, fairness comes at the expense of per-flow state. To keep the memory requirement independent of the flow count, the router can isolate aggregates of flows, rather than individual flows. We investigate the feasibility of protecting individual flows under such aggregate isolation in the context of Multiple Queue Fair Queuing (MQFQ), where the router maintains a fixed number of queues and associates multiple queues with each flow. MQFQ places packets in the shortest queue associated with their flow. The redundancy of multiple queues allows a flow to transmit at a fair rate even when one of its queues is congested. However, a misbehaving flow is able to acquire a larger than fair share of the bottleneck link capacity. We also discuss important implementation issues such as avoidance of packet reordering.**

I. Introduction

Fair Queuing (FQ) algorithms [1]–[7] dedicate a separate queue to each flow and schedule packets for transmission over a congested link so that every flow receives an average service rate that approximates its max-min fair share of the link capacity [8]. In comparison to the traditional First-In First-Out (FIFO) scheduling of packets, FQ provides a significant degree of isolation between flows. In particular, FQ algorithms reduce queuing delays for flows that consume less than the maximum fair share of the link capacity. Fair queuing exhibits superior resilience against misbehaving flows: if the rate of a User Datagram Protocol (UDP) [9] flow is unfairly high, the excessive transmission does not disrupt well-behaving flows; instead, FQ penalizes the aggressive flow through the accumulation and eventual discard of its packets at the router. Fair queuing also improves the fairness properties of end-to-end congestion control protocols: while the throughput of Transmission Control Protocol (TCP) [10], [11] flows in a network of FIFO routers is inversely proportional to their round trip times [12], [13] and hence not max-min fair, using FQ with sufficient buffers at bottleneck links enables TCP to transmit at max-min fair rates. Unfortunately, the fairness benefits of FQ come at the high price of maintaining per-flow state at the router.

We consider a simpler framework in which memory requirements are constant. This framework encompasses most methods that use a constant number of queues such as Stochastic Fair Queuing (SFQ) [14], Stochastic Fair Blue (SFB) [15], Random Early Detection with Preferential Dropping (RED-PD) [16], and other designs that require only a fixed amount of memory. When the number of flows becomes large, these schemes generally treat multiple flows as a single aggregate; for example, under SFQ multiple flows share one of a fixed number of queues. Although this isolates flow aggregates with similar assurances as provided by FQ to all flows, it offers no such isolation to individual flows within an aggregate. In particular, the throughput of TCP flows within an aggregate depends on the round trip time (RTT) of the flows. More importantly, flows within an aggregate have common queuing delay and loss characteristics. If well-behaving flows are in the same aggregate with a misbehaving flow, the latter can capture most of the link capacity allocated to its aggregate and thereby starve the well-behaving flows.

In this paper, we explore the feasibility of protecting individual flows while only using a constant number of queues. We investigate a technique where the router maintains a fixed number of queues and uses more than one of these queues for each flow. An incoming packet is enqueued in the shortest of the queues associated with its flow. The rationale for allowing a flow to use multiple queues rests on assigning different sets of queues to different flows: if a misbehaving flow fills up its set of queues, each flow that shares a queue with it can still utilize another queue unavailable to the misbehaving flow. We realize this technique in Multiple Queue Fair Queuing (MQFQ) which provides each flow with access to two queues. We also examine variations with more than two queues per flow but find such designs less beneficial because they allow a misbehaving flow access to a larger portion of the link capacity. We evaluate

MQFQ in comparison with SFQ and SFB, demonstrating the benefits of MQFQ. We also consider situations that can lead to packet reordering and present methods for avoiding them.

The rest of the paper is organized as follows. Section II discusses related work on FQ algorithms. Section III presents MQFQ in detail. Section IV discusses methods for avoiding packet reordering in MQFQ. Section V contains a comparative experimental evaluation of MQFQ, SFQ, and SFB. Finally, Section VI concludes the paper with a summary of our findings.

II. Related Work

A wide variety of queuing algorithms have been studied. A class of practical queuing disciplines serve all flows from a single queue. In the most common case, all flows are aggregated into a single FIFO queue with droptail, where arriving packets are discarded whenever the link buffer is full. Other dropping methods include Random Early Detection (RED) [17] and its numerous extensions.

A more complicated approach involves per-flow state. The Fair Queuing (FQ) [1] paradigm uses an independent queue for each flow. FQ guarantees a fair share of the link capacity for each flow but is costly to implement. Also, if the order of serving the queues is chosen carefully, FQ is able to offer low delay guarantees to responsive flows. WDRR [4] is the simplest FQ instance that serves queues in a fixed round-robin order but provides no delay guarantees. Stratified Round Robin [5] addresses this drawback of WDRR. Self-Clocked Fair Queuing (SCFQ) [6] and Worst-case Fair Weighted Fair Queuing (WF2Q) [7] support even tighter delay guarantees at the price of more computation.

There are also a number of hybrid designs which balance quality of service against resource usage. We consider two hybrid schemes, Stochastic Fair Queuing (SFQ) [14] and Stochastic Fair Blue (SFB) [15].

SFQ reduces memory requirements by using a small fixed number of queues. Each flow is mapped to one of the queues and shares it with other flows. Since the aggregation of flows into queues negates per-flow delay guarantees, SFQ serves the queues using round-robin order. SFQ strives to provide statistical fairness and some protection against misbehaving flows. Ideally, each queue hosts the same number of flows. A misbehaving flow has an impact only on flows in its own queue. Furthermore, if all k queues are backlogged, a single flow is capped at $1/k$ of the link capacity.

SFB extends Blue [18], which itself is an enhancement of RED. Blue maintains a single queue with controlled probability of packet discard. The discard probability increases upon queue overflows and decreases when the queue empties. If all flows are responsive, Blue provides early feedback adjusted for the current traffic pattern. SFB extends Blue by adding a Bloom filter to take over calculation of the discard probability. Hence, SFB is a hybrid approach that adds a fixed amount of state to improve fairness among flows. The Bloom filter uses multiple hash functions to assign each packet to several independent bins. Every bin has a fixed size and variable discard probability. An arriving packet is added to all its bins. If a bin overflows, SFB discards the packet and increases the discard probability of the bin. If a bin empties, its discard probability decreases. When SFB places a packet into the output queue, SFB sets the discard probability of the packet to the minimum among the discard probabilities of the packet's bins. The Bloom filter distinguishes between different flows and supports the discard probability computation, which follows the same algorithm as in Blue. A variation of SFB uses two Bloom filters and a separate queue to detect and rate-limit misbehaving flows.

The idea to identify and rate-limit large flows has been further explored in other designs. When the link is congested, CHOKe (CHOose and Keep/Kill) [19] compares an arriving packet with a packet chosen randomly from the single output queue. CHOKe drops both packets if they belong to the same flow. Approximate Fair Dropping (AFD) [20] maintains a history for recent packets to compute drop probabilities for incoming packets. Other proposals rely on sampling or mechanisms similar to Bloom filters [21]. The identify-and-limit approach is effective in blocking large greedy flows, which are easily identifiable. Furthermore, since flow sizes in most traffic patterns follow a heavy-tailed distribution, it is feasible to identify large flows with limited extra space and computation. However, the identify-and-limit approach suffers from the following drawbacks: (1) its effectiveness depends on the traffic pattern: e.g., during coordinated attacks, the number of misbehaving flows that need to be identified and rate-limited might exceed the maximum supported by the algorithm; (2) since identification takes time, rate-limiting kicks in only after some delay; (3) this approach ignores fairness among smaller flows, including milder cheaters that inflate transmission modestly enough to avoid detection. To ameliorate these problems, identify-and-limit schemes can adopt techniques proposed in this paper. In particular, one can identify and rate-limit large flows first and then apply our multiple-queue proposal.

III. Aggregated Queuing

Serving aggregated flows from a fixed number of queues offers effective balance between resource consumption and performance. It improves performance above that experienced with a single queue but fails to isolate flows completely. While perfect isolation is sacrificed for reduced state, our paper explores a possibility of outperforming SFQ using the same amount of memory. We propose Multiple Queue Fair Queuing (MQFQ), an SFQ enhancement that allows flows to utilize several queues. Instead of a single hash function as in SFQ, MQFQ uses multiple hash functions to determine a set of queues for a flow. When a packet arrives, MQFQ applies all hash functions to the source and destination IP addresses and port numbers from the packet header to compute potential queues. MQFQ enqueues the packet into the queue with the soonest service. If the currently used queue grows large, the flow switches to another of its queues. The rationale for making multiple queues available to a flow is an expected decrease in the number of flows that a misbehaving flow can starve completely. A side effect of this design choice is the ability of a misbehaving flow to flood multiple queues and thereby acquire an unfairly high throughput. Hence, there exists a trade-off between the degree of extra capacity surrendered to a misbehaving flow and the number of flows starved by the misbehaving flow. As we show later, two queues per flow constitute the best resolution of the trade-off. Unless explicitly stated otherwise, our subsequent references to MQFQ denote its instance with two hash functions.

As in SFQ, MQFQ serves all queues using round-robin order. We implemented MQFQ based on the SFQ default in ns-2. However, we improved the hash function implementation because the default hashed consecutive numbers into adjacent queues. In the process, we learned that design and implementation of a simple independent hash function is deceptively difficult but nevertheless crucial because the hashing affects performance significantly. We chose a cyclic redundancy check (CRC) checksum on a mix of data bits and bits from a randomly generated reproducible stream. To generate independent hash functions, we intersperse the data and random bits in varying ratios. As in the ns-2 default, our SFQ shares memory among all queues but allows a queue to grow beyond its fair memory share only if free buffer space is at least the queue size plus two more packets.

The MQFQ queues used by a flow can be thought of as a single virtual queue. In comparison to SFQ where two flows either interfere or not, MQFQ virtual queues reduce the likelihood of complete interference but create a new possibility of less damaging partial interference. In SFQ with k queues, probability P_{SFQ} that flow x interferes with flow y is the same as the probability of hashing into the same queue, i.e.,

$$P_{\text{SFQ}} = \frac{1}{k}$$

with a good hash function. In MQFQ, complete interference occurs when flow y shares all its queues

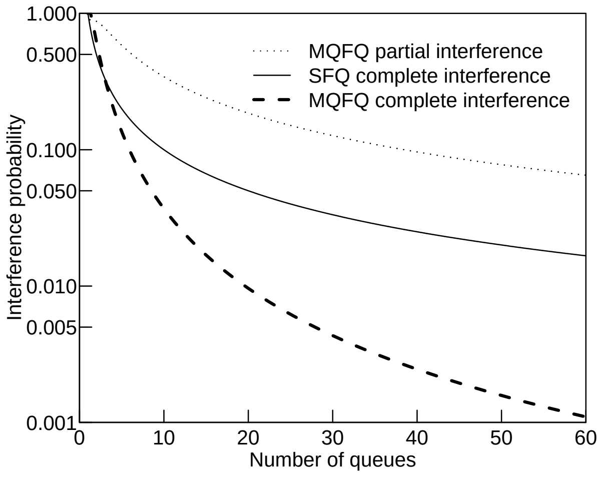

Fig. 1. Probabilities of flow interference in MQFQ and SFQ.

with flow x, i.e., either the two hash functions of y collide and map into a queue of x, or x hashes into both separate queues of y. Then, probability $P^{\text{comp}}_{\text{MQFQ}}$ of complete interference in MQFQ is equal to:

$$P^{\text{comp}}_{\text{MQFQ}} = \frac{1}{k^2}\left(2-\frac{1}{k}\right)+\frac{2}{k^2}\left(1-\frac{1}{k}\right)=\frac{4}{k^2}-\frac{3}{k^3}.$$

Partial interference occurs when flow y shares at least one of its queues with flow x, i.e., either the two hash functions of y collide and map into a queue of x, or x hashes into at least one of two separate queues of y. Probability $P^{\text{part}}_{\text{MQFQ}}$ of partial interference in MQFQ equals:

$$P^{\text{part}}_{\text{MQFQ}} = \frac{1}{k^2}\left(2-\frac{1}{k}\right)+\frac{4}{k}\left(1-\frac{1}{k}\right)^2=\frac{4}{k}-\frac{6}{k^2}+\frac{3}{k^3}.$$

Figure 1 depicts P_{SFQ}, $P^{\text{comp}}_{\text{MQFQ}}$, and $P^{\text{part}}_{\text{MQFQ}}$. As expected all three probabilities decrease when the number of queues grows. While SFQ interference and MQFQ partial interference diminish similarly as $O(\frac{1}{k})$, complete interference in MQFQ decreases much faster as $O(\frac{1}{k^2})$.

IV. Reordering

Since not all packets are placed in the same queue, the possibility of packet reordering within the same TCP flow must be carefully considered. A packet reordering in TCP will trigger duplicate acknowledgments. After three consecutive duplicate acknowledgments TCP assumes that congestion has caused a loss and scales back its transmission rate. This negatively impacts performance and should be avoided.

By always picking the shortest available queue we ensure that later packets cannot pass earlier packets from the same flow, hence reordering can be avoided. However, edge cases in implementation makes it tricky to precisely define the shortest queue. We assume a WDRR [4] implementation in which queues are serviced in round robin order. When each queue is serviced, a deficit representing how many bytes the queue is allowed to transmit is incremented by a weight parameter which signifies the priority of the queue. The queue is then allowed to transmit as many bytes as are in the deficit, and the deficit is decremented appropriately. The next queue is then serviced. The original queue will not be serviced again until all other queues have been serviced, we call this a round. The weight parameter is not particularly relevant to MQFQ; however, its inclusion in this discussion does not present any fundamental complications. Like WDRR, we require the weight parameter for every queue to be larger than the maximum packet size, so that at least one packet is sent every time a non-empty queue is serviced.

When determining which queue is the shortest, the deficit and weight of the queue must be taken into account. The number of rounds that elapse before a packet placed into a particular queue is serviced, which we call Rounds Before Service (RBS), is calculated by the following equation.

$$RBS = max\left(\left\lceil \frac{q+p-d-w}{w} \right\rceil, 0\right) \tag{1}$$

We define q as the length of the queue, p the length of the packet, d the deficit of the queue and w the weight of the queue. The first part represents the exact number of rounds needed before a packet is serviced when placed in this queue. To understand this, consider that whenever a queue is serviced, exactly w is added to its deficit and that the deficit is decreased in proportion with the amount of data sent, b. This means that the RBS decreases by exactly one every time a queue is serviced, unless it already has a value of zero (q becomes $q - b$ and d becomes $d + w - b$). There is no advantage from negative numbers since they will all be serviced in the first round, therefore we allow only non-negative numbers.

If packets are of a consistent length and they are placed in the queue with the smallest RBS, ties being broken by whichever queue will be serviced sooner, then reordering cannot occur. To prove this we consider all cases of what can happen between two arrivals from the same flow. A packet from another flow can arrive, causing the queue length of some queue to increase and possibly increasing that queue's RBS. This does not effect the relative ordering of packets which are already enqueued. Alternately, a queue, q_1, can be serviced, which decreases the RBS of that queue by one and place the service pointer after it. We examine what effects this decrease in RBS has. Define q_2 as the alternative queue for a particular flow. If the RBS of q_1 and q_2 were equal, then q_1 would have won the tie breaker and been chosen, it would also have been serviced soonest. After the decrease, the RBS of q_1 is smaller and hence

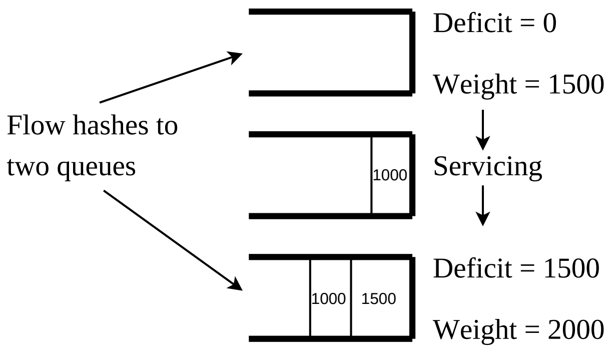

Fig. 2. Example of packet reordering: two 1000-byte packets follow a 1500-byte packet.

the packet goes into q_1 and is serviced in the correct order. If the RBS of q_1 is greater than that of q_2 then an incoming packet would have been placed in q_2 and after q_1 is serviced it will still be placed there, possibly with the help of a tie breaker. In the last case when the RBS of q_1 is less than that of q_2 then an incoming packet would have been placed in q_1 and will certainly be placed there after q_1 is serviced, since the discrepancy will be even greater. This shows that the changes in RBS are exactly consistent with how packets are actually sent out; therefore, these changes do not effect the order of packets that are already enqueued. Finally, an incoming packet will always be placed after all packets from its flow, since the previous packet would otherwise have been placed in a shorter queue. This last point is where we make use of the assumption that all packets are of the same size. When this does not hold a small packet might be sent out from a queue which did not have room for the larger, earlier packet. We have proved that reordering cannot occur when packets are of a consistent length.

When packets are allowed to be of different lengths then the last link in the proof is no longer true. We look at a specific example where reordering occurs. Consider figure 2, if two packets of length 1000 arrive, then the first is enqueued in queue 3 and the second in queue 1, which is the same order in which they are sent. However, if a packet of length 1500 is followed by one of length 1000 then the first is enqueued in queue 1 and the second in queue 3 and they are sent out in the reverse order. A naive attempt to remove this scenario by only enqueueing a packet in a queue if it has room for a maximum length packet, replacing p with the maximum length in equation 1, does not solve the problem. In the earlier example, the second packet is placed in queue 3 with the expectation that it will be sent out after waiting a round, instead it is sent out immediately.

Because of how reordering is triggered, it should not cause significant problems for TCP. In most situations TCP packets are of a consistent length and will not be reordered. In TCP the most common situations which cause large packets to be followed by small ones are lulls in the transmission. However, for three reorderings to occur, the transmission must continue to use small packets for a time. Even in this unlikely scenario, having a loss event erroneously inferred should not cause substantial performance degradation, since the flow is most likely transmitting at a slower than allowed rate. Despite this fact, an implementation should deal with reordering explicitly to prevent fringe cases from affecting performance.

We explore two practical methods for ensuring that reordering does not occur in an implementation even when packets are not of a constant size. First, a small buffer can be inserted at the output port. We calculate an upper bound for how much reordering can occur. Observe that for two packets p_a and p_b in the same flow, the second packet p_b had the option of being placed in the same queue as p_a with at least as good a position. If p_a is larger than p_b, then it might have been delayed by a round if placed in the position of p_b. Wherever p_a was queued, it must have been sent out earlier than it would have been in p_b's position; therefore, it cannot be delayed by more than a round after p_b is sent. This reasoning shows that one round is the most that a packet can be reordered by; therefore, we can create a small buffer at the output port which waits one round before sending a packet and resequences packets as necessary to avoid reordering. This completely eliminates reordering, even in the case where packets are of differing lengths.

Another option to avoid reordering packets makes use of the fact that many routers split packets into consistently sized cells before transmitting them over the switching fabric. This can be used to completely eliminate reordering. At the time when a packet is broken into cells, each cell should be treated independently and placed in the appropriate queue. The packet is reassembled as usual at the output port, since no cells are reordered, likewise no packets are reordered.

V. Evaluation

We evaluate MQFQ with respect to SFQ and SFB. All simulations are done with version 2.29 of network simulator ns-2 [22]. We use the default SFQ implementation in ns-2 with the improved hash function discussed earlier. We further modify this code to create MQFQ. SFB was downloaded from the Los Alamos National Laboratory [23] and adapted to version 2.29 of ns-2. All of the modifications and supporting scripts developed for this paper are available online [24]. Each experiment is run over a simple dumbbell topology. The bottleneck link capacity is 5 Mbps, and all access links run at 100 Mbps. The round trip propagation delay for constant bit rate (CBR) flows is fixed at 60 ms, since it has no effect on performance, while the delay for TCP flows is uniformly distributed between 60 and 80 ms. The packet size is always 1000 bytes. The forward bottleneck queue uses one of the schemes we investigate while all other queues are FIFO droptail. For all but one experiment, SFQ and MQFQ use 16 physical queues and SFB always uses 2 levels with 23 bins each. However, all three schemes use the same buffer size of 100 packets unless otherwise specified. A 100 packet buffer is twice the bandwidth-delay product, chosen so as to observe the queuing behavior for many flows. All other parameters are specific to each experiment. For each set of parameters we repeat the same experiment 10 times with different random seeds. The flows are sorted by throughput, and the standard deviation of each rank is plotted. To make the results more visible and to avoid overlap of the error bars, we offset the data points along the x-axis for MQFQ and SFB by 0.2 and 0.4, respectively.

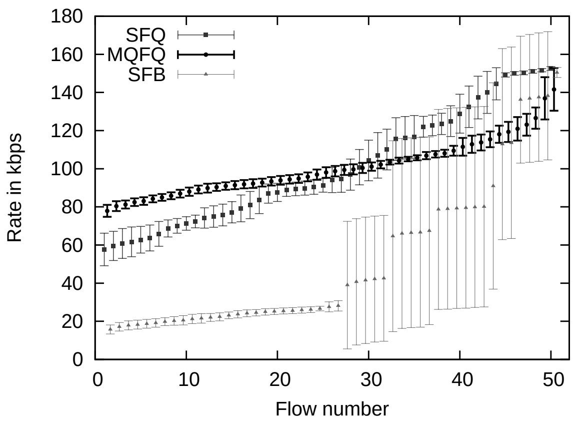

Fig. 3. 50 CBR flows at 0.15 Mbps each.

The following subsections show a series of experiments to compare MQFQ, SFQ, and SFB. We start by experimenting with CBR traffic to exclude any transient TCP behavior. Then we repeat these experiments with TCP instead of CBR flows to determine how each scheme handles responsive flows. Finally we evaluate the performance of aggregated queuing schemes with more than two hash functions.

A. Performance with constant bit rate flows

In the following experiments we evaluate the performance in terms of fairness under CBR traffic. We do this to separate TCP dynamics from queuing behavior. In the first experiment we use 50 CBR flows, each sending at 0.15 Mbps, which overloads the bottleneck link by 50 percent. In a sense, this experiment represents the case in which all flows are misbehaving, since they all transmit faster than they should. We are interested in how fairly each scheme can force unresponsive flows to distribute the available bandwidth in such an aggressive environment. The results are depicted in figure 3. Ideally, each

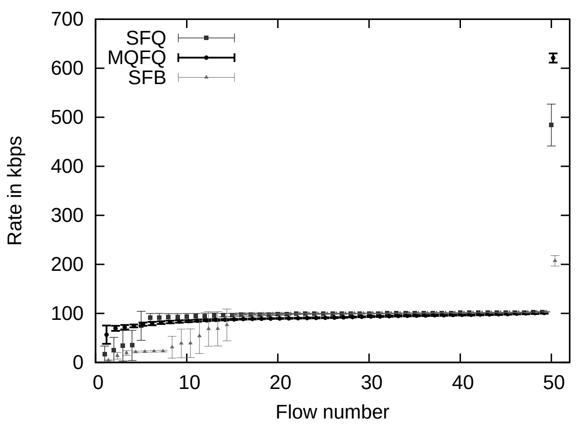

Fig. 4. 49 CBR flows at 0.1 Mbps each and 1 CBR flow at 2.5 Mbps.

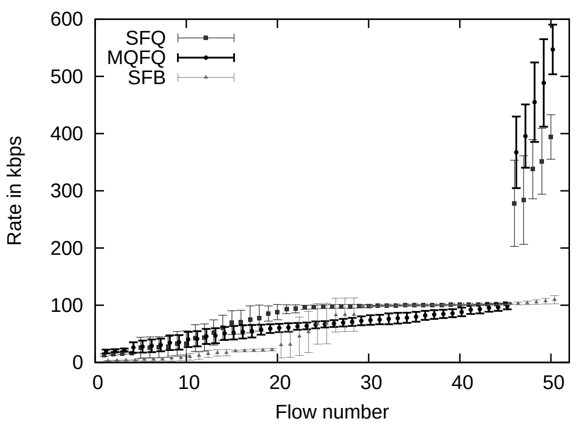

Fig. 5. 45 CBR flows at 0.1 Mbps each and 5 CBR flow at 1.5 Mbps.

flow should have a throughput of 100 kbps. We see that SFB fails to provide fairness because of the inability of its Bloom filter to single out and rate-limit a single flow. Instead SFB selects a certain number of flows to rate-limit by assigning a high discard probability. All other flows share the now available bandwidth and therefore avoid detection. However, because of excessively high discard probabilities, SFB is unable to utilize the link fully. In the figure, SFB flows with small error bars are rate-limited in all 10 experiments while all other flows are either rate-limited or not in each experiment, causing a bi-modal distribution which is not well represented by the large error bars.

When we compare SFQ and MQFQ experiments we see that the throughput range for MQFQ (78-141.5 kbps) is smaller and included within the range of SFQ (57.6-157.6 kbps). Since the range for MQFQ is included in the range of SFQ we conclude that MQFQ provides greater fairness in this setup.

In the following experiment we observe how a misbehaving flow affects the fairness under each scheme. A misbehaving flow is a flow that sends at a rate greater than its fair share and is not responsive to congestion signals. For the experiment in figure 4 considers 50 CBR flows, 49 of which are sending at their fair rate of 100 kbps and the last sends at 2.5 Mbps. We notice that many flows are unaffected by the misbehaving flow in all three scheme. SFB restricts the misbehaving flow to about 207 kbps which is much better than SFQ at 484 kbps or MQFQ at 620.8 kbps. SFQ suffers from a similar problem to SFB in that queues may not remain filled at all times. Queue underruns in some queues mean that the fair share of the other queues increases, causing the misbehaving flow to obtain a rate larger than the fair rate of a single queue, which would be 312.5 kbps in this case. Because MQFQ allows flows to occupy two queues, the queues will typically never underrun and be fully utilized. Therefore, as expected, the misbehaving flow is able to obtain the rate corresponding to the fair share of two queues.

On the left side of the figure 4 we see some flows suffering from the presence of the misbehaving flow. MQFQ improves the performance for these flows. SFB flows suffer the most from the misbehaving flow, this is because, as before, the Bloom filter becomes polluted due to unresponsive traffic and assigns high discard probabilities to packets despite queue underruns. Even when flows send at their fair rate SFB is unable to provide good performance due to its assumption that all flows are responsive.

In the last CBR experiment we increase the number of misbehaving flows to 5, each sending at 1.5 Mbps. Results are given in figure 5, as expected more flows send at reduced rates. In this extreme case MQFQ is unable to avoid limiting flows to nearly zero throughput. Moreover MQFQ allocates a substantial amount of bandwidth to the misbehaving flows, reducing the performance of all other flows. In this case, it appears that SFQ is better suited to confining misbehavior because it allocates less bandwidth to each misbehaving flow. SFB manages to restrict all misbehaving flows to their fair share. However, it still suffers from the same problem of low throughput due to polluted Bloom filters and only transmits at about 60% of the link capacity.

We have seen that MQFQ provides benefits when there is little misbehavior. And that when the number of misbehaving flows increases the performance of MQFQ decreases and becomes worse than that of SFQ. In the next section we study the performance of the schemes with responsive traffic.

B. Performance with responsive flows

We setup experiments in the same way as in the previous section except that we replace CBR with TCP traffic. These experiments correspond to normal network behavior and to that seen when misbehaving flows are rate-limited using a large flow detecting algorithm. To avoid capturing any transient behavior of TCP we

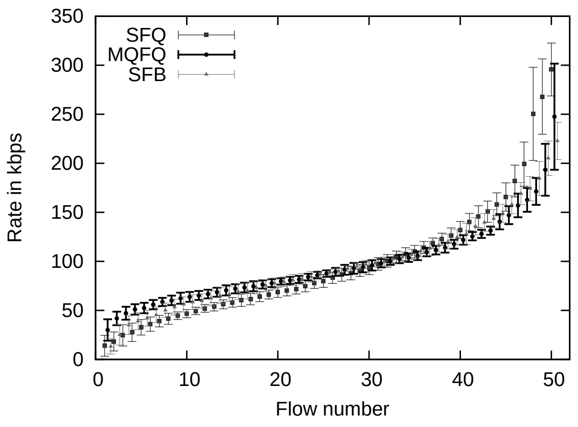

Fig. 6. 50 TCP flows, 100-packet buffer.

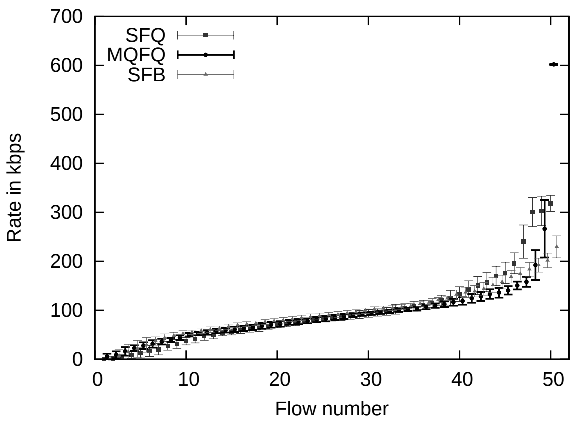

Fig. 8. 49 TCP flows and 1 CBR flow at 2.5 Mbps.

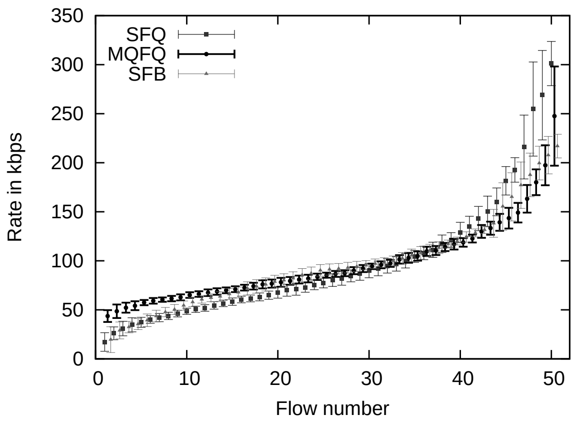

Fig. 7. 50 TCP flows, 200-packet buffer.

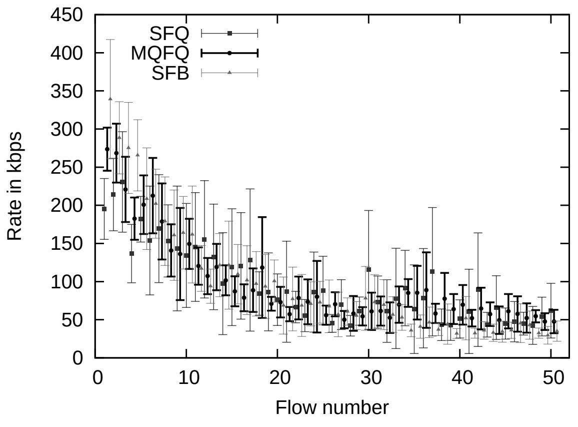

Fig. 9. 50 TCP flows sorted by increasing round trip time.

increased the experiment duration to 100 seconds and discard the first 10 seconds. Figures 6 and 7 show the result for a buffer size of 100 and 200, respectively. In both experiments MQFQ provides the best minimum throughput to flows. The performance for most flows is very similar for all three schemes. On the right side of the graph, where all the high throughput flows are, we note that SFQ performs worse than the other schemes, allocating a disproportionately high share to a single flow. comparing the two figures we see that increasing the buffer size improves fairness a little for the lowest throughput flows.

We are interested in how well the schemes react to the presence of an unresponsive flow sending at an unfair rate. Figure 8 shows the throughput for 49 TCP flows and one CBR flow sending at 2.5 Mbps. The CBR flow is always labeled as flow number 50. We see that none of the schemes can prevent some TCP flows from having a near zero throughput. SFB restricts a single misbehaving flow to a rate comparable to other flows. In SFQ the CBR flow forces all TCP flows that are in the same queue to back off and uses a single queue by itself achieving approximately the fair share rate of the queue, 312.5 kbps. Note that a queue's throughput is $1/k$ where k is the number of queues. In our case it is 312.5 kbps. We also note that in each experiment approximately two or three flows also dominate a queue and send at this rate. Those flows are either lucky enough to hash into a private queue, or other TCP flows backed off after losing several early packets. In MQFQ none of the TCP flows acquire their own queue because flows hash to multiple queues and interact with each other. The CBR flow sends at the rate of two queues since it forces all TCP flows in those queues to use other queues.

As commonly known, TCP throughput is inversely proportional to RTT. We study the impact of round trip times on fairness by modifying our analysis slightly. We create 50 TCP flows with round trip times ranging from 60 ms to 550 ms distributed at 10 ms intervals. When plotting the results in figure 9 we sort flows by RTT instead of by throughput as in the other experiments. The clear decreasing trend in all schemes shows that none of them provide particularly good fairness to flows experiencing different RTT.

Next we look at how performance changes for MQFQ when we vary the ratio of TCP flows to queues. To be able to compare different numbers of flows we plot the cumulative percentage of flows in each experiment on

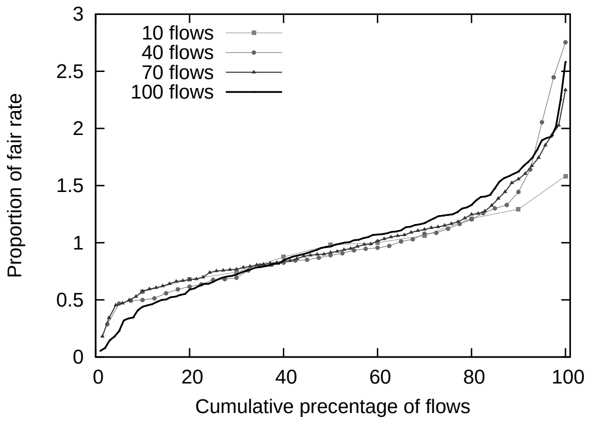

Fig. 10. Evaluation of MQFQ using 16 queues for varying numbers of flows.

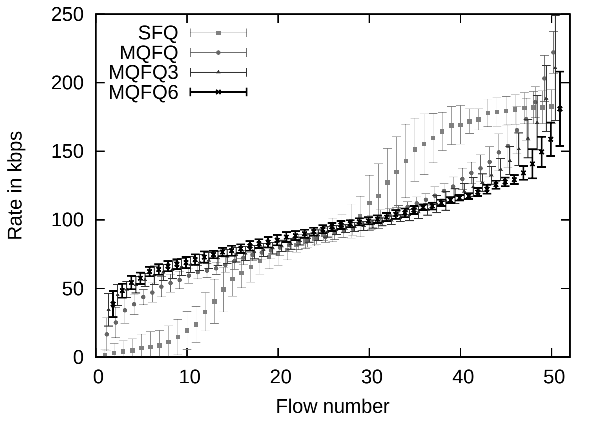

Fig. 11. 50 TCP flows.

the x-axis and the rate of the flow normalized so that 1 always corresponds to a fair rate on the y-axis. We vary the number of flows from 10 to 100 in steps of 30. Figure 10 shows the average of 3 experiment runs. As flows are added there is little difference in fairness. Clearly, the number of flows can be much greater than the number of queues, without adverse effects.

C. *Performance of MQFQ with more than two queues*

As discussed earlier we can vary the number of queues to which we hash flows. Using more queues minimizes the chance that a flow suffers from a misbehaving flow, but it also increases the amount of bandwidth a misbehaving flow can acquire. We denote the number of hash functions used by appending a number to MQFQ. For example, MQFQ3 uses 3 hash functions. For this experiment we use 32 queues instead of the usual 16. Figure 11 shows the performance for 50 TCP flows while figure 12 depicts 49 TCP flows and one misbehaving CBR flow at 2.5 Mbps. In figure 11, we see that the use of two queues improves performance, but that further performance benefits are small when using more than two queues. However, when one flow misbehaves then using more hash functions gives more throughput to the

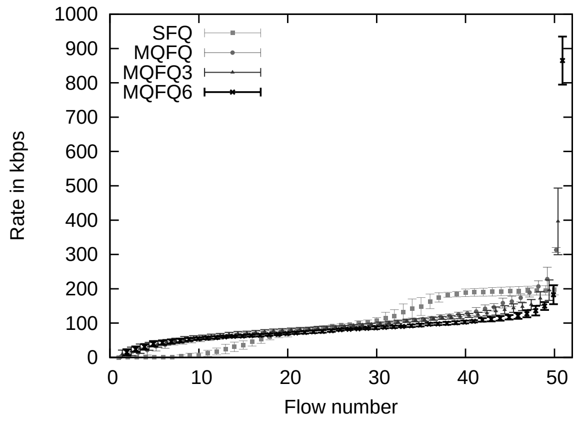

Fig. 12. 49 TCP flows and one CBR flow.

misbehaving flow. Although behaving flows have up to 6 queues to choose from, we still see flows with very low throughput. Using more than two hash functions does not provide any benefit. Figure 12 shows an important effect of MQFQ with many hash functions: although the misbehaving flow is able to acquire an increasingly large share of the bandwidth as more hash functions are used, this bandwidth is mainly taken from the fastest flows. The number of slow flows is not greatly affected. That being said, since slow flows remain consistent, there is little incentive to use more than two hash functions.

VI. Conclusion

In this paper, we explored how to protect individual flows from a misbehaving greedy flow in routers that provide isolation only between flow aggregates. We investigated MQFQ, a queuing discipline that does not require per-flow state in the router. MQFQ employs a number of independent hash functions to assign a set of different queues to each flow. When a packet arrives to the router, the router places the packet into the queue that will service the packet the quickest among all queues associated with the packet's flow. We explored the use of multiple hash functions and found that two is the optimal number of queues per flow. The paper analyzed performance of MQFQ in relation to two existing schemes SFQ and SFB. Under unresponsive CBR traffic, MQFQ provided better normal-case behavior and provided more protection from a misbehaving flow. SFB did not perform well in this case because it was not designed for handling purely unresponsive traffic. When the traffic mix included responsive TCP flows, MQFQ was again able to perform better in the normal case and provided protection to the weakest flows in the presence of misbehavior. The possibility of packet reordering by MQFQ was carefully examined since such reordering might severely undermine TCP performance. We developed simple techniques which prevent all packet reordering.

References

[1] A. Demers, S. Keshav, and S. Shenker, “Analysis and Simulation of a Fair Queueing Algorithm,” in *Proceedings of ACM SIGCOMM*, 1989, pp. 1–12.

[2] J. Nagle, “On Packet Switches With Infinite Storage,” *IEEE Transactions on Communications*, vol. 35, no. 4, pp. 435–438, April 1987.

[3] A. K. Parekh and R. G. Gallager, “A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks – The Multiple Node Case,” *IEEE Transactions on Networking*, vol. 2, no. 2, pp. 137–150, 1994.

[4] M. Shreedhar and G. Varghese, “Efficient Fair Queueing using Deficit Round Robin,” in *Proceedings of ACM SIGCOMM*, 1995, pp. 231–242.

[5] S. Ramabhadran and J. Pasquale, “Stratified Round Robin: A Low Complexity Packet Scheduler with Bandwidth Fairness and Bounded Delay,” in *Proceedings of ACM SIGCOMM*, 2003, pp. 239–249.

[6] S. Golestani, “A Self-Clocked Fair Queuing Scheme for Broadband Applications,” in *Proceedings of IEEE INFOCOM*, 1994, pp. 636–646.

[7] J. C. Bennett and H. Zhang, “WF2Q: Worst-case Fair Weighted Fair Queueing,” in *Proceedings of IEEE INFOCOM*, March 1996, pp. 120–128.

[8] D. Bertsekas and R. Gallager, *Data Networks*. Prentice-Hall, 1987.

[9] J. Postel, “User Datagram Protocol,” RFC 768, October 1980.

[10] M. Allman, V. Paxson, and W. Stevens, “TCP Congestion Control,” April 1999, RFC 2581.

[11] V. Jacobson, “Congestion Avoidance and Control,” in *Proceedings of ACM SIGCOMM*, August 1988.

[12] J. Mahdavi and S. Floyd, “TCP-Friendly Unicast Rate-Based Flow Control,” January 1997, end2end-interest mailing list.

[13] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, “Modeling TCP Throughput: A Simple Model and its Empirical Validation,” in *Proceedings of ACM SIGCOMM*, September 1998.

[14] P. E. McKenney, “Stochastic Fairness Queueing,” in *Proceedings of IEEE INFOCOM*, June 1990, pp. 733–740.

[15] W. Feng, D. D. Kandlur, D. Saha, and K. G. Shin, “Stochastic Fair Blue: A Queue Management Algorithm for Enforcing Fairness,” in *Proceedings of IEEE INFOCOM*, 2001, pp. 22–26.

[16] R. Mahajan, S. Floyd, and D. Wetherall, “Controlling High-Bandwidth Flows at the Congested Router,” in *Proceedings IEEE ICNP 2001*, November 2001.

[17] S. Floyd and V. Jacobson, “Random Early Detection Gateways for Congestion Avoidance,” in *Proceedings of IEEE/ACM Transactions on Networking*, August 1993, pp. 397–413.

[18] W. Feng, K. Shin, D. Kandlur, and D. Saha, “The BLUE Active Queue Management Algorithms,” in *Proceedings of IEEE/ACM Transactions on Networking*, August 2002, pp. 513–528.

[19] R. Pan, B. Prabhakar, and K. Psounis, “CHOKe, A Stateless Active Queue Management Scheme for Approximating Fair Bandwidth Allocation,” in *IEEE INFOCOM*, March 2000.

[20] R. Pan, L. Breslau, B. Prabhakar, and S. Shenker, “Approximate Fairness Through Differential Dropping,” *ACM SIGCOMM CCR*, vol. 33, no. 2, pp. 23–39, 2003.

[21] C. Estan and G. Varghese, “New Directions in Trafc Measurement and Accounting,” in *Proceedings of ACM SIGCOMM*, August 2002.

[22] “The Network Simulator – ns-2,” http://www.isi.edu/nsnam/ns/.

[23] “SFB Implementation,” http://public.lanl.gov/sunil/.

[24] C. Jechlitschek, “MQFQ Implementation in ns-2 and Supporting Scripts,” February 2007, http://www.arl.wustl.edu/~mgeorg/mqfq/.