TCP Performance under Aggregate Fair Queueing

Wei Sun and Kang G. Shin

Department of Electrical Engineering and Computer Science
The University of Michigan
Email: {wsunz, kgshin}@eecs.umich.edu

Abstract*— Per-flow fair queueing (FQ), when combined with appropriate buffer-management schemes, has been shown in [1] to outperform FIFO scheduling in terms of TCP throughput and fairness. We extend per-flow FQ to *aggregate-flow FQ* and evaluate its TCP performance under the assumption that core routers recognize *only* traffic aggregates, *not* individual TCP flows. First, we show that the combination of FQ scheduling and DropFront buffer-management schemes can cause a livelock problem for greedy traffic sources (where all the packets of the greedy sources are dropped), and propose a simple scheme to solve this problem. Next, using simulation, we show that when combined with proper buffer-management schemes such as ALQD (Approximated Longest Queue Drop), aggregate-flow FQ offers even better throughput than, but is not as fair as, per-flow FQ. By introducing a new buffer-management scheme called *ALQD+*, we improve significantly the fairness of aggregate-flow FQ over FIFO queueing. Finally, we analyze the performance of aggregate-flow FQ, and explain why its throughput and fairness are better than per-flow FQ and FIFO queueing, respectively. Overall, aggregate-flow FQ not only incurs lower scheduling and state-maintenance overheads at routers than per-flow FQ, but also provides performance comparable to per-flow FQ in terms of TCP throughput and fairness. These features make aggregate-flow FQ very attractive for use in the Internet backbones.

I. Introduction

Per-flow fair queueing (FQ) has been shown to possess many attractive features such as bounded delay and excellent fairness [2], [3]. Traditionally, combined with reservation-based schemes at routers and rate-based control schemes at end-hosts, FQ has been used to guarantee such Quality-of-Service (QoS) as end-to-end (e2e) delay and bandwidth allocation. Maintaining a large number of states for individual flows, however, causes high overhead for the routers. This problem makes per-flow FQ unscalable to a large number of flows and prevents its use in large core networks (e.g., Internet backbones).

To improve the scalability of per-flow scheduling, aggregate scheduling has been proposed. The main idea behind aggregate scheduling is illustrated in Fig. 1. Some parts of the network, called *aggregation regions*, only "see" aggregated (instead of individual) flows. Resource reservation, packet scheduling, and buffer management in an aggregation region are done on a per-aggregate basis. As shown in Fig. 1, the flows sharing the same path for a number of hops inside an aggregation region can be bundled together and treated as a *single* aggregate inside that region. In general, a traffic aggregate can be created and terminated at any point in the network, and it can also be created recursively. The router (e.g., S_1) that aggregates flows is called an *aggregator*, and the router (e.g., S_k) that splits the aggregate back into individual flows is called a *deaggregator*. Only aggregators and deaggregators "see" individual flows.

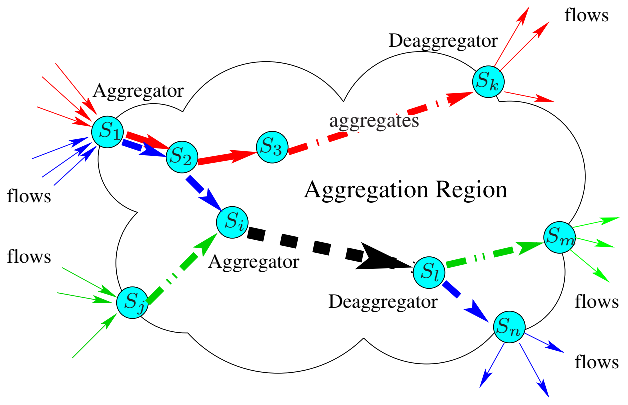


Fig. 1. Aggregate scheduling

Aggregate scheduling offers better scalability and induces less overhead than per-flow scheduling, as core routers only need to maintain states for traffic aggregates. It simplifies packet classification and scheduling significantly. However, under aggregate scheduling, flows in the same aggregate cannot isolate from, and protect against, each other, i.e., the performance of a flow would be affected by other flows in the same aggregate. Fortunately, this interference can be controlled if flows are aggregated fairly. We show this feature of fair aggregation by using FQ to aggregate traffic flows.

As a special case of aggregate scheduling, aggregate FQ uses FQ algorithms to bundle flows into aggregates as well as schedule aggregates at core routers. It has been shown to offer many advantages. For instance, for non-adaptive traffic (e.g., UDP flows), it is proven in [4] and [5] that aggregate FQ can provide e2e delay performance better than that of per-flow FQ.

Incorporating FQ into the congestion control of best-effort traffic was discussed first in [2]. Later, the authors of [1], [6] showed that, when combined with per-flow buffer-management schemes, per-flow FQ offers TCP flows better isolation and fairness, as well as better throughput than FIFO queueing. However, to the best of our knowledge, little has been done on the performance of aggregate FQ in congestion control. In this paper, we extend the work in [1], [6] significantly and study the problem of using aggregate FQ in core networks to support TCP traffic.

II. Related Work

The authors of [1] discussed the design issues associated with the use of per-flow FQ to support TCP. They showed that per-flow FQ alone provides few advantages over FIFO queueing, and that buffer management is also important. Instead of using global schemes such as DropTail and RED, they proposed per-flow buffer-management schemes Longest Queue Drop (LQD) and Approximated Longest Queue Drop (ALQD). Per-flow FQ plus per-flow buffer management was shown to provide better performance in terms of throughput and fairness under various conditions.

The authors of [6] generalized the work in [2] into a *Fair Queueing* (FQ) paradigm, and proposed it as a means of designing e2e congestion-control protocols. The FQ paradigm consists of two components: per-flow FQ and LQD buffer management. They showed, via simulation, that the FQ paradigm improved not only the average throughput of TCP flows, but also the speed of convergence to the equilibrium state, thus increasing the network stability.

All of the per-flow buffer-management schemes discussed so far (e.g., LQD and ALQD) drop packets only when the buffer is full. In contrast, the authors of [7] proposed a version of per-flow random early dropping/detection, called Balanced-RED (BRED). Their scheduling scheme was FIFO, and their main performance focus was on fairness without considering throughput. BRED was shown to offer better fairness than RED. Later in Section IV, we will adopt a similar idea to drop packets early from each aggregate.

For traffic aggregation, the authors of [8] defined a *traffic trunk* as an aggregate of traffic flows that belong to the same class. In the context of Multiprotocol Label Switching (MPLS), the flows in the same traffic trunk share the same path and are mapped onto the same Label-Switched Path (LSP). Our definition of an aggregate is very similar to the traffic trunk, except that the aggregates are bundled and scheduled by using FQ. Also, we assume that an e2e path in an aggregation region can be set up by using traffic engineering mechanisms similar to the way an LSP is set up in MPLS.

III. DropFront's Livelock Problem

The *DropFront* buffer management scheme—when it is necessary to drop a packet, the packet at the front of the queue is dropped—is shown to be able to trigger the TCP's fast retransmit/recovery mechanism more quickly than others, and thus, increase TCP throughput [9], [1]. However, when it is combined with FQ, a *livelock* can occur to UDP flows (more generally, greedy flows that are not responsive to packet dropping). When the actual arriving rate of a flow/aggregate, r, is greater than its service rate R at a router, the flow/aggregate could totally be shut down at the router.

Let's consider how this can occur. Suppose the packet size is fixed at S, and the reserved buffer space for the flow is N. Suppose there already exist N packets of the flow in the buffer, and their virtual finish times are

$$t_0, t_0 + \frac{S}{R}, \cdots, t_0 + (N-1)\frac{S}{R}.$$

New packets arrive at

$$t_1, t_1 + \frac{S}{r}, t_1 + 2\frac{S}{r}, \cdots, t_1 + N\frac{S}{r}, \cdots$$

If the server is busy all the time, and at t_1 all the N packets of the flow are still in the buffer, then according to the DropFront algorithm, the first packet in the queue (with finish time t_0) will be dropped; according to the FQ algorithm, the newly-arriving packet will be placed at the end of the queue with finish time $t_0 + N\frac{S}{R}$. As a result, the finish time of the packets in the queue will become:

$$t_0 + \frac{S}{R}, t_0 + 2\frac{S}{R}, \cdots, t_0 + N\frac{S}{R}.$$

Similarly to the above discussion, the current first packet in the queue (with finish time $t_0 + \frac{S}{R}$) will also be dropped when the next new packet arrives (at $t_1 + \frac{S}{r}$), because $\frac{S}{r} < \frac{S}{R}$ from $r > R$. Subsequently, all the packets will be dropped after spending some time in the buffer. The key reason for this is that packets in the buffer are transmitted based on their virtual finish times under the FQ algorithm. Newly-arriving packets have larger finish times than all the packets already in the queue. If the arriving rate is greater than the service rate, packets with larger finish times will keep entering the end of the queue, while the packets at the head of the queue are being dropped. The finish time of the packet at the head of the queue just keeps increasing. Therefore, the router will always serve packets with smaller finish times from other queues.

Note that in the above discussion, we assumed that all the packets in the queue have start and finish times. In practice, however, an efficient implementation of most FQ algorithms assigns start and finish times only to the first packet in each per-flow queue. Therefore, to solve the livelock problem, the FQ algorithm calculates the start and finish times only for the packet at the head of queue, and distinguishes whether the packet is transmitted or dropped. If the packet is transmitted, the start and finish times are also updated; if the packet is dropped because of other packets' arrival, only the packet itself will be dropped, the start time of the queue is not updated (the new finish time is calculated from the start time).

IV. ALQD+

Both LQD and ALQD were proposed in [1]. LQD keeps the queue-length information for each active flow in the buffer. When a new packet arrives and the buffer is full, a packet from the longest queue is dropped. ALQD is a simplified version of LQD. Instead of searching for the longest queue whenever a packet needs to be dropped, ALQD uses a register to record the longest queue during the previous queue operation. Then, whenever a packet needs to be dropped, a packet from that registered queue is dropped. On every queueing operation, the current queue length is compared to that of the registered queue and the register's content is updated as necessary.

Our simulation results show that the fairness of aggregate FQ using ALQD is not very promising. To improve the fairness of aggregate FQ, we extend ALQD and propose a new buffer-management scheme called ALQD+. The key components of ALQD+ include: i) random early dropping (we incorporate the idea of BLUE [10] into ALQD); ii) packet match in early

TABLE I
MODIFIED BLUE ALGORITHM

d_m:	packet dropping probability;
δ:	amount by which d_m is incremented/decremented;
$freeze\_time$:	minimum interval between two updates of d_m.

```
1. //Packet loss update (parameter: $count$):
   If (now - $last\_update > freeze\_time$)
      $d_m += \delta \cdot \max\{count, \frac{now - last\_update}{freeze\_time}\}$
      $last\_update$ = now
2. //Queue empty update:
   If (now - $last\_update > freeze\_time$)
      $d_m -= \delta$
      $last\_update$ = now
```

dropping (a packet at the head of a queue is dropped *only* upon the arrival of a packet from the same flow); and iii) updating the drop rate in *rounds*.

The flowchart of the algorithm is given in Fig. 2. When a packet belonging to queue k arrives at a server, the algorithm checks if the buffer is full: (i) when the buffer is already full, then if the length of queue k, $qlen_k$, already exceeds its fair share $share_k$, the first packet of queue k will be dropped; otherwise, as in ALQD, the first packet from the longest queue, max_q, will be dropped; (ii) (the *early-dropping step*) when the buffer is not full, if the total queue size exceeds $a\%$ of the total buffer size (B) and $qlen_k$ already exceeds $b\%$ of $share_k$, then go to step (iii); or if $qlen_k$ already exceeds $c\%$ of $share_k$, with probability d_k, go to step (iii); (iii) (the *packet-match step*) if the new packet and the first packet in queue are both from the same flow, then the first packet is dropped; otherwise, the new packet is accepted without dropping any packets. The role of this packet-match step is to improve the fairness of the algorithm: a packet from a given flow will not be dropped due to the arrival of another flow's packet.

Whenever a packet is dropped from a queue, the drop rate of the queue also needs to be updated. This leads to the third component of the algorithm—updating the drop rate in rounds. Since the flows are aggregated with fair queueing in aggregate FQ, packets from each flow are more evenly distributed within an aggregate. To take advantage of this feature and make the dropping fairer, we modify BLUE to drop packets in *rounds*. A *round* is a packet sequence starting at a packet from a given flow, and ending before the next packet from the same flow. For example, suppose an aggregate consists of 10 flows: 1, 2, 3, $\cdots$, and 10. Then, a round may start at a packet from flow 1 and end before the next packet from flow 1. Within a round, the drop rate is fixed, so packets from different flows have the same drop rate. At the same time, we count the number of packet drops in an aggregate within a round. Then, at the end of the round, the drop rate of the aggregate is updated according to the number of drops during the round.

The modified BLUE algorithm is given in Table I. When an update is triggered by a packet dropping, it supports a parameter $count$, which is the number of drops in the last round. Therefore, the drop-rate update is based on $count$.

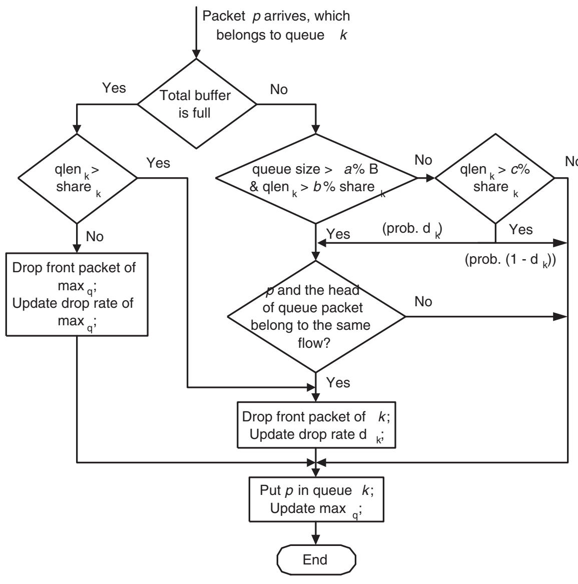

Packet p arrives, which belongs to queue k
Total buffer is full
Yes
No
qlen_k > share_k
Yes
No
queue size > a% B & qlen_k > b% share_k
No
qlen_k > c% share_k
No
Yes
(prob. d_k)
Yes
(prob. (1 - d_k))
Drop front packet of max_q; Update drop rate of max_q;
p and the head of queue packet belong to the same flow?
No
Yes
Drop front packet of k; Update drop rate d_k;
Put p in queue k; Update max_q;
End

Fig. 2. The ALQD+ algorithm

When the update is triggered by an empty queue, however, the algorithm remains the same.

V. SIMULATION DESIGN

Using the *ns2* simulator [11], we evaluated and compared the TCP performance under the following schemes (each of which consists of a pair of packet scheduling and buffer management algorithms): (i) <aggregate FQ, ALQD/ALQD+>; (ii) <per-flow FQ, ALQD>; (iii) <FIFO queueing, DropTail>; (iv) <FIFO queueing, RED>. The goal of the simulation is to see whether aggregate-flow FQ still retains the same advantage as per-flow FQ in terms of TCP performance. The main performance metrics of interest are throughput and fairness.

In the simulation, we used Weighted Fair Queueing (WFQ) as the scheduler for both per-flow and aggregate-flow FQ. In addition, for per-aggregate buffer management, we extended the ALQD scheme in [1] by using a weighted version of *fair share*. The fair share $share_k$ of flow (or aggregate) k is computed as $share_k = \frac{w_k \cdot B}{\sum_{k=1}^{n} w_k}$, where w_k is the weight of flow (or aggregate) k. In general, the weight of each flow and aggregate can be set arbitrarily. For simplicity, as in [1], [6], we set the default weight for a single flow to 1. The weight of an aggregate is simply set to the number of individual flows in it. For RED, we set $min_{th} = 25\%$, $max_{th} = 75\%$, and queue weight $w_q = 0.002$; for ALQD+, we set BLUE parameters *freeze_time* = 0.05s, and δ = 0.01. Based on our testing results, we set $a = b = c = 25$.

The network topology in Fig. 3 was used for our simulation. There are a total of N TCP sources, whose packets enter the network through edge routers IR_i ($1 \leq i \leq n$), then traverse core routers CR_1 and CR_2 before departing from edge routers ER_i ($1 \leq i \leq n$). Each flow enters IR_i and departs from the corresponding ER_i. In the case of aggregate-

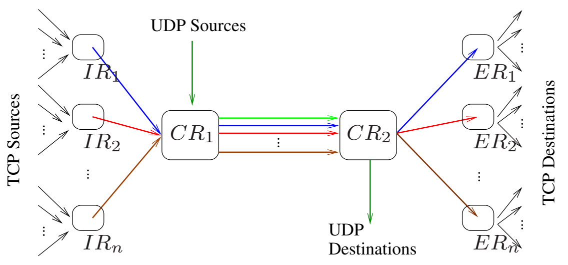
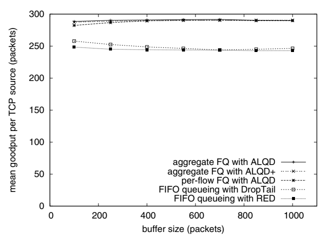
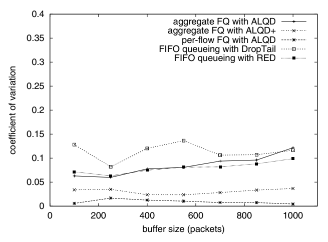


Fig. 3. Network topology in the simulation



(a) Goodput comparison



(b) Fairness comparison

Fig. 4. Performance of identical flows

flow FQ, all the flows sharing the same edge router IR_i were aggregated together, with each aggregate containing N/n flows. Therefore, CR_1 and CR_2 recognized the n aggregates only and scheduled them using FQ. The aggregates were later split back into flows at ER_i. In the simulation, we used $N = 100$ and $n = 10$.

The link bandwidth and delay were varied during the simulation. All of incoming links (between TCP sources and IR_i) and outgoing links (between ER_i and TCP destinations) had a bandwidth of 64 Kbps and a default delay of 10 ms. All the other links had a delay of 2 ms. The links between IR_i and CR_1 had a bandwidth of 640 Kbps (64K $\cdot$ 10); so did the links between CR_2 and ER_i. The link between the two core routers (CR_1 and CR_2) was the bottleneck (with the default bandwidth of 3.2 Mbps—only a half of the total source link bandwidth, 64K $\cdot$ 100). To test the isolation feature of aggregate scheduling, non-responsive UDP flows were also introduced in the network. CBR was used as the UDP sources, the total rate of which was 25% of the total bottleneck bandwidth. The weight of the UDP traffic was fixed at 10. Therefore, it was sending more than its fair share $(10/(100+10) = 1/11)$. All packets were 1000 bytes long. For each configuration, we executed six independent simulation runs. Each run lasted 100 seconds. To avoid the dynamics of TCP's initial slow-start phase, the data in the first 20 seconds were discarded. Only the data collected in the remaining 80 seconds were used in the evaluation.

We counted the number of packets that are correctly acknowledged by the receivers in a given time period as the throughput (or more accurately, goodput). The fairness was calculated as follows: for each TCP flow k, we first calculated its throughput A_k, the average value of six runs. Then, using $A_k, 1 \leq k \leq N$, we computed the overall average throughput $\widehat{A}$. From $\widehat{A}$ and A_k, we computed the *coefficient of variation* (ratio of the standard deviation to the mean) σ, which was used as the index of fairness.

VI. The Simulation Results

We ran simulations for three different cases: (i) all of the TCP flows are identical; (ii) flows have different round trip times (RTTs); and (iii) the aggregates have inaccurate weight values. All the results presented here are based on TCP Reno. We also ran simulations using TCP NewReno and SACK, the results of which were consistent with TCP Reno's and thus omitted.

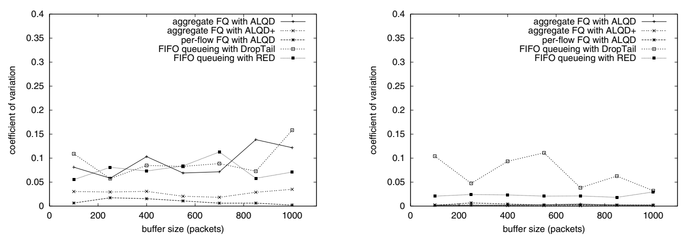
A. Identical flows

First, we compared the performance of ALQD and ALQD+. Fig. 4 plots the result for the case of homogeneous TCP flows. The throughput of aggregate FQ (under both ALQD and ALQD+) and per-flow FQ are much larger than the two FIFO queueing schemes. An interesting observation is that the throughput under aggregate FQ is even better than that of per-flow FQ, especially when the buffer size is small. Section VII will elaborate on this. The fairness of aggregate FQ under ALQD, however, is much worse than per-flow FQ, and is comparable to that of FIFO queueing. In contrast, the fairness under ALQD+ is improved significantly.

The main cause of this unfairness is the core routers' inability to differentiate individual flows in an aggregate. When a scheduler has to provide congestion feedback by dropping a packet in an aggregate, since it cannot distinguish among the flows in the aggregate, it may drop a packet from a well-behaving flow. Then, this well-behaving flow, instead of other misbehaving flows, will respond to the "feedback." To further confirm this, Fig. 5 plots the fairness among flows in the same aggregate as well as among different aggregates. Fig. 5 (a) shows the fairness among 10 flows in the first aggregate, while Fig. 5 (b) shows the fairness among 10 aggregates. Clearly, the unfairness problem exists among the flows *within* the same aggregate. In contrast, the fairness *among* aggregates is very close to that of per-flow FQ.

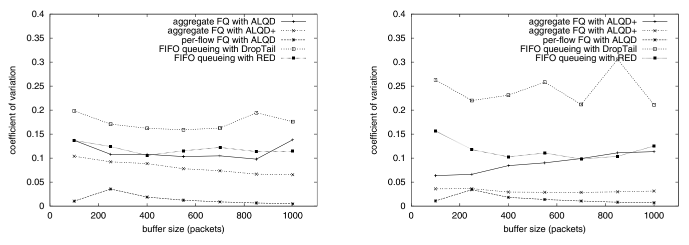
To further study the fairness problem, we examined the relationship between the bottleneck bandwidth and fairness. Fig. 6 shows that the fairness is related to the degree of congestion. Figs. 6 (a) and 6 (b) show the case when the bottleneck link bandwidth is 4.8 Mbps. Since the congestion is less severe, aggregate FQ is much farer than in the previous case. We also tested the case when the bottleneck link bandwidth is greater than 8 Mbps, which is greater than the total bandwidth of incoming links. In this case, the incoming links become the bottleneck. Since almost all the packets received by the aggregators IR_i can be transmitted by CR_1, both per-flow and aggregate FQ have virtually identical performances (both throughput and fairness). This shows that the fairness of aggregate FQ depends on the location of, and the severity of congestion at, the bottleneck. If the bottleneck is not at core routers, aggregate and per-flow FQ have virtually identical throughput and fairness.

In the following discussion, we will present the results for



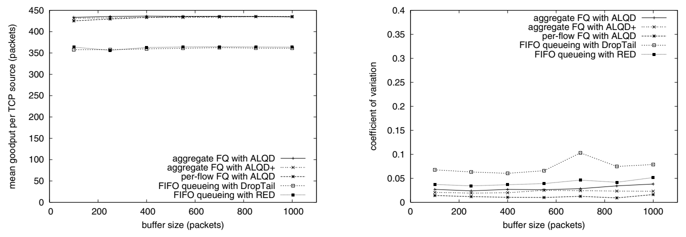
(a) Inside an aggregate (b) Among aggregates

Fig. 5. Fairness comparison



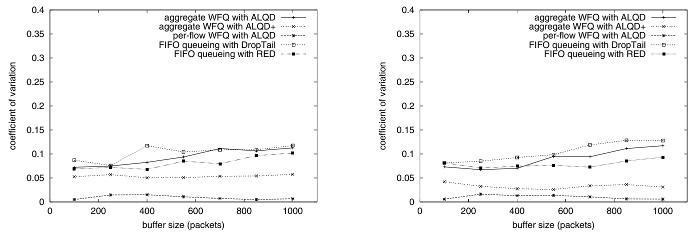
(a) Intra-aggregate heterogeneous (b) Inter-aggregate heterogeneous

Fig. 7. Fairness comparison: flows with heterogeneous RTTs



(a) Goodput comparison (b) Fairness comparison

Fig. 6. Performance under lower-level congestion



(a) Inaccurate Weights (90% accuracy) (b) Random ON/OFF Flows

Fig. 8. Fairness comparison: robustness of ALQD+

ALQD+ only and focus on the fairness of aggregate FQ. The average throughput results are similar to that in Fig. 4 and thus omitted.

B. Heterogeneous RTTs

We first explored the flow heterogeneity inside an aggregate: in each of the 10 aggregates, one flow had a longer RTT (100 ms incoming/outgoing link delay), and all the others had a shorter RTT (10 ms incoming/outgoing link delay). Then, we explored the heterogeneity among aggregates: all the flows in the first aggregate had a longer RTT (100 ms incoming/outgoing link delay) than those in other aggregates (10 ms incoming/outgoing link delay).

Fig. 7 plots the simulation results of the first case. In both cases, the fairness of aggregate FQ is better than that of the two FIFO queueing schemes. The fairness of the second case, however, is significantly better than that of the first. By further examining the throughput of all the flows with longer RTTs in both cases, we found that in the first case, flows with longer RTTs had much lower throughput than other flows, while those in the second case had, on average, almost the same throughput as others. This implies that when bundling flows into aggregates, flows with very different RTTs should be put in different aggregates.

C. Robustness

Since we used WFQ and weighted ALQD+, we need to set the weights of the aggregates in core routers. The previous work on per-flow FQ [1], [6] assumed all the flows to have an identical weight. In the case of aggregate FQ, we used the number of flows in an aggregate as its weight. However, in the real Internet, the accurate estimate of the number of flows in each aggregate may be difficult to obtain. Therefore, the weight of an aggregate may not be the same as the actual number of flows in it.

We ran simulations to test the robustness of aggregate FQ to inaccurate weight estimates. In the test, the weight of each aggregate remained the same (10). We used more flows than the number of flows in the first aggregate, while using fewer flows in the second aggregate and keeping all the other aggregates intact. Fig. 8 (a) shows the result when there are 10% more flows (i.e., 11) in the first aggregate and 10% fewer flows (i.e., 9) in the second aggregate. The fairness of aggregate FQ is still better than that of FIFO queueing.

To study further the robustness of aggregate FQ, we used 10 bulk FTP sources and four ON/OFF telnet sources in each aggregate. The weight of each aggregate was set to 12. Therefore, the actual number of flows in each aggregate changed dynamically, and thus, the weight for every aggregate could be inaccurate, which is the case in a real network. (In the previous case, only the weights for the first two aggregates were inaccurate, and remained inaccurate during the entire simulation.) Also, the accuracy of the weights was within 83.3% ($1 - \frac{2}{12}$) of the actual value. We only recorded the throughput of the 10 FTP flows and used them to study both fairness and throughput. As shown in Fig. 8 (b), the unfairness is less severe, implying that in the real Internet, aggregate FQ may not be as sensitive as shown in Fig. 8 (a).

In addition, algorithms for counting the number of active flows have been proposed in the literature. For example, the authors of [12] proposed a family of counting algorithms based on updating a bitmap at run-time. The algorithms were shown to have not only low memory requirements, but also very high accuracy. For example, the *adaptive bitmap* algorithm can count the number of distinct flows on a link that contains up to 100 million flows. It has better than 99% accuracy and requires only $2K$ bytes of memory; while the *triggered bitmap* uses even less memory, and is optimized for running multiple concurrent counting processes.

VII. Performance Analysis

A. The Advantages of Aggregate FQ

It is interesting to find, from our simulation results, that the average throughput under aggregate scheduling is even (slightly) higher than that under per-flow fair queueing. This can be explained as follows: suppose at a node there are n flows, each with reserved rate r_i $(1 \leq i \leq n)$, and some other flows with total reserved rate $\hat{R}$. In the case of aggregate FQ, these n flows are bundled into an aggregate with reserved rate $R = \sum_i^n r_i$. If all the flows are backlogged at the same time, then in the per-flow case, the weight of each of the n flows is $\frac{r_i}{R+\hat{R}}$; in the aggregate FQ case, the weight of each flow is also

$$\frac{R}{R+\hat{R}} \cdot \frac{r_i}{R} = \frac{r_i}{R+\hat{R}}. \tag{1}$$

However, if not all of the n flows are backlogged at the same time, and the total rate of the absent flows is r', then the weight of flow i in the per-flow case is $\frac{r_i}{R-r'+\hat{R}}$; while the weight in the aggregate FQ case becomes

$$\frac{R}{R+\hat{R}} \cdot \frac{r_i}{R-r'} = \frac{r_i}{R+\hat{R}} \cdot \frac{R}{R-r'} > \frac{r_i}{(R-r')+\hat{R}}. \tag{2}$$

In other words, flow i implicitly has a larger weight due to the absence of some other flows in the same aggregate. We call this *aggregation advantage*.

If all the flows in the aggregate are backlogged, but some other flows are absent (the total weight of which is also r'), in the aggregate FQ case the weight of flow i becomes

$$\frac{R}{R+(\hat{R}-r')} \cdot \frac{r_i}{R} = \frac{r_i}{R+(\hat{R}-r')}, \tag{3}$$

which is the same as in the per-flow FQ case.

In summary, flows using aggregate FQ have an advantage when other flows in the same aggregate are absent, and do not have any disadvantage in all other cases. This advantage makes the new aggregate FQ scheme more attractive. Also, since TCP traffic is usually bursty, it is very likely that all of the flows in the same aggregate are not backlogged at the same time, especially when the buffer size is small. Thus, aggregate FQ is good for TCP traffic.

B. Fairness Analysis

As was pointed out in [13], there are two classes of algorithms related to congestion control: packet scheduling and buffer management. Packet scheduling decides which packet to send next, and thus, determines the allocation of bandwidth; buffer management determines buffer allocation and packet queue length. From [14] we know that the maximum window size of flow i in steady state can be described as:

$$W_i = \tau \cdot C_i + B_i + 1, \tag{4}$$

where τ is the propagation delay, C_i and B_i are the bandwidth and buffer size available to flow i, respectively.

From this equation, given a set of flows, with per-flow FQ plus per-flow buffer management, both C_i and B_i are guaranteed. In contrast, in the case of FIFO and global buffer management, although the total bandwidth and buffer size are constant, a given flow could have no buffer space and hence no bandwidth at all, especially when UDP traffic is competing with TCP for resources. This is the reason per-flow FQ plus per-flow buffer management provides superior performance. Aggregate FQ lies between per-flow FQ and FIFO queueing. Although each aggregate receives guaranteed bandwidth and buffer space, it is difficult to guarantee how much buffer and bandwidth each flow *within* the aggregate will receive. Therefore, its fairness is worse than per-flow FQ. Using ALQD+, we can alleviate this unfairness problem.

VIII. Conclusions and Future Work

In this paper, we studied TCP performance under aggregate FQ. We first showed the livelock problem of FQ under the DropFront scheme, and proposed a simple solution to the problem. Then, to improve the fairness of aggregate FQ, we proposed a new active buffer-management scheme called ALQD+, which is farer than ALQD. Based on these two enhancements, via simulation, we showed that aggregate FQ provides not only excellent scalability (small scheduling and state-maintenance overheads) but also good performance, especially throughput. In addition to the simulation-based evaluation, we provided a simple performance analysis.

We still need to find ways to fine-tune the parameters of ALQD+. Also, it would be useful to study the TCP performance under two-way traffic and asymmetric paths, where ACK packets can be congested and delayed. These are matters of our future inquiry.

References

[1] B. Suter, T. V. Lakshman, D. Stiliadis, *et al.*, "Design considerations for supporting TCP with per-flow queueing," in *Proc. of IEEE INFOCOM'98*, Mar. 1998, pp. 299–306.

[2] S. Keshav, "Congestion control in computer networks," Ph.D. dissertation, Univ. of California (Berkeley), Sept. 1991.

[3] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: The single node case," *IEEE/ACM Trans. Networking*, vol. 1, no. 3, pp. 344–357, June 1993.

[4] J. A. Cobb, "Preserving quality of service guarantees in spite of flow aggregation," *IEEE/ACM Trans. Networking*, vol. 10, no. 1, pp. 43–53, Feb. 2002.

[5] W. Sun and K. G. Shin, "Delay bounds for end-to-end traffic aggregate under guaranteed rate scheduling algorithms," Dept. of EECS, Univ. of Michigan, Tech. Rep. CSE-TR-484-03, 2003.

[6] A. Legout and E. W. Biersack, "Revisiting the fair queueing paradigm for end-to-end congestion control," *IEEE Network*, vol. 16, no. 5, pp. 38–46, Sept./Oct. 2002.

[7] F. M. Anjum and L. Tassiulas, "Fair bandwidth sharing among adaptive and non-adaptive flows in the Internet," in *Proc. of IEEE INFOCOM'99*, Mar. 1999, pp. 1412–1420.

[8] D. Awduche, J. Malcolm, J. Agogbua, *et al.*, "Requirements for traffic engineering over MPLS," RFC 2702, Sept. 1999.

[9] T. V. Lakshman, A. Neidhardt, and T. J. Ott, "The drop from front strategy in TCP and in TCP over ATM," in *Proc. of IEEE INFOCOM'96*, Mar. 1996, pp. 1242–1250.

[10] W.-C. Feng, K. G. Shin, D. D. Kandlur, *et al.*, "The BLUE active queue management algorithms," *IEEE/ACM Trans. Networking*, vol. 10, no. 4, pp. 513–528, Aug. 2002.

[11] "ns2 Simulator." [Online]. Available: http://www.isi.edu/nsnam/ns/

[12] C. Estan, G. Varghese, and M. Fisk, "Counting the number of active flows on a high speed link," UCSD, Tech. Rep. CS2002-0705, 2002.

[13] B. Braden, D. Clark, J. Crowcroft, *et al.*, "Recommendations on queue management and congestion avoidance in the Internet," RFC 2309, Apr. 1998.

[14] T. V. Lakshman and U. Madhow, "The performance of TCP/IP for networks with high bandwidth-delay products and random loss," *IEEE/ACM Trans. Networking*, vol. 5, no. 3, pp. 336–350, June 1997.