Approximate Fairness Through Limited Flow List

Addisu Eshete and Yuming Jiang
Centre for Quantifiable Quality of Service in Communication Systems*
Norwegian University of Science and Technology, Trondheim, Norway
addisu.eshete@q2s.ntnu.no ymjiang@ieee.org

***Abstract*—Most of router mechanisms proposed for fair bandwidth sharing lack either (1) simplicity due to complexity of intricate per flow management of all connections (e.g., WFQ, SFQ), (2) heterogeneity due to a design targeting a specific traffic type, e.g., RED-PD and Fair RED (FRED) or (3) robustness due to requirement for proper router configurations (e.g., CSFQ). All of these severely impact the scalability of the schemes. This paper proposes a novel router fairness mechanism, namely Approximate Fairness through Partial Finish Time (AFpFT). Key to the design of AFpFT is a *tag* field the value of which defines the position of the packet in an aggregate queue shared by all flows. The specific of tag computation depends on the router's role—edge or inner—to the flow. While gateways closest to traffic source manage all flows, successive or inner routers only manage a limited subset at flow level. The managed flows are usually of higher rates than fair share. Following the heavy-tailed Internet flow distribution, these flows are indeed the minority in the Internet. Using extensive simulations, we show that the scheme is highly fair and potentially scalable unlike other proposed schemes.**

1. Introduction

The traditional way to achieve congestion control has been with the help of congestion avoidance algorithms implemented at the end hosts. These schemes, however, require all users to adopt them without offering any incentives for doing so. A user can end up with more bandwidth without using these schemes. In addition, a lot of emerging applications (e.g. multimedia, VoIP) do not implement them and therefore do not backoff when given congestion notifications. Researchers have therefore looked for router schemes to protect the Internet from severe congestion. Broadly speaking, such router schemes can either be *per flow fair queueing (PFFQ)* (e.g., [3], [7], [8]) or *per flow dropping* (e.g. [12]) algorithms. The PFFQ algorithms maintain a separate FIFO queue for each flow and, at each transmission epoch, decide the next packet to send from among the backlogged queues (or flows). Since flows are confined to their own queues, they can be protected and their service requirements be fulfilled in fine time scales. The second approach, using per flow dropping, is based on a simpler design with a single FIFO queue and it makes use of per flow accounting to determine the connections from which packets are dropped.

While per flow fair queueing schemes are powerful in providing intricate flow level fairness, this comes with a heavy price: complexity due to buffer partitioning and the amount of information that must be maintained for each of a possible million of flows. This raises questions about their feasibility in high speed implementations. Taking the well-known Weighted Fair Queuing (WFQ) [3] as an example, there is a need to keep information for all flows traversing the router for both the *packet system* and the fluid system that the WFQ is designed to emulate. The buffer at the router's output port must also be partitioned into separate queues, one for each flow served by the router. Another example is Start-time Fair Queueing (SFQ) [8]. In SFQ, to compute the start tag of every arriving packet, the server needs to hold the finish tag of the previous packet. The majority of Internet flows are "Web mice" that last for few round trip times. Continuously updating state for all these short-lived flows is very impractical.

Stochastic Fair Queueing [14] and Deficit Round Robin [19] are efficient and less complex designs of per flow queueing. In their implementations, packets are classified into a smaller number of queues by the use of hash functions. Still, to approach the FQs performance and avoid hash collisions, the number of logical queues (buffer partitioning) must be in the order of thousands.

Is it possible to retain the nice fairness qualities of per flow schemes without keeping states for all flows? Replicating the fairness of PFFQ with no states is a difficult challenge [10]. This is because the per flow parameters (e.g., the virtual flow finish times) are not stand-alone but depend on all flows traversing the router [10]. Due to this dependency, encoding the per flow parameters at the ingress nodes, and using them later for scheduling inside the network is not possible. Therefore, several attempts—with some degrees of success—have been made in the literature to approximate the fairness of PFFQ with limited flow states in core routers [12], [13], [16], [17], [21]. One novel approach is *Core Stateless Fair Queueing* (CSFQ) [21] where routers are configured as edge or core. Edge routers keep each flow's state and estimate the flow's arrival rate r_i and encode this information into the packet header. The encoded flow rate is later retrieved at the core routers and, together with the max-min fair share r_{share}[1], determines the flow's dropping probability as $\max(0, 1 - r_{share}/r_i)$. As the core routers do not maintain any per flow state, they are stateless. However, they still need to insert the newly computed outgoing rate $\min(r_i, r_{share})$ into the packet header and pass it on to the next router. Hence, the computation of fair share, and hence flow dropping rates, is based on implicit trust of

*"Centre for Quantifiable Quality of Service in Communication Systems, Centre of Excellence" appointed by The Research Council of Norway, funded by the Research Council, NTNU and UNINETT. http://www.q2s.ntnu.no

[1]For a congested router with output link capacity C and serving n flows, the fair share rate r_{share} is the solution to the condition $C = \sum_{i=1}^{n} \min(r_i, r_{share})$, where r_i is the incoming rate of flow i.

upstream nodes in the network. A faulty router along the flow's path can therefore insert inconsistent values into packet headers and severely undermine the fairness, and the overall performance, of CSFQ [22]. Therefore, CSFQ can neither transcend network boundaries nor withstand malfunctioning or wrongly configured routers. Besides, the flow dropping mechanism based entirely on incoming flow rate and the fair share at the router may not be suitable for closed-loop traffic, see Sec 3-B. Also, CSFQ can be unfair to long RTT flows (see Fig. 7).

In CHOKe [17], when a new packet arrives to the queue, it is compared to packets randomly drawn from the queue. If they belong to the same flow, all of them are dropped, else the randomly chosen packets are left intact and the arriving packet is dropped with a probability. The idea is that an ill-behaved flow has more packets in the buffer and hence a higher probability of dropping. The results in [17] show that the high rate flows can still achieve much more than the fair share.

As will be discussed in Sec 2-D, the high rate, large sized flows are the minority in the Internet. Both Approximate Fairness Dropping (AFD) [16] and RED with Preferential Dropping (RED-PD) [13] use the notion of partial flow state—state only for the "heavy-hitting" subset—to help provide fairness. The manner of identifying these flows is different. AFD keeps record of recent arrivals to find the flow rates, and the flow states to be kept is proportional in size to that of the buffer. But, the authors concede that fairness performance of AFD is not as good as that of CSFQ. RED-PD uses recent RED's drop history and TCP-friendliness criterion to identify the high bandwidth flows. An implicit assumption in the design is that all sources comply with a standard TCP implementation. A critical RED-PD configuration parameter is the target RTT R. Flows with more drops in the drop history than a TCP flow with RTT R are monitored and controlled at a prefilter. While the quality of fairness increases with R, this also increases the amount of flow state to be maintained.

This paper presents a router fairness scheme called *Approximate Fairness through partial Finish Time (AFpFT)*. Detailed design is presented in Sec 2, but main features are summarized here. An AFpFT queue is an aggregate queue shared by all flows, and sorts arriving packets based on encoded timestamps, hereafter called *tags*. An AFpFT router can act as edge or core to a flow, but there is no hard-and-fast router configuration per se. At network entrance of the flow, the first router acts as an edge and maintains the flow information. All downstream routers act as core or inner routers and keep only *partial* flow states; by partial, we mean information only about those flows that already have packets in the buffer. We call these flows *listed flows*, and their flow information or state is stored as records in a flow list **FL**. The size of **FL** is proportional to the memory allocated to the packet buffers, just as in AFD. By keeping the state for a limited subset of flows at inner nodes, AFpFT enables routers with less stateful and less complex but still powerful flow protection and fair bandwidth sharing. Extensive simulations under different operating conditions show that the mechanism is superior in fairness to other related schemes such as FRED and CSFQ. It can, for instance, restrict the high bandwidth flows (e.g., TCP flows with small round-trip times or aggressive flows that lack e2e congestion control) to a common fair share. Incidentally, the listed flows at core routers are usually such high rate flows, see Sec 2-D.

The rest of the paper is organized as follows. Detailed design of AFpFT is provided in the next section. Section 3 provides context on AFpFT's performance by comparing it with RED, FRED and CSFQ under different simulations conditions. Finally, conclusions are presented in Sec 4.

2. Detailed Design of AFpFT

The pseudocode of AFpFT is shown in Fig. 1, and notation of important variables is summarized in Table I. Central to AFpFT design are a packet header field called start tag, or simply *tag*, and a non-FIFO shared queue **Q**. Packet tags are routinely initialized to negative values at the source. When a packet arrives to an AFpFT queue **Q**, this is the sequence of router actions that would happen (details follow subsequently): first, a new tag value is computed for the packet and encoded into the packet's header; second, the flow of the arriving packet may be added to flow list **FL** or its flow record in flow list **FL** may be updated; third, the packet is then queued in **Q** in increasing order of its tag value; fourth, if the queue becomes full, the packet(s) at the tail are dropped and flow record(s) of dropped packet(s) may need to be updated. Note that tag computation is dependent on a local server variable called *virtual time* $v(t)$; $v(t)$ is the *tag* of the packet being transmitted at time t. We describe design details as follows.

TABLE I: Notations used in AFpFT tag computation.

Variable	Semantics
p_f^j	j^{th} packet of flow f
r	rate allocated to flows (also called flow weight)
$S(p_f^j)$	start tag or *tag* of packet p_f^j
$F(p_f^j)$	finish tag of packet p_f^j. For a start $F(p_f^0) = 0$.
$v(A[p_f^j])$	server's virtual time when p_f^j arrives to queue
Q	shared queue that enques packets based on *tag*

A. Tag computation

Packet tags define the position of the packet in the queue. They are computed upon packet arrival and encoded into packet header. Based on whether or not the packet's flow entry can be found in the flow list[2], there are two ways[3] to compute tags.

Tag Encoding 1: When a packet arrives, the server examines its flow list for the finish tag. If the flow is *listed* (e.g., flow entry is found), tag computation is borrowed from SFQ ([8]) and is given by Eq. 1 below (see Fig 1, Algorithm 1, line 9).

1) On arrival of packet p_f^j, $S(p_f^j)$ and $F(p_f^j)$ are:

[2]While any source-destination combination obtainable from a packet header can be used, we define a flow in this paper as a stream of packets with matching source and destination addresses and ports. After a period of inactivity, idle flows expire and are removed from the flow list.

[3]The need for two different tag computations is presented in Sec 2-D.

$$S(p_f^j) = \max(v[A(p_f^j)], F(p_f^{j-1})) \quad j \geq 1 \tag{1}$$

$$F(p_f^j) = S(p_f^j) + \frac{l_f^j}{r} \quad j \geq 1 \tag{2}$$

2) At the start, server virtual time is 0. During a busy period, the server virtual time at t is equal to the start tag of packet currently in service. At the end of busy periods, $v(t)$ can be reset to 0.

In (2), we use r instead of r_f. This means all bottlenecked flows obtain equal share and the servers maintain a single r. The actual flow share is insensitive to the r value, but depends on the number and incoming rates of flows traversing the server. In our implementation, we use $r = 10kbps$.

Tag Encoding 2: If a flow's entry is unavailable, the tag directly assumes the virtual time of the server upon packet's arrival, i.e., $S(p_f^j) = v[A(p_f^j)]$ (see Fig 1, Algorithm 1, line 11).

Since the start tag carried over from a previous node has no significance in tag recomputation on the current node, AFpFT is more robust to router problems, unlike CSFQ (see [22]). The packet tag encoding at a router is merely based on the availability of flow's entry in the list. Whether or not the router maintains the flow defines the router's role as explained below.

B. Router roles

When a router sees a *negative* packet tag value, it assumes it is ingress to the flow in question and adds the flow to the list and updates the per flow information, i.e., finish time of previous packet. If the tag value is nonnegative, the router assumes the value is encoded at upstream nodes and therefore acts as an inner node to the flow (see Fig 1, Algorithm 1, lines 5–9). In that case, the router only maintains flow information if the packets of that flow have been queued in the buffer.

Summarizing Secs 2-A and 2-B: computation of tag $S(.)$ takes two forms: (1) the per flow computation given by Eq. 1 for all flows at the edge and for flows with buffered packets in inner routers; (2) $S(p_f^j) = v[A(p_f^j)]$, otherwise.

C. Buffer Management

When the buffer becomes full, we choose to drop packet(s) with the highest *tag(s)*, found at the tail of the buffer (see Fig 1, Block A). Since the packet just enqued (line 12) could be larger than the packet(s) at the tail, more than one packets may need to be dropped. When a packet of a flow is dropped, its flow record, if any, should be updated. In case the router acts as inner node to the flow, and there are no any other packets of this flow in the buffer (i.e., $count_i \leq 0$), the flow record is removed (Fig 1, Algorithm 1, lines 21–22, and also in Algorithm 2, lines 9–10). Probably the trickiest part is the correction of the flow finish time when a packet is dropped (line 20). For ease of understanding, consider a congested router serving well-behaved flows and an aggressive high rate flow f. Flow f often finds its packets queued near the tail, as will be explained shortly. When a flow f packet arrives, it first updates $finish_f$ in the flow list before being queued near the tail and getting dropped because of congestion.

Flow List FL Variables:
$count_f$: count of flow f packets in **Q**
$finish_f$: finish time of flow f
Functions:
$tag(p)$: start tag field in packet p's header
$edge(p)$: $tag(p) < 0$ (Is router edge?)
$v(t)$: server virtual time at t
$conn(p)$: flow/connection id of packet p

Algorithm 1: **Enqueing**

```
1: Upon receiving $p_f^j$ at $t$
2: if ($f \notin$ FL) then
3:     add $f$ to flow list FL
4: $count_f \Longleftarrow count_f + 1$
5: if ($edge(p_f^j)$ or ($f \in$ FL and $count_f \geq 2$)) then
6:     update $f$ variables in FL
7:     compute $S(p_f^j)$ // Eq. 1
8:     compute $F(p_f^j)$ and save in $finish_f$ // Eq. 2
9:     $tag(p_f^j) \Longleftarrow S(p_f^j)$ // encoding type 1
10: else
11:     $tag(p_f^j) \Longleftarrow v(t)$ // encoding type 2
12: enque $p_f^j$ into Q // based on $tag(p_f^j)$ value
13:
14: // Block A—buffer management when queue becomes full.
15: while (Q-size > Q-limit) do
16:     draw packet $p$ from Q tail
17:     $i \Longleftarrow conn(p)$
18:     if ($i \in$ FL) then
19:         $count_i \Longleftarrow count_i - 1$
20:         $finish_i \Longleftarrow finish_i - length(p)/r$
21:     if (!$edge(p)$ and $count_i < 1$) then
22:         remove flow $i$ from FL
23:     drop $p$
```

Algorithm 2: **Dequeuing**

```
1: draw $p$ from Q head
2: if ($p$ exists) then
3:     // Block B— $p$ is packet in service
4:     extract $tag(p)$ from $p$ header
5:     $v(t) \Longleftarrow tag(p)$
6:     $i \Longleftarrow conn(p)$
7:     if ($i \in$ FL) then
8:         $count_i \Longleftarrow count_i - 1$
9:     if (!$edge(p)$ and $count_i < 1$) then
10:         remove flow $i$ from FL
11: else
12:     // Block C— Q empty; reset server & flow params
13:     $v(t) \Longleftarrow 0.0$
14:     $\forall j \in$ FL : $finish_j \Longleftarrow 0.0$, $count_j \Longleftarrow 0$
15: return $p$
```

Fig. 1: Enque and Dequeue in AFpFT

In the long run, the flow's finish time no longer indicates the flow's transmission progress, but its packet "obituaries" instead. Therefore, when a packet p of a flow is dropped, since its contribution to the flow's throughput is *zero*, the flow's state (e.g., $finish_f$) should be as if the packet had never arrived to the queue in the first place. In short, we must cancel out the contribution of p's arrival to the flow finish time. Finding the exact contribution is generally not possible because the $finish_f$ updating is done upon p's arrival, usually much earlier than the time of its dropping and we do not have information about the previous values of $finish_f$. Therefore, we fairly approximate this contribution by $length(p)/r$ (see line 20). Before we end this section, we provide an example showing the fairness power of AFpFT and the importance of this correction (see Fig. 2).

Example: Figure 2 shows the result of 20 CBR sources sending at four different flow rates of 0.5Mbps, 1Mbs, 1.5Mbps, and 2Mbps for a total traffic rate of 25Mbps. The capacity of the AFpFT link is 20Mbps. Flow groups 3 and 4 send more than the fair share rate of 1.25Mbps. As can be seen, without applying Algorithm 1, line 20, the flows in the last group are punished while the third flow group is transmitting at full incoming rate (see Fig. 2a). With the introduction of the patch, both high rate flow groups are exactly limited to their max-min fair share (see Fig. 2b).

D. Rationale for AFpFT

Generally, designing a fairness scheme on a par with PFFQ without keeping per flow state is almost an impossible task [10]. Therefore, AFpFT's aim is to approximate fairness with partial flow information maintained at inner nodes. A key observation useful for the fairness approximation is that packets already buffered in **Q** at any time t must all have tags greater than the server's virtual time *v(t)*. For a flow f and its packet p_f^j which arrives at the server at time t, consider the two possibilities:

1) (High bandwidth f) It is more likely that the flow is listed as it may have sent many packets and some are still in the buffer. In this case, flow information is maintained at the server and the tag computation is given by Eq. 1.
2) (Flow f sending at or less than fair share[4]) When p_f^j arrives, all its previous packets may have left the server and the flow is unavailable in the flow list. The fact that previous packets have left at time of p_f^j arrival indicates that the start tags of the departed packets must have been less than the virtual time, i.e., $S(p_f^{j-1}) \leq v(A(p_f^j))$. We also boldly estimate $F(p_f^{j-1}) \leq v(A(p_f^j))$, which in turn implies that $S(p_f^j) = v(t)$ (Fig 1, Algorithm 1, line 11). The packet is therefore placed at or near the head of queue, getting priority over other (high bandwidth) flows with buffered packets.

Most backbone routers in the Internet have a bandwidth-delay product of buffering for each link [1] where delay here refers to round-trip-delay. An implicit assumption useful for AFpFT design is that the number of high bandwidth flows during a window of round trip delay and the buffer size (in packets) should be of the same order of magnitude. To clarify the need for the above assumption, consider a scenario where we have a congested router with a mix of both well-behaved and high bandwidth flows. Assume the buffer size is less than the number of high bandwidth flows. As a consequence, per flow state of some of the high bandwidth flows will not be maintained and therefore the vast majority of their packets use $S(p) = v(t)$ tag encoding (Fig 1, Algorithm 1, line 11) and compete with the well-behaved flows. This has a potential to impair the fairness of AFpFT. Fortunately, there are strong arguments that these high bandwidth flows are very few. A related but entirely independent work [11] using real traces of commercial networks shows that the number of listed (a.k.a. *bottlenecked*) flows is much smaller than the total number of flows in progress. In addition, numerous measurement studies ([13], [16], [18], [23]) report that Internet flow distribution is skewed with respect to sizes, and rates of flows. Accordingly, a large proportion of Internet flows are indeed short-lived web traffic that end up in TCP slow start phase. The majority of bytes are accounted for by very few high bandwidth flows. Further, flow rates and flow sizes are also strongly correlated [23]. The high rate flows passing through a particular router at a specific time instant will be even fewer. Using AFpFT, therefore, we only need to allocate buffer slots for (or equivalently add to flow list) the few "heavy-hitters". By keeping state only for these flows, AFpFT makes sure that the few relatively high rate flows cannot get more than the fair share. In addition, the small flows often use the $S(p) = v(t)$ to get a service ahead of the high rate flows. Giving time and space priority to small flows is usually taken as a good design principle. AFpFT inevitably fulfills that objective without making it an explicit design goal.

[4]Fair share is a function of the number and rates of flows, see footnote 1.

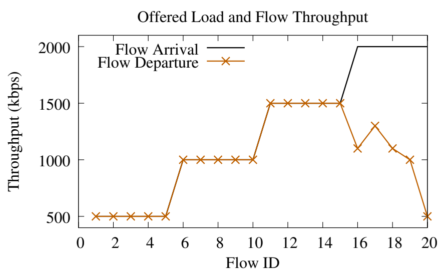
3. Performance Evaluation

A. Topologies and Parameters

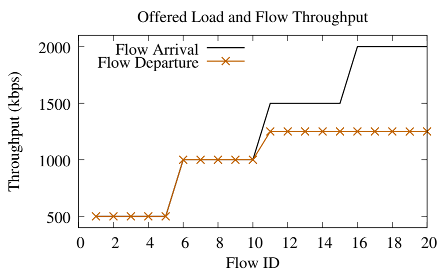
We implemented AFpFT in ns-2. Unless otherwise stated, the topologies shown in Fig. 4, and simulation parameters summarized in Table II are used for evaluation. AFpFT flow fairness and link utilization are compared with Adaptive RED [6], FRED [12], and CSFQ [21]. RED and development of its control parameters is summarized in an earlier related work [4, §3]. FRED and CSFQ implementations are freely available [20]. Quite briefly, FRED fairness mechanism is through fair allocation of buffer space to flows, while CSFQ adopts per flow dropping rates $(r_i - r_{share})/r_i$ to bring down the outgoing rates of bottlenecked flows to r_{share}. K and K_α are averaging constants (time windows) used for estimating incoming flow rates and the fair share r_{share}, respectively.

B. Single Link Case

Fig. 4a shows 32 long-lived TCP (New Reno) flows and a single CBR flow competing for a scarce 1Mbps bottleneck. CBR flow sends at full bottleneck capacity of 1Mbs, and the TCP windows are unlimited. The bottleneck buffer size is 100kB. Figures 3 shows the results averaged over 30 replications. The max-min fair share in the above scenario is 30.3kbps per flow. AFpFT is an extremely fair scheme, providing an average of 29.5 kbps to each TCP flow, close to the ideal fair flow share. It also manages to restrict the nonadaptive UDP flow to the fair share. The same cannot be said for RED with Drop Tail which allows unfair link domination (in excess of a whopping 70%) by the unresponsive flow. Average TCP flow throughput is a meagre 8.4 kbps in RED. All flows



(a) AFpFT without $finish_f$ adjustment.



(b) AFpFT with $finish_f$ adjustment.

Fig. 2: Offered Load and Flow Throughput of 20 CBR sources under AFpFT.

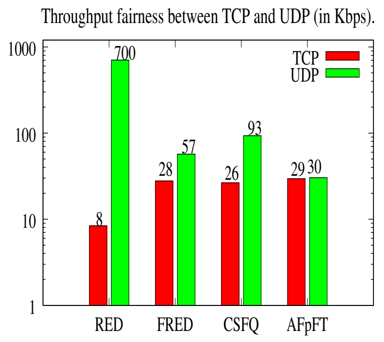
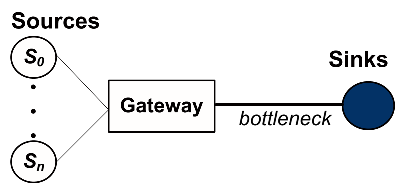
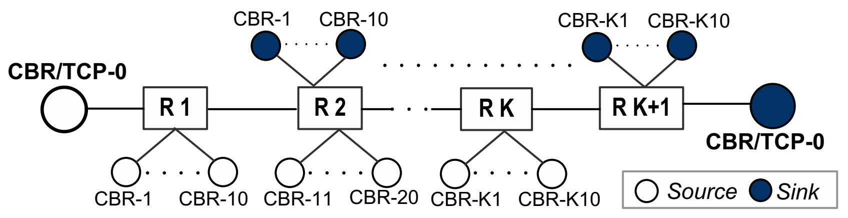


Fig. 3: UDP and average TCP throughput.



(a) Single Congested Link.



(b) Multiple Congested Links.

Fig. 4: Topology used for evaluation.

TABLE II: Default Settings used for Evaluation.

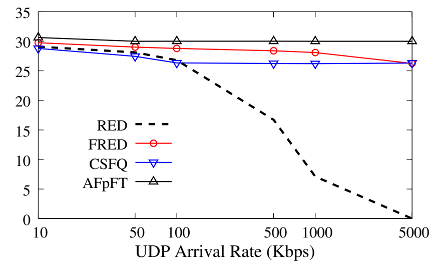
General Parameters		Algorithms	
Link and Buffer		**RED**	
Link speed	1Mbps	$\min_{th}$	25% Buf lim.
Prop. delay	1ms	$\max_{th}$	75% Buf lim.
Buf. limit	50kB	*adaptive*	yes
Traffic Source		**FRED**	
TCP version	New Reno	$\min_{th}$	25% Buf lim.
TCP segment size	960 B	$\max_{th}$	75% Buf lim.
UDP packet size	1000 B	**CSFQ**	
Flow start time	$t \in [0.0, 5.0)$	K	100 ms
Simulation		K_α	200 ms
Simul. Duration	50s	Buf thresh.	50% Buf lim.
Results taken	2nd half	Flow weights	Equal
Replications	30 or 100	**AFpFT**	
Confidence level	90%	Flow weights r	10kbps

are uniformly punished even though only UDP is responsible for congestion. FRED significantly improves upon RED due to its per flow dropping rates. Nevertheless, UDP still gets twice as much as an average TCP flow (57kbps vs 28 kbps). In addition, despite the fact that all TCP flows are identical with respect to round trip times, congestion control algorithm and receiver window sizes, a maximum difference of 12kbps ($\approx$ 40% of ideal share) is noted between TCP flow throughput values in FRED. Using CSFQ and AFpFT this difference is, however, less than 3kbps and 1kbps, respectively. The Jain's fairness index[5] quantifies the overall fairness between these identical TCP flows: 0.9329 (RED), 0.9905 (FRED), 0.9994 (CSFQ) and 0.9999 (AFpFT). While CSFQ is better than FRED in fair distribution of bandwidths among TCP flows, its performance with regard to TCP bandwidth quota is generally poorer than FRED. Average TCP flow throughput is 26kbps and UDP takes up more than 3 times the fair share. This may largely be explained by the inadvertent packet dropping scheme employed by CSFQ and its consequential impact on TCP throughput. The per flow dropping rate is based entirely on the incoming flow rate r_i and fair share r_{share} and is given by $\frac{r_i - r_{share}}{r_i}$. The aim is to limit the throughput of a congested flow i to the fair share r_{share}. Dropping with the above rate can, due to the unresponsive nature of UDP, successfully bring down UDP flow output rates to r_{share}. However, dropping TCP flows with the above rate[6], rather than limit the flow rate to the fair rate r_{share}, may potentially reset the congestion windows. This occasional but unfortunate situation can degrade average TCP flow throughput.

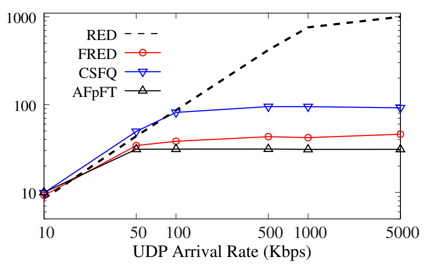
What happens to TCP throughput if we vary the incoming rate of the UDP flow? In Fig 5, the UDP source rate increases by a factor of 500 along the x-axis. Most values are in logarithmic scale. Performance difference becomes clearer with higher incoming UDP rates. RED is very poor at restricting high bandwidth flows. The linear RED curve in 5b indicates that the UDP share is a linear function of the incoming UDP rate. When the UDP arrival rate is 5Mbps, over 99.99% of link capacity is used by UDP, completely starving out the well-behaved TCP flows. As before, CSFQ offers approximately

[5]For a network of n connections, Jain's fairness index $f = (\sum x_i)^2/(n\sum x_i^2)$, where x_i is the resource share of connection i [9]. An ideal fair system has a score of 1.0.

[6]A TCP friendly drop rate in the steady state should consider, among others, the window limits and round trip delays ([5], [13], [15]).



(a) Average TCP Throughput Comparison.



(b) UDP Throughput Comparison.

Fig. 5: Average TCP and UDP Throughput in kbps as incoming UDP rate is varied.

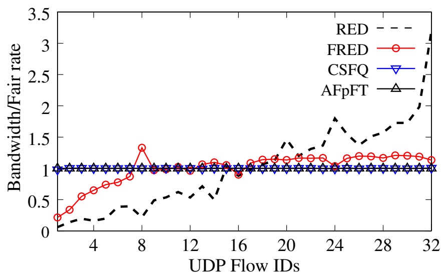


Fig. 6: Normalized throughput allocated for UDP flows.

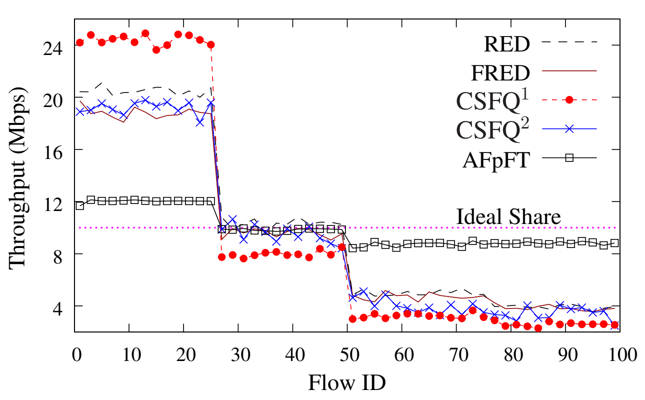


Fig. 7: Flow Throughput Fairness over a Gigabit link.

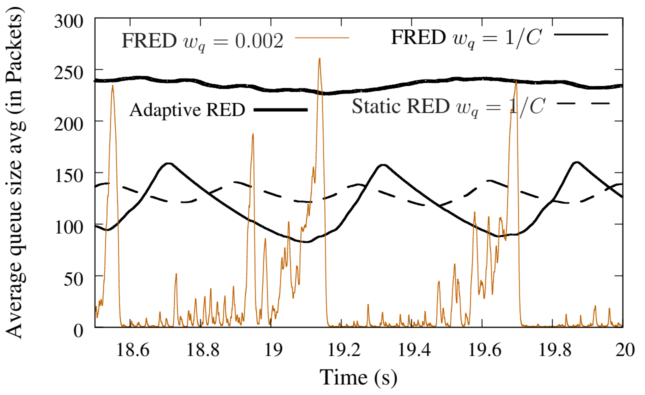


Fig. 8: Impact of *parameter sensitivity* on FRED's link utilization.

100kbps to UDP which is several times larger than the TCP share. While FRED is better at controlling high bandwidth flows than CSFQ, the TCP share slightly decreases with increasing UDP traffic rate. AFpFT provides precisely the same fair share to TCP at all levels of incoming UDP noise.

Another set of experiments consisting of only CBR flows is carried out. A total of 32 flows (with flow i sending at $i \times 0.3125$Mbps) are simulated. The results are shown in Figure 6. Following our argument earlier in this section, this all UDP scenario is the natural environment for CSFQ's fair bandwidth allocation. AFpFT performs as good as CSFQ, allocating exactly the fair share of 0.3125Mbps to all flows even though the last flow, for instance, sends 32 times as fast as the first one. FRED, by contrast, fails to deliver the fair bandwidth allocation; indeed, the maximum allocation can be several times larger than the minimum allocation.

C. Link Scalability and Different RTTs.

This section considers fairness when the TCP flows have largely distinct RTT delays in a high-speed environment. We increase the link speed to 1Gbps and its delay 5ms. Now we have a total of 100 TCP flows divided, based on their RTTs, into equal groups of size 25: G1, ..., G4. The RTTs are respectively 20ms, 40ms, 80ms, and 100ms. The queueing delay at a gigabit link is insignificant. Flow i belongs to G$\lceil i/25 \rceil$. We apply caution here when dimensioning the buffer size. Optimal buffer sizing is still an active field of networking research and the bandwidth-delay product is usually used as a general rule of thumb to provision buffer size [1]. We use 500 packets as our buffer size—a fraction of the bandwidth-delay product here. Under current routers (FIFO with Drop Tail), average TCP throughput is inversely proportional to RTT [15]. Then the flows in groups $G1, \ldots, G4$ attain throughput r_1, $r_1/2$, $r_1/4$, $r_1/5$, respectively, where G1 flow throughput $r_1 = 21.33Mbps$. Figure 7 illustrates per flow throughput under different schemes.

First, the most unexpected of the observations: with default $K = 100ms$ and $K_\alpha = 200ms$, CSFQ performance, labeled CSFQ^1 in the figure, is worse than RED and FRED. Using RED, for instance, the average throughput per G1/G4 flow are 20.6/3.9 Mbps, respectively. Corresponding CSFQ values are 24.3 and 2.6 in Mbps. We believe that the averaging constants are too relaxed for CSFQ to be able to accurately estimate and control the rates of low RTT high speed TCP flows. With tighter window constants of $K = 20ms$ and $K_\alpha = 40ms$, CSFQ fairness shown as CSFQ^2 is much better, e.g., the average G1/G4 per flow throughput are 19 Mbps and 3.4 Mbps. Of all schemes, however, only AFpFT is significantly fairer: G1 flows obtain only 20% more than the fair share and G4 flows receive on average 12% less than the fair share. We find that FRED is unfair and only marginally better than RED. G1 flows in FRED receive on average 4.8× than those of G4. Therefore, fairness in a network with a mix of different RTT flows, as is the case in the Internet, is very poor with FIFO, RED, FRED and CSFQ. In addition, FRED produces both the least total throughput and total link utilization of all schemes. In steady state, we obtain the following link utilizations: RED (99.9%), FRED (92.7%), CSFQ (95.7%) and AFpFT (98.9%).

The RED version imbedded in FRED is to blame for the poorer link utilization of FRED. FRED was proposed much earlier than the *adaptive* and *gentle*[7] parameters introduced to enhance RED's performance. FRED may therefore suffer from poor queue utilization caused by sensitivity to the parameters (see [6, §5.1]). The default FRED values $\max_p = 0.1$ and $w_q = 0.002$ are too conservative and not optimal for the scenario in hand. RED's recommended queue averaging constant w_q, for instance, is $1 - exp(-1/C)$, where C is the link speed in $packets/sec$. Since C is very high—1.25×10^5 in this case—we can fairly approximate it as w_q[8] $\simeq 1.0/C$. Without the *adaptive* feature, the static parameters do not allow FRED to operate optimally. Adaptive RED, on the other hand, can improve throughput performance by self-tuning, enabling it to maintain a target *avg* away from $\max_{th}$. And *gentle* smoothly increases the dropping probability p_b when the *avg* exceeds $\max_{th}$. As a consequence, queue utilization in FRED, and generally in non-adaptive RED, is poor. Figure 8 shows the link utilization curves for non-adaptive RED, Adaptive RED with *gentle*, FRED with default w_q, and FRED with $w_q = 1/C$ all in a typical run of heavy load scenario. Adaptive RED always attempts to maintain *avg* around $\frac{1}{2}(\max_{th} + \min_{th}) = 250$. We see that the default value $w_q = 0.002$ is too large, making *avg* extremely sensitive to the actual queue size. The behavior of FRED ($w_q = 1/C$) is similar to the non-adaptive RED (with default $w_q = 1/C$), i.e. both increase *avg* in cycles corresponding to alternating periods of high loss and low loss. Hereafter, we use $w_q = 1/C$ for FRED.

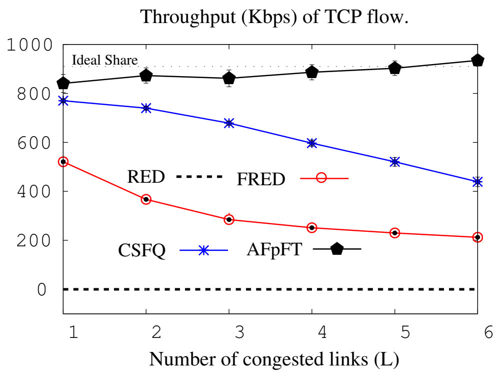
D. Multiple Congested Link

This section discusses how flow throughput fairness is affected across multiple congested links. Topology for this section is borrowed from [21] and shown Fig 4b. The links connecting hosts to routers are 20Mbps, and routers to routers 10Mbps. Each 10Mbps link is traversed by 11 flows: 10 CBR cross flows each with rates 2 Mbps and a well-behaved flow 0. In the first experiment, flow 0 is TCP. In the second experiment, flow 0 is CBR sending close to fair share of 0.909Mbps. To remove bias, we collect throughput of flow 0 as a function of the number of congested links, L, for the second half of the simulation. Flows start times are uniformly distributed on $[0, 5.0]$. Results from 100 replications are shown in Fig. 9.

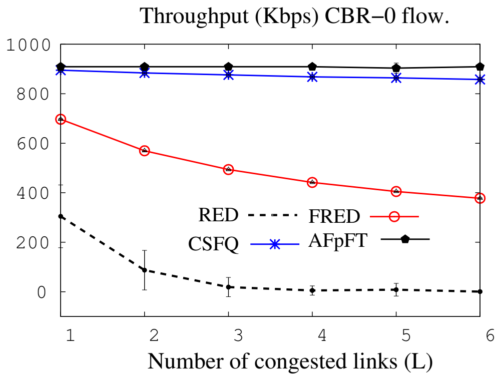
Regardless of whether the flow is TCP or UDP, its throughput decreases as it traverses multiple congested links. Since UDP does not react to congestion, its throughput is generally higher than that of TCP. There is a total lack of protection for TCP flow when RED is used, see Fig. 9a. Corresponding UDP

[7]*adaptive* and *gentle* features can improve performance. *adaptive* makes RED robust against variations in traffic conditions by auto-tuning the parameters for optimal performance. An equivalent way is to *manually* configure the RED parameters that suit the traffic conditions in the network.

[8]We use the approximation: $(1 - \frac{1}{C})^C \simeq e^{-1}$. $w_q \approx 1/C$ for all realistic cases since C is usually large enough. Following this approximation, RED's *avg* computation can then be simplified as: $avg = (1 - w_q)avg + w_q q = ((C - 1)avg + q)/C$.



(a) Flow 0 is TCP.



(b) Flow 0 is UDP with rate 1 Mbps.

Fig. 9: How flow throughput scales with number of congested links.

throughput *decays* with increasing number of links when using RED. The results here are generally consistent with those in [21, Fig. 8]. In FRED, UDP throughput decreases steadily with the number of congested links. CSFQ scales much better than both RED and FRED: It provides a stable throughput share to CBR with increasing links, and a slowly decreasing share to TCP. The only exception to the usual trend of throughput decline is AFpFT which reasonably provides the fair share to the flow no matter what the kind of flow traffic or the number of links traversed. A very important, but seemingly discrepant, observation is the fact that TCP flow throughput slightly increases with L in AFpFT. This non-intuitive observation is in sharp contrast to all other schemes, and can be explained as follows. For small L, due to smaller propagation delays, many packets may arrive before previous packets of the flow have left the queue. As explained in Sec 2-D, this causes the flow to be listed, and the arriving packets may be queued at or near the queue tail. This means there is a higher probability of packet drops and/or high queuing delays. This in turn causes timeouts and slower TCP sending rates. For large propagation delays (i.e., multiple congested links), it is the opposite: When packets arrive, there may be no previous packets in the queue. Such packets use tag encoding 2 (see Sec 2-A) and are pushed to the front of the queue as soon as they arrive. The overall result is fewer packet drops, smaller average queuing delay, and higher flow throughput.

E. Other Traffic Models

Web Traffic Model.: We consider performance of Web traffic which forms the most dominant portion of Internet traffic. Such flows are typically short-lived and often end up during the slow start phase. We model such traffic as a Poisson arrival process with an average of 25 new sessions per second, and the size of each session (file) Pareto distributed with average size of 30kB (about 30 packets) and shape parameter 1.3. The model captures the heavy-tailed (highly variable) nature of Web file sizes and their transmission times [2]. Session statistics such as mean transfer times are important for such flows. We simulate the flows together with a 5Mbps CBR flow under a dumbbell topology: 10Mbps, 1ms link with buffer size of 100 packets. The results are summarized in Table III.

AFpFT performs the best in terms of fulfilling the short transfer demands of *mice* flows. Half of the web flows finish the transfers under 50ms. The mean transfer times of flows are 640ms (RED), 150 ms (FRED), 220ms (CSFQ) and 110ms (AFpFT). A flow, on the average, completes its web transfer twice as fast in AFpFT as in CSFQ. In all schemes, over a thousand flows have completed their transfers within 50s of simulation. The number of flows that complete transfers is largest in AFpFT than in any other scheme.

TABLE III: Web Session Statistics under Different Router Schemes.

	Percentage of Flows With Transfer Times					
Scheme	< 0.05s	< 0.5s ≥ 0.05	< 1.0s ≥ 0.5	< 2.0s ≥ 1.0	< 5.0s ≥ 2.0	≥ 5.0s
RED	6.70	64.75	16.00	6.50	4.80	1.30
FRED	26.90	69.50	2.28	0.89	0.33	0.09
CSFQ	24.60	67.74	5.00	1.50	0.97	0.16
AFpFT	50.20	47.23	1.46	0.74	0.28	0.08

ON-OFF Traffic Model.: The bottleneck is now used by $N-1$ CBR sources sending at the fair rate, and 1 bursty ON-OFF source. We choose $N = 20$. The ON and OFF periods are taken from exponential distributions with means of 0.2s and $(N-1) \times 0.2s = 19 \times 0.2s$, respectively. During ON period, the ON-OFF source sends at full link capacity of 10Mbps, making it highly bursty. Then it goes idle during the OFF period. ON-OFF sources are normally challenging for AFpFT because at the start of an ON period a packet potentially arrives after packets of previous ON periods have left the buffer. Our interest here is how well the algorithms can restrict this bursty source. Of all packets sent by the ON-OFF source, 92% (RED), 22% (FRED), 28% (CSFQ) and 21% (AFpFT) have been delivered. The result confirms that AFpFT matches FRED in restricting the bursty ON-OFF source.

4. Conclusion

Designing core-stateless versions of fair PFFQ is not an easy task. The main hurdle is that computation of flow parameters is dependent on other interacting flows and it is impossible to determine these parameters at network edges. The goal of AFpFT is to approximate the fairness of per flow fair queueing algorithms with minimum states possible. Unlike some of the existing works [12], [13], we make no assumptions about the kind of traffic in the Internet, nor we explicitly define routers as core or edge as in [21]. Where flows enter the network, the first router acts as edge and keeps the flow state. This is generally a sound assumption since edge routers manage fewer flows and, being closest to the traffic sources, are ideally suited to provide flow level fairness. Inside the network where there are many more flows, we manage only a subset of those flows that have relatively high rates and, following the heavy-tailed Internet flow distribution, this subset is generally small in number. The state requirement in the core network can therefore be limited (from above by the buffer size). Extensive simulations show that the fairness performance of AFpFT is superior to related schemes such as CSFQ, RED, and FRED.

References

[1] N. Beheshti, Y. Ganjali, M. Ghobadi, N. McKeown, and G. Salmon. Experimental Study of Router Buffer Sizing. In *Proc. of IMC*, pages 197–210, 2008.

[2] M. E. Crovella and A. Bestavros. Self-similarity in world wide web traffic: evidence and possible causes. In *IEEE/ACM Trans. on Networking*, volume 5, Dec 1997.

[3] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Proc. of ACM/SIGCOMM*, 1989.

[4] A. Eshete and Y. Jiang. On the Flow Fairness of Aggregate Queues. In *Proc. of BCFIC*, pages 120–127, Feb 2011.

[5] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the internet. *IEEE Trans. on Networking*, 7(4):458–472, Aug 1999.

[6] S. Floyd, R. Gummadi, and S. Shenker. Adaptive RED: An algorithm for increasing the robustness of RED's Active Queue Management. Tech. Report, 2001.

[7] S. J. Golestani. A Self-Clocked Queueing Scheme for Broadband Applications. In *Proc. of IEEE Infocom*, pages 636–646, June 1994.

[8] P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. In *ACM SIGCOMM*, 1996.

[9] R. Jain. *The Art of Computer Systems Performance Analysis,.* John Wiley and Sons, 1991.

[10] J. Kaur and H. M. Vin. Core-stateless guaranteed rate scheduling algorithms. In *Proc. of INFOCOM*, number 1484–1492, 2001.

[11] A. Kortebi, L. Muscariello, S. Oueslati, and J. Roberts. Evaluating the number of active flows in a scheduler realizing fair statistical bandwidth sharing. In *Proc. of ACM/SIGMETRICS*, pages 217 – 228, 2005.

[12] D. Lin and R. Morris. Dynamics of random early detection. In *ACM SIGCOMM CCR*, volume 27, pages 127 – 137, 1997.

[13] R. Mahajan, S. Floyd, and D. Wetherall. Controlling high-bandwidth flows at the congested router. In *Proc. of ICNP*, pages 192 – 201, 2001.

[14] P. McKenney. Stochastic fairness queueing. In *Proc. of INFOCOM*, pages 733–740, 1990.

[15] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: a simple model and its empirical validation. In *Proc. of ACM SIGCOMM*, pages 303–314, 1998.

[16] R. Pan, L. Breslau, B. Prabhakar, and S. Shenker. Approximate fairness through differential dropping. *ACM SIGCOMM CCR*, 33:23–39, 2003.

[17] R. Pan, B. Prabhakar, and K. Psounis. CHOKe - a stateless active queue management scheme for approximating fair bandwidth allocation. In *Proc. of Infocom*, 2000.

[18] F. Qiana, A. Gerber, Z. M. Mao, S. Sen, O. Spatscheck, and W. Willinger. TCP Revisited: A Fresh Look at TCP in the Wild. In *Proc. of IMC*, pages 76–89, Nov 2009.

[19] S. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. *IEEE/ACM Trans. Networking*, 4:375–385, June 1996.

[20] I. Stoica. http://www.cs.berkeley.edu/ istoica/csfq/, Dec 2000.

[21] I. Stoica, S. Shenker, and H. Zhang. Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks. In *SIGCOMM*, pages 118—130, 1998.

[22] I. Stoica, H. Zhang, and S. Shenker. Self-Verifying CSFQ. In *Proc. of INFOCOM*, pages 21–30, 2002.

[23] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker. On the characteristics and origins of internet flow rates. *Proc. of SIGCOMM*, pages 309–322, 2002.