APPROXIMATE FAIR ALLOCATION OF LINK BANDWIDTH

APPROXIMATE FAIR DROPPING (AFD), AN ACTIVE QUEUE MANAGEMENT SCHEME, ALLOCATES LINK BANDWIDTH IN AN APPROXIMATELY FAIR MANNER. AFD-NFT, AN ENHANCEMENT TO AFD, PERFORMS SIMILARLY AND IS MUCH EASIER TO IMPLEMENT.

Rong Pan
Balaji Prabhakar
Stanford University

Lee Breslau
AT&T Labs—Research

Scott Shenker
International Computer Science Institute

Internet designers assumed users would be congestion sensitive—that is, they would automatically cut down their sending rates when the network became congested. Thus, the Internet takes a passive role in providing quality of service (QoS) under heavy user loads. As the Internet grows, however, the variety of users increases, and this assumption becomes invalid. Consequently, the Internet can no longer guarantee high-quality service to all users. If routers could more actively participate in bandwidth distribution, the Internet would be more robust and could accommodate more diverse users.[1]

Two general categories of fair-bandwidth-allocation mechanisms, each with its own drawbacks, exist.[2] The first category, which includes fair queuing (FQ)[3] and its many variants,[4,5] uses complex packet-scheduling algorithms that are more difficult to implement than first-in first-out (FIFO) queuing. Algorithms in the second category—active queue management schemes with enhancements for fairness, such as flow random early detection (FRED),[6] and stochastic fair blue (SFB)[7]—are based on FIFO queuing. They are easy to implement and are much fairer than the original random early detection (RED) design,[8] but they don't aim to provide max-min fairness among numerous flows.

In other work, we propose approximate fair dropping (AFD),[9] a router mechanism that achieves approximately max-min fair bandwidth allocations with relatively low complexity. In this article, we propose an AFD implementation that can mimic the original design's performance while retaining much less state.

Approximate fair dropping

Like RED, AFD is an active queue management scheme that uses a FIFO queue and drops packets probabilistically as they arrive. AFD, however, bases flow-dropping decisions not only on queue size but also on its estimate of the flow's (say flow *i*) current sending rate r_i. To achieve max-min fairness, we define the dropping function d_i as $(1 - r_{\text{fair}} r_i^{-1})_+$. As a result, fair share $r_i(1 - d_i) = \min(r_i, r_{\text{fair}})$ bounds each flow's throughput. Hence, AFD does not distribute drops evenly across flows but applies them differentially to flows with different rates. What sets AFD apart from other queue management algorithms is its simple and systematic approach to estimating r_i and r_{fair}.

To estimate r_i, AFD recognizes that, like flow size distribution, flow rate distribution is long-tailed—that is, fast flows send most bytes, and most flows are slow. For example, Figure 1 shows the cumulative distributions of the 1-second flow rates for three different traces. In these data sets, 10 percent of the flows represent between 60 and 90 percent of the total bytes. Therefore, a sample of recent traffic consists mainly of bytes from faster flows and, typically, these flows send at or above the fair share. Most slow flows won't show up in the sample and AFD can ignore them because it won't drop them. Thus, AFD needs only keep state proportional to the number of fast flows, which is much less than per-flow state. To do this, AFD maintains a shadow buffer of b arrival packet samples (headers only).

Suppose flow i has m_i packets in the shadow buffer. AFD can approximate i's arrival rate by $m_i = br_iR^{-1}$, where R is the aggregate arrival rate. Clearly, we can rewrite the drop function as $d_i = 1 - (m_{\text{fair}}m_i^{-1})$, where $m_{\text{fair}} = br_{\text{fair}}R^{-1}$.

AFD obtains m_{fair} implicitly. Varying m_{fair} intentionally causes $\Sigma_i r_i(1 - d_i)$ to change accordingly, which makes the queue length fluctuate. It will stabilize when $\Sigma_i r_i(1 - d_i)$ equals the outgoing link capacity, at which point $m_{\text{fair}} = br_{\text{fair}}R^{-1}$. To ensure the queue length stabilizes near a target value, AFD updates m_{fair}, using the equation

$$m_{\text{fair}}(t) = m_{\text{fair}}(t-1) + \alpha(q(t-1) - q_{\text{target}}) - \beta(q(t) - q_{\text{target}})$$

where $q(t)$ is the queue length at the tth sample, $q(t-1)$ is the queue length at the previous sample, and q_{target} is the target queue size. Constants α and β are configurable parameters. We discuss in detail how we set these parameters elsewhere.[9]

Using this method, we can infer m_{fair} dynamically with no additional state. Analysis indicates that AFD's memory requirements are a small fraction of those needed for the packet buffers.[9]

We have evaluated AFD's performance in a variety of simulations. One simulation setup consists of seven transmission control protocol (TCP) flow groups (five flows each) with different congestion control mechanisms and round-trip times (RTTs). The congested-link bandwidth is 10 Mbps, thus R_{fair} equals 286 Kbps. Figure 2 compares AFD's performance to that of RED and FRED. Figure 2a shows the average throughput received by each flow group, and Figure 2b depicts the corresponding drop probability of each flow group. These results demonstrate that AFD provides a good approximation to fair bandwidth allocation by differentially dropping packets.

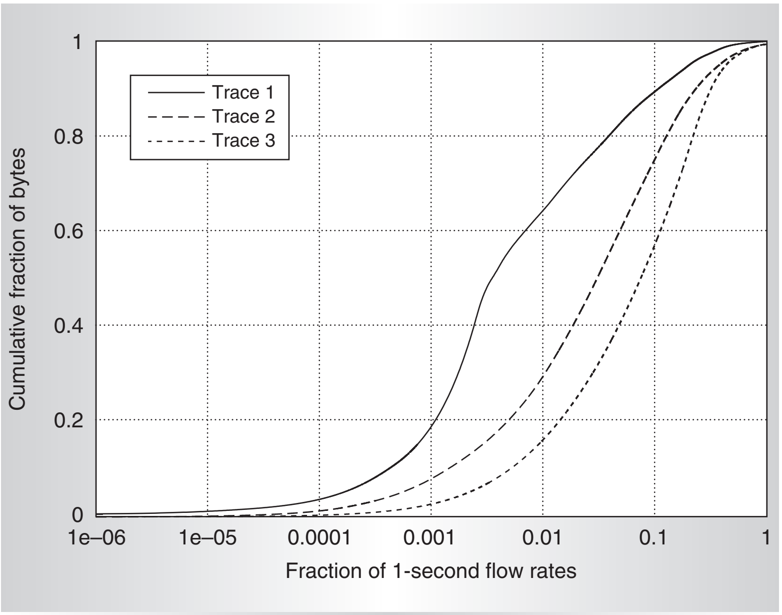

Figure 1. Complementary distribution of 1-second flow rates for three traces. Most bytes in the sample are from fast flows, with 10 percent of the flows representing between 60 and 90 percent of the total bytes.

Implementing AFD

Although AFD theoretically requires only one data structure—the shadow buffer—to function, it is infeasible to recount m_i on each packet arrival. Hence, a direct implementation of the AFD algorithm, which we refer to as the AFD-SB design, requires two data structures:

- a shadow buffer that stores a recent sample of packet arrivals and
- a flow table that keeps the packet count of each flow in the shadow buffer.

We can implement the flow table structure using a hash table or a content-addressable memory with $O(1)$ lookup time. AFD updates the shadow buffer probabilistically. When a packet arrives with probability p (p^{-1} is the update interval), AFD replaces a random packet in the shadow buffer with the arriving packet. Although we could remove packets using FIFO, random replacement avoids synchronization problems. After a replacement, the

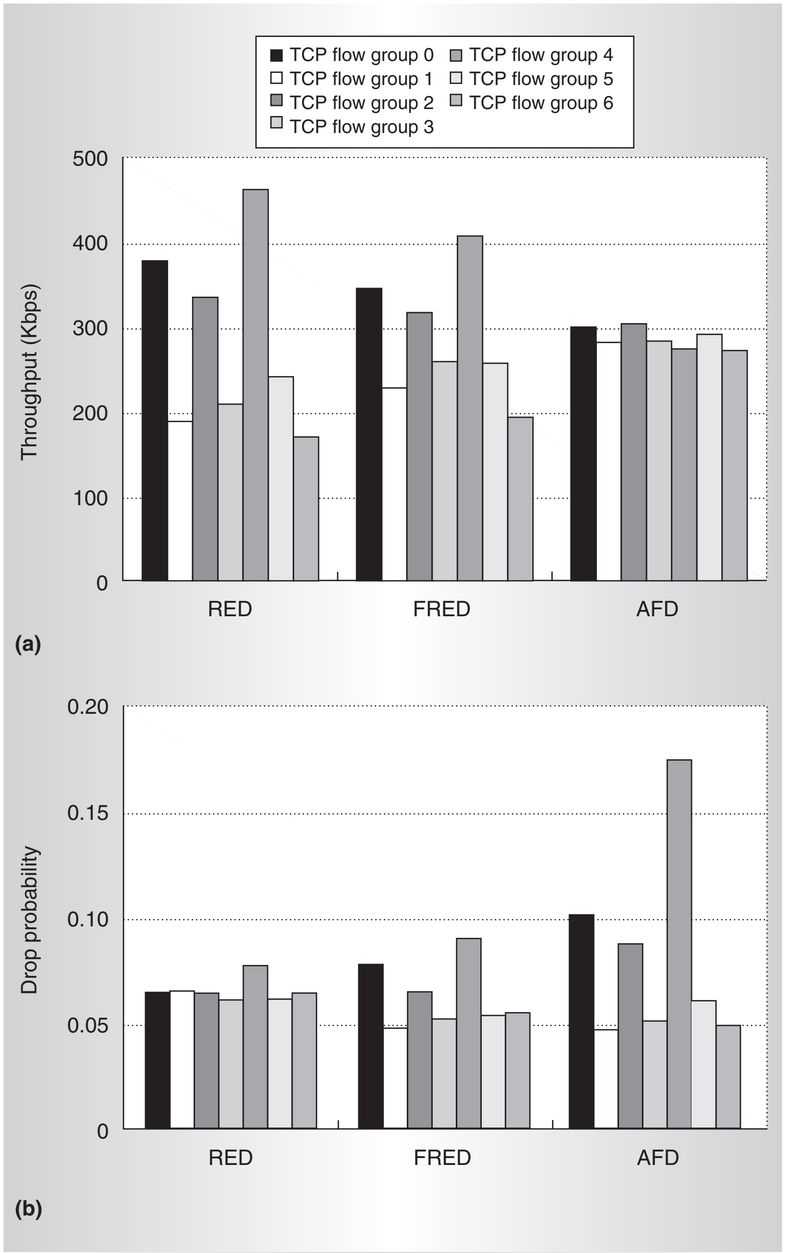

Figure 2. Performance of RED, FRED, and AFD for average throughput received by seven simulated TCP flow groups (a) and the corresponding drop probability for each flow group (b).

packet count for the flow to which the victim packet belongs (say flow *i*) decreases by one, $m_i = m_i - 1$. Conversely, the packet count for the flow to which the incoming packet belongs (say flow *j*) increases by one, $m_j = m_j + 1$. Assume the shadow buffer contains *b* packets from *N* flows, then $\Sigma_{i=1}^{N} m = b$.

Figure 3 shows a simple example of this packet replacement process. A shadow buffer of size 12 holds packets from three flows. These flows have two, six, and four packets in the shadow buffer. When a flow 3 packet arrives, AFD randomly chooses a flow 2 packet for the newly arrived packet to replace. Flow 2's packet count in the flow table thus decreases by one while flow 3's packet count increases by one. These operations maintain the data structures (the shadow buffer and the flow table) used to guide dropping decisions, which are separate from the FIFO buffers in which actual packets are queued.

Single data structure

A randomized approximation of AFD that keeps only one data structure, the flow table, can reduce AFD-SB's memory requirement. The shadow buffer is only logically present in the sense that

$$\sum_{i=1}^{N} m_i = b$$

still holds. AFD increments the flow table on packet insertions as before. The challenge is removing a packet from the logical shadow buffer—that is, decreasing a flow's packet count by one—without linearly traversing the flow entries. Ideally, a new algorithm would mimic AFD-SB's performance: It would remove a flow *i*'s packet with a probability $p_i = m_i b^{-1}$. Therefore, on average, new packets would replace all flow *i*'s packets, m_i, after *b* updates.

The initial AFD-FT (flow table) design (to be consistent with our other work,[9] we refer to this design as AFD-FT) works as follows: When it is time to update the logical shadow buffer, AFD-FT uniformly chooses a small set of flow IDs, *S*. If *s* is the size of *S*, each flow has equal probability sN^{-1} of being in the set. Given that flow *i* is in *S*, it has a probability of $m_i(\Sigma_{j\in S} m_j)^{-1}$ to have its count decreased by one. AFD-FT tries to approximate $p_i = m_i b^{-1}$ under AFD-SB with $p_i = sN^{-1}\ m_i(\Sigma_{j\in S} m_j)^{-1}$. AFD-FT can approximate AFD-SB's performance when there are no large flows whose packet counts are much larger than those of other flows. If such flows exist, however, AFD-FT tends to limit their throughput under the fair share. This gives flows an equal chance of being present in *S*, even though a flow ($\in$ *S*) with more packets has a higher probability of being reduced. Therefore, flow *i* with a higher packet count has a lower than $m_i b^{-1}$ chance of being reduced at each update. Consequently, on average, its total count deduction is less than m_i after b updates, leading to a higher drop probability.

Using the example in Figure 3, Figure 4 illustrates how AFD-FT behaves when s equals one. By choosing one flow at random, flows 1, 2, and 3 each have a one-third chance of being reduced by one. Under AFD-SB, however, the chances for these three flows are one-sixth, one-half, and one-third. Thus, while AFD-SB needs six updates on average to reduce flow 1's count by one, AFD-FT needs only three updates. AFD-FT favors small flows and is biased against fast flows. As our later simulations show, this bias against larger flows can lead to a significant throughput penalty.

New flow table design

To improve AFD-FT's performance, we propose a new AFD flow table design, which we refer to as AFD-NFT (new flow table). AFD-NFT achieves the performance of AFD-SB with AFD-FT's state requirement.

When it is time to decrease a flow's packet count by one (that is, remove a packet from the logical shadow buffer), AFD-NFT draws a small set S of flow IDs uniformly from the flow entries, if such a set does not already exist. A flow $i(\in S)$'s packet count decreases by one with a probability of $m_i(\Sigma_{j\in S}\ m_j)^{-1}$. These operations are exactly the same for both AFD-FT and AFD-NFT. The next step, however, represents the crucial difference between the two: AFD-FT chooses new set S for each update. AFD-NFT, on the other hand, uses the same set S for the next $u = a \times (\Sigma_{j\in S}\ m_j)$ updates, where the constant $a < 1$. After u updates, AFD-NFT chooses a new set and repeats the same operations.

Figure 4 shows how AFD-NFT would work if $a = 0.5$ and $s = 1$. Each flow has a one-third chance of being drawn. When AFD-NFT selects flow 1, m_1 decreases to one. Because $u = 1$, the algorithm will draw a new flow for the next table update. Suppose AFD-NFT chooses flow 2 instead, with $u = 3$. Flow 2 will be the victim flow for the following two table updates before the algorithm selects a new flow. Similarly, if flow 3 is drawn, AFD-NFT will use it for the next update.

Analysis

Recall that our goal for AFD-NFT is to match the performance of AFD-SB, replac-

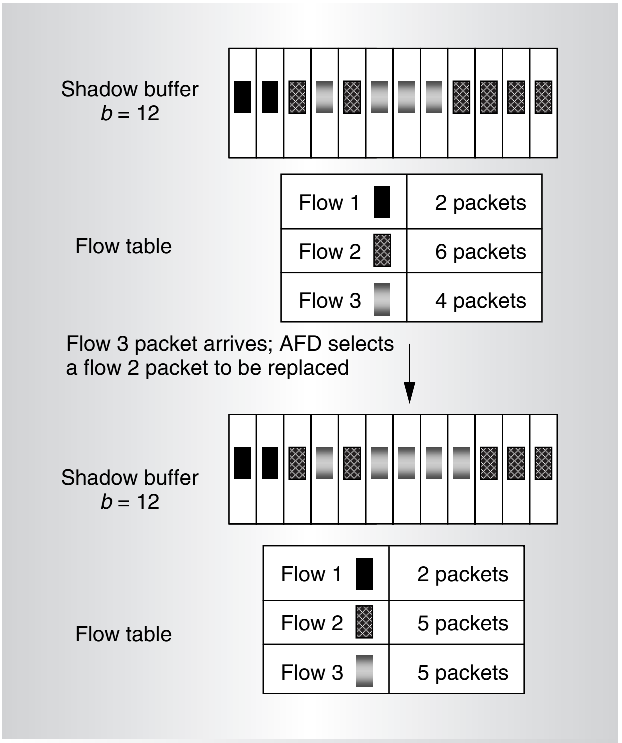

Figure 3. Packet-dropping decisions in AFD-SB. When a packet arrives, AFD randomly selects a packet in the shadow buffer for replacement.

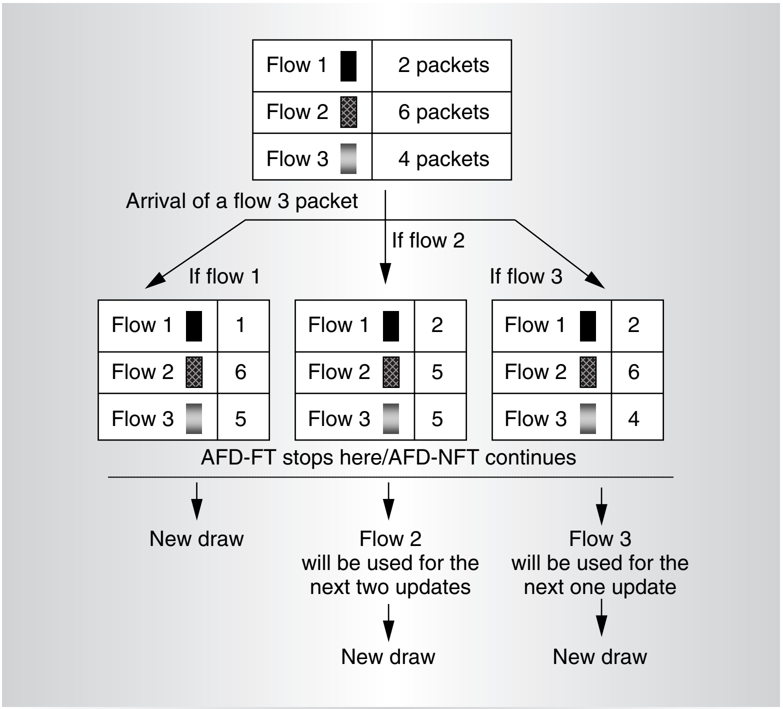

Figure 4. AFD-FT and AFD-NFT designs. When a new packet arrives at the shadow buffer, the algorithms update the flow tables.

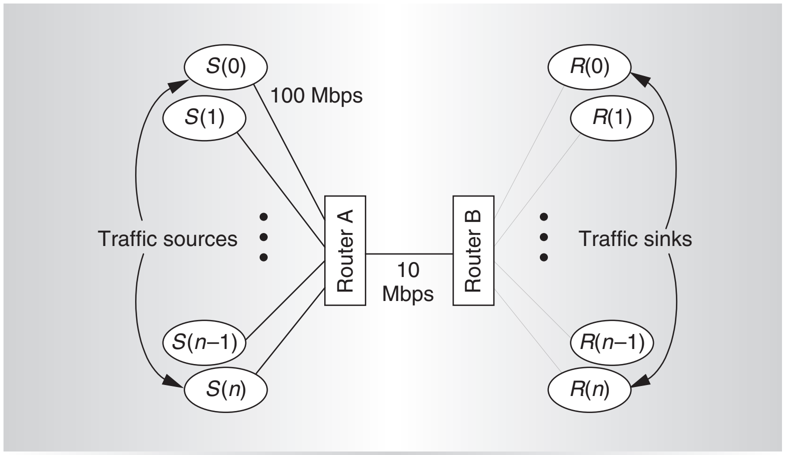

Figure 5. Basic simulation topology used in our evaluations.

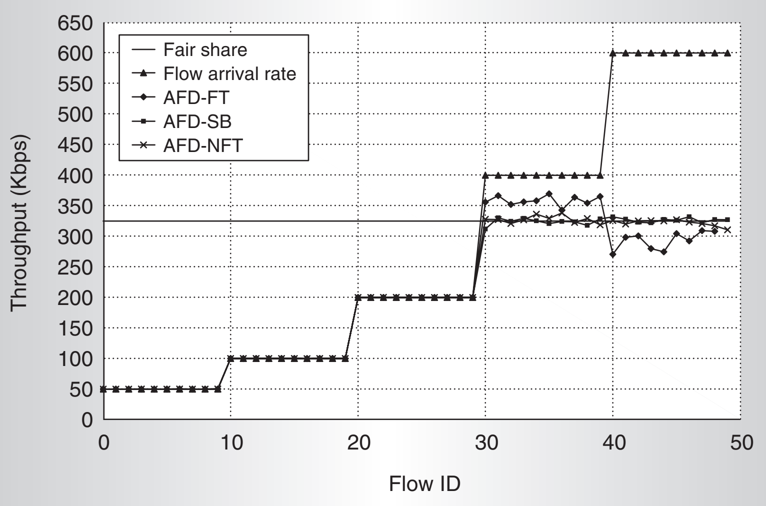

Figure 6. Offered load and throughput for 50 CBR flows using the three AFD designs.

ing approximately m_i of flow i's packets after b updates. By the law of large numbers, we can prove that AFD-NFT's performance, on average, is the same as AFD-SB's. We outline the proof as follows:

- Each flow has the same chance of being chosen for set S, and the probability, p_s, is sN^{-1}.
- A flow's average packet count equals
$$\frac{\sum_{i=1}^{i=N} m_i}{N} = \frac{b}{N}.$$
Assuming $s << N$, the average total packet count in chosen set S is bsN^{-1}. Thus, the total packets to be replaced are $absN^{-1}$. To replace b packets, we need to draw
$$N_s = \frac{b}{absN^{-1}} = \frac{N}{as}$$
sets.
- Given a set S and flow $i(\in S)$, there are on average
$$NPS_i = \frac{m_i}{\sum_{j\in S} m_j} \times u =$$
$$\frac{m_i}{\sum_{j\in S} m_j} a \times \sum_{j\in S} m_j = am_i$$
flow i packets to be replaced.
- When we combine the above three arguments, after b updates, the average number of flow i packets replaced equals
$$NP_i = \frac{s}{N} \times \frac{N}{as} \times am_i = m_i,$$
which matches AFD-SB's behavior.

Simulation results

We evaluate AFD-NFT's performance in several scenarios and compare it to AFD-SB and AFD-FT. Figure 5 depicts our simulation topology. Unless otherwise stated, the latencies at the access links are 2 ms, and the latency at the congested link is 20 ms. In all the experiments, $b = 1{,}000$, $a = 0.06$, and $s = 5$.

Performance improvement

Figure 6 compares the performance of AFD-SB, AFD-FT, and AFD-NFT for a simulation run in which five constant bit rate (CBR) flow groups of 10 flows each compete for the congested link bandwidth of 10 Mbps. The sending rates for the groups are 50 Kbps, 100 Kbps, 200 Kbps, 400 Kbps, and 600 Kbps. Results show that AFD-NFT can mimic AFD-SB's performance by providing each flow its fair share. AFD-FT penalizes the aggressive flows by limiting their throughput to be under their fair share.

Although the performance penalty is mild in this scenario, AFD-FT can severely punish aggressive flows. Table 1 lists the simulation's flow table access statistics for p_S, the probabil-

Table 1. Flow table access results from both theoretical analysis and actual simulation data.

	p_s		NPS_i (average no. of packets)		NP_i (average no. of packets)		N_s (no. of sets)	
Flow group ID	sN^{-1}	**Simulation data**	am_i	**Simulation data**	m_i	**Simulation data**	$N(as)^{-1}$	**Simulation data**
0	0.1	0.098	0.222	0.207	3.70	3.63		
1	0.1	0.100	0.444	0.411	7.41	7.42		
2	0.1	0.100	0.888	0.808	14.81	14.82	180	167
3	0.1	0.100	1.778	1.646	29.63	29.66		
4	0.1	0.100	2.667	2.453	44.44	44.53		

ity of a flow being in a set; NPS_i, the average number of flow i's packets replaced; NP_i, the average number of flow i's packets replaced after b updates; and N_s, the average number of sets drawn after b updates. Because the variance between individual flows is very small, as Figure 6 shows, we average the statistical data within the 10 flows in each group to more easily present them. Clearly, the data obtained from the simulation closely agrees with predictions from our analysis.

We next evaluate the performance of AFD designs in the presence of an on-off source. In this setup, an on-off source shares the congested link with 35 TCP flows, where R_{fair} equals 278 Kbps. The bursty source sends at the access link speed (100 Mbps) for a very short period, t_{on}, and then is idle for time t_{off}. Its average sending rate is 100 Mbps $\times\ t_{\text{on}}(t_{\text{on}} + t_{\text{off}})^{-1}$. We plot only the bursty source throughput in Figure 7 because it shows the largest discrepancies among the three AFD algorithms. The TCP flows use the remaining link bandwidth; differences among those flows are small. In the figure, the leftmost bars in each grouping represent the on-off source throughput when its average sending rate is only half of R_{fair}, in which case all three algorithms allocate the bandwidth fairly—that is, they supply the flow its request bandwidth. As the plot shows, however, as the on-off flow becomes burstier and sends above $2R_{\text{fair}}$, AFD-FT starts penalizing it. The burstier the flow, the more severe the penalty. Conversely, AFD-SB and AFD-NFT do not penalize flows for burstiness.

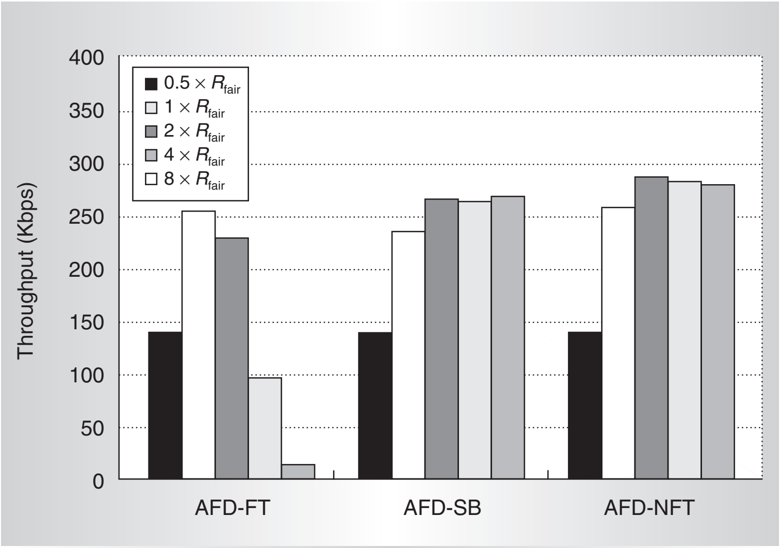

Figure 7. Performance of the three AFD algorithms in the presence of a bursty on-off source. As a flow becomes burstier, AFD-FT begins to penalize it, while AFD-SB and AFD-NFT do not.

Figure 8 (next page) represents a simulation in which the traffic mix is one user datagram protocol (UDP) source sharing the link with seven groups (five flows per group) of TCP flows with different congestion control methods. For generalized window control mechanisms, the window increase has a form of $w + c_1 w^{-k}$, and the decrease of a form $w - c_2 w^l$,[10] where k, l, and w are parameters defined in the TCP algorithm. The seven groups in the simulation have different values of c_1, c_2, k, l, and RTTs, as tabulated in Table 2. The normal TCP flow has the form $c_1 = 1.0$, $c_2 = 0.5$, $k = 1.0$, and $l = 1.0$). In Figure 8a, the rightmost bars in each grouping represent the throughput of the more aggressive UDP flow under the different algorithms. Results show once again that the AFD-NFT design can mimic the performance of AFD-SB while AFD-FT fails to do so.

Comparable performance

Removing the UDP flow from the simulation leaves only TCP flows with various congestion parameters to compete against each other. As Figure 8b shows, AFD-NFT performs as well as AFD-FT, and all three AFD designs allocate bandwidth fairly.

AFD behaves reasonably well when flows with different RTTs share a link.[9] To ensure that AFD-NFT's performance is not worse, we construct another experiment. In this simulation, we separate flows into four groups with 10 flows in each group. The RTTs (propagation delay only) are 37.5 ms, 75 ms, 112.5 ms, and 150 ms. Figure 9 shows that AFD-NFT's performance is similar to that of AFD-SB and AFD-FT. Although some discrepancies among flows with different RTTs exist, they are not significant.

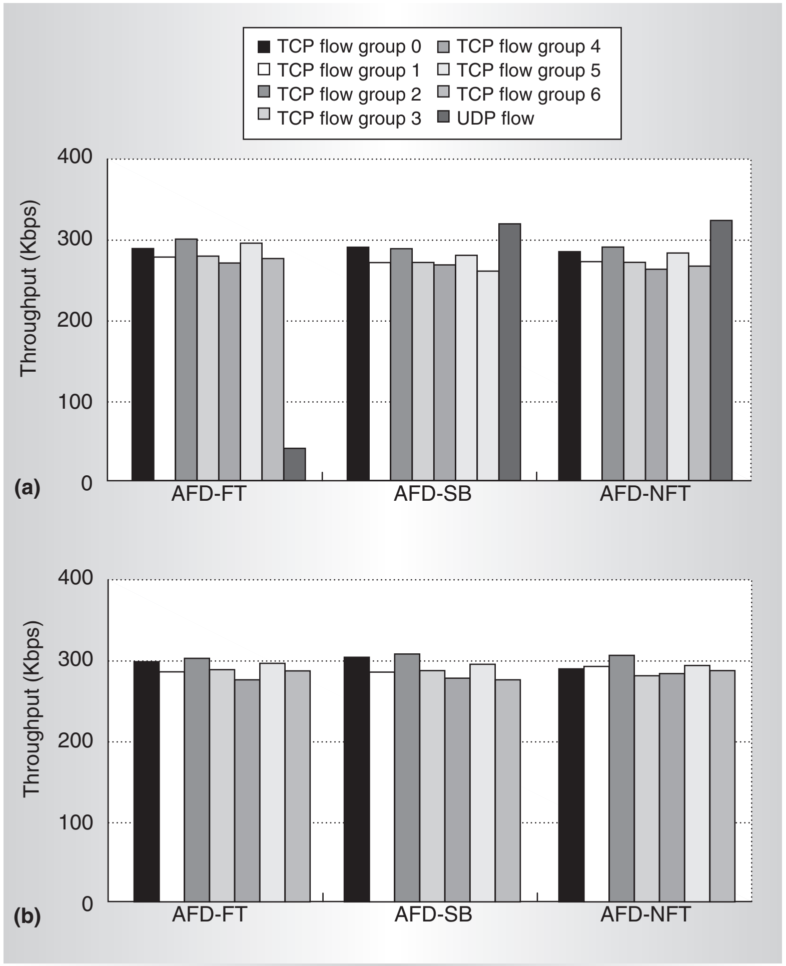

Figure 8. Mixed TCP traffic with and without a UDP flow. Under AFD-FT, the UDP traffic is not treated fairly (a). When TCP flows compete only against each other, however, all three AFD algorithms allocate bandwidth fairly (b).

Table 2. Mixed TCP traffic configuration.

Flow group ID	c_1	c_2	K	L	RTT (ms)
0	1.00	0.90	1.00	1.00	25
1	0.75	0.31	1.00	1.00	25
2	2.00	0.50	1.00	1.00	25
3	1.50	1.00	2.00	0.00	25
4	1.00	0.50	0.00	1.00	25
5	1.00	0.50	1.00	1.00	25
6	1.00	0.50	1.00	1.00	100

Memory requirement

As we have shown, AFD-NFT provides reasonably fair bandwidth allocation, and all operations on the forwarding path are $O(1)$. Thus, the main question regarding whether AFD-NFT is practical lies in its memory requirement. Because set S is small (usually fewer than 10 flows), registers can easily store the set's flow IDs. The flow table, however, requires some memory buffering.

The flow table size relates directly to number of flows N in the shadow buffer. In the traces we have seen,[9] N is typically less than one-fourth of b, the number of packets in the logical shadow buffer. We also find that to achieve good performance, b should be roughly $10r_{\text{fair}}^{-1}R$. Hence, N equals $2.5r_{\text{fair}}^{-1}R$. It is difficult to estimate r_{fair} on a typical Internet link. A conservative estimate assumes r_{fair} equals 56 Kbps, the slow telephone modem speed. Then, for a 1-Gbps link, it is simple to obtain that N is on the order of a few thousand. Therefore, we can easily implement the flow table using a standard hash table or CAM. The memory overhead is limited.

Fair bandwidth allocation designs differ in the extent to which they carefully manage bandwidth allocation. The extremes of the spectrum are complete fairness (FQ) on one end and unmanaged allocation (RED) on the other. We believe that Internet flows will use a wide variety of congestion-control algorithms, which might not be TCP compatible. Routers will thus frequently need to allocate bandwidth to flows. We designed AFD for such a scenario. AFD approximates FQ's performance, but drastically reduces its state requirement. This and other data suggest that AFD-NFT will provide a good approximation to fair bandwidth allocation in a wide range of scenarios, typically providing bandwidth allocations within ±15 percent of the fair share. MICRO

References

1. S. Floyd and K. Fall, "Promoting the Use of End-to-End Congestion Control in the Internet," *IEEE/ACM Trans. Networking*, no. 4, Aug. 1999, pp. 458-472.
2. B. Braden et al., "Recommendations on Queue Management and Congestion Avoidance in the Internet," Internet Eng. Task Force RFC 2309 (informational), Apr. 1998; www.ietf.org/rfc/rfc2309.txt.
3. A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queuing Algorithm," *J. Internetworking Research and Experience*, Oct. 1990, pp. 3-26.
4. P. McKenny, "Stochastic Fairness Queuing," *Proc. Infocom 1990*, IEEE Press, 1990, pp. 733-740.
5. I. Stoica, S. Shenker, and H. Zhang, "Core-Stateless Fair Queueing: Achieving Approximately Fair Bandwidth Allocations in High-Speed Networks," *Proc. ACM SIGComm 1998,* ACM Press, 1998, pp. 118-130.
6. D. Lin and R. Morris, "Dynamics of Random Early Detection," *Proc. ACM SIGComm 1997*, ACM Press, 1997, pp. 127-137.
7. W. Feng et al., "Stochastic Fair Blue: A Queue Management Algorithm for Enforcing Fairness," *Proc. Infocom 2001*, IEEE Press, 2001, pp. 1520-1529.
8. S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," *IEEE/ACM Trans. Networking*, vol. 1, no. 4, Aug. 1993, pp. 397-413.
9. R. Pan et al., "Approximate Fairness through Differential Dropping," to appear in *Computer Comm. Rev.*, vol. 33, no. 1, Jan. 2003.
10. D. Bansal and H. Balakrishnan, "Binomial Congestion Control Algorithms," *Proc. Infocom* 2001, IEEE Press, 2001, pp. 631-640.

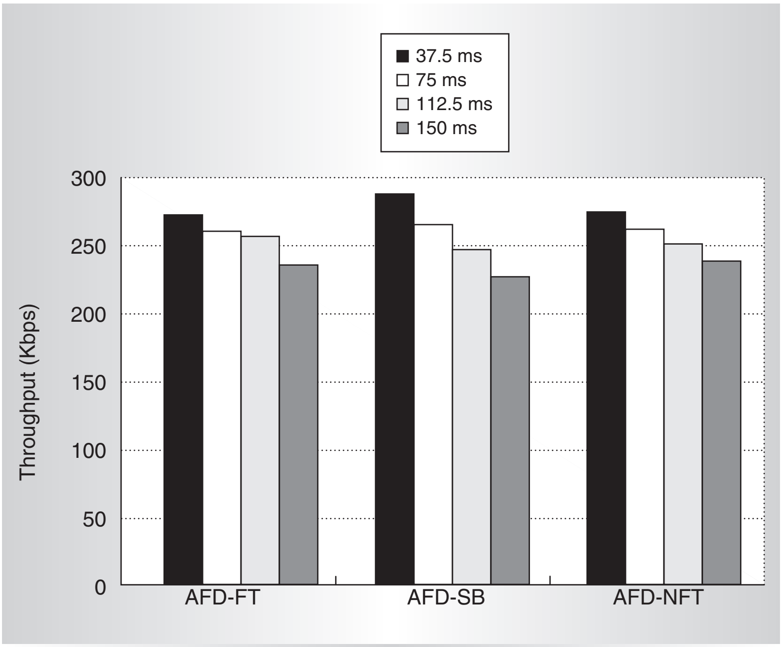

Figure 9. Four TCP flow groups with different RTTs (maximum is 150 ms). All three algorithms perform similarly.

Rong Pan is a research scientist at Stanford University. Her research interests include congestion control, active queue management, TCP performance, and efficient network simulation. Pan has a PhD in electrical engineering from Stanford University.

Balaji Prabhakar is an assistant professor of electrical engineering and computer science at Stanford University. His research interests include network algorithms, wireless networks, Web caching, network pricing, information theory, and stochastic network theory. Prabhakar has a PhD from the University of California at Los Angeles.

Lee Breslau heads the Internetworking Research Department AT&T Labs—Research in Menlo Park, California. His research interests include packet scheduling, real-time service, network measurement, and routing. Breslau has a PhD from the University of Southern California.

Scott Shenker is group leader at the International Computer Science Institute's Center for Internet Research. His research interests range from computer performance modeling and computer networks to game theory and economics. Shenker has a PhD from the University of Chicago.

Direct questions and comments about this article to Rong Pan, Stanford University, Packard Rm. 270, 350 Serra Mall, Stanford, CA 94305-9510; rong@stanford.edu.