A Flow Table-Based Design to Approximate Fairness

Rong Pan, Stanford University, rong@stanford.edu
Lee Breslau, AT&T Labs-Research, breslau@research.att.com
Balaji Prabhakar, Stanford University, balaji@stanford.edu
Scott Shenker, ICIR, shenker@icir.org

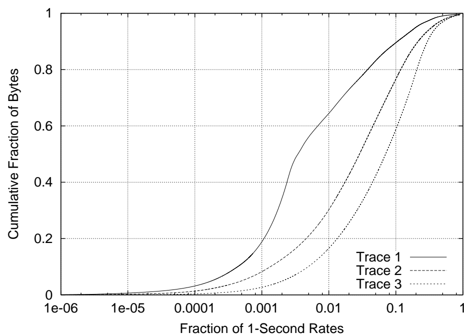

Fig. 1. Complementary Distribution of 1-Second Rates

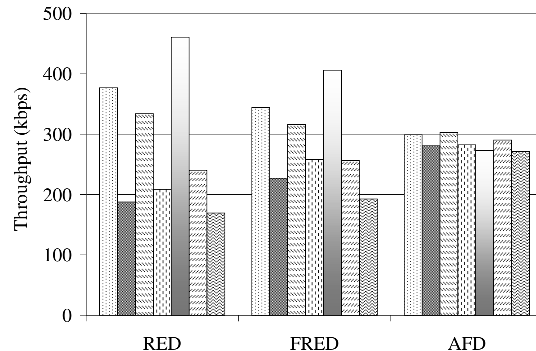

Fig. 2. Background: Mixed Traffic - throughput

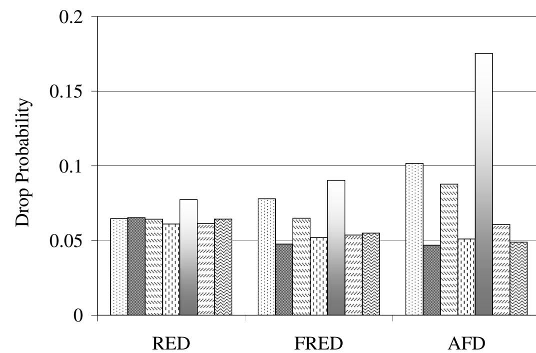

Fig. 3. Background: Mixed Traffic - drop probability

Abstract—**The current Internet architecture relies on congestion avoidance mechanisms implemented in the transport layer protocols, like TCP, to provide good service under heavy load. If routers distribute bandwidth fairly, the Internet would be more robust and could accommodate more diversity of end hosts. Most of the mechanisms proposed to accomplish this can be grouped into two general categories. The first category, which includes Fair Queueing (FQ [4]) and its many variants, uses packet scheduling algorithms that are more difficult to implement compared to FIFO queueing. The algorithms in the second category, active queue management schemes with enhancements for fairness (*e.g.*, FRED [8], SFB [5]), are based on FIFO queueing. They are easy to implement and are much fairer than the original RED [6] design, but they don't provide max-min fairness among a large population of flows. Recently, a router mechanism, AFD [14] (Approximate Fair Dropping), has been proposed to achieve approximately max-min fair bandwidth allocations with relatively low complexity. In this paper, we propose an implementation of AFD which can mimic the performance of the original design with much less state.**

I. Background

Approximate Fair Dropping (AFD) [14] is an active queue management scheme which uses a FIFO queue and drops packets probabilistically upon arrival. However, the decision whether to drop a packet (say from flow i) or not is based not only on the queue size but also on an estimate of the flow's current sending rate r_i. To achieve max-min fairness, the dropping function is defined to be $d_i = (1 - \frac{r_{fair}}{r_i})_+$. As a result, the throughput of each flow is bounded by the fair share: $r_i(1-d_i) = min(r_i, r_{fair})$. Hence, drops do not occur evenly across flows but are applied differentially to flows with different rates. The key design aspects of AFD lie in the methods by which r_i and r_{fair} are estimated.

To estimate r_i, AFD uses the observation that, like the distribution of flow sizes, the distribution of flow rates is also long-tailed, i.e. most bytes are sent by *fast* flows and a vast majority of 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% of the flows represent 60% - 90% of the total bytes transmitted. Therefore, a sample of the recent traffic would largely consist of bytes from faster flows and, typically these are the flows that send at rates equal to or above the fair share. Most slow flows won't show up in the sample and can be ignored anyway since they won't be dropped. AFD takes advantage of this property of traffic samples to estimate flow rates. Therefore, it only needs to keep state proportional to the number of fast flows, which is much less than per-flow state. Specifically, AFD maintains a shadow buffer of b arrival packet samples (header only). Suppose currently a flow i has m_i packets in the shadow buffer. Then its arrival rate can be approximated by $r_i = R\frac{m_i}{b}$, where R is the aggregate arrival rate. It is clear that the drop function can be rewritten as

$$d_i = (1 - \frac{m_{fair}}{m_i})_+ \quad (1)$$

where $m_{fair} = b\frac{r_{fair}}{R}$.

In AFD, m_{fair} is obtained implicitly. Note that if m_{fair} is varied intentionally, $\sum_i r_i(1-d_i)$ would change accordingly. As a result, the queue length fluctuates. It will stabilize when

$\sum_i r_i(1-d_i)$ equals the outgoing link capacity, at which point $m_{fair} = b\frac{r_{fair}}{R}$. To enforce the queue length stabilizing around a target value, AFD updates m_{fair} periodically as follows,

$$m_{fair}(t) = m_{fair}(t-1) + \alpha(q(t-1) - q_{target}) - \beta(q(t) - q_{target}) \quad (2)$$

where $q(t)$ is the queue length at the t-th 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. The detailed discussion regarding how to set these parameters can be found in [14]. Using the above method, we can infer m_{fair} dynamically with no additional state.

The performance of AFD has been evaluated in a variety of scenarios using simulation. One typical simulation result is shown in Figures 2 and 3. The simulation set up consists of 7 TCP flow groups (5 flows each) with different congestion control mechanisms and RTTs.[1] The congested link bandwidth is 10Mbps, therefore R_{fair} equals 286Kbps. The performance of AFD is compared against RED and FRED. Figure 2 shows the average throughput received by each flow group. The corresponding drop probability of each flow group is depicted in Figure 3. The results demonstrate that AFD provides a good approximation to fair bandwidth allocation by differentially dropping packets.

In Section II, we discuss ways to implement AFD and introduce an improved mechanism, AFD-NFT. Analysis in Section III shows that by the law of large numbers AFD-NFT behaves like AFD on average. We evaluate the performance of these schemes and present the results in Section IV. Finally we conclude in Section VI.

II. Implementation

Although theoretically AFD 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 which stores a recent sample of packet arrivals and a flow table which keeps the packet count of each flow that appeared in the shadow buffer. The flow table structure can be implemented using a hash table or a CAM, which has O(1) lookup time. The update of the shadow buffer occurs probabilistically. When a packet arrives, with probability $\frac{1}{s}$ (s is the update interval), a random packet in the shadow buffer is chosen and replaced with the arriving packet. Although we could remove packets in a FIFO way, random replacement avoids synchronization problems. After a replacement, the packet count for the flow which the victim packet belongs to (say flow i) is reduced by one, $m_i = m_i - 1$. Conversely, the packet count for the flow which the incoming packet belongs to (say flow j) is increased by one, $m_j = m_j + 1$. Assume the shadow buffer size is b packets and there are N flows which have packets present in the shadow buffer, then $\sum_{i=1}^{N} m_i = b$.

[1] The topology and the simulation set up is the same as the one to be discussed in subsection IV-B.

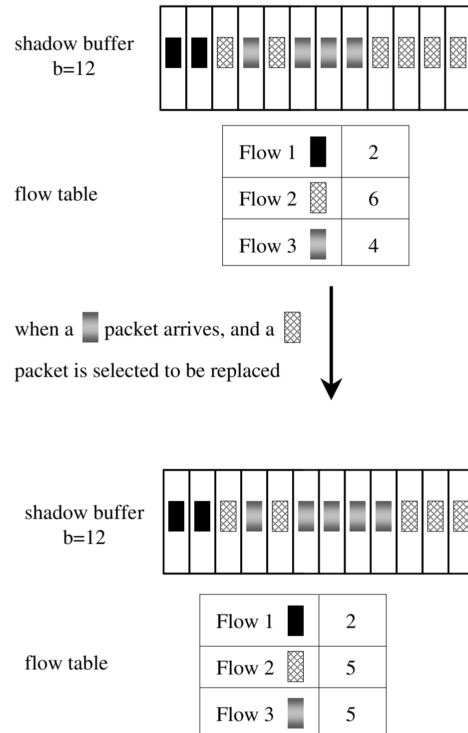

Fig. 4. AFD-SB Design

Figure 4 shows a simple example of the above process. The shadow buffer of size 12 hold packets from three different flows: *Flow 1*, *2* and *3*. These flows have 2, 6 and 4 packets present in the shadow buffer respectively. When a *Flow 3*'s packet arrives, a randomly chosen *Flow 2*'s packet is replaced by this newly arrived packet. As a result, *Flow 2*'s packet count in the flow table is decreased by one while *Flow 3*'s packet count is incremented by one. These operations maintain the data structures (the shadow buffer and the flow table) used to guide dropping decisions. Note that these data structures occupy memory which is separate from the FIFO buffers in which actual packets are queued. In the next two subsections, we discuss ways in which we can simplify the implementation of AFD.

A. Reducing Memory

Aiming to reduce the memory requirement of AFD-SB, we propose a randomized approximation of AFD, which keeps only one data structure, the flow table. The shadow buffer is only logically present in the sense that $\sum_{i=1}^{N} m_i = b$ still holds. Incrementing the flow table upon packet insertions is straightforward and is the same as before. The challenge is how to remove a packet from the logical shadow buffer, i.e. to decrement a flow's packet count by one, without linearly traversing the flow entries. In the ideal case of mimicking AFD-SB's performance, we would like a flow i's packet to be removed with a probability $p_i = m_i b^{-1}$. Therefore *on average*, all m_i of flow i's packets are replaced after b updates.

The initial AFD-FT design[2] works as follows: When it is time

[2] To be consistent with [14], we refer to this design as AFD-FT throughout this

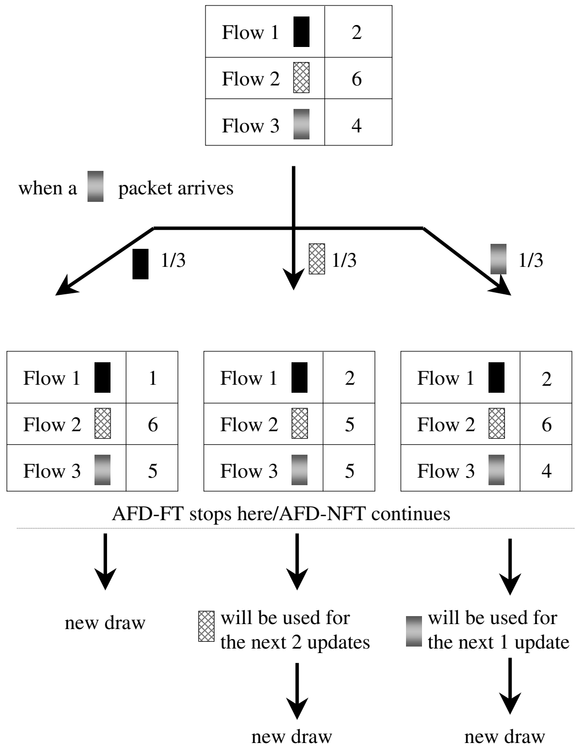

Fig. 5. AFD-FT and AFD-NFT Designs

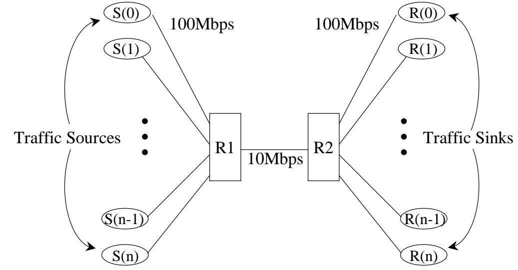

Fig. 6. Basic Simulation Topology

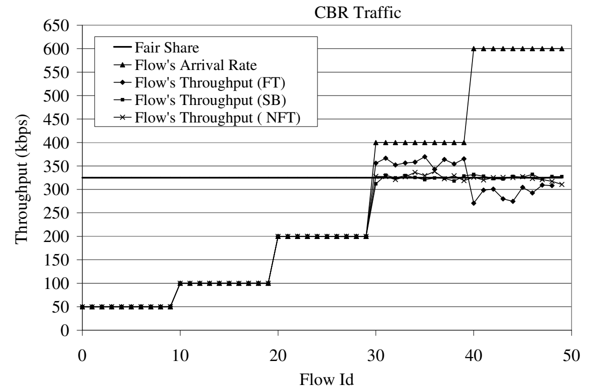

Fig. 7. Offered Load and Throughput for 50 CBR Flows under AFD designs

to update the logical shadow buffer, a small set of flow ids, S, are chosen uniformly. Let s be the size of the set S. Then each flow has an equal probability of sN^{-1} to be present in the set. Given that a flow i is chosen, with a probability $m_i(\sum_{j\epsilon S} m_j)^{-1}$, its count is decremented by one. AFD-FT tries to approximate $p_i = m_i b^{-1}$ under AFD-SB with $p_i = sN^{-1}m_i(\sum_{j\epsilon S} m_j)^{-1}$. AFD-FT can approximate AFD-SB's performance when there are no large flows whose packet counts are much bigger than that of other flows. However if such flows do exist, then AFD-FT tends to penalize them by limiting their throughput to be below the fair share. The reason for this is that all flows have an equal chance of being present in the set S, even though a flow (ϵ S) with higher packet count has a higher probability of being decremented. Therefore, a flow i with a higher packet count has a lower chance than $m_i b^{-1}$ to be decremented for each update. As a result, *on average*, its total count deduction is less than m_i after b updates, leading to a higher drop probability. Using the same example in Figure 4, Figure 5 illustrates how AFD-FT would behave when s equals one. By choosing one flow at random, Flow 1, 2 and 3 have an equal chance of $\frac{1}{3}$ to be decremented by one. Under AFD-SB, however, the chances for these three flows are $\frac{1}{6}$, $\frac{1}{2}$, and $\frac{1}{3}$ respectively. Thus, while it needs 6 updates on average to reduce the Flow 1 count by one in AFD-SB, it takes only 3 updates to do so in AFD-FT. Small flows are favored, and there is a bias against fast flows. As our later simulations show, this bias against larger flows can lead to a very significant throughput penalty.

paper.

B. New Flow Table Design

To improve upon the performance of AFD-FT, 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 the state requirement of AFD-FT, and it works as follows: When it is time to decrement a flow's packet count by one (i.e. removing a packet from the logical shadow buffer), draw a small set of S flow ids uniformly from the flow entries if such a set does not exist. A flow i (ϵ S)'s packet count is reduced by one with a probability of $m_i(\sum_{j\epsilon S} m_j)^{-1}$. Notice that the above operations are exactly the same for both AFD-FT and AFD-NFT. The next step, however, represents the crucial difference between the two. Under AFD-FT, a new set S is chosen for each update. Under AFD-NFT, on the other hand, once a set S is chosen, it is used for the next $m = a * (\sum_{j\epsilon S} m_j)$ updates. The constant a is a parameter < 1. After m updates, a new set is chosen again and the same operations are repeated. Figure 5 shows how AFD-NFT would work with $a = 0.5$ and $s = 1$. Each flow has a chance of $\frac{1}{3}$ of being drawn. When *Flow 1* is selected, m_1 is reduced to one. Since $m = 1$, a new flow will be drawn for the next table update. Suppose *Flow 2* is chosen instead, with $m = 3$, *Flow 2* will be the victim flow for the following two table updates before a new flow is selected. Similarly, if *Flow 3* is drawn, the flow will be used for the next update. In the next two sections, we will demonstrate that AFD-NFT performs as well as AFD-SB.

	P_s		NPS_i		NP_i		N_s	
FlowGrp ID	sN^{-1}	statistics	am_i	statistics	m_i	statistics	$N(as)^{-1}$	statistics
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		

TABLE I
FLOW TABLE ACCESS STATISTICS

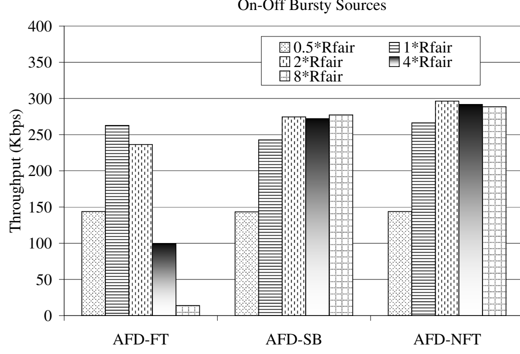

Fig. 8. Bursty On-Off Source

III. ANALYSIS

Recall that our goal in designing AFD-NFT is to match the performance of AFD-SB so that, after b updates, a flow i has on average m_i of its packets replaced. By the law of large numbers, we can prove that the performance of AFD-NFT is the same as that of AFD-SB on average. An outline of the proof follows:

1) We know from the above that each flow has the same chance of being chosen in the set S, and the probability, P_s, is sN^{-1}.

2) The average packet count of a flow equals

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

Therefore, assume $s << N$, the average total packet counts in a chosen set S is sbN^{-1}. As a result, the total of packets that are replaced equals $sbaN^{-1}$. So, to replace b packets, we need to draw,

$$N_s = \frac{b}{sbaN^{-1}} = \frac{N}{as}$$

number of sets.

3) Given a set S and a flow i (ϵ S), there are *on average*

$$\mathrm{NPS_i} = \frac{\mathrm{m_i}}{\sum_{\mathrm{j}\epsilon \mathrm{S}} \mathrm{m_j}} * \mathrm{m} = \frac{\mathrm{m_i}}{\sum_{\mathrm{j}\epsilon \mathrm{S}} \mathrm{m_j}} \mathrm{a} * \sum_{\mathrm{j}\epsilon \mathrm{S}} \mathrm{m_j} = \mathrm{am_i}$$

number of flow i packets to be replaced.

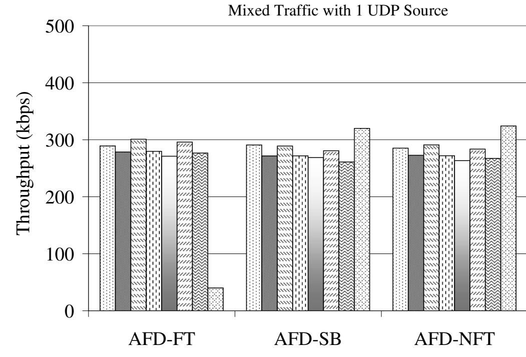

Fig. 9. Mixed TCP Traffic with a UDP flow

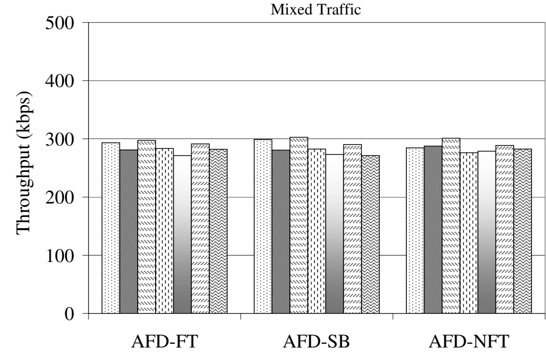

Fig. 10. Mixed TCP Traffic

4) Combine the above three arguments, after b updates, the average number of flow i packets replaced is equal to

$$NP_i = \frac{s}{N} * \frac{N}{as} * am_i = m_i.$$

which matches the desired behavior of matching AFD-SB.

IV. SIMULATION RESULTS

We evaluate the performance of AFD-NFT in a variety of scenarios and compare it against AFD-SB and AFD-FT. Our simulation topology is depicted in Figure 6. Unless otherwise stated, the latencies at the access links are 2ms and the latency at the congested link is 20ms. In all the experiments, b is chosen to be 1000, a is set to 0.06 and s equals 5. We present five simulation results in this section, which are separated into two subsections: in subsection IV-A, we demonstrate that AFD-NFT can perform as well as AFD-SB in the cases where AFD-FT behaves poorly; second, we show that all three algorithms perform similarly in other cases.

A. Performance Improvement

CBR Traffic: Figure 7 shows a simulation run in which five CBR flow groups (10 flows each) compete for the congested link bandwidth of 10Mbps. The sending rates for each group are 50Kbps, 100Kbps, 200Kbps, 400Kbps and 600Kbps. The performance comparison among different AFD designs is presented in Figure 7. The result shows 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

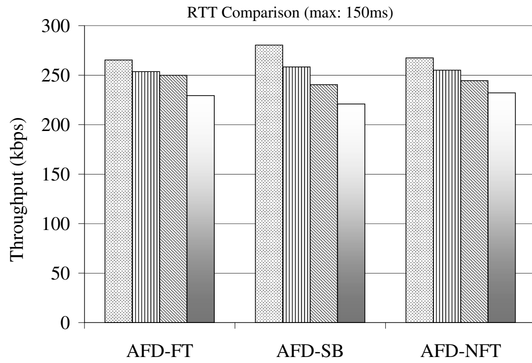

Fig. 11. Four TCP Flow Groups with Different RTTs (max = 150ms)

FlowGrp ID	a	b	k	l	RTT
0	1.0	0.9	1.0	1.0	25ms
1	0.75	0.31	1.0	1.0	25ms
2	2.0	0.5	1.0	1.0	25ms
3	1.5	1.0	2.0	0.0	25ms
4	1.0	0.5	0.0	1.0	25ms
5	1.0	0.5	1.0	1.0	25ms
6	1.0	0.5	1.0	1.0	100ms

TABLE II

MIXED TCP TRAFFIC CONFIGURATION

under their fair share. Although the performance penalty is mild in this scenario, we will show below that AFD-FT can severely punish aggressive flows. The flow table access statistics in this simulation are collected and tabulated in Table I. Since the variance among individual flows is very small, as seen in Figure 7, the statistical data is averaged within the ten flows in each group so that it can be more easily presented. It is clear that the data obtained from the simulation is in very close agreement with what the analysis predicts.

Bursty On-Off Sources: We next evaluate the performance of AFD designs in the presence of an on-off source. In this setup, an on-off source is sharing the congested link with 35 TCP flows, where R_{fair} equals 278Kbps. The bursty source sends at the speed of the access link (100Mbps) for a very short period, t_{on}, and then goes idle for time t_{off}. Its average sending rate is 100Mbps*$t_{on}(t_{on} + t_{off})^{-1}$. Only the throughput of the bursty source is plotted in Figure 8 since it shows the biggest discrepancies among different AFD algorithms. The TCP flows utilize the rest of the link bandwidth and the differences among those flows are small. Note that the left-most bars in the diagram represent the throughput that the on-off source gets when its average sending rate is only half of R_{fair}, in which case all three algorithms allocate the bandwidth fairly, i.e. provide the flow its request bandwidth. However, as the plot shows, when the on-off flow gets more bursty and sends above $2R_{fair}$, AFD-FT starts penalizing it. The more bursty a flow is, the more severe the penalty. Conversely, AFD-SB and AFD-NFT allocate bandwidth fairly; flows are not penalized for their burstiness.

Mixed TCP Traffic with one UDP source: Figure 9 represents a simulation case where the traffic mix is one UDP source sharing the link with 7 groups (5 flows per group) of TCP flows with different congestion control methods. For generalized window control mechanisms, the window increase has a form of $w + aw^{-k}$, and the decrease of a form $w - bw^{l}$. The 7 groups in the simulation have different values of (a,b,k,l) and RTTs, which is tabulated in Table II. Note that the normal TCP has the form of (1.0, 0.5, 1.0, 1.0). The right-most bars represent the throughput of the UDP flow under the different algorithms. The result shows once again that the AFD-NFT design can mimic the performance of AFD-SB while AFD-FT fails to do so.

B. Comparable Performance

Mixed TCP Traffic: We remove the UDP flow from the above simulation. Only TCP flows with various congestion parameters compete against each other. The results in Figure 10 show that AFD-NFT performs equally well in the cases where AFD-FT excels. All three AFD designs allocate bandwidth in a fair manner.

Different RTTs: AFD behaves reasonably well, though not ideally, in the cases where flows with different RTTs are sharing a link [14]. To exhibit that AFD-NFT does not perform worse, we perform this experiment. In this simulation, flows are separated into 4 groups, 10 flows in each group. The RTTs (propagation delay only) are 37.5ms, 75ms, 112.5ms and 150ms respectively. Figure 11 shows that AFD-NFT's performance is similar to that of AFD-SB and AFD-FT: although there are some discrepancies among flows with different RTTs, the differences are not significant.

V. MEMORY REQUIREMENT

In the previous session, we have shown that AFD-NFT provides reasonably fair bandwidth allocation. All the operations on the forwarding path are O(1). So the main question regarding whether AFD-NFT is practical or not lies in its memory requirement. Since the size of the set S is small(usually less than 10), the flow ids in S can be easily stored using registers. It is the flow table that requires some memory buffering.

The size of the flow table is directly related to the number of flows, N, present in the shadow buffer. In the various traces we have seen [14], N is typically less than one fourth of b, the number of packets in the logical shadow buffer. We also find that , in order to achieve a good performance, b should be roughly $10\frac{R}{r_{fair}}$. Hence, N equals $2.5\frac{R}{r_{fair}}$. It is hard to estimate the value of r_{fair} on a typical Internet link. To make a conservative estimate, we assume r_{fair} equals 56Kbps, the slow telephone modem speed. Then for a link capacity of 1Gbps, it is simple to obtain that N is on the order of a few thousand. Therefore, the flow table can be easily implemented using a standard hash table or CAM. The memory overhead is very limited.

VI. CONCLUSION

We have proposed a new flow table based AFD design, AFD-NFT. AFD-NFT reduces drastically the state requirement of AFD algorithm, and yet has virtually identical performance. This and other data suggests that in a wide range of scenarios AFD provides a good approximation to fair bandwidth allocation, typically providing bandwidth allocations within +/-15% of the fair share.

REFERENCES

[1] Bansal, D., and Balakrishnan, H., "Binomial Congestion Control Algorithms " *Proceedings of Infocom '2001*, April 2001.

[2] `http://www.caida.org/analysis/AIX/plen_hist/`

[3] Braden, B., Clark, D., Crowcroft, J., Davie, B., Deering, S.,Estrin, D., Floyd, S., Jacobson, V., Minshall, G., Partridge, C., Peterson, L.,Ramakrishnan,K., Shenker,S., Wroclawski, J., Zhang, L., "Recommendations on queue management and congestion avoidance in the internet", *IETF RFC (Informational) 2309*, April 1998.

[4] Demers, A., Keshav, S. and Shenker, S., "Analysis and simulation of a fair queueing algorithm", *Journal of Internetworking Research and Experience*, pp 3-26, Oct. 1990. Also in Proceedings of ACM SIGCOMM'89, pp 3-12.

[5] Feng, W., Shin, K., Kandlur, D. and Saha, D., "Stochastic Fair Blue: A Queue Management Algorithm for Enforcing Fairness", *Proceedings of INFOCOM'2001*, April, 2001.

[6] Floyd, S. and Jacobson, V., "Random Early Detection Gateways for Congestion Avoidance", *IEEE/ACM Transaction on Networking*, 1(4), pp 397-413, Aug. 1993.

[7] Floyd, S., and Fall, K., "Promoting the Use of End-to-End Congestion Control in the Internet", *IEEE/ACM Transactions on Networking*, August 1999.

[8] Lin, D. and Morris, R., "Dynamics of random early detection", *Proceedings of ACM SIGCOMM'97*, pp 127-137, Oct. 1997.

[9] Mahajan, R. and Floyd, S. "Controlling High-Bandwidth Flows at the Congested Router", ACIRI, Berkeley, California, Nov. 2000.

[10] McKenny, P., "Stochastic Fairness Queueing", *Proceedings of INFOCOM'90*, pp 733-740.

[11] Nagle, J., "On packet switches with infinite storage", *Internet Engineering Task Force*, RFC-970, December, 1985.

[12] Ott, T., Lakshman, T. and Wong, L., "SRED: Stabilized RED", *Proceedings of INFOCOM'99*, pp 1346-1355, March 1999.

[13] Pan, R., Breslau, L., Prabhakar, B. and Shenker, S., "Approximate Fairness through Differential Dropping (summary)", *Student Poster Session at Proceedings of SIGCOMM'01*, also appeared at *Computer Communication Review*, January 2002.

[14] Pan, R., Breslau, L., Prabhakar, B. and Shenker, S., "Approximate Fairness through Differential Dropping", *http://www.research.att.com/~breslau/papers/afd-techreport.ps.gz*, under submission.

[15] Pan, R., Prabhakar, B. and Psounis, K., "CHOKe - A Stateless Active Queue Management Scheme For Approximating Fair Bandwidth Allocation", *Proceedings of INFOCOM'00* March 2000.

[16] Stoica, I., Shenker, S. and Zhang, H., "Core-Stateless Fair Queueing: Achieving Approximately Fair Bandwidth Allocations in High Speed Networks", *Proceedings of ACM SIGCOMM'98.*

[17] ns - Network Simulator (Version 2.1b6).