Probabilistic Packet Scheduling: Achieving Proportional Share Bandwidth Allocation for TCP Flows

Ming Zhang, Randolph Wang, Larry Peterson
Department of Computer Science
Princeton University
35 Olden Street
Princeton, NJ 08544
{mzhang, rywang, llp}@cs.princeton.edu

Arvind Krishnamurthy
Department of Computer Science
Yale University
51 Prospect Street
New Haven, CT 06520
arvind@cs.yale.edu

***Abstract*— This paper describes and evaluates a probabilistic packet scheduling (PPS) algorithm for providing different levels of service to TCP flows. With our approach, each router defines a local currency in terms of tickets and assigns tickets to its inputs based on contractual agreements with its upstream routers. A flow is tagged with tickets to represent the relative share of bandwidth it should receive at each link. When multiple flows share the same bottleneck, the bandwidth that each flow obtains is proportional to the relative tickets assigned to that flow. Simulations show that PPS does a better job of proportionally allocating bandwidth than DiffServ and weighted CSFQ. In addition, PPS accommodates flows that cross multiple currency domains.**

***Index Terms*—Flow control, Scheduling, Quality of service.**

I. Introduction

Providing different levels of service among competing flows has been a subject of intense research in recent years. Early approaches, beginning with Weighted Fair Queuing (WFQ) and culminating in the Integrated Services architecture [1], [2], [3], were able to make strong promises about the level of service provided to a given flow, but at the expense of scalability since routers must maintain per-flow state. Subsequent development of the Differentiated Services architecture [4], [5], [6], [7], [8], segregated flows into a small number of service classes (making the solution scalable), but at the expense of being able to make only relative statements about the service a given flow receives.

This paper proposes an intuitively simple alternative based on probabilistic packet scheduling (PPS). It works as follows. Each router in the network defines its own currency in terms of *tickets* and assigns these tickets to its inputs based on the contractual agreements with other routers. A flow is first tagged at a TCP source with tickets—in the appropriate local currency—that represent the relative share of bandwidth it should receive. At each hop, the flow's tickets are traded for a new set of tickets—in terms of the target router's currency—according to an appropriate currency exchange rate. Routers probabilistically decide when to forward/drop packets based on the number of tickets and the current congestion level. When multiple flows share the same bottleneck link, PPS ensures that each flow obtains a bandwidth that is proportional to the tickets associated with that flow.

Our approach defines a new point in the design space for providing different levels of service to competing flows. Like DiffServ, PPS scales well since it does not require per-flow state, and it makes only relative differentiations among flows. However, PPS differs from DiffServ in the manner in which it allocates bandwidth when there is a disparity between the link capacities and network traffic load. DiffServ installs service profiles at end hosts to represent the target rates of flows. When the links have excess bandwidth or when they fail to match the target rates, the link bandwidth is allocated in a random manner, an observation that is supported by our simulations. PPS, on the other hand, always ensures that a flow will receive its proportional share of bandwidth at its bottleneck link.

More recent work, such as Core-Stateless Fair Queueing (CSFQ) [9] and CHOKe [10], have focused on approximating fair bandwidth allocation in a stateless way. However, PPS has the advantage of being able to provide different levels of service for flows traversing multiple domains. Although weighted CSFQ allows one to assign different weights to different flows, it does not achieve proportional bandwidth allocation for TCP flows as well as PPS. Also, CSFQ assigns each flow the same weight at all routers; it cannot accommodate situations where the relative weights of flows differ from router to router. In contrast, PPS allows a flow to trade tickets it is granted in the source domain for an equitable number of tickets in each target domain along its path. (This trade is governed by an exchange rate that is easily computed by the routers connecting the two domains, based on both static inter-domain service agreements and the dynamic traffic loads.) In PPS, each domain is free to make its own decision regarding bandwidth allocation, and these decisions can be insulated from other domains. This scheme is applicable in today's Internet due to the autonomy of service providers. Additionally, bandwidth allocation can be controlled through either sender-based or receiver-based schemes.

The rest of this paper describes PPS in more detail and presents the results of simulations that show that PPS achieves a proportional allocation of bandwidth among competing TCP

This work is supported in part by DARPA contract F30602–00–2–0561. Wang is supported by NSF Career Award CCR-9984790, and Krishnamurthy is supported by NSF Career Award CCR-9985304.

flows. The paper concludes with a discussion of the relative strengths and weaknesses of our approach, as compared with the alternatives.

II. Algorithm

Our goal is to develop an algorithm that enables networks to provide different levels of service in the framework of best effort delivery. The algorithm needs to satisfy the following properties. First, it should ensure that each network flow receives a share of the bandwidth that is consistent with contractual agreements. Second, the bandwidth allocations should be reactive to dynamic changes in network traffic. Third, the algorithm should not degrade link utilization.

In this section we describe an algorithm with these properties. It consists of three components: a packet tagger, a relabeler, and a packet scheduler. Initially, each packet is tagged with some number of tickets by a TCP source. The tag is updated at each hop along the path according to some local currency exchange rate. The tag is then used to determine whether or not to drop the packet should it encounter congestion. The resulting algorithm attains proportional bandwidth allocation without requiring routers to maintain per-flow state.

The primary result of this paper is that proportional bandwidth allocation for TCP flows can be achieved by strategically dropping a small number of packets. This result is surprising given the following aspects of the algorithm. First, the algorithm employs a variety of rate estimation algorithms each of which is fundamentally inaccurate. Second, when a packet is dropped, TCP sources react drastically by halving their transmission rates; normally routers cannot communicate with the TCP sources to make small changes to their transmission rates. Third, when a flow does not require its proportional share, a scenario that is likely to occur frequently, the algorithm proportionally distributes the excess bandwidth amongst the remaining flows. This ability to reallocate bandwidth in a proportional manner requires the PPS system to dynamically identify the excess bandwidth available at the routers, an activity that could introduce further inaccuracies. In spite of the above-mentioned characteristics and the resulting complex interactions between the various mechanisms that constitute the algorithm, a well-engineered PPS system can achieve proportional bandwidth allocation with a great level of accuracy.

A. Tickets

There are two entities of interest in the network—end hosts and routers—both of which are configured with a currency, in terms of tickets, and assign some number of tickets-per-second (t/s) to their inputs based on some contractual agreement. For a router, the inputs would be the various incoming links, while for an end host, the inputs would correspond to the TCP sources resident on the host. Suppose entity P issues OutTktRate t/s to input A. Packets arriving from A would be tagged with tickets in P's currency, but these tickets should not exceed OutTktRate t/s.

Figure 1 shows an example. The link capacities and ticket allocations for the configuration are indicated in the figure. For example, S_3 has issued, in its own currency, 1000 t/s to the link (S_0, S_3), which has a capacity of 10Mb/s. Suppose there are three TCP flows: A between S_0 and S_6, B between S_1 and S_6, and C between S_2 and S_6. In order for a packet to reach S_6 from S_0, it is first tagged with tickets by S_0 in S_3's currency. When it arrives at S_3, the tag is relabeled based on the currency exchange rate of S_3 and S_5 (which is 1500:900) before it can go to the next hop. This relabeling process continues until the packet reaches S_6. Based on the exchange rates, the ticket rate of A, B, and C will be converted to 600, 300, and 300 t/s, respectively in S_5's currency. Because the bandwidth of link (S_5, S_6) is only 1Mb/s, it is the bottleneck for flows A, B and C. When all three flows are active, A, B, and C should obtain 0.5Mb/s, 0.25Mb/s, and 0.25Mb/s, respectively. When B is silent, A and C should divide the full bandwidth of link (S_5, S_6) by a 3:1 ratio. When B has a constant bit rate of 0.16Mb/s, which is less than its proportional share bandwidth of 0.25Mb/s, B should obtain 0.16Mb/s while A and C should obtain 0.56Mb/s and 0.28Mb/s, respectively.

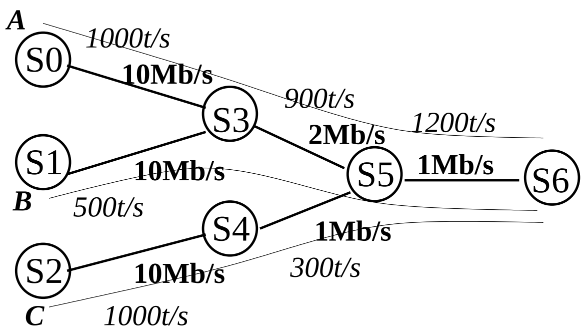

Fig. 1. Example of sender-based scheme where flows A, B, and C share the same bottleneck (S_5, S_6)

B. Ticket Tagging

Suppose TCP source A is issued OutTktRate t/s by its controlling entity, and let the throughput of A be AvgRate packets/second (p/s) at this point in time. We simply tag the outgoing packet with OutTktRate / AvgRate tickets. That is, the tickets on the outgoing packet are inversely proportional to the instantaneous throughput of A. To estimate the instantaneous throughput, we use the TSW algorithm described in [6].

TSW measures the instantaneous throughput, AvgRate, by revising its rate estimate upon the arrival of each ACK and decaying the past history over a time period, WinLength, which is a configurable parameter. It can therefore smooth the bursts of TCP traffic as well as be sensitive to instantaneous rate variations. The decision to employ a TSW rate estimation algorithm is primarily an engineering decision, and it can therefore be replaced by a different rate estimation algorithm without requiring changes to PPS.

C. Packet Scheduling

We first consider bandwidth allocation among one-hop TCP flows, then extend our discussion to multi-hop TCP flows in Sections II-D and II-E. Suppose at some instant there are n active TCP flows—$C_1, C_2, \ldots, C_n$—contending for link L. They are issued OutTktRate_1, OutTktRate_2, . . ., OutTktRate_n t/s by their controlling entity P. We use InTktRate and InPktRate to represent the t/s and p/s link L receives from all the flows.

Our scheduler needs to make the following guarantee: If a flow C_i tags AvgTkt = InTktRate / InPktRate tickets onto each of its packets, C_i should receive its proportional bandwidth share of InPktRate $\times$ OutTktRate$_i$ / InTktRate p/s. Here, InPktRate is close to the capacity of L during congestion. If we can ensure that each flow tags approximately the same number of AvgTkt tickets onto its packets, we will achieve proportional bandwidth allocation.

1) Ticket-Based RED (TRED): Our packet scheduling algorithm (Ticket-based RED) is based on the random early detection (RED) algorithm [11]. RED has three phases representing normal operation, congestion avoidance, and congestion control. During normal operation, RED does not drop any packets. During the congestion avoidance phase, each packet drop serves to notify the TCP source to reduce its sending rate. The exact drop probability is a function of the average queue length. The average queue length is calculated using a low-pass filter from the instantaneous queue lengths, which allows transient bursts in the queue. During the congestion control phase, all incoming packets are dropped.

Like RED, TRED also has three phases. It operates in the same way as RED in the normal operation and congestion control phases. During congestion avoidance, however, TRED uses a different method to calculate the drop probability. In addition to AvgQLen, TRED uses two other variables: ExpectTkt and InTkt; the former is an estimate of the number of tickets the link expects a flow to tag onto its packets, and the latter represents the number of tickets carried by an arriving packet. The TRED algorithm operates as follows.

```
Upon each packet arrival:
    compute AvgQLen the same as in RED
    compute InTktRate using TSW
    compute MinTkt as defined below
    if InTkt $<$ K $\times$ MinTkt
        compute ExpectTktRate
        and ExpectPktRate using TSW
    if AvgQLen $\leq$ MinThresh
        enqueue the packet
    if MinThresh $<$ AvgQLen $<$ MaxThresh
        calculate probability $p$ as in RED
        ExpectTkt = ExpectTktRate
                / ExpectPktRate
        $p = p\times$ (ExpectTkt / InTkt) $^3$
        if $p > 1$ then $p = 1$
        drop packet with probability $p$
    if MaxThresh $\leq$ AvgQLen
        drop the packet
```

where variable MinTkt represents the least number of tickets seen on an arriving packet during some interval. It is important to note that although TSW is applied on a per-flow basis in [6], we apply TSW to the aggregation of all traffic arriving on a particular link.

Now we consider what happens when the link is in congestion avoidance phase. We multiply the drop probability p by (ExpectTkt / InTkt)^3. As a result, those flows that put fewer tickets on their packets are more likely to lose packets and back off than those that tag more tickets onto their packets. When a flow backs off, its instantaneous throughput slows down, and it will begin to tag more tickets onto its packets. In the end, we expect all the flows to tag approximately ExpectTkt tickets on their packets.

As we said in the beginning of Section II-C, we optimally expect each flow to tag AvgTkt = InTktRate / InPktRate tickets onto its packets, but in reality, we cannot precisely calculate AvgTkt. However, the algorithm tries to keep ExpectTkt around AvgTkt. When ExpectTkt is less than AvgTkt, which means those flows are obtaining more than their proportional share of bandwidth, AvgQLen will increase accordingly. This causes the flows to slow down and tag more tickets onto their future packets. As a result, ExpectTkt will increase. On the other hand, when ExpectTkt is greater than AvgTkt, the algorithm will cause it to decrease. In the end, the algorithm tries to keep ExpectTkt around AvgTkt, so as to proportionally allocate bandwidth among the competing flows.

Within this overall strategy of using packet drops to nudge TCP flows to obtain proportional bandwidth shares, there are two issues to be addressed.

The first issue is how to scale the drop probability based on the number of tickets carried by the packet. We have experimented with a wide variety of functions and evaluated their effectiveness with respect to the three success metrics: proportional allocation, high link utilization, and quick convergence to an equilibrium state. When the drop probability p is multiplied by (ExpectTkt / InTkt)^n for certain values of n, the system achieves the desired goals listed above. The magnitude of n has an effect on system performance. For small values of n, proportional allocation is difficult to achieve as TRED fails to sufficiently penalize those TCP flows that manage to obtain a link bandwidth that is greater than their proportional share. For large values of n, the system becomes too reactive to small changes in flow throughputs resulting in unnecessary packet drops and some unpredictable effects. Our experience is that setting $n = 3$ allows the system to achieve the desired goals.

The second issue that TRED needs to address is how to estimate ExpectTkt when some of the flows are not making full use of their proportional shares, as might be the case with Constant Bit Rate (CBR) flows. We call these *rich flows* because they put more tickets onto their packets than normal flows and could potentially skew the estimated ExpectTkt value. To address this issue, we use a simple heuristic to filter out the packets of rich flows by considering only the packets in the range of [MinTkt, $K\times$ MinTkt]. As shown in the algorithm, we calculate ExpectTkt as the average number of tickets on only those packets that fall within this range. On the one hand, K should be large enough so that most packets of normal flows will fall into this range. From the above analysis of TRED, we know that most packets of normal flows will carry approximately the same number of tickets. On the other hand, packets of a rich flow that can only make use of less than $1/K$ of its proportional bandwidth will be filtered out. Because packets of some rich flows are not filtered out, the skewed ExpectTkt, which we denote ExpectTkt′, could be K times the correct ExpectTkt, in the worse case. That will cause the skewed dropping probability, denoted p', to be K^n times the correct p. Because in RED,

$$p = MaxP \times \text{(AvgQLen - MinThresh)/(MaxThresh - MinThresh)}$$

Suppose with the correct ExpectTkt and p, the system reaches the equilibrium state with AvgQlen. To correct for the skewed ExpectTkt$'$ and p', AvgQlen$'$ will decrease until

$$\text{(AvgQLen}' \text{ - MinThresh) = (AvgQLen - MinThresh) / } K^n$$

In our simulations, we choose K to be 2.

D. Tag Relabeling

A flow may go through many hops before reaching the destination. Because different entities may have their own local currencies, tickets in one currency are only meaningful to the entity that issues them. Thus, when going from one entity to another, we need to relabel the tags according to some currency exchange rate. We calculate the exchange rate at each link as follows:

$$\text{XRate = OutTktRate / InTktRate}$$

As before, InTktRate corresponds to the t/s the link receives at some instant. The OutTktRate is the t/s assigned to the link by its controlling entity. For each packet, we relabel its tag as follows:

$$\text{OutTkt = InTkt} \times \text{XRate}$$

In other words, the relabeling algorithm simply converts InTkt in one entity's currency to OutTkt in the currency of the next hop entity.

E. Multi-Hop Flows

A TCP flow C may utilize multiple links along its path. If the throughput of the flow does not increase when we increase the bandwidth of all links other than L, then L is the bottleneck for C. C may have several bottlenecks or no bottlenecks at all. In the latter case, the throughput of C is limited by its upper level application not by the network. Suppose flow C originates from source A, which has been assigned OutTktRate t/s by P. Also suppose that C's bottleneck is link (S, T). Based on the per-hop exchange rates from A to S, we can convert OutTktRate in P's currency into InTktRate$_c$ t/s in S's currency. Suppose the throughput of C is AvgRate, the total t/s and p/s of all flows of link (S, T) are InTktRate and InPktRate. We say flow C obtains its proportional share of bandwidth if:

$$\text{AvgRate = InPktRate} \times \text{InTktRate}_c \text{ / InTktRate}$$

Because link (S, T) is the bottleneck of C, it must be congested. This means that InPktRate is close to the capacity of the link. We now explain why flow C obtains its proportional share of bandwidth. Since link (S, T) is congested, from the TRED algorithm we know that each packet of C will carry approximately AvgTkt = InTktRate / InPktRate tickets when passing through link(S, T). As a result, the bandwidth that C obtains on link (S, T) is approximately:

$$\begin{aligned}\text{AvgRate}\\ &= \text{InTktRate}_c \text{ / AvgTkt}\\ &= \text{InPktRate} \times \text{InTktRate}_c \text{ / InTktRate}\end{aligned}$$

Because link (S, T) is the bottleneck of C, the throughput of C should equal the bandwidth that C obtains on the link. Thus, C will obtain its proportional share of bandwidth.

F. Receiver-Based Algorithm

In today's Internet, it's often the case that the receiver, rather than the sender, is a more appropriate entity to determine the level of service provided to traffic flows. For example, when mutiple users (receivers) are downloading files from a web server (sender), it is more appropriate to allocate bandwidth of the contended links according to the contractual agreements that the receivers have made. In the sender-based scheme, each entity defines its currency (S-currency) and assigns some t/s in S-currency to its input links or TCP sources. In the receiver-based scheme, each entity also defines its currency (R-currency) and assigns some t/s in R-currency to its output links or TCP sinks. The idea behind the receiver-based scheme is that we try to reconstruct S-currency from R-currency for each entity and compute how many t/s in S-currency an entity should issue to its input links or TCP sources. After this, we simply run the sender-based algorithm to achieve proportional bandwidth allocation among TCP flows. The receiver-based scheme assumes symmetric routing, which means the path of ACK packets is the reverse of that of data packets.

1) ACK Packet Tagging and Relabeling: In the sender-based scheme, only data packets are tagged by the TCP sources and then relabeled at each hop. In the receiver-based scheme, data packets are still tagged and relabeled as before. ACK packets are also tagged by the TCP sinks and then relabeled at each hop. The difference between the tags on data packets and the tags on ACK packets is that the former is used to calculate the drop probability of the data packet during congestion, while the latter is used to calculate how many t/s in S-currency an entity should assign to its input links or TCP sources.

The tagging and relabeling algorithms for ACK packets are similar to those for data packets.

Tagging algorithm at TCP sink:

- Before sending out an ACK packet, calculate the sending rate of ACK packets, AckAvgRate
- Tag the ACK packet with AckOutTktRate / AckAvgRate tickets.

Relabeling algorithm at link:

- Upon arrival of each ACK packet, calculate t/s carried by the ACK packets, AckInTktRate
- Calculate the exchange rate for an ACK packet as AckXRate = AckOutTktRate / AckInTktRate.
- Relabel the ACK packet with AckOutTkt = AckInTkt $\times$ AckXRate.

The AckOutTktRate stands for t/s that an entity assigns to its output links or TCP sinks in R-currency. We use the tickets carried by ACK packets to calculate how many t/s in S-currency that an entity should assign to its input links or TCP sources as follows:

OutTktRate = AckInTktRate

This means that the ticket rate a link or TCP source can tag on its outgoing data packets equals to the ticket rate it receives from the incoming ACK packets. From the above, we can deduce that at each link:

InTktRate = AckOutTktRate

This means that the ticket rate a link or a TCP sink receives from the incoming data packets equals to the ticket rate it tags on its outgoing ACK packets. So for each link, the exchange rate is:

XRate
= OutTktRate / InTktRate
= AckInTktRate / AckOutTktRate
= 1 / AckXRate

Having defined S-currency for each entity, we can now run the sender-based algorithm in the same way as described before.

2) Proportional Bandwidth Allocation: The bandwidth allocation for competing flows in the receiver-based algorithm is similar to that in sender-based algorithm. Suppose flow C goes from source B on end host Q to sink A on end host P. A is issued AckOutTktRate t/s by P and the bottleneck of the flow is link (S, T). Based on the per-hop AckXRate from A to T, we can convert AckOutTktRate in P's R-currency into AckInTktRate$_c$ t/s in T's R-currency. Suppose the throughput of flow C is AvgRate p/s. Link (S, T) receives AckInTktRate t/s from the ACK packets of all the flows. The throughput of the flows is InPktRate p/s, which is close to the capacity of link (S, T) during congestion. We say C obtains its proportional share of bandwidth if it satisfies:

AvgRate = InPktRate$\times$ AckInTktRate$_c$ / AckInTktRate

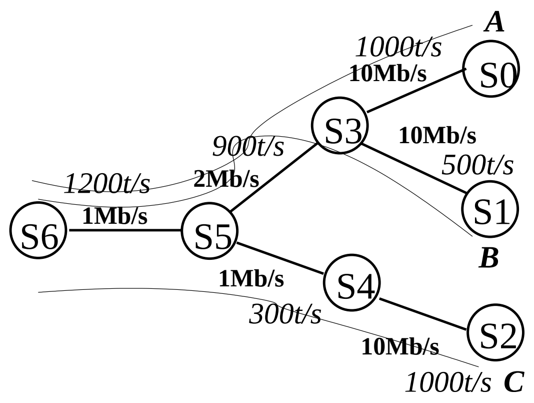

Fig. 2. Example of receiver-based scheme where flows A, B, and C share the same bottleneck (S_6, S_5)

For example, in Figure 2 there are three TCP flows: A from S_6 to S_0, B from S_6 to S_1, and C from S_6 to S_2. The bandwidth of link (S_6, S_5) is 1Mb/s, which is the bottleneck of the 3 flows. When all three flows are active, A, B, and C should obtain 0.5Mb/s, 0.25Mb/s, and 0.25Mb/s bandwidth, respectively.

G. Ticket Policing

As discussed in Section II-B and II-D, a TCP source or link tags each outgoing packet subject to the constraint that the rate at which tickets are consumed does not exceed OutTktRate t/s. To ensure the source or link adheres to this rate, we measure the actual ticket consuming rate, ActualTktRate, and then adjust the amount of tickets tagged to the packet as follows:

At source:

OutTkt = OutTktRate / AvgRate $\times$
OutTktRate / ActualTktRate

At links:

OutTkt = InTkt $\times$ XRate $\times$
OutTktRate / ActualTktRate

Without such an adjustment, a source or link may have a higher or lower ticket sending rate than was allocated to it.

III. Simulation Results

This section reports the results of several simulations designed to evaluate the algorithm's ability to proportionally allocate bandwidth among TCP flows. We use the ns-2 network simulator for our simulations [12]. We conducted each experiment on both sender-based and receiver-based scheme. The configurations of sender-based and receiver-based scheme are similar for each experiment except that the TCP sources and sinks are reversed, so we shall expect their simulation results to be similar for each experiment. In all experiments, the WinLength parameter used in the TSW algorithm is set to 5 seconds [6]. In RED, the MinThresh and MaxThresh are set to 5 and 55 packets, and the maximum dropping probability is 0.2. All experiments run for 200 seconds of simulated time.

A. One-Hop Configuration

Our first experiment studies our algorithm when there is a single congested link. We use the configuration shown in Figure 3, where 30 TCP flows share a 4.65Mb/s bottleneck link (P, Q). The flows are assigned an incremental number of t/s, ranging from 100 to 3000. The RTT for all flows is 26ms; we study the influence of RTT separately in another experiment. The throughputs are measured over the whole simulation. As shown in Figure 4, the achieved throughput is proportional to the number of t/s given to each flow. The link utilization of (P, Q) is 99%.

B. Multi-Hop Configuration

We next study the bandwidth allocation among multi-hop flows. Figure 5 shows the network configuration we used. It has 20 flows: A_1 - A_{10} and B_1 - B_{10}. All flows share the

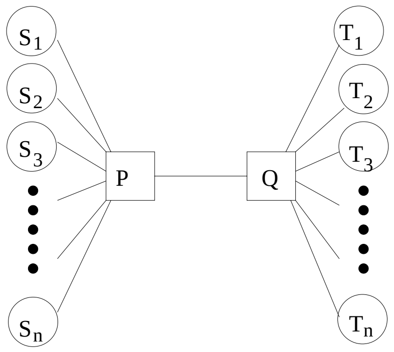

Fig. 3. One-hop configuration. All n flows share the same bottleneck (P, Q)

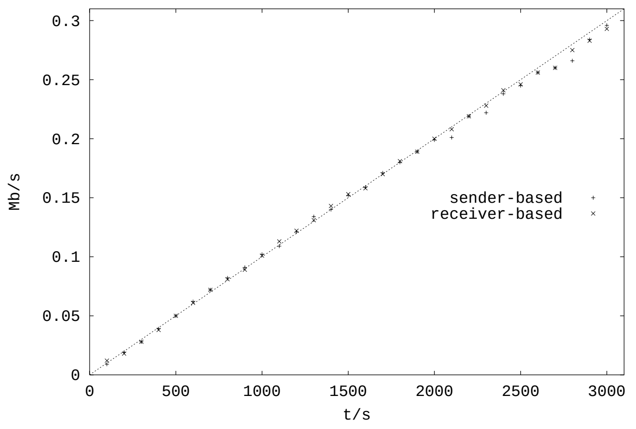

Fig. 4. Bandwidth allocation for one-hop configuration. The x and y coordinate of each point represent the t/s issued to the flow and measured throughput of the flow, respectively. The achieved throughput is proportional to the number of t/s given to each flow.

1.65Mb/s bandwidth of bottleneck link (P_3, P_4). Because link (P_1, P_3) is assigned twice as many t/s as link (P_2, P_3) by P_3, we expect that flow A_i obtains twice as much bandwidth as B_i ($1 \leq i \leq 10$). Also, among either A_1 to A_{10} or B_1 to B_{10}, the throughput of each flow should be proportional to the number of t/s given to that flow. As shown in Figure 6 they do. The link utilization of (P_3, P_4) is 99%.

C. Proportionally Sharing Unused Capacity

Sometimes a flow cannot make full use of its proportional share of bandwidth because the application generates bytes at a lower rate. The unused bandwidth should be proportionally allocated among the other flows so as to achieve high link utilization. To test the ability of our algorithm to achieve high link utilization in a proportional way, we again use the configuration from Figure 5, but this time the traffic from flow A_i, B_i ($6 \leq i \leq 10$) are generated by an application that transmits at a fixed rate of 0.03Mb/s, less than their share of bandwidth, so each of them should obtain 0.03Mb/s bandwidth. We expect flows A_i, B_i ($1 \leq i \leq 5$) to divide the excess bandwidth proportionally. As shown in Figure 7, this is exactly what happens. The link utilization of (P_3, P_4) is 99%.

D. Variable Traffic

This experiment evaluates how well the algorithm adjusts to variations in the source sending rate. We use the same configuration as in Figure 1, but when the simulation begins, only flows B and C are active; flow A becomes active after 100 seconds. The results are shown in Figure 8. As the plot clearly shows, B and C obtain approximately 0.75Mb/s and 0.25Mb/s of bandwidth from bottleneck link (S_5, S_6) in the first 100 seconds, because they have 900 and 300 t/s in S_5's currency, respectively. After A starts up, A, B, and C quickly converge to 0.5Mb/s, 0.25Mb/s, and 0.25Mb/s, respectively.

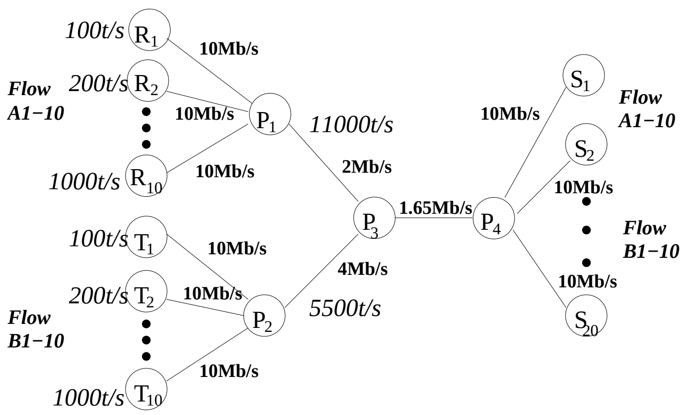

Fig. 5. Multi-hop configuration. All 20 flows share the same bottleneck (P_3,P_4)

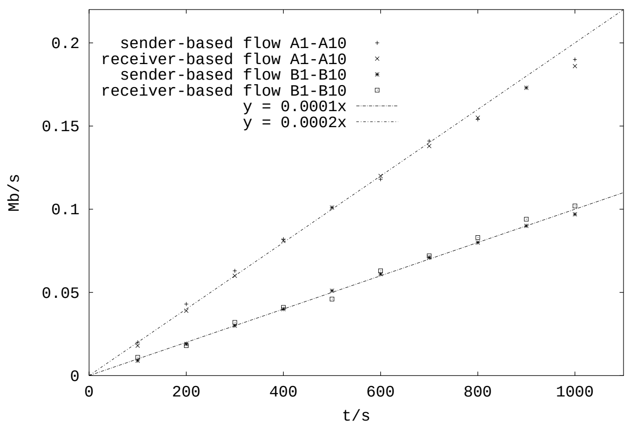

Fig. 6. Bandwidth allocation for multi-hop configuration. Flow A_i obtains twice as much bandwidth as B_i ($1 \leq i \leq 10$).

E. Variable Ticket Allocation

Our algorithm can flexibly control bandwidth allocation among TCP flows by dynamically adjusting the rate at which tickets are issued. This permits an application to adjust its ticket share in an effort to maintain a certain transmission speed. To see this, we again use the topology in Figure 1, where each link is issued the same amount of tickets in the beginning. After 100 seconds, the t/s issued to link (S_0, S_3) changes from 1000 to 600, and the t/s issued to link (S_1, S_3) changes from 500 to 900. We expect the throughput of A to change from 0.5Mb/s to 0.3Mb/s and the throughput of B to change from 0.25Mb/s to 0.45Mb/s. The throughput of C should not change. As can be seen in Figure 9, the system behaves as expected. The important point is that our algorithm keeps bandwidth allocation decisions local. That is, the variation of A and B has virtually no influence on C.

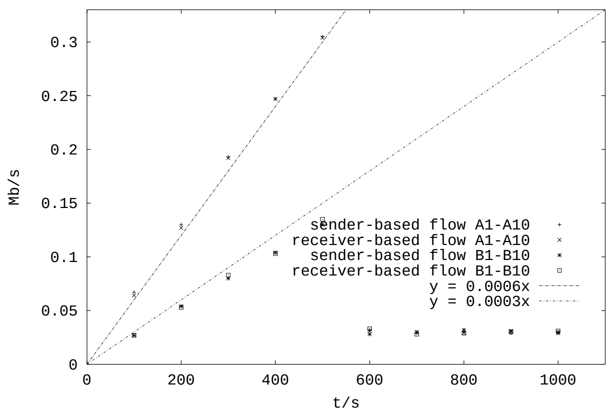

Fig. 7. Proportional sharing of unused bandwidth. Each of flows A_i, B_i ($6 \leq i \leq 10$) consumes 0.03Mb/s bandwidth. Flows A_i, B_i ($1 \leq i \leq 5$) divide the excess bandwidth proportionally.

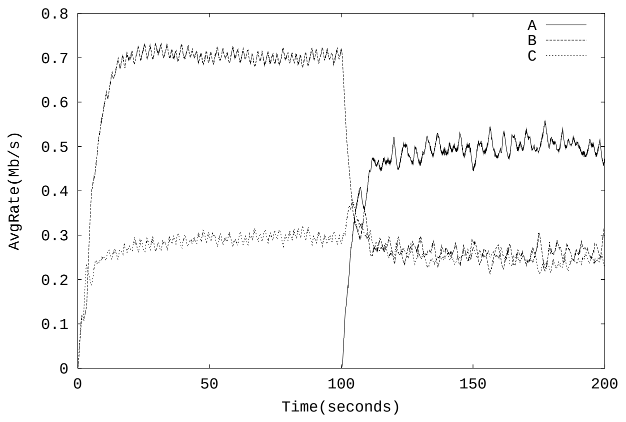

Fig. 8. Adapting to new traffic. The x-axis is time and the y-axis is instantaneous throughput. Flows B and C obtain approximately 0.75Mb/s and 0.25Mb/s of bandwidth from bottleneck (S_5, S_6) in the first 100 seconds. After A starts up, A, B, and C converge to 0.5Mb/s, 0.25Mb/s, and 0.25Mb/s, respectively

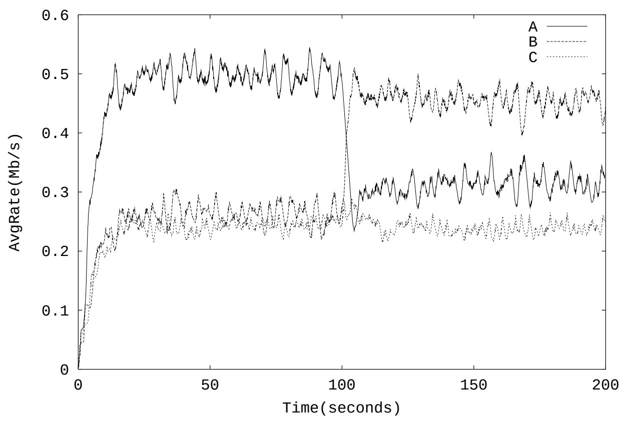

Fig. 9. Bandwidth allocation as ticket rates change. As the t/s issued to link (S_0, S_3) changes from 1000 to 600 and the t/s issued to link (S_1, S_3) changes from 500 to 900, the throughput of A changes from 0.5Mb/s to 0.3Mb/s, and the throughput of B changes from 0.25Mb/s to 0.45Mb/s. The throughput of C does not change.

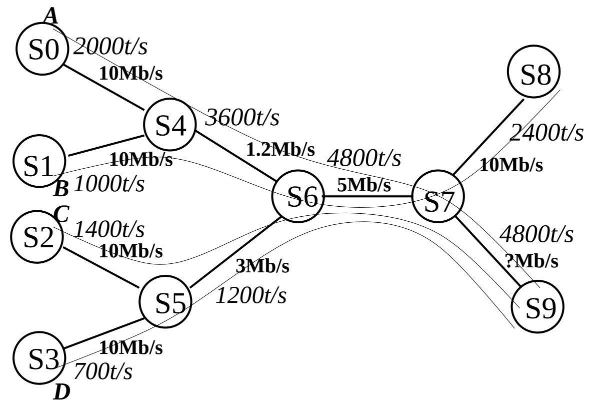

Fig. 10. Complex configuration with multiple bottlenecks. When the bandwidth of (S_7, S_9) is 0.9Mb/s, A shares the same bottleneck (S_7, S_9) with C and D. When the bandwidth of (S_7, S_9) is 1.8Mb/s, A shares the same bottleneck (S_4, S_6) with B.

F. Multiple Output Links

In all the experiments up to this point, the flows share a common bottleneck link, and each router has only one output link. In this experiment, we study how the algorithm behaves when different flows have different bottlenecks and routers have multiple output links. We ran 2 experiments using the topology given in Figure 10. In this scenario, there are 4 flows—A, B, C, and D—running between (S_0, S_9), (S_1, S_8), (S_2, S_9), and (S_3, S_9), respectively.

In the first experiment, we set the bandwidth of (S_7, S_9) to 0.9Mb/s. The bottleneck of flows A, C, and D is (S_7, S_9), while the bottleneck of flow B is (S_4, S_6). The 2000, 1400, and 700 t/s of flow A, C, and D are converted to 2400, 800, and 400 t/s in S_7's currency. So A, C, and D should divide the bandwidth of (S_7, S_9) by 6:2:1. Moreover, since A is a rich flow of (S_4, S_6) while B is not, B should obtain the remaining 0.6Mb/s bandwidth of (S_4, S_6). The measured results are shown in Table I.

Note that although flow A has twice as many t/s as B, the actual bandwidth A obtains is almost the same as B. This may seem unfair at first glance, but because A and B are going to different destinations, they have different bottlenecks in the network. The throughput of A is limited by (S_7, S_9), so it cannot make full use of its proportional share of bandwidth at (S_4, S_6), and the unused bandwidth of A is allocated to B. From this experiment, we know that the actual bandwidth obtained by a flow is related to both its assigned ticket rate and its bottleneck.

In the second experiment, we change the bandwidth of (S_7, S_9) to 1.8Mb/s. Now the bottleneck of flows A and B is (S_4, S_6), while the bottleneck of flows C and D is still (S_7, S_9). Flows A and B should divide the 1.2Mb/s bandwidth of (S_4, S_6) by 2:1. Again, although flow A, C, and D have 2400, 800, and 400 t/s in S_7's currency, A is limited by its bottleneck at (S_4, S_6); C and D should share the remaining bandwidth of (S_7, S_9) by 2:1. The actual results are shown in Table II.

G. RTT Biases

It is well-known that TCP has a bias against flows with large round trip times (RTT). To understand the relationship between our algorithm and RTT, we conduct two sets of experiments with configurations depicted in Figure 3, where 10 flows share a bottleneck link (P, Q) of 1.1Mb/s.

In scenario I, all flows are assigned 200 t/s. We first set all the RTTs to 30ms. This setting serves to demonstrate that our algo-

TABLE I

Multiple Bottlenecks: Scenario I. A, C, and D share the same bottleneck (S_7, S_9) and proportionally divide its capacity.

Flow	Sender-based Rate (Mb/s)	Receiver-based Rate (Mb/s)	Expected Rate (Mb/s)
A	0.56	0.55	0.60
B	0.64	0.65	0.60
C	0.23	0.23	0.20
D	0.11	0.12	0.10

TABLE II

Multiple Bottlenecks: Scenario II. A and B share the same bottleneck (S_4, S_6) and proportionally divide its capacity.

Flow	Sender-based Rate (Mb/s)	Receiver-based Rate (Mb/s)	Expected Rate (Mb/s)
A	0.74	0.74	0.80
B	0.45	0.45	0.40
C	0.69	0.68	0.67
D	0.36	0.37	0.33

rithm is able to fairly split bandwidth among competing flows. We then let the RTT of the flows vary, incrementally, from 30ms to 300ms. Optimally, each flow should obtain 0.110Mb/s bandwidth. As can be seen from Table III, when all the RTTs are the same, our algorithm can more fairly allocation bandwidth than Reno TCP with RED. When RTTs vary, although there is still a bias against long-RTT flows in our algorithm, it's much better than that of Reno TCP. This is because when a flow with large RTT backs off, it begins to tag more tickets onto each of its packets. When contending with other flows, its packets are more likely to get through and the flow recovers to its share of bandwidth faster than Reno TCP.

TABLE III

Variable RTT: Scenario I. σ is the standard deviation. PPS can more fairly allocation bandwidth than Reno TCP.

RTT ms	PPS Mb/s	Reno Mb/s	RTT ms	PPS Mb/s	Reno Mb/s
30	0.107	0.108	300	0.087	0.060
30	0.110	0.099	270	0.090	0.055
30	0.110	0.105	240	0.098	0.071
30	0.109	0.118	210	0.100	0.086
30	0.107	0.112	180	0.105	0.079
30	0.107	0.113	150	0.110	0.114
30	0.108	0.118	120	0.118	0.115
30	0.114	0.099	90	0.122	0.127
30	0.109	0.116	60	0.123	0.159
30	0.110	0.108	30	0.129	0.233
σ	0.002	0.007	σ	0.014	0.051

In scenario II, each flow is assigned an incremental number of t/s, ranging from 100 to 1000. We first set the RTT of flow 1, which has 100 t/s, to 260ms, and the RTT of all other flows to 26ms. We then reset the experiment so that the RTT of flow 10, which has 1000 t/s, is 260ms and the RTT of all other flows is 26ms. As we can see from the results shown in Figure 11, a larger RTT has a more negative influence on flow 10 than flow 1. This is because the proportional share of bandwidth of flow 10 is greater than that of flow 1. When both flows back off, flow 10 loses more bandwidth than flow 1. So under a large RTT, it takes longer for flow 10 to recover to its share of bandwidth than for flow 1.

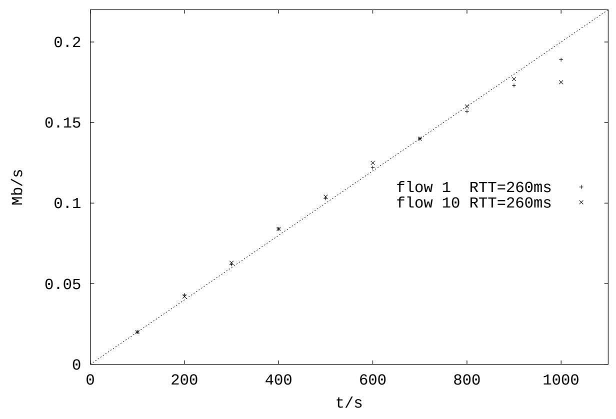

Fig. 11. Variable RTT: Scenario II. A larger RTT has more negative influence on a flow with larger ticket share.

H. Comparison with DiffServ

DiffServ installs service profiles at end hosts and tags each packet with one bit (in/out) to indicate if the packet is beyond the limits set by its service profile [6]. When congestion happens, routers preferentially drop packets sent outside the profile. DiffServ works well when the link capacity matches the service profiles, but this condition is inherently hard to achieve. Because the service profiles are just expected sending rates, they do not take into account the full path taken by flows. It is possible that many flows are contending for some link in the middle of the network, or those links that were expect to be shared are temporarily idle. It is impossible to guarantee that the link capacity matches the total target profile rate of contending flows at any time at any place in the network. But in reality, the service profiles are usually associated with the user payments. A user who pays twice the amount that another user pays would expect to receive twice as much bandwidth. So it would be reasonable to allocate bandwidth in proportion to the service profiles.

To evaluate the impact of this effect, we run a series of experiments that measure the behavior of DiffServ when there is a mismatch between link capacity and service profiles. We use the topology in Figure 3, which has 3 flows (A, B, and C) contending for bottleneck link (P, Q) with a bandwidth of 1.2Mb/s. In each experiment, the ticket assignment used by PPS are at the same ratio (3:2:1) as the service profiles, and our algorithm allocates bandwidth in proportion to the service profiles of the three flows, independent of the available capacity.

As we had expected, when the target sending rate (service profile) of A, B, and C match the 1.2Mb/s capacity of the shared link, DiffServ and our algorithm work equally well. When the target rate of A, B and C are below the available link capacity,

TABLE IV

DIFFSERV CANNOT PROPORTIONALLY ALLOCATE BANDWIDTH WHEN PROFILES DO NOT MATCH CAPACITY (MB/S).

Flow	Service Profile	Measured DiffServ	Service Profile	Measured DiffServ
A	0.15	0.43	1.20	0.42
B	0.10	0.40	0.80	0.43
C	0.05	0.37	0.40	0.35
Total	0.30	1.20	2.40	1.20

DiffServ allocates the excess bandwidth arbitrarily. When the expected sending rate of A, B and C exceed the capacity available on the link, many in-profile packets are dropped, causing DiffServ to degenerate into best effort. This is shown in Table IV.

I. Comparison with CSFQ

Stoica, Zhang, and Shenker propose Core-Stateless Fair Queueing (CSFQ)[9] to achieve fair queueing without using per-flow state in the core of an island of routers. Packets are marked with their sending rate at the edge routers, then core routers preferentially drop packets from a flow based on its fair share and the rate encoded in the packets. They also extend CSFQ to weighted CSFQ by assigning different weights to different flows.

Although CSFQ is able to fairly allocate bandwidth for TCP flows, weighted CSFQ does not work as well in proportionally allocating bandwidth among TCP flows. Because TCP is self-adaptive, we would expect a set of competing TCP flows to obtain approximately equal share of bandwidth under some scheduling strategy, such as CSFQ, as long as this strategy does not have a bias against any particular TCP flow. But it does not mean that the scheduling strategy can also achieve proportional bandwidth allocation. As an example, we run weighted CSFQ[13] using the same configuration as in Section III-A, except that each flow is assigned an incremental weight, ranging from 1 to 30, instead of tickets. Comparing Figure 12 with Figure 4, we can see that weighted CSFQ does not achieve proportional bandwidth allocation for TCP flows as well as PPS.

The reason is weighted CSFQ tries to drop the portion of a flow's packets that exceed its weighted share of bandwidth. While weighed CSFQ can achieve proportional bandwidth allocation for non-responsive flows, such as UDP, with a high accuracy, it does not do equally well for TCP flows. A TCP flow will cut down its sending rate by half in response to a single packet drop [14], so dropping all those packets that exceed a flow's weighted share of bandwidth is a little too drastic for TCP and may lead to some unpredictable effects. In comparison, PPS nudges TCP in a more gentle and controlled fashion.

Weighted CSFQ can assign different weights to different flows, but each flow has the same weight at all routers. This can be viewed as a special case of PPS—all the entities have the same currency and keep all the exchange rates as 1. However, there need not be a single currency in PPS. Each entity is free to adopt its own currency, so each router has more flexibility in bandwidth allocation. For example, in Figure 1, if S_5 wishes to allocate more bandwidth to flows coming from link (S_3, S_5), it can simply assign more tickets to that link. In CSFQ, however, S_5 has to trace to S_0 and S_1 and ask them to assign more weights to their incoming flows. This process would be difficult when the number of flows is large or when a flow goes through many links to reach S_5. Also in PPS, each entity is free to make its own decision regarding bandwidth allocation, and those decisions can be insulated from other entities. For example, in Figure 1, while S_3 can freely decide how many tickets should be assigned to link (S_0, S_3) and (S_1, S_3), it cannot influence the allocations by S_4 and S_5. As shown in Section III-E, when S_3 changes the ticket allocation, the variation of flows A and B has no influence on C, although C shares link (S_5, S_6) with A and B. But in CSFQ, if the weight of a flow is changed, it may influence any flow that share a link with it. Finally, in PPS the bandwidth allocation can be controlled through both sender-based and receiver-based schemes. This would be important in many scenarios as we explained in the beginning of Section II-F.

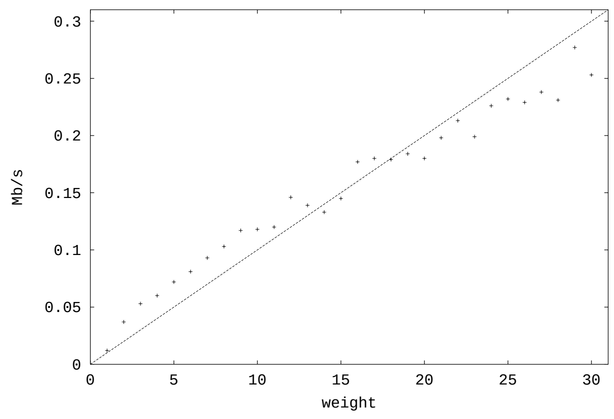

Fig. 12. Bandwidth allocation for weighted CSFQ. The x and y coordinate of each point represent the weight of the flow and measured throughput of the flow.

J. Ticket Bits

We use 8-bit tags in all the experiments reported in this section, which means that the number of tickets on each packet is in the range [0, 255]. We have experimented with both fewer and more bits, and as one would expect, the more bits we use, the finer differentiation we can make among TCP flows. Also with more bits, we have more flexibility in configuring the entities. Because tickets are relabeled at each hop according to some currency exchange rate, we need to carefully configure the currency of each entity and its contractual agreements with other entities. Otherwise when trading tickets from one currency to another, it's possible that the tickets go beyond the maximum limit or drop down to 0. There are two points to make, however. First, the number of bits needed is not related to the number of hops across the network but to the exchange rates of each hop. Hence, we are not concerned that more complex topologies will require more bits as long as the exchange rates are configured in a reasonable way. Second, the number of bits needed is dependent on the number of levels of service one wants to provide at a given router. It is independent of the number of flows one is trying push through the router, because most competing flows in a router will tag approximately the same number of tickets on their packets.

From a practical point of view, Stoica and Zhang describe how the 13-bit ip_off field in IP header can be added to the 4 bits from the type of service (TOS) to create a 17-bit tag [9]. Our simulations suggest that this is enough for our approach.

IV. Related Work

PPS was directly motivated by lottery scheduling [15], which is a mechanism by which an OS grants tickets to processes competing for CPU cycles. Like lottery scheduling, the most powerful aspect of PPS is that the bandwidth obtained by a flow is proportional to the relative share of tickets it has.

There are other schemes that can achieve proportional bandwidth allocation, *e.g.*, [16], [17]. In addition to proportional fairness, PPS better accommodates flows traversing multiple domains by exchanging tickets between different local currencies.

CHOKe tries to approximate fair bandwidth allocation in a stateless way. A packet is matched with a random packet in the queue. If they are from the same flow, both packets are dropped, If not, the packet is admitted with a certain probability. By doing this, flows with higher sending rate will get punished. CHOKe may encounter problem when the number of flows is large while the queue size is limited. But it does not require an extra field in packet header like CSFQ and PPS.

Turning to other schemes, FRED [18] monitors only active flows in the queue and determines the dropping probability of a flow according to the buffer space it consumes. SRED [19] identifies the misbehaving flows by caching the recently seen flows. BRED [20] maintains state for flow having packets in buffer and makes accept/drop decisions based on the number of packets that the flow has in buffer.

V. Conclusions

This paper describes probabilistic packet scheduling algorithm for providing different level of service among TCP flows. Simulations shows that it proportionally allocates bandwidth among competing flows that share the same bottleneck. The bandwidth allocation decision can be controlled in both a sender-based and a receiver-based way. Our future work will involve extending it to UDP flows, dynamically adjusting RED queue parameters, studying receiver-based scheme with asymmetric routing and experimenting it under more realistic traffic load and complex topologies.

Acknowledgments

The authors would like to thank Limin Wang, Brad Karp and the anonymous reviewers for their useful comments.

References

[1] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," *Proceedings of ACM SIGCOMM*, pp. 362–373, Aug. 1989.
[2] S. Shenker, R. Braden, and D. Clark, "Integrated services in the internet architecture: an overview," *Internet RFC 1633*, June 1994.
[3] I. Stoica and H. Zhang, "Providing guaranteed services without per flow management," *Proceedings of SIGCOMM*, pp. 81–94, Aug. 1999.
[4] K. Nichols, V. Jacobson, and L. Zhang, "An approach to service allocation in the internet," *Internet Draft*, Nov. 1997.
[5] D. Clark and J. Wroclawski, "An approach to service allocation in the internet," *Internet Draft*, July 1997.
[6] D. Clark and W. Fang, "Explicit allocation of best-effort packet delivery service," *IEEE/ACM Transactions on Networking*, vol. 6, no. 4, pp. 362–373, Aug. 1998.
[7] Z. Wang, "User-share differentiation (usd) scalable bandwidth allocation for differentiated services," *Internet Draft*, Nov. 1997.
[8] J. Crowcroft and P. Oechslin, "Differentiated end-to-end internet services using a weighted proportional fair sharing tcp," *Computer Communication Review*, vol. 28, no. 3, July 1998.
[9] I. Stoica, S. Shenker, and H. Zhang, "Core-stateless fair queueing: achieving approximately fair bandwidth allocations in high speed networks," *Proceedings of SIGCOMM*, pp. 118–130, Aug. 1998.
[10] R. Pan, B. Prabhakar, and K. Psounis, "Choke, a stateless active queue management scheme for approximating fair bandwidth allocation," *IEEE INFOCOM*, Mar. 2000.
[11] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, July 1993.
[12] ns2 (online), "http://www.isi.edu/nsnam/ns," .
[13] csfq (online), "http://www.cs.berkeley.edu/˜istoica/csfq/," .
[14] V. Jacobson and M. Karels, "Congestion avoidance and control," *Proceedings of ACM SIGCOMM*, pp. 314–329, Aug. 1988.
[15] C. Waldspurger and W. Weihl, "Lottery scheduling: flexible proportional share resource management," *Proceedings of OSDI*, pp. 1–12, Nov. 1994.
[16] S.H.Low and D.E.Lapsley, "Optimization flow control, i: basic algorithm and convergence," *IEEE/ACM Transactions on Networking*, vol. 7, no. 6, pp. 861–874, Dec. 1999.
[17] F.P.Kelly, "Charging and rate control for elastic traffic," *European Transactions on Telecommunications*, vol. 8, pp. 33–37, 1997.
[18] D. Lin and R. Morris, "Dynamics of random early detection," *Proceedings of ACM SIGCOMM*, pp. 127–137, Sept. 1997.
[19] T. Ott, T. Lakshman, and L. Wong, "Sred: stabilized red," *IEEE INFOCOM*, pp. 1346–1355, Mar. 1999.
[20] F. Anjum and L. Tassiulas, "Fair bandwidth sharing among adaptive and non-adaptive flows in the internet," *IEEE INFOCOM*, pp. 1412–1420, Mar. 1999.