Probabilistic Packet Scheduling: Achieving Proportional Share Bandwidth Allocation

Ming Zhang, Randy Wang, Larry Peterson, Arvind Krishnamurthy*
Department of Computer Science
Princeton University

Technical Report TR–632–01
February 5, 2001

Abstract

This paper describes and evaluates a probabilistic packet scheduling algorithm for achieving proportional bandwidth allocation among TCP flows. With our approach, either end hosts or edge routers tag each packet with a ticket that represents the share of network bandwidth this flow should receive. Routers then probabilistically decide when to forward/drop a packet based on the value of this ticket and the current congestion level. Our approach accommodates network topologies that span multiple domains by allowing a packet to trade in the tickets it was granted in the source domain for an equitable number of tickets in the target domain. The bandwidth allocation can be controlled by either a sender-based or a receiver-based version of the algorithm.

1 Introduction

Achieving a fair allocation of network bandwidth 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 [2, 8], 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 [6, 1, 13], 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. Recent work by Stoica, Zhang, and Shenker, called Scalabe Core (SCORE) [9, 10, 11], attempts to bridge this gap by having only edge routers maintain per-flow state, and then encoding this state in packets for use by core routers in the middle of the network.

This paper proposes an intuitively simple alternative based on probabilistic packet scheduling (PPS). The approach is inspired by the probabilistic lottery scheduling algorithm used by some CPU schedulers [12]. It works as follows. Either end hosts or edge routers tag each packet with a ticket that represents the share of network bandwidth this flow should receive. Routers then run a variant of the RED algorithm [3] to probabilistically decide when to forward/drop a packet based on the value of this ticket and the current congestion level. Routers also adjust the ticket assigned to a packet based on the level of competition for its outgoing links.

Our approach is difficult to couch as a simple variant of one of the known techniques. It really defines a new point in the design space for bandwidth allocation.

- Like WFQ, PPS gives each flow a weighted share of the available capacity. However, it does this probabilistically rather than requiring per-flow state.

- Like DiffServ, PPS scales well since it does not require per-flow state, and it makes only relative distinctions among flows. However, PPS has two important advantages over DiffServ. First, it uses more bits to represent the ticket than DiffServ uses to distinguish between classes, which makes it possible for PPS to differentiate service with a higher level of accuracy. Second, PPS better accommodates network topologies that span multiple domains by allowing a packet to trade in the tickets it was granted in the source do-

*This work supported in part by DARPA contract F30602–00–2–0561. Krishnamurthy's address is Department of Computer Science, Yale University, 312 AK Watson, New Haven, CT 06520.

main for an equitable number of tickets in the target domain. This trade—resulting in packet relabeling—is governed by an exchange rate that is easily computed by the router connecting the two domains, based on both static inter-domain service agreements and the dynamic traffic flow.

- Like SCORE, PPS scales well since it does not require per-flow state. However, PPS makes a different tradeoff of simplicity versus hardness of guarantees. That is, SCORE makes absolute bandwidth guarantees (PPS makes only relative differentiations), but at the expense of requring a non-trivial admission control mechanism. PPS does not require network-wide admission control, although a given source host or edge router is free to deny new flows so as to ensure that existing flows receive an adequate share of the network. A source is also free to reallocate its ticket shares in attempt to maintain a particular bit rate.

The rest of this paper describes PPS in more detail, and presents the results of a comprehensive set of simulations that show that PPS achieves a fair allocation of network bandwidth among a collection of competing TCP flows. The paper concludes with a discussion of the relative strengths and weaknesses of our approach, as compared with the alternatives.

2 Algorithm

This section describes PPS, which 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 the local workload. The tag is then used to determine whether or not to drop the packet should it encounter congestion. We have developed both sender-based and receiver-based versions of PPS. We first introduce the sender-based algorithm, and postpone discussion of receiver-based algorithm until Section 2.7.

2.1 Tickets

There are two entities of interest in the network—end hosts and routers—both of which define a currency in terms of tickets, and assign some number of tickets-per-second (t/s) to their inputs in a manner that reflects the relative importance of the inputs. 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 thse tickets should not exceed OutTktRate t/s. If at some instant there are packets going from A to P, we call A an active source of P, and all of A's tickets issued by P are active tickets. P, in turn, could have multiple output links. Suppose L is one of them, we let InTktRate represent the active t/s that L receives from P.

Before giving the definition of fairness, we define a *bottleneck* for a TCP flow. TCP flow C may utilize multiple links from source to destination. If the bandwidth of the flow does not increase when we increase the bandwidth of all links other than L, then L is the bottleneck for flow 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 link L is the bottleneck for flow C. We call C a *poor* flow of L. If L is not the bottleneck of C, we call C a *rich* flow of L. We use $\mathsf{InTktRate}_p$ and $\mathsf{InPktRate}_p$ to represent the total t/s and packets-per-second (p/s) L receives from all of its poor flows, and use $\mathsf{InTktRate}_r$ and $\mathsf{InPktRate}_r$ to represent the total t/s and p/s that L receives from all of its rich flows. If L has at least one poor flow, it must be in congestion. If the capacity of L is Capacity p/s, we have:

$$\mathsf{InTktRate} = \mathsf{InTktRate}_p + \mathsf{InTktRate}_r$$
$$\mathsf{Capacity} = \mathsf{InPktRate}_p + \mathsf{InPktRate}_r$$

2.2 Fairness for One-Hop Flows

We first consider bandwidth allocation among one-hop TCP flows; we extend our discussion to multi-hop TCP flows in Sections 2.5 and 2.6. Suppose at some instant there are n active TCP sources $C_1, C_2, \ldots, C_n$, contending for link L. They are issued $\mathsf{OutTktRate}_1$, $\mathsf{OutTktRate}_2$, …, $\mathsf{OutTktRate}_n$ t/s by the controlling entity P. Link L receives $\mathsf{InPktRate}_p$ p/s and $\mathsf{InTktRate}_p$ t/s from all its poor flows. We say that C_i obtains its "fair" share of bandwidth if one of the following two conditions is satisfied:

1. If C_i is a rich flow of L, C_i obtains the bandwidth it requires;

2. If C_i is a poor flow of L, C_i obtains $\mathsf{InPktRate}_p \times \mathsf{OutTktRate}_i$ / $\mathsf{InTktRate}_p$ p/s of bandwidth.

Notice that rich flows do not make full use of their fair share of bandwidth while poor flows try to obtain

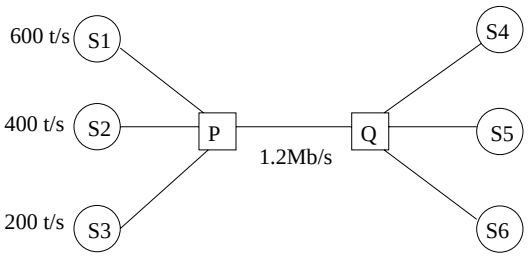

Figure 1: Example of Fair Allocation of Bandwidth

as much bandwidth as they can. For example, in Figure 1, entity P defines 1200 t/s in its currency, and it issues 600, 400 and 200 t/s to TCP sources C_1, C_2 and C_3, respectively. If C_1, C_2 and C_3 are all poor flows of L, they should divide the bandwidth of L at the ratio of 3:2:1. If C_1 has a constant bit rate (CBR) of 0.3Mb/s, which means it is a rich flow of L, C_1 should obtain 0.3Mb/s bandwidth while C_2 and C_3 should divide the remaining 0.9Mb bandwidth of L at the ratio 2:1. Similarly, if C_1 is silent, C_2 and C_3 should divide the entire bandwidth of L at the ratio of 2:1.

2.3 Ticket Tagging

Before sending a packet, the TCP source tags it with some number of tickets. Later, during the transmission of the packet through congested links along its path, the tag is used to determine the probability of dropping the packet. Suppose TCP source A is issued OutTktRate t/s by its controlling entity and the throughput of A is AvgRate 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 [1].
Initially,

```
WinLength = CONSTANT;
AvgRate = 0;
TswFront = 0;
```

Upon the arrival of each ACK packet, TSW update its state variables as follows:

```
PacketsInTsw = AvgRate × WinLength;
NewPackets = PacketsInTsw
        + AckedPackets;
AvgRate = NewPackets
        / (now - TswFront + WinLength);
TswFront = Now;
```

TSW maintains three state variables: WinLength, which is pre-configured and measured in units of time; AvgRate, which is the estimated instantaneous throughput; and TswFront, which is the arrival time of last ACK. The rate estimator can smooth the bursts of TCP traffic as well as be sensitive to instantaneous rate variation. TSW estimates the throughput upon the arrival of each ACK and decays the past history over time.

2.4 Packet Scheduling

Suppose at some instant there are n active TCP flows—C_1, C_2, …, C_n—contending for link L. They are issued $\mathsf{OutTktRate}_1$, $\mathsf{OutTktRate}_2$, …, $\mathsf{OutTktRate}_n$ t/s by their controlling entity P. Without loss of generality, suppose $C_1, C_2, \ldots, C_m$ are poor flows ($0 \leq m \leq n$) and the others are rich flows. If a poor flow C_i ($0 \leq i \leq m$) tags AvgTkt = $\mathsf{InTktRate}_p$ / $\mathsf{InPktRate}_p$ tickets onto each of its packets, according to the ticket tagging algorithm, the throughput of C_i should be $\mathsf{InPktRate}_p \times \mathsf{OutTktRate}_i$ / $\mathsf{InTktRate}_p$ p/s, which means C_i obtains its fair share of bandwidth. Thus, if we can ensure that each poor flow tags approximately the same number of AvgTkt tickets onto its packets, we will achieve a fair bandwidth allocation among them. At the same time, rich flows should obtain the bandwidth they require. Actually, rich flows will tag more than AvgTkt tickets onto their packets because they do not make full use of their fair share of bandwidth; this is why we call them rich flows.

2.4.1 RED algorithm

Our packet scheduling algorithm is based on the random early detection (RED) algorithm [3]. RED keeps the overall throughput high while maintaining a small average queue length, and tolerates transient congestions. RED operates as follows. When the queue length exceeds a certain threshold, it drops incoming packets with some probability. The packet loss causes the affected TCP flows to slow down. 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. Persistent congestion in the queue is reflected by a high average queue length and results in a high drop probability.
Upon each packet arrival:

```
AvgQLen = (1–wght) × AvgQLen
        + wght × CurrentQLen;
```

if AvgQLen $\leq$ MinThresh
    enqueue the packet
if MinThresh $<$ AvgQLen $<$ MaxThresh
    calculate probability p
    drop the packet with probability p
if MaxThresh $\leq$ AvgQLen
    drop the arriving packet

RED has three operating phases, which correspond to the AvgQLen being in the range of [0, MinThresh], (MinThresh, MaxThresh) and [MaxThresh, ∞). The three phases represent normal operation, congestion avoidance, and congestion control, respectively. During normal operation, RED doesn't drop any packets. During the congestion avoidance phase, each packet drop serves to notify the TCP source to reduce its sending rate. During the congestion control phase, all incoming packets are dropped.

2.4.2 Ticket-Based RED (TRED)

Ticket-based RED (TRED) is a modification of RED. Like RED, TRED also has three phases: normal operation, congestion avoidance, and congestion control. TRED 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 poor TCP 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 [1], 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, the more tickets a packet carries, the less the probability that it is dropped. Since packets that belong to rich flows are likely to carry more tickets than packets belonging to poor flows, their packets are less likely to be dropped, and hence the poor flows are more likely to back off during congestion. This adjustment reflects our intention to preserve the bandwidth obtained by rich flows. Amongst poor flows, those that put fewer tickets on their packets are more likely to back off than those that tag more tickets to their packets. When a flow backs off, its sending rate and instantaneous throughput slow down and it will begin to tag more tickets onto its packets. In the end, we expect all poor flows to tag approximately the same number of tickets on their packets; we call this number AvgTkt', but we cannot know it exactly. Note that although AvgTkt' does not equal ExpectTkt, a greater ExpectTkt does reflect a greater AvgTkt', and vice versa.

In the above algorithm, we multiply the drop probability p by a cubic function of (ExpectTkt / InTkt). This function represents a tradeoff between maintaining high link utilization and achieving fast convergence. If we use a function with higher rank, the number of tickets on the packets of poor flows will converge to AvgTkt' faster, but the link utilization will be lower because we drop packets with less than ExpectTkt tickets more aggressively in the congestion avoidance phase. The algorithm also ensures that the number of tickets on most packets of poor flows will be in the range of [MinTkt, $K\times$ MinTkt] for $K \geq 1$. In our experiments, the number of tickets on more than 98% of the packets of poor flows is in this range if we set $K = 1.4$.

The next issue is how to calculate ExpectTkt, where our goal is to keep ExpectTkt close to AvgTkt. This is because the packets of rich flows will be dropped with a relative low probability since they carry more than AvgTkt tickets. At the same time, if a poor flow tags much fewer tickets onto its packets than another poor flow, the packet drop probability of the former will be much higher than that of the latter. However, both probabilities are less than 1. We

calculate ExpectTkt as the average number of tickets on those packets that fall in the range of [MinTkt, $K\times$ MinTkt]. As AvgTkt equals to InTktRate_p / InPktRate_p, most packets of poor flows will be in this range. Although it is possible that some packets of rich flows will also fall in this range, ExpectTkt is still in the range of [AvgTkt, $K\times$ AvgTkt].

As we said in the beginning of Section 2.4, we optimally expect each poor flow to tag AvgTkt = InTktRate_p / InPktRate_p tickets onto its packets, but in reality, we cannot differentiate between poor flows and rich flows, meaning we cannot precisely calculate AvgTkt. However, the algorithm tries to keep AvgTkt' around AvgTkt by adjusting ExpectTkt. When AvgTkt' is less than AvgTkt, which means poor flows obtain more than their fair share of bandwidth, AvgQLen will increase accordingly. This causes the poor flows to slow down, and tag more tickets onto their future packets. As a result, MinTkt and ExpectTkt will increase together with AvgTkt'. On the other hand, when AvgTkt' is greater than AvgTkt, the algorithm will cause it to decrease. In the end, the algorithm tries to keep AvgTkt' close to AvgTkt, so as to fairly allocate bandwidth among poor flows.

2.5 Tag Relabeling

The previous sections consider only TCP flows with one hop. In real networks, a flow may go through many hops before reaching the destination. To allocate fair share of bandwidth among multi-hop flows, we need to relabel the tags on packets at each hop. 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:

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

As before, InTktRate corresponds to the active t/s the link receives at some instant, and it is computed with the TSW algorithm. The OutTktRate is the t/s issued to the link by its controlling entity. For each packet, we relabel its tag as follows:

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

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

2.6 Fairness for Multi-Hop Flows

Suppose a TCP flow C originates from source A, which has been issued 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 poor flows of link (S, T) are InTktRate_p and InPktRate_p. We say flow C obtains its fair share of bandwidth if:

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

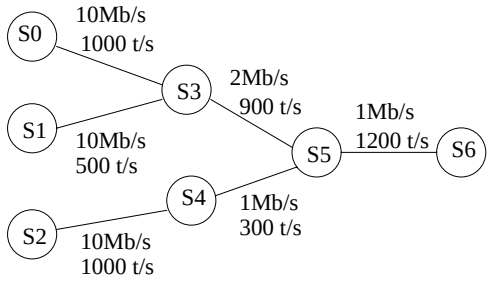

Figure 2: Multi-Hop Example

In Figure 2 for example, 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. Links (S_0, S_3) and (S_1, S_3) receive 1000 and 500 t/s from S_3, respectively; link (S_2, S_4) receives 1000 t/s from S_4; and links (S_3, S_5) and (S_4, S_5) receive 900 and 300 t/s from S_5, respectively. The bandwidth of link (S_5, S_6) is 1Mb/s. Tickets of flow A, B and C are converted to 600, 300 and 300 t/s, respectively, in S_5's currency. When all three flows are poor, A, B and C should obtain 0.5Mb/s, 0.25Mb/s and 0.25Mb/s bandwidth respectively. When B has a constant bit rate of 0.16Mb/s, which is less than its fair 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. When B is silent, A and C should divide the full bandwidth of link (S_5, S_6) at the ratio of 2:1.

We now explain how the algorithm ensures that flow C obtains its fair share of bandwidth. Since link (S, T) is a bottleneck for C, C is a poor flow of the link. From the TRED algorithm, we know that each packet of C will carry approximately AvgTkt tickets when passing through link(S, T), so the bandwidth that C obtains on link (S, T) is approximately InTktRate_c / AvgTkt. Because link (S, T) is the bottleneck of

C, AvgRate, the throughput of C should equal to the bandwidth that C obtains on link (S, T). Thus, we get:

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

This is what we are trying to achieve. Thus, C will obtain its fair share of bandwidth.

2.7 Receiver-Based Algorithm

In the sender-based algorithm, each entity defines its own currency (S-currency) and issues some t/s in S-currency to its input links or TCP sources. In the receiver-based algorithm, each entity defines its own currency (R-currency) and issues some t/s in R-currency to its output links or TCP sinks. The idea behind the receiver-based algorithm 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 can simply run the sender-based algorithm to achieve fair bandwidth allocation among TCP flows.

2.7.1 ACK Packet Tagging and Relabeling

In the sender-based algorithm, only data packets are tagged by the TCP sources and then relabeled at each hop. In the receiver-based algorithm, data packets are still tagged and relabeled, but 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 when congestions occurs, while the latter is used to calculate how many t/s in S-currency an entity should issue 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, using the TSW algorithm. 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, using the TSW algorithm. 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 issues 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 issue to its input links or TCP sources as follows:

$$\text{OutTktRate} = \text{AckInTktRate}$$

This means the ticket rate a link or an agent 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:

$$\text{InTktRate} = \text{AckOutTktRate}$$

This means the ticket rate a link or a 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,

$$\begin{aligned} &\text{XRate} \\ &= \text{OutTktRate} \,/\, \text{InTktRate} \\ &= \text{AckInTktRate} \,/\, \text{AckOutTktRate} \\ &= 1 \,/\, \text{AckXRate} \end{aligned}$$

For any flow C that passes through link L, we use AckInTktRate_c and InTktRate_c to represent the t/s L receives from ACK packets and data packets of C. We use AckOutTktRate_c and OutTktRate_c to represent the t/s L tags onto ACK packets and data packets of C. We can further show that:

$$\begin{aligned} \text{OutTktRate}_c &= \text{AckInTktRate}_c \\ \text{InTktRate}_c &= \text{AckOutTktRate}_c \end{aligned}$$

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

2.7.2 Fairness

The fairness definition for receiver-based algorithm is similar to that for sender-based algorithm. Suppose a TCP flow C goes to 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 ACK exchange rates 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_p t/s from the ACK packets of all poor flows. The throughput of all poor flows is InPktRate_p p/s. We say C obtains its fair share of bandwidth, if they satisfy:

$$\text{AvgRate} = \text{InPktRate}_p \times \text{AckInTktRate}_c / \text{AckInTktRate}_p$$

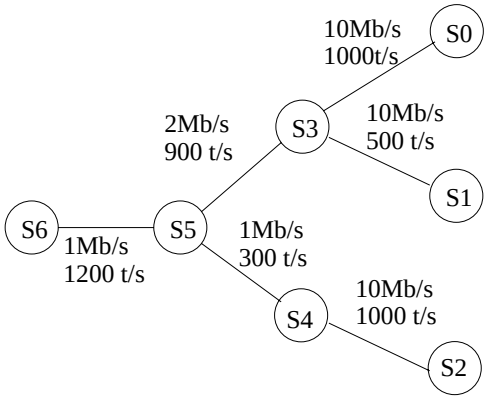

Figure 3: Example of Receiver-Based Algorithm

For example, in Figure 3 there are three TCP flows: A between S_6 and S_0, B between S_6 and S_1, and C between S_6 and S_2. Links (S_3, S_0) and (S_3, S_1) receive 1000 and 500 t/s from S_3, respectively; link (S_4, S_2) receives 1000 t/s from S_4; links (S_5, S_3) and (S_5, S_4) receive 900 and 300 t/s from S_5, respectively. The bandwidth of link (S_6, S_5) is 1Mb/s. Tickets of flow A, B and C are converted to 600, 300 and 300 t/s, respectively, in S_5's R-currency. When all three flows are poor flows, A, B, and C should obtain 0.5Mb/s, 0.25Mb/s and 0.25Mb/s bandwidth, respectively. When B has a constant bit rate of 0.16Mb/s, which is less than its fair 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. When B is silent, A and C should divide the full bandwidth of link (S_6, S_5) at the ratio of 2:1.

We now explain why our receiver-based algorithm can fairly allocate bandwidth for flow C. Suppose link (S, T) receives InTktRate_c and InTktRate_p t/s from data packets of flow C and all poor flows respectively, and the ACK exchange rate of this link is AckXRate. From the sender-based algorithm:

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

But for link (S, T):

$$\text{InTktRate}_c = \text{AckOutTktRate}_c$$
$$\text{InTktRate}_p = \text{AckOutTktRate}_p$$
$$\text{AckOutTktRate}_c = \text{AckInTktRate}_c \times \text{AckXRate}$$
$$\text{AckOutTktRate}_p = \text{AckInTktRate}_p \times \text{AckXRate}$$

Combine all of the above, we get:

$$\text{AvgRate} = \text{InPktRate}_p \times \text{AckInTktRate}_c / \text{AckInTktRate}_p$$

So our receiver-based algorithm will fairly allocate bandwidth for any TCP flow.

2.8 Ticket Policing

As discussed in Section 2.3 and 2.5, 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 sending rate, ActualTktRate, with the TSW algorithm, and then adjust the amount of tickets that are tagged to the packet as follows:

At source:

$$\text{OutTkt} = \text{OutTktRate} / \text{AvgRate} \times \text{OutTktRate} / \text{ActualTktRate}$$

At links:

$$\text{OutTkt} = \text{InTkt} \times \text{XRate} \times \text{OutTktRate} / \text{ActualTktRate}$$

Without such adjustment, a source or link may have a higher or lower ticket sending rate than its allocated rate.

3 Simulation Results

This section reports the results of several simulations designed to evaluate our probabilistic packet scheduling algorithm's ability to fairly allocate bandwidth among TCP flows. We use the NS network simulator for our simulations [7]. We conducted each of the following experiments for both the sender-based and receiver-based algorithm, although we show the results for the only sender-based algorithm. There were no qualitative differences between the results of sender-based and receiver-based algorithms. In all experiments, the WinLength parameter used in the TSW algorithm is set to 60 seconds [1]. To amortize the instability during initialization, all experiments run for 600 seconds of simulated time.

3.1 One-Hop Configuration

Our first experiment measures how fairly our algorithm allocates bandwidth when there is a single congested link. We use the configuration shown in Figure 4, where nine TCP flows share a 1.5Mb/s bottleneck link. The flows are assigned an incremental number of t/s, ranging from 100 to 900. 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 5, the achieved throughput is proportional to the number of t/s given to each flow.

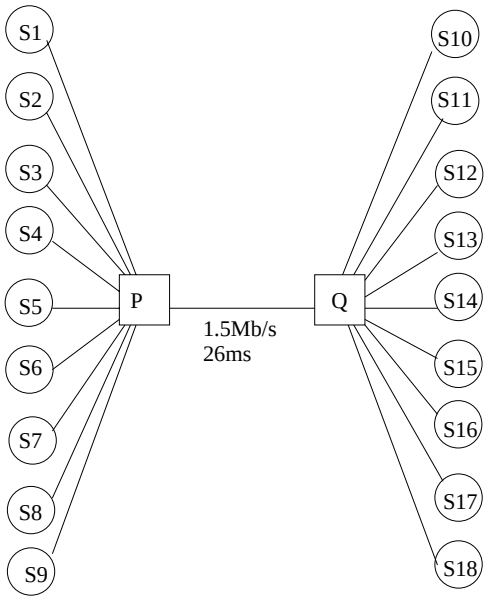

Figure 4: Configuration for one-hop simulation.

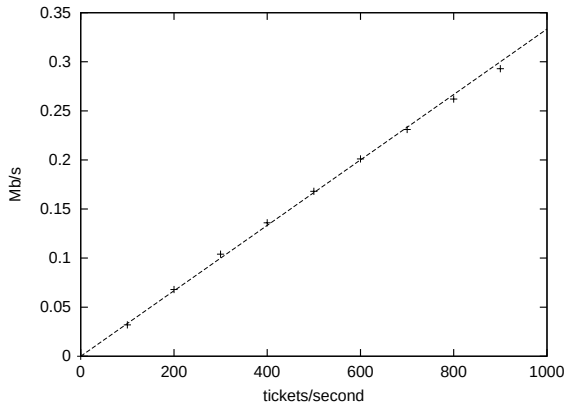

Figure 5: Bandwidth allocation for one-hop configuration. The x and y coordinate of each point represent the t/s issued to the flow and measured bandwidth of the flow.

3.2 Multi-Hop Configuration

We next study the bandwidth allocation among multi-hop TCP flows. Figure 6 shows the network configuration we tested. It has 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. Links (S_0, S_3) and (S_1, S_3) receive 1000 and 500 t/s from S_3; link (S_2, S_4) receives 1000 t/s from S_4; and links (S_3, S_5) and (S_4, S_5) receive 900 and 300 t/s, respectively, from S_5. All three flows share the bottleneck 1Mb/s bandwdith of link (S_5, S_6). Tickets of flow A, B and C are converted to 600, 300 and 300 t/s, respectively, in S_5's currency. According to our fairness definition, A, B and C should obtain 0.5Mb/s, 0.25Mb/s and 0.25Mb/s of bandwidth, respectively, and as shown in the middle column of Table 1, they do. As the table also shows that the three flows were able to consume 98% of the bottleneck link's capacity.

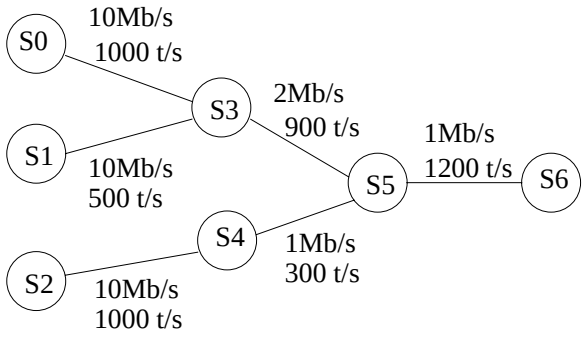

Figure 6: Configuration for multi-hop simulation.

Flow	Measured Rate (Mb/s)	Expected Rate (Mb/s)
A	0.48	0.50
B	0.25	0.25
C	0.25	0.25
Total	0.98	1.00

Table 1: Bandwidth allocation for multi-hop configuration.

3.3 Variable Traffic

This experiment evaluates how well the algorithm adjusts to variations in the source sending rate. We use the same topology as in Figure 6, but when the simulation begins, only flow B and C are active; flow A becomes active after 300 seconds. The results are shown in Figure 7, where the x-axis is time and the y-axis is instantaneous throughput. 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 300 seconds, but after A starts up, A, B and C quickly converge to 0.5Mb/s, 0.25Mb/s and 0.25Mb/s, respectively.

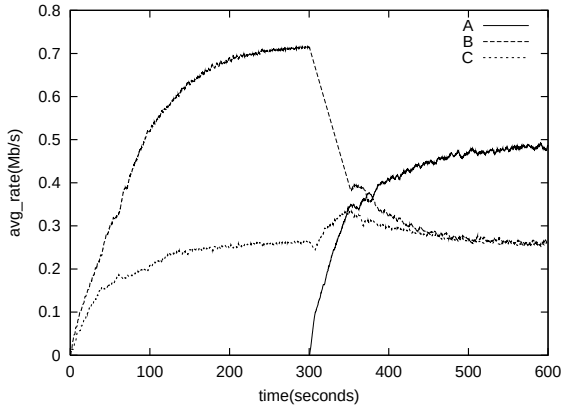

Figure 7: Adapting to new traffic.

3.4 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 share in effort to maintain a certain transmission speed. To see this, we again use the topology in Figure 6, where in the beginning, each link is issued the same amount of tickets as in Section 3.2, but after 300 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. By our fairness criteria, we expect the instantaneous throughput of A to change from 0.5Mb/s to 0.3Mb/s, and the instantaneous 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 8, the system behaves as expected. The important point is that our algorithm insulates bandwidth allocation decisions from one another—that is, the variation of A and B has virtually no influence on C.

3.5 Fairly Sharing Unused Capacity

Sometimes a flow cannot make full use of its fair share of bandwidth because the application generates bytes at a lower rate. The unused bandwidth should be fairly 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 fair way, we again

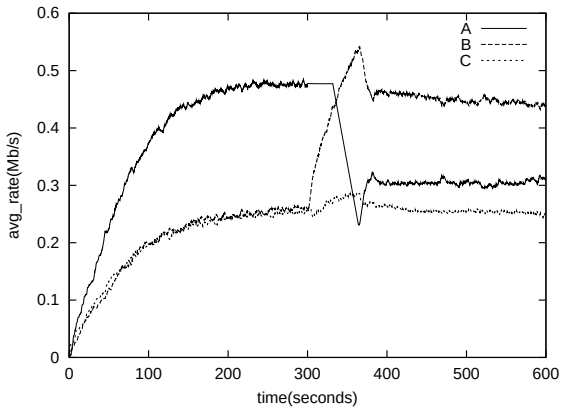

Figure 8: Bandwidth allocation as ticket rates change.

use the topology from Figure 6, but this time the traffic from flow B is generated by an application that transmits at a fixed rate of 0.16Mb/s, less than its fair share of 0.25Mb/s. Flow A and C should divide the remaining 0.84Mb/s bandwidth at the ratio of 2:1 since their local tickets are converted to 600 and 300 t/s in S_5's currency. As shown in Table 2, this is exactly what happens.

Flow	Measured Rate (Mb/s)	Expected Rate (Mb/s)
A	0.53	0.56
B	0.16	0.16
C	0.29	0.28
Total	0.98	1.00

Table 2: Fair sharing of unused capacity.

3.6 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 our algorithm can fairly allocate bandwidth when some flows have different bottlenecks and routers have multiple output links. We ran a series of experiments using the topology given in Figure 9. In this scenario, there are three TCP flows—A, B and C—running between (S_0, S_8), (S_1, S_7) and (S_2, S_8), respectively. Links (S_0, S_3) and (S_1, S_3) are issued 1000 and 500 t/s by S_3; link (S_2, S_4) is issued 1000 t/s by S_4; links (S_3, S_5) and (S_4, S_5) are issued 900 and 300 t/s by S_5; and link(S_5, S_6) is issued 1200 t/s by S_6.

In the first experiment, we assume the bandwidth of link (S_6, S_7) is 1Mb/s and the bandwidth of link (S_6,

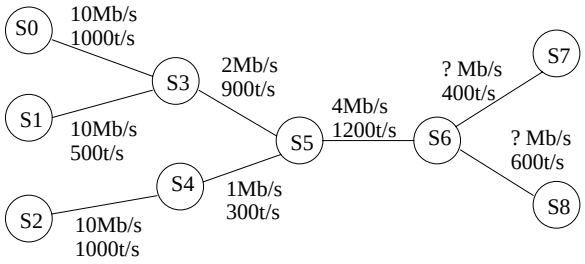

Figure 9: Complex configuration with multiple bottleneck links.

S_8) is 10Mb/s. The bottleneck of flows A and B is link (S_3, S_5), while the bottleneck of flow C is link (S_4, S_5). By our fairness definition, flow A and B should divide the 2Mb/s bandwidth of link (S_3, S_5) at the ratio of 2:1, and flow C should obtain all the bandwidth of link (S_4, S_5). The actual results are shown in Table 3.

Flow	Measured Rate (Mb/s)	Expected Rate (Mb/s)
A	1.24	1.33
B	0.70	0.67
C	0.98	1.00

Table 3: Multiple Bottleneck Links: Scenario I.

In the second experiment, we set the bandwidth of link (S_6, S_7) to 10Mb/s and the bandwidth of link (S_6, S_8) to 1Mb/s. Now, the bottleneck of flows A and C is link (S_6, S_8), while the bottleneck of flow B is link (S_3, S_5). The 1000 t/s of links (S_0, S_3) and (S_2, S_4) are converted to 600 and 300 t/s in S_6's currency. By our fairness criteria, flow A and C should obtain 0.67Mb/s and 0.33Mb/s bandwidth from link (S_6, S_8), respectively. Moreover, since A is a rich flow of link (S_3, S_5) and B is a poor flow, B should obtain the remaining 1.33Mb/s bandwidth of link (S_3, S_5). The measured results are shown in the middle column of Table 4.

Flow	Measured Rate (Mb/s)	Expected Rate (Mb/s)
A	0.63	0.67
B	1.28	1.33
C	0.35	0.33

Table 4: Multiple Bottleneck Links: Scenario II.

Note that although link (S_0, S_3) is issued twice as many t/s as link (S_1, S_3), the actual bandwidth A obtains is less than that of B. This may seems 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 link (S_6, S_8), so it cannot make full use of its fair share of bandwidth at link (S_3, S_5) 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 issued ticket rate and its bottleneck.

In a final experiment, we set the bandwidth of links (S_6, S_7) and (S_6, S_8) to 1Mb/s. This means that the bottleneck of flow A and C is link (S_6, S_8) and the bottleneck of flow B is link (S_6, S_7). By our fairness definition, flow A and C should obtain 0.67Mb/s and 0.33Mb/s bandwidth from link (S_6, S_8), and flow B should obtain the full bandwidth of link (S_6, S_7). These results are confirmed in Table 5.

Flow	Measured Rate (Mb/s)	Expected Rate (Mb/s)
A	0.64	0.67
B	0.98	1.00
C	0.34	0.33

Table 5: Multiple Bottleneck Links: Scenario III.

3.7 RTT Biases

It is well-known that TCP has a bias against flows with large round trip time. To understand the relationship between our algorithm and RTT, we experimented with two different configurations.

The first configuration is depicted in Figure 10, where ten flows share a bottleneck of 1.0Mb/s and all input links are all issued 200 t/s. Given this configuration, we first set all the RTTs to 30ms. (This setting also serves to demonstrate that our algorithm is able to finely split bandwidth among many competing flows.) We then let the RTT of the flows vary, incrementally, from 30ms to 300ms. As can be seen from Table 6, there is only a slight bias against long-RTT flows; each flow gets close to one-tenth of the available capacity. This is because when a flow with large RTT backs off, it tags more tickets onto each of its packets. When contending with other flows, the large-RTT packets are more likely to get through and the flow recovers to its fair share of bandwidth faster than standard TCP.

The second configuration is depicted in Figure 4. In this case, we first set the RTT of flow 1, which is issued 100 t/s, to 300ms, and the RTT of all other flows to 30ms. We then reset the experiment so that the RTT of flow 9, which is issued 900 t/s, is 300ms and the

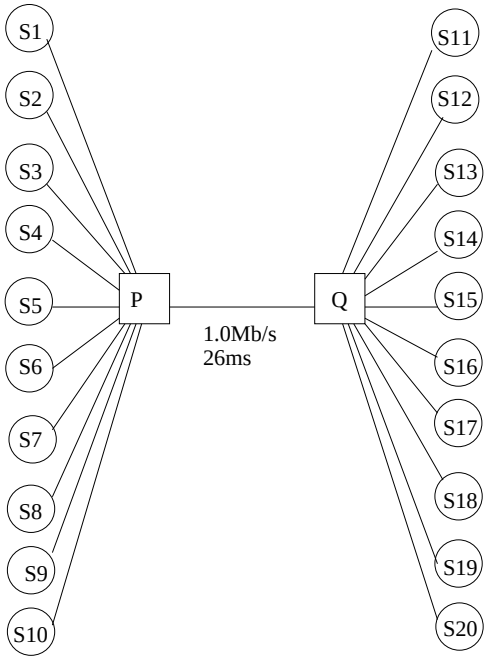

Figure 10: Configuration used for RTT experiments: Scenario I.

RTT of all other flows is 30ms. As we can see from the results shown in Figure 11 and Figure 12, a large RTT has more negative influence on flow 9 than flow 1. This is because the fair share of bandwidth of flow 9 is greater than that of flow 1. When both flows back off, flow 9 loses more bandwidth than flow 1, so it takes longer for flow 9 to recover to its fair share of bandwidth than for flow 1.

RTT (ms)	Measured Rate (Mb/s)	RTT (ms)	Measured Rate (Mb/s)
30	0.100	300	0.093
30	0.099	270	0.093
30	0.100	240	0.095
30	0.099	210	0.095
30	0.100	180	0.097
30	0.100	150	0.099
30	0.100	120	0.099
30	0.099	90	0.101
30	0.100	60	0.102
30	0.100	30	0.103
Total	0.997	Total	0.977

Table 6: Variable RTT: Scenario I.

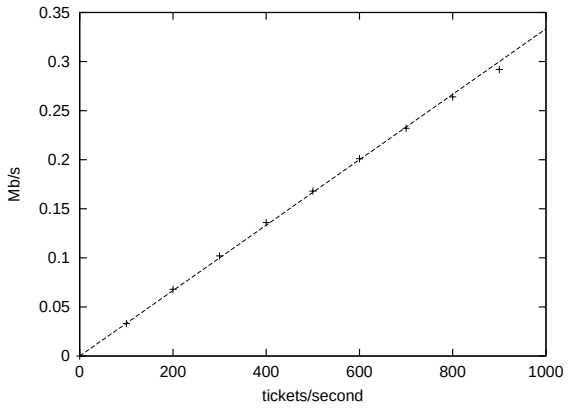

Figure 11: Variable RTT: Scenario II. RTT of flow 1 is 300ms, RTT of all other flows is 30ms

3.8 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 [1]. 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 contending for some link in the middle of the network, or those links that were expect to be shared are temporarily idle. It is not possible to guarantee the link capacity matches the total target profile rate of contending flows at any time at any place in the network.

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 13, which has 3 flows (A, B and C) contending for link (P, Q) with a bandwidth of 1.2Mb/s. For each scenario, our algorithm allocates bandwidth in proportion to the service profiles of the three flows, independent of the available capacity. In the tables that follow, the second column gives the service profile used by DiffServ, the third column gives the measured rate achieved by each flow using DiffServ, and the fourth column gives the measured rate achieved by the PPS algorithm. The tables do not show the actual ticket assignment used by PPS, but they were at the same ratio (3:2:1) as the service profiles.

1. The target sending rate (service profile) of A, B and C are 0.6Mb/s, 0.4Mb/s and 0.2Mb/s, respectively, which matches the 1.2Mb/s capacity of the shared link. As we can see from Table 7, Diff-

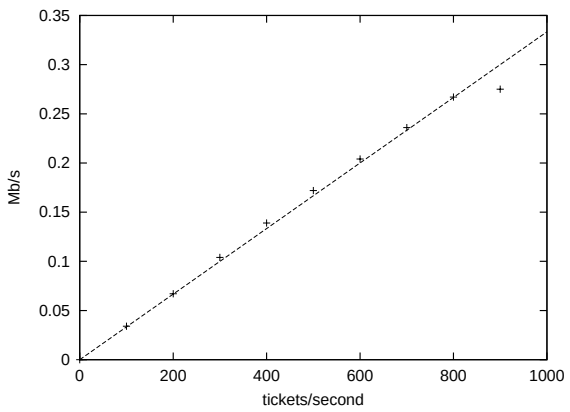

Figure 12: Variable RTT: Scenario II. RTT of flow 9 is 300ms, RTT of all other flows is 30ms

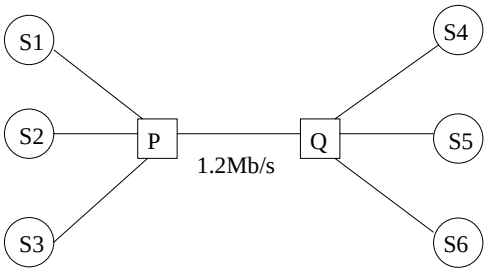

Figure 13: Configuration for comparing PPS with DiffServ.

Serv and our algorithm work equally well when the link capacity matches the service profiles.

2. The target rate of A, B and C are 0.15Mb/s, 0.1Mb/s and 0.05Mb/s, respectively, which is far below the available 1.2Mb/s link capacity. As we can see from Table 8, DiffServ allocates the bandwidth that exceeds the total expected bandwidth arbitrarily, while PPS allocates the bandwidth in proportion to the t/s assigned to each flow.

3. The expected sending rate of A, B and C are 1.2Mb/s, 0.8Mb/s and 0.4Mb/s, respectively, which exceeds the capacity available on link (P, Q). As a consequence, many in-profile packets are dropped, causing DiffServ to degenerate to best effort. The results are shown in Table 9.

3.9 Ticket Bits

We use 8-bit tags in all the experiments reported in this section, which means 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. There are two important points to make, however. First, the number of bits needed is independent of the number of hops across the network. Tickets are relevant on only a single router at a time. Hence, we are not concerned that more complex topologies will require more bits. Second, the number of bits needed is dependent on the levels of service one wants to differentiate among on a given router. It is independent of the number of flows one is trying push through the router.

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 [11]. Our simulations suggest that this is more than enough for our approach.

Flow	Service Profile	Measured DiffServ	Measured PPS
A	0.60	0.59	0.58
B	0.40	0.40	0.40
C	0.20	0.21	0.21
Total	1.20	1.20	1.19

Table 7: PPS versus DiffServ: Profile matches capacity (Mb/s).

Flow	Service Profile	Measured DiffServ	Measured PPS
A	0.15	0.43	0.58
B	0.10	0.42	0.40
C	0.05	0.35	0.21
Total	0.30	1.20	1.19

Table 8: PPS versus DiffServ: Profile less than capacity (Mb/s).

4 Non-Responsive Flows

Tickets represent the share of network bandwidth a flow should receive. When congestion occurs, those packets without enough tickets are likely to be dropped. When an adaptive protocol like TCP detects such drops, it sends fewer packets—placing more tickets on each packet—thereby increasing the likelihood of its packets being delivered. The question is what to do about non-adaptive flows, such as those managed by UDP. One option is nothing. Such flows

Flow	Service Profile	Measured DiffServ	Measured PPS
A	1.20	0.41	0.58
B	0.80	0.42	0.40
C	0.40	0.37	0.21
Total	2.40	1.20	1.19

Table 9: PPS versus DiffServ: Profile greater than capacity (Mb/s).

would effectively be penalized for sending at too fast a rate since the flows' packets would have insufficient tickets to make them through a congested link. This would effectively force all flows to be adaptive, which is arguably a good thing.

An alternative is for routers to harvest the tickets on any packets they are about to drop, and distribute these tickets to other needy packets, thereby improving their odds of being delivered. Clearly, this cannot be done on a per-flow basis without suffering the same scalability problems as IntServ, but it is possible to recover lost tickets on a router-wide basis.

To simplify the discussion, we assume only the sender-based scheme for one-hop UDP flows; the approach can be extended to multi-hop scenarios and the receiver-based approach. The tagging and relabeling mechanisms of UDP flows are similar to those of TCP. We measure the sending rate, AvgRate, at the UDP source, tag OutTktRate / AvgRate tickets on each outgoing packet, and relabel each packet based on some currency exchange rate when it passes through a link. We then modify the packet scheduling algorithm as follows:

Upon each packet arrival:

compute AvgQLen *the same as in RED*
compute InTktRate *the same as in TRED*
if AvgQLen $\leq$ MinThresh
    enqueue the packet
if MinThresh $<$ AvgQLen $<$ MaxThresh
    AvgTkt = InTktRate / Capacity ;
    $p = max(0, 1 -$ InTkt / AvgTkt$)$;
    if $p > 0$
        augment the tickets on the packet from InTkt *to* AvgTkt;
    drop the packet with probability p
if MaxThresh $\leq$ AvgQLen
    drop the arriving packet

In the above pseudo-code, Capacity stands for the link capacity and AvgTkt represents the number of tickets the link expects for each incoming packet. If all incoming packets carry AvgTkt tickets, the total incoming packet rate will be InTktRate / AvgTkt = Capacity. Now, suppose UDP source A is issued OutTktRate_a t/s from its controlling entity P and the sending rate of A is AvgRate p/s. The output link capacity is Capacity p/s and the link receives InTktRate t/s from P. When congestion happens:

$$\begin{aligned}&\text{InTkt} = \text{OutTktRate}_a \text{ / AvgRate}\\ &\text{AvgTkt} = \text{InTktRate / Capacity}\\ &p = max(0, 1 - \text{InTkt / AvgTkt})\\ &\quad = max(0, 1 - \text{OutTktRate}_a \times\\ &\qquad \text{Capacity / AvgRate / InTktRate})\end{aligned}$$

If AvgRate / Capacity $\leq$ OutTktRate_a / InTktRate, which means the sending rate of A is less than or equals to its fair share of bandwidth, then $p = 0$. We'll keep the packet of A. If AvgRate / Capacity $>$ OutTktRate_a / InTktRate, which means sending rate of A is greater than its fair share, then $0 < p < 1$. The actual bandwidth A obtains is:

$$\begin{aligned}&(1-p) \times \text{AvgRate}\\ &= \text{OutTktRate}_a \times \text{Capacity / AvgRate}\\ &\qquad \text{/ InTktRate} \times \text{AvgRate}\\ &= \text{Capacity} \times \text{OutTktRate}_a \text{ / InTktRate}\end{aligned}$$

This is exactly the fair share of bandwidth that A should obtain. Note that if a packet should be dropped with probability p $(0 < p < 1)$ but not dropped, its tickets are augmented to AvgTkt. The reason is that only $(1-p)$ fraction of the total packets from A can survive, which means p fraction of its tickets are lost. We want to keep the total tickets of A unchanged after passing through the link. So we augment the tickets on those survived packets $1/(1-p)$ times. As a result, the total t/s on those survived packets of A is:

$$\begin{aligned}&\text{AvgTkt} \times (1-p) \times \text{AvgRate}\\ &= \text{InTktRate / Capacity} \times \text{Capacity}\\ &\qquad \times \text{OutTktRate}_a \text{ / InTktRate}\\ &= \text{OutTktRate}_a\end{aligned}$$

So at one hop, a UDP flow obtains its fair share of bandwidth, and at the same time, keeps the same total amount of t/s.

Note that this idea is similar to that of SCORE [9], in which each flow tags its packets with its sending rate. When congestions happen, the packet is dropped with some probability p, which is determined by the estimated fair share rate and the sending rate of the flow.

5 Related Work

As described in the Introduction, PPS can be understood relative to WFQ/IntServ, DiffServ, and SCORE. The additional point revealed by the simulations is that PPS is much more effective than DiffServ when profiles are less or more than the available link capacity, which is likely to happen somewhere in the network, especially at the boundary between DiffServ domains.

Also as mentioned in the Introduction, PPS was directly motivated by lottery scheduling [12], 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 it the bandwidth obtained by a flow is proportional to the relative share of ticket rate that it is issued. Also there need not be a single currency—each network is free to adopt its own currency (the boundary routers merely need to implement the exchange rate). The probabilistic strategy eliminates the need for per-flow state, yet fairly allocates bandwidth among competing flows.

Turning to other similar schemes, Gupta proposed a priority scheduling of packets belonging to different users [4]. Higher priority packets always depart the routers first. This scheme may result in the starvation of lower priority users. Compared with their scheme, PPS ensures the users with less allocated tickets rate always obtain their fair share of bandwidth.

MacKie-Mason and Varian propose a smart market mechanism that attaches a bid to each packet for each hop [5]. This bid is used to determine the packet's chance of being forwarded. But there is no obvious relationship among the hop-by-hop bids. Our algorithm puts tickets onto packets at end hosts and the tag is relabeled at each hop according to the exchange rates. When a packet carries less tickets than those required by the link and gets dropped, the corresponding sender will adapt and put more tickets onto future packets.

6 Conclusions

This paper describes probabilistic packet scheduling algorithm (PPS) for achieving fair bandwidth allocations among TCP flows. We use tickets to represent a relative share of bandwidth that a flow should receive. Packets are tagged with tickets at TCP sources, and then relabeled at each hop according to an exchange rate calculated from the current aggregate traffic. When congestion occurs, PPS calculates the drop probability of a packet based on the congestion level and the tickets the packet carries. We give a definition of fair bandwidth allocation and demonstrate how the algorithm can ensure such fairness for TCP flows. Both sender-based and receiver-based algorithms have been developed. The main advantages of the algorithm are its scalability and simplicity.

References

[1] D. Clark and W. Fang. Explicit allocation of best-effort packet delivery service. *IEEE/ACM Transactions on Networking*, 6(4):362–373, Aug. 1998.

[2] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *Proceedings of ACM SIGCOMM*, pages 362–373, Aug. 1989.

[3] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, July 1993.

[4] A. Gupta, D. Stahl, and A. Whinston. Priority pricing of integrated services networks. *Internet Economics, L. McKnight and J. Bailey (eds.)*, pages 253–279, 1997.

[5] J. MacKie-Mason and H. Varian. Economic faq's about the internet. *Internet Economics, L. McKnight and J. Bailey (eds.)*, pages 27–63, 1997.

[6] K. Nichols, V. Jacobson, and L. Zhang. An approach to service allocation in the internet. *Internet Draft*, Nov. 1997.

[7] ns 2 (online). http://www.isi.edu/nsnam/ns.

[8] S. Shenker, R. Braden, and D. Clark. Integrated services in the internet architecture: an overview. *Internet RFC 1633*, 1994.

[9] I. Stoica, S. Shenker, and H. Zhang. Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks. *Proceedings of SIGCOMM*, pages 118–130, Aug. 1998.

[10] I. Stoica and H. Zhang. Lira: An approach for service differentiation in internet. *Proceedings of NOSSDAV*, July 1998.

[11] I. Stoica and H. Zhang. Providing guaranteed services without per flow management. *Proceedings of SIGCOMM*, pages 81–94, Aug. 1999.

[12] C. Waldspurger and W. Weihl. Lottery scheduling: Flexible proportional-share resource management. *Proceedings of OSDI*, pages 1–12, Nov. 1994.

[13] Z. Wang. User-share differentiation (usd) scalable bandwidth allocation for differentiated services. *Internet Draft*, May 1998.