CHOKe
A stateless active queue management scheme for approximating fair bandwidth allocation

Rong Pan, Balaji Prabhakar*, Konstantinos Psounis
Department of Electrical Engineering
*Department of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305
{rong,balaji,kpsounis}@leland.stanford.edu

Abstract—**We investigate the problem of providing a fair bandwidth allocation to each of n flows that share the outgoing link of a congested router. The buffer at the outgoing link is a simple FIFO, shared by packets belonging to the n flows. We devise a simple packet dropping scheme, called CHOKe, that discriminates against the flows which submit more packets/sec than is allowed by their fair share. By doing this, the scheme aims to approximate the fair queueing policy. Since it is stateless and easy to implement, CHOKe controls unresponsive or misbehaving flows with a minimum overhead[1].**

Keywords—**fair queueing, RED, active queue management, scheduling algorithm.**

I. Introduction

The Internet provides a connectionless, best effort, end-to-end packet service using the IP protocol. It depends on congestion avoidance mechanisms implemented in the transport layer protocols, like TCP, to provide good service under heavy load. However, a lot of TCP implementations do not include the congestion avoidance mechanism either deliberately or by accident. Moreover, there are a growing number of UDP-based applications running in the Internet, such as packet voice and packet video. The flows of these applications do not back off properly when they receive congestion indications. As a result, they aggressively use up more bandwidth than other TCP compatible flows. This could eventually cause "Internet Meltdown" [2]. Therefore, it is necessary to have router mechanisms to shield responsive flows from unresponsive or aggressive flows and to provide a good quality of service (QoS) to all users.

As discussed in [2], there are two types of router algorithms for achieving congestion control, broadly classified under the monikers "scheduling algorithms" and "queue management algorithms". The generic scheduling algorithm, exemplified by the well-known Fair Queueing (FQ) algorithm, requires the buffer at each output of a router to be partitioned into separate queues each of which will buffer the packets of one of the flows [3]. Packets from the flow buffers are placed on the outgoing line by a scheduler using an approximate bit-by-bit, round-robin discipline. Because of per flow queueing, packets belonging to different flows are essentially isolated from each other and one flow cannot degrade the quality of another. However, it is well-known that this approach requires complicated per flow state information, making it too expensive to be widely deployed.

To reduce the cost of maintaining flow state information, Stoica et al [16] have recently proposed a scheduling algorithm called Core Stateless Fair Queueing (CSFQ). In this method routers are divided into two categories: edge routers and core routers. An edge router keeps per flow state information and estimates each flow's arrival rate. These estimates are inserted into the packet headers and passed on to the core routers. A core router simply maintains a stateless FIFO queue and, during periods of congestion, drops a packet randomly based on the rate estimates. This scheme reduces the core router's design complexity. However, the edge router's design is still complicated. Also, because of the rate information in the header, the core routers have to extract packet information differently from traditional routers. Another notable scheme which aims to approximate FQ at a smaller implementation cost is Stochastic Fair Queueing (SFQ) proposed by McKenny [13]. SFQ classifies packets into a smaller number of queues than FQ using a hash function. Although this reduces FQ's design complexity, SFQ still requires around 1000 to 2000 queues in a typical router to approach FQ's performance [12].

Thus, scheduling algorithms can provide a fair bandwidth allocation, but they are often too complex for high-speed implementations and do not scale well to a large number of users. On the other hand, queue management algorithms have had a simple design from the outset. Given their simplicity, the hope is to approximate fairness. This class of algorithms is exemplified by Random Early Detection (RED) [5]. A router implementing RED maintains a single FIFO to be shared by all the flows, and drops an arriving packet at random during periods of congestion. The drop probability increases with the level of congestion. Since RED acts in *anticipation* of congestion, it does not suffer from the "lock out" and "full queue" problems [2] inherent in the widely deployed Drop Tail mechanism. By keeping the average queue-size small, RED reduces the delays experienced by most flows. However, like Drop Tail, RED is unable to penalize unresponsive flows. This is because the percentage of

[1] This research is supported in part by a TERMAN Fellewship and a Stanford Graduate Fellowship.

packets dropped from each flow over a period of time is almost the same. Consequently, misbehaving traffic can take up a large percentage of the link bandwidth and starve out TCP friendly flows.

To improve RED's ability for distinguishing unresponsive users, a few variants (like RED with penalty box [6] and Flow Random Early Drop (FRED) [11]) have been proposed. However, these variants incur extra implementation overhead since they need to collect certain types of state information. RED with penalty box stores information about unfriendly flows while FRED needs information about active connections. The recent paper by Ott et al [14] proposes an interesting algorithm called Stabilized RED (SRED) which stabilizes the occupancy of the FIFO buffer, independently of the number of active flows. More interestingly, SRED estimates the number of active connections and finds candidates for misbehaving flows. It does this by maintaining a data structure, called the "Zombie list", which serves as a proxy for information about recently seen flows. Although SRED identifies misbehaving flows, it does not propose a simple router mechanism for penalizing misbehaving flows. The CHOKe algorithm proposed below simultaneously identifies and penalizes misbehaving flows, and is simpler to implement than SRED.

In summary, all of the router algorithms (scheduling and queue management) developed thus far have been either able to provide fairness or simple to implement, but not both simultaneously. This has led to the belief that the two goals are somewhat incompatible (see [17]).

This paper takes a step in the direction of bridging fairness and simplicity. Specifically, we exhibit an active queue management algorithm, called CHOKe, that is simple to implement (since it requires no state information) and differentially penalizes misbehaving flows by dropping more of their packets. By doing this, CHOKe (CHOose and Keep for responsive flows, CHOose and Kill for unresponsive flows) aims to approximate max-min fairness for the flows that pass through a congested router[2].

The basic idea behind CHOKe is that the contents of the FIFO buffer form a "sufficient statistic" about the incoming traffic and can be used in a simple fashion to penalize misbehaving flows. When a packet arrives at a congested router, CHOKe draws a packet at random from the FIFO buffer and compares it with the arriving packet. If they both belong to the same flow, then they are both dropped, else the randomly chosen packet is left intact and the arriving packet is admitted into the buffer with a probability that depends on the level of congestion (this probability is computed exactly as in RED). The reason for doing this is that the FIFO buffer is more likely to have packets belonging to a misbehaving flow and hence these packets are more likely to be chosen for comparison. Further, packets belonging to a misbehaving flow arrive more numerously and are more likely to trigger comparisons. The intersection of these two high probability events is precisely the event that packets belonging to misbehaving flows are dropped. Therefore, packets of misbehaving flows are dropped more often than packets of well-behaved flows[3].

The rest of the paper is organized as follows: Section 2 explains our motivation and goals for using the CHOKe mechanism and describes the CHOKe algorithm (and a few variants) in detail. The simulation results are presented in Section 3. In Section 4, we propose and analyze models for the CHOKe algorithm. Our conclusions are presented in Section 5.

II. Motivation, Goals, and the Algorithm

Our work is motivated by the need for a simple, stateless algorithm that can achieve flow isolation and/or approximate fair bandwidth allocation. As mentioned in the introduction, existing algorithms (like RED, FQ and others) are either simple tc implement or able to achieve flow isolation, but not both simultaneously.

We seek a solution to the above problem in the context of the Internet. Thus, we are motivated to find schemes that differentially penalize "unfriendly" or "unresponsive" flows[4], which implies bad implementations of TCP, and UDP-based flows. Further, we seek to preserve some key features that RED possesses; such as its ability to avoid global synchronization[5], its ability to keep buffer occupancies small and ensure low delays, and its lack of bias against bursty traffic. By doing this, in the absence of unfriendly or unresponsive flows our algorithm will perform similarly to RED.

Next, we need a benchmark to compare the extent of fairness achieved by our solution. Maxmin fairness suggests itself as a natural candidate for two reasons: (a) It is well-defined and widely understood in the context of computer networks (see [1], page 526, or [10]), and (b) the FQ algorithm is known to achieve it. However, for any scheme to achieve perfect maxmin fairness without flow state information seems almost impossible. Maxmin fairness is not suitable in our context since we do not identify the flow(s) with the minimum resource allocation and maximize its (their) allocation. Instead we identify and reduce the allocation of the flows which consume the most resources. In other words, we attempt to *minimize the resource consumption of the maximum flow* or seek to achieve *minmax fairness*[6]. The resource freed up as a result of minimizing the maximum flow's consumption is distributed among the other flows. In the Internet context the former flows are either unfriendly TCP or UDP, and the latter flows are TCP.

[2] Note that we implicitly assume that the statistical characteristics and QoS requirements of each of the flows are identical. There is no loss of generality in making this assumption, since the paradigm of Class Based Queueing, e.g. as proposed in [4], allows one to extend the basic CHOKe scheme to a network consisting of heterogeneous flows.

[3] To our knowledge, the only other algorithm that makes a random comparison to identify misbehaving flows is SRED. The idea of making a random comparison, observed independently by us, is directly taken advantage of in CHOKe (i.e. without maintaining state information) to differentially drop packets belonging to misbehaving flows.

[4] see [7] for a formal definition of these terms

[5] Global synchronization refers to the situation where a lot of connections decrease or increase their window size at the same time, as happens under the Drop Tail mechanism (cf. [5]).

[6] Although we have not formally defined minmax fairness and don't explicitly invoke it, our meaning should be clear to the reader.

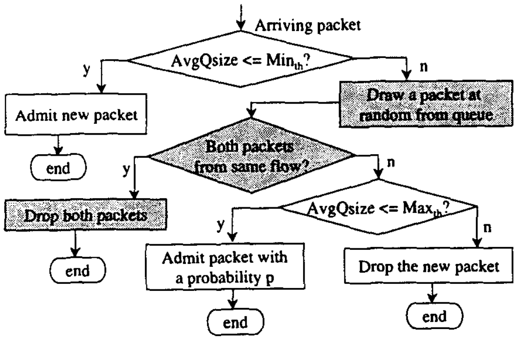

Fig. 1. The CHOKe algorithm

A. The CHOKe algorithm

Suppose that a router maintains a single FIFO buffer for queueing the packets of all the flows that share an outgoing link. We describe an algorithm, CHOKe, that differentially penalizes unresponsive and unfriendly flows. The state, taken to be the number of active flows and the flow ID of each of the packets, is assumed to be unknown to the algorithm. The only observable for the algorithm is the total occupancy of the buffer.

CHOKe calculates the average occupancy of the FIFO buffer using an exponential moving average, just as RED does. It also marks two thresholds on the buffer, a minimum threshold min_{th} and a maximum threshold max_{th}.

If the average queue size is less than min_{th}, every arriving packet is queued into the FIFO buffer. If the aggregated arrival rate is smaller than the output link capacity, the average queue size should not build up to min_{th} very often and packets are not dropped frequently. If the average queue size is greater than max_{th}, every arriving packet is dropped. This moves the queue occupancy back to below max_{th}. When the average queue size is bigger than min_{th}, each arriving packet is compared with a randomly selected packet, called drop candidate packet, from the FIFO buffer. If they have the same flow ID, they are both dropped. Otherwise, the randomly chosen packet is kept in the buffer (in the same position as before) and the arriving packet is dropped with a probability that depends on the average queue size. The drop probability is computed exactly as in RED. In particular, this means that packets are dropped with probability 1 if they arrive when the average queue size exceeds max_{th}. A flow chart of the algorithm is given in Figure 1. In order to bring the queue occupancy back to below max_{th} as fast as possible, we still compare and drop packets from the queue when the queue size is above the max_{th}.

In general, one can choose $m > 1$ packets from the buffer, compare all of them with the incoming packet, and drop the packets that have the same flow ID as the incoming packet. Not surprisingly, we shall find that choosing more than one drop candidate packet improves CHOKe's performance. This is especially true when there are multiple unresponsive flows; indeed, as the number of unresponsive flows increases, it is necessary to choose more drop candidate packets. However, since we insist on a completely stateless design, we cannot a priori know how many unresponsive flows are active at any time (and then choose a suitable value for m). It turns out that we can automate the process so that the algorithm chooses the proper value of $m \geq 1$. One way of achieving this is to introduce an intermediate threshold int_{th} which partitions the interval between min_{th} and max_{th} into two regions. When the average buffer occupancy is between min_{th} and int_{th} the algorithm can set $m =. 1$ and when the average buffer occupancy is between int_{th} and max_{th} it sets $m = 2$.[7] More generally, we can introduce multiple thresholds which partition the interval between min_{th} and max_{th} into k regions $R_1, R_2, \ldots, R_k$ and choose different values of m depending on the region the average buffer occupancy falls in. For example, we could choose $m = 2 \cdot i$ $(i = 1, \ldots, k)$, when the average queue size lies in region R_i. Obviously, we need to let m increase monotonically with the average queue size.

CHOKe is a truly stateless algorithm. It does not require any special data structure. Compared to a pure FIFO queue, there are just a few simple extra operations that CHOKe needs to perform: draw a packet randomly from the queue, compare flow IDs, and possibly drop both the incoming and the candidate packets. Since CHOKe is embedded in RED, it inherits the good features of RED mentioned previously. Finally, as a stateless algorithm, it's nearly as simple to implement as RED. To see this, let us consider the details of implementation. Drawing a packet at random can be implemented by generating a random address from which a packet flow ID is read out. Flow ID comparison can be done easily in hardware. It is arguably more difficult to drop a randomly chosen packet since this means removing it from a linked-list. Instead of doing this, we propose to add one extra bit to the packet header. The bit is set to one if the drop candidate is to be dropped. When a packet advances to the head of the FIFO buffer, the status of this bit determines whether it is to be immediately discarded or transmitted on the outgoing line.

III. Simulation Results

This section presents simulation results of CHOKe's performance in penalizing misbehaving flows and thus approximating fair bandwidth allocation. We shall use the RED and Drop Tail schemes, whose complexities are close to that of CHOKe, for comparison. The simulations range over a spectrum of network configurations and traffic mixes. The results are presented in three parts: single congested link, multiple congested links, and multiple misbehaving flows.

A. Single Congested Link

To illustrate CHOKe's performance when there is a single congested link, we consider the standard network configuration shown in Figure 2. The congested link in this network is be-

[7] When the buffer occupancy exceeds max_{th} we start dropping each incoming packet but m remains the same.

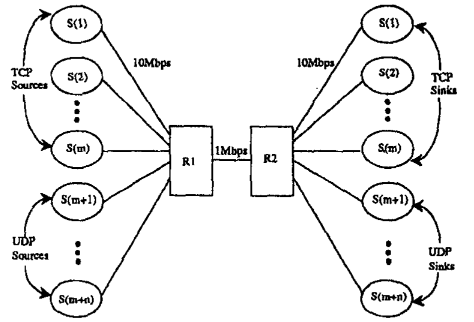

Fig. 2. Network Configuration: m TCP sources, n UDP sources

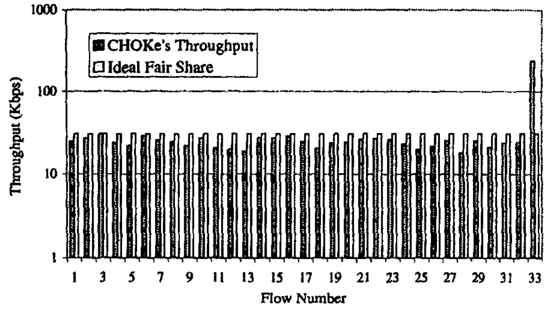

Fig. 4. CHOKe: Throughput Per Flow

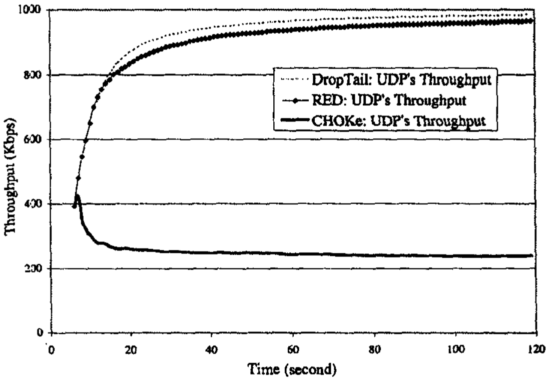

Fig. 3. UDP Throughput Comparison

tween the routers $R1$ and $R2$. The link, with capacity of 1 Mbps, is shared by m TCP and n UDP flows. An end host is connected to the routers using a 10 Mbps link, which is ten times the bottleneck link bandwidth. All links have a small propagation delay of 1ms so that the delay experienced by a packet is mainly caused by the buffer delay rather than the transmission delay. The maximum window size of TCP is set to 300 such that it doesn't become a limiting factor of a flow's throughput. The TCP flows are derived from FTP sessions which transmit large sized files. The UDP hosts send packets at a constant bit rate (CBR) of r Kbps, where r is a variable. All packets are set to have a size of 1K Bytes.

To study how much bandwidth a single nonadaptive UDP source can obtain when routers use different queue management schemes, we set up the following simulation: there are 32 TCP sources ($Flow1$ to $Flow32$) and 1 UDP source ($Flow33$) in the network. The UDP source sends packets at a rate r = 2 Mbps, twice the bandwidth of the bottleneck link, such that the link $R1$-$R2$ becomes congested. The minimum threshold min_{th} in the RED and CHOKe schemes is set to 100, allowing on average around 3 packets per flow in the buffer before a router starts dropping packets. Following [5], we set the maximum threshold max_{th} to be twice the min_{th}, and the physical queue size is fixed at 300 packets. The throughput of the UDP flow under different router algorithms: DropTail, RED and CHOKe, is plotted in Figure 3.

From Figure 3, we can clearly see that the RED and DropTail gateways do not discriminate against unresponsive flows. The UDP flow takes away more than 95% of the bottleneck link capacity and the TCP connections can only take the remaining 50 Kbps. CHOKe, on the other hand, improves the throughput of the TCP flows dramatically by limiting the UDP throughput to 250 Kbps, which is only around 25% of the link capacity. The total TCP flows' throughput is boosted from 50 Kbps to 750 Kbps.

To gauge the degree to which CHOKe achieves fair bandwidth allocation, the individual throughput of each of the 33 connections in the simulation above, along with their ideal fair shares, are plotted in Figure 4. Although the throughput of the UDP flow ($Flow33$) is still higher than the rest of the TCP flows, it can be seen that each TCP is allocated a bandwidth relatively close to its fair share. In CHOKe, a packet could be dropped because of a match or a random discard like in RED. A misbehaving flow, which has a high arrival rate and a high buffer occupancy, incurs packet dropping mostly due to matches. On the other hand, the packets of a responsive flow are hard to be matched, and therefore get dropped mainly because of random discard. In the above simulation, we find that matches are responsible for 85% of the UDP packet dropping, while 70% of the TCP packets dropping are caused by random discard.

We vary the UDP arrival rate r to study CHOKe's performance under different traffic load conditions. The simulation results are summarized in Figure 5, where the UDP's throughput versus the UDP flow arrival rate is plotted. The drop percentage of the UDP flow is also shown in the figure. The min_{th} and the max_{th} are set to 30 and 60 packets in this case to show that the CHOKe scheme works under different threshold settings. From the plot, we can see that CHOKe drops 23% of the UDP packets when its arrival rate is as low as 100 Kbps. As the UDP arrival rate increases, the drop percentage goes up as well. It drops almost all of the packets (98.3%) when the arrival rate reaches 10 Mbps. The average TCP flow's throughput stays almost constant. In comparison, RED's performance under different traffic

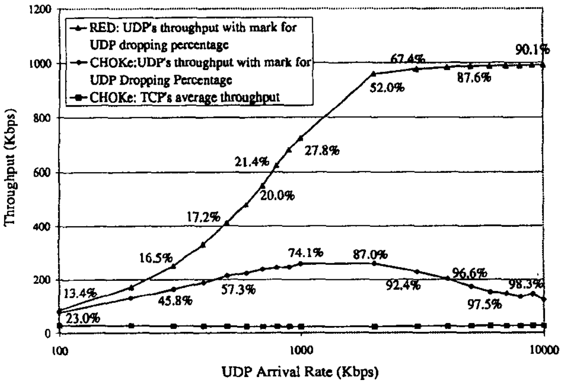

Fig. 5. Performance under Different Traffic Load

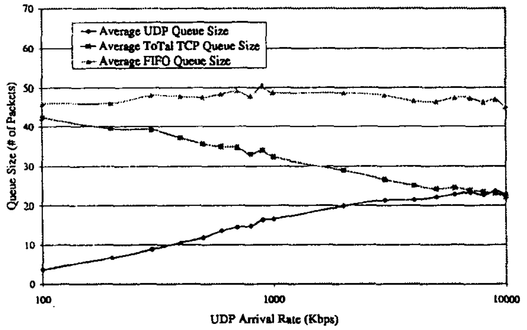

Fig. 6. CHOKe's Queue Distribution Under Different Loadings

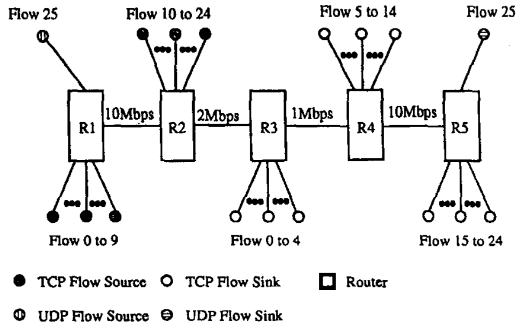

Fig. 7. Topology of Multiple Links

Flowid	T_{R2-R3}	T_{R3-R4}	T_{R4-R5}	Flowid	T_{R2-R3}	T_{R3-R4}	T_{R4-R5}
0	107.715	-	-	13	33.979	32.488	-
1	117.993	-	-	14	35.010	33.193	-
2	108.149	-	-	15	33.898	32.352	32.352
3	114.630	-	-	16	33.518	31.837	31.837
4	114.277	-	-	17	34.169	32.569	32.569
5	33.654	32.244	-	18	34.820	32.949	32.949
6	34.603	33.328	-	19	34.820	33.437	33.437
7	33.464	31.674	-	20	35.362	34.359	34.359
8	35.715	34.576	-	21	35.823	34.494	34.494
9	33.789	32.569	-	22	34.955	33.138	33.138
10	34.088	32.216	-	23	33.816	32.379	32.379
11	37.396	36.094	-	24	36.013	34.522	34.522
12	37.884	36.637	-	25	740.311	332.854	332.854

TABLE I

EACH FLOW'S THROUGHPUT AT DIFFERENT LINKS

(T_{Ri-Rj}=THROUGHPUT AT LINK RI-RJ)

load is shown in Figure 5 as well. It is obvious that RED can't provide protection against greedy connections. The unresponsive flows use up all the network bandwidth and starve out the well-behaving flows. When the average queue size goes above the max_{th} and all the arrival packets are dropped, RED becomes a Drop Tail scheme.

Figure 6 shows the queue distribution among the flows for different traffic load conditions. It is not surprising that CHOKe can control the average queue size as RED does since it imitates RED on this sense. When the UDP arrival rate is 100 Kbps, only a few times the rate of a single TCP flow, CHOKe is able to detect this small difference and drops 23% of the UDP traffic. When the UDP arrival rate goes up, its share of the queue occupation increases. Therefore, it becomes easier to catch a UDP flow packet as a drop candidate. Besides, with the increasing arrival rate, the UDP flow triggers more comparisons. As a result, the probability of obtaining a match UDP packet increases. Associated with each matching, there are two packets that get dropped: the incoming one and the one from the queue. So when the probability of matching approaches 0.5, for each incoming UDP packet, there is on average $0.5 \cdot 0 + 0.5 \cdot 2 = 1$ packet that is being dropped; i.e. the proportion of dropped UDP packets approaches 1 and of the UDP flow's goodput goes to zero. This intuitively explains why the UDP throughput goes down under heavy load and why the average queue size of the UDP flow doesn't even become the dominant portion of the queue usage. The detailed discussions will be covered in Section IV.

B. Multiple Congested Links

So far we have seen the performance of CHOKe in a simple network configuration with one congested link. In this section, we study how CHOKe performs when there are multiple congested links in the network. A sample network configuration with five routers is constructed as shown in Figure 7. The first link between router *R1* and *R2* (*R1-R2*) has a capacity of 10 Mbps so that when the sources connected to it send packets at high rate, the following link *R2-R3* becomes congested. The third link, *R3-R4*, has only half the bandwidth of the link *R2-R3*and becomes congested since the link's arrival rate exceeds its capacity. The final link *R4-R5* is lightly loaded. Using these links, we can demonstrate CHOKe's performance under cascaded, multiple congested links. In total there are 25 TCP flows and 1 UDP flow in the network, whose sources and sinks are shown in the figure. The UDP source sends packets at a rate of 2 Mbps while the other network parameters remain the same with Figure 5.

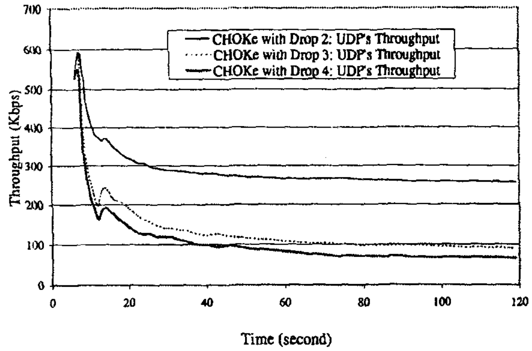

Fig. 8. CHOKe with Multiple Drops: Throughput Comparison

Table I lists the throughput of each flow at various links in the network. We see that after competing with 25 TCP flows in the first congested link *R2-R3*, the UDP flow loses around 63% of its traffic when it sends data at the same rate as the link bandwidth (2 Mbps). The UDP traffic that gets through to the next link, *R3-R4*, constitutes 74% of the arrival rate at that link. The flow suffers an additional 55% loss at this link when competing with the remaining 20 TCP flows. Comparing these results with the single congested link case, we can see that each of the cascaded congested links behaves roughly as if it was a single congested link. Multiple congested links therefore have a multiplicative effect on UDP packet losses. Since TCP flows can automatically detect their bottleneck link bandwidth, they suffer much less loss.

¿From the simulation results in Table I and the discussion above, one infers that since TCP flows are responsive to congestion indication and adjust their packet injection rates accordingly, their packet loss rate in a network under the CHOKe scheme is quite small. But nonadaptive flows, like UDP, suffer from severe packet losses.

C. Multiple Misbehaving Flows

We now study the effect of the generalized CHOKe algorithm where more than one drop candidate is drawn from the buffer. Figure 8 shows the performance of the CHOKe algorithm with one, two and three drop candidates. The network configuration for the simulation is the same as the one in Figure 2. The rate for the UDP source is 2 Mbps. Since CHOKe with $m - 1$ candidates has a maximum drop of m packets ($m - 1$ candidate packets $+1$ incoming packet), it will be referred to as CHOKe with drop m. (Under this terminology the basic CHOKe scheme will be referred to as CHOKe with drop 2.) Figure 8 shows that CHOKe with multiple drops has a better control over the unresponsive UDP traffic than the basic CHOKe algorithm, which is not surprising[8].

When there are many UDP flows in the network, CHOKe with multiple drops exhibits its advantage over the basic algorithm. A simulation configuration with 32 TCP and 5 UDP sources is set up, using the basic network topology shown in Figure 2. All the UDP sources are assumed to have the same arrival rate. The min_{th} and max_{th} are still set up to be 30 and 60 packets. The simulation results for the basic CHOKe algorithm are given in Figure 9. As shown in the figure, the throughput of the UDP sources goes up monotonically with their arrival rate. As a result, there is almost no bandwidth left for the TCP sources. Although the total UDP flows occupy almost all the buffer space, each UDP connection takes only around 20% of the queue. As a result, the chance of catching a right victim is low and UDP flows can't be regulated as desired.

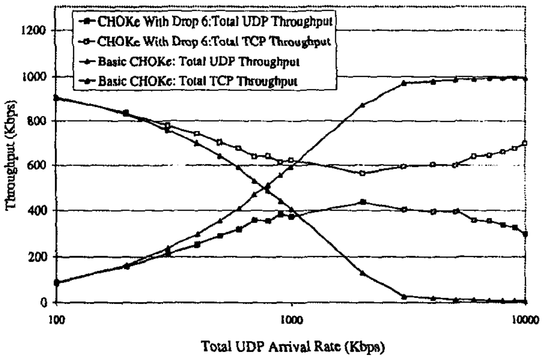

Fig. 9. CHOKe: Throughput for 32 TCP and 5 UDP configuration

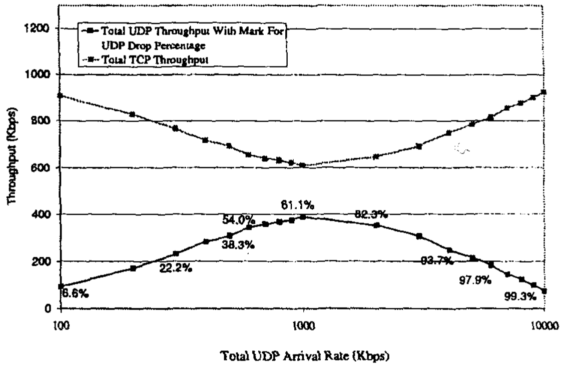

Fig. 10. Self Adjusting CHOKe: Throughput for 32 TCP and 5 UDP configuration

On the other hand, CHOKe with 5 drops boosts the throughput of the TCP flows in this situation, as shown in Figure 9. Because multiple drop candidates are selected from the queue, the chance of catching the bad flows increases. Therefore, CHOKe with multiple drops can penalize those flows that are hard to detect but use more than their fair share of network bandwidth.

The above simulation illustrates the need for multiple drops when there are multiple unresponsive flows. Since the average queue size is a good indication of congestion level, we use it to automatically decide on the appropriate number of drops. The scheme works as follows: the region between min_{th} and max_{th}

[8] It is interesting to observe that the performance improvement from CHOKe drop 3 to CHOKe drop 4 is very small.

is divided into k subregions and the number of drops in a region is set to $2 \cdot i$ ($i = 1 \ldots k$). The simulation results, plotted in Figure 10, clearly show that this self adjusting mechanism works well in case of multiple unresponsive flows.

IV. Modeling and Analysis of the Algorithm

This section develops some mathematical models for analyzing the CHOKe algorithm. We distinguish among the following three versions of CHOKe:

- *Original CHOKe,* in which the drop candidate packet is chosen randomly from the queue.
- *Front CHOKe,* in which the drop candidate is always the packet at the head of the queue.
- *Back CHOKe,* in which the drop candidate is always the packet at the tail of the queue.

The last two variations are introduced because of the difficulty of analyzing the original CHOKe. For both the front and back CHOKe simple models are presented, that are analyzable, and are reasonable approximations of the actual scheme. In the next two sections we use queues with Poisson arrivals and exponential services although this is unrealistic. Other than tractability, these models allow us to gain some simple insights. We discuss the suitability and the implications of these models for realistic scenarios in Section IV-C.

A. Front CHOKe

Consider a queue with N independent Poisson arrivals, each of rate λ_i, and independent exponential service times. The queueing discipline is first-in-first-out (FIFO) and the mean service time of each packet is assumed to be $1/\mu$. To ease the exposition, let us first consider only two arrival processes with arrival rates λ_1 and λ_2. We shall refer to the packets of these flows as type 1 and type 2 packets, respectively.

An arriving packet is either admitted to the queue or dropped upon arrival depending on the outcome of certain comparisons, as explained next. Each arriving packet is compared with the packet at the head of the queue (if the queue is nonempty). If the types of both packets are the same, then they are both dropped. Else, the arriving packet is admitted to the queue. Of course, if a packet arrives at an empty queue, then it is automatically admitted. We assume that the queue has an infinite waiting room and that packets can only be dropped either when they arrive or when they are at the head of the queue. For now let us also suppose that $\lambda_1 + \lambda_2 < \mu$ so that the queue is stable. We shall later see that with the dropping scheme in place the queue will be stable for all values of λ_1, λ_2 and μ. The assumption of stability guarantees that an equilibrium distribution exists for the queue-size process.

Write p_{1,a_1} (respectively, p_{2,a_1}) for the probability that an *arriving* packet of type 1 sees a type 1 (respectively, type 2) packet at the head of the queue. Let p_{0,a_1} be the probability that an arriving type 1 packet sees an empty queue. The well-known PASTA[9] property [18] asserts that $p_{i,a_1} = p_i$ for $i = 0, 1, 2$, where the p_i are the corresponding equilibrium probabilities as seen at an arbitrary instant of time. Since we have assumed that both the arrival processes are Poisson and independent, the same reasoning gives that the probability, p_{i,a_2}, that an arriving type 2 packet sees a type i packet at the head of the queue also equals p_i, for $i = 1, 2$ and the propapility that it sees an empty queue also equals p_0.

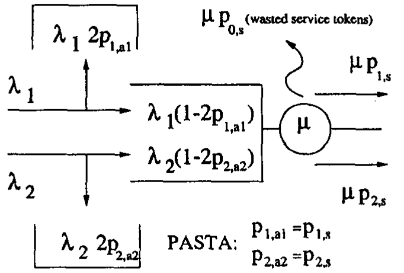

Fig. 11. Front CHOKe model

Given that the services are i.i.d. and exponential of rate $1/\mu$, we can represent the service process by an independent Poisson process of rate μ. Thus, service tokens arrive according to a rate μ Poisson process and will liberate the packet at the head of the queue (whatever its type), so long as the queue is nonempty. If the queue is empty when a service token arrives, then, of course, the service token is wasted. Write $p_{i,s}$, $i = 1, 2$, for the probability that a *service token* sees a type i packet at the head of the queue. And let $p_{0,s}$ be the probability that a service token arrives at an empty queue. Applying the PASTA property again, we see that $p_{i,s}$ also equals p_i for $i = 0, 1, 2$.

We summarize these observations as follows: $p_{i,a_1} = p_{i,a_2} = p_{i,s} = p_i$ for $i = 0, 1, 2$. Of course, $p_0 + p_1 + p_2 = 1$.

We now use a rate conservation argument to evaluate the p_i. Consider just type 1 packets. The rate at which these packets arrive is λ_1. A proportion p_1 of these packets is dropped at arrival. A further proportion p_1 will be dropped from the head of the queue. (We note in passing that packets are always dropped in pairs.) Therefore, the rate of departure of type 1 packets from the queue is $\lambda_1(1 - 2p_1)$. But to each type 1 packet that leaves the queue there corresponds a service token that liberated it. Since service tokens arrive at rate μ and a proportion p_1 of them liberate type 1 packets, the rate at which service tokens cause type 1 departures is μp_1 (see Figure 11).

The requisite rate conservation equation is therefore $\lambda_1(1 - 2p_1) = \mu p_1$. Solving for p_1 we obtain that

$$p_1 = \frac{\lambda_1}{\mu + 2\lambda_1}.$$

Similarly,

$$p_2 = \frac{\lambda_2}{\mu + 2\lambda_2}.$$

The form of these probabilities is somewhat surprising: they do not depend on the arrival rate (or, indeed, the number) of other incoming flows. Since μp_i is the departure rate of type

[9] PASTA: Poisson Arrivals See Time Averages

i packets, this in turn implies that the "goodput" of each flow depends only on its own arrival rate and on the service rate μ. Clearly the validity of these formulas relies heavily on our use of the PASTA property and cannot be expected to generalize to non-Poisson settings. Nevertheless, given the Poisson assumption it is equally true for other variants. For example, if we consider a scheme where both the drop candidate packet and the packet to be serviced are chosen randomly from the queue, it is again true that $p_{1,a_1} = p_{1,a_2} = p_{1,s} = p_1$ and hence that $p_1 = \frac{\lambda_1}{\mu+2\lambda_1}$. More generally, whenever there is a symmetry between the service discipline (e.g. FIFO, random, etc) and the comparison/dropping discipline (respectively, packet at the head of the queue, randomly chosen packet, etc) one can invoke the PASTA property. Observe that p_1 and p_2 are strictly less then 1/2 for all values of λ_i and μ. This implies that p_0 is always strictly positive, ensuring that the queue-size process is positive recurrent for all values of λ_i and μ.

When the number of flows, N, is bigger than two, all of the previous PASTA arguments will go through and one obtains

$$p_i = \frac{\lambda_i}{\mu + 2\lambda_i}, \qquad (1)$$

from which the goodput of flow i is seen to be μp_i. For $N > 2$ stability is not automatically guaranteed and one requires that the arrival rates λ_i satisfy

$$\sum_{i=1}^{N} p_i = \sum_{i=1}^{N} \frac{\lambda_i}{\mu + 2\lambda_i} < 1 \qquad (2)$$

Equation (2) merely expresses the fact that the total effective arrival rate should be less than the average service rate since $\frac{\lambda_i}{\mu+2\lambda_i} = \frac{\lambda_i(1-2p_i)}{\mu}$. Note that the condition in Equation (2) is weaker than the usual stability condition (net arrival rate strictly less than service rate):

$$\sum_{i=1}^{N} \frac{\lambda_i}{\mu} < 1, \qquad (3)$$

in the sense that any positive vector $(\lambda_1, \ldots, \lambda_N)$ that satisfies (3) also satisfies (2). This is simply a consequence of the inequality $\frac{\lambda_i}{\mu} > \frac{\lambda_i}{\mu+2\lambda_i}$.

Table II compares the throughputs T_i of independent Poisson flows sharing a single FIFO buffer with service rate $\mu = 1$ and different arrival rates λ_i. The column "Simulation" gives the throughputs obtained by simulating the queue and the column "Theory" gives throughputs obtained from the formulas derived above.

B. Back CHOKe

Back CHOKe refers to the situation where the drop candidate packet is always chosen from the back of the queue. According to the algorithm, the most recently admitted packet and an incoming packet will be dropped if their flow ids are identical. Again if the server chooses to serve the packet at the back of the queue, and arrivals and services are Poisson, the symmetry between the service and dropping disciplines allows one to invoke the PASTA property and obtain results similar to that of the previous section. But this scheme has the obvious disadvantage that the departure order of packets is reversed, rendering this scheme impractical.

Input				*Simulation*				*Theory*			
μ	λ_1	λ_2	λ_3	P_0	T_1	T_2	T_3	P_0	T_1	T_2	T_3
1	0.5	1	-	0.416	0.251	0.333	-	0.417	0.250	0.333	-
1	1	2	-	0.267	0.333	0.401	-	0.267	0.333	0.400	-
1	2	3	-	0.173	0.399	0.428	-	0.171	0.400	0.429	-
1	3	4	-	0.128	0.428	0.444	-	0.127	0.429	0.444	-
1	3	5	-	0.117	0.429	0.455	-	0.117	0.429	0.455	-
1	3	6	-	0.109	0.429	0.462	-	0.110	0.429	0.462	-
1	0.3	0.6	0.9	0.219	0.188	0.274	0.319	0.218	0.188	0.273	0.321
1	0.5	1	1.5	0.047	0.253	0.328	0.373	0.042	0.250	0.333	0.375

TABLE II

FRONT CHOKE SIMULATION AND MODEL COMPARISON.

T_i =THROUGHPUT OF FLOW $i = \mu p_i$

On the other hand, it is more difficult to analyze the situation in which packets are dropped from the back but serviced from the front. We seek a compromise between tractability and practisability by introducing the following further modification. Suppose the router records, in a separate memory location "Mloc", the flow id of the most recently admitted packet. The id of each arriving packet is compared against the entry in Mloc and the packet is dropped if there is an agreement in the ids. Else, the incoming packet is admitted and its id is stored in Mloc. Notice that in this scheme packets are only dropped when they arrive and that it is possible for a packet to be dropped even when the queue is empty.

With independent Poisson arrivals (and regardless of the service distribution), this scheme bears a striking resemblance to the interacting particle system of the *marching soldiers*. In the marching soldiers problem, soldiers (infinite or finite in number) are placed at integer points on the x-axis. They all face the positive y-axis direction. The soldier at location i has a clock that ticks according a Poisson process of rate λ_i, independently of the clocks of the other soldiers. When his/her clock ticks, a soldier is allowed to take a step forward. The interaction is caused by the requirement that no soldier may be more than one step ahead of his/her immediate neighbors. This interaction may be said to be "fair" in the sense that the speed with which a soldier can move is regulated by the speed with which his/her neighbors (and, by extension, the speed with which the whole file) can move.

To relate this to the version of back CHOKe we are studying, suppose that N soldiers are placed at the vertices of a fully connected graph (so that everybody is everybody else's neighbor), and that we record the id of the soldier who most recently advanced. Impose the requirement that no soldier can take two consecutive steps before at least one of the others has taken a step. This problem is exactly the version of back CHOKe we wish to analyze.

Given independent Poisson arrivals and N sources the system can be modelled as a Markov chain, where the state is equal to the id stored in Mloc. The transition probabilities for this chain are $P_{ij} = \frac{\lambda_j}{S-\lambda_i}$, where $S = \sum_{k=1}^{N} \lambda_k$. The Markov chain is reversible since the condition $\pi_i P_{ij} = \pi_j P_{ji}$ is satisfied by the stationary distribution:

$$\pi_i = \frac{\lambda_i(S-\lambda_i)}{\sum_{k=1}^{N} \lambda_k(S-\lambda_k)}. \tag{4}$$

Thus, given the λ_i, one can easily evaluate the π_i, which are the long run average number of packets of type i that are admitted. Since admitted packets are never dropped, assuming that the service rate is large enough, π_i equals the share of the bandwidth of the outgoing link that source i obtains.

An obvious interesting generalization is to store the id of the $M \leq N$ most recently admitted packets in Mloc. It is clear that we still have a Markov chain with $N!/(N-M)!$ states, where each state corresponds to an ordered vector denoting the type of the M most recently admitted packets. An arriving packet will not be admitted if its id is in Mloc. Although the chain is no longer reversible, it is dynamically reversible [9], as we shall soon see. Associate with each state $s = (i, j, \ldots, k)$ the conjugate state $s^+ = (k, \ldots, j, i)$ which is just the reversal of s. We point out that the entries in state s are distinct, as required by our scheme. Now, for each pair of states r, s and their respective conjugates r^+, s^+, dynamic reversibility can be verified by checking that the condition $\pi_r P_{rs} = \pi_{s^+} P_{s^+r^+}$ holds.

For concreteness and notational simplicity, consider the case $M = 2$. Write (ij) for the state of Mloc when the most recent packet to enter the queue is of type j and the second most recent entry into the queue is of type i. The transition probabilities are $P_{(ij),(jk)} = \frac{\lambda_k}{S-\lambda_i-\lambda_j}$. Recall that $(ij)^+ = (ji)$. It is easy to check that the condition:

$$\pi_{(ij)} P_{(ij),(jk)} = \pi_{(jk)^+} P_{(jk)^+,(ij)^+} = \pi_{(kj)} P_{(kj),(ji)}$$

is satisfied by the stationary distribution:

$$\pi_{ij} = \frac{\lambda_i\lambda_j(S-\lambda_i-\lambda_j)}{\sum_{i=1}^{N}\sum_{j=1}^{N} \lambda_i\lambda_j(S-\lambda_i-\lambda_j)}, \tag{5}$$

for $i \neq j$ and $\pi_{ij} = 0$ for $i = j$.

As before, packets that enter the queue will never be dropped. Thus the throughput of source i is simply equal to π_i, where for $i = 1, \ldots, N$, $\pi_i = \sum_{j=1}^{N} \pi_{ij}$. The condition for the stability of the queue is:

$$\sum_{i=1}^{N} \lambda_i \pi_i < \mu.$$

Table III presents some results for back CHOKe. As it can be seen by Equations (4),(5) the stationary distribution is independent of μ. However, for these results to make sense μ should satisfy the stability condition. Comparing the first and the second row we see the improvement we get by back CHOKe even for a memory of one, over the case where we admit all the packets. Admitting all the packets is equivalent to having no memory. In this case, the transition probabilities are simply equal to $P_{ij} = \lambda_j/S$. The relationship between memory and throughput is tabulated in the other rows of the table.

Input							*Results*				
N	M	λ_1	λ_2	λ_3	λ_4	λ_5	T_1	T_2	T_3	T_4	T_5
4	0	1	1	1	7	-	0.1	0.1	0.1	0.7	-
4	1	1	1	1	7	-	0.1875	0.1875	0.1875	0.4375	-
4	2	1	1	1	7	-	0.2273	0.2273	0.2273	0.3182	-
4	3	1	1	1	7	-	0.25	0.25	0.25	0.25	-
4	1	1	2	3	4	-	0.1286	0.2286	0.3000	0.3429	-
4	2	1	2	3	4	-	0.1733	0.2533	0.2800	0.2933	-
5	3	1	2	3	4	5	0.1405	0.1953	0.2135	0.2226	0.2281
5	3	1	3	5	7	9	0.1101	0.2034	0.2220	0.2300	0.2345

TABLE III

BACK CHOKE RESULTS. T_i =THROUGHPUT OF FLOW $i = \pi_i$.

C. Modeling implications

Our objectives in this section are twofold: First, we interpret and summarize the formulas obtained for the theoretical models. Second, we comment on the use of these models to understand the networks with UDP and TCP flows simulated in Section III.

We begin with front CHOKe. Table II shows that as the offered rate of a flow increases, it's throughput does not increase proportionally. For example, with λ_1 held constant at 3, and as λ_2 increases from 4 to 5 to 6, the throughput of flow 2 increases only slightly (from 0.444 to 0.455 to 0.462) while the throughput of flow 1 remains constant at 0.429 due to the independence between flows. This is in keeping with the notion of fairness that greedy flows are penalized more as their offered load increases.

Now consider the simulation scenario of Section III. There are 32 TCP flows and 1 UDP flow. Write λ_{UDP} for the *offered rate* of the UDP flow and μ for the service rate which is equal to 1000 Kbps, the capacity of the bottleneck link. Although the traffic is far from being Poisson, Equation (1) approximates the situation well enough. From Figure 5 we can see that when λ_{UDP} = 1000 Kbps, the UDP drop percentage is 74.1% (thus UDP goodput is about 280 Kbps). The theoretical drop percentage is $2\cdot\frac{1000}{2\times1000+1000} = 66\%$, which is close enough to the simulation value of 74.1%. For λ_{UDP} = 500 Kbps, the drop probabilities are 50% (model) and 57.3% (simulation). And for λ_{UDP} = 3000 Kbps the corresponding numbers are 85.7% (model) and 92.4% (simulation). Since TCP flows are sensitive to congestion, they lack the independent increments property of the Poisson process and the model cannot predict their behavior reliably. However, since UDP is completely congestion unaware, its instantaneous packet submission rate does not vary and the model captures it's behavior well enough even though its distribution is not Poisson.

Another interesting point is that for $\lambda_i \gg \mu \Rightarrow p_i \simeq 1/2$. This is evident in Figure 6 for the UDP flow. Further, T_i can never be more than 50%. In other words a flow cannot consume more than half of the bottleneck bandwidth. Since $2p_i \simeq 1$, nearly all the packets of very aggressive flows are dropped, as is the case in Figure 5[10]. At the other extreme when $\lambda_i \ll \mu$, $p_i \simeq \lambda_i/\mu$, $p_i/p_j \simeq \lambda_i/\lambda_j$. In a sense this ratio of the drop probabilities is really a statement about the fairness of the dropping scheme. That is, flows which have a high arrival rate relative to other flows incur a higher proportion of drops.

Now consider back CHOKe and the results tabulated in Table III. As in front CHOKe, we observe that as the difference between the arrival rates of the various flows increases, the difference between the throughput does not increase proportionally. For example, the last two columns of Table III show that the difference in the maximum throughput of flow 5 is very small even when its arrival rate is more than doubled.

The most interesting observation with back CHOKe has to do with the memory M. It is clear from the first three rows of Table III that as the memory increases, the scheme becomes more fair since the throughput of each flow approaches its fair share. In the special case where $N - M = 1$ the scheme behaves like a round robin algorithm, resulting in perfect fairness among the flows. The amount of memory can be considered as the counterpart of the number of drop candidate packets in the original CHOKe, even though in the model of back CHOKe we don't drop packets from the queue as we do in the original scheme. In Figure 8 we observe that although as we increase the number of drop candidate packets the results are better, the improvement between two consecutive numbers of drop candidate packets is getting smaller. This conclusion can be also drawn from Table III. We note that by using a memory of 1, the maximum throughput decreases by $0.7 - 0.4375 = 0.2625$, by using a memory of 2 it decreases by $0.4375 - 0.3182 = 0.1193 < 0.2625$, and by using $M = 3$ it decreases by $0.0682 < 0.1193$.

V. Conclusions

This paper proposes a packet dropping scheme, CHOKe, which aims to approximate fair queueing at a minimal implementation overhead. Simulations suggest that it works well in protecting congestion-sensitive flows from congestion-insensitive or congestion-causing flows. Analytical models were derived for gaining insights about the algorithm and for understanding the simulations. Further work involves studying the performance of the algorithm under a wider range of parameters, network topologies and real traffic traces, obtaining more accurate theoretical models and insights, and considering hardware implementation issues.

VI. Acknowledgment

Balaji Prabhakar thanks David Clark of MIT for several stimulating discussions on the role of "state" in high-speed scheduling algorithms. These discussions have influenced much of the work in this paper.

References

[1] Bertsekas, D. and Gallager, R., *Data Networks*, Second edition, Prentice Hall, 1992.

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

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

[4] Floyd, S. and Jacobson, V., "Link-sharing and Resource Management Models for Packet Networks", *IEEE/ACM Transactions on Networking*, Vol. 3 No. 4, pp. 365-386, August 1995.

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

[6] Floyd, S., and Fall, K., "Router Mechanisms to Support End-to-End Congestion Control", *LBL Technical report*, February 1997.

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

[8] Floyd, S., Fall, K. and Tieu, K., "Estimating Arrival Rates from the RED Packet Drop History", *http://www.aciri.org/floyd/end2end-paper.html*, April 1998.

[9] Kelly, F., *Reversibility and Stochastic Networks*, John Wiley & Sons 1979.

[10] Keshav, S., "Congestion Control in Computer Networks", PhD Thesis, published as UC Berkeley TR-654 , September 1991.

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

[12] Manin, A. and Ramakrishnan K., "Gateway Congestion Control Survey", *IETF RFC (Informational) 1254*, August 1991.

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

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

[15] Pan, R. and Prabhakar, B., "CHOKe - A simple approach for providing Quality of Service through stateless approximation of fair queueing", *Stanford CSL Technical report CSL-TR-99-779*, March 1999.

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

[17] Suter, B., Lakshman, T., Stiliadis, D. and Choudhury, A., "Efficient Active Queue Management for Internet Routers", *Interop 98*.

[18] Wolff, R. *Stochastic Modeling and the Theory of Queues*, Prentice Hall 1989.

[19] ns - Network Simulator (Version 2.0), October 1998.

[10] One wonders why UDP does not consume half the bottleneck bandwidth for large λ_{UDP}, since this is what the model predicts. The answer lies in the fact (observed in simulations) that even though UDP packets occupy half the buffer space, the packet *at the front* of the buffer belongs to the UDP flow with a probability much less than half, since it has survived many more comparisons and is more likely to belong to TCP flows.