CHOKe
A stateless mechanism for providing Quality of Service in the Internet

Balaji Prabhakar, Rong Pan
Department of Electrical Engineering
Stanford University
Stanford, CA 94305
{balaji,rong}@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.

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 [1][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 [17] 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 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 [18]).

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.

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[1].

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. Our conclusions are presented in Section 4.

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 to 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[2], 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[3], its ability to keep buffer occupancies small and ensure low delays, and its lack of bias against bursty traffic. By doing this CHOKe performs similarly to RED in the absence of unfriendly or unresponsive flows.

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(s)* or seek to achieve *minmax fairness*. The appendix defines minmax fairness and [16] proves that the minmax allocation vector equals the maxmin allocation vector.

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. 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 replaced (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

[1] 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.

[2] See [7] for a formal definition of these terms.

[3] 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]).

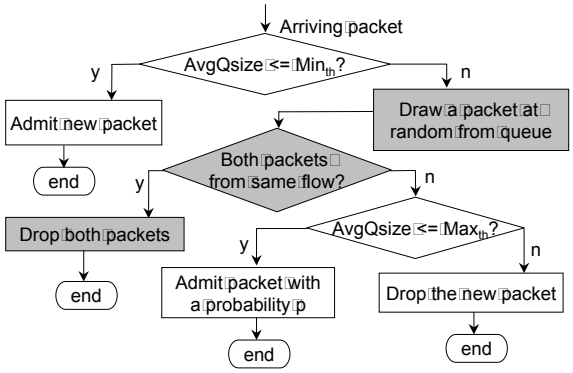

Fig. 1. The CHOKe algorithm

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 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 lies 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 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.

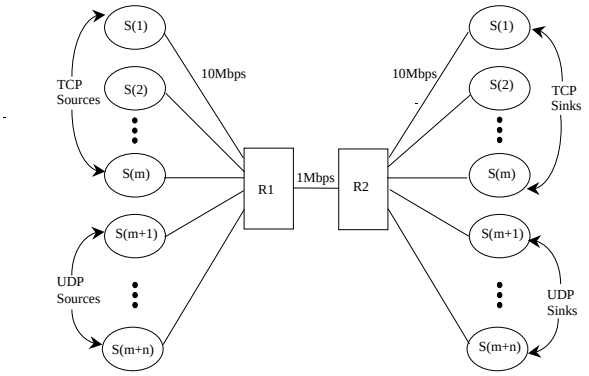

Fig. 2. Network Configuration: m TCP sources, n UDP sources

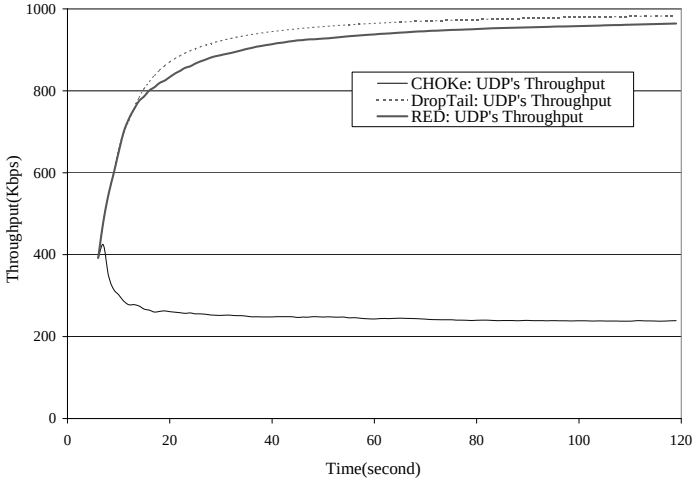

Fig. 3. UDP Throughput Comparison

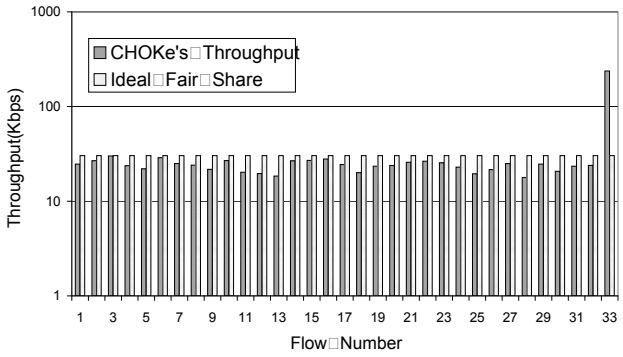

Fig. 4. CHOKe: Throughput Per Flow

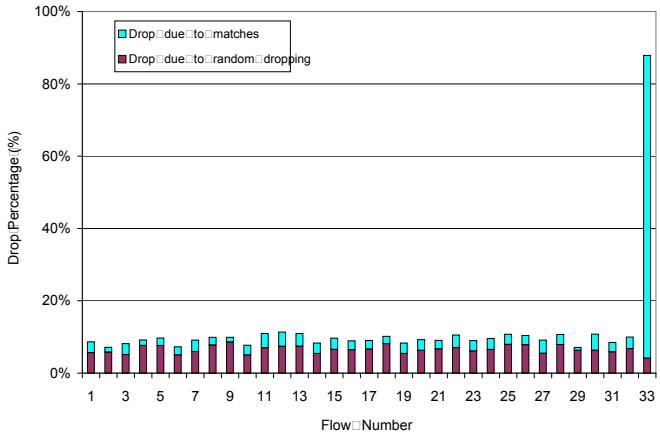

Fig. 5. CHOKe: Drop Decomposition

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 between 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 (so 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, so 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 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 un-

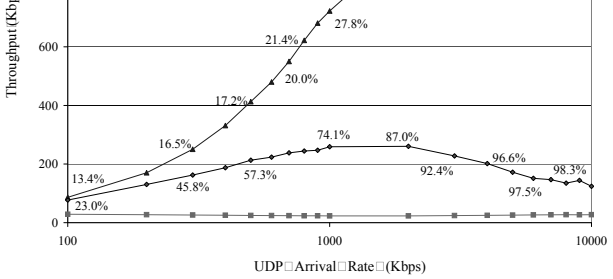

Fig. 6. Performance under Different Traffic Load

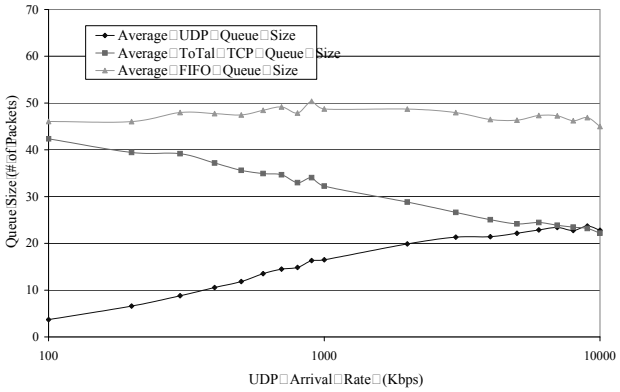

Fig. 7. CHOKe's Queue Distribution Under Different Loadings

responsive 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 about 250 Kbps, which is only 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, a match is rarer for the packets of a responsive flow and drops occur mainly due to random discard. In the above simulation we find that matches are responsible for 85% of UDP packet drops, while 30% of the TCP packet drops are due to matches. The drop decomposition is shown in Figure 5.

Next, we vary the UDP arrival rate r to study CHOKe's performance under different traffic load conditions. The simulation results are summarized in Figure 6, 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 values of min_{th} and max_{th} are set at 30 and 60 respectively to study the performance of CHOKe under different threshold settings. From the plot, we 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. The average TCP flow's throughput stays almost constant. For comparison, RED's performance under different traffic load is also shown in Figure 6. As can be seen RED is unable to discriminate against unresponsive flows.

Figure 7 shows the queue distribution among the flows for different traffic load conditions. It is interesting to note that the occupancy of the UDP flow increases monotonically with its offered rate. This increases the chance that a match is obtained for UDP packets. Since two packets are dropped when there is a match, as UDP occupancy approaches 50% (and hence the probability of a match conditioned on a UDP arrival approaches 0.5), each arriving UDP packet causes on average one UDP drop. That is, the proportion of dropped UDP packets approaches 1 and the UDP flow's goodput goes to zero. This intuitively explains why the UDP throughput decreases drastically under heavy loading.

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.

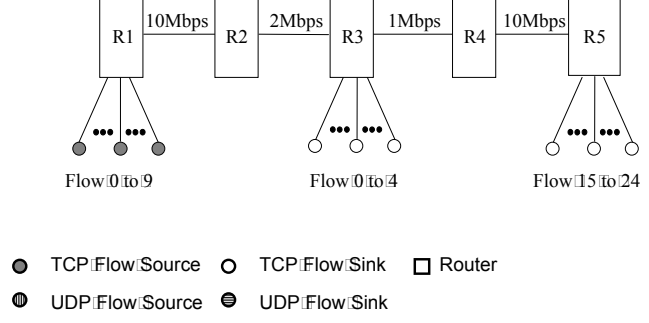

Fig. 8. Topology of Multiple Links

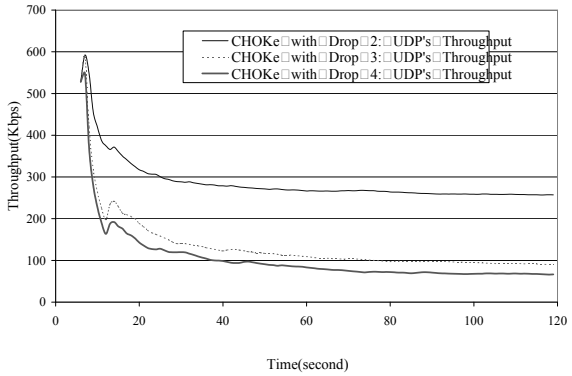

Fig. 9. CHOKe with Multiple Drops: Throughput Comparison

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)

A sample network configuration with five routers is constructed in Figure 8. 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 a 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 as in the setting of Figure 6. 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 see that each of the cascaded congested links behaves roughly as if it was a single congested link. Multiple congested links therefore have an additive effect on UDP packet losses. On the other hand, TCP flows can automatically detect their bottleneck link bandwidth and therefore 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 severe packet

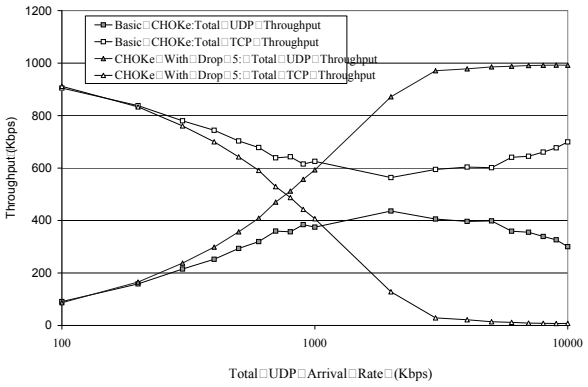

Fig. 10. CHOKe: Throughput for 32 TCP and 5 UDP configuration

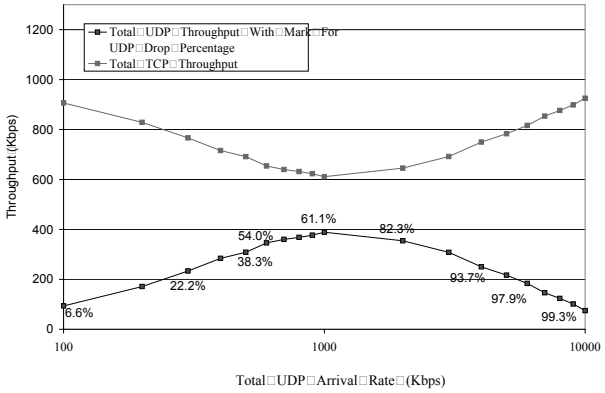

Fig. 11. Self Adjusting CHOKe: Throughput for 32 TCP and 5 UDP configuration

losses.

C. Multiple Drop Candidates and Misbehaving Flows

We now study the effect of the generalized CHOKe algorithm where more than one drop candidate is drawn from the buffer. Figure 9 shows the performance of the CHOKe algorithm with one, two and three drop candidates for the network configuration of 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. Not surprisingly, Figure 9 shows that CHOKe with multiple drops has a better control over the unresponsive UDP traffic[4].

When there are many UDP flows in the network, CHOKe with multiple drops exhibits its advantage over the basic algorithm (i.e. CHOKe drop 2). A simulation configuration with 32 TCP and 5 UDP sources is set up, similar to the configuration in Figure 2. All the UDP sources are assumed to have the same arrival rate. The values of min_{th} and max_{th} are still set at 30 and 60 respectively. The results for CHOKe drop 2 and CHOKe drop 5 are given in Figure 10. 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, since each UDP connection takes only around 20% of the queue. As a result, the chance of obtaining a match for the packets of a UDP flow are slim.

On the other hand, CHOKe with 5 drops boosts the throughput of the TCP flows. Clearly, choosing multiple drop candidates improves the chance of dropping UDP packets and improving TCP throughput.

Figure 10 demonstrates the need for multiple drops when there are multiple unresponsive flows. However, the number of unresponsive traversing link will typically be unknown to the algorithm. One way of automating the process of determining the appropriate number of samples that must be chosen for comparison is to observe that the average queue size is a good indicator of the congestion level. For a fixed service rate, congestion increases with the arrival rate and, more importantly, with a decrease in the number of drops. We use this observation and automate as follows: Divide the region between min_{th} and max_{th} into a number, say k, of subregions. If the average queue size lies in region i (where region i is closer to min_{th} than region $i+1$), then the number of drops is set to $f(i)$ $(i = 1 \ldots k)$, where $f(i)$ is a monotonically increasing function of i. Figure 11 shows a plot of the performance of the self adjusting mechanism when there are 5 UDPs. In this plot k is chosen to be 8 and $f(i) = 2 \cdot i$.

[4]However, it is interesting to observe that the performance improvement from CHOKe drop 3 to CHOKe drop 4 is very small.

IV. MODELS

Due to page restrictions this section is very brief, mostly summarizing material presented elsewhere ([16]). Since tractable models for the actual CHOKe algorithm are hard to find, we introduce and study two close variants: Front and Back CHOKe. These differ from the actual algorithm in the sampling distribution; i.e. the location in the FIFO buffer from which candidates are chosen for comparison with the incoming packet. Thus Front CHOKe compares an incoming packet with the packet at the head of the buffer, while Back CHOKe compares it with the packet that arrived last. There is a slight further modification to Back CHOKe: it only drops incoming packets, but not packets from the buffer. This modification is not so much for analytical tractability (since dropping the packet from the back of the buffer is just as easy to analyze as Front CHOKe) as it is for studying aspects of the algorithm that Front CHOKe does not adequately model. The summaries are presented below. Details may be found in [16].

A. Front CHOKe

We model the system as a queue with N independent Poisson arrivals, each of rate λ_i. We assume that the service times are mutually independent, independent of the arrivals and exponentially distributed with mean $1/\mu$. The queueing discipline is first-in-first-out (FIFO). The analysis invokes the PASTA[5]property and computes the goodput obtained by flow i to equal $\frac{\mu\lambda_i}{\mu+2\lambda_i}$. The condition for the queue to be stable is: $\sum_{i=1}^{N} \frac{\lambda_i}{\mu+2\lambda_i} < 1$, which due to packet dropping is much weaker than the usual condition: $\sum_{i=1}^{N} \lambda_i < 1$.

B. Back CHOKe

In Back CHOKe, the IDs of the M ($M < N$, where N is the total number of flows) recently admitted packets are stored in memory. The ID of every incoming packet is checked against these M IDs. If there is a match, then the incoming packet is dropped. This system is a Markov chain is similar to a variant of the "Marching Soldiers Problem". The chain is shown to be reversible (for $M = 1$) or dynamically reversible (for $M > 1$) and its equilibrium distribution is computed. Various observations about its fairness properties are made[6].

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. Further work involves studying the performance of the algorithm under a wider range of parameters and network topologies, obtaining accurate theoretical models and insights, and considering hardware implementation issues.

APPENDIX

A. Fairness

We first recall the definition of max-min fairness and express it in a form that is convenient for us here. We then define min-max fairness. A proof that these two notions are equivalent is given in [16]. Throughout we consider a single link with capacity C, serving N sessions. We assume that session i requires a bandwidth of REQ_i, i = 1, 2, ... N. The problem is to allocate the bandwidth according to the following fairness criteria.

A.1 Definition: Max-min fair

An allocation $\mathbf{R} = \{R_1, R_2, ...R_N\}$ of resources is said to be *max-min* fair if it satisfies the following:

(i) If $\sum_{i=1}^{N} REQ_i \leq C$, then

$$R_i = REQ_i \quad i = 1, 2, ...N \tag{1}$$

[5]PASTA: Poisson Arrivals See Time Averages

[6]Observe that the special case $N - M = 1$ results in a perfectly fair allocation through a round robin scheme.

(ii) If $\sum_{i=1}^{N} REQ_i > C$, then

$$\sum_{i=1}^{n} R_i = C \quad , \quad R_i \leq REQ_i \quad i = 1, 2, ...N \tag{2}$$

and, if there is a session p and a bandwidth allocation $\bar{\mathbf{R}}$ such that $R_p < \bar{R}_p$, then there must be another session p' such that,

$$R_{p'} \leq R_p \quad and \quad R_{p'} > \bar{R}_{p'} \tag{3}$$

A.2 Definition: min-max fair

An allocation $\mathbf{R} = \{R_1, R_2, ...R_N\}$ is said to be *min-max* fair if it satisfies the following:

(i) If $\sum_{i=1}^{n} REQ_i \leq C$, then

$$R_i = REQ_i \quad i = 1, 2, ...N \tag{4}$$

(ii) If $\sum_{i=1}^{n} REQ_i > C$ then,

$$\sum_{i=1}^{N} R_i = C \quad , \quad R_i \leq REQ_i \quad i = 1, 2, ...N \tag{5}$$

and, if there is a session p and a bandwidth allocation $\bar{\mathbf{R}}$ with $R_p > \bar{R}_p$, then there must be another session p' such that,

$$R_{p'} \geq R_p \quad and \quad R_{p'} < \bar{R}_{p'} \tag{6}$$

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] Pan, R. and Prabhakar, B., Konstantinos Psounis, "CHOKe - A stateless active queue management scheme for approximating fair bandwidth allocation" *Stanford CSL Technical report CSL-TR-99-787*, September 1999.

[17] 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*.

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

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

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