CHOKe - A simple approach for providing Quality of Service through stateless approximation of fair queueing

Rong Pan
Balaji Prabhakar

Technical Report No.: CSL-TR-99-779

March 1999

CHOKe
- A simple approach for providing Quality of Service through stateless approximation of fair queueing

Rong Pan Balaji Prabhakar

Technical Report No.: CSL-TR-99-779

March, 1999

Computer System Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
William Gates Computer Science Building, A-408
Stanford, California 94305-9040
<pubs@shasta.stanford.edu>

Abstract

We consider the problem of providing a fair bandwidth allocation to each of n flows that share an outgoing link at a congested router. The buffer at the outgoing link is a simple FIFO, commonly shared by packets belonging to the n flows. We devise a simple packet dropping scheme, 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.

Key Words & Phrases: fair queueing, RED, active queue management, scheduling algorithm

1 Introduction

The Internet provides a connectionless, best effort, end-to-end packet service using the IP protocol. It depends on the congestion avoidance mechanisms implemented in the transport layer protocols, like TCP, to provide good services under heavy load. However, a lot of the TCP implementations do not include the congestion avoidance mechanism either by mistake or on purpose. Besides, there are a growing number of UDP-based applications running in the Internet, such as packet voice, packet video. All these flows 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 Collapse". A recent Internet draft RFC 2309 strongly recommends further research of router mechanisms to deal with unresponsive or aggressive flows [1].

As discussed in [1], there are two classes of router algorithms related to congestion control: “scheduling algorithm” and “queue management”. Several scheduling algorithms have been proposed to provide fairness in the bottleneck link bandwidth sharing. Fair Queueing (FQ), which has been widely studied, maintains separate output queues for each flow [2]. Upon congestion, a packet is dropped from the longest queue. In this scheme, packets are scheduled using an approximate bit-by-bit, round-robin discipline, which actually achieves the max-min fair bandwidth allocation. By per flow queueing, all traffic flows are essentially isolated from each other. Therefore, unresponsive flows can not degrade the performance of TCP friendly connections. However, it is well known that this approach requires complicated per flow states, which is not cost-effective to be widely deployed.

Core Stateless Fair Queueing (CSFQ), a variant of Fair Queueing, was proposed to simplify backbone router's design complexity [5]. In this method, routers are divided into two categories: edge routers and core routers. An edge router keeps per flow states and estimates each flow's arrival rate. These estimations are inserted into the packet headers and then passed to the core routers. A core router merely maintains a stateless FIFO queue and drops a packet randomly based on the rate estimation. This scheme reduces the core router’s design complexity. However, the edge router's design is still complicated. Besides, because of the rate information in the header, the core routers have to extract packet information differently from traditional routers. Stochastic Fair Queueing (SFQ), another variant of FQ, reduces fair queueing's address look up complexity [3]. Nonetheless, it requires extra data structures in order to match FQ's performance. Other scheduling algorithms, such as Class Based Queueing (CBQ), require per flow states as well [4].

While the scheduling schemes can provide a fair bandwidth allocation with complicated structures, the queue management mechanisms, although simple and easy to implement, usually fail to provide fair service. The queue management schemes, such as Drop Tail and RED, are designed to control the queue length by dropping packets when necessary. It is well known that the widely deployed Drop Tail method can cause the “Lock Out” and “Full Queue” problems [1]. RED, an active queue management algorithm, was proposed to solve these issues [6]. By keeping the average queue size small, RED avoids the bias against bursty traffic and reduces the delays experienced by most flows. RED drops the arrival packets randomly based on the average queue size. In this way, it avoids the problem of global synchronization, where many flows reduce their window size at the same time. However, like Drop Tail, RED doesn’t penalize unresponsive traffics. A flow's fraction of the aggregate packet drops roughly equals to its fraction of the aggregate arrival rate. In other words, the percentage of packets dropping for each flow over a period of time is almost the same. Consequently, misbehaving traffics could take up a large percentage of the link bandwidth and starve out the TCP friendly flows.

To improve RED's capability of handling unresponsive users, a few methods have been proposed, for instance, Flow Random Early Detection (FRED) [7] and RED with penalty box [8]. Even though these two router mechanisms do not require per-flow states, they all need extra data structures to collect certain types of state information. FRED keeps per-active-connection states and RED with penalty box stores information about unfriendly flows.

All the schemes discussed so far drop packets without examining them. A new RED mechanism called Stabilized RED (SRED) suggests an idea of comparing the arriving packet with a randomly chosen packet that recently preceded it into the buffer [9]. The scheme maintains a data structure, "Zombie List", which keeps the information regarding recently seen flows. The SRED router estimates the number of active flows based on “Zombie List”. When a packet arrives at the SRED router, it is compared against a randomly chosen packet from the “Zombie List”. The arriving packet is dropped randomly based on the packet comparison result and the estimation of the number of active flows. The dropping probability increases if the active flow estimation is bigger or packet comparison ends in matching. In this way, SRED penalizes the unfriendly flows and controls the buffer length independent of the number of active flows. However, this scheme needs a large “Zombie List” data structure.

From the discussions above, we can see that there are generally two trends in search of solutions to the problem of unfriendly flows. One trend is based on the almost perfect fair queueing. In this group, router algorithms are proposed to reduce FQ’s design complexity while keeping FQ’s good feature of max-min fair bandwidth allocation. The other trend is based on RED. Several methods are suggested to improve RED’s capability of handling misbehaving users. However, all the schemes discussed so far fail to provide fairness with a minimum overhead.

In this paper we propose a new method - the CHOKe (CHOose and Keep for responsive flows; CHOose and Kill for unresponsive flows) algorithm. CHOKe is based on the fact that the FIFO queue itself forms a statistical information about packet arrivals upon which we can use to identify bad users. The basic CHOKe scheme is to choose a victim candidate from the FIFO queue when a packet arrives, and kill both the incoming and victim candidate if they belong to the same flow. However if the victim candidate and the arriving packet are from different flows, the newly arrived packet will be admitted into the router with a probability based on average queue size and no action will be taken towards the victim packet. CHOKe is very similar to SRED in that it selectively drops the packets of unresponsive flows by making some comparison. However, CHOKe does not need any state information and does not maintain any extra data structures except the data buffers.

In the below, Section 2 explains our motivations and goals for using the CHOKe mechanism. Section 3 describes the CHOKe algorithm in detail. The simulation results are presents in section 4. Instead of choosing only one victim candidate, CHOKe can be modified to select multiple packets from the FIFO queue. All the victim packets which have the same flow ID as that of the arrival packet are dropped. The effect of this multiple drops is studied in section 5 in the context of multiple unfriendly flows, followed by discussions and conclusions in section 6.

2 Motivations and Goals

Our work is motivated by the lack of a simple, stateless router algorithm that can achieve fair bandwidth allocation and flow isolation. We assume that packets are queued into different FIFO queues according to its priority and there is a general priority based scheduler in the router to schedule the packets. The queue management algorithm is proposed to provide fair service to the flows that share one FIFO queue. The new scheme should give the maximum penalty to unresponsive flows and prevent TCP friendly flows from being sacrificed. As a result, it minimizes the maximum bandwidth that unresponsive flows can take and boosts up the well-behaving flow's throughput. We want our scheme to keep RED's advantages such as no bias against bursty traffic, avoidance of global synchronization and lower-delay interactive service. It is not our intention to achieve the almost perfect fairness obtained by Fair Queueing. We would like to provide fairness with a lowest cost. By fairness, we mean minimizing the maximum bandwidth that unresponsive flows can take, instead of the max-min fairness achieved by the Fair Queueing. Overall, our goal of implementing the CHOKe algorithm can be itemized as the following:

1) min-max fair bandwidth allocation;

2) protection from aggressive sources;

3) no bias against bursty traffic;

4) no global synchronization by deploying this algorithm;

5) low latency for interactive applications;

6) minimum cost or implementation overhead.

3 Basic CHOKe algorithm

This section explains the basic algorithm used in the CHOKe gateways. As RED, it calculates the average queue size using an exponential moving average. The average queue size is compared against two thresholds, a minimum and a maximum threshold. If the average queue size is less than the minimum threshold, 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 the minimum level and no packets are dropped. If the average queue size is greater than the maximum threshold, every arriving packet is dropped. This step moves the queue occupancy back to below the maximum threshold.

When the average queue size is between the minimum and maximum threshold, each arrival packet is compared to a randomly selected packet from the FIFO queue. If they have the different flow ID, the newly arrived packet is dropped with the probability based on the average queue size. The dropping probability function is chosen to be the same as the one used in RED as well. However, if both packets have the same flow ID, both of them are dropped. The main idea here is to admit a new packet into the FIFO queue based on both the queue distribution and average queue size.

The algorithm is better illustrated by the simple example below. Let's assume several properly implemented TCP flows share a bottleneck link with a congestion-unaware UDP flow. We also assume all flows begin transmitting at the same time. The TCP flows start off with a large window size and the UDP flow has a constant bit rate which is almost the same as the link capacity.

Initially, the FIFO queue in the CHOKe gateway has a size zero. The arriving packets are queued into the buffer without drops. Soon, average queue size goes up and over the minimum threshold since all flows send out packets aggressively. Then the incoming packets start to be dropped and, associated with them, some packets are dropped from the queue as well. At this moment, each flow takes up a certain amount of space in the buffer and no flow has much more packets in the queue than the other flows. So when a packet is chosen from the queue, the chance of it having the same flow ID as the incoming packet is small. Therefore, we argue that the algorithm behaves mostly like RED up to this point.

Then, the TCP flows start to throttle back because of packets dropping. But the UDP flow still persistently send data at the same speed. As a result, traffic arrival distribution is different from what it was before: the UDP flow makes a greater fraction of the total incoming traffic compared to any other flow. This affects the queue distribution among all the flows. The major difference between RED and CHOKe is the way they handle this situation, where the arrival rate and queue occupancy are not equally distributed among all the flows.

RED discards packets based on the average queue size only. When a packet arrives at a queue whose size is between the minimum and maximum threshold, it has a probability of being dropped. But the probability is independent of the flow type. In other words, all flows get the same probability of dropping at any time window. Even though a flow with more arrivals has more drops as well, the ratio between a flow's total packet drops to its total packet arrivals stays roughly the same among all flows.

Unlike RED, CHOKe penalizes the ill-behaving users when the incoming traffic rate and queue occupancy are not well balanced among all the flows. In the example discussed above, a UDP packet can be easily caught as a victim candidate when the UDP connection uses more network resources. The action of dropping both the incoming packet and the victim not only lowers the admission rate of the UDP flow but also reduce its queue occupancy. Besides, the flows with higher arrival rate have a higher probability of dropping simply because they trigger the CHOKe algorithm more often. This process continues until the queue is well balanced again. As a consequence, there is no flow can use up a major fraction the bottleneck link bandwidth. From TCP flow's point of view, they are favored by the CHOKe gateway. First, the probability of being chosen as a victim candidate is small. Second, given being caught, it is given a second chance because it can be left intact if the flow ID comparison ends in mismatching. Because the UDP flows don't back off in the case of congestion, the probability is very high that those TCP victim candidates are left unharmed. Last, since CHOKe is invoked upon a packet arrival, a flow with lower arrival rate has a smaller drop rate as well. A TCP flow packet, therefore, has less chance of being dropped.

From the discussions above, we can see that CHOKe achieves the goals: 1)min-max fair bandwidth allocation and 2)protection from aggressive sources. How CHOKe achieves the other goals will be shown below.

As RED, CHOKe estimates the average size by using an exponential moving average. By keeping the average queue size low, the delay experienced by the users is lower and a burst of data can beadmitted into the buffer without dropping. Also a packet is dropped randomly in CHOKe so there is no global synchronization, where a lot of connections decrease or increase their windows at the same time.

It is obvious that CHOKe is a truly stateless algorithm, which does not require any data management structure like all the other algorithms discussed do. Compared to a pure FIFO queue, there are just a few extra operations that CHOKe have to do: drawing a packet randomly from a queue, comparing the flow ID and dropping both the incoming and victim packets. Flow ID comparison can be easily done in hardware. Drawing a packet can be implemented as generating a random address at which a packet flow ID is read out. The operation of victim packet dropping is arguably more difficult because a packet has to be removed from a link-list queue. Instead of breaking the link list, we propose to add one extra bit in the link-listed packet header. The bit is set to one when the victim candidate is decided to be dropped. Depending on the status of this bit, when a packet moves up to the queue front, it is either discarded or sent out. In this way, CHOKe can be implemented using minimum overhead and therefore goal 6) is accomplished.

4 Simulation Result

In this section, we present our simulation results, which show that CHOKe can successfully provide fair bandwidth allocation and protect TCP friendly flows. RED and Drop Tail schemes, whose complexities are similar to that of CHOKe, are used as the comparisons. It is not compared against Fair Queueing, which achieves the ideal fairness by implementing the expensive, complicated per flow queueing.

The network configuration used in our simulations is shown in Figure 1. There are m TCP sources and n UDP sources sending packets. Accordingly, there are m TCP sinks and n UDP sinks receiving data. S(i) sends packets to R(i) where i = 1, 2, 3... etc. In return, R(i) sends back acknowledgments if it is a TCP sink. A UDP sink, however, simply absorbs the data without sending packets back. K1 and K2 are gateways in the network. The link between them has a capacity of 1Mbps, which is the bottleneck link. Every host is connected to a router by a 10Mbps link. FTP sessions are assumed as TCP traffics. The window size of a TCP traffic is set to be 100. All packets are set to have the size of 1K Bytes. The UDP hosts send packets at a constant bit rate (CBR) of r Kbps, where r is a variable.

Figure 1. Network Configuration:
m TCP sources, n UDP sources

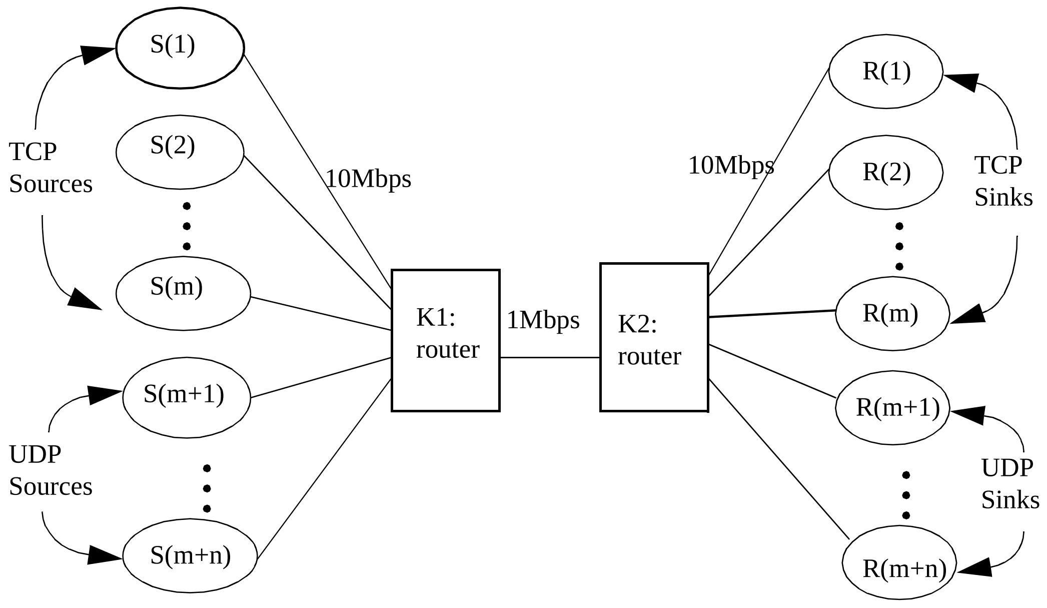


Figure 2. Drop Tail's Throughput Comparison
(32 TCP sources, 1 UDP source)

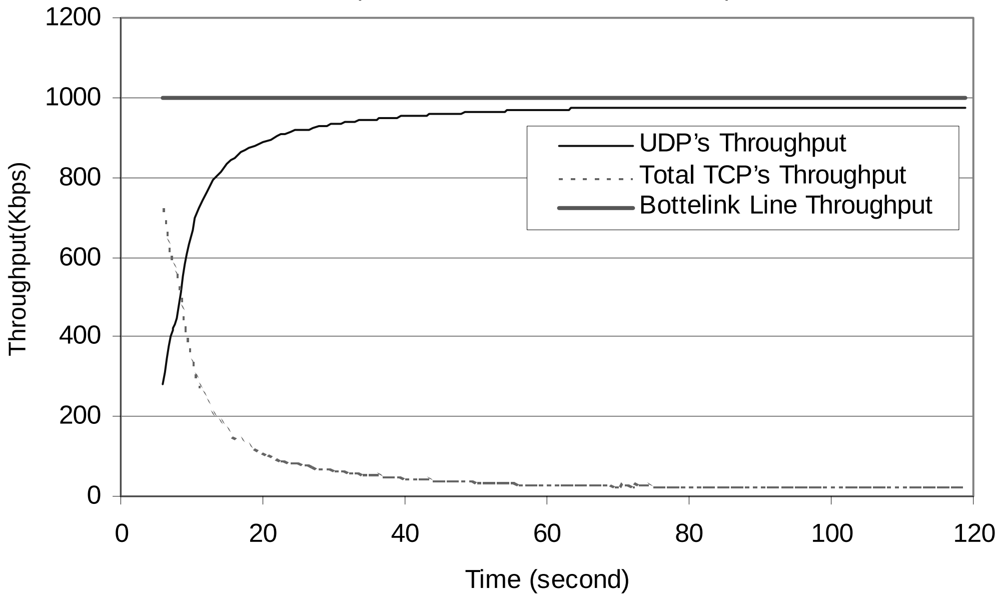


Figure 2 to Figure 4 show the throughputs of the TCP and UDP flows using different router algorithms: Drop-Tail, RED and CHOKe. There are m = 32 TCP sources (Flow #1 to Flow #32) and n = 1 UDP source (Flow #33) in the simulation. The UDP source sends packets at a rate of r = 2 Mbps, which is twice the bandwidth of the bottleneck link.

Figure 3. RED's Throughput Decomposition
(32 TCP sources, 1 UDP source)

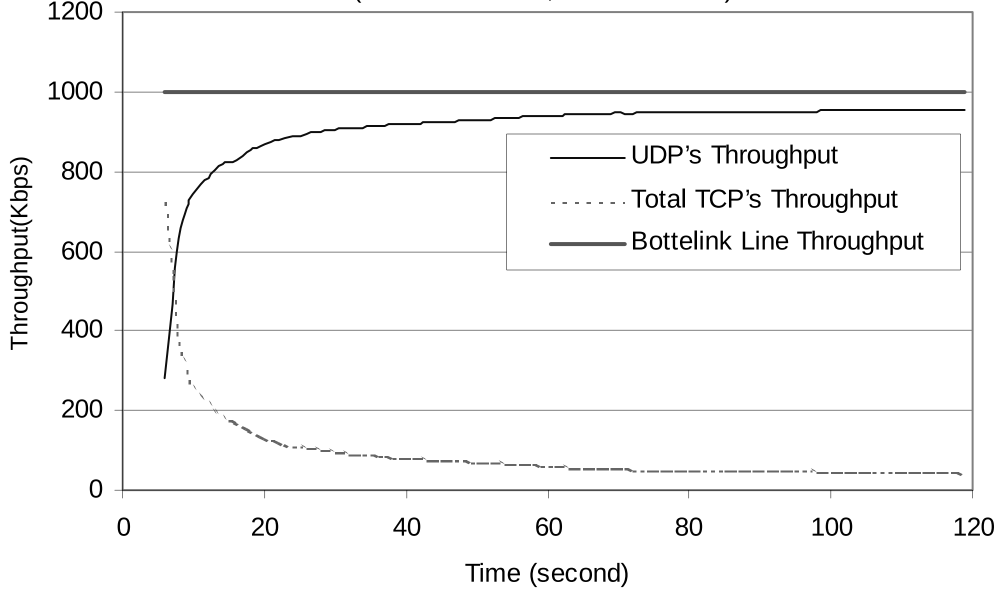
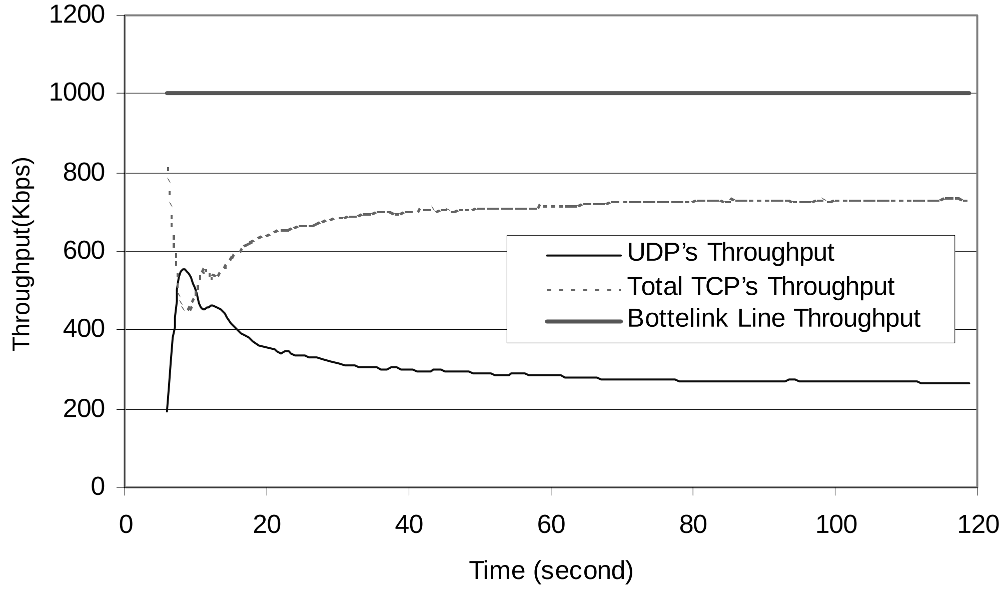


Figure 4. CHOKe's Throughput Decomposition
(32 TCP sources, 1 UDP source)



From these figures, we can clearly see that the RED and Drop-Tail gateways can't provide fair bandwidth allocation. 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. As shown in Figure 4, the UDP flow's throughput at steady state is 250 Kbps, which is only 25% of the link capacity. The throughout of per TCP flow, however, is boosted from 1.6 Kbps to 23.4 Kbps.

The individual throughputs of the 33 connections (32 TCP flows + 1 UDP flow) are plotted in Figure 5 to Figure 7. Y axis is drawn in Log scale. As shown in Figure 7, although the UDP flow's throughput is still higher than the rest TCP flows, CHOKe provides much better service to the TCP flows by simply dropping both the incoming and the victim packet when they have the same flow ID.

Figure 5. Drop Tail: Throughput per Flow

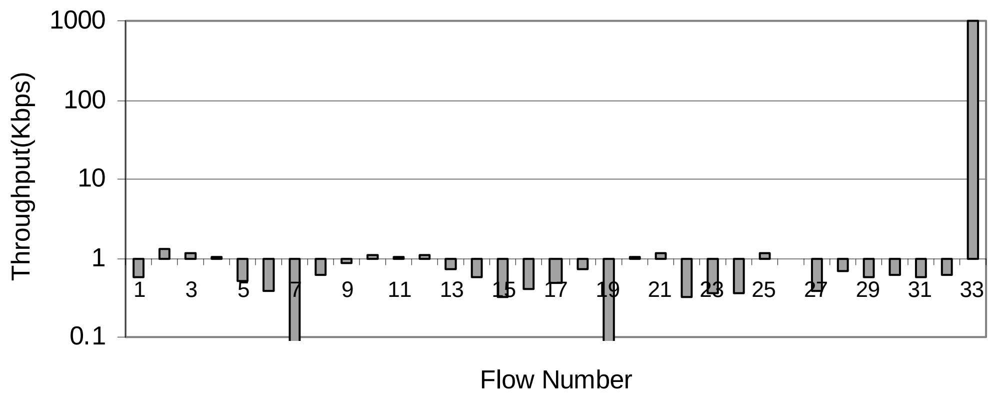


Figure 6. RED: Throughput per Flow

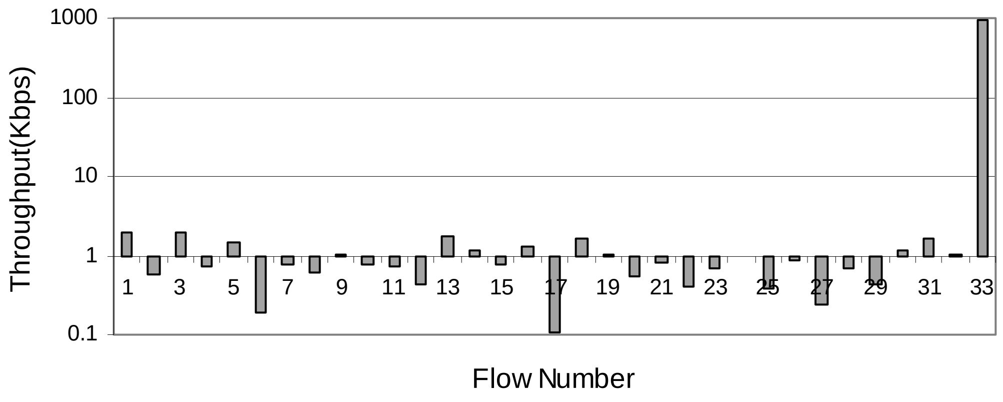


Figure 7. CHOKe: Throughput per Flow

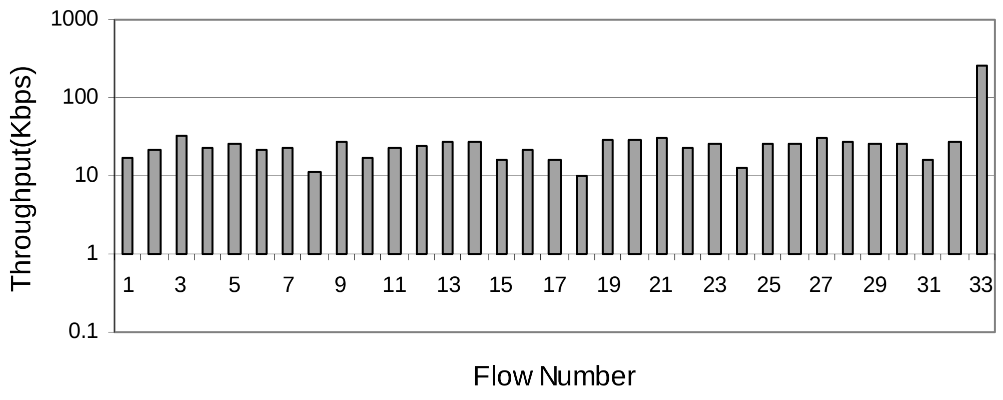
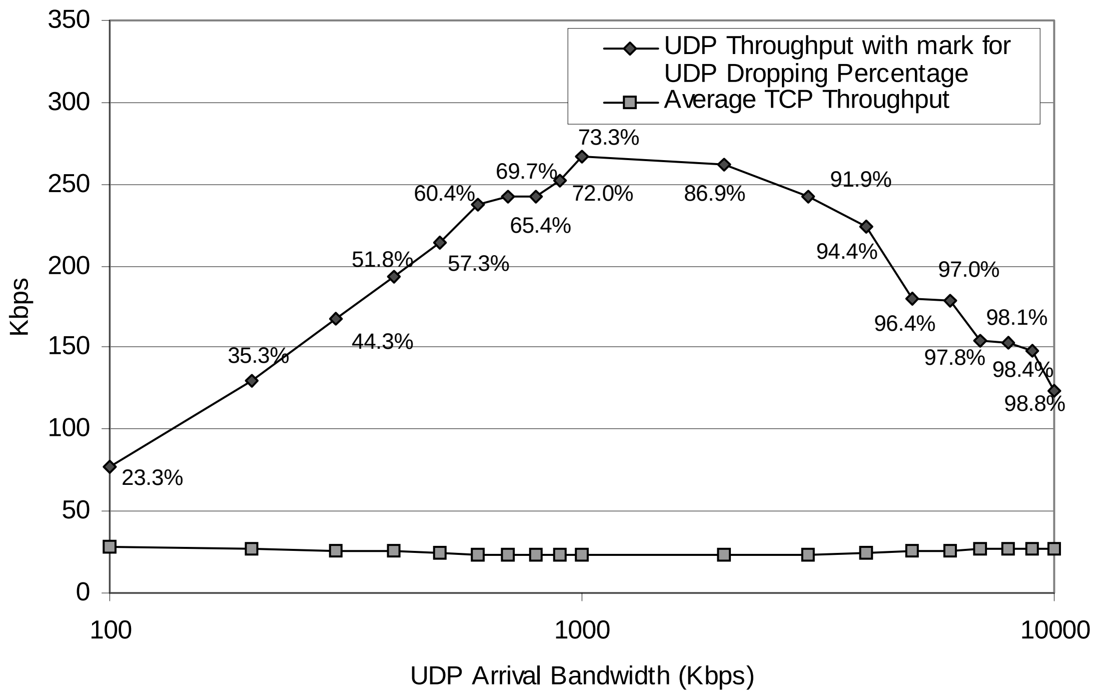


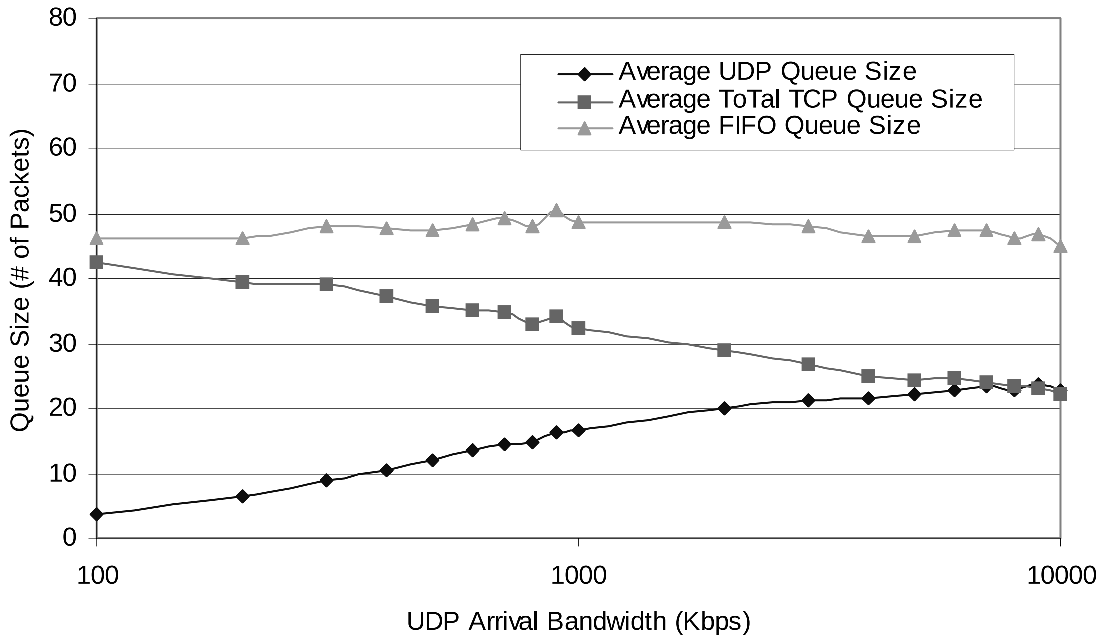
Figure 8. CHOKe's Performance Under Different Traffic Load
(32 TCP sources, 1 UDP source)



The CHOKe's performance under different traffic load conditions is illustrated in Figure 8, where the flow throughputs versus the UDP flow arrival rate are plotted. The drop percentage of the UDP flow is also shown in Figure 8. From the plot, we can see that CHOKe drops 23% of the UDP packets when its arrival rate is as low as 100Kbps. As the UDP arrival rate increases, the drop percentage goes up as well. It drops almost all of the packets (99%) when the arrival rate reaches 10Mbps. The average TCP flow's throughput stays almost constant.

Figure 9 shows the queue distribution among the flows for different traffic load conditions. The minimum and maximum threshold are set to 30 and 60 packets. It is obvious that CHOKe can control the average queue size as RED does. Note that when the UDP arrival rate is at 100Kbps, it is only a few times the rate of a single TCP flow and one tenth of the total TCP arrival rate. However, CHOKe is able to detect this small difference and drops 23% of the UDP traffic. With the UDP arrival rate goes up, its share of the queue occupation increases. Because of the aggressive dropping scheme that CHOKe adopts, the average of queue size of the UDP flow can never become the dominant portion of the queue usage.

Figure 9. Queue Distribution for Different Traffic Load



As a comparison, RED's performance under different traffic load is shown in Figure 10 and 11. It is obvious that RED can not provide protection against greedy connections. The unresponsive flows use up all the network bandwidth and start out the well-behaving flows. Also, RED becomes a Drop Tail scheme under heavy load since the queue size goes above maximum threshold. All the arrival packets are dropped.

Figure 10. RED's Performance Under Different Traffic Load
(32 TCP sources, 1 UDP source)

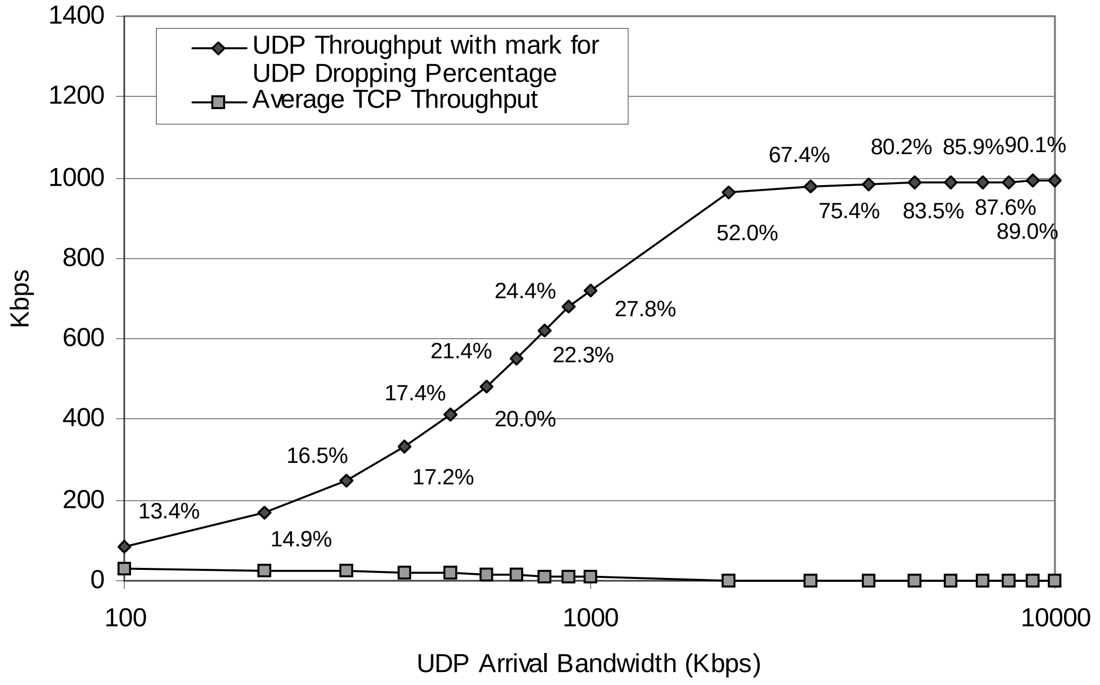
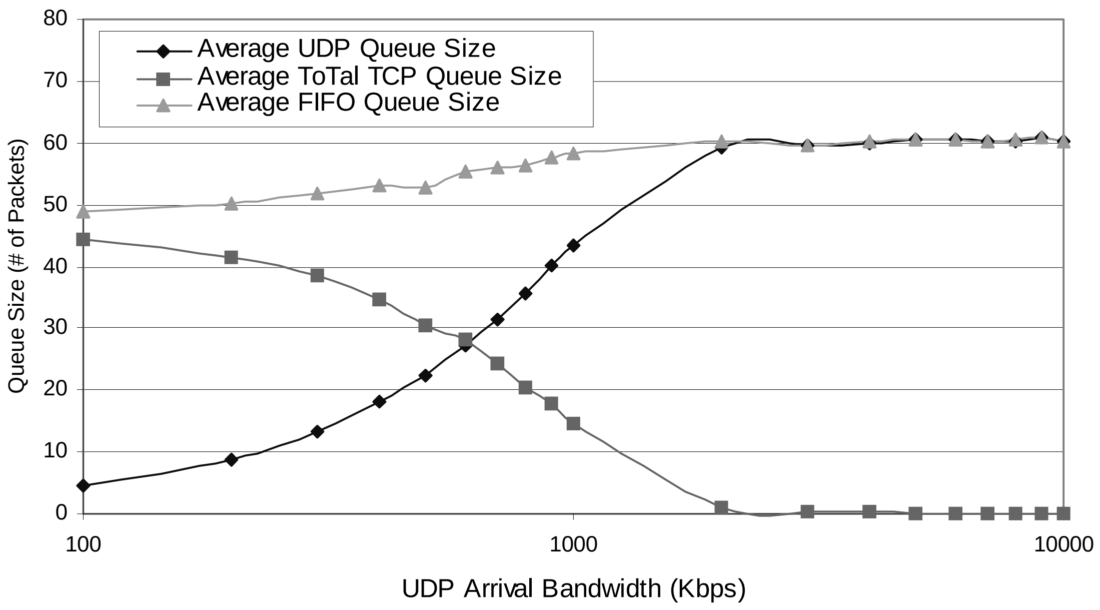


Figure 11. Queue Distribution for Different Traffic Load
(32 TCP sources, 1 UDP source)



5 CHOKe with Multiple Drops

To study the effect of multiple drops, we modified the basic CHOKe so that multiple victim candidates are dawn from the FIFO queue. The dropping disciplines are as the following:

1) Multiple victim candidates are drawn from the FIFO queue;

2) Any victim candidate whose flow ID is the same as that of the incoming packet is killed;

3) As long as there is one victim candidate dropped, the arriving packet is dropped as well. The incoming candidate is enqueued into the FIFO queue only if none of the victim candidates has its flow ID.

Figure 12-13 show the performance of the CHOKe algorithm with two and three victim candidates. The network configuration for the simulation is the same as the one in Figure 4 (32 TCP sources share a 1 Mbps bottleneck link with 1 UDP source. The CBR rate for the UDP source is at 2Mbps). From these two figures, we can see that CHOKe with multiple candidates can improve the basic CHOKe scheme's performance. However, the performance difference between CHOKe with two or three victim candidates is hardly noticeable. So CHOKe with two victim candidates seems to be the optimal cost-effective solution. Since CHOKe with m-1 victim candidates has a maximum drop of m packets (m-1 victim packets + 1 incoming packet), it will be referred to as CHOKe with drop m. As a comparison to the basic CHOKe scheme, the performance of CHOKe with drop 3, under different traffic load, is illustrated in Figure 15. The corresponding queue distribution is depicted in Figure 14.

Figure 12. CHOKe with 2 victim candidates: Throughput Decomposition
(32 TCP sources, 1 UDP source)

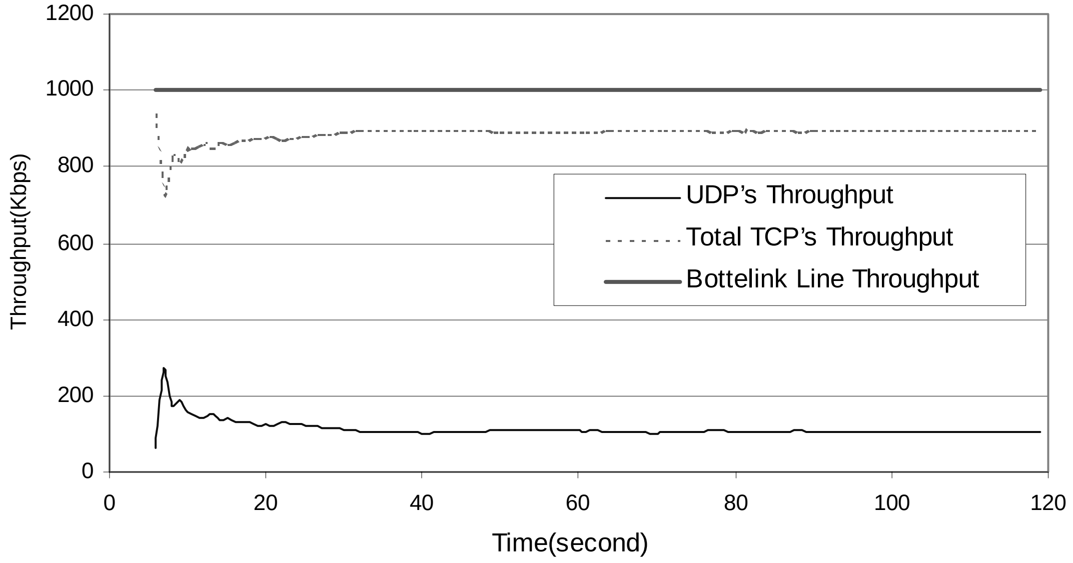


Figure 13. CHOKe with 3 victim candidates: Throughput Decomposition
(32 TCP sources, 1 UDP source)

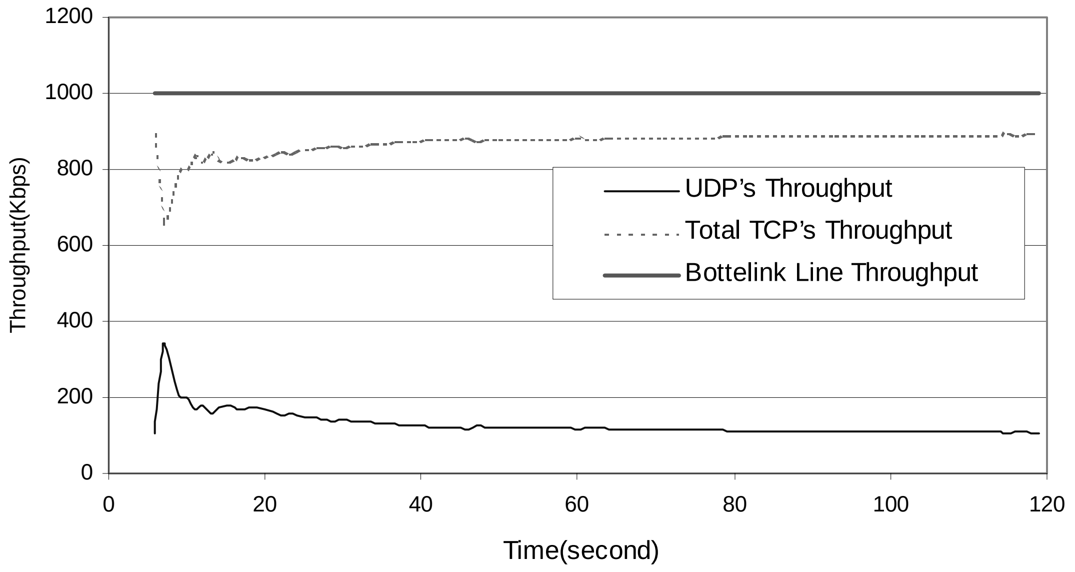


Figure 14. CHOKe with Drop 3: Queue Distribution for Different Traffic Load

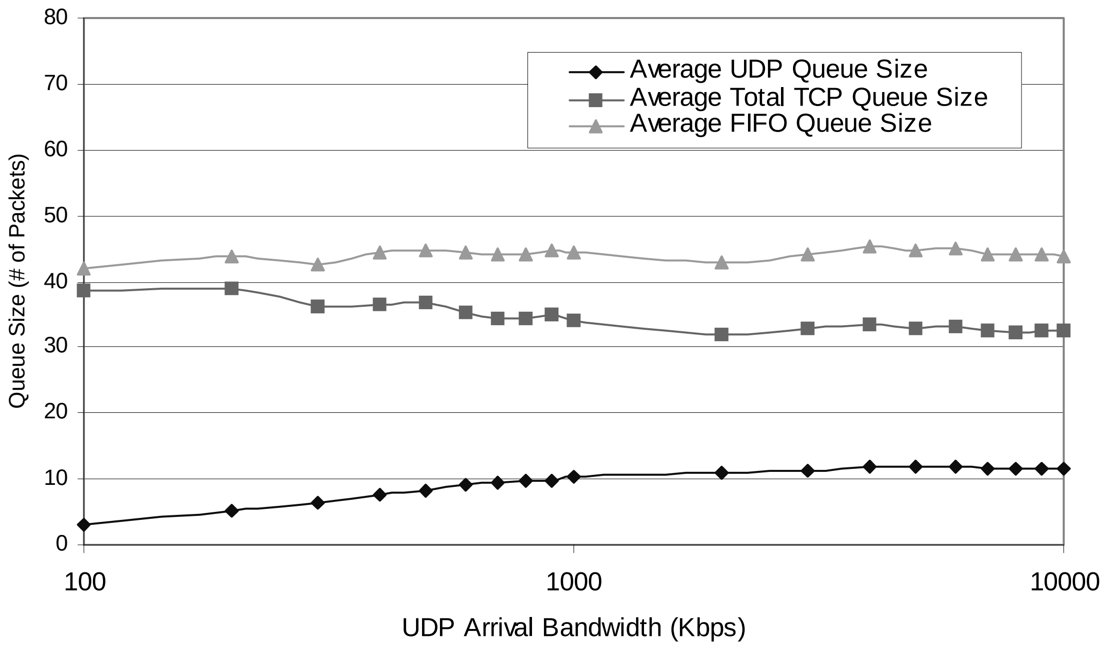
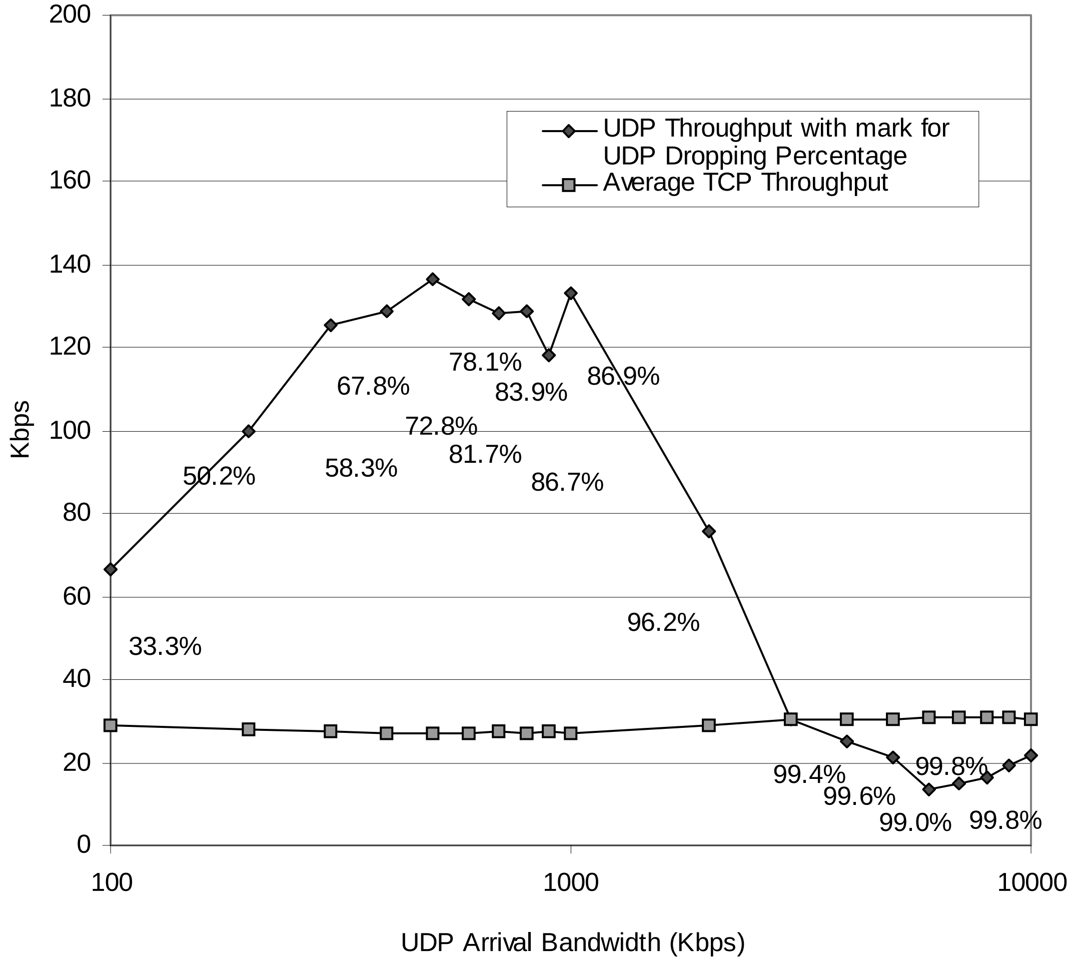


Figure 15. CHOKe with drop 3:
Performance Under Different Traffic Load (32 TCP sources, 1 UDP source)



As expected, because of its more aggressive dropping scheme, CHOKe with drop 3 has a better control over the unresponsive UDP traffic than the basic CHOKe algorithm. Besides, average queue size is kept smaller so that traffics experience less delay. Yet, we have to notice that the performance improvement, brought by the multiple drops, is not radical in this case, around 10-15%.

When there are many UDP flows in the network, CHOKe with multiple drops exhibits its advantage over the basic algorithm. A simulation configuration is set up where there are 32 TCP sources and 5 UDP sources. All the UDP sources are assumed to have the same arrival rate. The minimum and maximum queue thresholds are still set up to be 30 and 60 packets. The simulation results for the basic CHOKe algorithm are given in Figure 16 and 17. As shown in Figure 16, the throughput of the UDP sources goes up monotonically with their arrival rate. As a result, there is almost no bandwidth left for TCP sources. From Figure 17, we can see that, 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.

Figure 16. Basic CHOKe Scheme: Performance Comparison
(32 TCP sources and 5 UDP sources)

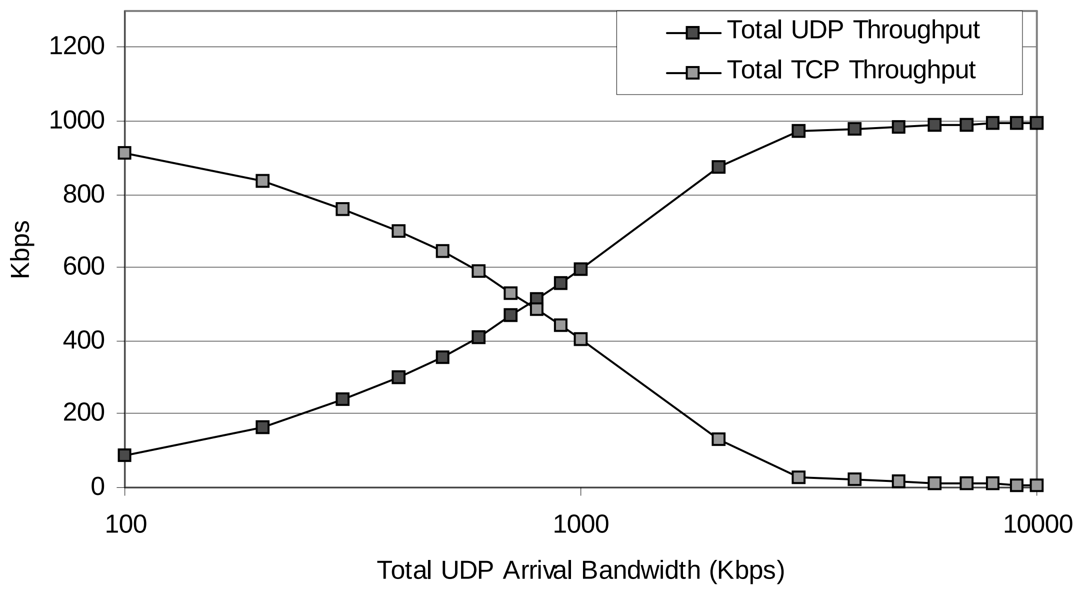


Figure 17. Basic CHOKe Scheme: Queue Distribution
(32 TCP sources, 5 UDP sources)

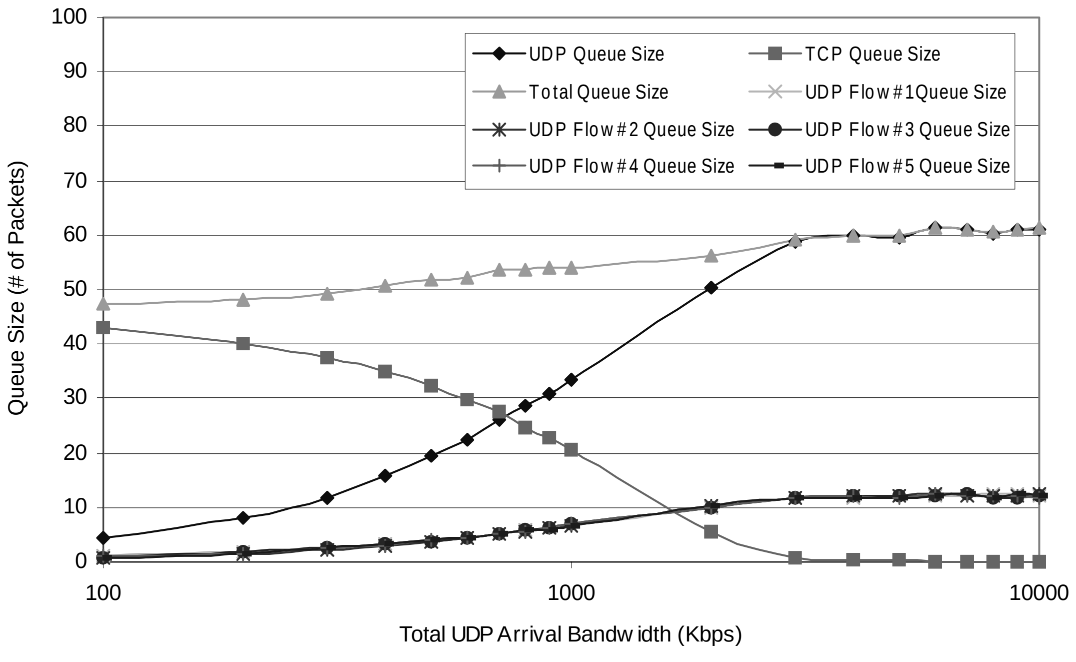


Figure 18. CHOKe with Drop 6: Performance Comparison
(32 TCP sources and 5 UDP sources)

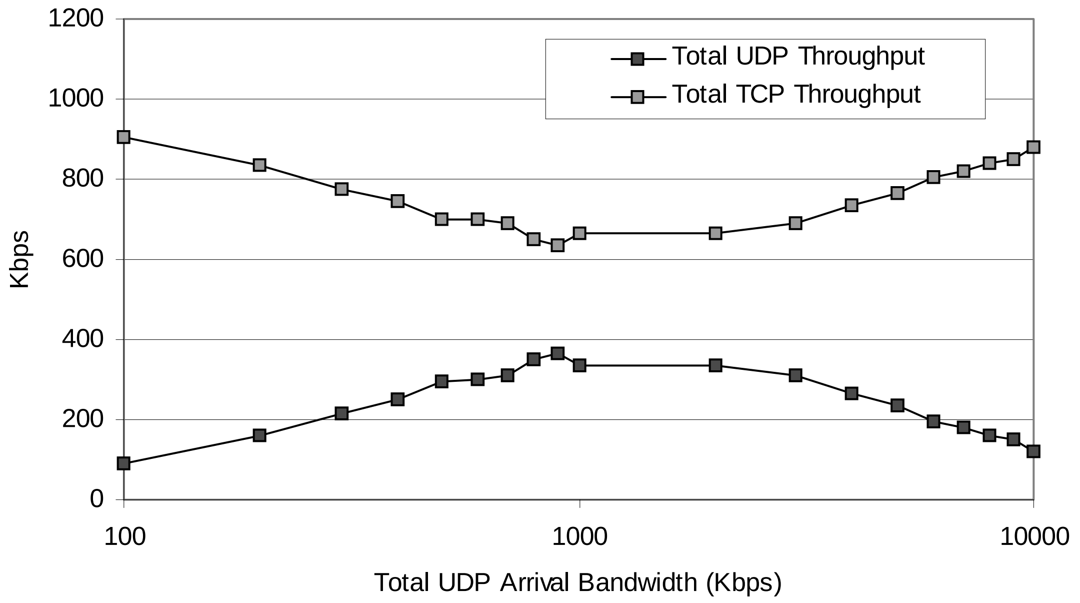
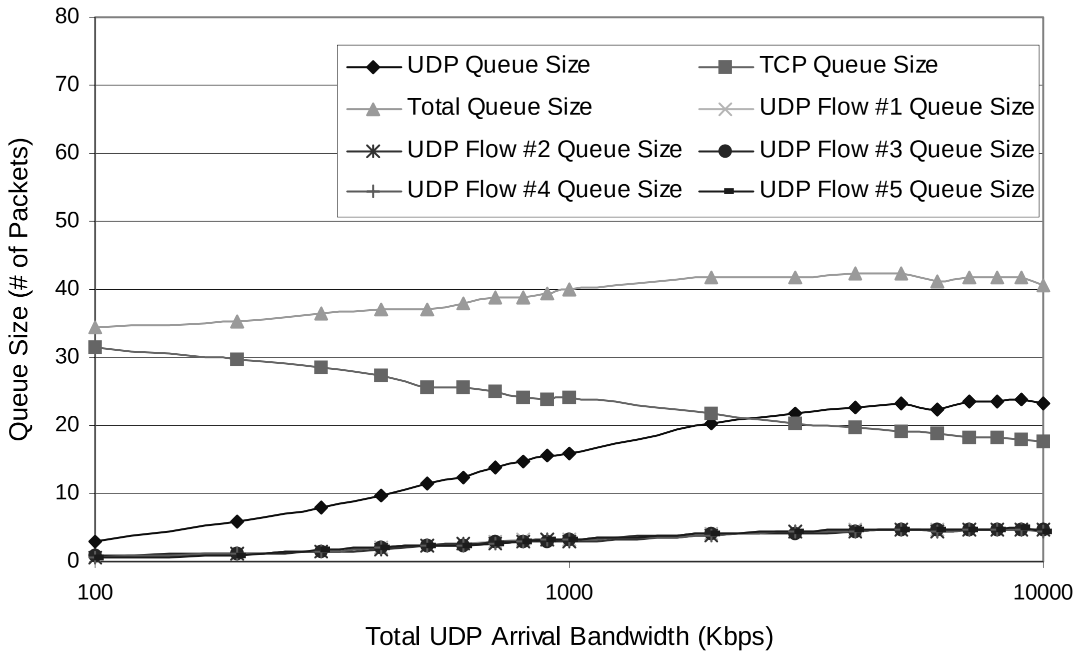


Figure 19. CHOKe with Drop 6: Queue Distribution
(32 TCP sources, 5 UDP sources)



CHOKe with multiple drops reinforce the TCP flows' throughput by its vigorous dropping method, as depicted in Figure 18 and 19. Because multiple victim candidates are selected from the queue, the chance of catching the bad flows increases. Therefore, the CHOKe with multiple drops can penalize those flows that are hard to detect but use more than their fair share network bandwidth.

6 Discussions and Conclusions

As discussed in section 5, the basic CHOKe scheme only works well with a small number of unresponsive flows. When there are more ill-behaving users, the number of queue packet drops has to be increased correspondingly in order to achieve our goals. Therefore, the implementation gets complicated. A new implementation is being worked on to reduce the design complexity.

In summary, we have proposed a new congestion control scheme, CHOKe, which can limit the throughput of misbehaving users with a minimum overhead. It uses the statistical information that exists in the packet buffers to identify and penalize bad flows. When there is no unresponsive flow at the gateway, CHOKe behaves just like RED. It maintains all the good features that RED has.

7 Acknowledgment

Balaji Prabhakar thanks David Clark of MIT for several conversations in Spring 98 that led to this work.

8 Reference

[1] Braden, B. etc, “Recommendations on queue management and congestion avoidance in the internet”, *IETF RFC* (Informational) 2309, April 1998.

[2] 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.

[3] McKenny, P., “Stochastic Fairness Queueing”, *Proceedings of INFOCOM'90*, pp 733-740.

[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] 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.*

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

[7] Lin, D. and Morris, R., “Dynamics of random early detection”, *Proceedings of ACM SIGCOMM’97*, pp 127-137, Oct. 1997.

[8] Floyd, S., and Fall, K., “Router Mechanisms to Support End-to-End Congestion Control”, LBL Technical report, February 1997.

[9] Ott, T., Lakshman, T. and Wong, L., “SRED: Stabilized RED”, to appear in *Proceedings of INFOCOM’99*, March. 1999.

[10] Floyd, S., and Fall, K., “Promoting the Use of End-to-End Congestion Control in the Internet”, February 1998. Under submission to *IEEE/ACM Transactions on Networking*.

[11] Floyd, S., Fall, K.and Tieu, K., “Estimating Arrival Rates from the RED Packet Drop History”, April 1998.

[12] ns-Network Simulator (Version 2.0), http://www-nrg.ee.lbl.gov/ns/#version2.