Approximate Fair Dropping for Variable-Length Packets

The CHOKe router algorithm provides an approximately fair bandwidth allocation at a low implementation cost. The authors consider performance and implementation issues associated with CHOKe, with particular emphasis on variable-length packets.

Konstantinos Psounis
Rong Pan
Balaji Prabhakar
Stanford University

To provide high-quality service under heavy user loads, the Internet depends on congestion avoidance mechanisms implemented in the transport-layer such as the transmission-control protocol (TCP). However, many TCP implementations don't include—either deliberately or by accident—a congestion avoidance mechanism. Moreover, a growing number of user datagram protocol (UDP)-based applications running on the Internet don't back off properly when they receive congestion indications. As a result, these applications aggressively use more bandwidth than other TCP-compatible flows. Therefore, it's necessary to have router mechanisms to shield responsive flows from unresponsive or aggressive flows and to provide good quality of service.[1]

All of the router algorithms (scheduling and queue management) discussed in the "Router algorithms" box have either provided fairness or were simple to implement, but not both simultaneously. In this article, we explore the CHOKe[2] (choose and keep for responsive flows, choose and kill for unresponsive flows) algorithm, which combines fairness and simplicity, and we address approximating byte-by-byte fairness and implementation issues of the algorithm.

Background

CHOKe uses the observation that the FIFO-buffer contents form sufficient statistics about incoming traffic to penalize misbehaving flows in a simple fashion. The state, taken to be the number of active flows and the flow identification of each of the packets, is assumed to be unknown to the algorithm. The only observable data for the algorithm is the total occupancy of the buffer.

Specifically, CHOKe calculates the average occupancy of the FIFO buffer using an exponential moving average window exactly as in the random early detection (RED) algorithm.[3] It marks two thresholds on the buffer, a minimum threshold, min_{th} and a maximum threshold, max_{th}. When the average queue size is less than min_{th}, every arriving packet is queued into the FIFO buffer. When the average queue size is larger than min_{th}, each arriving packet is compared with a randomly selected packet, called a drop candidate packet, from the FIFO buffer. Packets with the same flow identification are both dropped; otherwise, the randomly chosen packet remains in the buffer, and the arriving packet is dropped with a probability dependent on queue size. The drop probability is computed exactly as in RED. In particular, when the average queue size is greater than

0272-1732/01/$10.00 © 2001 IEEE

Router algorithms

The two types of router algorithms for congestion control are scheduling and queue management.[1] The well-known fair queuing (FQ) algorithm exemplifies the scheduling algorithm class. FQ requires partitioning the buffer at each router output into separate queues, each of which buffers each flows' packets.[2] Because of per-flow queuing, packets belonging to different flows are essentially isolated from each other, and one flow cannot degrade the quality of another. However, it's well known by researchers 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, the recently proposed scheduling algorithm called core stateless FQ[3] divides routers into two categories: edge and core. 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. Even though this scheme reduces the core router's design complexity, the edge router's design is still complicated. Also, core routers have to extract packet information differently from traditional routers, which further increases the complexity of the scheme.

Another notable scheme, which aims to approximate FQ at a smaller implementation cost, is stochastic fair queuing.[4] 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 approximately 1,000 to 2,000 queues in a typical router to approach FQ's performance.[5]

Scheduling algorithms can allocate fairly, but they are often too complex for high-speed implementations and don't scale well to a large number of users. Conversely, queue management algorithms are usually simple. Given their simplicity, the hope is to approximate fairness. Random early detection (RED)[6] exemplifies this class of algorithms. A router implementing RED maintains a single FIFO shared by all the flows and drops an arriving packet at random during periods of congestion. The drop probability increases with the congestion level. By keeping the average queue size small, RED reduces the delays experienced by most flows. However, RED can't penalize unresponsive flows.

To improve RED's ability for penalizing unresponsive users, a few variants—such as RED with penalty box[7] and flow random early drop[8]—have been proposed. However, these variants incur extra implementation overhead since they collect certain types of state information. Another interesting variant is stabilized RED (SRED).[9] SRED evens out the occupancy of the FIFO buffer independently of the number of active flows. More interestingly, it estimates the number of active connections and finds candidates for misbehaving flows. It achieves this by maintaining a data structure, called the Zombie list, which serves as a proxy for information about recent flows. Although SRED identifies misbehaving flows, it doesn't propose a simple router mechanism for penalizing misbehaving flows.

References

1. B. Braden et al., "Recommendations on Queue Management and Congestion Avoidance in the Internet," *Internet Engineering Task Force* (*IETF) RFC (Informational) 2309*, Apr. 1998; http://www.rfc-editor.org/rfc.html.
2. A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queuing Algorithm," *J. Internetworking Research and Experience*, pp. 3-26, Oct. 1990. Also *Proc. ACM Sigcomm*, ACM, New York, 1989, pp. 3-12.
3. I. Stoica, S. Shenker, and H. Zhang, "Core-Stateless Fair Queuing: Achieving Approximately Fair Bandwidth Allocations in High Speed Networks," *Proc. ACM Sigcomm*, ACM, New York, 1998.
4. P. McKenny, "Stochastic Fairness Queuing," *Proc. IEEE Infocom*, IEEE Press, Piscataway, N.J., 1990, pp. 733-740.
5. A. Manin and K. Ramakrishnan, "Gateway Congestion Control Survey," *IETF RFC (Informational) 1254*, Aug. 1991; http://www.rfc-editor.org/rfc.html.
6. S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," *IEEE/ACM Trans. Networking*, vol. 1, no. 4, 1993, pp. 397-413.
7. S. Floyd and K. Fall, "Router Mechanisms to Support End-to-End Congestion Control," Lawrence Berkeley Laboratories tech. report, 1997.
8. D. Lin and R. Morris, "Dynamics of Random Early Detection," *Proc. ACM Sigcomm*, ACM, New York, 1997, pp. 127-137.
9. T. Ott, T. Lakshman, and L. Wong, "SRED: Stabilized RED," *Proc. IEEE Infocom*, IEEE Press, Piscataway, N.J., 1999, pp. 1346-1355.

max_{th}, every arriving packet is dropped. This moves the queue occupancy back to below max_{th}. Figure 1 (next page) shows a flow chart of the algorithm.

An intuitive reason for why this scheme penalizes unresponsive flows is that the FIFO buffer is more likely to have packets belonging to a misbehaving flow, and hence, the algorithm is more likely to choose these packets 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 causes dropping of packets belonging to misbehaving flows. Therefore, packets from misbehaving flows are dropped more often than packets from well-behaved flows.

In general, the algorithm can choose $m > 1$ packets from the buffer, compare all of them

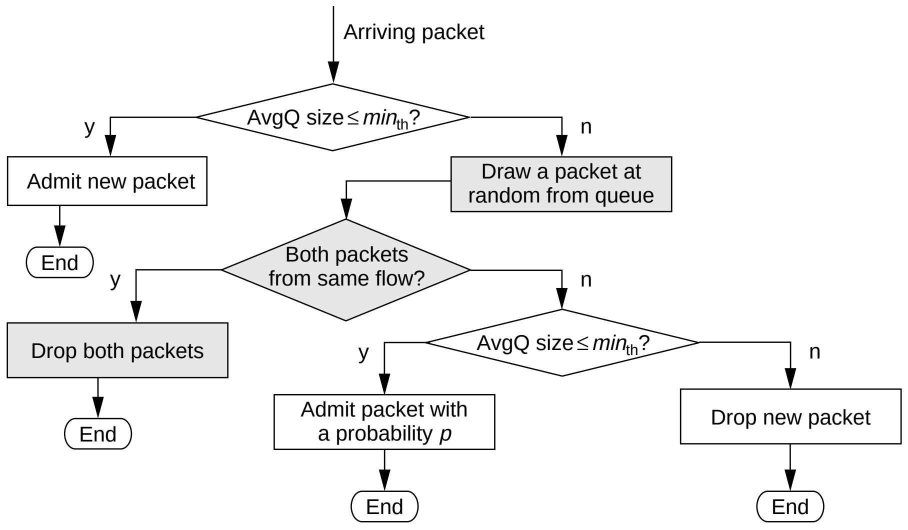


Figure 1. The CHOKe algorithm flow chart. Nonshaded boxes represents the RED algorithm.

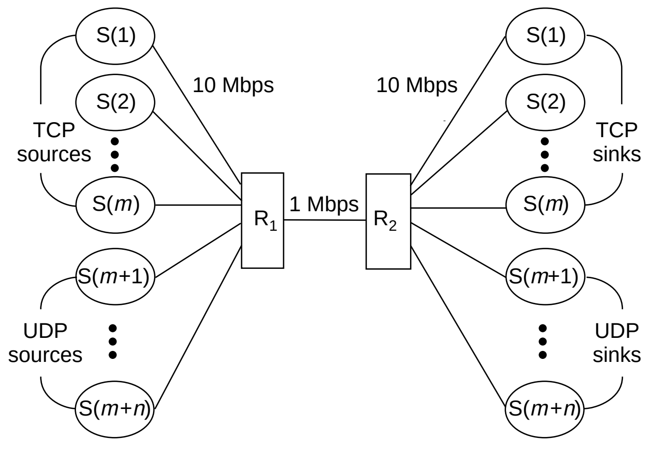


Figure 2. Network configuration: m = TCP sources, n = UDP sources.

with the incoming packet, and drop the packets that have the same flow identification as the incoming packet. Not surprisingly, 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's necessary to choose more candidate packets for dropping. One way to automate the process so that the algorithm chooses the proper value of $m \pm 1$ 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 $m = 2$. More generally, we can introduce multiple thresholds that partition the interval between min_{th} and max_{th} into k regions R_1, R_2, …, R_k and choose different values of m, depending on the region in which the average buffer occupancy falls. Obviously, we need to let m increase monotonically with the average queue size.

Constant- and variable-length packets

Simulation results of CHOKe's performance in penalizing misbehaving flows enable approximating fair bandwidth allocation, under both constant- and variable-length packets.

Constant-length packets

Figure 2 illustrates the network configuration of CHOKe's performance when there is a single congested link. We simulate a particular scenario where the 1-Mbps link between routers R_1 and R_2 is shared by 1-UDP and 32-TCP sources. The UDP source sends packets at a rate of 2,000 Kbps. Using the Network Simulator Version 2.0,[4] we plot in Figure 3 the throughput of the UDP flow under different router algorithms: DropTail (packets are accepted at the queue if the queue is not full, and dropped otherwise), RED, and CHOKe. It's evident that while both DropTail and RED are unable to prevent the UDP source from consuming up to 95% of the link capacity, CHOKe limits the UDP source's throughput to about 25% of the link capacity. This prevents the TCP sources from being shut out by the UDP source.

When there are many UDP flows in the network, CHOKe approximates fairness by drawing more than one sample from the queue. We set up a simulation configuration with 32-TCP and 5-UDP sources using the basic network topology shown in Figure 2. All UDP sources are assumed to have the same arrival rate, which varies simultaneously from 100 Kbps to 10,000 Kbps. Figure 4 gives the simulation results for the CHOKe algorithm.

In this simulation we used the automated process previously described to decide on the number of samples to draw. In particular, we divided the region between min_{th} and max_{th} into three subregions and the number of samples drawn in a region is set to $2 \times i - 1$ ($i = 1, 2, 3$). By drawing up to five samples, CHOKe manages to protect responsive flows.

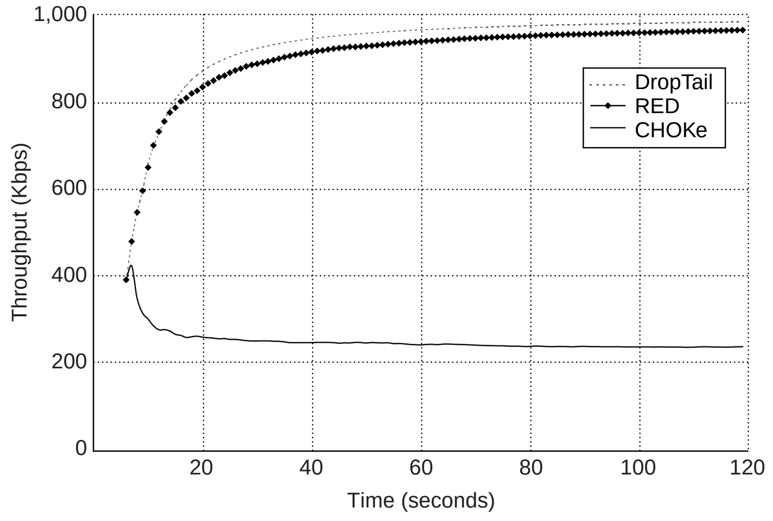


Figure 3. UDP throughput comparison.

Variable-length packets

One problem with the basic CHOKe algorithm is that it treats all packets the same regardless of their size. As a result, flows with larger packet sizes get more bandwidth than flows with smaller packet sizes. More precisely, let F_1 and F_2 denote two flows with equal arrival rates (measured in bytes/sec) but different packet sizes, equal to S_1 and $S_2 = 2 \times S_1$ respectively. Since the two flows have the same arrival rate, the former will send, on average, twice the number of packets than the latter. Thus, F_1 packets will trigger more comparisons and occupy a larger proportion of the FIFO buffer, resulting in more drops compared to the packets of F_2. Let T_1 and T_2 be the throughputs of F_1 and F_2 respectively in bytes/sec. It's easy to see that $T_1 < T_2$.

Figure 5 (next page) depicts this phenomenon. The simulation setup for this figure is based on the network topology of Figure 2. In this case the bottleneck link with bandwidth 1,000 Kbps is shared by 2-UDP and 15-TCP sources. Both UDP flows have a rate of 1,000 Kbps, but their packet sizes are 800 and 400 bytes respectively. TCP packet sizes equal 400 bytes. It's evident from the figure that basic CHOKe fails to provide fairness between the two UDP flows. The throughput of the UDP flow with the large packets equals 424 Kbps while the throughput of the other UDP flow equals only 230 Kbps. The rest of the bandwidth is almost equally divided among the TCP flows.

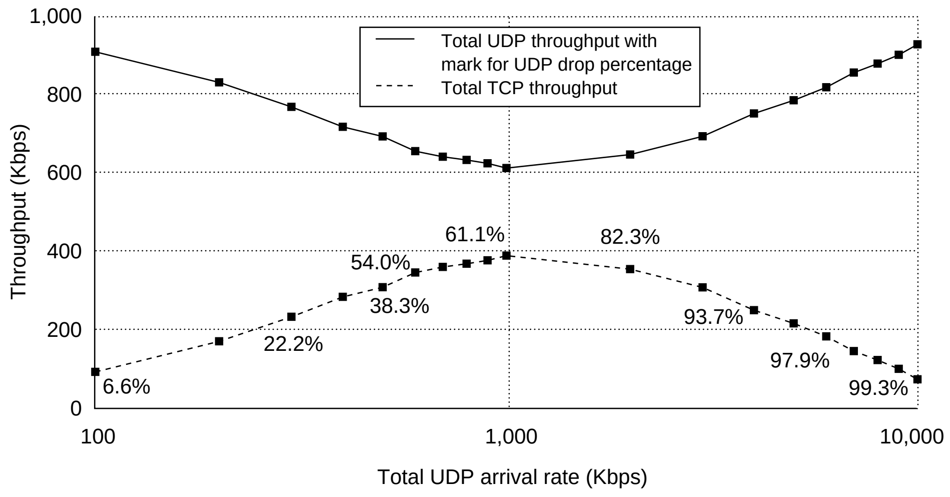


Figure 4. Self-adjusting CHOKe: throughput for 32-TCP and 5-UDP configuration.

Chopping the packets

Cleary, random sampling on a per-packet basis doesn't lead to fairness at the byte level, and hence basic CHOKe cannot ensure fairness among flows with variable-length packets. Thus, there is a need to penalize flows according to the number of bytes, rather than packets, sent by them. To accomplish this, the algorithm would ideally need to sample a random byte from the buffer and determine the packet, and hence the flow to which this byte belongs. If, as a result of the comparison, it's determined that the byte should be dropped, a dilemma occurs: Although it's bytewise fair, it's impracticable since individual bytes cannot be dropped from within a packet. Some-

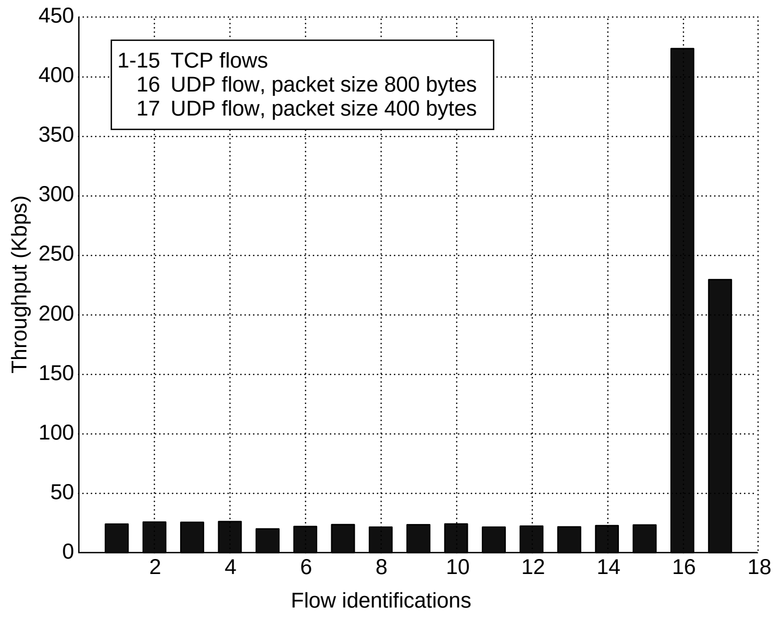


Figure 5. Throughput in a single congested link of a 1,000-Kbps bandwidth.

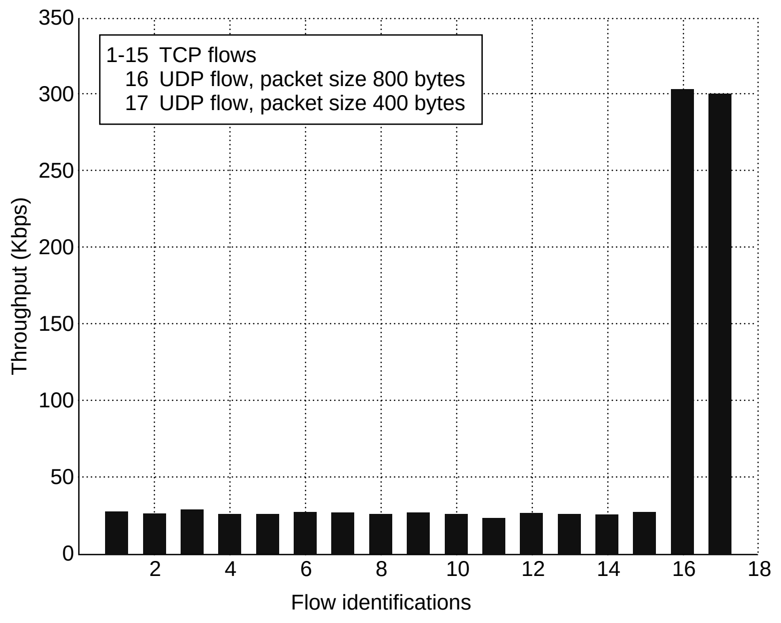


Figure 6. Throughput in a single congested link of a 1,000-Kbps bandwidth. Rate of both UDP flows is 1,000 Kbps. Each 800-byte UDP packet is chopped to two 400-byte packets.

what surprisingly, we find that it suffices to drop the whole packet. That is, it's enough for the sampling process to take place at the byte level, while the dropping process can drop entire packets.

Since byte-level sampling is not practical, we seek a mechanism to approximate it. One mechanism for this is to use a data structure for drawing samples. The number of data structure entries corresponding to each packet are proportional to its size. If $S_{\min}$ is the size of the smallest packet in the queue, a packet of size S will have S mod $S_{\min}$ entries in the data structure. Uniformly sampling the data structure entries is a good approximation of byte-level sampling.

In a system, packets are stored in memory at arbitrary places, and a linked list is maintained to indicate the order of the packets in the FIFO queue. Thus, sampling can take place in this list. To make the sampling process byte-level fair, it suffices to fragment the packets, since this will insert in the list the proper number of entries corresponding to each packet. We refer to this fragmentation as chopping.

Actually, some implementations do segment packets into fixed-size cells anyway. In particular, this is common practice among high-performance switch designs because it makes memory management easier and more efficient. In these cases we get the chopping for free as a part of the process.

Analysis

Byte-by-byte fairness in the CHOKe scheme using packet chopping can be explained by comparing the number of drops faced by two flows whose arrival rates are equal, while their packet sizes differ.

In particular, assume flows F_1 and F_2 have packet sizes S_1 and S_2 respectively, and identical arrival rates (measured in bytes/sec). Let $S_1 = K \times S_2$. To simplify analysis, assume that all the packets of a particular flow are the same size. (For flows with variable-packet sizes, the analysis can be extended by averaging over a large number of packets.) According to the chopping scheme, packets of flow F_1 will be chopped into K parts. We aim to compare the expected number of bytes dropped from flows F_1 and F_2, denoted by N_1 and N_2 respectively, over a period of time during which S_1 new bytes arrive at the front of the queue from each flow.

Since the packets of size S_1 will be chopped into K parts of size S_1/K, upon arrival, the probability of match p_m for both flows will be the same. It's easy to see why this is true when there are no packet drops. For this probability to remain approximately the same when packet drops take place, the packet drops faced by both flows should be approximately the same.

For flow F_1, every time there is a match, the newly arrived packets of size S_1 and all the items of size S_1/K that correspond to the matched chopped packet will be dropped. Thus

$$N_1 = (S_1 + K \times S_1/K) \times p_m = 2S_1 \times p_m.$$

For flow F_2, we examine the aggregate effect of K arrivals to compare with the previous case. If we consider that the number of packets of the flow in the queue is much larger than K—this is reasonable since the flow is supposed to be misbehaving and causing congestion—then the number of packets out of K that cause packet drops follow binomial distribution with parameters K and p_m. Every time there is a match, the newly arrived packet of size $S_2 = S_1/K$ and the matched packet of the same size will be dropped. Thus

$$N_2 = Kp_m \times (2 \times S_1/K) = N_1.$$

Simulation results

To evaluate the performance of the chopping scheme, we ran various simulations in a single congested link. We compared the throughput used by flows with the same arrival rate but different packet sizes.

Figure 6 plots the throughput obtained by 2-UDP and 16-TCP flows, when the simulation scenario is the same as in Figure 5. The two UDP flows use up the same amount of bandwidth, despite their packet size difference, while the TCP flows are not affected by the chopping mechanism.

In Figures 7 and 8 we vary the rate of both UDP flows simultaneously from 100 Kbps to 10,000 Kbps. The rest of the simulation parameters are the same as in Figure 5. In Figure 7 only one sample (drop candidate) is drawn from the queue. Since one sample is not enough to sufficiently penalize the unresponsive UDP flows, Figure 8 shows the bandwidth allocation when two samples are drawn. Note that the UDP with a large packet size gets a lot of bandwidth under basic CHOKe while chopping packets equalizes the performance of the two UDPs without taking precious bandwidth from the TCP flows. The conclusion is that chopping packets seems to work both for any UDP rate and under sufficient and insufficient numbers of drop candidates.

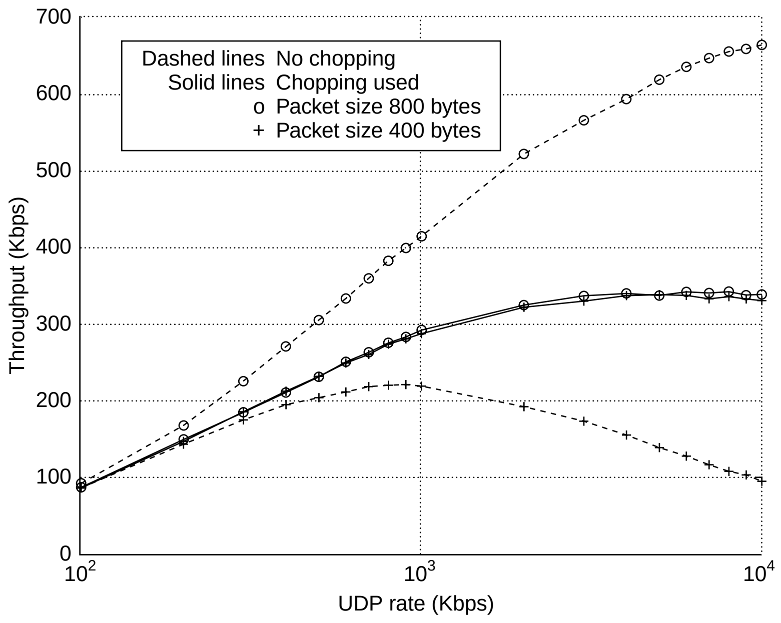


Figure 7. UDP throughput in a single congested link of a 1,000-Kbps bandwidth with one sample drawn from the queue.

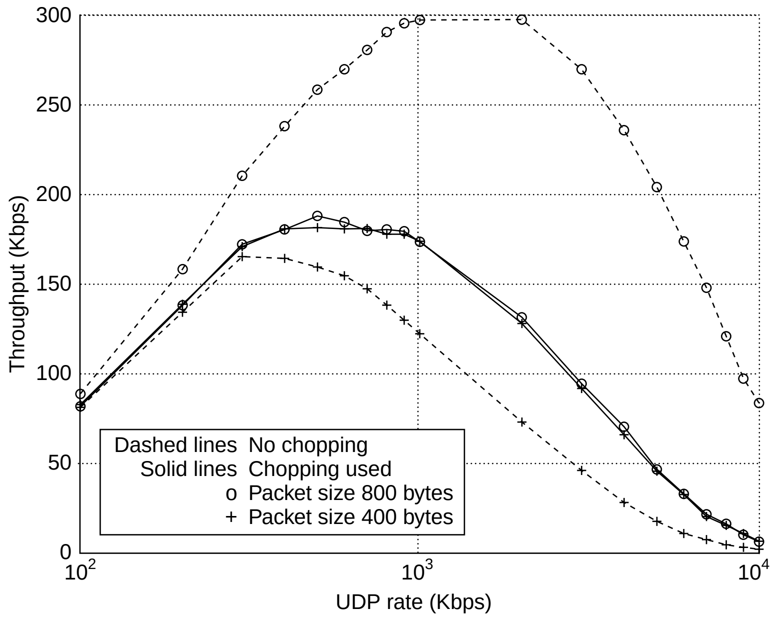


Figure 8. UDP throughput in a single congested link of a 1,000 Kbps bandwidth with two samples drawn from the queue.

A UDP flow that chooses a large packet size,

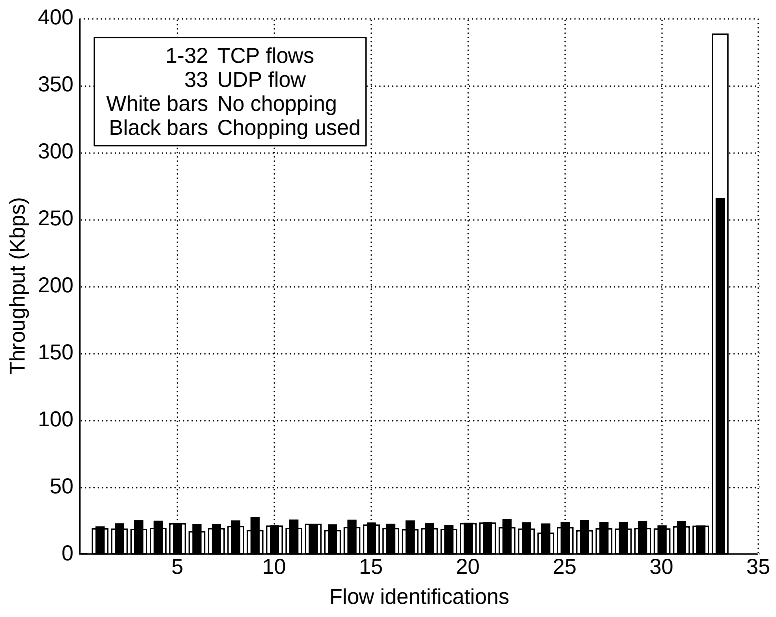


Figure 9. Throughput in a single congested link of a 1,000-Kbps bandwidth. UDP flow rate is 1,000 Kbps.

can have an advantage over both other UDP flows and of TCP flows. Figure 9 compares the throughput obtained by 1-UDP and 32-TCP flows with packet sizes of 800 and 400 bytes respectively. It's evident that chopping reduces the throughput obtained by the UDP flow, and thus protects TCP flows from UDP flows that try to use up more bandwidth by choosing large packet sizes.

Packet chopping can be applied to any flow, including TCPs. However, the TCP rate is dictated by their back-off mechanism and the degree of congestion. Thus, the analysis of the chopping mechanism does not hold for TCP flows. Given that UDP flows consume most of the available bandwidth and TCP flows in general back off properly, it's not critical to investigate the problem of different packet sizes among TCP flows. We mainly used TCP flows in the previous simulations to introduce randomness and make the scenarios more realistic, and not to study their throughput, which we've done extensively elsewhere.[2]

Implementation issues

Roughly speaking, packet switches are stored in main memory (SDRAM) at arbitrary places. A linked list is maintained in a different, faster physical memory (SRAM) to indicate the packet order in the FIFO queue. The list items contain pointers to the memory pages where the actual packets are stored.

One way to implement basic CHOKe is to randomly sample an address from the memory where the linked list is stored. In case of a match, a single bit flag of the current item of the linked list can be set to one, to indicate that the corresponding packet should be dropped. Marking the item is preferred over removing it because removal is an expensive operation since it requires breaking the list. To save memory space, the system can immediately move (at the time of marking) the respective memory address where the actual packet is stored into the free list. (The free list keeps track of all available memory spaces for new-packet storage.) At the head of the queue, packets corresponding to linked list items whose flag bits are zero are sent to the outgoing line, while packets whose flag bits equal one are ignored, see Figure 10a.

When CHOKe draws multiple drop candidates from the queue, the algorithm needs to perform this operation more than once. This makes the whole process slower. However, there are cases where due to the linked list's large size that it is stored in more than one physical SRAM. In these cases, the algorithm may sample from each distinct memory in parallel to accelerate the procedure, see Figure 10b.

Cases where the same memory has multiple FIFOs may require drawing samples until a sample is drawn from the correct FIFO. This assumes it's possible to tell which FIFO a sample belongs to, which can be accomplished by using a couple of bits to distinguish between FIFOs.

Packet chopping, which the enhanced version of CHOKe supports, adds complexity. Packet chopping may require marking more than one consecutive item on the linked list. This is necessary when a large packet that corresponds to more than one item in the list should be dropped. In such a case, the flag of each of these items should be set to one. Therefore, it may be necessary to traverse the list both forward and backward, since the random sample can be any of the items that correspond to the large packet. However, if possible memory waste can be tolerated by the

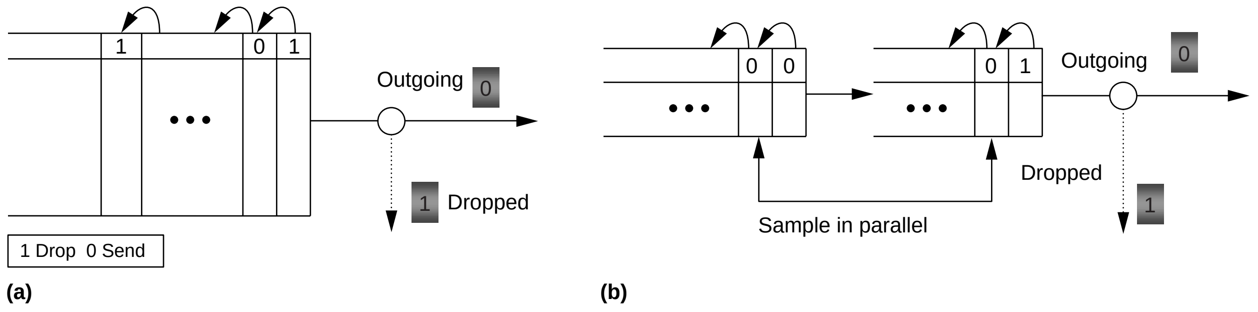


Figure 10. Implementation for single packet dropping (a) and multiple packet dropping (b).

system, there is no need to mark the rest of the cells and free the respective memory occupied by all the cells of the chosen packet. At the defragmentation phase, any packet with one or more of its cells marked can just be ignored.

Front CHOKe

A different approach to avoid these implementation challenges is to simply not use random sampling. In particular, all systems keep track of the head and the tail of a FIFO queue. Therefore, it's very easy to always choose, for comparison, the identification of the packet at the queue's head or tail. We call these variations of the basic algorithm Front CHOKe and Back CHOKe.

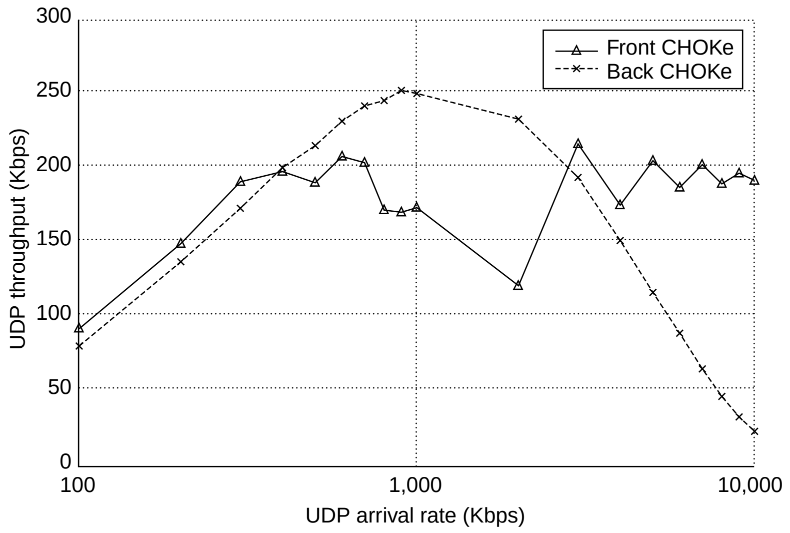


Figure 11. Throughput performance: Front CHOKe versus Back CHOKe.

Front CHOKe aids bursty traffic. Indeed, if a flow sends bursts of lengths less than the queue size, and separates them by silent periods longer than the time it takes to service all packets of a burst, then the flow is not subject to drops. On the other hand, Back CHOKe penalizes bursty traffic. Bursts will result in drops, while packets that are spread out in a uniform manner over time will prevent drops. Since TCP traffic is bursty while UDP is not, Front CHOKe is more appropriate. Front CHOKe avoids random sampling without compromising performance in terms of TCP throughput.

To examine Front CHOKe's performance, we set up a simulation configuration with 32-TCP and 1-UDP sources, using the basic network topology shown in Figure 2. We varied the UDP source rate from 100 Kbps to 10,000 Kbps. Figure 11 compares bandwidth allocation of the UDP source under basic CHOKe and Front CHOKe. When the UDP rate is low, the two schemes perform nearly the same. As the UDP rate increases from 500 Kbps to 2,000 Kbps, Front CHOKe assigns a larger proportion of the bandwidth to TCP flows since it aids bursty traffic. For larger UDP rates—more than 2,000 Kbps—Front CHOKe controls the UDP bandwidth, but performs worse than basic CHOKe. However, since these rates are twice the bottleneck bandwidth and therefore unrealistic, Front CHOKe could be used successfully. Multiple drops could be incorporated to Front CHOKe by choosing as drop candidates the *m* packets that are closer to the head of the queue. A generalization of that idea is to spread out the *m* drop candidate positions along the queue. This has the advantage of penalizing large bursts, but it requires that the hardware maintain more pointers.

We are currently studying the implementation of the algorithm and some variants in a commercial high-speed router. MICRO

Acknowledgments

A Stanford Graduate Fellowship and a Terman Fellowship supported this research in part.

References

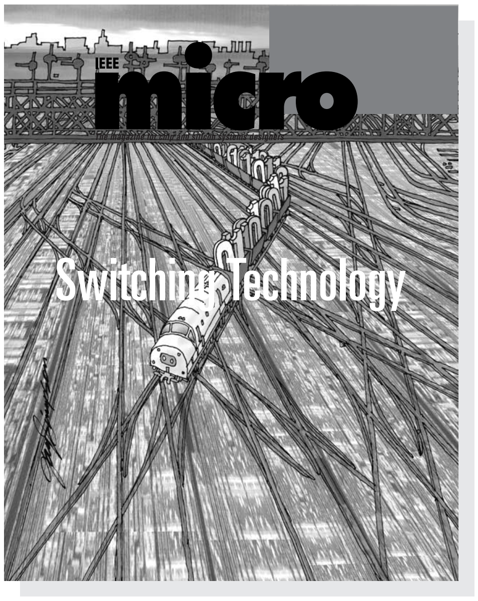
1. S. Floyd and K. Fall, "Promoting the Use of End-to-End Congestion Control in the Internet," *IEEE/ACM Trans. Networking*, Aug. 1999.
2. R. Pan, B. Prabhakar, K. Psounis, "CHOKe, A Stateless Active Queue Management Scheme for Approximating Fair Bandwidth Allocation," *Proc. IEEE Infocom*, IEEE, Piscataway, N.J., 2000, pp. 942-951.
3. S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," *IEEE/ACM Trans. Networking*, vol. 1, no. 4, 1993, pp. 397-413.
4. Network Simulator Version 2.0; http://www.isi.edu/nsnam/ns.

Konstantinos Psounis is a PhD candidate in the Electrical Engineering Department at Stanford University. He received an MS from Stanford in electrical engineering. Psounis graduated (his first degree) from the Electrical Engineering and Computer Science Department of the National Technical University of Athens, Greece. He researches probabilistic, scalable algorithms for Internet-related problems. He has worked mainly in Web caching and performance, and congestion control.

Rong Pan is a PhD candidate in the Electrical Engineering Department at Stanford University. Prior to Stanford, she was with Bell Labs, Lucent Technologies as a member of the technical staff, where she worked on Ethernet switch and wireless ATM designs. Her research interests are active-queue management schemes at network routers. She is interested in both router algorithm design and modeling.

Balaji Prabhakar is an assistant professor of electrical engineering, computer science and (by courtesy) of engineering-economic systems and operations research at Stanford University. He received his PhD from the Electrical Engineering Department at the University of California, Los Angeles. He has also served as a postdoctoral fellow at the Basic Research Institute in the Mathematical Sciences (BRIMS), Hewlett-Packard Labs, England. He was a research scientist and lecturer at the Electrical Engineering and Computer Science Department at MIT. Prabhakar's research interests include high-speed computer and wireless networks, simple packet switch schedulers and router mechanisms for Internet quality of service, Web caching and accelerating content delivery, computer network pricing mechanisms, and ad hoc wireless network algorithms. His theoretical interests include stochastic network theory and its relationship to information theory, randomized algorithms arising in networking, and probability theory. He has been awarded the NSF Career award, the Alfred Sloan Fellowship, an Okawa Foundation Research Grant, a Terman Fellowship from Stanford University, and the Erlang Prize.

Direct comments about this article to Konstantinos Psounis, Stanford University, Department of Electrical Engineering, Barnes 4B, Stanford, CA 94305; kpsounis@leland.stanford.edu.



IEEE

micro

The magazine for chip and silicon systems designers

IEEE Micro helps practitioners and academics understand the likely direction of strategic technologies in chips, systems, software, and applications. These industry professionals turn to *IEEE Micro* for the inside story about new designs and how they evolved, design trade-offs, performance, and implementation issues, as well as current electronic, legal, and standards issues, books and software reviews, and key industry trends.

Readership

IEEE Micro reaches the managers, engineers, and designers of small, high-performance systems who influence system designs in their companies and determine market winners from losers.

Editorial Calendar for 2001

January-February
Hot Interconnects

This issue focuses on the hardware and software architecture and implementation of high-performance interconnections on chips. Topics include network-attached storage; voice and video transport over packet networks; network interfaces, novel switching and routing technologies that can provide differentiated services; and active network architecture.
Ad close date: 2 January

March-April
Hot Chips

An extremely popular annual issue, Hot Chips presents the latest developments in microprocessor chip and system technology used to construct high-performance workstations and systems.
Ad close date: 1 March

May-June
Mobile/Wearable computing

The new generation of cell phones and powerful PDAs has made mobile computing practical. Wearable computing will soon be moving into the deployment stage.
Ad close date: 1 May

July-August
General Interest

IEEE Micro gathers together the latest details on new developments in chips, systems, and applications.
Ad close date: 1 July

September-October
Embedded Fault-Tolerant Systems

To avoid loss of life, certain computer systems—such as those in automobiles, railways, satellites, and other vital systems—cannot fail. Look for articles that focus on the verification and validation of complex computers, embedded computing system design, and chip-level fault-tolerant designs.
Ad close date: 1 September

November-December
RF-ID and noncontact smart card applications

Equipped with radio-frequency signals, small electronic tags can locate and recognize people, animals, furniture, and other items.
Ad close date: 1 November

IEEE Micro is a bimonthly publication of the IEEE Computer Society. Authors should submit paper proposals to micro-ma@computer.org; include author name(s) and full contact information.