Stochastic Fairness Queuing*

Paul E. McKenney
Information and Telecommunications Sciences and Technology Division
SRI International
Menlo Park, CA 94025
(415) 859-4910 / mckenney@sri.com

Abstract

Fairness queuing has recently been proposed as an effective way to insulate users of large computer communication datagram networks from congestion caused by the activities of other (possibly ill-behaved) users. Unfortunately, fairness queuing as proposed by Shenker et al. [1] requires that each conversation† be mapped into its own queue. While there are many known methods of implementing this type of mapping, they are relatively slow, requiring numerous memory references, and thus do not lend themselves to a software or firmware implementation capable of operating in high-speed networks.

This paper presents a class of algorithms collectively called "stochastic fairness queuing" that are probabilistic variants of fairness queuing. These algorithms do not require an exact mapping, and thus are suitable for high-speed software or firmware implementation. Furthermore, these algorithms span a broad range of CPU, memory, and fairness tradeoffs.

1 Introduction

Current datagram networks are vulnerable to congestive collapse when offered load approaches network capacity [2]. Although several end-to-end congestion-avoidance algorithms have been proposed [2, 3, 4, 5, 6], none of them have been shown to perform optimally in today's high-speed, high bandwidth-delay-product networks [1, 2]. This has led some researchers to conclude that gateways must participate in congestion avoidance [6]. To this end, it has been proposed that gateways use a fairness-queuing algorithm [1, 7, 8].

This fairness-queuing algorithm operates by maintaining a separate first-come-first-served (FCFS) queue for each conversation, as shown in Figure 1. Since the queues are serviced in a bit-by-bit round-robin fashion,‡ ill-behaved conversations that attempt to use more than their fair share of network resources will face longer delays and larger packet-loss rates than well-behaved conversations that remain within their fair share. Shenker has presented results showing that this algorithm performs well with a variety of topologies and traffic patterns [1].

Maintaining a separate queue for each conversation requires that the gateway be able to map from source-destination address pair to the corresponding queue on a per-packet basis. There are a number of methods of accomplishing this [9]; however, they are relatively slow (requiring numerous memory references), and are therefore unsuitable for use in gateways operating in high-speed networks. See Appendix B for a discussion of possible implementations of fairness queuing.

*This work was supported by the Rome Air Development Center and the Defense Advanced Research Projects Agency under contract number F30602-89-C-0015, and by SRI Internal Research and Development.

†A "conversation" consists of all packets with a given source-destination address pair.

‡Bit-by-bit round-robin services queues in an order that allocates bandwidth equally to the queues. If all packets are the same size, this degenerates to simple round-robin service.

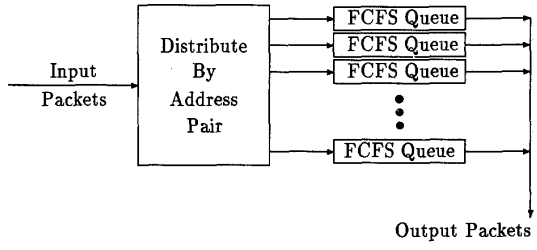


Figure 1: Fairness Queue

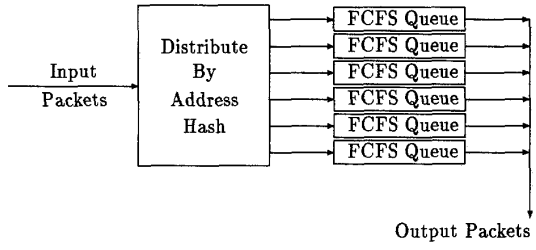


Figure 2: Stochastic Fairness Queue

In summary, although fair queuing exhibits excellent behavior, its computational requirements render it infeasible for use in high-speed networks.§ Since we cannot afford the perfect justice provided by fairness queuing, we turn to stochastic fairness queuing, which will be shown to provide reasonable justice at a price we can afford.

Stochastic fairness queuing can be most easily understood by comparing it to strict fairness queuing. The major differences are that the queues are serviced in strict round-robin order and that a simple hash function is used to map from source-destination address pair into a fixed set of queues, as in the (very small) six-queue example shown in Figure 2. If the number of queues is large compared to the number of conversations, each conversation will with high probability be assigned to its own queue. If two conversations collide, they will continue to collide, resulting in each conversation of the pair persistently receiving less than its share of bandwidth. This situation is alleviated by periodically perturbing the hash function (as described in McKenney [11]) so that conversations that collide during one time period are very unlikely to collide during the next.

Simulation results presented in Section 4 show that stochastic fairness queuing can achieve a performance that is within a factor of three of that of fairness queuing and is almost two orders of magnitude better than that of simple FCFS queuing.

Stochastic fairness queuing can act in concert with many

§However, fair queuing is quite feasible in low-speed networks, many of which still exist [10].

end-to-end congestion-avoidance algorithms, such as the DEC-bit scheme and the Jacobson/Karels TCP [1].

Stochastic fairness queuing can also be combined with resource reservation algorithms such as flows [12], MCHIP [13], and ST [14]. These algorithms require per-connection information (such as maximum and average allowed throughputs), which is obtained from the user, possibly through negotiation with a network resource scheduler.

A network that provided both best-effort and reservation services would use a reservation algorithm to control the bandwidth used by predictable traffic, and stochastic fairness queuing to allocate the remaining bandwidth fairly. The reservation algorithm would leave some fraction of the total bandwidth for best-effort services.

Note that this hybrid datagram/connection approach allows so-called "hot pairs"* of hosts to be handled in a natural way; these hosts could use reservation services to obtain the needed bandwidth.

Section 2 presents analytic expressions for the expected performance of a stochastic fairness queue. Section 3 exhibits a particular implementation of a stochastic fairness queue, and Section 4 presents results from simulation of this implementation. Section 5 presents alternative implementations spanning a broad range of CPU, memory, and fairness tradeoffs. Section 6 discusses future work, and Section 7 contains concluding remarks. Appendix A contains pseudo-code for the particular implementation of stochastic fairness queuing used in the simulations.

2 Analysis

The analysis of stochastic fairness queuing closely parallels that of hash tables with chaining. The only difference is that a collision in a hash table causes a search of only half of the linked list (on the average), while a collision in a stochastic fairness queue causes all of the colliding conversations to share the queue. An analysis of hash table performance may be found in Graham et al. [15]. Adapting this analysis to stochastic fairness queues and assuming a large number of queues gives the number of conversations that a given conversation can expect to share its queue with (counting itself), represented by

$$EC = \alpha + 1 \tag{1}$$

and

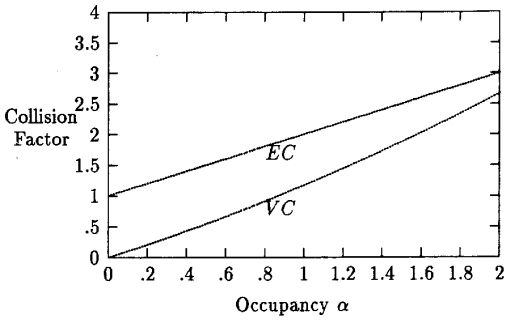
$$VC = \frac{\alpha^2}{6} + \alpha \ , \tag{2}$$

where EC is the expected number of conversations, VC is the variance in the expected number of conversations, and α is the ratio of the number of conversations to the number of queues. A plot of these values is shown in Figure 3.

Consider a stochastic fairness queue that is empty (its occupancy is zero). Then a new conversation is guaranteed to be given a queue with exactly one occupant (the conversation itself). This is indicated on the plot by a value of 1 for EC and of 0 for VC. On the other hand, a stochastic fairness queue with as many conversations as queues (occupancy of one) has a value of 2 for EC and a value of about 1.17 for VC. This indicates that a new conversation will share its queue with one other conversation on the average, but that the actual number of conversations in the queue may vary considerably from that value.

Note that the expected number and variance of conversations sharing a given queue will be low when the occupancy is low. This indicates that the stochastic fairness queue will behave in a very consistent, predictable manner when given a sufficient

*A "hot pair" of hosts needs to exchange an unusually high volume of traffic. An example of a hot pair of hosts might be a mail gateway.



EC–Expected Value of Collision Factor
VC–Variance of Collision Factor

Figure 3: Expected Collision Factor and Variance

number of queues (for instance, about five or ten times the number of conversations). Feldmeier has collected data showing that the number of concurrent conversations passing through MIT's ARPANET gateway in early 1988 was almost always less than 180 [16]. This would indicate that about 1000 to 2000 queues would suffice; this number can be easily accommodated by today's large random-access memories.

The fact that both EC and VC are continuous with respect to the occupancy indicates that the algorithm is not prone to sudden failure, but instead gracefully degrades under overload.

3 Example Implementation

The following subsections describe the requirements for the hash function, exhibit a particular function that meets those requirements, and demonstrate the data structures and algorithms used by a specific implementation of stochastic fairness queuing.

3.1 Hash Function

The example implementation uses a hash function to map from source-destination address pair to queue index. This hash function must give a high information content, as defined by Jain in [17], but must also allow perturbation such that address pairs that collide for one perturbation value are very unlikely to collide for a different perturbation value.

Two hash functions were used in simulations. The first is the high-level data-link control (HDLC) procedure (ISO-3309-1979) cyclic redundancy check (CRC) [18]. Hardware implementations of the HDLC CRC are commercially available. This hash function is perturbed by multiplying each byte by a sequence number in the range from 1 to 255 before applying the CRC.

The second is the simple software algorithm given by

$$\text{hash} = \text{ROL}(\text{src}, \text{seq}) + \text{dst} \ , \tag{3}$$

where "ROL" is the circular rotate-left function implemented as a single instruction on many computers, "src" is the Internet Prococol (IP) source address, "seq" is a sequence number in the range from zero to 31 that is used to perturb the hash function, and "dst" is the IP destination address. Although this hash function does not give as high an information content as the HDLC CRC, it can be implemented very efficiently on many computers and performs very well in simulations provided that the number of queues is not a power of two.

Note that this software hashing function is specific to IP; more work is needed to find a software hash function that can efficiently handle variable-length addresses such as those found in the ISO protocols.

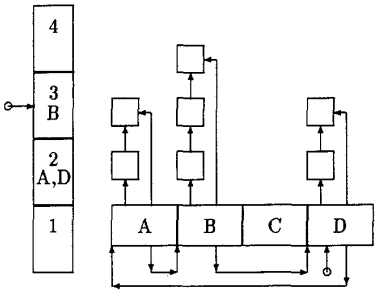


Figure 4: Stochastic Fairness Queuing Data Structures

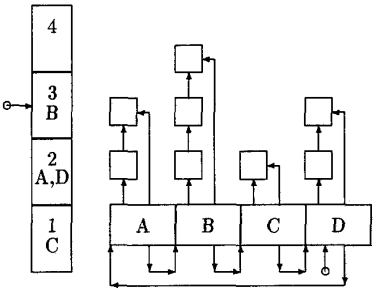


Figure 5: SFQ After Arrival on Queue "C"

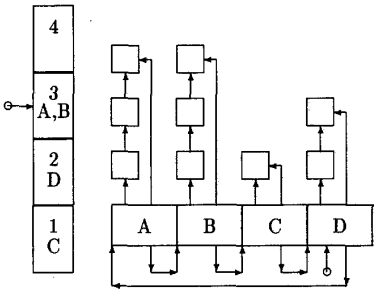


Figure 6: SFQ After Arrival on Queue "A"

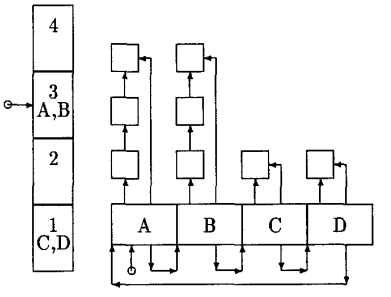


Figure 7: SFQ After Departure from Queue "D"

3.2 Data Structures and Algorithm

A particular algorithm from the class of stochastic fairness-queuing algorithms was simulated in order to provide proof of concept. This section describes the data structures used by this particular algorithm (see Figure 4) and demonstrates this algorithm (see Appendix A for a formal description). See Section 5 for discussion of other instances of stochastic fairness queuing.

The type of stochastic fairness queue simulated consists of an array of finite-length queues (the lettered boxes in Figure 4); a doubly-linked "active list" that includes only those queues that are non-empty (in this case, the queues labeled "A", "B", and "D"); a round-robin pointer that points to the queue that is to be serviced next (currently "D"); an array of doubly-linked "number-of-elements lists" (the numbered boxes in Figure 4, one for each possible non-zero queue length); and a maximum-size pointer to the element of this array corresponding to the longest queue (currently three). The links for the number-of-elements lists have been omitted from the figure in the interest of readability; list one is empty, list two contains queues "A" and "D", list three contains queue "B", and list four is empty. Queue "C" is empty, and therefore does not appear on either the active list or any of the number-of-elements lists.

The purpose of the active list and the round-robin pointer is to allow the next departing packet to be located without wasting time in scanning over empty queues. The purpose of the number-of-elements list and the maximum size pointer is to allow the longest queue to be located without wasting time searching.*

The algorithm simply maintains these data structures, as will be demonstrated by running through four examples: (1) a packet arriving for queue "C" (that is, an arriving packet whose source-destination address-pair is hashed to queue "C"); (2) a packet arriving for queue "A"; (3) a packet leaving queue "D"; and (4) a packet leaving queue "C".

A packet arriving for queue "C" will be appended to queue "C". Queue "C" will be linked into the active list and added to number-of-elements list (NEL) number 1. This results in the configuration shown in Figure 5.

A packet subsequently arriving for queue "A" will be appended to queue "A". Queue "A" is already in the active list, and thus need not be added; however, it must be deleted from NEL 2 and added to NEL 3. This results in the configuration shown in Figure 6.

A packet leaving queue "D" will cause queue "D" to be deleted from NEL 2 and added to NEL 1. The active list is not changed, but the round-robin pointer is advanced from queue "D" to queue "A". This will result in the configuration shown in Figure 7.

The round-robin pointer must advance through queues "A" and "B" before a packet may leave queue "C". Thus, one packet will leave each of queues "A" and "B", resulting in the configuration shown in Figure 8.

At this point, the lone packet from queue "C" may depart. This will result in the removal of queue "C" from both the active list and NEL 1, as shown in Figure 9.

Note that all of the algorithm's operations are time complexity $O(1)$,† suitable for implementation in high-speed software or firmware. In particular, adding a packet to a queue requires two doubly-linked-list operations and one singly-linked-list operation

*This occurs when the buffer pool is exhausted, in which case buffers will be stolen from the longest queue in order to accommodate arriving packets. This buffer theft is the only purpose of the number-of-elements list; eliminating this data structure would allow significant speedup. This and other variants of stochastic fairness queuing are discussed in Section 5.

†The "big-O" notation describes the asymptotic performance of an operation or algorithm within a constant factor [9]. Thus, an $O(1)$ operation is guaranteed to complete in a fixed amount of time, regardless of the size of the problem.

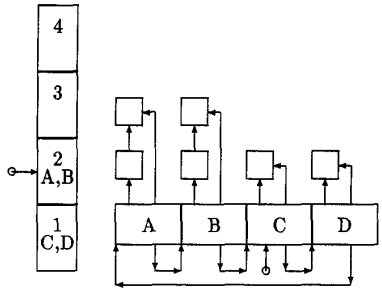


Figure 8: SFQ After Departure from Queues "A" and "B"

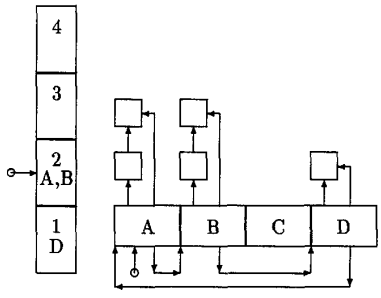


Figure 9: SFQ After Departure from Queue "C"

(in addition to the hashing operation), unless the buffer pool was exhausted, in which case it requires an additional two doubly-linked-list operations and one singly-linked-list operation (for a total of four doubly-linked-list operations, two singly-linked-list operations, and one hashing operation). Deleting a packet from a queue always requires two doubly-linked-list operations and one singly-linked-list operation.

Because of the fact that it is never necessary to do any scanning of the data structures comprising a stochastic fairness queue or any source-destination-address-pair comparisons, the operation count is quite small, compared to that of fairness queuing.

4 Simulation

The behavior of stochastic fairness queuing was studied using two different simulations. The first consists of a single overloaded node with no transport protocol action; this was used to do parametric studies. The second is a more realistic simulation of multinode networks with several transport protocols [19].

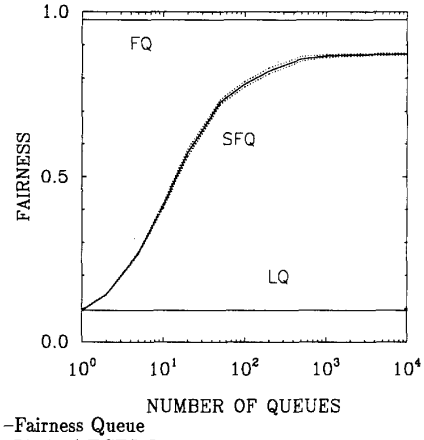
4.1 Parametric Studies

The object of the simulation is to determine whether the behavior of stochastic fairness queuing is a good approximation of that of fairness queuing. A very simple simulation suffices for this purpose. The simulation consists of a single node with four saturated input lines and one output line. All packets are pure datagrams of equal length; no transport-layer protocol was simulated. There are 20 conversations, one of which is ill-behaved, generating as much input traffic on the average as the other 19 combined. During each time interval, one packet departs from the node and four are offered to the node. The conversation to which a given input packet belongs is randomly chosen.

Each queue making up the stochastic fairness queue is a finite FCFS queue, and a perturbable variant of the HDLC CRC is used as the hash function. Hash function switching is done in such a way as to avoid packet reordering; newly occupied queues are appended to the end of the active list, and buffers are stolen from the longest queue to accommodate packets that arrive when the buffer pool is exhausted. A per-conversation fairness policy is used, and the fairness granularity is irrelevant, since all packets are of equal size.

The baseline stochastic fairness queuing run used 160 queues (eight times the number of conversations), each with a maximum length of five packets, a buffer pool containing space for up to 160 packets, and a hash function switch interval such that the hash function was perturbed for every 1000 input packets. The simulation runs until 10,000 packets have been offered to the node,* at which point the number of packets output per conversation is printed.

The figure of merit used to analyze the results is the ratio of the bandwidth granted to the least-fortunate conversation to that granted to the most-fortunate conversation. A perfectly fair algorithm will have a fairness of one (since it will treat all conversations exactly equally). As points of reference, the fairness of fairness queuing, baseline stochastic fairness queuing, and of a length-five FCFS queue are 0.98, 0.81, and 0.095 packets per conversation, respectively.



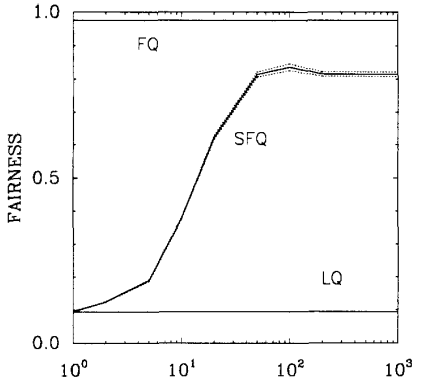
FQ –Fairness Queue
LQ –Limited FCFS Queue
SFQ–Stochastic Fairness Queue

Figure 10: Effect of Number of Queues

Increasing the number of queues in the stochastic fairness queue increased its performance, as shown in Figure 10. The lines labeled "SFQ" show the mean value of the fairness for stochastic fairness queuing taken over five runs and the 95% confidence interval. The lines labelled "FQ" and "LQ" show the performance of fairness queuing and length-five FCFS queuing, respectively. The 95% confidence bounds for FQ and LQ are barely wider than the line itself, and are not shown. The performance of SFQ should converge to that of LQ as the number of queues approaches one, and should converge to that of FQ as the number of queues grows without bound. The fairness for stochastic fairness queuing given 1000 queues (50 times the number of conversations) is 0.86, or 88% of that of pure fairness queuing and over nine times that of FCFS queuing.

Increasing the number of buffers in the stochastic fairness queue increased its performance, as shown in Figure 11. The lines

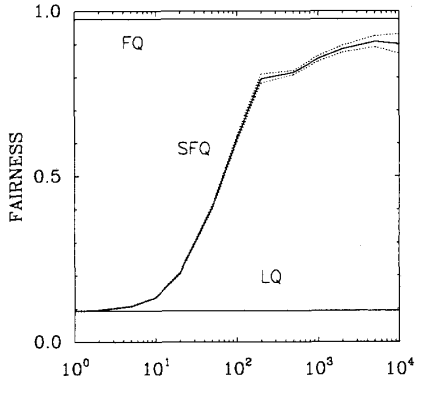
*Or, equivalently, until about 2500 packets (125 per conversation, on the average) have been output from the stochastic fairness queue (since the node is running under four-times overload).



FQ –Fairness Queue
LQ –Limited FCFS Queue
SFQ–Stochastic Fairness Queue

Figure 11: Effect of Number of Buffers

labelled "SFQ" show the mean value of the fairness for stochastic fairness queuing taken over five runs and the 95% confidence interval. Since each queue can contain five packets at most, and there can be 20 queues at most (one per conversation), the algorithm can make use of 100 buffers at most. This can be seen in Figure 11; the performance levels off above 100 buffers.



FQ –Fairness Queue
LQ –Limited FCFS Queue
SFQ–Stochastic Fairness Queue

Figure 12: Effect of Hash Function Switch Interval

Increasing the length of the hash function switching interval increases performance up to a point, due to the fact that the transient unfairness associated with the switchover occurs less frequently.* However, as the duration of the switching interval approaches the average length of the conversation, performance starts decreasing, as can be seen in Figure 12. This is due to the fact that conversations that collide do so for most of their lifetime.

*The transient unfairness is caused by the fact that an ill-behaved conversation can have two queues at its disposal during the switchover, while the well-behaved conversations will limit themselves to a throughput appropriate for a single queue.

Good values for the switch interval appear to lie between twice the queue-flush time of the stochastic fairness queue and one-tenth of the average conversation duration.

Additional simulations were run with buffer stealing disallowed and with a hash-function switching method that allows some packet reordering. Both of these modifications allow more CPU-efficient implementations. Disallowing buffer stealing has very little effect, given enough memory for each conversation to have a full queue. The faster hash-function switching method has almost no effect on fairness, but is more compatible with existing transport protocols, as will be seen in the next section.

These results demonstrate that the performance of stochastic fairness queuing can approach that of pure fairness queuing, achieving almost two orders of magnitude improvement over FCFS queuing.

4.2 Transport Protocol Studies

Transport protocol studies were performed using the REAL network simulation package [19]. The scenarios described in Shenker et al. [1] were run using a version of stochastic fairness queuing that used buffer stealing, a per-conversation fairness policy, and a per-byte fairness policy. An efficient hash-function switching method was used that allowed packets to be reordered, as the method that avoids reordering can cause occasional packet loss from low-throughput conversations, which severely decreases the throughput when those conversions are using TCP with Van Jacobson's modifications. Runs were made using HDLC CRC and using the software algorithm described earlier for the hash function: as expected, essentially identical results were obtained. Except as noted below, the FTP throughput results for stochastic fairness queuing were within ten percent of those for fairness queuing.

Fairness queuing can provide lower delay to small packets than can stochastic fairness queuing, as a consequence of the greater amount of state maintained by the former. For example, in scenario 1 (an underloaded gateway passing two FTP and two Telnet conversations) fairness queuing provides the Telnet conversations about 17 times lower delay than it provides to the FTP conversations, while stochastic fairness queuing provides roughly a factor of four improvement in delay. None of the scenarios included Telnet and FTP conversations sharing a large bandwidth-delay-product link; it seems likely that this would greatly reduce the importance of queuing delay.

Fairness queuing includes a mechanism that actively punishes conversations perceived to be malicious. Since stochastic fairness queuing does not attempt to judge users' intents, the results of scenario 3 (an overloaded gateway passing one well-behaved FTP, one well-behaved Telnet, and one ill-behaved FTP) differ significantly. Fairness queuing almost completely shuts down the ill-behaved FTP. However, stochastic fairness queuing grants both FTPs roughly equal bandwidth; the ill-behaved FTP gets about 20% more bandwidth in exchange for a packet-loss rate of over 95%.

These results demonstrate that stochastic fairness queuing works well in the presence of real-world transport protocols.

5 Alternative Implementations

The stochastic fairness-queuing algorithms span a broad range of CPU, memory, and fairness tradeoffs. This allows an algorithm to be configured for a specific environment or range of environments. Configuration consists of selecting structural parameters and values for tuning parameters.

The structural parameters include (1) queuing discipline; (2) hash function; (3) hash function switching method; (4) active-list insertion policy; (5) buffer-theft policy; (6) fairness granularity;

and (7) fairness policy.

The queuing discipline used in the simulation is finite FCFS; a packet that arrives while its queue is full is discarded. Alternative queuing disciplines include the various forms of random-drop queues.

As mentioned earlier, the simulation uses a variant of HDLC CRC and a simple rotate-and-add function as the hash functions. There are many possible alternative hash functions, including the Fletcher checksum used in OSI, other types of CRC, and various ad hoc functions consisting of sequences of simple functions such as shifts, adds, and exclusive-ORs.

The simulated algorithm initially took great care to avoid transmitting packets out of order while switching between perturbations of the hash function. However, this caused occasional packet loss, which caused conversations using TCP with Van Jacobson's modifications to unnecessarily decrease their throughput.

Therefore, the simulated algorithm now simply perturbs the hash function while continuing to use a single set of queues and lists. This method has the disadvantages of causing out-of-order packet transmission and allowing a higher degree of unfairness for a short time after the switchover (since an ill-behaved source might have sole use of two separate queues during this time), but is much simpler, allows a more efficient implementation, and is more compatible with existing transport protocols.

The simulated algorithm always appends newly occupied queues to the end of the active list. An alternative method would be to probabilistically insert newly occupied queues containing small packets onto the head of the active list. This would grant smaller average delay to small packets (which tend to be Telnet and Transmission Control Protocol [TCP] acknowledgement packets), but would increase the complexity of the algorithm.

The buffer-theft policy implemented in the simulation is to always remove the next packet that would have been output from the longest queue. Alternative policies include dropping a randomly selected packet from the longest queue and simply refusing to do buffer theft. The latter alternative is particularly attractive, as it allows the number-of-elements lists to be dispensed with, thereby greatly reducing the complexity of the algorithm. Adding a packet to a stochastic fairness queue that does not do buffer theft requires one singly-linked-list operation, with an additional doubly-linked-list operation if the queue was initially empty. Deleting a packet from a queue also requires one singly-linked-list operation, with an additional doubly-linked-list operation if this was the last packet in the queue. Queues tend to be long under heavy load, so this configuration of stochastic fairness queuing actually consumes *fewer* CPU resources when heavily loaded. On the other hand, this method is likely to require more buffers than the method simulated to achieve a comparable level of fairness.

The simulation used a packet fairness granularity: that is, a single packet is output from each queue, regardless of packet length. An alternative would be an approximate bit or byte fairness granularity. This can be implemented efficiently using an approach similar to that used in Shenker's fairness-queuing algorithm [1]. This alternative has the advantage of allocating bandwidth more fairly in the presence of differing packet sizes, but increases the complexity of the algorithm, especially since care must be taken to avoid indefinitely postponing packets already queued.

The fairness policy used by the simulation was equal allocation per host conversation. Many other fairness policies can be envisioned, many of which can be implemented by including different fields from the packet header in the hash function. For example, a policy of equal allocation per *network* conversation can be implemented by hashing only the network portions of the source-destination address pair. This fairness policy might be used in transit networks.* Another example would be a policy of equal allocation per TCP connection, which could be implemented by including the TCP port numbers in the hash function.†

The tuning parameters are (1) number of queues; (2) number of buffers; (3) maximum queue length; and (4) hash function switching interval.

Each of these parameters was a command-line argument to the simulation program; the observed effects are described in Section 4. Note that some protocols (in particular, older versions of the Network File System protocol) place restrictions on the maximum queue length; if the queue length is too short, very poor performance will result [20].

6 Future Work

Additional work needs to be done to evaluate the different possible instances of the stochastic fairness-queuing algorithms presented in Section 5. These results would provide the information needed to select the algorithm that best fits a given processor and network architecture.

The simulations performed to date used either a single-hop network with no protocol action (that is, pure datagram switching) and uniform packets, or a small network with some representative transport protocols. More work needs to be done to determine the effectiveness of SFQ in large networks and in real (as opposed to simulated) gateways in real networks.

7 Conclusions

This paper has presented and analyzed a class of probabilistic variants of Shenker's fairness-queuing algorithm (called "stochastic fairness queuing" algorithms) that are suitable for use in high-speed computer communications networks and that span a broad range of CPU, memory, and fairness tradeoffs. A particular instance of this algorithm has been shown to have behavior approaching that of fairness queuing (when given sufficient resources), and to exhibit graceful degradation under overload, without sudden failure.

8 Acknowledgements

I owe many thanks to Craig Partridge, Mike Frankel, Phil Karn, Mark Lewis, John Nagle, Richard Ogier, K. K. Ramakrishnan, Vlad Rutenburg, Nachum Shacham, Scott Shenker, Greg Skinner, Lixia Zhang, and to the anonymous referees for discussions, comments, and much constructive criticism. I am indebted to Diane Lee and Mark Lewis for their support of this effort and I am especially grateful to Srinivasan Keshav for integrating a version of stochastic fairness queuing into his REAL network simulation system.

A Algorithm

This appendix describes the specific instance of the class of stochastic fairness queuing algorithms that was simulated. See Section 5 for discussion of other instances of this class of algorithms.

The following subsections show algorithms for dequeuing packets and enqueuing packets.

*At first glance, this policy seems to have the disadvantage of encouraging institutions to register many different networks to increase their share of bandwidth. The transit networks can prevent this form of abuse by simply refusing to pass routing information for the excess networks.

†This policy might have the disadvantage of requiring very large numbers of queues in addition to the blatant (but not unprecedented) layering violation.

A.1 Dequeue a Packet

The following algorithm removes a packet from a stochastic fairness queue:

1. If currently switching to a new perturbation of the hash function and the active list is empty, complete the switch (start outputting from the new queue).
2. If the active list is empty, exit (this implies that the entire SFQ is empty).
3. Output a packet from the queue pointed to by the round-robin pointer.
4. Advance the round-robin pointer.
5. Delete the queue from the NEL it was in. If this NEL is now empty and the maximum-size pointer points to this NEL, decrement the maximum-size pointer (this implies that we just output from the longest queue).
6. If the queue still contains packets, add it to the next-smaller NEL, otherwise delete it from the active list.

A.2 Enqueue a Packet

The following algorithm adds a packet to a stochastic fairness queue if possible, or discards the packets if not:

1. Compute the hash function to obtain the queue index.
2. If the queue indexed by the hash function is full or if the buffer pool is exhausted and this is the longest queue, discard the packet and exit.
3. If the buffer pool is exhausted, discard a packet from the longest queue as follows:
   (a) Locate the first queue on the NEL referenced by the maximum-size pointer.
   (b) Discard the packet at the head of this queue.
   (c) Remove the queue from the NEL, if this is the only queue on this NEL, and decrement the maximum-size pointer.
   (d) If the queue still contains packets, add it to the next-smaller NEL; otherwise delete it from the active list.
4. If the queue indexed by the hash function is empty, add it to the active list; otherwise, delete it from its NEL.
5. Add the packet to the queue.
6. Add the queue to the appropriate NEL list, incrementing the maximum-size pointer if necessary.

B Alternatives for Fair Queuing Implementations

Much of the motivation for stochastic fairness queuing stems from two aspects of fair queuing that do not lend themselves well to high-speed implementations.

The first aspect is the packet scheduling technique that fair queuing uses to provide bit-by-bit round-robin service. This technique requires addition to and deletion from a priority queue of length equal to the number of conversations flowing through the queue. The best known priority queue algorithms provide time complexity of $O(\log_2(n))$, where n is the length of the priority queue. The number of conversations in a gateway has be measured to be as large as 180 [16]. Since $\log_2(180)$ is approximately 7.5, the computational cost of just the packet scheduling part of fair queuing can exceed to total computational cost of an efficient version of stochastic fairness queuing.

The second aspect is the technique of using a one-to-one mapping from source-destination address pair into the corresponding queue. The remainder of this section examines alternative implementations of this mapping and shows how each is deficient for high-speed networks. In some cases it is necessary to look at machine-language implementations; in these cases, the Motorola MC68020 processor will be used.

At first glance, it would appear that whatever strategy was used to look up routes would suffice for address-pair mapping. However, while routing updates (which modify the routing data structure) can be processed by a low-priority background task, modifications to the fair queue structure occur on a per-packet basis. Thus, unlike routing, fair queueing cannot amortize the cost of adding a new conversation over many data packets.

Another approach is to rely on hardware in the form of content-addressable memories. The growing demand for gateways that support multiple protocols, some of which are still evolving, render this approach impractical for internetworking gateways for the foreseeable future.

The simplest and fastest way of mapping from source-destination address pair into queue is to use a simple array, indexed directly by the binary number formed by concatenating the bits representing the source and destination addresses. For example, IP has relatively small 32-bit addresses, so that the corresponding index would be a 64-bit quantity. Unfortunately, this results in an infeasible 2^{64} element array, eliminating this approach from consideration.

Various types of search trees are heavily used in database applications [9]. These methods are relatively slow, requiring numerous memory references for access and complex algorithms for updates, making them unsuitable for use in switches operating in high-speed networks.

Another alternative is the trie [9]. For example, one could imagine maintaining a 256-way trie indexed by successive bytes of the IP address pair. The first byte of the IP address pair would be used as an index into a table of 256 possibly NULL pointers. Each non-NULL pointer would point to its own table of 256 pointers, again possibly NULL, indexed by the second byte of the IP address. These tables of pointers would form an eight-level tree; non-NULL pointers at the eighth level would point to a queue header. NULL pointers at all levels are placeholders for address pairs that do not correspond to any currently active conversation.

We will assume that conversations average at least three packets in length; thus an implementation of a fair queuing trie should be optimized for the second and subsequent packets in a conversation. Since there are only eight levels in an IP trie, it makes sense to fully unroll the loop that traverses the trie. Assuming that a pointer to the concatenated IP address pair and a pointer to the root of the trie are preloaded into registers, each segment of the unrolled loop will contain three instructions. The first instruction will load the next byte of the address pair into a register while incrementing the pointer to the address, the second instruction will use this byte to index into the current level of the trie, loading a pointer to the next level, and the third instruction will branch to a special handler if this pointer is NULL. NULL pointers would be encountered upon receipt of the first packet of a new conversation; the special handler would allocate and initialize memory needed to add the new address pair to the trie.

Use of a trie thus requires 24 instructions to map from IP address pair to the corresponding queue in the best case; this ignores the added instructions needed to allocate structures for new conversations and the need to scan the trie periodically to dispose of structures corresponding to conversations that have ended. In contrast, an efficient implementation of a stochastic

fairness queue requires only 10 instructions in the *worst* case to map from IP address pair to the corresponding queue.* As noted earlier, IP has relatively short addresses; stochastic fairness queuing's advantage is greater for longer addresses.

A final alternative is the use of hash tables with chaining. The best case instruction count for a hashed fair queue in an IP network is almost as good as the worst case for stochastic fairness queuing. The difference is due to the fact that the hashed fair queue must compare the address in the packet to that of the first queue header in the chain; fair queuing must reference address fields three times as often as stochastic fairness queuing.* In addition, a hashed fair queue must periodically scan its queues in order to dispose of those corresponding to conversations that have ended, and must allocate and initialize new queues upon arrival of a packet that is part of a new conversation. The overhead due to these activities will depend on traffic statistics.

In summary, the worst-case execution speed stochastic fairness queuing is faster than the best-case execution speed of all of these implementations of fair queuing, and this advantage is larger for protocols with longer addresses, e.g., the ISO protocol suite.

References

[1] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queuing algorithm. *SIGCOMM '89*, 1989.

[2] D. Bertsekas and R. Gallager. *Data Networks.* Prentice-Hall, Inc., 1987.

[3] H. Hayden. Voice flow control in integrated packet networks. Technical Report LIDS-TH-1152, MIT Laboratory for Information and Decision Systems, 1981.

[4] Raj Jain and K.K. Ramakrishnan. Congestion avoidance in computer networks with a connectionless network layer. Technical Report DEC-TR-506, Digital Equipment Corporation, Maynard, Massachusetts, August 1987.

[5] John Robinson, Dan Friedman, and Martha Steenstrup. Congestion control in BBN packet-switched networks. *Computer Communications Review*, 20(1):76–90, January 1990.

[6] Van Jacobson. Congestion avoidance and control. In *SIGCOMM '88*, pages 314–329, August 1988.

[7] John Nagle. On packet switches with infinite storage. *IEEE Transactions on Communications*, March 1987.

[8] James R. Davin and Andrew T. Heybey. Router algorithms for resource allocation. Technical Report White Paper, MIT Laboratory for Computer Science, July 1989.

[9] Donald Knuth. *The Art of Programming.* Addison-Wesley, 1973.

*This assumes that the hashing function is implemented in software; the instruction count might decrease somewhat given a hash function implemented in hardware.

*Stochastic fairness queuing must scan the packet's address pair once in order to compute the hash function. Fair queuing must in addition scan the packet's address pair and the queue header's address pair in order to compare the two. Protocols with short address fields such as IP may allow fair queuing implementations that cache the packet's address pair in machine registers, but this is not likely to be practical for protocols with longer address fields. Note that it is not possible to overlap computation of the hash function with comparison of the address pairs, since the hash function must be computed before the queue can be located.

[10] Kirk Lougheed. Low-speed lines still heavily used. Comments to the IAB Performance and Congestion Control Working Group, July 1989.

[11] Paul E. McKenney. High-speed event counting and classification using a dictionary-hash technique. *ICPP '89*, 1989.

[12] Lixia Zhang. *A New Architecture for Packet Switching Network Protocols.* PhD thesis, Massachusetts Institute of Technology, 1989.

[13] Gurudatta M. Parulkar. The next generation of internetworking. *Computer Communications Review*, 20(1):18–43, January 1990.

[14] COIP Working Group. ST-II specification. Working document for IETF COIP Working Group, November 1989.

[15] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics.* Addison-Wesley, 1989.

[16] David C. Feldmeier. Estimated performance of a gateway routing-table cache. Technical Report MIT/LCS/TM-352, MIT Laboratory for Computer Science, March 1989.

[17] Raj Jain. A comparison of hashing schemes for address lookup in computer networks. Technical Report DEC-TR-593, Digital Equipment Corporation, February 1989.

[18] International Organization for Standardization. *Data Communication - High-Level Data Link Control Procedures - Frame Structure*, 1979.

[19] Srinivasan Keshav. REAL user's manual. Technical Report REAL distribution kit, University of California at Berkeley, 1989.

[20] Chuck Hedrick. Protocols restrict queue lengths. Comments to the IAB Performance and Congestion Control Working Group, July 1989.