Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.829 Fall 2002 **Lecture 10: Scheduling for Fairness** October 8 2002

Overview. Scheduling as a mechanism. Fair queueing. Core-stateless fair queueing (CSFQ).

These notes are based on scribed notes from 6.829 in 2001 by Ken Takusagawa.

Readings:

1. A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm", *Internetworking: Research and Experience*, Vol. 1, No. 1, pp. 3–26, 1990. (Or read ACM SIGCOMM 1989 version.)

2. I. Stoica, S. Shenker, H. Zhang, "*Core*-Stateless Fair Queueing: Achieving Approximately Fair Bandwidth Allocations in High Speed Network," in *Proc. ACM Sigcomm*, September 1998, Vancouver, Canada.

1 Introduction

In the last lecture, we discussed two ways in which routers can help in congestion control: by signaling congestion, using either packet drops or marks (*e.g.,* ECN), and by cleverly managing its buffers (*e.g.,* using an active queue management scheme like RED). Today, we turn to the third way in which routers can assist in congestion control: *scheduling.*

Specifically, we will first study a particular form of scheduling, called *fair queueing* (FQ) that achieves (weighted) fair allocation of bandwidth between flows (or between whatever traffic aggregates one chooses to distinguish between. While it achieves weighted-fair bandwidth allocations, FQ requires *per-flow* queueing and per-flow state in each router in the network. This is often prohibitively expensive. We will also study another scheme, called *core-stateless fair queueing* (CSFQ), which displays an interesting design where "edge" routers maintain per-flow state whereas "core" routers do not. They use their own non-per-flow state complimented with some information reaching them in packet headers (placed by the edge routers) to implement fair allocations.

1.1 Scheduling

The fundmental problem solved by a scheduling algorithm running at a router is to answer the following question: "Which packet gets sent out next, and when?" The choice of which flow (or flow aggregate; we use the term "flow" to mean one or the other in this lecture, unless mentioned otherwise) is the next one to use an outbound link helps apportion bandwidth between flows. At the same time, a scheduling algorithm can also decide when to send any given packet (or flow's packet), a choice that can help guarantee (or bound) packet latencies through the router.

An important practical consideration, perhaps the most important consideration in the design of router scheduling algorithms concers *state management.* The key concern here is that schemes that maintain per-flow state are not scalable, at least in several router designs. This is in stark contrast to buffer management schemes like RED, where the state maintained by the router is independent of the number of flows. However, RED does not achieve fair bandwidth allocations amongst flows, whereas good scheduling algorithms can (precisely because they tend to directly control which flow's packet gets to use the link next).

It's also worth appreciating that scheduling for weighted-fair bandwidth allocation and scheduling to provide latency guarantees are actually orthogonal to each other and separable. For instance, one can imagine a hypothetical router that starves a flow for 6 days each week and gives extremely high bandwidth for one whole day, so the flow receives some pre-determined share of bandwidth, but does not receive any latency guarantees. Scheduling schemes have been developed that provide both bandwidth and latency guarantees to flows, although they tend to be rather complex and require care in how these requirements are specified.

Broadly speaking, all schedulers can be divided into two categories: *work-conserving* and *non-work-conserving*. The former, of which fair queueing is an example, *never* keep an outbound link idle if there is even one packet in the system waiting to use that link. In contrast, the latter might, for instance to provide delay bounds. Another example of the latter kind is a strict time-division multiplexing system like the one we saw in Problem 1 or Problem Set 1.

1.2 Digression: What happens to a packet in a router?

To understand how scheduling fits into the grand scheme of things that happen in a router, it's worth understanding (for now, at a high level) all the different things that go on in a router. For concreteness, we look at an IP router. The following steps occur on the *data path*, which is the term given to the processing that occurs for each packet in the router. We only consider unicast packets for now (i.e., no multicast).

1. Validation. When a packet arrives, it is validated to check that the IP version number is correct and that the header checksum is correct.

2. Destination lookup. A longest-prefix-match (LPM) lookup is done on the forwarding table using the destination IP address as lookup key, to determine which output port to forward the packet on. Optionally, an IP source check may also be done to see if the port in which this packet is a valid, expected one. This is often used to help protect against certain denial-of-service (DoS) attacks.

3. Classification. Many routers *classify* a packet using information in the IP header fields (*e.g.,* the "type-of-service" or TOS field, now also called the "differentiated services code point" or DSCP) and information in higher layers (*e.g.,* transport protocol type, TCP/UDP port number, etc.). This is also called "layer-4 classification." The result of a classification is typically a queue that the packet is enqueued on. In theory, one can classify packets so each connection (or flow) gets its own queue, specified by a unique combination of source and destination addresses, source and destination transport-layer ports, and TOS field, but this turns out to be prohibitively expensive because of the large number of dynamically varying queues the router needs to keep track of. In practice, routers are usually set up with a fixed number of queues, and a network operator can set up classification rules that assign packets to queues. A classified queue can be fine-grained, comprising packets from only a small number

of transport-layer flows, or it could be coarse-grained comprising packets from many flows (*e.g.,* a large traffic aggregate of all packets from 18.0.0.0/8 to 128.32.0.0/16).

We'll study classification techniques and router architectures in more detail in later lectures.

4. Buffer management. Once the router has decided which queue, it needs to append the packet to it. At this time, it should decide whether to accept this packet or drop it (or mark ECN), or in general to drop some other packet to accept this one. It might run a simple strategy like drop-tail, or it might run something more sophisticated like RED or one of its many variants.

5. Scheduling. Asynchronous to specific packet arrivals, and triggered only by queue occupancy, the router's link scheduler (conceptually one per link) decides which of several queues with packets for the link to allow next on the link. The simplest scheduling strategy is FIFO, first-in, first-out.

1.3 Back to fairness

In this lecture we will consider bandwidth scheduling algorithms that achieve *max-min fairness.* With max-min fairness, no other allocation of bandwidth has a larger minimum, and this property recursively holds. Recall from Lecture 3 what max-min fairness provides:

1. No flow receives more than its request, r_i.
2. No other allocations satisfying (1) has a higher minimum allocation.
3. Condition (2) recursively holds as we remove the minimal user and reduce the total resource accordingly.

This is by no means the only definition of fairness, but it's a useful one to shoot for. One consequence of max-min fairness is that all allocations converge to some value α such that all offered loads $r_i < \alpha$ are given a rate r_i, while all inputs $r_i > \alpha$ are given a rate equal to α. Of course, the total rate available C (the output link's rate) is doled out amongst the n input flows such that

$$\sum_{i=1}^{n} \min(r_i, \alpha) = C. \tag{1}$$

Fair queueing (FQ) is a scheme that achieves allocations according to Equation 1. CSFQ is a distributed approximation to FQ that generally comes close, without the associated per-flow complexity in all routers.

2 Fair queueing

Fair queueing is conceptually simple to understand: it is like round-robin scheduling amongst queues, except it takes special care to handle variable packet sizes. The primary practical concern about it is its requirement of per-flow state. The good thing, though, is that weighted variants of fair queueing add little additional complexity.

We emphasize that we're going to use the term "flow" without a precise definition of it, on purpose. Think of a "flow" as simply the result of the classification step from Section 1.2, where there are a set of queues all vying for the output link. We wish to apportion the bandwidth of this link in a weighted fashion amongst these queues ("flows").

Pure versions of fair queueing have proved to be hard to deploy in practice because the number of flows (and therefore queues) a router would have to have is in the tens of thousands and *a priori* unknown. However, bandwidth apportioning schemes among a fixed set of queues in a router are gaining in popularity.

2.1 Bit-by-bit fair queueing

The problem with simple round-robin amongst all *backlogged* queues (i.e., queues with packets in them) is that variable packet sizes cause bandwidth shares to be uneven. Flows with small packets get penalized.

To understand how to fix this, let's consider an idealized bit-level model of the queues vying for the link. In this world, if we could forward one bit per flow at a time through the link, we would implement a round-robin scheme and bandwidth would be apportioned fairly. Unfortunately, in real life, packets are not *preemptible*, *i.e.,* , once you start sending a packet, you need to send the whole thing out; you cannot temporarily suspend in the middle of a packet and start sending a different one on the same link (this would make demultiplexing at the other end a pain, and would in fact require additional per-bit information!).

Sticking with the idealized bit-based model, let's define a few terms.

Definition: **Round**. One complete cycle through all the queues of sending one bit per flow.

Note that the time duration of a round depends on the number of backlogged queues.

Let $R(t)$ be the round number at time t. If the router can send μ bits per second, and there are N active flows, the number round number increases at rate

$$\frac{dR}{dt} = \frac{\mu}{N}. \tag{2}$$

The greater the number of active flows, the slower the rate of increase of the round number. However, the key point is that the *number of rounds* for a complete packet to finish being transmitted over the link is *independent* of the number of backlogged queues. This insight allows us to develop a scheduling strategy for fair queueing.

Suppose a packet arrives and is classified into queue α, and it is the i-th packet on the queue for flow α. Let the length of the packet be p_i^α bits.

In what round number S_i^α does the packet reach the head of the queue? ("S" for "start.")

There are two cases to consider while answering this question.

1. If the queue is empty, then the packet reaches the head of the queue in the current round $R(t)$.

2. If the queue is already backlogged, then the packet reaches the head of the queue in the round right after the packet in front of it finishes, F_{i-1}^α. ("F" for "finish.") [1]

[1] We are dropping a few tiny $+1$ and -1 terms here.

Combining the two cases, we have in general

$$S_i^\alpha = \max(R(t), F_{i-1}^\alpha). \tag{3}$$

Now, in what round does this packet finish? This depends *only* on the length of the packet, since in each round exactly one of its bits is sent out. Therefore

$$F_i^\alpha = S_i^\alpha + p_i^\alpha, \tag{4}$$

where p_i^α is the size of the i^{th} packet of flow α.

Using Equations 3 and 4, we can determine the finish round of every packet in every queue. We can do this as soon as the packet is enqueued, but it requires much less state to do it when each packet reaches the head of a queue.

We emphasize, again, that these formulae are independent of the number of backlogged queues. Using round numbers "normalizes" out the number of flows and is an important point.

2.2 Packet-level fair queueing

We are now in a position to emulate bit-by-bit fair queueing at the packet-level. The emulation is simple:

Send the packet which has the smallest finishing round number.

This approximates bit-by-bit fair queueing by "catching up" to the what it *would have done* on a discrete packet-by-packet basis. The packet sent out next is the packet which had been starved the most while sending out the previous packet (from some other queue in general). The algorithm is illustrated in Figure 1.

This is not the only emulation possible, but it seems like the most intuitive one. Some other ones don't work; for instance, what if we had sent out packets by the largest finishing round number? This won't work because a flow sending very large packets would always be chosen. A different emulation might choose the smallest starting round number. Thinking about whether this can be made to work (or not) is left as an exercise to the reader.

2.3 Some implementation details

Some implementation details are worth nailing down.

2.3.1 Buffer management

Each queue in the fair queueing system needs to manage its buffers. First off, preallocating a fixed amount of memory for each queue and drop whenever a queue becomes full is a bad idea, since we don't know how traffic is going to be distributed across queues.

A better idea is to have a global buffer pool, and the queues grow as necessary into the pool.

The most common strategy for dropping packets in fair queueing is to drop from the longest queue or from the queue with largest finish number (they aren't always the same and they have different fairness properties). Presumably one can run RED on these queues if one likes, too. One might drop-from-front, or from the tail, as well.

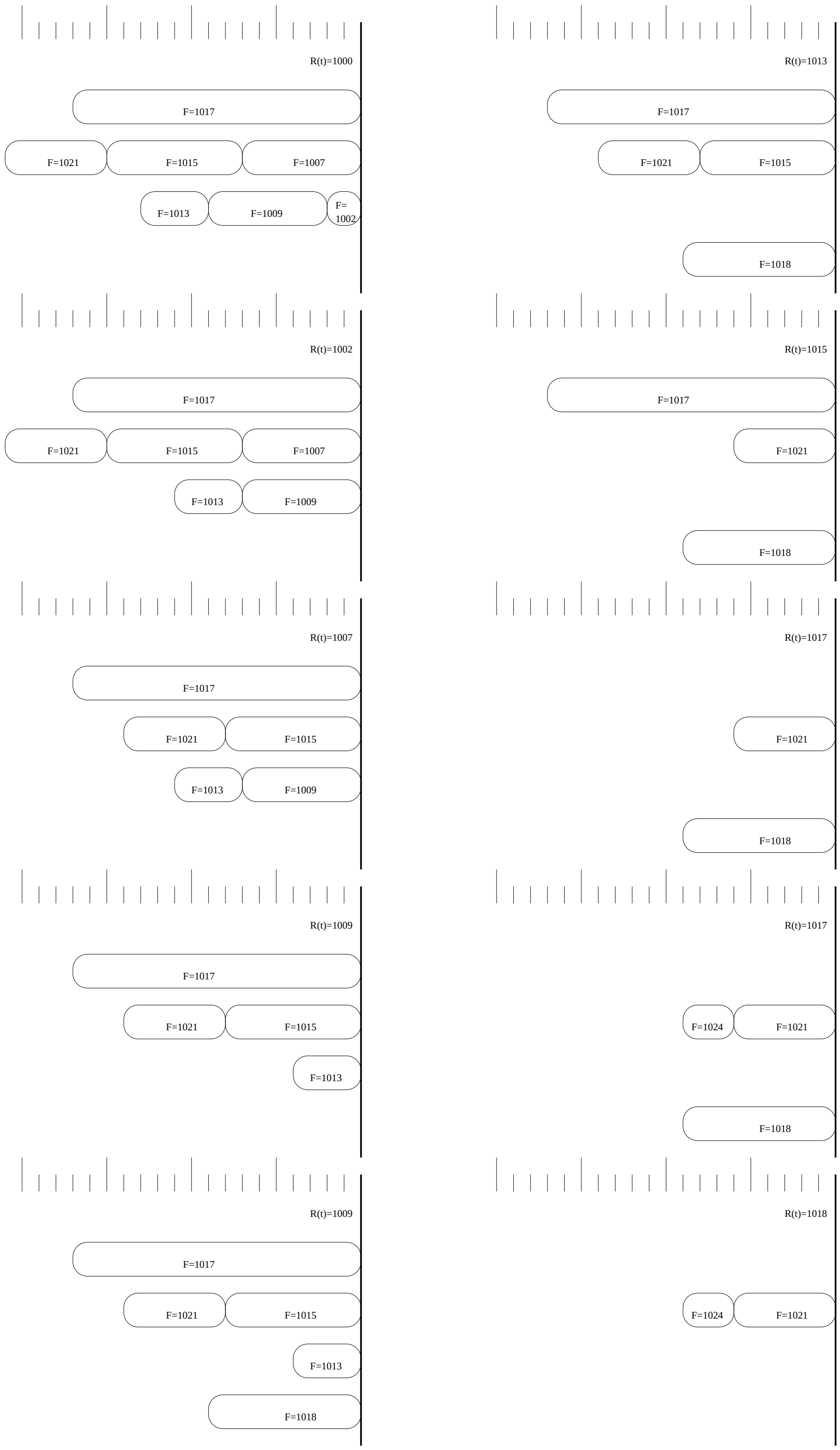

Figure 1: Fair queueing in action. (Thanks to abhi shelat for figure.)

2.3.2 State maintenance

Do we need to maintain the finishing round F_i^α for every packet? In other words, do we need to maintain *per-packet state*?.

No, since the finishing round can be calculated on the fly. The router maintains the finish round F_i^α for the packets on the front of the queue, and when a packet gets sent out, it uses the length of the packet after it to calculate the F_{i+1}^α of the new head of the queue.

The trick is figuring out what finish round number we should give to a packet that arrives into an empty queue. One approach that works well is to give it a start round number equal to the finish number (or start round number) of the packet currently in service.

2.3.3 Difficulties

In practice, fair queueing is challenging because classification may be slow; the more problematic issue is the need to maintain metadata pointers to memory not on the central chip of the router if one wants to implement "pure" flow-level fair queueing. A "core" router may have to maintain 50,000–100,000 flows, and have to process each packet in under 500ns (for 1 Gbps links) and 50ns (for 10 Gpbs links). It seems practical today to build routers with a large number of queues, provided there's some fixed number of them. A large, variable number of queues is particularly challenging and difficult to implement in high-speed routers.

3 CSFQ: *Core*-Stateless Fair Queueing

3.1 Overview

In the first half of this lecture, we discussed a fair queuing scheme that divides the available bandwidth according to max-min fairness. The drawback of the FQ approach was that costly per-flow statistics needed to be maintained. In a typical backbone router, there may be 50,000-100,000 flows at a given time. At gigabit speeds, there may not be enough time or memory to manage per-flow meta-data.

We now turn to an approximation to FQ that refines it in a way that circumvents the problem of maintaining per-flow statistics. The *Core*-Stateless FQ scheme, CSFQ, distinguishes core routers, the higher-speed and busier routers at the "core" of an Internet AS backbone from edge routers. In typical deployment, edge routers might handle thousands of flows, while core routers might handle 50k-100k flows. CSFQ exploits this gap by delegating the management of per-flow statistics to the edge routers. Edge routers then share this information with core routers by labeling each packet that they forward. Core routers, in turn, can use the labels to allocate bandwidth fairly among all incoming flows. It is important to realize that in the case of CSFQ, edge routers run essentially the same algorithm as core routers (including probabilistically dropping incoming packets); however, edge routers have the added responsibility of maintaining per-flow state. In general, of course, edge and core routers in such an approach could run very different algorithms.

Here are the key points of CSFQ:

1. Dynamic Packet State: Edge routers label each packet with an estimate of the arrival rate for each flow. Per-flow statistics are maintained here.

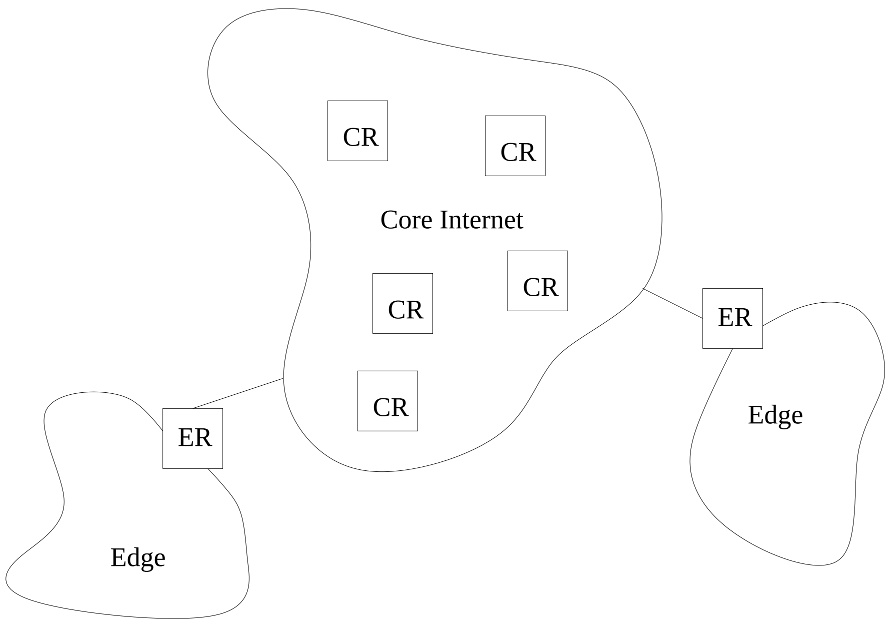

Figure 2: The core and the edges of an Internet backbone network. CR = core router; ER = edge router.

2. Core routers use (1) estimated arrival rates provided on packet labels, and (2) an internal measure of fair-share, to compute the probability of dropping each incoming packet. Every packet that is accepted is processed and relabeled with new arrival rate information.

3. The estimation procedure for the "fair-share" value convergences rapidly to the optimal value. Cheaters cannot win too much extra bandwidth.

There are two goals for a CSFQ router:

1. Maintain max-min fairness for bandwidth allocation.

2. Avoid having to keep per-flow statistics in high-speed core routers.

Note that goal (2) prevents a core router from maintaining per-flow queues. Therefore, once a packet has been accepted by a core router, it sits in one of a small number of queues until it is eventually processed. Hence, the only action the core router can take in order to achieve (1) is to drop packets from greedy flows. Notably absent is the ability to schedule when a packet is to be sent. In order to avoid patterns of synchronization, packet dropping is done probabilistically using both information accumulated and added to the packet by an edge router, and a global parameter estimated by the core router.

3.2 Dynamic packet state

In order to achieve max-min fairness, a router needs to know the arrival rate, $r_i(t)$ for each flow. Under CSFQ, edge routers are assigned the task of estimating these rates. This estimation is computed by taking an exponentially weighted average of samples. In this paper, the weighting is slightly more complicated than previous exponentially weighted schemes that we have discussed.

That is: if t_i^k, l_i^k are the arrival time and length of the kth packet in flow i, then $r_i(t)$ is computed according to the equation

$$r_i^{new} = (1 - e^{-T_i^k/K})\frac{l_i^k}{T_i^k} + e^{-T_i^k/K} r_i^{old}, \tag{5}$$

where $T_i^k = t_i^k - t_i^{k-1}$ and K is some constant. The T_i^k is the i^{th} interarrival time, $t_i^k - t_i^{k-1}$. Instead of using a fixed constant as their weighting factor, the authors argue that the $e^{-T_i^k/K}$ weight converges faster under a broader set of situations and isn't as sensitive to the specific packet-length distribution. As each incoming packet arrives, the new r_i is computed for this flow, a decision about whether to drop the packet is made, and if the router decides to forward the packet, the packet is *labeled* with r_i and forwarded. This idea of setting some information in the header of the packet is called *dynamic packet state* and is an architectural concept that generalizes beyond CSFQ.

3.3 Fair-share rate

Given a set of flows with arrival rates, $r_1(t), r_2(t), ..., r_n(t)$, each edge and core router must determine the fair-share bandwidth according to max-min by assigning each flow the minimum of their requested bandwidth and the fair-share bandwidth left: *i.e.,* it needs to compute the fair-share rate according to Equation 1. Suppose there's some way of calculating α (this is easy for an edge to do, but we'll see in a bit how a core router might estimate it).

Given α, then enforcing a fair rate to incoming traffic isn't too hard. This can be done using random packet drops, according to the following equation.

$$P(\text{drop packet from stream } i) = \max(1 - \frac{\alpha}{r_i(t)}, 0) \tag{6}$$

If we have three flows that transmit at rates $r_1 = 8, r_2 = 6, \text{and } r_3 = 2$ Mbps and these flows are sharing a 10 Mbps link, then $\alpha = 4$ Mbps. Then, if packets were dropped with probabilities $0.5, 0.33$, and 0 respectively, a max-min-fair allocation can be achieved.

Hence, a router needs only two pieces of information in order achieve max-min fairness: α, which is one number per outgoing link at a router, and $r_i(t)$, the arrival rate for each flow. Fortunately, the edge routers calculate $r_i(t)$ using Equation 5 and provide it to the core routers using dynamic packet state, so what remains is the estimation of α. In addition to the architectural concept of dynamic packet state, a noteworthy intellectual contribution of this paper is how α may be estimated at the core routers.

The α fair-share parameter determines the rate at which the router accepts (and therefore processes) packets. So let us define $F(\alpha)$ as the core router's acceptance rate in bits/s. If max-min fairness according to Equation 1 is made to hold, then

$$F(\alpha) = \sum_i \min(r_i(t), \alpha) \tag{7}$$

That is, the aggregate acceptance rate for the router is the sum of the rates for each of the router's flows. If a flow is below its fair-share, then it will contribute $r_i(t)$, and otherwise, it will be capped at its fair-share limit.

The big picture of our strategy is as follows. Each router estimates $F(\alpha)$, the total (aggregate) rate at which packets are accepted into (forwarded by) the router. At the same time, each core router estimates the aggregate input rate (A) into each of its egress links; this is independent of the number of flows and can be done using an equation similar to Equation 5. Together, these two pieces of information, together with knowledge of whether the link (of capacity C) is "congested" or not, allow the router to *estimate* $\tilde{\alpha}$, an estimate for α. This estimation relies on the behavior of the function $F(\tilde{\alpha})$ as a function of $\tilde{\alpha}$.

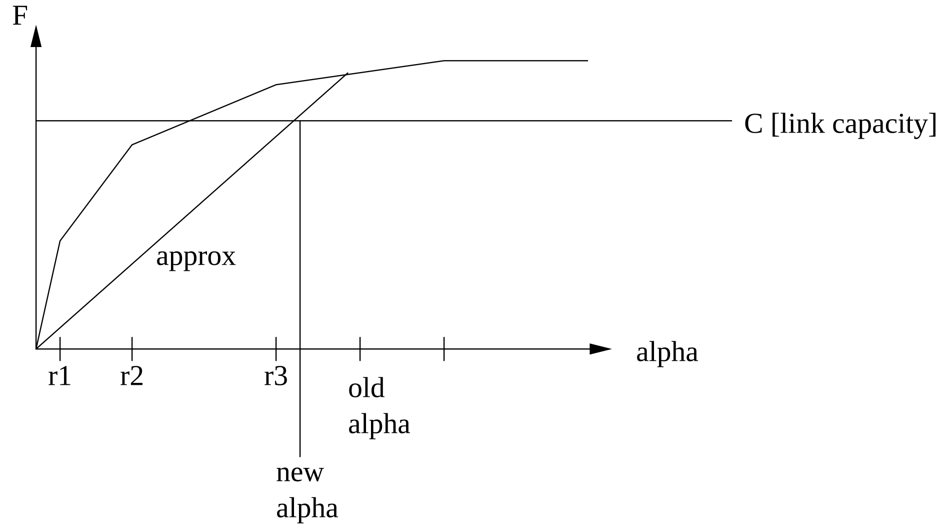

Figure 3: $F(\tilde{\alpha})$ vs $\tilde{\alpha}$.

So consider the graph shown in Figure 3 of $F(\cdot)$ as a function of $\tilde{\alpha}$, and assume that the flow arrival rates are in non-decreasing order such that $r_1 \leq r_2 \leq \ldots \leq r_n$.

When $\tilde{\alpha}$ is less than r_1, the aggregate acceptance rate is $n \times \tilde{\alpha}$ and the slope of the curve is n. However, as soon as $\alpha > r_1$, then the first stream wil be capped at r_1, and the aggregate max-min acceptance rate will now be $(n-1)\tilde{\alpha} + r_1$. That is, the function now has slope $n-1$, so it is flatter. Continuing thus, it is easy to see that $F(\tilde{\alpha})$ is continuous, piecewise linear, with n pieces, with the slope of each piecewise linear portion (as $\tilde{\alpha}$ increases) smaller than the previous one (ending in a line parallel to the horizontal axis when $\tilde{\alpha}$ exceeds the largest input rate, r_n). It is therefore a *concave* function.

Hence, in order to determine $\tilde{\alpha}$ so as to use exactly the capacity in the outbound link, the core router needs to solve the equation $F(\tilde{\alpha}) = C$. Whereas an edge router can store n pieces of information that encode the curve $F(\cdot)$ (this information is simply the arriving per-flow rates), a core router cannot. Instead, the core router simply approximates $F(\cdot)$ with a straight line. If the current acceptance rate (forwarding rate) is $\tilde{F}$ and we already have an estimate $\tilde{\alpha}$ for α, then we can approximate

$$F(\alpha) \approx \frac{\tilde{F}}{\tilde{\alpha}}\alpha. \tag{8}$$

During the processing of packets, the core router simply:

1. Observes the system over a given time period.

2. If the system is "uncongested" then the acceptance parameter, α is increased to $\max_i(r_i)$. With this value, the probability of dropping a packet is 0 for all flows.

3. If the system is "congested" then α is too large, and we need to re-estimate it. The new α is calculated by setting to the solution to the equation $F(\alpha) = C$, where $F(\alpha)$ is approximated using Equation 8.

Because $F()$ is concave, the core router will eventually converge on the correct α value. The only remaining issue is how a router decides if one of its links is "congested" or not. This is done by looking at the estimate of the aggregate arrival rate A and seeing if it is smaller than C for a period of K ms ("uncongested") or larger ("congested"). If neither is true, then α is left unchanged.

In summary, CSFQ is a hybrid scheme that asymptotically converges to max-min fairness. It incorporates two neat ideas: a decomposition of edge and core routers with dynamic packet state transferring information to the core routers, and a simple technique to estimate the fair rate in a core router. The technique generalizes to weighted fair queueing too, although the case of different weights in each router in a CSFQ island is ill-defined and not supported. The paper reports performance results under various conditions, although most of the evaluated conditions don't dramatically vary the number of flows or cause bandwidth oscillations.

Of course, CSFQ suffers from some drawbacks too. All routers in an island need to be simultaneously upgraded; dynamic packet state needs to be implemented somehow, and the proposal is to use the IP fragmentation header fields; and the consequences of buggy or malicious routers needs to be explored.