Universal Packet Scheduling

Radhika Mittal[†] *Rachit Agarwal*[†] *Sylvia Ratnasamy*[†] *Scott Shenker*[†‡]

[†]*UC Berkeley* [‡]*ICSI*

Abstract

In this paper we address a seemingly simple question: *Is there a universal packet scheduling algorithm?* More precisely, we analyze (both theoretically and empirically) whether there is a single packet scheduling algorithm that, at a network-wide level, can perfectly match the results of *any* given scheduling algorithm. We find that in general the answer is "no". However, we show theoretically that the classical Least Slack Time First (LSTF) scheduling algorithm comes closest to being universal and demonstrate empirically that LSTF can closely replay a wide range of scheduling algorithms in realistic network settings. We then evaluate whether LSTF can be used *in practice* to meet various network-wide objectives by looking at popular performance metrics (such as mean FCT, tail packet delays, and fairness); we find that LSTF performs comparable to the state-of-the-art for each of them. We also discuss how LSTF can be used in conjunction with active queue management schemes (such as CoDel) without changing the core of the network.

1 Introduction

There is a large and active research literature on novel packet scheduling algorithms, from simple schemes such as priority scheduling [31], to more complicated mechanisms to achieve fairness [16, 29, 32], to schemes that help reduce tail latency [15] or flow completion time [7], and this short list barely scratches the surface of past and current work. In this paper we do not add to this impressive collection of algorithms, but instead ask if there is a single *universal* packet scheduling algorithm that could obviate the need for new ones. In this context, we consider a packet scheduling algorithm to be both how packets are served inside the network (based on their time of arrival and their packet header) and how packet header fields are initialized at the edge of the network; this definition includes all the classical scheduling algorithms (FIFO, LIFO, priority, round-robin) as well as algorithms that incorporate dynamic packet state [15, 35, 36].

We can define a universal packet scheduling algorithm (hereafter UPS) in two ways, depending on our viewpoint on the problem. From a theoretical perspective, we call a packet scheduling algorithm *universal* if it can replay any *schedule* (the set of times at which packets arrive to and exit from the network) produced by any other scheduling algorithm. This is not of practical interest, since such schedules are not typically known in advance, but it offers a theoretically rigorous definition of universality that (as we shall see) helps illuminate its fundamental limits (i.e., which scheduling algorithms have the flexibility to serve as a UPS, and why).

From a more practical perspective, we say a packet scheduling algorithm is universal if it can achieve different desired performance objectives (such as fairness, reducing tail latency, minimizing flow completion times). In particular, we require that the UPS should match the performance of the best known scheduling algorithm for a given performance objective. [1]

The notion of universality for packet scheduling might seem esoteric, but we think it helps clarify some basic questions. If there exists no UPS then we should *expect* to design new scheduling algorithms as performance objectives evolve. Moreover, this would make a strong argument for switches being equipped with programmable packet schedulers so that such algorithms could be more easily deployed (as argued in [33]; in fact, it was the eloquent argument in this paper that caused us to initially ask the question about universality).

However, if there is indeed a UPS, then it changes the lens through which we view the design and evaluation of packet scheduling algorithms: e.g., rather than asking whether a new scheduling algorithm meets a performance objective, we should ask whether it is easier/cheaper to

[1] For this definition of universality, we allow the header initialization to depend on the objective being optimized. That is, while the basic scheduling operations must remain constant, the header initialization can depend on whether you are seeking fairness or minimal flow completion time.

implement/configure than the UPS (which could also meet that performance objective). Taken to the extreme, one might even argue that the existence of a (practical) UPS greatly diminishes the need for programmable *scheduling* hardware.[2] Thus, while the rest of the paper occasionally descends into scheduling minutae, the question we are asking has important practical (and intriguing theoretical) implications.

This paper starts from the theoretical perspective, defining a formal model of packet scheduling and our notion of replayability in §2. We first prove that there is no UPS, but then show that Least Slack Time First (LSTF) [24] comes as close as any scheduling algorithm to achieving universality. We also demonstrate empirically (via simuation) that LSTF can closely approximate the schedules of many packet scheduling algorithms. Thus, while not a perfect UPS in terms of replayability, LSTF comes very close to functioning as one.

We then take a more practical perspective in §3, showing (via simulation) that LSTF is comparable to the state of the art in achieving various objectives relevant to an application's performance. We investigate in detail LSTF's ability to minimize mean flow completion time, minimize tail latencies, and achieve per-flow fairness. We also consider how LSTF can be used in multi-tenant situations to achieve multiple objectives simultaneously.

LSTF is only a packet scheduling algorithm, and does not provide any active queue management. Rather than augmenting the basic LSTF logic with a queue management algorithm, we show in §4 that LSTF can, instead, be used to implement AQM at the edge of the network. This novel approach to AQM is a contribution in itself, as it allows the algorithm to be upgraded without changing internal routers.

We then discuss the feasibility of implementing LSTF in routers (§5) and provide a brief overview of related work (§6) before concluding with a discussion of open questions in §7.

2 Theory: Replaying Schedules

This section delves into the theoretical viewpoint of a UPS, in terms of its ability to *replay* a given schedule.

2.1 Definitions and Overview

Network Model: We consider a network of store-and-forward routers connected by links. The input load to the network is a fixed set of packets $\{p \in P\}$, their arrival times $i(p)$ (*i.e.,* when they reach the ingress router), and the path $path(p)$ each packet takes from its ingress to its egress router. We assume no packet drops, so all packets eventually exit. Every router executes a nonpreemptive scheduling algorithm which need not be work-conserving or deterministic and may even involve oracles that know about future packet arrivals. Different routers in the network may use different scheduling logic. For each incoming load $\{(p, i(p), path(p))\}$, a collection of scheduling algorithms $\{A_\alpha\}$ (router α implements algorithm A_α) will produce a set of packet output times $\{o(p)\}$ (the time a packet p exits the network). We call the set $\{(path(p), i(p), o(p))\}$ a *schedule*.

Replaying a Schedule: Applying a different collection of scheduling algorithms $\{A'_\alpha\}$ to the same set of packets $\{(p, i(p), path(p))\}$ produces a new set of output times $\{o'(p)\}$. We say that $\{A'_\alpha\}$ *replays* $\{A_\alpha\}$ on this input if and only if $\forall p \in P$, $o'(p) \leq o(p)$.[3]

Universal Packet Scheduling Algorithm: We say a schedule $\{(path(p), i(p), o(p))\}$ is *viable* if there is at least one collection of scheduling algorithms that produces that schedule. We say that a scheduling algorithm is *universal* if it can replay *all* viable schedules. While we allowed significant generality in defining the scheduling algorithms that a UPS seeks to replay (demanding only that they be nonpreemptive), we insist that the UPS itself obey several practical constraints (although we allow it to be preemptive for theoretical analysis, but then quantitatively analyze the nonpreemptive version in §2.3).[4] The three practical constraints we impose on a UPS are:

(1) Uniformity and Determinism: A UPS must use the same deterministic scheduling logic at every router.

(2) Limited state used in scheduling decisions: We restrict a UPS to using only (i) packet headers, and (ii) static information about the network topology, link bandwidths, and propagation delays. It cannot rely on oracles or other external information. However, it can modify the header of a packet before forwarding it (resulting in *dynamic packet state* [36]).

(3) Limited state used in header initialization: We assume that the header for a packet p is initialized at its ingress node. The additional information available to the ingress

[2] Note that the case for programmable hardware as made in recent work on P4 and the RMT switch [11, 12] remains: these systems target programmability in header parsing and in how a packet's processing pipeline is defined (i.e., how forwarding 'actions' are applied to a packet). The P4 language does not currently offer primitives for scheduling and, perhaps more importantly, the RMT switch does not implement a programmable packet scheduler; we hope our results can inform the discussion on whether and how P4/RMT might be extended to support programmable scheduling.

[3] We allow the inequality because, if $o'(p) < o(p)$, one can delay the packet upon arrival at the egress node to ensure $o'(p) = o(p)$.

[4] The issue of preemption is somewhat complicated. Allowing the original scheduling algorithms to be preemptive allows packets to be fragmented, which then makes replay extremely difficult even in simple networks (with store-and-forward routers). However, disallowing preemption in the candidate UPS overly limits the flexibility and would again make replay impossible even in simple networks. Thus, we take the seemingly hypocritical but only theoretically tractable approach and disallow preemption in the original scheduling algorithms but allow preemption in the candidate UPS. In practice, when we care only about approximately replaying of schedules, the distinction is of less importance, and we simulate LSTF in the nonpreemptive form.

for this initialization is limited to: (i) $o(p)$ from the original schedule[5] and (ii) $path(p)$. Later, we extend the kinds of information the header initialization process can use, and find that this is a key determinant in whether one can find a UPS.

We make three observations about the above model. First, our model assumes greater capability at the edge than in the core, in keeping with common assumptions that the network edge is capable of greater processing complexity, exploited by many architectural proposals [13, 30, 35]. Second, when initializing a packet p's header, a UPS can only use the input time, output time and the path information for p itself, and must be *oblivious* [19] to the corresponding attributes for *other* packets in the network. Finally, the key source of impracticality in our model is the assumption that the output times $o(p)$ are known at the ingress. However, a different interpretation of $o(p)$ suggests a more practical application of replayability (and thus our results): *if we assign $o(p)$ as the "desired" output time for each packet in the network, then the existence of a UPS tells us that if these goals are viable then the UPS will be able to meet them.*

2.2 Theoretical Results

For brevity, in this section we only summarize our key theoretical results. Interested readers can find detailed proofs in Appendix A.

Existence of a UPS under omniscient initialization: Suppose we give the header-initialization process extensive information in the form of times $o(p,\alpha)$ which represent when p was scheduled by router α in the original schedule. We can then insert an n-dimensional vector in the header of every packet p, where the i^{th} element contains $o(p,\alpha_i)$ with α_i being the i^{th} hop in $path(p)$. Every time a packet arrives at a router, the router can pop the value at the head of this vector and use that as its priority (earlier values of output times get higher priority). This can perfectly replay any viable schedule (proof in Appendix A.2), which is not surprising, as having such detailed knowledge of the internal scheduling of the network is tantamount to knowing all the scheduling decisions made by the original algorithm. For reasons discussed previously, our definition limited the information available to the output time from the network as a whole, and not from each individual router; we call this *black-box* initialization.

Nonexistence of a UPS under black-box initialization: We can prove by counter-example (described in Appendix A.3) that *there is no UPS* under the conditions stated in §2.1. Given this impossibility result, we now ask *how close can we get to a UPS?*

Natural candidates for a near-UPS: Simple priority scheduling [6] can reproduce all viable schedules on a single router, so it would seem to be a natural candidate for a near-UPS. However, for multihop networks it may be important to make the scheduling of a packet dependent on what has happened to it earlier in its path. For this, we consider Least Slack Time First (LSTF) [24].

In LSTF, each packet p carries its slack value in the packet header, which is initialized to $slack(p) = (o(p) - i(p) - t_{min}(p, src(p), dest(p)))$ at the ingress, where $src(p)$ is the ingress of p; $dest(p)$ is the egress of p; $t_{min}(p,\alpha,\beta)$ is the time p takes to go from router α to router β in an empty network. Therefore, the slack of a packet is merely its total time in the network (*i.e.,* the time between arrival and departure in the original schedule) minus the minimum time it requires to traverse the network and indicates the maximum queueing time (excluding the transmission time at any router) that the packet could tolerate without violating the replay condition. Each router, then, schedules the packet which has the least remaining slack at the time when its last bit is transmitted. Before forwarding the packet, the router overwrites the slack value in the packet's header with its remaining slack (*i.e.,* the previous slack time minus how much time it waited in the queue before being transmitted).

Note that there are other ways to implement this algorithm, such as having a static packet header as in Earliest Deadline First (EDF) and using additional state in the routers (reflecting the value of t_{min}) to compute the priority for a packet at each router, but here we chose to use an approach with dynamic packet state. We provide more details about EDF and prove its equivalence to LSTF in Appendix A.5.

Key Results: Our analysis shows that the difficulty of replay is determined by the number of *congestion points*, where a *congestion point* is defined as a node where a packet is forced to "wait" during a given schedule. Our theorems show the following key results:

1. Priority scheduling can replay all viable schedules with no more than one congestion point per packet, and there are viable schedules with no more than two congestion points per packet that it cannot replay. (Proof in Appendix A.6.)

2. LSTF can replay all viable schedules with no more than two congestion points per packet, and there are viable schedules with no more than three congestion points per packet that it cannot replay. (Proof in Appendix A.7.)

3. There is no scheduling algorithm (obeying the aforementioned constraints on UPSs) that can replay *all* viable schedules with no more than three congestion points per packet, and the same holds for larger numbers of congestion points. (Proof in Appendix A.3.)

[5] Note that this ingress router can directly observe $i(p)$ as the time the packet arrives.

[6] By simple priority scheduling, we mean that the ingress assigns priority values to the packets and the routers simply schedule packets based on these static priority values.

Topology	Link Utilization	Scheduling Algorithm	Fraction of packets overdue Total	> T
I2:1Gbps-10Gbps	70%	Random	0.0021	0.0002
I2 1Gbps-10Gbps	10% 30% 50% 90%	Random	0.0007 0.0281 0.0221 0.0008	0.0 0.0017 0.0002 4×10^{-6}
I2:1Gbps-1Gbps I2:10Gbps-10Gbps I2 / 10	70%	Random	0.0204 0.0631 0.0127	8×10^{-6} 0.0448 0.00001
RocketFuel Datacenter	70%	Random	0.0246 0.0164	0.0063 0.0154
I2 1Gbps-10Gbps	70%	FIFO FQ SJF LIFO FQ/FIFO+ FQ: SJF/FIFO	0.0143 0.0271 0.1833 0.1477 0.0152 0.0297	0.0006 0.0002 0.0019 0.0067 0.0004 0.0003

Table 1: LSTF Replayability Results across various scenarios. T represents the transmission time of the bottleneck link.

Main Takeaway: *LSTF is closer to being a UPS than simple priority scheduling, and no other candidate UPS can do better in terms of handling more congestion points.*

Intuition: The reason why LSTF is superior to priority scheduling is clear: by carrying information about previous delays in the packet header (in the form of the *remaining* slack value), LSTF can "make up for lost time" at later congestion points, whereas for priority scheduling packets with low priority might repeatedly get delayed (and thus miss their target output times). The reason LSTF can always handle up to two congestion points per packet is that, for this case, each congestion point is either the first or the last point where the packet waits and we can prove that any extra delay seen at the first congestion point during the replay can be naturally compensated for at the second. With three or more congestion points there is no way for LSTF (or any other packet scheduler) to know how to allocate the slack among them; one can create counterexamples where unless the scheduling algorithm makes precisely the right choice in the earlier congestion points, at least one packet will miss its target output time.

2.3 Empirical Results

The previous section clarified the theoretical limits on a *perfect* replay. Here we investigate, via ns-2 simulations [5], how well (a nonpreemptable version of) LSTF can *approximately* replay schedules in realistic networks.

Experiment Setup: *Default scenario.* We use a simplified Internet-2 topology [2], identical to the one used in [26] (consisting of 10 routers and 16 links in the core). We connect each core router to 10 edge routers using 1Gbps links and each edge router is attached to an end host via a 10Gbps link.[7] The number of hops per packet is in the range of 4 to 7, excluding the end hosts. We refer to this topology as I2:1Gbps-10Gbps. Each end host generates UDP flows using a Poisson inter-arrival model, and our default scenario runs at 70% utilization. The flow sizes are picked from a heavy-tailed distribution [8, 9]. Since our focus is on packet scheduling, not dropping policies, we use large buffer sizes that ensure no packet drops.

Varying parameters. We tested a wide range of experimental scenarios by varying the following parameters from their default values: access vs core link bandwidths, network utilization, network topology, network scale and traffic workloads. We present results for a small subset of these scenarios here: (1) the default scenario with network utilization varied from 10-90% (2) the default scenario but with 1Gbps link between the endhosts and the edge routers (I2:1Gbps-1Gbps), with 10Gbps links between the edge routers and the core (I2:10Gbps-10Gbps) and with all link capacities in the I2:1Gbps-1Gbps topology reduced by a factor of 10 (I2 / 10) and (3) the default scenario applied to two different topologies, a bigger Rocketfuel topology [34] (with 83 routers and 131 links in the core) and a full bisection bandwidth datacenter (fat-tree) topology from [7] (with 10Gbps links). Note that our other results were generally consistent with those presented here.

Scheduling algorithms. Our default case, which we expected to be hard to replay, uses completely arbitrary schedules produced by a *random* scheduler (which picks the packet to be scheduled randomly from the set of queued up packets). We also present results for more traditional packet scheduling algorithms: FIFO, LIFO, fair queuing [16], and SJF (shortest job first using priorities). We also looked at two scenarios with a mixture of scheduling algorithms: one where half of the routers run FIFO+ [15] and the other half run fair queuing, and one where fair queueing is used to fair share between two classes of traffic, with one class being scheduled with SJF and the other class being scheduled with FIFO.

Evaluation Metrics: We consider two metrics. First, we measure the fraction of packets that are overdue (i.e., which do not meet the original schedule's target). Second, to capture the *extent* to which packets fail to meet their targets, we measure the fraction of packets that are overdue by more than a threshold value T, where T is one transmission time on the bottleneck link ($\approx 12\mu s$ for 1Gbps). We pick this value of T both because it is sufficiently small that we can assume being overdue by this small amount is of negligible practical importance, and also because this is the order of violation we should expect given that our implementation of LSTF is non-preemptive. While we may have many small violations of replay (because of non-preemption), one would hope that

[7] We use higher than usual access bandwidths for our default scenario to increase the stress on the schedulers in the routers. We also present results for smaller access bandwidths, which have better replay performance.

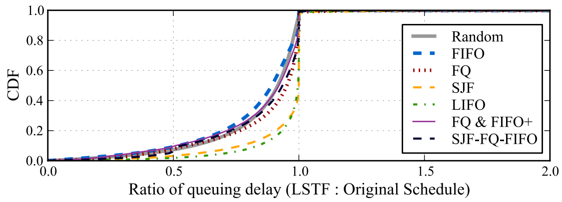


Figure 1: Ratio of queuing delay with varying packet scheduling algorithms, on the default Internet-2 topology at 70% utilization.

most such violations are less than T.

Results: Table 1 shows the simulation results for LSTF replay for various scenarios, which we now discuss.

(1) Replayability. Consider the column showing the fraction of packets overdue. In all but three cases (we examine these shortly) over 97% of packets meet their target output times. In addition, the fraction of packets that did not arrive within T of their target output times is much smaller; even in the worst case of SJF scheduling (where 18.33% of packets failed to arrive by their target output times), only 0.19% of packets are overdue by more than T. Most scenarios perform substantially better: e.g., in our default scenario with Random scheduling, only 0.21% of packets miss their targets and only 0.02% are overdue by more than T. Hence, we conclude that even without preemption LSTF achieves good (but not perfect) replayability under a wide range of scenarios.

(2) Effect of varying network utilization. The second row in Table 1 shows the effect of varying network utilization. We see that at low utilization (10%), LSTF achieves exceptionally good replayability with a total of only 0.07% of packets overdue. Replayability deteriorates as utilization is increased to 30% but then (somewhat surprisingly) improves again as utilization increases. This improvement occurs because with increasing utilization, the amount of queuing (and thus the average slack across packets) in the original schedule also increases. This provides more room for slack re-adjustments when packets wait longer at queues seen early in their paths during the replay. We observed this trend in all our experiments though the exact location of the "low point" varied across settings.

(3) Effect of varying link bandwidths. The third row shows the effect of changing the relative values of access/edge *vs.* core links. We see that while decreasing access link bandwidth (I2:1Gbps-1Gbps) resulted in a much smaller fraction of packets being overdue by more than T (0.0008%), increasing the edge-to-core link bandwidth (I2:10Gbps-10Gbps) resulted in a significantly higher fraction (4.48%). For I2:1Gbps-1Gbps, packets are paced by the endhost link, resulting in few congestion points thus improving LSTF's replayability. In contrast, with I2:10Gbps-10Gbps, both the access and edge links have a higher bandwidth than most core links; hence packets (that are no longer paced at the endhosts or the edges) arrive at the core routers very close to one another and hence the effect of one packet being overdue *cascades* over to the following packets. Decreasing the absolute bandwidths in I2/10, while keeping the ratio between access and edge links the same as that in I2:1Gbps-1Gbps did not produce significant variance in the results over I2:1Gbps-1Gbps, indicating that the relative link capacities have a greater impact on the replay performance than the absolute link capacities.

(4) Effect of varying topology. The fourth row in Table 1 shows our results using different topologies. LSTF performs well in both cases: only 2.46% (Rocketfuel) and 1.64% (datacenter) of packets fail replay. These numbers are still somewhat higher than our default case. The reason for this is similar to that for the I2:10Gbps-10Gbps topology – all links in the datacenter fat-tree topology are set to 10Gbps, while in our simulations, we set half of the core links in the Rocketfuel topology to have bandwidths smaller than the access links.

(5) Varying Scheduling Algorithms. Row five in Table 1 shows LSTF's ability to replay different scheduling algorithms. We see that LSTF performs well for FIFO, FQ, and the combination cases (a mixture of FQ/FIFO+ and having FQ share between FIFO and SJF); e.g., with FIFO, fewer than 0.06% of packets are overdue by more than T. However, there are two problematic cases: SJF and LIFO fare worse with 18.33% and 14.77% of packets failing replay (although only 0.19% and 0.67% of packets are overdue by more than T respectively). The reason stems from a combination of two factors: (1) for these algorithms a larger fraction of packets have a very small slack value (as one might expect from the scheduling logic which produces a larger skew in the slack distribution), and (2) for these packets with small slack values, LSTF *without preemption* is often unable to "compensate" for misspent slack that occurred earlier in the path. To verify this intuition, we extended our simulator to support preemption and repeated our experiments: with preemption, the fraction of packets that failed replay dropped to 0.24% (from 18.33%) for SJF and to 0.25% (from 14.77%) for LIFO.

(6) End-to-end (Queuing) Delay. Our results so far evaluate LSTF in terms of measures that we introduced to test universality. We now evaluate LSTF using the more traditional metric of packet delay, focusing on the queueing delay a packet experiences. Figure 1 shows the CDF of the ratios of the queuing delay that a packet sees with LSTF to the queuing delay that it sees in the original schedule, for varying packet scheduling algorithms. We were surprised to see that most of the packets actually have a smaller queuing delay in the LSTF replay than in the original schedule. This is because LSTF eliminates "wasted waiting", in that it never makes packet A wait behind packet B if packet B is going to have significantly more waiting later in its path.

(7) Comparison with Priorities. To provide a point of comparison, we also did a replay using simple priorities for our default scenario, where the priority for a packet p is set to $o(p)$ (which seemed most intuitive to us). As expected, the resulting replay performance is much worse than LSTF: 21% packets are overdue in total, with 20.69% being overdue by more than T. For the same scenario, LSTF has only 0.21% packets overdue in total, with merely 0.02% packets overdue by more than T.

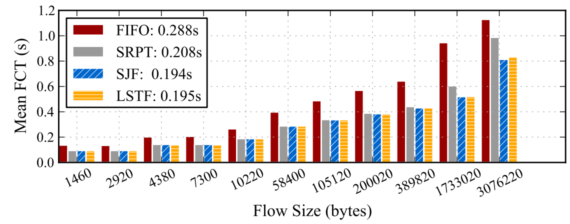
Summary: We observe that, in almost all cases, less than 1% of the packets are overdue with LSTF by more than T. The replay performance initially degrades and then starts improving as the network utilization increases. The distribution of link speeds has a bigger influence on the replay results than the scale of the topology. Replay performance is better for scheduling algorithms that produce a smaller skew in the slack distribution. LSTF replay performance is significantly better than simple priorities replay performance, with the most intuitive priority assignment.

3 Practical: Achieving Various Objectives

While replayability demonstrates the theoretical flexibility of LSTF, it doesn't provide evidence that it would be practically useful. In this section we look at how LSTF can be used *in practice* to meet the following network-wide objectives: minimizing mean flow completion time, minimizing tail latencies, and achieving per-flow fairness. We also consider how LSTF can be used in multi-tenant situations to achieve multiple objectives simultaneously.

Since the knowledge of a previous schedule is unavailable in practice, instead of using a given set of output times (as done in §2.3), we now use heuristics to assign the slacks in an effort to achieve these objectives. Our goal here is not to outperform the state-of-the-art for each objective in all scenarios, but instead we aim to be competitive with the state-of-the-art in most common cases.

In presenting our results for each objective, we first describe the slack initialization heuristic we use and then present some ns-2 [5] simulation results on (i) how LSTF performs relative to the state-of-the-art scheduling algorithm and (ii) how they both compare to FIFO scheduling (as a baseline to indicate the overall impact of specialized scheduling for this objective). As our default case, we use the I2 1Gbps-10Gbps topology using the same workload as in the previous section (running at 70% average utilization). We also present aggregate results at different utilization levels and for variations in the default topology (I2 1Gbps-1Gbps and I2 / 10), for the bigger Rocketfuel topology, and for the datacenter topology (for selected objectives). The switches use non-preemptive scheduling (including for LSTF) and have finite buffers (packets with the highest slack are dropped when the buffer is full). Unless otherwise specified, our experiments use TCP flows



Expt. Setup	Mean FCT (s)			
	FIFO	SRPT	SJF	LSTF
I2 1Gbps-10Gbps at 30% util.	0.189	0.183	0.182	0.182
I2 1Gbps-10Gbps at 50% util.	0.212	0.189	0.185	0.185
I2 1Gbps-10Gbps at 70% util.	0.288	0.208	0.194	0.195
I2 1Gbps-1Gbps at 70% util.	0.252	0.209	0.202	0.202
I2 / 10 at 70% util.	0.898	0.658	0.620	0.621
Rocketfuel at 70% util.	0.304	0.240	0.228	0.228
Datacenter at 70% util.	0.058	0.018	0.016	0.015

Figure 2: The graph shows the mean FCT bucketed by flow size for the I2 1Gbps-10Gbps topology with 70% utilization obtained with FIFO, SRPT and SJF (using priorities and LSTF). The legend indicates the mean FCT across all flows. The table indicates the mean FCTs for varying settings.

with router buffer sizes of 5MB for the WAN simulations (equal to the average bandwidth-delay product for our default topology) and 500KB for the datacenter simulations.

3.1 Mean Flow Completion Time

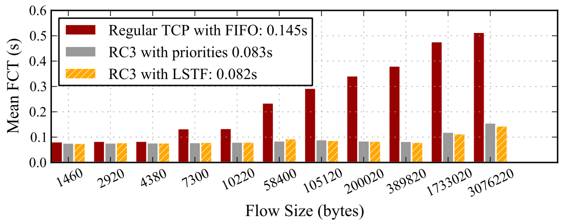
While there have been several proposals on how to minimize flow completion time (FCT) via the transport protocol [17,26], here we first focus on *scheduling*'s impact on FCT, while using standard TCP at the endhosts. In [7] it is shown that (i) Shortest Remaining Processing Time (SRPT) is close to optimal for minimizing the mean FCT and (ii) Shortest Job First (SJF) produces results similar to SRPT for realistic heavy-tailed distribution. Thus, these are the two algorithms we use as benchmarks.

Slack Initialization: The slack for a packet p is initialized as $slack(p) = fs(p) * D$, where $fs(p)$ is the size of the flow to which p belongs and D is a value much larger than the queuing delay seen by any packet in the network. We use a value of $D = 1$ sec for our simulations.

Evaluation: Figure 2 compares LSTF with three other scheduling algorithms – FIFO, SJF and SRPT with *starvation prevention* as in [7] [8]. SJF has a slightly better performance than SRPT, both resulting in a significantly lower mean FCT than FIFO. LSTF's performance is nearly the same as SJF.

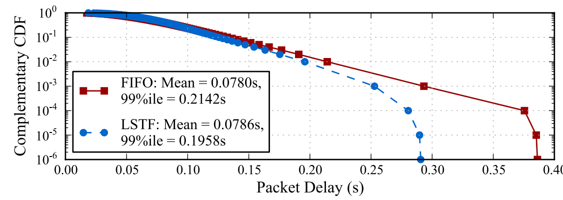
We now look at how in-network scheduling can be used along with changes in the endhost TCP stack to achieve the same objective. We use RC3 [26] as our comparison-point for this objective (as it has better performance than RCP [17] and is simple to implement). In RC3 the senders aggressively send additional packets to quickly use up the available network capacity, but these packets are sent at lower priority levels to ensure that the regular traffic is not penalized. Therefore, it allows near-optimal bandwidth

[8]The router always schedules the earliest arriving packet of the flow which contains the highest priority packet.



Expt. Setup	Mean FCT (s)		
	TCP-FIFO	RC3-priorities	RC3-LSTF
I2 1Gbps-10Gbps at 30% util.	0.145	0.083	0.082
I2 1Gbps-10Gbps at 50% util.	0.159	0.094	0.089
I2 1Gbps-10Gbps at 70% util.	0.180	0.107	0.102
I2 1Gbps-1Gbps at 30% util.	0.134	0.075	0.073
I2 / 10 at 30% util.	0.32	0.215	0.233
Rocketfuel at 30% util.	0.171	0.102	0.101

Figure 3: The graph shows the mean FCT bucketed by flow size for the I2 1Gbps-10Gbps topology with 30% utilization for regular TCP using FIFO and for RC3 using priorities and LSTF. The legend indicates the mean FCT across all flows. The table indicates the mean FCTs for varying settings.



Expt. Setup	Mean Delay (s)		99%ile Delay (s)	
	FIFO	LSTF	FIFO	LSTF
I2 1Gbps-10Gbps at 30% util.	0.0411	0.0411	0.0911	0.0868
I2 1Gbps-10Gbps at 50% util.	0.0515	0.0517	0.1287	0.1195
I2 1Gbps-10Gbps at 70% util.	0.0780	0.0786	0.2142	0.1958
I2 1Gbps-1Gbps at 70% util.	0.0771	0.0771	0.2163	0.216
I2 / 10 at 70% util.	0.5762	0.5765	1.9393	1.9367
Rocketfuel at 70% util.	0.1891	0.1883	3.8139	3.7199
Datacenter at 70% util.	0.0250	0.0239	0.1352	0.1100

Figure 4: Tail packet delays for LSTF compared to FIFO. The graph shows the Complementary CDF of packet delays for the I2 1Gbps-10Gbps topology at 70% utilization with the mean and 99%ile packet delay values indicated in the legend. The table shows the corresponding results for varying setting.

utilization, while maintaining the cautiousness of TCP.

Slack Initialization: The slack for a packet p is initialized as $slack(p) = prio_{rc3} * D$, where $prio_{rc3}$ is the priority of the packet assigned by RC3 and D is a value much larger than the queuing delay seen by any packet in the network. We use a value of $D = 1$ sec for our simulations.

Evaluation: To evaluate RC3 with LSTF, we reuse the ns-3 [6] implementation of RC3 (along with the same TCP parameters used by RC3, such as an initial congestion window of 4), and implement LSTF in ns-3. Figure 3 shows our results. We see that using LSTF with RC3 performs comparable to (and often slightly better than) using priorities with RC3, both giving significantly lower FCTs than regular TCP with FIFO.

3.2 Tail Packet Delays

Clark et. al. [15] proposed the FIFO+ algorithm, where packets are prioritized at a router based on the amount of queuing delay they have seen at their previous hops, for minimizing the tail packet delays in multi-hop networks. FIFO+ is identical to LSTF scheduling where all packets are initialized with the same slack value.

Slack Initialization: All incoming packets are initialized with the same slack value (we use an initial slack value of 1 second in our simulations). With the slack update taking place at every router, the packets that have waited longer in the network queues are naturally given preference over those that have waited for a smaller duration.

Evaluation: We compare LSTF (which, with the above slack initialization, is identical to FIFO+) with FIFO, the primary metric being the 99%ile end-to-end one way delay seen by the packets. Figure 4 shows our results. To better understand the impact of the two scheduling policies on the packet delays, we present our results using UDP flows, which ensures that the input load remains the same in both cases, allowing a fair comparison for the in-network packet-level behaviour. With LSTF, packets that have traversed through more number of hops, and have therefore spent more slack in the network, get preference over shorter-RTT packets that have traversed through fewer hops. While this might produce a slight increase in the mean packet delay, it reduces the tail. This in-line with the observations made in [15].

3.3 Fairness

Fairness is a common scheduling goal, which involves two different aspects: *asymptotic* bandwidth allocation (eventual convergence to the fair-share rate) and *instantaneous* bandwidth allocation (enforcing this fairness on small time-scales, so every flow experiences the equivalent of a per-flow pipe). The former can be measured by looking at long-term throughput measures, while the latter is best measured in terms of the flow completion times of relatively short flows (which measures bandwidth allocation on short time scales). We now show how LSTF can be used to achieve both of these goals, but more effectively the former than the latter. Our slack assignment heuristic can also be easily extended to achieve weighted fair queuing, but we do not present those results here.

Slack Initialization: The slack assignment for fairness works on the assumption that we have some ballpark notion of the fair-share rate for each flow and that it does not fluctuate wildly with time. Our approach to assigning slacks is inspired from [38]. We assign $slack = 0$ to the first packet of the flow and the slack of any subsequent packet p_i is then initialized as:

$$slack(p_i) = max\Big(0, slack(p_{i-1}) + \frac{1}{r_{est}} - \big(i(p_i) - i(p_{i-1})\big)\Big)$$

where $i(p)$ is the arrival time of the packet p at the ingress and r_{est} is an estimate of the fair-share rate r^*. We show that the above heuristic leads to asymptotic fairness, for *any* value of r_{est} that is less than r^*, as long as all flows use the same value. The same heuristic can also be used to

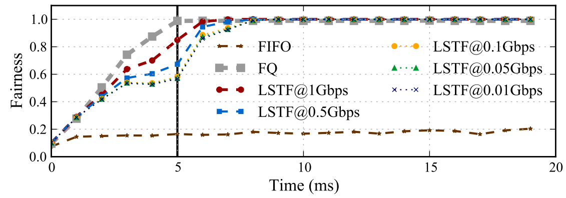


Figure 5: Fairness for long-lived flows on Internet2 topology. The legend indicates the value of r_{est} used for LSTF slack initialization.

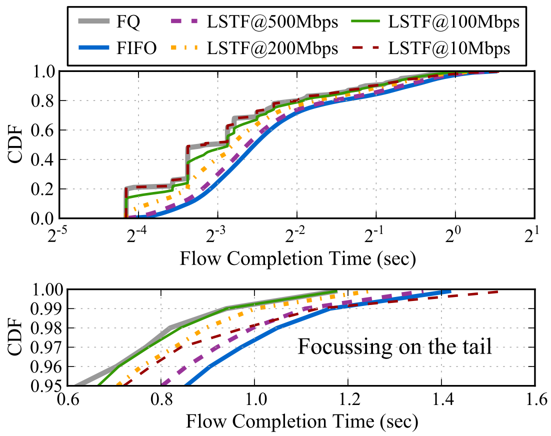


Figure 6: CDF of FCTs for the I2 1Gbps-10Gbps topology at 70% utilization.

provide instantaneous fairness, when we have a complex mix of short-lived flows, where the r_{est} value that performs the best depends on the link bandwidths and their utilization levels. A reasonable value of r_{est} can be estimated using knowledge about the network topology and traffic matrices, though we leave a detailed exploration of this to future work.

Evaluation: Asymptotic Fairness. We evaluate the asymptotic fairness property by running our simulation on the Internet2 topology with 10Gbps edges, such that all the congestion is happening at the core. However, we reduce the propagation delay, to make the experiment more scalable, while the buffer size is kept large (50MB) so that fairness is dominated by the scheduling policy and not by how TCP reacts to packet drops. We start 90 long-lived TCP flows with a random jitter in the start times ranging from 0-5ms. The topology is such that the fair share rate of each flow on each link in the core network (which is shared by up to 13 flows) is around 1Gbps. We use different values for $r_{est} \leq$ 1Gbps for computing the initial slacks and compare our results with fair queuing (FQ). Figure 5 shows the fairness computed using Jain's Fairness Index [23], from the throughput each flow receives per millisecond. Since we use the throughput received by each of the 90 flows to compute the fairness index, it reaches 1 with FQ only at 5ms, after all the flows have started. We see that LSTF is able to converge to perfect fairness, even when r_{est} is 100X smaller than r^*. It converges slightly sooner when r_{est} is closer

Expt. Setup	Avg FCT across bytes (s)			Best r_{est} (Mbps)	Reasonable r_{est} Range (Mbps)
	FIFO	FQ	LSTF		
I2 1Gbps-10Gbps at 30% util.	0.563	0.537	0.538	300	10-900
I2 1Gbps-10Gbps at 50% util.	0.626	0.549	0.555	200	10-800
I2 1Gbps-10Gbps at 70% util.	0.811	0.622	0.632	100	50-200
I2 1Gbps-1Gbps at 70% util.	0.766	0.630	0.652	100	50-400
I2 / 10 at 70% util.	4.838	2.295	2.759	10	10-20
Rocketfuel at 70% util.	0.964	0.796	0.824	100	50-300

Table 2: Mean FCT averaged across bytes for FIFO, FQ and LSTF (with best r_{est} value) across varying settings. The last column indicates the range of r_{est} values that produce mean FCTs within 10% of the best r_{est} result.

to r^*, though the subsequent differences in the time to convergence decrease with decreasing values of r_{est}. The detailed explanation of how this works has been provided in Appendix B for interested readers.

Evaluation: Instantaneous Fairness. As one might expect, the choice of r_{est} has a bigger impact on instantaneous fairness than on asymptotic fairness. A very high r_{est} value would not provide sufficient isolation across flows. On the other hand, a very small r_{est} value can starve the long flows. This is because the assigned slack values for the later packets of long flows with high sequence numbers would be much higher than the actual slack they experience. As a result, they will end up waiting longer in the queues, while the newer flows that keep coming in with smaller sequence numbers (and therefore much smaller slacks) would end up getting a higher precedence.

To verify this intuition, we evaluated our LSTF slack assignment scheme by running our standard workload with a mix of TCP flows ranging from sizes 1.5KB - 3MB on our default I2 1Gbps-10Gbps topology at 70% utilization, with 50MB buffer size. Note that the traffic pattern is now bursty and the instantaneous utilization of a link is often lower or higher than the assigned average utilization level. The CDF of the FCTs thus obtained is shown in Figure 6. As expected, the distribution of FCTs looks very different between FQ and FIFO. FQ isolates the flows from each-other therefore significantly reducing the FCT seen by short to medium size flows, compared to FIFO. The long flows are also helped a little by FQ, again due to the isolation provided from one-another.

LSTF performance varies somewhere in between FIFO and FQ, as we vary r_{est} values between 500Mbps to 10Mbps. A high value of $r_{est} = 500Mbps$ does not provide sufficient isolation and the performance is close to FIFO. As we reduce the value of r_{est}, the "isolation-effect" increases. However, for very small r_{est} values (e.g. 10Mbps), the tail FCTs for the long flows becomes much higher than FQ, due to the starvation effect explained before.

We try to capture this trade-off between isolation for short and medium sized flows and starvation for long flows, by using average FCT across bytes (in other words, the average FCT weighted by flow size) as our key metric. We term the r_{est} value that achieves the sweetest spot in this trade-off as the "best" r_{est} value. The r_{est} values that

Mechanism	A: Mean FCT (s)	B: 99%ile packet delay (s)
A - SJF, B - FIFO Round-robin across A and B	0.188	0.0850
A - LSTF (§3.1), B - LSTF (§3.2) Round-robin across A and B	0.189	0.0822

(a) Fairness across Class A and B

Mechanism	A: Mean FCT (s)	B: 99%ile packet delay (s)
A - SJF, B - FIFO A prioritized over B	0.181	0.1593
A - LSTF (§3.1), B - LSTF (§3.2) B's slacks incremented by K	0.182	0.1533

(b) Class A has higher priority than B

Table 3: Class A wants to minimize mean flow completion time. Class B wants to minimize tail packet delays.

produce average FCT which is within 10% of the value produced by the best r_{est} are termed as "reasonable" r_{est} values. Table 2 presents our results across different settings. We find that (1) LSTF produces significantly lower average FCT than FIFO, performing only slightly worse than FQ (2) As expected, the best r_{est} value decreases with increasing utilization and with decreasing bandwidths (as in the case of I2 / 10 topology), while the range of reasonable r_{est} values gets narrower with increasing utilization and with decreasing bandwidths.

Thus, for instantaneous fairness, LSTF would require knowledge of the network topology (in particular, the link bandwidths) and on-line measurement of traffic matrices and link utilization levels. While these are available in most WANs, they do impose a higher burden on deploying LSTF than on FQ or other such scheduling algorithms.

3.4 Multi-tenancy

As network virtualization becomes more popular, networks are often called upon to support multiple tenants or traffic classes, with each having their own networking objectives. Network providers can enforce isolation across such tenants (or classes of traffic) through static bandwidth provisioning, which can be implemented via dedicated hard-wired links [1, 4] or through multiqueue scheduling algorithms such as fair queuing or round robin [16]. LSTF can work in conjunction with both of these isolation mechanisms to meet the desired performance objective for each tenant (or class of traffic).

Table 3(a) shows our evaluation for this using our default I2 1Gbps-10Gbps topology, with 50MB router buffers, running TCP flows at 70% utilization. Half of the traffic was marked as Class A (targeting small mean flow completion time) and the remaining half as Class B (targeting small tail packet delays), with isolation being enforced across the two classes via round robin. We find that using LSTF with round-robin for isolation performs comparable to doing SJF and FIFO for the two classes with round-robin for isolation.

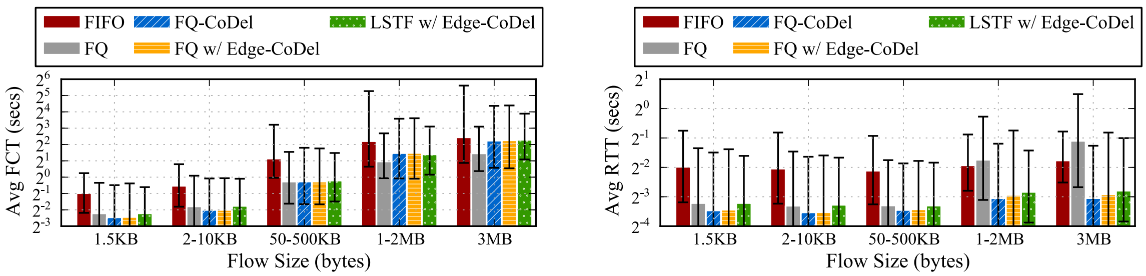
While LSTF can provide some degree of per-flow fairness (as in the previous subsection), it cannot provide such isolation or fairness on a per-class or per-tenant basis. This is because for class-based fairness (which also includes hierarchical fairness) the appropriate slack assignment for a packet at a particular ingress depends on the input from other ingresses (since these packets can belong to the same class). However, if two or more classes/tenants are separated by strict prioritization, LSTF can be used to enforce the appropriate precedence order, along with meeting the individual performance objective for each class. We evaluate this using the same setting as in the previous case, with the results shown in Table 3(b). We assign the Class A and Class B slack as per the LSTF heuristic for the respective objectives and increment the assigned slack of all Class B packets by a large constant K (bigger than the highest slack assigned across packets in Class A by a value which is more than the maximum queuing delay a packet can see in the network). This ensures that Class B always gets a lower priority than Class A and the resulting performance is comparable to doing SJF and FIFO for the two classes and separating them using strict priorities.

4 Incorporating Network Feedback

Context: So far we have considered packet scheduling in isolation, whereas in the Internet today routers send implicit feedback to hosts to via packet drops (or marking, as in ECN). This is often called Active Queue Management (AQM), and its goal is to reduce the per-packet delays while keeping throughput high. We now consider how we might generalize our LSTF approach to incorporate such network feedback as embodied in AQM schemes such as RED [18] and CoDel [27].

LSTF is just a scheduling algorithm and cannot perform AQM on its own. Thus, at first glance, one might think that incorporating AQM into LSTF would require implementing the AQM scheme in each router, which would then require us to find a universal AQM scheme in order to fulfill our pursuit of universality. On the contrary, LSTF enables a novel edge-based approach to AQM based on the following insights: (1) In addition to scheduling packets LSTF produces a very useful by-product, carried by the slack values in the packets, which gives us a precise measure of the one-way queuing delay seen by the packet and can be used for AQM (2) As long as appropriate packets are chosen, it does not matter *where* they are being dropped (or marked) – whether it is inside the core routers or at the edge.

We describe and evaluate our edge-based approach to AQM in the context of CoDel [27], which is considered to be the state-of-the-art AQM scheme for wide area networks. In CoDel, the amount of time a packet has spent in a queue is recorded as the sojourn time. A packet is dropped if its sojourn time exceeds a fixed target (set to 5ms [28]), and if the last packet drop happened beyond a certain interval (initialized to 100ms [28]). As



Expt. Setup	r_{est} (Mbps)	Avg FCT across bytes (s) FIFO	FQ	FQ-CoDel	FQ w/ Edge-CoDel	LSTF w/ Edge-CoDel	Avg RTT across bytes (s) FIFO	FQ	FQ-CoDel	FQ w/ Edge-CoDel	LSTF w/ Edge-CoDel
I2 1Gbps-10Gbps at 70% util.	100	0.811	0.622	0.642	0.633	0.641	0.0756	0.0733	0.0642	0.0646	0.0661
I2 1Gbps-1Gbps at 70% util.	100	0.766	0.630	0.642	0.637	0.658	0.0716	0.0702	0.0639	0.0643	0.0666
I2 / 10 at 30% util.	40	0.918	0.836	0.897	0.887	0.907	0.0998	0.1085	0.0792	0.0798	0.0826
I2 / 10 at 50% util.	30	1.706	1.214	1.430	1.369	1.427	0.1384	0.1752	0.0901	0.0918	0.1001
I2 / 10 at 70% util.	10	4.837	2.295	3.687	3.658	3.739	0.2779	0.3752	0.1182	0.1281	0.1388
I2 / 10, half RTTs at 70% util.	10	4.569	2.023	3.196	3.245	3.405	0.2555	0.3607	0.0995	0.1131	0.1165
I2 / 10, double RTTs at 70% util.	10	5.098	2.769	4.243	4.125	4.389	0.325	0.4172	0.1591	0.1640	0.1843
Rocketfuel at 70% util.	100	0.964	0.796	0.840	0.813	0.835	0.0922	0.0991	0.0793	0.0788	0.0836

Figure 7: The figures show the average FCT and RTT values for I2/10 at 70% utilization (LSTF uses fairness slack assignment with $r_{est} = 10 Mbps$). The error bars indicating the 10^{th} and the 99^{th} percentile values and the y-axis is in log-scale. The table indicates the average FCT and RTTs (across bytes) for varying settings.

and when more packets are dropped, the interval value is reduced using a control law, which divides the initial interval value by the square root of the number of packets dropped. The interval is refreshed (re-initialized to 100ms) when the queue becomes empty, or when a packet sees a sojourn time less than the target.[9] An extension to CoDel is FQ-CoDel [20], where the scheduler round-robins across flows and the CoDel control loop is applied to each flow individually. The interval for a flow is refreshed when there are no more packets belonging to that flow in the queue. FQ-CoDel is considered to be better than CoDel in all regards[10], even by one of the co-developers of CoDel [3].

Edge-CoDel: We aim to approximate FQ-CoDel from the edge by (i) using LSTF to implement per-flow fairness in routers and (ii) using the values carried by LSTF to guide the egress router as to whether or not to drop the packet. For (i), the initial slack values are assigned based on our fairness scheme described in §3.3. For (ii), an extra field is added to the packet header at the ingress which stores the assigned slack value (called the initial slack field), which remains untouched as the packet traverses the network. The other field where the ingress stores the assigned slack value is updated as per the LSTF algorithm; we call this the current slack field. During dequeue at the egress, the precise amount of queuing delay seen by the packet within the network can be computed by simply comparing the initial slack field and the current slack field. At the egress router for each flow we then run the FQ-CoDel logic for when to drop packets, keeping the control law and the parameters (the target value and the initial interval value) the same as in FQ-CoDel. We call this approach Edge-CoDel.

There are only two things that change in Edge-CoDel as compared to FQ-CoDel. First, instead of looking at the sojourn time of each queue individually, Edge-CoDel looks at the total queuing time of the packet across the entire network. The second change is with respect to how the CoDel interval is refreshed. As mentioned before, in traditional FQ-CoDel, there are two events that trigger a refresh in the interval (1) when a packet's sojourn time is less than the target and (2) when all the queued-up packets for a given flow have been transmitted. While Edge-CoDel can react to the former, it has no explicit way of knowing the latter. To address this, we refresh the interval based on the difference in the send time of two consecutive packets. When the difference in the time at which two consecutive packets (say p_1 and p_2) were sent by the source (found using TCP timestamps that are enabled by default) is more than a certain threshold, we conclude that the interval should be refreshed as it is likely that there is no build-up of packets from that flow in the network. Clearly, this *refresh threshold* must be greater than CoDel's target queuing delay value. We find that a refresh threshold of 2-4 times the target value (10-20ms) works reasonably well. We elaborate more on the effect of picking different refresh thresholds in our evaluation below.

Results: In our experiments, we compare four different

[9] CoDel is a little more complicated than this, and while our implementation follows the CoDel specification [28], our explanation has been simplified, highlighting only the relevant points for brevity.

[10] We verified the same using our simulations with varying settings. For all cases, FQ-CoDel resulted in smaller mean RTT across packets and also in smaller flow completion times (averaged across bytes) as compared to CoDel. For example, for our default I2 1Gbps-10Gbps topology at 70% utilization, FQ-CoDel resulted in 0.0642s of mean RTT and 0.642s of mean FCT while the corresponding values for CoDel were 0.0748s and 0.815 respectively.

schemes: (1) FIFO without AQM (to set a baseline), (2) FQ without AQM (to see the effects of FQ on its own), (3) FQ-CoDel (to provide the state-of-the-art comparison) (4) LSTF scheduling (with slacks assigned to meet the fairness objective using appropriate r_{est} values) in conjunction with Edge-CoDel. As we move from (3) to (4), we make two transitions – first is with respect to the scheduling done inside the network (perfect isolation with FQ vs approximate isolation with LSTF) and the second is the shift of AQM logic from inside the network to the edge. Therefore, as an incremental step in between the two transitions, we also provide results for FQ with Edge-CoDel, where routers do FQ across flows (with the slack values maintained only for book-keeping) and AQM is done by Edge-CoDel. This allows us see how well Edge-CoDel works with perfect per-router isolation. The refresh threshold we use for Edge-CoDel in both cases is 20ms (4 times the CoDel target value). The buffer size is increased to 50MB so that AQM kicks in before a natural packet drop occurs.

Figure 7 shows our results for varying settings and schemes. The main metrics we use for evaluation are the FCT and the per-packet RTT, since the goal of an AQM scheme is to maintain high throughput (or small FCTs) while keeping the RTTs small. The two graphs show the average FCT and the average RTT across flows bucketed by their size for the I2 / 10 topology at 70% utilization.[11] To compute the average RTT, we first compute the average RTT value per flow (by taking the mean across the measured RTTs of all data packets belonging to the flow) and then taking the average of this value across flows in the same bucket. As expected, we find that while FQ helps in reducing the FCT values as compared to FIFO, it results in significantly higher RTTs than FIFO for long flows. FQ-CoDel reduces the RTT seen by long flows compared to FQ (with the short flows having RTT smaller than FIFO and comparable to FQ). What is new is that, shifting the CoDel logic to the edge through Edge-CoDel while doing FQ in the router makes very little difference as compared to FQ-CoDel. As we experiment with varying settings, we find that in some cases, FQ with Edge-CoDel results in slightly smaller FCTs at the cost of slightly higher RTTs than FQ-CoDel. We believe that this is due to the difference in how the CoDel interval is refreshed with Edge-CoDel and with in-router FQ-CoDel. Replacing the scheduling algorithm with LSTF again produces minor differences in the results compared to FQ-CoDel. Both the FCT and the RTT are slightly higher than FQ-CoDel for almost all cases, and we attribute the differences to LSTF's *approximation* of round-robin service across flows. Nonetheless, the average FCTs obtained are significantly lower than FIFO and the average RTTs are significantly lower than both FIFO and FQ for all cases.

Refresh Threshold (ms)	Avg FCT across bytes (s)	Avg RTT across bytes (s)
10	3.578	0.143
20	3.739	0.139
30	3.954	0.135
40	4.079	0.132

Table 4: Effect of varying refresh threshold on I2/10 topology at 70% utilization running LSTF (r_{est} = 10Mbps) with Edge-CoDel.

To see whether our results were highly dependent on the refresh threshold value, consider Table 4 which shows the average FCT and RTT values for varying refresh thresholds. We find that there are very minor differences in the results as we vary this threshold, because the dominating cause for refreshing the interval is when a packet sees a queuing delay less than the CoDel target. However, the general trend is that increasing the refresh threshold increases the FCT and decreases the RTT. This is because with increasing refresh threshold, the interval is reset to the larger 100ms value less frequently. This results in more packet drops for the long flows, causing an increase in FCTs, but a decrease in the RTT values.

Summary: The *used slack* information available as a by-product from LSTF can be effectively used to emulate an AQM scheme from the edge of the network. While we evaluate this insight in the context of packet drops in CoDel, showing that LSTF with EdgeCoDel performs comparable to FQ-CoDel, our edge-based approach can also be adapted for other AQM schemes such as ECN.

5 LSTF Implementation

We now ask whether LSTF can be implemented in the routers? LSTF execution *at a particular router* is no more complex than the execution of fine-grained priorities. To see this, suppose a packet p arrives a router α at time $i(p,\alpha)$, with slack $slack(p,\alpha)$. As mentioned in §2, LSTF prioritizes packets based on their remaining slack value at the time when their last bit is transmitted. This term is given by $(slack(p,\alpha) - (t - i(p,\alpha)) + T(p,\alpha))$ at any time t while p is waiting at α. $T(p,\alpha)$ is the transmission time of p at α, which is added to account for the remaining slack of p, relative to other packets, when its *last bit* is transmitted. Since t is same for all packets at any given point of time when the packets are being compared at α, the deciding term is $(slack(p,\alpha) + i(p,\alpha) + T(p,\alpha))$. This term can be computed when the packet arrives at α and can be attached to the packet as its priority value (programmable header processing mechanisms [11, 12] can be used to easily execute this step).

Fine-grained prioritization can be carried out in almost constant time using specialized data-structures such as pipelined heap (p-heap) [10,22]. The p-heap datastructure implemented in hardware by Bhagwan et. al [10] using

[11] AQM produces a bigger impact on this topology due to smaller link capacity and more queuing than for our default topology. Hence we show the graphs for this case, with aggregate metrics presented for others in the table.

a 0.35 micron CMOS technology can support a line rate of 10Gbps with over 4 billion priority levels, which are more than enough to support LSTF with slack values assigned at a granularity of microseconds. While we use nanosecond granularity for the slack assignment in all our simulations with LSTF, we verified that using a coarser granularity of microseconds has negligible impact on the replay results (for our default scenario described in §2.3, nanosecond granularity resulted in a total of 0.214% overdue packets, while microsecond granularity resulted in 0.215% overdue packets).

Right before a packet p is transmitted by the router, its slack can be overwritten by the remaining slack value, computed by simply subtracting the stored priority value $(slack(p,\alpha)+i(p,\alpha)+T(p,\alpha))$ with the sum of the current time and the packet's transmission time $(T(p,\alpha))$.

Thus, while the hardware implementation of these algorithms would require some effort, it does not appear a significant challenge to implement LSTF at linespeed.

6 Related Work

The literature of packet scheduling is vast, and ever-increasing. Here we only touch on a few topics most relevant to our work.

The real-time scheduling literature has studied optimality of scheduling algorithms (in particular EDF and LSTF) for single and multiple processors [24, 25], where a scheduling algorithm is said to be optimal if it can (feasibly) schedule a set of tasks that can be scheduled by any other algorithm. Liu and Layland [25] proved the optimality of EDF for a single processor in hard real-time systems. LSTF was then shown to be optimal for single-processor scheduling as well, while being more effective than EDF (though not optimal) for multi-processor scheduling [24]. In the context of networking, [14] provides theoretical results on emulating the schedules produced by a single output-queued switch using a combined input-output queued switch with a smaller speed-up of at most two. To the best of our knowledge, the optimality or universality of a scheduling algorithm for a network of inter-connected resources (in our case, switches) has never been studied before.

LSTF and EDF have been used before in networking to achieve different goals. Deadline aware congestion control algorithms, inspired from the optimality of EDF, have been proposed for meeting flow deadlines in datacenters [7, 21, 37]. The FIFO+ algorithm [15] uses LSTF to reduce tail packet delays in multi-hop networks.

A recent paper [33] proposed programmable hardware in the dataplane for packet scheduling and queue management, in order to achieve various network objectives without the need for physically replacing the hardware. It uses simulation of three schemes (FQ, CoDel+FQ, CoDel+FIFO) competing on three different metrics to show that there is no "silver bullet" solution. As mentioned earlier, our work is inspired by the questions the authors raise; we adopt a broader view of scheduling in which packets can carry dynamic state leading to the results presented here.

In our early position paper, to appear in HotNets 2015, we describe our vision of a UPS and some early results on LSTF as a UPS. Here, we add detailed theoretical results, a more comprehensive evaluation along with additional performance objectives and address AQM.

7 Conclusion

This paper started with a theoretical perspective by analyzing whether there exists a single *universal* packet scheduling algorithm that can perfectly replay all viable schedules. We proved that while such an algorithm cannot exist, LSTF comes closest to being one (in terms of the number of congestion points it can deal with). We then empirically demonstrated the ability of LSTF to approximately replay a wide range of scheduling algorithms under varying network settings. Replaying a given schedule, while of theoretical interest, requires the knowledge of viable output times, which is not available in practice.

Hence, we next considered if LSTF can be used in practice to achieve various performance objectives. We showed via simulation how LSTF, combined with heuristics to set the slack values at the ingress, can do a reasonable job of minimizing mean flow completion time, minimizing tail latencies, and achieving per-flow fairness.

Noting that scheduling is often used along with AQM to prevent queue build up, we then showed how LSTF can be used to implement a version of AQM at the network edge, with performance roughly comparable to the state-of-the art (FQ-CoDel).

While an initial step towards understanding the notion of a Universal Packet Scheduler, our work leaves several theoretical questions unanswered, three of which we mention here. First, we showed existence of a UPS with omniscient header initialization, and nonexistence with limited-information initialization. *What is the least information we can use in header initialization in order to achieve universality?* Second, we showed that, in practice, the fraction of overdue packets is small, and most are only overdue by a small amount. *Are there tractable bounds on both the number of overdue packets and/or their degree of lateness?* Third, while we have a formal characterization for the scope of LSTF with respect to replaying a given schedule, and we have simulation evidence of LSTF's ability to meet several performance objectives, we do not yet have any formal model for the scope of LSTF in meeting these objectives. *Can one describe the class of performance objectives that LSTF can meet?*

References

[1] Global Consortium to Construct New Cable System Linking US and Japan to Meet Increasing Bandwidth Demands. http://googlepress.blogspot.com/2008/02/global-consortium-to-construct-new_26.html.

[2] Internet2. http://www.internet2.edu/.

[3] Kathie Nichol's CoDel presented by Van Jacobson. http://www.ietf.org/proceedings/84/slides/slides-84-tsvarea-4.pdf.

[4] Microsoft Invests in Subsea Cables to Connect Datacenters Globally. http://goo.gl/GoXfxH.

[5] The network simulator ns-2. http://www.isi.edu/nsnam/ns/.

[6] Ns-3. http://www.nsnam.org/.

[7] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pfabric: Minimal near-optimal datacenter transport. 2013.

[8] M. Allman. Comments on bufferbloat. *ACM SIGCOMM Computer Communication Review*, 2013.

[9] T. Benson, A. Akella, and D. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. ACM Internet Measurement Conference (IMC)*, 2012.

[10] R. Bhagwan and B. Lin. Fast and scalable priority queue architecture for high-speed network switches. In *Proc. IEEE Infocom 2000*, 2000.

[11] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 2014.

[12] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proc. ACM SIGCOMM*, 2013.

[13] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: A retrospective on evolving sdn. In *Proc. ACM HotSDN*, 2012.

[14] S.-T. Chuang, A. Goel, N. McKeown, and B. Prabhakar. Matching output queueing with a combined input/output-queued switch. *Selected Areas in Communications, IEEE Journal on*, 1999.

[15] D. D. Clark, S. Shenker, and L. Zhang. Supporting real-time applications in an integrated services packet network: Architecture and mechanism. *SIGCOMM Comput. Commun. Rev.*, 1992.

[16] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *SIGCOMM Comput. Commun. Rev.*, 1989.

[17] N. Dukkipati and N. McKeown. Why Flow-Completion Time is the Right Metric for Congestion Control. *ACM SIGCOMM Computer Communication Review*, 2006.

[18] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, 1993.

[19] A. Gupta, M. T. Hajiaghayi, and H. Räcke. Oblivious network design. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, SODA '06. Society for Industrial and Applied Mathematics, 2006.

[20] T. Hoeiland-Joergensen, P. McKenney, D. Taht, J. Gettys, and E. Dumazet. Flowqueue-codel. *IETF Informational*, 2013.

[21] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing flows quickly with preemptive scheduling. In *Proc. ACM SIGCOMM*, 2012.

[22] A. Ioannou and M. G. H. Katevenis. Pipelined heap (priority queue) management for advanced scheduling in high-speed networks. *IEEE/ACM Trans. Netw.*, 2007.

[23] R. Jain, D.-M. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *CoRR*, 1998.

[24] J. Y.-T. Leung. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica*, 1989.

[25] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 1973.

[26] R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Recursively cautious congestion control. In *Proc. USENIX NSDI*, 2014.

[27] K. Nichols and V. Jacobson. Controlling queue delay. *Queue*, 2012.

[28] K. Nichols and V. Jacobson. Controlled delay active queue management: draft-nichols-tsvwg-codel-02. *Internet Requests for Comments-Work in Progress, http://tools. ietf. org/id/draft-nichols-tsvwg-codel-01. txt, Tech. Rep*, 2014.

[29] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Trans. Netw.*, 1993.

[30] B. Raghavan, M. Casado, T. Koponen, S. Ratnasamy, A. Ghodsi, and S. Shenker. Software-defined internet architecture: Decoupling architecture from infrastructure. In *Proc. ACM HotNets*, 2012.

[31] S. Blake and D. Black and M. Carlson and E. Davies and Z. Wang and W. Weiss. An Architecture for Differentiated Services. RFC 2475.

[32] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. *ACM SIGCOMM Comput. Commun. Rev.*, 1995.

[33] A. Sivaraman, K. Winstein, S. Subramanian, and H. Balakrishnan. No silver bullet: Extending sdn to the data plane. In *Proc. ACM HotNets*, 2013.

[34] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proc. ACM SIGCOMM*, 2002.

[35] I. Stoica, S. Shenker, and H. Zhang. Core-stateless fair queueing: A scalable architecture to approximate fair bandwidth allocations in high-speed networks. *IEEE/ACM Trans. Netw.*, 2003.

[36] I. Stoica and H. Zhang. Providing guaranteed services without per flow management. In *Proc. ACM SIGCOMM*, 1999.

[37] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: Meeting deadlines in datacenter networks. In *Proc. ACM SIGCOMM*, 2011.

[38] L. Zhang. Virtual clock: A new traffic control algorithm for packet switching networks. *SIGCOMM Comput. Commun. Rev.*, 1990.

Appendix

A Proofs: Analytical Replayability Results

This section contains theoretical proofs for the analytical replayability results presented in §2. We begin with defining some notations used throughout in the proofs.

A.1 Notations

We use the following notations for our proofs, some of which have been already defined in the main text:

Relevant nodes:

- $src(p)$: Ingress of a packet p.
- $dest(p)$: Egress of a packet p.

Relevant time notations:

- $T(p,\alpha)$: Transmission time of a packet p at node α.
- $o(p,\alpha)$: Time when the first bit of p is scheduled by node α in the original schedule.
- $o(p) = o(p,dest(p)) + T(p,dest(p))$: Time when the last bit of p exits the network in the original schedule (which is non-preemptive).
- $o'(p)$: Time when the last bit of p exits the network in the replay (which may be preemptive in our theoretical arguments).
- $i(p,\alpha)$ and $i'(p,\alpha)$: Time when p arrives at node α in the original schedule and in the replay respectively.
- $i(p) = i(p,src(p)) = i'(p)$: Arrival time of p at its ingress. This remains the same for both the original schedule and the replay.
- $t_{min}(p,\alpha,\beta)$: Minimum time p takes to start from node α and exit from node β in an uncongested network. It therefore includes the propagation delays and the store-and-forward delays of all links in the path from α to β and the transmission delays at α and β. Handling the edge case: $t_{min}(p,\alpha,\alpha) = T(p,\alpha)$
- $slack(p) = o(p) - i(p) - t_{min}(p,src(p),dest(p))$: Total slack of p that gets assigned at its ingress. It denotes the amount of time p can wait in the network without any of its bits getting serviced.
- $slack(p,\alpha,t) = o(p) - t - t_{min}(p,\alpha,dest(p)) + T(p,\alpha)$: Remaining slack of the last bit of p at time t when it is at node α. We derive this expression in Appendix A.4.

Other miscellaneous notations

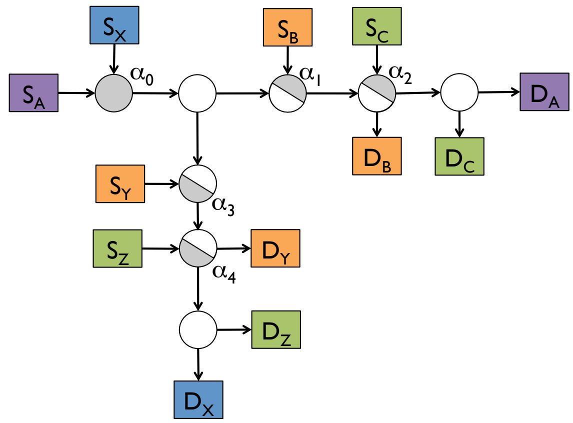
- $path(p,\alpha,\beta)$: The ordered set of nodes and links in the path taken by p to go from α to β. The set also includes α and β as the first and the last nodes.
- $path(p) = path(p,src(p),dest(p))$
- $pass(\alpha)$: Set of packets that pass through node α.

A.2 Existence of a UPS under Omniscient Header Initialization

Algorithm: At the ingress, insert an n-dimensional vector in the packet header, where the i^{th} element contains $o(p,\alpha_i)$, α_i being the i^{th} hop in $path(p)$. Every time a packet p arrives at the router, the router pops the value at the head of the vector in p's header and uses that as the priority for p (earlier values of output times get higher priority). This can perfectly replay any schedule.

Proof: We can prove that the above algorithm will result in no overdue packets (which do not meet their original schedule's target) using the following two theorems:
Theorem 1: If for any node α, $\exists p' \in pass(\alpha)$, such that using the above algorithm, the last bit of p' exits α at time $(t' > (o(p',\alpha) + T(p',\alpha)))$, then $(\exists p \in pass(\alpha) \mid i'(p,\alpha) \le t'$ and $i'(p,\alpha) > o(p,\alpha))$.
Proof by contradiction: Consider the first such $p^* \in pass(\alpha)$ that gets late at α (i.e. its last bit exits α at time $t^* > (o(p^*,\alpha) + T(p^*,\alpha))$). Suppose the above condition is not true i.e. $(\forall p \in pass(\alpha) \mid i'(p,\alpha) \le o(p,\alpha)$ or $i'(p,\alpha) > t^*)$. In other words, if p arrives at or before time t^*, it also arrives at or before time $o(p,\alpha)$. Given that all bits of p^* arrive at or before time t^*, they also arrive at or before time $o(p^*,\alpha)$. The only reason why the last bit of p^* would wait until time $(t^* > o(p^*,\alpha) + T(p^*,\alpha))$ in our work-conserving replay is if some other bits (belonging to higher priority packets) were being scheduled after time $o(p^*,\alpha)$, resulting in p^* not being able to complete its transmission by time $(o(p^*,\alpha) + T(p^*,\alpha))$. However, as per our algorithm, any packet p_{high} having a higher priority than p^* at α must have been scheduled before p^* in the original schedule, implying that $(o(p_{high},\alpha) + T(p_{high},\alpha)) \le o(p^*,\alpha)$. [12] Therefore, some bits of p_{high} being scheduled after time $o(p^*,\alpha)$, implies them being scheduled after time $(o(p_{high},\alpha) + T(p_{high},\alpha))$. This means that p_{high} is already late and contradicts our assumption that p^* is the first packet to get late. Hence proved that if for any node α, $\exists p' \in pass(\alpha)$, such that using the above algorithm, the last bit of p' exits α at time $(t' > (o(p',\alpha) + T(p',\alpha)))$, then $(\exists p \in pass(\alpha) \mid i'(p,\alpha) \le t'$ and $i'(p,\alpha) > o(p,\alpha))$
Theorem 2: $\forall \alpha, (\forall p \in pass(\alpha) \mid i'(p,\alpha) \le i(p,\alpha))$.
Proof by contradiction: Consider the first time when some packet p^* arrives late at some node α^* (i.e. $i'(p^*,\alpha^*) > i(p^*,\alpha^*)$). In other words, α^* is the first node in the network to see a late packet arrival, and p^* is the first late arriving packet. Let α_{prev} be the node visited by p^* just before arriving at α^*. p^* can arrive at a time later than $i(p^*,\alpha^*)$ at α^* only if the last bit of p^* exits α_{prev} at time $t_{prev} > o(p^*,\alpha_{prev}) + T(p^*,\alpha_{prev})$. As per Theorem 1 above, this is possible only if some packet p' (which may or may not be same as p^*) arrives at α_{prev} at time $i'(p',\alpha_{prev}) > o(p',\alpha_{prev}) \ge i(p',\alpha_{prev})$ and $i'(p',\alpha_{prev}) \le t_{prev} < i'(p^*,\alpha^*)$. This contradicts our assumption that α^* is the first node to see a late arriving packet. Therefore, $\forall \alpha, (\forall p \in pass(\alpha) \mid i'(p,\alpha) \le i(p,\alpha))$.

Combining the two theorems above: Since $\forall \alpha (\forall p \in pass(\alpha) \mid i'(p,\alpha) \le i(p,\alpha))$, with the above algorithm, $\forall \alpha (\forall p \in pass(\alpha))$, all bits of p exit α before $(o(p,\alpha) + T(p,\alpha))$. Therefore, the algorithm can perfectly replay any viable schedule.



Node	Packet(arrival time, scheduling time)
	Case 1
α_0	$a(\mathbf{0},0)$; $x(\mathbf{0},1)$
α_1	$a(1,1)$, $b_1(2,2)$, $b_2(3,3)$, $b_3(4,4)$
α_2	$c_1(2,2)$, $c_2(3,3)$; $a(2,\mathbf{4})$
α_3	$x(2,2)$, $y_1(2,3)$, $y_2(3,4)$
α_4	$z(2,2)$, $x(3,\mathbf{3})$
	Case 2
α_0	$x(\mathbf{0},0)$; $a(\mathbf{0},1)$
α_1	$a(2,2)$, $b_1(2,3)$, $b_2(3,4)$, $b_3(4,5)$
α_2	$c_1(2,2)$, $c_2(3,3)$, $a(3,\mathbf{4})$
α_3	$x(1,1)$, $y_1(2,2)$, $y_2(3,3)$
α_4	$z(2,2)$, $x(2,\mathbf{3})$

Figure 8: Example showing non-existence of a UPS with Blackbox Initialization. A packet represented by p belongs to flow P, with ingress S_P and egress D_P, where $P \in \{A,B,C,X,Y,Z\}$. For simplicity assume all packets are of the same size and all links have a propagation delay of zero. All uncongested routers (white), ingresses and egresses have a transmission time of zero. The congestion points (shaded grey) have transmission times of $T = 1$ unit.

A.3 Nonexistence of a UPS under blackbox initialization

Proof by counter-example: Consider the example shown in Figure 8. For simplicity, assume all the propagation delays are zero, the transmission time for each congestion point (shaded in grey) is 1 unit and the uncongested (white) routers have zero transmission time. [13] All packets are of the same size.

The table illustrates two cases. For each case, a packet's arrival and scheduling time (the time when the packet is scheduled by the router) at each node through which it passes are listed. A packet represented by p belongs

[12] Given that the original schedule is non-preemptible, the next packet gets scheduled only after the previous one has completed its transmission.

[13] These assignments are made for simplicity of understanding. The example will hold for any reasonable value of propagation and transmission delays.

to flow P, with ingress S_P and egress D_P, where $P \in \{A,B,C,X,Y,Z\}$. The packets have the same *path* in both cases. For example, a belongs to Flow A, starts at ingress S_A, exits at egress D_A and passes through three congestion points in its path α_0, α_1 and α_2; x belongs to Flow X, starts at ingress S_X, exits at egress D_X and passes through three congestion points in its path α_0, α_3 and α_4; and so on.

The two critical packets we care about in this example are a and x, which interact with each-other at their first congestion point α_0, being scheduled by α_0 at different times in the two cases (a before x in Case 1 and x before a in Case 2). But, notice that for both cases,

- a enters the network from its ingress S_A at congestion point α_0 at time 0, and passes through two other congestion points α_1 and α_2 before exiting the network at time $(4+1)$ [14].
- x enters the network from its ingress S_X at congestion point α_0 at time 0, and passes through two other congestion points α_3 and α_4 before exiting the network at time $(3+1)$.

a interacts with packets from Flow C at its third congestion point α_2, while x interacts with a packet from Flow Z at its third congestion point α_4. For both cases,

- Two packets of Flow C (c_1, c_2) enter the network at times 2 and 3 at α_2 before they exit the network at time $(2+1)$ and $(3+1)$ respectively.
- z enters the network at time 2 at α_4 before exiting at time $2+1$.

The difference between the two cases comes from how a interacts with packets from Flow B at its second congestion point α_1 and how x interacts with packets from Flow Y at its second congestion points α_3. Note that α_1 and α_3 are the last congestion points for Flow B and Flow Y packets respectively and their exit times from these congestion points directly determine their exit times from the network.

- Three packets of Flow B (b_1, b_2, b_3) enter the network at times 2, 3 and 4 respectively at α_1. In Case 1, they leave α_1 at times $(2+1),(3+1),(4+1)$ respectively, providing no lee-way [15] for a at α_0, which leaves α_1 at time $(1+1)$. In Case 2, (b_1, b_2, b_3) leave at times $(3+1),(4+1),(5+1)$ respectively, providing lee-way for a at α_0, which leaves α_1 at time $(2+1)$.
- Two packets of Flow Y (y_1, y_2) enter the network at times 2 and 3 respectively at α_3. In Case 1, they leave at times $(3+1),(4+1)$ respectively, providing a lee-way for x at α_0, which leaves α_3 at time $(2+1)$. In Case 2, (y_1, y_2) exit at times $(2+1),(3+1)$, providing no lee-way for x at α_0, which leaves α_3 at time $(1+1)$.

Note that the interaction of a and x with Flow C and Flow Z at their third congestion points respectively, is what ensures that their eventual exit time remains the same across the two cases inspite of the differences in how a and x are scheduled in their previous two hops.

Thus, we can see that $i(a)$, $o(a)$, $i(x)$, $o(x)$ are the same in both cases (also indicated in bold blue). Yet, *Case 1* requires a to be scheduled before x at α_0, else packets will get delayed at α_1, since it is required that α_1 schedules a at a time no more than 3 units if it is to meet its target output time. *Case 2* requires x to be scheduled before a at α_0, else packets will be delayed at α_3, where it is required to schedule x at a time no more than 2 units if it is to meet its target output time. Since the attributes $(i(\cdot), o(\cdot), path(\cdot))$ for both a and x are exactly the same in both cases, any deterministic UPS with Blackbox Initialization will produce the same order for the two packets at α_0, which contradicts the situation where we want a before x in one case and x before a in another.

A.4 Deriving the Slack Equation

We now prove that for any packet p waiting at any node α at time t_{now}, the remaining slack of the last bit of p is given by $slack(p,\alpha,t_{now}) = o(p) - t_{now} - t_{min}(p,\alpha,dest(p)) + T(p,\alpha)$.

Let $t_{wait}(p,\alpha,t_{now})$ denote the total time spent by p on waiting behind other packets at the nodes in its path from $src(p)$ to α (including these two nodes) until time t_{now}. We define $t_{wait}(p,\alpha,t_{now})$, such that it excludes the transmission times at previous nodes which gets captured in t_{min}, but includes the local service time received by the packet so far at α itself.

$$\begin{aligned}
slack(p,\alpha,t_{now}) =& slack(p) - t_{wait}(p,\alpha,t_{now}) + T(p,\alpha) && \text{(1a)}\\
=& o(p) - i(p) - t_{min}(p,src(p),dest(p))\\
& - t_{wait}(p,\alpha,t_{now}) + T(p,\alpha) && \text{(1b)}\\
=& o(p) - i(p) - (t_{min}(p,src(p),\alpha)\\
& + t_{min}(p,\alpha,dest(p)) - T(p,\alpha))\\
& - t_{wait}(p,\alpha,t_{now}) + T(p,\alpha) && \text{(1c)}\\
=& o(p) - t_{min}(p,\alpha,dest(p)) + T(p,\alpha)\\
& - (i(p) + t_{min}(p,src(p),\alpha)\\
& - T(p,\alpha) + t_{wait}(p,\alpha,t_{now})) && \text{(1d)}\\
=& o(p) - t_{min}(p,\alpha,dest(p)) + T(p,\alpha) - t_{now} && \text{(1e)}
\end{aligned}$$

(1a) is straightforward from our definition of LSTF and how the slack gets updated at every time slice. $T(p,\alpha)$ is added since α needs to locally consider

[14] +1 is added to indicate transmission time at the last congestion point. As mentioned before, we assume the propagation delay to the egress and the transmission time at the egress are both 0.

[15] It is required that α_1 must schedule a by at most time 3 in order for it to exit the network at its target output time.

the slack of the last bit of the packet in a store-and-forward network. (1c) then uses the fact that for any α in $path(p)$, $(t_{min}(p,src(p),dest(p)) = t_{min}(p,src(p),\alpha) + t_{min}(p,\alpha,dest(p)) - T(p,\alpha))$. $T(p,\alpha)$ is subtracted here as it is accounted for twice when we break up the equation for $t_{min}(p,src(p),dest(p))$. (1e) then follows from the fact that the difference between t_{now} and $i(p)$ is equal to the total amount of time the packet has spent in the network until time t_{now} i.e. $(t_{now} - i(p) = (t_{min}(p,src(p),\alpha) - T(p,\alpha)) + t_{wait}(p,\alpha,t_{now}))$. We need to subtract $T(p,\alpha)$, since by our definition, $t_{min}(p,src(p),\alpha)$ includes transmission time of the packet at α.

A.5 LSTF and EDF Equivalence

In our network-wide extension of EDF scheduling, every router computes a *local* deadline of a packet p based on the static header value $o(p)$ and additional state information about the minimum time the packet would take to reach its destination from the router. More precisely, each router (say α), uses $priority(p) = (o(p) - t_{min}(p,\alpha,dest(p)) + T(p,\alpha))$ to do priority scheduling, with $o(p)$ being the value carried by the packet header, initialized at the ingress and remaining unchanged throughout. EDF is equivalent to LSTF, in that for a given viable schedule, the two produce exactly the same replay schedule.

Proof: Consider any node α with non-empty queue at any given time t_{now}. Let $P(\alpha,t_{now})$ be the set of packets waiting at α at time t_{now}. A packet will then be scheduled by α as follows:

With EDF: Schedule packet $p_{edf}(\alpha,t_{now})$, where

$$p_{edf}(\alpha,t_{now}) = \underset{p\in P(\alpha,t_{now})}{\operatorname{argmin}} \; (priority(p,\alpha))$$
$$priority(p,\alpha) = o(p) - t_{min}(p,\alpha,dest(p)) + T(p,\alpha)$$

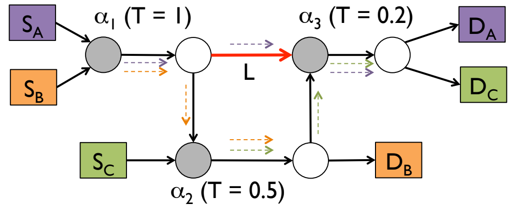
With LSTF: Schedule packet $p_{lstf}(\alpha,t_{now})$, where

$$p_{lstf}(\alpha,t_{now}) = \underset{p\in P(\alpha,t_{now})}{\operatorname{argmin}} \; (slack(p,\alpha,t_{now}))$$
$$slack(p,\alpha,t_{now}) = o(p) - t_{min}(p,\alpha,dest(p)) + T(p,\alpha) - t_{now}$$

The above expression for $slack(p,\alpha,t_{now})$ has been derived in §A.4. Thus, $slack(p,\alpha,t_{now}) = priority(p,\alpha) - t_{now}$. Since t_{now} is the same for all packets, we can conclude that:

$$\underset{p\in P(\alpha,t_{now})}{\operatorname{argmin}} \; (slack(p,\alpha,t_{now})) = \underset{p\in P(\alpha,t_{now})}{\operatorname{argmin}} \; (priority(p,\alpha))$$
$$\implies p_{lstf}(\alpha,t_{now}) = p_{edf}(\alpha,t_{now})$$

Therefore, at any given point of time, all nodes with non-empty queues will schedule the same packet with both EDF and LSTF. [16] Hence, EDF and LSTF are equivalent.



Node	Packet(arrival time, scheduling time)
α_1	$a(0,0), b(0,1)$
α_2	$b(2,2), c(2,2.5)$
α_3	$c(3,3), a(3,3.2)$

Figure 9: Example showing replay failure with simple priorities for a schedule with two congestion points per packet. A packet represented by p belongs to flow P, with ingress S_P and egress D_P, where $P \in \{A,B,C\}$. All packets are of the same size. For simplicity assume all links (except L) have a propagation delay of zero. L has a propagation delay of 2. All uncongested routers (white circles), ingresses and egresses have a transmission time of zero. The three congestion points – $\alpha_1, \alpha_2, \alpha_3$ have transmission times of $T = 1$ unit, $T = 0.5$ units and $T = 0.2$ units respectively.

A.6 Simple Priorities Replay Failure for Two Congestion Points Per Packet

In Figure 9, we present an example which shows that simple priorities can fail in replay when there are two congestion points per packet, no matter what information is used to assign priorities. At α_1, we need to have $priority(a) < priority(b)$, at α_2 we need to have $priority(b) < priority(c)$ and at α_3 we need to have $priority(c) < priority(a)$. This creates a priority cycle where we need $priority(a) < priority(b) < priority(c) < priority(a)$, which can never be possible to achieve with simple priorities.

We would also like to point out here that priority assignment for perfect replay in networks with smaller complexity (with single congestion point per packet) requires detailed knowledge about the topology and input load. More precisely, if a packet p passes through congestion point α_p, then its priority needs to be assigned as $priority(p) = o(p) - t_{min}(p,\alpha_p,dest(p)) + T(p,\alpha_p)$.[17] Hence, we need to know where the congestion point occurs in a packet's path, along with the final output times,

[16] Assuming ties are broken in the same way for both per-router EDF and LSTF, such as by using FCFS.

[17] The proof that this would work for at most one congestion point per packet follows from the fact that the only scheduling decision made in a packet p's path is at the single congestion point α_p. This decision is same as what will be made with per-router EDF (just for at most one congestion point per packet), which we proved is equivalent to LSTF in §A.5, which in turn always gives a perfect replay for one (or to be more precise, at most two) congestion points per packet (as we shall prove in §A.7).

to assign the priorities. In the absence of this knowledge, priorities cannot replay even a single congestion point.

A.7 LSTF: Perfect Replay for at most Two Congestion Points per Packet

A.7.1 Main Proof

We first prove that LSTF can replay any schedule with at most two congestion points per packet. Note that we work with bits in our proof, since we assume a pre-emptive version of LSTF. Due to store-and-forward routers, the remaining slack of a packet at a particular router is represented by the slack of the last bit of the packet (with all other bits of the packet having the same slack as the last bit).

In order for a replay failure to occur, there must be at least one overdue packet, where a packet p is said to be overdue if $o'(p) > o(p)$. This implies that p must have spent all of its slack while waiting behind other packets at a queue in some node α at say time t, such that $slack(p,\alpha,t) < 0$. Obviously, α must be a congestion point.

Necessary Condition for Replay Failure with LSTF: If a packet p^* sees negative slack at a congestion point α when its last bit exits α at time t^* in the replay (i.e. $slack(p^*,\alpha,t^*) < 0$), then $(\exists p \in pass(\alpha) \mid i'(p,\alpha) \le t^*$ and $i'(p,\alpha) > o(p,\alpha))$. We prove this in §A.7.2.

Key Observation: *When there are at most two congestion points per packet, then no packet p can arrive at any congestion point α in the replay, after its corresponding scheduling time at α in the original schedule (.i.e. $i'(p,\alpha) > o(p,\alpha)$ is not possible). Therefore, by the necessary condition above, no packet can see a negative slack at any congestion point.*

Proof by contradiction: Suppose that there exists α^*, which is the first congestion point (in time) that sees a packet which arrives after its corresponding scheduling time in the original schedule. Let p^* be this first packet that arrives after the corresponding scheduling time in the original schedule at α^* ($i'(p^*,\alpha^*) > o(p^*,\alpha^*)$). Since there are at most two congestion points per packet, either α^* is the first congestion point seen by p^* or the last (or both).

(1) If α^* is the first congestion point seen by p^*, then clearly, $i'(p^*,\alpha^*) = i(p^*,\alpha^*) \le o(p^*,\alpha^*)$. This contradicts our assumption that $i'(p^*,\alpha^*) > o(p^*,\alpha^*)$.

(2) If α^* is not the first congestion point seen by p^*, then it is the last congestion point seen by p^*. If $i'(p^*,\alpha^*) > o(p^*,\alpha^*)$, then it would imply that p^* saw a negative slack before arriving at α^*. Suppose p^* saw a negative slack at a congestion point α_{prev}, before arriving at α^* when its last bit exited α_{prev} at time t_{prev}. Clearly, $t_{prev} < i'(p^*,\alpha^*)$. As per our necessary condition, this would imply that there must be another packet p', such that $i'(p',\alpha_{prev}) > o(p',\alpha_{prev})$ and $i'(p',\alpha_{prev}) \le t_{prev} < i'(p^*,\alpha^*)$. This contradicts our assumption that α^* is the first congestion point (in time) that sees a packet which arrives after its corresponding scheduling time in the original schedule.

Hence, no congestion point can see a packet that arrives after its corresponding scheduling time in the original schedule (and therefore no packet can get overdue) when there are at most two congestion points per packet.

We finally present, in §A.7.3, an example where LSTF replay failure occurs with no more than three congestion points per packet, thus completing our proof that LSTF can replay any schedule with at most two congestion points per flow and can fail beyond that.

A.7.2 Proof for Necessary Condition for Replay Failure with LSTF

We start with the following observation that we use in our proof.

Observation 1: If all bits of a packet p exit a router α by time $o(p,\alpha) + T(p,\alpha)$, then p cannot see a negative slack at α.

Proof for Observation 1: As shown previously in §A.4,

$$slack(p,\alpha,t) = o(p) - t_{min}(p,\alpha,dest(p)) + T(p,\alpha) - t$$

Therefore,

$$\begin{aligned}
& slack(p,\alpha,o(p,\alpha)+T(p,\alpha)) \\
& = o(p) - t_{min}(p,\alpha,dest(p)) + T(p,\alpha) - (o(p,\alpha)+T(p,\alpha)) \\
& \text{But, } o(p) = o(p,\alpha) + t_{min}(p,\alpha,dest(p)) + wait(p,\alpha,dest(p)) \\
& \implies slack(p,\alpha,o(p,\alpha)+T(p,\alpha)) = wait(p,\alpha,dest(p)) \\
& \implies slack(p,\alpha,o(p,\alpha)+T(p,\alpha)) \ge 0
\end{aligned}$$

where $wait(p,\alpha,dest(p))$ is the time spent by p in waiting behind other packets in the original schedule, after it left α, which is clearly non-negative.

We now move to the main proof for the necessary condition.

Necessary Condition for Replay Failure: If a packet p^* sees negative slack at a congestion point α when its last bit exits α at time t^* in the replay (i.e. $slack(p^*,\alpha,t^*) < 0$), then $(\exists p \in pass(\alpha) \mid i'(p,\alpha) \le t^*$ and $i'(p,\alpha) > o(p,\alpha))$.

Proof by Contradiction: Suppose this is not the case .i.e. there exists p^* whose last bit exits α at time t^*, such that $slack(p^*,\alpha,t^*) < 0$ and $(\forall p \in pass(\alpha) \mid i'(p,\alpha) > t^*$ or $i'(p,\alpha) \le o(p,\alpha))$. In other words, if $i'(p,\alpha) \le t^*$, then $i'(p,\alpha) \le o(p,\alpha)$. We can show that if this holds, then p^* cannot see a negative slack at α, thus violating our assumption.

We take the set of all bits which exit α at or before time t^* in the LSTF replay schedule. We denote this set as $S_{bits}(\alpha,t^*)$. Since all of these bits (and the corresponding packets) must arrive at or before time t^*, as per our assumption, $(\forall b \in S_{bits}(\alpha,t^*) \mid i'(p_b,\alpha) \leq o(p_b,\alpha))$, where p_b is denoted as the packet to which bit b belongs. Note that $S_{bits}(\alpha,t^*)$ also includes all bits of p^* as per our definition of $S_{bits}(\alpha,t^*)$.

We now prove that no bit in $S_{bits}(\alpha,t^*)$ can see a negative slack (and therefore p^* cannot see a negative slack at α), leading to a contradiction. The proof comprises of two steps:

Step 1: Using the same input arrival times of each packet at α as in the replay schedule, we first construct a *feasible schedule* at α up until time t^*, denoted by $FS(\alpha,t^*)$, where by feasibility we mean that no bit in $S_{bits}(\alpha,t^*)$ sees a negative slack.

Step 2: We then do an iterative transformation of $FS(\alpha,t^*)$ such that the bits in $S_{bits}(\alpha,t^*)$ are scheduled in the order of their *least remaining slack times*. This reproduces the LSTF replay schedule from which $FS(\alpha,t^*)$ was constructed in the first place. However, while doing the transformation we show how the schedule remains feasible at every iteration, proving that the LSTF schedule finally obtained is also feasible up until time t^*. In other words, no packet sees a negative slack at α in the resulting LSTF replay schedule up until time t^*, contradicting our assumption that p^* sees a negative slack when it exits α at time t^* in the replay.

We now discuss these two steps in details.

Step 1: Construct a feasible schedule at α up until time t^* (denoted as $FS(\alpha,t^*)$) for which no bit in $S_{bits}(\alpha,t^*)$ sees a negative slack.

(i) Algorithm for constructing $FS(\alpha,t^*)$: Use priorities to schedule each bit in $S_{bits}(\alpha,t^*)$, where $\forall b \in S_{bits}(\alpha,t^*) \mid$ $priority(b) = o(p_b,\alpha)$. (Note that since both $FS(\alpha,t^*)$ and LSTF are work-conserving, $FS(\alpha,t^*)$ is just a shuffle of the LSTF schedule up until t^*. The set of time slices at which a bit is scheduled in $FS(\alpha,t^*)$ and in the LSTF schedule up until t^* remains the same, but *which* bit gets scheduled at a given time slice is different.)

(ii) In $FS(\alpha,t^*)$, all bits b in $S_{bits}(\alpha,t^*)$ exit α by time $o(p_b,\alpha)+T(p_b,\alpha)$.

Proof by contradiction: Suppose the statement is not true and consider the first bit b^* that exits after time $(o(p_{b^*},\alpha)+T(p_{b^*},\alpha))$. We term this as b^* got late at α due to $FS(\alpha,t^*)$. Remember that, as per our assumption, $(\forall b \in S_{bits}(\alpha,t^*) \mid i'(p_b,\alpha) \leq o(p_b,\alpha))$. Thus, given that all bits of p_{b^*} arrive at or before time $o(p_{b^*},\alpha)$, the only reason why the delay can happen in our work-conserving $FS(\alpha,t^*)$ is if some other higher priority bits were being scheduled after time $o(p_{b^*},\alpha)$, resulting in p_{b^*} not being able to complete its transmission by time $(o(p_{b^*},\alpha)+T(p_{b^*},\alpha))$. However, as per our priority assignment algorithm, any bit b' having a higher priority than b^* at α must have been scheduled before the first bit of p_{b^*} in the non-preemptible original schedule, implying that $(o(p_{b'},\alpha)+T(p_{b'},\alpha)) \leq o(p_{b^*},\alpha)$. Therefore, a bit b' being scheduled after time $o(p_{b^*},\alpha)$, implies it being scheduled after time $(o(p_{b'},\alpha)+T(p_{b'},\alpha))$. This contradicts our assumption that b^* is the first bit to get late at α due to $FS(\alpha,t^*)$. Therefore, all bits b in $S_{bits}(\alpha,t^*)$ exit α by time $o(p_b,\alpha)+T(p_b,\alpha)$ as per the schedule $FS(\alpha,t^*)$,.

(iii) Since all bits in $S_{bits}(\alpha,t^*)$ exit by time $o(p_b,\alpha)+T(p_b,\alpha)$ due to $FS(\alpha,t^*)$, no bit in $S_{bits}(\alpha,t^*)$ sees a negative slack at α (from Observation 1).

Step 2: *Transform $FS(\alpha,t^*)$ into a feasible LSTF schedule for the single switch α up until time t^*.*

(Note: The following proof is inspired from the standard LSTF optimality proof that shows that for a single switch, any feasible schedule can be transformed to an LSTF schedule).

Let $fs(b,\alpha,t^*)$ be the scheduling time slice for bit b in $FS(\alpha,t^*)$. The transformation to LSTF is carried out by the following pseudocode:

1: **while** true **do**
2:     Find two bits, b_1 and b_2, such that:
        $(fs(b_1,\alpha,t^*) < fs(b_2,\alpha,t^*))$ **and**
        $(slack(b_2,\alpha,fs(b_1,\alpha,t^*))$
        $< slack(b_1,\alpha,fs(b_1,\alpha,t^*)))$ **and**
        $(i'(b_2,\alpha,t^*) \leq fs(b_1,\alpha,t^*))$
3:     **if** no such b_1 and b_2 exist **then**
4:         $FS(\alpha,t^*)$ is an LSTF schedule
5:         **break**
6:     **else**
7:         $swap(fs(b_1,\alpha,t^*), fs(b_2,\alpha,t^*))$ ▷ swap the scheduling times of the two bits. [18]
8:     **end if**
9: **end while**
10: Shuffle the scheduling time of the bits belonging to the same packet, to ensure that they are in order.
11: Shuffle the scheduling time of the same-slack bits such that they are in FIFO order

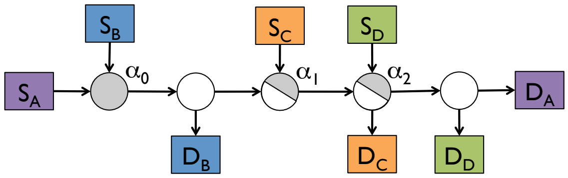
Line 7 above will not cause b_1 to have a negative slack, when it gets scheduled at $fs(b_2,\alpha,t^*)$ instead of $fs(b_1,\alpha,t^*)$. This is because the difference in $slack(b_2,\alpha,t)$ and $slack(b_1,\alpha,t)$ is independent of t and so:

$$slack(b_2,\alpha,fs(b_1,\alpha,t^*)) < slack(b_1,\alpha,fs(b_1,\alpha,t^*))$$
$$\Longrightarrow slack(b_2,\alpha,fs(b_2,\alpha,t^*)) < slack(b_1,\alpha,fs(b_2,\alpha,t^*))$$

Since $FS(\alpha,t^*)$ is feasible before the swap, $slack(b_2,\alpha,fs(b_2,\alpha,t^*)) \geq 0$. Therefore, $slack(b_1,\alpha,fs(b_2,\alpha,t^*)) > 0$ and the resulting $FS(\alpha,t^*)$ after the swap remains feasible.

Lines 10 and 11 will also not result in any bit getting a negative slack, because all bits participating in the shuffle

[18] Note that we are working with bits here for easy expressibility. In practice, such a swap is possible under the preemptive LSTF model.



Node	Packet(arrival time, scheduling time)
Original Schedule	
α_0	$a(0,0), b(0,1)$
α_1	$a(1,1), c_1(2,2), c_2(3,3)$
α_2	$d_1(2,2), d_2(3,3), a(2,4)$
LSTF Replay	
Node	*Packet(arrival time, scheduling time)*
α_0	$b(0,0), a(0,1)$
α_1	$c_1(2,2), a(2,3), c_2(3,4)$
α_2	$d_1(2,2), d_2(3,3), a(4,4)$

Figure 10: Example showing replay failure with LSTF when there is a flow with three congestion points. A packet represented by p belongs to flow P, with ingress S_P and egress D_P, where $P \in \{A,B,C,D\}$. For simplicity assume all links have a propagation delay of zero. All uncongested routers (white), ingresses and egresses have a transmission time of zero. The three congestion points (shaded grey) have transmission times of $T = 1$ unit

have the same slack at any fixed point of time in α.

Therefore, no bit in $S_{bits}(\alpha, t^*)$ has a negative slack at α after any iteration.

Since no bit in $S_{bits}(\alpha, t^*)$ has a negative slack at α in the swapped LSTF schedule, it contradicts our statement that p^* sees a negative slack when its last bit exits α at time t^*. Hence proved that if a packet p^* sees a negative slack at congestion point α when its last bit exits α at time t^* in the replay, then there must be at least one packet that arrives at α in the replay at or before time t^* and later than the time at which it is scheduled by α in the original schedule.

A.7.3 Replay Failure Example with LSTF

In Figure 10, we present an example where a flow passes through three congestion points and a replay failure occurs with LSTF. When packet a arrives at α_0, it has a slack of 2 (since it waits behind d_1 and d_2 at α_2), while at the same time, packet b has a slack of 1 (since it waits behind a at α_0). As a result, b gets scheduled before a in the LSTF replay. a therefore arrives at α_1 with slack 1 at time 2. c_1 with a zero slack is prioritized over a. This reduces a's slack to zero at time 3, when c_2 is also present at α_1 with zero slack. Scheduling a before c_2, will result in c_2 being overdue (as shown). Likewise, scheduling c_2 before a would have resulted in a getting overdue. Note that in this failure case, a arrives at α_1 at time 2, which is greater than $o(a, \alpha_1) = 1$.

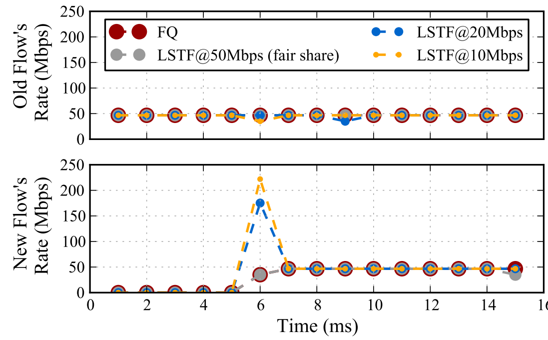


Figure 11: 20 flows share a single bottleneck link of 1Gbps and a 21st flow is added after 5ms. The graph shows the rate allocations for an old flow and the new flow with Fair Queuing and for LSTF with varying r_{est}.

B Understanding how LSTF provides long-term fairness

The reason behind why any slack assignment with $r_{est} < r^*$ leads to convergence to fairness is quite straightforward and is explained by the control experiment shown in Figure 11. 20 long-lived TCP flows share a single bottleneck link of 1Gbps (giving a fair share rate of 50Mbps) and a 21st flow is added after 5ms. Since the first 20 flows have started early, the queue at the bottleneck link already contains packets belonging to these flows.

When $r_{est} = 50Mbps$, the actual queuing delay experienced by a packet is almost equal to the slack value assigned to it. Therefore, at any given point of time, the first packet of each flow present in the queue will have a slack value which is approximately equal to zero. The next packet of each flow will have a higher slack value (around 1500bytes/50Mbps = 0.24ms). By the time the corresponding first packets of every flow in the queue have been transmitted, the slack values of the next packet would also have been reduced to zero and so on. It therefore produces a round-robin pattern for scheduling packets across flows, as is done by FQ. Therefore, when the 21st flow starts at 5ms, with the first packet coming in with zero slack, the next one with 0.24ms slack and so on, it immediately starts following the round-robin pattern as well.

However, when r_{est} is smaller than $50Mbps$, then the packets of the old flows already present in the queue have a higher slack value than what they actually experience in the network. The first packet of every flow in the queue therefore has a slack which is more than 0 when the 21st flow comes in at 5ms. The earlier packets of the new flow therefore get precendence over any of the existing packets of the old flows, resulting in the spike in the rate allocated to the new flow as shown in Figure 11. Nonetheless, with the slack of every newly arriving packet of the 21st flow being higher than the previous one and with the slack of the already queued up packet decreasing with time, the slack value of the first packet in the queue for new flow and the old flows soon *catch up* with each other and the schedule starts following a round robin pattern again. The closer r_{est} is to the fair-share rate, the sooner the slack values of the old flows and the new flow *catch up* with each other. The time that a packet ends up waiting in the queue is upper-bounded by the time it would have waited, had all the flows arrived at the same time and were being serviced at their fair share rate.

One can see how the above logic can be extended for achieving weighted fairness. Moreover, when a packet sees multiple bottlenecks, the slack update (subtraction of the duration for which the packet waits) at the first bottleneck ensures that the next bottleneck takes into account the rate-limiting happening at the first one and the packets are given precedence accordingly.