EFD: An efficient low-overhead scheduler

Jinbang Chen[1], Martin Heusse[2], and Guillaume Urvoy-Keller[3]

[1] Eurecom, Sophia-Antipolis, France
jinbang.chen@eurecom.fr
[2] Grenoble-INP / UJF-Grenoble 1 / UPMF-Grenoble 2 / CNRS, LIG UMR 5217 Grenoble, France
martin.heusse@imag.fr
[3] Laboratoire I3S CNRS, Université de Nice, Sophia Antipolis, France
urvoy@unice.fr

Abstract. Size-based scheduling methods receive a lot of attention as they can greatly enhance the responsiveness perceived by the users. In effect, they give higher priority to small interactive flows which are the important ones for a good user experience. In this paper, we propose a new packet scheduling method, *Early Flow Discard* (EFD), which belongs to the family of Multi-Level Processor Sharing policies. Compared to earlier proposals, the key feature of EFD is the way flow bookkeeping is performed as flow entries are removed from the flow table as soon as there is no more corresponding packet in the queue. In this way, the active flow table remains of small size at all times. EFD is not limited to a scheduling policy but also incorporates a buffer management policy. We show through extensive simulations that EFD retains the most desirable property of more resource intensive size-based methods, namely low response time for short flows, while limiting lock-outs of large flows and effectively protecting low/medium rate multimedia transfers.

Keywords: size-based scheduling, performance, LAS, Run2C

1 Introduction

Size-based scheduling has received a lot of attention from the research community with applications to Web servers [15], Internet traffic [3, 14, 16] or 3G networks [2, 10]. The key idea is to favor short flows at the expense of long ones because short flows are in general related to interactive applications like Email, Web browsing or DNS requests/responses; unlike long flows which represent background traffic. Such a strategy pays off as long as long flows are not completely starved and this generally holds without further intervention for Internet traffic where short flows represent a small portion of the load and thus cannot monopolize the bandwidth.

Despite their unique features, size-based scheduling policies have not yet been moved out of the lab. We believe the main reasons behind this lack of adoption are related to the following general concerns about size-based scheduling approaches:

– Size-based scheduling policies are in essence state-full: each flow needs to be tracked individually. Even though one can argue that those policies should be deployed at bottleneck links which are presumably at the edge of network – hence at

a location where the number of concurrent flows is moderate – the common belief is that stateful mechanisms are to be avoided in the first place.
- Size-based scheduling policies are considered to overly penalize long flows. Despite all its drawbacks, the legacy scheduling/buffer management policy, FIFO/drop tail, does not discriminate against long flows while size-based scheduling solutions tend to impact both the mean response time of flows but also their variance as long flows might lock-out each others.
- As their name indicates, size-based scheduling policies consider a single dimension of a flow, namely, its accumulated size. Still, persistent low rate transfers often convey key traffic, *e.g.*, voice over IP conversations. As a result, it seems natural to account both for the rate and the accumulated amount of bytes of each flow.

A number of works address partially the aforementioned shortcomings of size-based scheduling policies. Although, to the best of our knowledge, none of them fulfill simultaneously the above objectives. This paper presents a new scheduling policy, EFD (Early Flow Discard) that aims at fulfilling the following objectives: (i) Low response time to small flows; (ii) Low bookkeeping cost, *i.e.*, the number of flows tracked at any given time instant remains consistently low; (iii) Differentiating flows based on volumes but also based on rate; (iv) Avoiding lock-outs.

EFD manages the physical queue of an interface (at the IP level) as a set of two virtual queues corresponding to two levels of priority: the high priority queue first and the low priority queue at the tail of the buffer. Formally, EFD belongs to the family of Multi-Level Processor Sharing policies (see Section 2) and is effectively a PS+PS scheduling policy. The key feature of EFD is the way flow bookkeeping is performed. In EFD, we keep an active record only for flows that have at least one packet in the queue. This simple approach allows to fulfill the entire list of objectives listed above. Specifically, in EFD the active flow table size is bounded to a low value. Also, although EFD has a limited memory footprint, it can discriminate against bursty and high rate flows. EFD is not limited to a scheduling policy but also incorporates a buffer management policy, where the packet with smallest priority gets discarded when the queue is full, as opposed to drop tail which blindly discards packets upon arrival. This mechanism is similar to the one used in previous works [13, 4].

Section 2 gives an overview of the related works mentioned above. Section 3 presents the proposed scheduling scheme. The simulation environment, including network setup, network topology and workload appear in Section 4. Then we use simulations to evaluate its performance and compare with other schedulers in Section 5. Finally we conclude the paper in Section 6.

2 Related Work

Classically, size-based scheduling policies are divided into blind and non-blind scheduling policies. A blind size-based scheduling policy is not aware of the job[1] size while a non-blind is. Non blind scheduling policies are applicable to servers [15] where the job

[1] Job is a generic entity in queueing theory. In the context of this work, a job corresponds to a flow.

size is related to the size of the content to transfer. A typical example of non blind policy is the Shortest Remaining Processing Time (SRPT) policy, which is optimal among all scheduling policies, in the sense that it minimizes the average response time.

For the case of network appliances (routers, access points, etc.) the job size, i.e. the total number of bytes to transfer, is not known in advance. Several blind size-based scheduling policies have been proposed. The Least Attained Service (LAS) policy [13] bases its scheduling decision on the amount of service received so far by a flow. LAS is known to be optimal if the flow size distribution has a decreasing hazard rate (DHR) as it becomes, in this context, a special case of the optimal Gittins policy [5]. Some representatives of the family of Multi-Level Processor Sharing (MLPS) scheduling policies [8] have also been proposed to favor short flows. An MLPS policy consists of several levels corresponding to different amounts of attained service of jobs, with possibly a different scheduling policy at each level. In [3], Run2C, which is a specific case of MLPS policy, namely PS+PS, is proposed and contrasted to LAS. With Run2C, short jobs, which are defined as jobs shorter than a specific threshold, are serviced with the highest priority while long jobs are serviced in a background PS queue. Run2C features key characteristics: (i) As (medium and) long jobs share a PS queue, they are less penalized than under LAS; (ii) It is proved analytically in [3] that a M/G/1/PS+PS queue offers a smaller average response time than an M/G/1/PS queue, which is the classical model of a network appliance featuring a FIFO scheduling policy and shared by homogeneous TCP transfers; (iii) Run2C avoids the lock-out phenomenon observed under LAS [7], where a long flow might be blocked for a large amount of time by another long flow.

Run2C and LAS share a number of drawbacks. Flow bookkeeping is complex. LAS requires to keep one state per flow. Run2C needs to check, for each incoming packet, if it belongs to a short or to a long flow. The latter is achieved in [3] thanks to a modification of the TCP protocol so as to encode in the TCP sequence number the actual number of bytes sent by the flow so far. Such an approach, which requires a global modification of all end hosts, is questionable[2]. Moreover, both LAS and Run2C classify flows based on the accumulated number of bytes they have sent, without taking the flow rate into account.

Some approaches propose to detect long flows by inserting the flow in the table probabilistically [4, 12, 9]. The key idea here is to perform a simple random test (with a low probability of success) upon packet arrival to decide if the corresponding flow should be inserted in the table. As long flows generate many packets, it is unlikely to miss them, while many short flow simply go unnoticed. These approaches differ in the way they trade false positive rate against the speed of detection of a long flow.

So far, a single work addresses the problem of accounting for rates in size-based scheduling [7]. It consists in a variant of LAS, Least Attained Recent Service (LARS), where the amount of bytes sent by each flow decays with time according to a fading factor β. LARS is able to handle differently two flows that have sent a similar amount

[2] Other works aim at favoring short flows, by marking the packets at the edge of the network so as to relieve the scheduler from flow bookkeeping [11]. However, the deployment of DiffServ is not envisaged in the near future at the Internet scale.

of bytes but at different rates and it also limits the lock out duration of one long flow by another long flow to a maximum tunable value.

3 Early Flow Discard

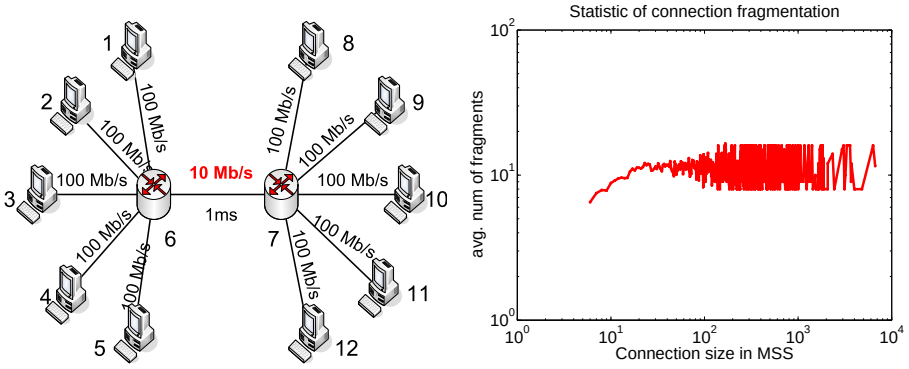
In this section, we describe how EFD manages space and time priority. EFD belongs to the family of Multi-Level Processor Sharing scheduling policy. EFD features two queues. The low priority queue is served only if the high priority queue is empty. Both queues are drained in a FIFO manner at the packet level (which is in general modeled as a PS queue at flow level). In terms of implementation, a single physical queue for packet storage is divided into two virtual queues. The first part of the physical queue is dedicated to the virtual high priority queue while the second part is the low priority queue. A pointer is used to indicate the position of the last packet of the virtual high priority queue. This idea is similar to the one proposed in the Cross-Protect mechanism [9]. We now turn our attention to the flow management in EFD and the enqueuing and dequeuing operations. We eventually discuss the spatial policy used when the physical queue gets full.

3.1 Flow management

EFD maintains a table of active flows, defined here as the set of packets that share a common identity, consisting of a 5-tuple: source and destination addresses, source and destination ports and protocol number. Flows remain in the table as long as there is one corresponding packet in the buffer and discarded when the last packet leaves. Consequently, a TCP connection (or UDP transfers) may be split over time into several fragments handled independently of each other by the scheduler. Note that unlike most scheduling mechanisms that keep per flow states, EFD does not need to use any garbage collection mechanism to clean its flow table. This happens automatically upon departure of the last packet of the flow. A flow entry keeps track of several attributes, including flow identity, flow size counter, number of packets in the queue.

Packet enqueuing For each incoming packet, a lookup is performed in the flow table of EFD. A flow entry is created if the lookup fails and the packet is put at the end of the high priority queue. Otherwise, the flow size counter of the corresponding flow entry is compared to a preset threshold *th*. If the flow size counter exceeds *th*, then the packet is put at the end of the low priority queue; otherwise the packet is inserted at the end of the high priority queue. The purpose of *th* is to favor the start of each flow. In our simulations, we use a *th* of 20 packets (up to 30 Kbytes for packets with size of 1500 bytes each). Obviously, if a connection is broken into several fragments, from the scheduler's perspective, then each time it will handle each fragment as a unique one and assign the start (within threshold *th*) of each fragment a high priority, by means of directing all packets making up the start of each fragment into the high priority queue. We believe that this makes sense as this happens only if the connection has not been active for a significant time –it has not been backlogged for a while– and thus can be considered as fresh.

In practice, several phenomena can lead to break a connection into many fragments. For instance, during connection establishment, the TCP slow start algorithm limits the number of packets in flight so that it does not continuously occupy the buffer. This is however not a problem, as those flows are smaller than th and thus the start of the TCP transfer will receive a high priority. If the flow lasts longer and it is effectively able to use its share of the capacity, then the connection will eventually occupy the buffer without interruption and therefore stay in the flow table. Figure 1(b) illustrates such a scenario (Section 4 details the experimental setup). It is apparent that, as the connection size increases, the number of fragments tends to reach a limit so that, for the longest connections, a small number of fragments correspond to many packets.



(a) Network topology

(b) Number of fragments per connection - workload of 8Mbit/s

Fig. 1.

Packet dequeuing When a packet leaves the queue or gets dropped, it decreases the number of queued packets of the corresponding flow entry. The flow entry stays in the table as long as one corresponding packet is in the queue. So **the flow table size is bounded by the physical queue size** in packets[3]. Indeed, in the worst case, there are as many entries as distinct flows in the physical queue, each with one packet.

This policy ensures that the flow table remains of small size. Also if a flow sends at high rate for a short period of time, its packets will be directed to the low priority queue only for the limited period of time during which the flow is backlogged: EFD is sensitive to flow burstiness.

3.2 Buffer management

When a packet arrives to a queue that is full, EFD first inserts the arriving packet to its appropriate position in the queue, and then drops the packet that is at the end of

[3] In most if not all active equipments – routers, access points – queues are counted in packets and not in bytes.

the (physical) queue. This buffer policy implicitly gives space priority to short [4] flows, which differs from the traditional droptail buffer management policy. This approach is similar to the Knock-Out mechanism of [4] and the buffer management proposed to LAS in [13]. As large flows in the Internet are mostly TCP flows, we can expect that they will recover from a loss event with a fast retransmit; unlike short flows that might time out.

4 Performance Evaluation Set Up

In this section, we present the network set up – network topology and workload – used to evaluate the performance of EFD and to compare it to other scheduling policies. All simulations are done using QualNet [1].

4.1 Network Topology

We evaluate the performance of EFD and compare it to other scheduling policies for the case of a single bottleneck network, using a classical dumbbell topology depicted in Fig. 1(a).

A group of senders (nodes 1 to 5) are connected to a router (node 6) by 100Mbps bandwidth links and a group of receivers (nodes 8 to 12) are connected to another router (node 7) with a 100Mbps bandwidth link. The two aggregation routers are connected to each other with a link at 10Mbps. All links have 1 ms propagation delay.

All nodes use FIFO queues, except the bottleneck node which uses one of the four scheduling policies that we compare in this work: FIFO, LAS, RuN2C or EFD. The bottleneck buffer has a finite size of 300 packets.

4.2 Workload generation

Data transfer requests arrive according to a Poisson process, the server and the client are picked at random and the content requested is distributed according to a bounded Zipf distributed flow sizes. A bounded Zipf distribution is a discrete analog of a continuous bounded Pareto distribution.

Transfers are performed under TCP or UDP depending on the simulation. In all cases, the global load is controlled by tuning the arrival rate of requests. For each simulation set-up, we consider an underload and an overload regime, which correspond respectively to workloads of 8 and 15 Mb/s (80% and 150% of the bottleneck capacity). For TCP simulations, we use the GENERIC-FTP model of Qualnet, which corresponds to an unidirectional transfer of data. For UDP transfers, we use a CBR application model where one controls the inter-packet arrival time. The latter enables to control the exact rate at which packets are sent to the bottleneck. In both TCP and UDP cases, IP packets have a size of 1500 bytes.

[4] Due to the discussion in the above paragraph, a short flow is a part of a connection whose rate is moderate.

5 Performance Evaluation

In this section, we compare the performance of EFD to other scheduling policies. Our objective is to illustrate the ability of EFD to fulfill the 4 objectives listed in the introduction, namely (i) low bookkeeping cost, (ii) low response time to small flows, (iii) avoiding lock-outs, (iv) protecting long lasting delay sensitive flows.

To illustrate the first 3 items, we consider a TCP workload with homogeneous transfers, i.e., transfers that take place on paths having similar characteristics. For the last item - protecting long lived delay sensitive flows - we add a UDP workload to the TCP workload in the form of a CBR traffic, in order to highlight the behavior of each scheduler in presence of long lasting delay sensitive flows.

5.1 Overhead of flow state keeping

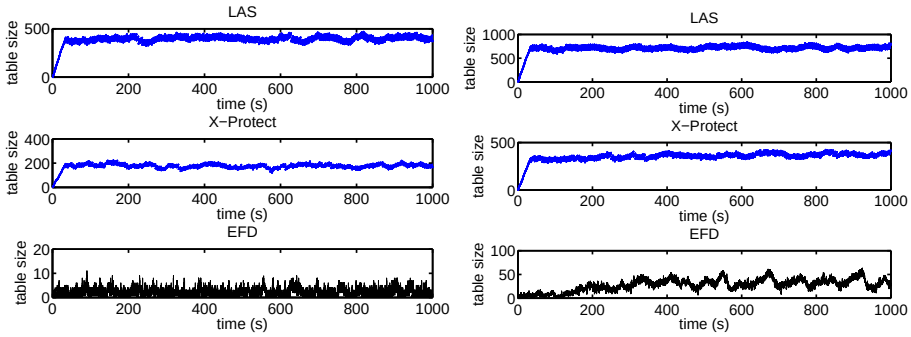
The approaches to maintain the flow table in the size-based scheduling policies proposed so far can be categorized as follows:

- Full flow table approach as in LAS [13]. An argument in favor of keeping one state per active flow is that the number of flows to handle remains moderate as it is expected that such a scheduling policy be implemented at the edge of the Internet.
- No flow table approach: an external mechanism marks the packets or the information is implicit (coded in the SEQ number in Run2C) [3, 11]
- Probabilistic approaches: a test is performed at each packet arrival for flows that have not already be incorporated in the flow table [4, 9, 12]. The test is calibrated in such a way that only long flows should end up in the flow table. Still, false positives are possible. Several options have been envisaged to combat this phenomenon especially, a re-testing approach [12] or an approach where the flows in the flow table are actually considered as long flows once they have generated more than a certain amount of packets/bytes after their initial insertion [4].
- EFD deterministic approach: the EFD approach is fully deterministic as flow entries are removed from the flow table once they have no more packet in the queue.

In this section, we compare all the approaches presented except the ”No flow table approach” for our TCP workload scenario (see Section 4.2). We consider one representative of each family: LAS, X-Protect and EFD. We term X-Protect a Multi-Level Processor Scheduling policy that maintains two queues, similarly to Run2C, but uses the probabilistic mechanism proposed in [9] to track long flows[5]. As for the actual scheduling of packets, X-Protect mimics Run2C based on the information it possesses. If the packet does not belong to a flow in the flow table nor passes the test, it is put in the high priority queue. If it belongs to a flow in the flow table, it is put either in the high priority queue or in the low priority queue, depending on the amount of bytes sent by the flow. We use a threshold of 30KB, similar to the one used for EFD.

The evolution of flow table size over time for load of 8Mbit/s (underload) and 15Mbit/s (overload) are shown in Fig. 2. For LAS and X-Protect, the flow table is visited every 5 seconds and the flows that have been inactive for 30 seconds are removed.

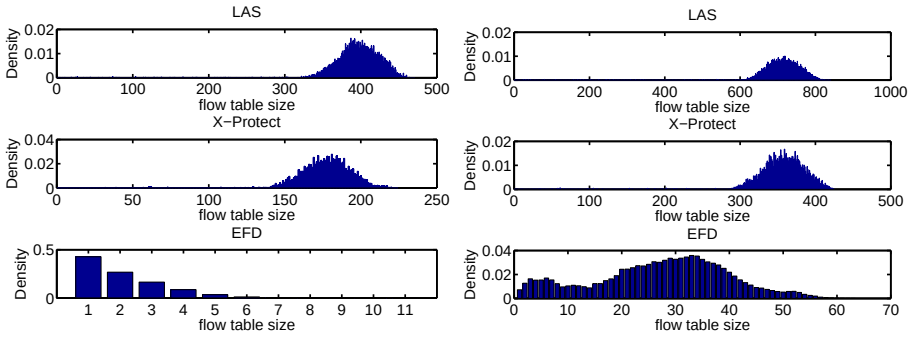
[5] Note that this mechanism is proposed in [9] to do admission control function and not a scheduling.



(a) workload of 8Mbit/s (underload) (b) workload of 15Mbit/s (overload)

Fig. 2. Evolution of flow table size over time

We observe how X-Protect roughly halves the number of tracked flows, compared to LAS. By contrast, EFD reduces it by one order of magnitude. The reason why X-Protect offers deceptive performance is the race condition that exists between the flow size distribution and the probabilistic detection mechanism. Indeed, even though a low probability, say 1%, is used to test if a flow is a long, there exists so many short flows that the number of false positives becomes quite large, which prevents the flow table from being significantly smaller than in LAS. The histograms in Fig. 3 confirm the good performance of EFD in underload and also overload, as EFD keeps the flow table size to a few 10s of entries at most. Note that this is clearly smaller than the actual queue size (300 packets) that constitutes an upper bound on the flow table size in EFD as explained before.



(a) workload of 8Mbit/s (underload) (b) workload of 15Mbit/s (overload)

Fig. 3. Histogram of the flow table size

5.2 Mean response time

Response time is a key metric for a lot of applications, especially interactive ones. An objective of EFD and size-based scheduling policies in general is to favor interactive applications, hence the emphasis put on response time. We consider four scheduling policies: FIFO, LAS, Run2C and EFD. FIFO is the current de facto standard and it is thus important to compare the performance of EFD to this policy. LAS can be considered as a reference in terms of (blind) size-based scheduling policies as a lot of other disciplines have positioned themselves with respect to LAS. Run2C, for instance, aims at avoiding the lock out of long flows observed more often with LAS than for *e.g.* FIFO. We do not consider the X-protect policy discussed in Section 5.1, as Run2C can be considered as a perfect version of X-protect since Run2C distinguishes packets of flows below and above the threshold *th* (we use the same threshold *th* for both EFD and Run2C).

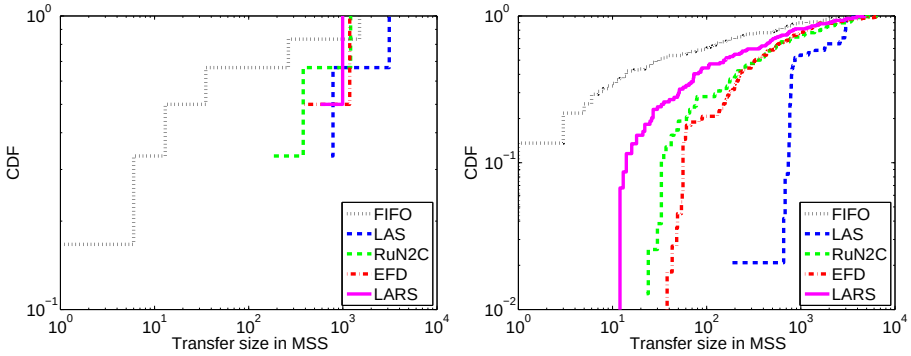
Response times are computed only for flows that complete their transfer before the end of the simulation. When comparing response times, one must thus also consider the amount of traffic due to flows that terminated their transfer and to flows that did not complete. The lack of completion of a flow can be due to a premature end of simulation. However, in overload and for long enough simulations as in our case, the main reason is that they were set aside by the scheduler.

We first turn our attention to the aggregate volumes of traffic per policy for the underload and overload cases. We observe no significant difference between the different scheduling policies in terms both of number of complete and incomplete connections. The various scheduling policies lead to a similar level of medium[6] utilization.

In contrast, when looking at the distribution of incomplete transfers, it appears that the flows killed by the different scheduling policies are not the same. We present in Fig. 4 the distribution of incomplete transfers where the size of a transfer is the total amount of MSS packets transferred at the end of the simulation. A transfer is deemed incomplete if we do not observe a proper TCP tear down with two FIN flags. As expected, we observe that FIFO tends to kill a lot of small flows while the other policies discriminate long flows.

Distributions of the response times for the (complete) short and long transfers in underload and overload conditions are presented in Fig. 5. Under all load conditions, LAS, EFD and Run2C manage to significantly improve the response time of the short flows as compared to FIFO. EFD and Run2C offer similar performance. They both have a transition of behavior at about *th* value ($th = 20$ MSS). Still, the transition of EFD is smoother than the one of Run2C. This was expected as Run2C applies a strict rule: below or above *th* for a given transfer, whereas EFD can further cut a long transfer into fragments which individually go first to the high priority queue. Overall, EFD provides similar or slightly better performance than Run2C with a minimal price in terms of flow bookkeeping. LAS offers the best response time of size-based scheduling policies in our experiment for small and intermediate size flows. For large flows its performance are equivalent to the other policies in underload and significantly better for the overload case. However, one has to keep in mind that in overload conditions, LAS deliberately

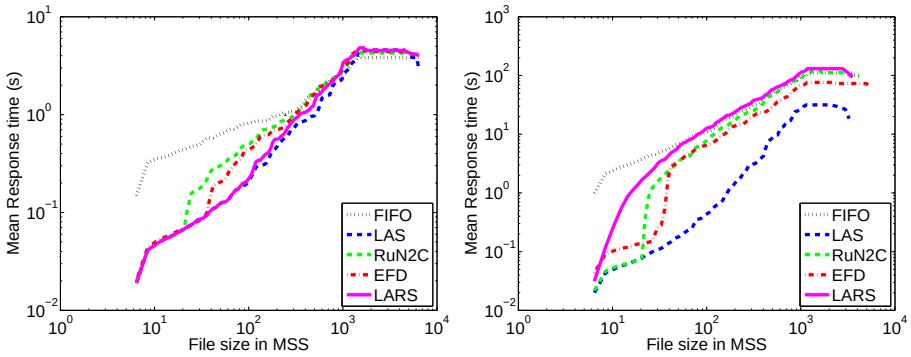
[6] The medium is the IP path as those policies operate at the IP level.



(a) workload of 8Mbit/s (underload) (b) workload of 15Mbit/s (overload)

Fig. 4. Distributions of incomplete transfers size

killed a large set of long flows (see Fig. 4), hence its apparent better performance. LARS behaves similarly to LAS in underload and degrades to fair queueing –which brings it close to FIFO in this case– when the networks is overloaded.



(a) workload of 8Mbit/s (underload) (b) workload of 15Mbit/s (overload)

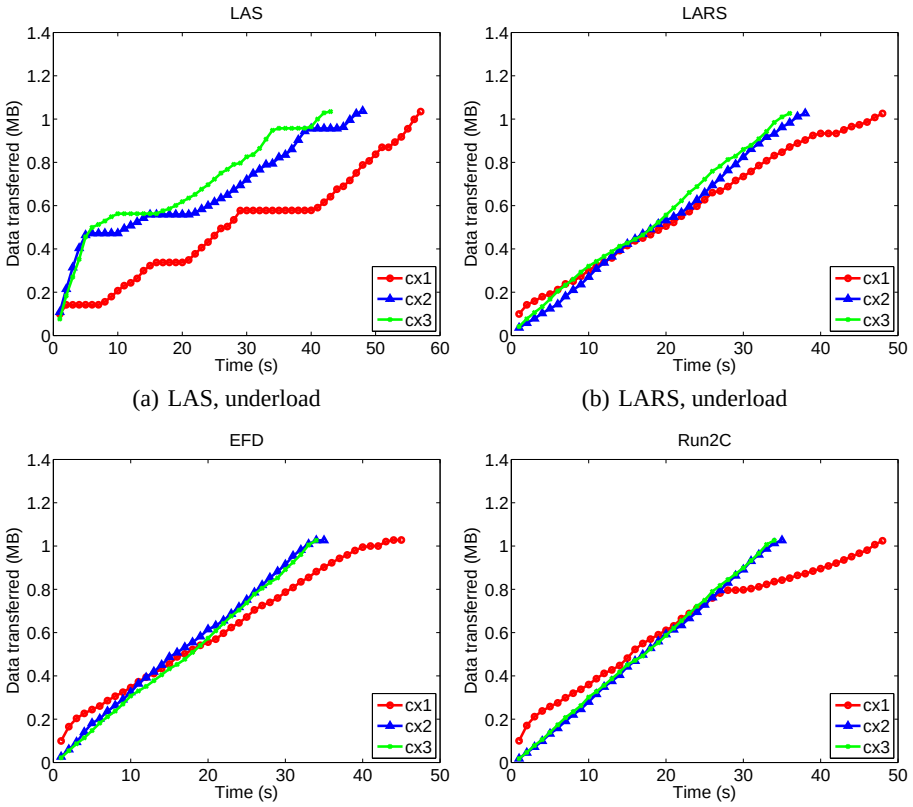
Fig. 5. Conditional mean response time

5.3 Lock-outs

The low priority queue of EFD is managed as a FIFO queue. As such, we expect EFD, similarly to Run2C, to avoid lock-outs observed under LAS whereby an ongoing long transfer is blocked for a significant amount of time by a newer transfer of significant size. This behavior of LAS is clearly observable in Figure 6(a) where the progress (accumulated amount of bytes sent) over time of the 3 largest transfers of one of the above simulations[7]. We indeed observe large periods of times where the transfers experience

[7] Those 3 connections did not start at the same time, the time axis is relative to their starting dates.

no progress, which leads to several plateaus. This is clearly in contrast to the cases of LARS, EFD and to a lesser extent of Run2C, for the same connections, shown in Figures 6(b), 6(c) and 6(d) respectively. The progress of the connections in the latter cases is indeed clearly smoother with no noticeable plateau.



(a) LAS, underload (b) LARS, underload

(c) EFD, underload (d) Run2C, underload

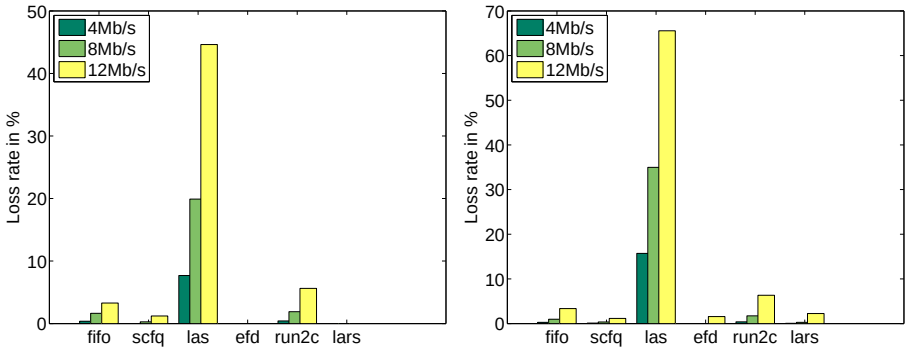
Fig. 6. Time diagrams of the 3 largest TCP transfers under LAS, LARS, EFD and Run2C (underload), relative to the start of each transfer

5.4 The Case of Multimedia Traffic

In the TCP scenario considered above, FTP servers were homogeneous in the sense that they had the same access link capacity and the same latency to each client. The transfer rate was controlled by TCP. In such conditions, it is difficult to illustrate how EFD takes into accounts the actual transmission rate of data sources. In this section, we have added a single CBR flow to the TCP workload used previously.

We consider two rates 64Kb/s and 500Kb/s for the CBR flow, representing typical audio (e.g., VoIP) and video stream (e.g., YouTube video - even though the YouTube uses HTTP streaming) respectively. The background load also varies - 4, 8 and 12Mbps-

which correspond to underload/moderate/overload regimes as the bottleneck capacity is 10 Mbps. To avoid the warm-up period of the background workload, the CBR flow is started at time t=10s and keeps on sending packets continuously until the end of the simulation. The simulation lasts for 1000 seconds. Since small buffers are prone to packet loss, we assign to the bottleneck a buffer of 50 packets, instead of 300 packets previously. The loss rates experienced by the CBR flow are given in Fig. 7, in which a well-known fair scheduling scheme called SCFQ [6] is added for the comparison, aparting from other disciplines mentioned hereinbefore.



(a) a CBR flow with rate of 64Kb/s (b) a CBR flow with rate of 500Kb/s

Fig. 7. Loss rate experienced by a CBR flow in different background loads

As we can see from the figure, for the case of a CBR flow with rate of 64Kbps, LAS discards a large fraction of packets even at low load. This was expected as LAS only considers the accumulated volume of traffic of the flow and even at 64 kbps, the CBR flow has sent more than 8 MB of data in 1000 s (without taking the Ethernet/IP layers overhead into account). In contrast, FIFO, SCFQ and Run2C offer low loss rates in the order of a few percents at most. As for EFD and LARS, they effectively protect the CBR flow under all load conditions.

As the rate of the CBR flow increases from 64Kbps to 500Kbps, no packet loss is observed for EFD in underload/moderate load conditions, similarly to SCFQ, whereas the other scheduling disciplines (FIFO, LAS, Run2C and LARS) are hit at various degrees. In overload, EFD and LARS blow up similarly to LAS (which still represents an upper bound on the loss rate as the CBR flow is continuously granted the lowest priority). EFD behaves slightly better than LARS as the load in the high priority queue is by definition lower under EFD than under Run2C.

When looking at the above results from a high level perspective, one can think at first sight that FIFO and SCFQ do a decent job as they provide low loss rates to the CBR flow in most scenarios (under or overload). However, those apparently appealing results are a side effect of a well-known and non desirable behavior of FIFO. Indeed, under FIFO, the non responsive CBR flow adversely impacts the TCP workload, leading to high loss rates. This is especially true for the CBR flow working at 500 kbps. SCFQ tends

to behave similarly if not paired with an appropriate buffer management policy [6]. In contrast, LARS and EFD offer a nice trade-off as they manage to simultaneously grant low loss rates to the CBR flow with a low penalty to the TCP background workload. Run2C avoids the infinite memory of LAS but still features quite high loss rates since the CBR flow remains continuously stuck in the low priority queue.

Overall, EFD manages to keep the desirable properties of size-based scheduling policies and in addition manages, with a low bookkeeping cost, to protect multimedia flows as it implicitly accounts for the rate of this flow and not only its accumulated volume.

6 Conclusion

In this paper, we have proposed a simple but efficient packet scheduling scheme called *Early Flow Discard* (EFD) that uses a fixed threshold for flow discrimination while taking flow rates into account at the same time. EFD possesses the key feature of keeping an active record only for flows that have one packet at least in the queue. With this strategy, EFD caps the amount of active flow that it tracks to the queue size in packets.

Extensive network simulations revealed that EFD, as a blind scheduler, retains the good properties of LAS like small response times to short flows. In addition, a significant decrease of bookkeeping overhead, of at least one order of magnitude is obtained as compared to LAS, which is convincing from a practical point of view. Lock-outs which form the Achilles' heel of LAS are avoided in EFD, similarly to Run2C. In contrast to LAS and Run2C, EFD inherently takes both volume and rate into account in its scheduling decision due to the way flow bookkeeping is performed. We further demonstrated that EFD can efficiently protect low/medium multimedia flows in most situations.

Future directions of research on EFD will be to test its applicability to WLAN infrastructure networks, where the half-duplex nature of the MAC protocol needs to be taken into account [16].

References

1. QualNet 4.5. Scalable Networks
2. Aalto, S., Lassila, P.: Impact of size-based scheduling on flow level performance in wireless downlink data channels. Managing Traffic Performance in Converged Networks pp. 1096–1107 (2007)
3. Avrachenkov, K., Ayesta, U., Brown, P., Nyberg, E.: Differentiation between short and long tcp flows: Predictability of the response time. In: Proc. IEEE INFOCOM (2004)
4. Divakaran, D.M., Carofiglio, G., Altman, E., Primet, P.V.B.: A flow scheduler architecture. In: Networking. pp. 122–134 (2010)
5. Gittins, J.: Multi-armed bandit allocation indices. Wiley-Interscience (1989)
6. Golestani, S.: A self-clocked fair queueing scheme for broadband applications. In: INFOCOM '94. Networking for Global Communications., 13th Proceedings IEEE. pp. 636 –646 vol.2 (Jun 1994)
7. Heusse, M., Urvoy-Keller, G., Duda, A., Brown, T.X.: Least attained recent service for packet scheduling over wireless lans. In: WoWMoM 2010 (2010)

8. Kleinrock, L.: Computer Applications, Volume 2, Queueing Systems. Wiley-Interscience, 1 edn. (April 1976)
9. Kortebi, A., Oueslati, S., Roberts, J.: Cross-protect: Implicit service differentiation and admission control. In: IEEE HPSR (2004)
10. Lassila, P., Aalto, S.: Combining opportunistic and size-based scheduling in wireless systems. In: MSWiM '08: Proceedings of the 11th international symposium on Modeling, analysis and simulation of wireless and mobile systems. pp. 323–332. ACM, New York, NY, USA (2008)
11. Noureddine, W., Tobagi, F.: Improving the performance of interactive tcp applications using service differentiation. In: Computer Networks Journal. pp. 2002–354. IEEE (2002)
12. Psounis, K., Ghosh, A., Prabhakar, B., Wang, G.: Sift: A simple algorithm for tracking elephant flows, and taking advantage of power laws. In: 43rd Annual Allerton Conference on Control, Communication and Computing (2005)
13. Rai, I.A., Biersack, E.W., Urvoy-keller, G.: Size-based scheduling to improve the performance of short tcp flows. IEEE Network pp. 12–17, vol.19 (2004)
14. Rai, I.A., Urvoy-Keller, G., Vernon, M.K., Biersack, E.W.: Performance analysis of las-based scheduling disciplines in a packet switched network. In: SIGMETRICS 2004/PERFORMANCE 2004: Proceedings of the joint international conference on Measurement and modeling of computer systems. vol. 32, pp. 106–117. ACM Press, New York, NY, USA (June 2004)
15. Schroeder, B., Harchol-Balter, M.: Web servers under overload: How scheduling can help. ACM Trans. Internet Technol. (1), 20–52, vol.6 (2006)
16. Urvoy-Keller, G., Beylot, A.L.: Improving flow level fairness and interactivity in wlans using size-based scheduling policies. In: MSWiM '08: Proceedings of the 11th international symposium on Modeling, analysis and simulation of wireless and mobile systems. pp. 333–340. ACM (2008)