Optimal Scheduling for Jobs with Progressive Deadlines

Kristen Gardner
Computer Science Department
Carnegie Mellon University
ksgardne@cs.cmu.edu

Sem Borst
Alcatel-Lucent Bell Labs
sem@research.bell-labs.com

Mor Harchol-Balter
Computer Science Department
Carnegie Mellon University
harchol@cs.cmu.edu

***Abstract*—This paper considers the problem of server-side scheduling for jobs composed of multiple pieces with consecutive (progressive) deadlines. One example is server-side scheduling for video service, where clients request flows of content from a server with limited capacity, and any content not delivered by its deadline is lost. We consider the simultaneous goals of 1) minimizing overall loss, and 2) differentiating loss fractions across classes of flows in proportion to relative weights. State-of-the-art policies, like Discriminatory Processor Sharing and Weighted Fair Queueing, use a fixed static proportional allocation of service rate and fail to achieve both goals. The well-known Earliest Deadline First policy minimizes overall loss, but fails to provide proportional loss across flows, because it treats packets as independent jobs.**

This paper introduces the Earliest Progressive Deadline First (EPDF) class of policies. We prove that all policies in this broad class minimize overall loss. Furthermore, we demonstrate that many EPDF policies accurately differentiate loss fractions in proportion to class weights, satisfying the second goal.

I. Introduction

Real-time scheduling theory deals with a server that receives requests from clients, where each request has a deadline. In contrast to traditional scheduling theory, in which the goal typically is to minimize response time, the primary goal of real-time scheduling is to find a server-side schedule that completes each request by its deadline, if possible.

In all prior work on real-time scheduling, each job (request) has only a single deadline. In this paper, we introduce the concept of *progressive* deadlines. A job with progressive deadlines consists of multiple pieces of work, each with its own deadline. Thus, there is a series of deadlines associated with a single job. We assume that progressive deadlines are hard deadlines to the end of service; if a piece of a job is not complete by its deadline, that piece is *dropped/lost*, though subsequent pieces of the job continue to be delivered, and those may still complete on time. Throughout, we define the *overall loss fraction* as the total amount of work (content) that is lost divided by the total amount of work that is requested.

Jobs with progressive deadlines are actually quite common in many networking applications. As an example throughout this paper, we discuss the problem of server-side scheduling algorithms for video service. Consider a single server that must provide video content to multiple clients (see Figure 1). Each client i plays its video at rate d_i. A client must receive each packet of her video in time for it to be played, and any packet not delivered on time is lost (clients are known to be intolerant to rebuffering [7]). The server can deliver packets to clients early; however, there is a limit to how much content a client can store since the client has a limited buffer. Servers are aware of the limit at each client and thus will not send content to a client with a full buffer faster than d_i.

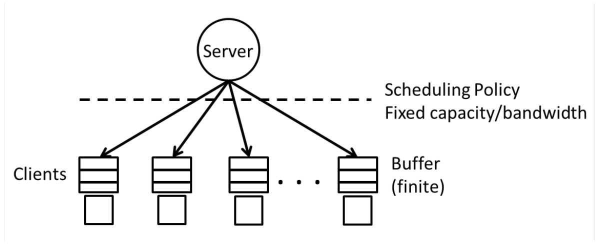


Fig. 1: A single server must provide video content to multiple clients with progressive deadlines, subject to a fixed capacity limit. When more content is being requested than the server can provide, the server must decide which content to drop.

Ideally, the server is able to meet all progressive deadlines. However, the server has a fixed bandwidth/capacity, C, that limits the amount of service that it can provide and thus its ability to satisfy all clients' requests on time. At all times, the *server-side scheduling decision* is to determine to which clients to provide service, and how to split its limited capacity among these clients. The goal of these scheduling policies is to enhance clients' Quality of Experience (QoE). There are many components that influence a client's QoE; however, one key factor is the fraction of the client's content that is lost. Furthermore, clients may have different tolerances to loss; for example, some classes of clients may have paid more to receive a better QoE.

In this paper, we consider two equally important goals for improving client QoE. First, we want to minimize the overall loss fraction across all clients. Second, we want to distribute loss fraction inversely proportionally to class weights. For example, if the high weight class has twice the weight of the low weight class, then the loss fraction for the high weight

This material is based upon work supported by the National Science Foundation Graduate Research Fellowshow Program under Grant No. DGE-1252522. The early stages of the work were done in an internship project at Bell Labs.

This work was funded in part by NSF-CMMI-1334194 as well as a Computational Thinking grant from Microsoft Research.

class should equal half that for the low weight class. Although existing server-side scheduling policies consider each of these goals individually, none consider them together.

Servers typically use Discriminatory Processor Sharing (DPS) or its discretized approximations to provide different levels of service to different classes of clients (see Section II for more detail). DPS-like policies set *service rates* according to class weights. These policies are deadline-agnostic and thus DPS neither minimizes the overall loss fraction, nor is it able to control per-class loss fractions.

The Earliest Deadline First (EDF) policy is known to minimize the overall loss fraction (see Section II). However, EDF cannot provide different service levels to different classes because, under EDF, each packet is considered its own job and has its own deadline; packets are not associated with flows.

Progressive deadlines allow us to achieve both our goals. Each packet is associated with its flow, and the deadlines for the packets within a flow are the progressive deadlines for the flow. By grouping packets at the flow level, a scheduling policy can make decisions based on the properties of entire flows, rather than just individual packets, as in EDF. This enables us to define a broad class of policies that considers *both* packet deadlines and flow weights.

We introduce the Earliest Progressive Deadline First (EPDF) class of policies. Policies in this class always provide content to the flow(s) with the soonest upcoming deadline(s). This is equivalent to serving the flow(s) with the least content in the client buffer(s), where content is viewed in terms of the duration stored - *not* the number of bytes stored. We assume that the server is aware of each client's buffer contents at all times, since it knows the buffer limit at each client and how much it has sent to each client (this is an approximation ignoring TCP technicalities). When the server has insufficient capacity to meet all progressive deadlines, and there are multiple flows with empty buffers (i.e., flows that will incur loss if they do not receive content immediately), EPDF policies may distribute loss among these flows *according to any rule*. In Section IV, we define several rules that work toward the goal of distributing loss fraction in proportion to class weights.

In this paper, we address three central questions:

1) **Is the entire EPDF class optimal with respect to minimizing the overall loss fraction?** Intuitively, the optimality of the entire EPDF class may seem to be a consequence of the optimality of the traditional EDF policy, since EDF is a member of EPDF. However, EPDF is a much broader class of policies that allows for a considerable amount of flexibility in scheduling decisions. For example, an EPDF policy might choose to entirely starve one flow, while meeting all of another flow's progressive deadlines. In Section V, we *prove* that the entire EPDF class is in fact optimal for minimizing overall loss fraction.
2) **How does the performance of EPDF compare to that of state-of-the-art policies, like DPS, with respect to allocating loss fraction in proportion to class weights?** We show that EPDF is able to achieve a specified target loss ratio between classes, while DPS is not (see Figure 2). This is because DPS provides *service* in proportion to weights, which is not the same as providing *loss* in proportion to weights (see Figure 3). Furthermore, we show that DPS is extremely sensitive to fluctuations in system parameters, making it even more difficult to choose appropriate DPS weights.
3) **Are the EPDF policies practically feasible?** While we do not pursue implementation issues in detail, several policies in the EPDF class are easy to implement because they require only minimal state information. We show that these attain nearly the same performance as the best EPDF policies with respect to allocating loss in proportion to class weights.

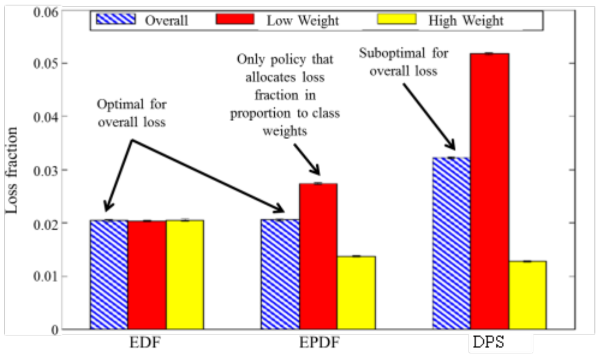


Fig. 2: A subset of our results indicating the difficulties in meeting the dual goals of minimizing overall loss fraction and allocating loss fraction in proportion to class weights. Only EPDF is able to achieve both goals simultaneously.

II. Prior Work

Neither the goal of minimizing the overall loss fraction, nor the goal of differentiating among classes, is new to server-side scheduling. The Earliest Deadline First (EDF) policy is known to minimize the overall fraction of content lost [13], [17]. Under EDF, each packet is viewed as a separate job and has its own deadline. At all times, the server sends the packet with the soonest deadline. EDF is a well-studied policy, the optimality of which has been proven in a wide variety of scenarios [1], [10], [11], [12], [13], [14], [17]. However, because EDF allows only a single deadline per job, it cannot associate packets with flows, and it therefore cannot achieve our second goal of allocating loss fraction in proportion to class weights. The Weighted Earliest Due Date (WEDD) policy extends EDF by using class weights to determine which packet(s) should be sent when the server has insufficient capacity to meet all deadlines [4]. Fluid versions of both EDF and WEDD are members of the EPDF class, which we introduce in Section IV.

Historically, the IntServ and DiffServ paradigms aimed to provide differentiated service levels among classes of clients, favoring clients of higher classes [3], [5]. However, such paradigms only guarantee that the quality of service experienced by different classes of clients will differ, not the factor

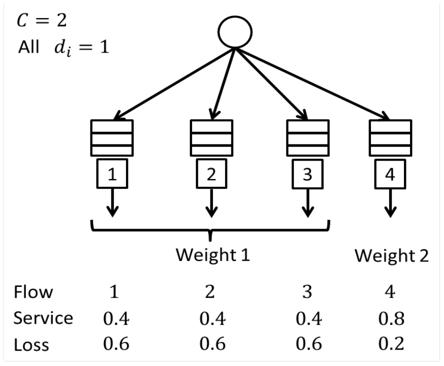


Fig. 3: *Proportional service $\neq$ proportional loss.* A server with capacity $C = 2$ provides content to four clients, each with bitrate $d = 1$ and no stored buffer contents. Clients 1, 2, and 3 have weight 1, and client 4 has weight 2. If the server allocates its capacity in proportion to weight, as in DPS, clients 1, 2, and 3 each receive rate 0.4, and client 4 receives rate 0.8. Thus, clients 1, 2, and 3 all lose a 0.6 fraction of their content, while client 4 loses a 0.2 fraction of its content. Loss fractions are not in proportion to class weights.

by which it will differ. Our goal is to provide a *specific ratio* of *loss fraction* among different classes.

Under state-of-the-art policies like Discriminatory Processor Sharing (DPS), each client i is assigned a fixed share, ϕ_i [8]. At all times, flow i is allocated rate $\frac{\phi_i}{\sum_j \phi_j}C$, where the sum is taken over all flows currently competing for service. While ϕ_i may be set to take into account class weights, ϕ_i is deadline-agnostic. As a result, DPS achieves neither of our two goals. With respect to minimizing the overall loss fraction, a DPS server has no flexibility to provide additional service to clients that are losing content. Thus, DPS has no control over the total content lost. DPS is also unable to allocate loss fraction in proportion to class weights. The first issue is illustrated in Figure 3: providing service rate in proportion to class weights is not equivalent to distributing loss fraction in proportion to class weights, and finding a mapping between these is difficult. Secondly, as we will see, DPS is highly sensitive to fluctuations in system parameters (load, service capacity, relative class loads), meaning that as these parameters change, the weights would have to be constantly readjusted. Hence, DPS and the discretized approximations that are currently in use - including Weighted Fair Queueing (WFQ) [6], [15], Worst-Case Fair Weighted Fair Queueing [2], Self-Clocked Fair Queueing [9], and others (see [18] and [16] for a review of additional policies) - achieve neither of our two goals.

III. Model

Throughout, we assume that there is a single server with finite capacity C. Clients arrive to the system with average rate λ. Each client i arrives at time A_i and requests a video of duration B_i, which it will watch at a constant rate of d_i bytes/second (see Table I). Each client also has an associated weight w_i indicating its importance. Each client i has a buffer that can store at most $V_i^{\max}$ seconds of content. We denote the duration of the content stored in the buffer of the i^{th} flow at time t by $V_i(t)$. Note that $V_i(t)$ is *not* the number of bytes in buffer i, but rather the number of *seconds* of video in buffer i, which takes into account the bitrate d_i at which flow i is being played. $V_i^{\max}(t)$ denotes the maximum duration of content that *could be* stored by flow i at time t, namely, $V_i^{\max}(t) = \min\{B_i - (t - A_i)\ ,\ V_i^{\max}\}$. For example, if flow i arrives at time $A_i = 0$s with duration $B_i = 60$s and $V_i^{\max} = 10$s, then $V_i^{\max}(t) = 10$s for $0 \leq t < 50$s, but $V_i^{\max}(t) = 60 - t$ for $50 \leq t \leq 60$s.

For each flow i, the server is aware of $V_i^{\max}$, w_i, d_i, A_i, and B_i, and therefore is also able to infer $V_i(t)$, the exact buffer state at time t. It is the server's decision alone when to deliver content and to whom. In practice, clients might request additional content when their buffers are low; however, for the purpose of this paper, we leave all decisions up to the server, since the server is aware of all clients' states.

Observe that in practice, a flow is composed of multiple small chunks. For example, a 2 hour movie may consist of 3600 2-second chunks. Our model instead considers an idealized fluid system, whereby the "chunks" are infinitesimally small. This is the natural convention used in the analytical modeling of flows (see [15]). We also assume that clients remain in the system for the duration of their video and wish to play content continuously; they do not pause and resume the video, and they do not depart before the video has ended.

The rate d_i determines the progressive deadlines for flow i: a video that is requested at time A_i and has duration B_i requires $d_i(t - A_i)$ bytes by time t, for $A_i \leq t \leq A_i + B_i$. A flow is **active** from its arrival time A_i until time $A_i + B_i$, at which time it becomes **expired**. Active flows may be **incomplete** or **complete**. An active-incomplete flow (denoted by act&inc) has not yet received all of its content, whereas an active-complete flow (denoted by act&comp) has received all of its content, but still has video remaining in its buffer. For both types of active flows, there remains content that the client has yet to watch. *The server splits its capacity, C, among active-incomplete flows only*, according to some scheduling policy, which must answer two questions:

1) At all times t, which set of flows should receive service?
2) How much service should each of these flows receive? We will denote by $c_i(t)$ the service rate provided to flow i at time t.

If a flow i does not have any content in its buffer at time t, and receives service rate $< d_i$, then the flow will *lose* content. The instantaneous loss rate at t is $d_i - c_i(t)$, the difference between the rate at which the client needs content and the rate at which the server provides it. The instantaneous loss fraction at t is $\frac{d_i - c_i(t)}{d_i}$. The total loss fraction from a flow is the ratio of the total amount of content lost to the total amount

A_i	The arrival time of flow i
B_i	The duration of flow i
C	The total available service rate
$c_i(t)$	The service rate allocated to flow i at time t
d_i	The rate at which flow i requires content
$V_i(t)$	The duration of the content in flow i's buffer at time t, in units of time
$V_i^{\max}$	The fixed capacity limit of flow i's buffer, in units of time
$V_i^{\max}(t)$	The maximum duration of content that can be stored in flow i's buffer at time t, in units of time
w_i	The weight of flow i

TABLE I: Summary of notation

of content requested:

$$\text{Total fraction lost from flow } i = \frac{1}{d_i B_i} \int_{\substack{t \in [A_i, A_i+B_i] \\ s.t.\ V_i(t)=0}} (d_i - c_i(t))^+ dt. \quad (1)$$

The total loss fraction from a set of flows is the ratio of the total amount of content lost from all flows in the set to the total amount of content requested by all flows in the set (this is not the same as the average of the total fractions lost from each flow in the set).

Limitations of our Model. Our model does not address some practical considerations. Firstly, our server may be part of a Content Delivery Network (CDN) from which it obtains content, and its own buffer might be limited, causing the server to drop content. We ignore this loss at the server by assuming that the server's buffer is unlimited. Secondly, we assume that all server scheduling decisions are made at the application layer, and ignore TCP considerations below this layer.

IV. Earliest Progressive Deadline First

We propose the Earliest Progressive Deadline First (EPDF) class of scheduling policies. At a high level, the idea behind all EPDF policies is to allocate the full C to the active-incomplete flow(s) with the earliest progressive deadline(s). An equivalent, and perhaps more intuitive, way of thinking about this policy is that C is always shared equally among the active-incomplete flows with the smallest $V_i(t)$, i.e., the smallest buffer contents. These are also the flows with the next progressive deadline, because a flow's next progressive deadline is the time at which that flow would begin dropping content if it receives no additional service. This is exactly the time until its current buffer content depletes.

We describe EPDF scheduling first in the case of no loss at time t, and then in the case of loss at time t. If no loss is incurred at time t, then it must be the case that either $\min_i\{V_i(t)\} > 0$, or $\min_i\{V_i(t)\} = 0$ and $\sum_{i:\ V_i(t)=0} d_i < C$, where only thoss i are considered that correspond to active-incomplete flows at time t. In this case, all EPDF policies behave in the same way (see Figure 4). C is shared among the active-incomplete flows with *smallest* $V_i(t)$ in proportion to bitrate, ignoring weights. That is, if for flows i and j, $d_i = 2d_j$ and $V_i(t) = V_j(t)$, then under EPDF the server will set $c_i(t) = 2c_j(t)$. If one of these flows i has $V_i(t) = V_i^{\max}(t)$, and the nominal share of flow i is higher than d_i, then flow i is only allocated d_i, and the excess is re-allocated to flows that are not rate-limited, if any, starting with those with smallest buffer contents. Thus in Figure 4, flow 2 is limited to a share of 1, while flows 1 and 4 split the remaining capacity in proportion to their bitrates.

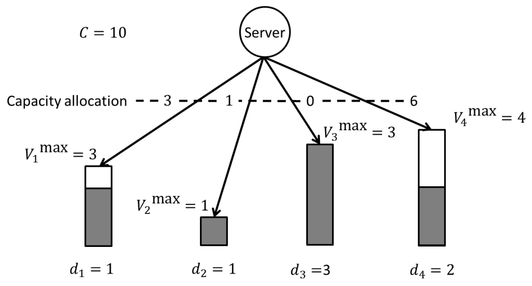


Fig. 4: In this example, all flows have content in their buffers, so there is no loss. In this case, all EPDF policies divide service capacity in proportion to bitrate among the flows with the smallest buffer contents, provided that these flows are not at their maximum buffer contents.

In the case where there is insufficient capacity to meet all progressive deadlines at t (this occurs if $\min_i\{V_i(t)\} = 0$ and $\sum_{i:\ V_i(t)=0} d_i > C$), the EPDF class is flexible in choosing how to drop content (see Figure 5). Ideally, the weights should indicate how we would like loss to be distributed among the flows; however, the EPDF class also includes policies that ignore weights. *The only rules for distributing loss are that only flows with empty buffers may receive service, and no flow i may receive content at rate $> d_i$ while another flow is losing content.* We now proceed to specify three policies in the EPDF class; these differ in how they distribute loss among flows with *empty buffers*.

A. Unweighted

Under the Unweighted policy, the server provides the same instantaneous loss fraction to each flow with an empty buffer, ignoring class weights. This is equivalent to allocating capacity among flows with empty buffers in proportion to their bitrates. That is, each flow i with $V_i(t) = 0$ receives service rate

$$\frac{d_i}{\sum_{i:\ V_i(t)=0} d_i} C.$$

This policy can be viewed as a fluid version of the traditional Earliest Deadline First policy.

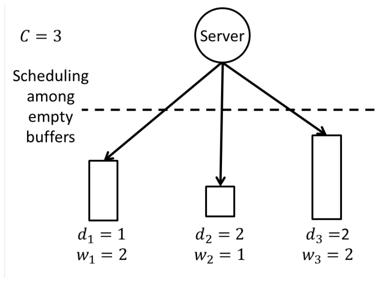


Fig. 5: The server has capacity $C = 3$, and must provide content to three flows with total bitrate 5, hence loss must occur. EPDF policies have many choices for how to distribute this loss among the 3 flows with empty buffers.

In the example in Figure 5, flows 2 and 3 have twice the bitrate of flow 1, and so they receive twice the service rate. Hence flow 1 receives service rate $\frac{3}{5}$, and flows 2 and 3 receive service rate $\frac{6}{5}$. The three flows' weights are not considered, and the instantaneous loss fraction for all three flows is $\frac{2}{5}$.

B. *Weighted Fractional Loss*

The Weighted Fractional Loss policy sets the instantaneous per-class loss fractions in proportion to class weights whenever loss is incurred. Let L_m denote the instantaneous loss rate for class m. L_m is chosen so that for all classes m and n,

$$\frac{w_m L_m}{\sum_{i \in m} d_i} = \frac{w_n L_n}{\sum_{j \in n} d_j},$$

and

$$\sum_m L_m = \sum_{i:\ V_i(t)=0} d_i - C.$$

Each class m receives total service rate $\sum_{i \in m:\ V_i(t)=0} d_i - L_m$, and this rate is allocated among the flows in the class in proportion to bitrate. Hence, all of the flows in the same class receive the same instantaneous loss fraction, and the instantaneous loss fraction incurred by flows in different classes is in proportion to their class weights.

In the example in Figure 5, a total amount of 2 loss must be incurred. The total bitrate of weight 1 flows is $d_2 = 2$, and the total bitrate of weight 2 flows is $d_1 + d_3 = 3$. Hence of the 2 loss, $\frac{8}{7}$ goes to the weight 1 class, and $\frac{6}{7}$ goes to the weight 2 class. This gives the weight 1 class an instantaneous loss fraction of $\frac{8/7}{2} = \frac{4}{7}$, which is twice $\frac{6/7}{3} = \frac{2}{7}$, the instantaneous loss fraction of the weight 2 class.

C. *Historical Weighted Fractional Loss*

The Weighted Fractional Loss policy above provides instantaneous loss fractions in proportion to class weights. However, this is not the same as providing total loss fractions in proportion to class weights. Recall that total loss fraction for a class is calculated as the total amount lost, divided by the total amount requested. Flows that incur no loss contribute to the denominator, but not the numerator, and so balancing the instantaneous loss fractions may be insufficient to balance the total loss fractions.

The Historical Weighted Fractional Loss policy overcomes this problem by keeping track of the total loss fraction that each class has incurred so far. At any moment when loss occurs, the server compares the total loss fractions incurred so far by each class, and allocates service so as to bring these total loss fractions towards the target ratio specified by the class weights.

In the example in Figure 5, which flows receive service depends on the total loss fraction each class has incurred thus far. Suppose that up to time t, class 1 has a total loss fraction of $\frac{1}{5}$, and class 2 has a total loss fraction of $\frac{1}{8}$. In order to achieve the 2:1 ratio specified by the class weights, class 1 needs to incur more loss, and so the entire service rate will be provided to the class 2 flows. Within the class, the service rate is allocated in proportion to bitrate, so flow 1 receives rate 1, flow 3 receives rate 2, and flow 2 receives rate 0.

V. Proof of Optimality of EPDF

In this section we prove the optimality of the class of EPDF policies with respect to the overall loss fraction, across all sample paths.

Let $D_i(s,t) = d_i(t-s)$ represent the bitrate, d_i, integrated over the time interval $[s,t]$. Let $X(s,t)$ be the total amount of content drained during $[s,t]$, including content from any flows that become either active or inactive during that time interval. Let $W_i(s,t)$ be the amount of content that is provided by the server to flow i during $[s,t]$. Let $S_{\text{act}}(t)$ be the set of active flows at time t, and $S_{\text{emp}}(t)$ the set of flows with empty buffers at time t, including any possible inactive flows. Let $S_{\text{emp\&act}}(t)$ be the set of active flows with empty buffers at time t, and $S_{\text{nonemp}}(t)$ the set of flows with non-empty buffers at time t, which must all be active (see Table II).

When referring to a policy-dependent variable, we will indicate the policy with a superscript. When referring to a subset of flows, we will indicate the subset with a subscript.

Theorem 1. *For all policies π, including those that might be able to anticipate future demands, and all policies $\xi \in \Pi^{EPDF}$ in the class of EPDF policies,*

$$X^{\pi}(0,t) \le X^{\xi}(0,t) \quad \forall\ t \ge 0. \tag{2}$$

The remainder of this section is devoted to proving Theorem 1, which is equivalent to proving that the overall loss *fraction* is minimized under all policies in EPDF. The proof is surprisingly non-trivial.

Consider a policy $\xi \in \Pi^{\text{EPDF}}$ from the class of EPDF policies and some arbitrary policy π. Suppose $X^{\pi}(0,t) \le X^{\xi}(0,t)$ for all $t < t_0$, and $X^{\pi}(0,t_0) = X^{\xi}(0,t_0)$. We will prove that just after t_0, we still have $X^{\pi}(0,t) \le X^{\xi}(0,t)$; that is, $X^{\pi}(t_0,t) \le X^{\xi}(t_0,t)$ for all $t \in [t_0,\ t_0 + \Delta_t]$ for some small $\Delta_t > 0$. We allow for the eventuality that a flow may just have arrived or expired at time t_0, but may exclude consideration of any arrivals or departures during $(t_0,\ t_0 + \Delta_t)$ by taking Δ_t to be sufficiently small.

We start with two simple yet crucial observations.

$D_i(s,t) = d_i(t-s)$	Maximum amount of content flow i can drain during time interval $[s,t]$
$S_{\text{act}}(t)$	Set of active flows at time t
$S_{\text{emp}}(t)$	Set of flows with empty buffers at time t
$S_{\text{emp\&act}}(t)$	Set of active flows with empty buffers at time t
$S_{\text{nonemp}}(t)$	Set of flows with non-empty buffers at time t
$W_i(s,t)$	Total amount of content provided to flow i during time interval $[s,t]$
$X(s,t)$	Total amount of content drained from all flows' buffers during time interval $[s,t]$

TABLE II: Notation used in the optimality proof

A. First, under any policy, $X_i(t_0,t)$, the amount of content drained for flow i during $[t_0,t]$ is bounded from above by $D_i(t_0,t)$, with equality when flow i's buffer is nonempty throughout that interval. Equality is guaranteed for all $t \in [t_0,\ t_0+\Delta_t]$ when Δ_t is sufficiently small and flow i's buffer is non-empty at time t_0.

B. Second, under any policy π, $X_{S^{\pi}_{\text{emp}}(t_0)}(t_0,t)$ is bounded above by $C(t-t_0)$ for all $t \in [t_0,\ t_0+\Delta_t]$. Under policy ξ, for sufficiently small Δ_t, the bound is tight because the flows with empty buffers receive the full capacity C and immediately drain all content received. Note that the aggregate bitrate of the flows with empty buffers must exceed C because otherwise these buffers would immediately start to fill under policy ξ.

Observe that the set of active flows is policy-independent and must remain the same for some length of time $\Delta_t > 0$ after t_0. The above two observations then yield the following two upper bounds for $X^{\pi}(t_0,t)$ for all $t \in [t_0,\ t_0+\Delta_t]$:

$$X^{\pi}(t_0,t) \leq \sum_{i\in S_{\text{act}}(t_0)} D_i(t_0,t), \tag{3}$$

and

$$X^{\pi}(t_0,t) \leq C(t-t_0) + \sum_{i\in S^{\pi}_{\text{nonemp}}(t_0)} D_i(t_0,t). \tag{4}$$

We now proceed to compare $X^{\xi}(t_0,t)$ with the upper bounds from (3) and (4), depending on whether $S^{\xi}_{\text{emp\&act}}(t_0) = \emptyset$ or not. We first consider the case $S^{\xi}_{\text{emp\&act}}(t_0) = \emptyset$, which means that the set $S^{\xi}_{\text{nonemp}}(t_0)$ of flows with nonempty buffers at time t_0 consists of the entire set $S_{\text{act}}(t_0)$ of active flows at time t_0. It then follows from Observation A that

$$X^{\xi}(t_0,t) = \sum_{i\in S_{\text{act}}(t_0)} D_i(t_0,t), \tag{5}$$

so that $X^{\xi}(t_0,t) \geq X^{\pi}(t_0,t)$ for all $t \in [t_0,\ t_0+\Delta_t]$ by (3).

We now turn to the case $S^{\xi}_{\text{emp\&act}}(t_0) \neq \emptyset$. Observations A and B then imply that, for all $t \in [t_0,\ t_0+\Delta_t]$,

$$X^{\xi}(t_0,t) = C(t-t_0) + \sum_{i\in S^{\xi}_{\text{nonemp}}(t_0)} D_i(t_0,t), \tag{6}$$

It remains to be shown that (6) $\geq$ (4) for all $t \in [t_0,\ t_0+\Delta_t]$, which is true provided $S^{\pi}_{\text{nonemp}}(t_0) \subseteq S^{\xi}_{\text{nonemp}}(t_0)$, or equivalently, $S^{\pi}_{\text{emp}}(t_0) \supseteq S^{\xi}_{\text{emp}}(t_0)$. Hence it suffices to prove Lemma 1, which completes the proof of Theorem 1.

Lemma 1. *If for all $t \leq t_0$, $X^{\xi}(0,t) \geq X^{\pi}(0,t)$ and $X^{\xi}(0,t_0) = X^{\pi}(0,t_0)$, then $S^{\pi}_{emp}(t_0) \supseteq S^{\xi}_{emp}(t_0)$.*

Proof of Lemma 1

The proof focuses on the set of flows $S^{\xi}_{\text{emp}}(t_0)$ with empty buffers under policy ξ at time t_0, and shows that all these flows must have empty buffers under policy π as well, i.e., $S^{\pi}_{\text{emp}}(t_0) \supseteq S^{\xi}_{\text{emp}}(t_0)$.

Let $s_0 \leq t_0$ be the last time up to t_0 at which any of the flows in $S^{\xi}_{\text{nonemp}}(t_0)$ received service under policy ξ, or if no such flows exist, the last time at which policy ξ used less than the full service capacity C.

In order to prove that all the flows in $S^{\xi}_{\text{emp}}(t_0)$ must have empty buffers at t_0 under policy π, we consider the following three quantities for this set of flows:

1) $\sum_{i\in S^{\xi}_{\text{emp}}(t_0)} d_i V^{\pi}_i(s_0)$, the total buffer contents (in bytes) at s_0 under policy π
2) $W^{\pi}_{S^{\xi}_{\text{emp}}(t_0)}(s_0,t_0)$, the total amount of service provided during $[s_0,t_0]$ under policy π
3) $X^{\pi}_{S^{\xi}_{\text{emp}}(t_0)}(s_0,t_0)$, the total amount of content drained during $[s_0,t_0]$ under policy π.

The total buffer contents at t_0 under policy π of the flows in $S^{\xi}_{\text{emp}}(t_0)$ may then be expressed as

$$\begin{aligned}\sum_{i\in S^{\xi}_{\text{emp}}(t_0)} d_i V^{\pi}_i(t_0) \;&=\; \sum_{i\in S^{\xi}_{\text{emp}}(t_0)} d_i V^{\pi}_i(s_0) \\ &+\; W^{\pi}_{S^{\xi}_{\text{emp}}(t_0)}(s_0,t_0) \\ &-\; X^{\pi}_{S^{\xi}_{\text{emp}}(t_0)}(s_0,t_0).\end{aligned} \tag{7}$$

A similar expression, with the superscript π replaced by ξ, holds for the total buffer contents at t_0 of the flows in $S^{\xi}_{\text{emp}}(t_0)$ under policy ξ.

We now proceed to compare each of the above three quantities with the corresponding ones under policy ξ.

1) By definition of s_0, some flow in $S^{\xi}_{\text{nonemp}}(t_0)$, if any such flow exists, must have received service at s_0 under policy ξ, or policy ξ must have used less than the full service capacity C at s_0. Furthermore, under policy ξ, all of the flows in $S^{\xi}_{\text{emp}}(t_0)$ that started before s_0 must have smaller buffer contents at s_0 than any of the flows in $S^{\xi}_{\text{nonemp}}(t_0)$. Indeed, by definition of s_0, no flow in $S^{\xi}_{\text{nonemp}}(t_0)$ received any service during $[s_0,t_0]$, so any content in those buffers at t_0 must already have been there at s_0. If some flow in $S^{\xi}_{\text{emp}}(t_0)$ that started before s_0 had a larger buffer content at s_0, some of that content should remain in its buffer at t_0, which would contradict the definition of $S^{\xi}_{\text{emp}}(t_0)$.

We deduce that all the flows in $S^{\xi}_{\text{emp}}(t_0)$ that started before s_0 must have been at their maximum buffer content at s_0 under policy ξ. Thus the total buffer contents of these flows at s_0 under policy ξ is $\sum_{i \in S^{\xi}_{\text{emp}}(t_0)} d_i V_i^{\max}(s_0)$, which is an upper bound for any policy. Hence,

$$\begin{aligned} \sum_{i \in S^{\xi}_{\text{emp}}(t_0)} d_i V_i^{\xi}(s_0) &= \sum_{i \in S^{\xi}_{\text{emp}}(t_0)} d_i V_i^{\max}(s_0) \\ &\geq \sum_{i \in S^{\xi}_{\text{emp}}(t_0)} d_i V_i^{\pi}(s_0). \end{aligned} \tag{8}$$

2) By definition of s_0, none of the flows in $S^{\xi}_{\text{nonemp}}(t_0)$ received any service in $[s_0, t_0]$ under policy ξ, and the full service capacity C was used throughout this entire interval. Hence the total amount of service received by the flows in $S^{\xi}_{\text{emp}}(t_0)$ during this interval is $W^{\xi}_{S^{\xi}_{\text{emp}}}(s_0, t_0) = C(t_0 - s_0)$, which is an upper bound for any policy. Hence,

$$W^{\xi}_{S^{\xi}_{\text{emp}}(t_0)}(s_0, t_0) = C(t_0 - s_0) \geq W^{\pi}_{S^{\xi}_{\text{emp}}(t_0)}(s_0, t_0). \tag{9}$$

3) By definition of s_0, none of the flows in $S^{\xi}_{\text{nonemp}}(t_0)$ received any service in $[s_0, t_0]$ under policy ξ. Since these flows' buffers are non-empty at t_0 by definition, they must have been non-empty throughout $[s_0, t_0]$. Thus the total amount of content drained for these flows during $[s_0, t_0]$ under policy ξ must have been $X^{\xi}_{S^{\xi}_{\text{nonemp}}(t_0)}(s_0, t_0) = \sum_{i \in S^{\xi}_{\text{nonemp}}(t_0)} D_i(s_0, t_0)$, which is an upper bound for any policy. Hence,

$$\begin{aligned} X^{\xi}_{S^{\xi}_{\text{nonemp}}(t_0)}(s_0, t_0) &= \sum_{i \in S^{\xi}_{\text{nonemp}}(t_0)} D_i(s_0, t_0) \\ &\geq X^{\pi}_{S^{\xi}_{\text{nonemp}}(t_0)}(s_0, t_0). \end{aligned} \tag{10}$$

We further know that $X^{\xi}(0, s_0) \geq X^{\pi}(0, s_0)$ and $X^{\xi}(0, t_0) = X^{\pi}(0, t_0)$, which implies

$$X^{\xi}(s_0, t_0) \leq X^{\pi}(s_0, t_0). \tag{11}$$

Combining (11) and (10), we find

$$X^{\xi}_{S^{\xi}_{\text{emp}}(t_0)}(s_0, t_0) \leq X^{\pi}_{S^{\xi}_{\text{emp}}(t_0)}(s_0, t_0). \tag{12}$$

Substituting (8), (9), and (12) into (7) and the corresponding expression for policy ξ, we obtain

$$\sum_{i \in S^{\xi}_{\text{emp}}(t_0)} d_i V_i^{\pi}(t_0) \leq \sum_{i \in S^{\xi}_{\text{emp}}(t_0)} d_i V_i^{\xi}(t_0).$$

By definition of $S^{\xi}_{\text{emp}}(t_0)$, however, the latter term is zero, which then implies that $V_i^{\pi}(t_0) = 0$ for all flows $i \in S^{\xi}_{\text{emp}}(t_0)$, implying $S^{\xi}_{\text{emp}}(t_0) \subseteq S^{\pi}_{\text{emp}}(t_0)$. $\square$

Remark The statement of Lemma 1 does *not* hold if the condition $X^{\xi}(0, t_0) = X^{\pi}(0, t_0)$ is omitted. That is, it is not necessarily the case that $S^{\pi}_{\text{emp}}(t) \supseteq S^{\xi}_{\text{emp}}(t)$ for all t. It is possible that under policy ξ, all flows' buffers become empty, but that policy π chooses to fill one flow's buffer while leaving the others' empty. This strategy may cause there to be fewer empty buffers under policy π, but in order to reach this scenario, more content must have been lost under policy π than under policy ξ. Figure 6 gives an example. Such counterintuitive cases make Theorem 1 non-trivial.

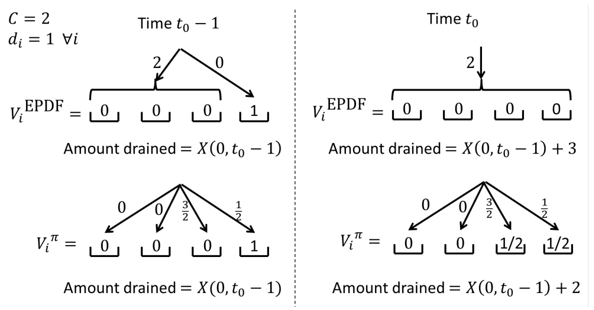


Fig. 6: If we do not have $X^{\xi}(0, t_0) = X^{\pi}(0, t_0)$, then it is possible that $S^{\pi}_{\text{emp}}(t_0) \subseteq S^{\xi}_{\text{emp}}(t_0)$. In this example, ξ and π have provided service in exactly the same way up to time $t_0 - 1$, so the buffer contents of all flows and the amount drained so far are identical (all flows have the same bitrate, $d = 1$). At $t_0 - 1$, ξ continues to split the capacity $C = 2$ among the three flows with empty buffers, while π serves the third flow at rate $3/2$ and the fourth flow at rate $1/2$. By time t_0, there are fewer empty buffers under π than under ξ.

VI. Evaluation

In this section, we evaluate the performance of the three EPDF policies introduced in Section IV (Unweighted, Weighted Fractional Loss, and Historical Weighted Fractional Loss), as well as three versions of Discriminatory Processor Sharing (DPS). Under DPS, each flow i is associated with a value ϕ_i, and at all times t, receives service rate $\frac{\phi_i}{\sum_j \phi_j} C$, where the sum is taken over all active-incomplete flows at time t. We consider three versions of DPS that use different values for ϕ. Under the DPS - bitrate policy, $\phi_i = d_i$; under DPS - weight, $\phi_i = w_i$; and under DPS - weight and bitrate, $\phi_i = d_i w_i$. Note that DPS is aware of buffer capacity limits, and if a flow i is at its buffer limit, DPS will not provide a service rate that exceeds d_i.

In Section VI-A, we assess the ability of the policies to achieve the goals of 1) minimizing the overall loss fraction, and 2) distributing total per-class loss fraction in proportion to class weights. In Section VI-B, we study the sensitivity of the policies to various system parameters.

A. Performance

We conduct our simulations using both artificial workloads and a trace collected from a video server.

For the artificial (toy) workload, we assume that flows arrive according to a Poisson process with rate λ and that flow durations are distributed Exponentially with mean $\frac{1}{\mu} = 4$. We set the server capacity, C, to 6. We assume that clients drain content either at bitrate $d_i = 1$ or $d_i = 2$, and that clients choose each bitrate with probability 1/2. We use two weight classes, with the high weight class having twice the weight of the low weight class. This means that our goal is for low weight clients to incur twice the loss fraction of high weight clients. Clients belong to each class with probability 1/2, and bitrate and weight are independent. We set the system load, $\rho = \frac{\lambda \bar{d}}{C\mu}$, to 0.5, where $\bar{d}$ is the average requested bitrate. Finally, we vary the buffer capacity limit, using limits of 1, 2, 3, and 4, as well as infinite buffer capacities. We show only the results for buffer limits of 2 (see Figure 7(a)).

We also conduct a trace-driven simulation. The trace consists of all requests made to a Windows Media Server using the Microsoft Media Server (MMS) protocol at our university over the last year. The trace contains 14,550 distinct arrivals. The arrival process is extremely bursty, with the average arrival rate for a day ranging from 0 to 0.0025 arrivals/second, and a squared coefficient of variation of the interarrival times of over 29. The mean video duration is 548 seconds, and the requested bitrates span a factor of 5. In our trace-driven simulation, the server capacity is $C = 50$ and the average system load is $\rho = 0.5$. Results are shown in Figure 7(b).

As shown in Figures 7(a) and (b), all of the EPDF policies minimize the total loss fraction, while the DPS policies do not. The DPS - weight and DPS - weight and bitrate policies result in different loss fractions for the low and high weight classes, but the ratio is much higher than the target value of 2. Low weight jobs suffer disproportionately under DPS.

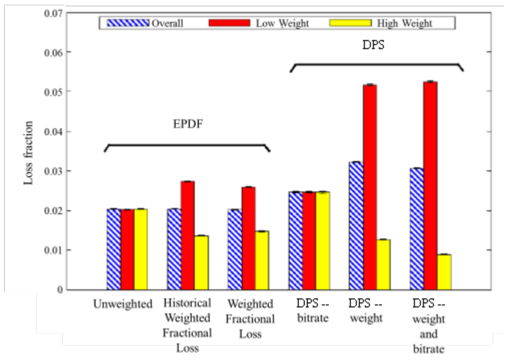
The Historical Weighted Fractional Loss EPDF policy exactly achieves the target loss ratio. However, it may be impractical to implement due to the need to maintain a complete history of the total loss fraction for each class. The Weighted Fractional Loss policy approximates the performance of the Historical Weighted Fractional Loss policy, while eliminating the need for extensive state information. Our results demonstrate that the Weighted Fractional Loss policy nearly achieves the target loss ratio. It does not obtain the exact ratio because it proportionally balances instantaneous loss fraction rather than total loss fraction (see Section IV-C).

B. Sensitivity

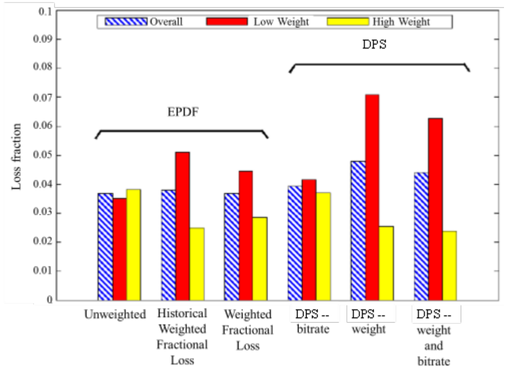
We also conduct experiments to assess the sensitivity of the EPDF and DPS policies to various system parameters, including 1) the overall system load, ρ, 2) the relative load of low weight jobs, $\frac{\rho_{\text{low wgt}}}{\rho}$, and 3) the server capacity, C. We fix the parameters for each experiment as described in Section VI-A, varying only the parameter under investigation.

As expected, the overall loss fraction for all policies increases as either load, ρ, increases, or capacity, C, decreases. In the remainder of this section, we only consider the ability of policies to meet the target ratio of per-class loss fraction.

Figure 8 shows the results of the three sensitivity experiments under the artificial workload. The DPS - weight, and DPS - weight and bitrate policies are highly sensitive to all three parameters. This indicates that not only are the DPS policies unable to achieve the target loss fraction ratio, but the ratio that they do provide is impossible to predict. On the other hand, the EPDF policies are highly insensitive to all the system parameters with respect to meeting the target loss ratio.



(a) Artificial workload



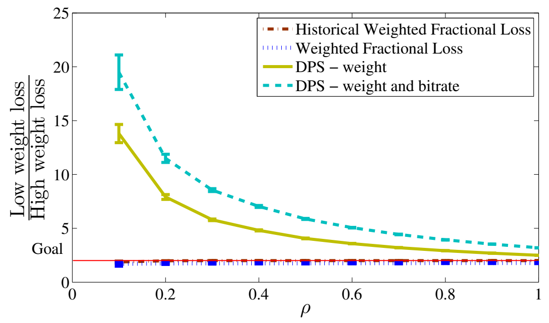
(b) Trace

Fig. 7: Results from the **(a)** artificial workload, and **(b)** trace. Overall loss fraction (blue striped), total loss fraction for low weight jobs (red/dark), and total loss fraction for high weight jobs (yellow/light) for each of the six policies. The EPDF policies all minimize the overall loss fraction. The Historical Weighted Fractional Loss policy and the Weighted Fractional Loss policy both achieve exact or near-exact target ratios for loss fractions. The DPS policies neither minimize the overall loss fraction, nor achieve the target ratio between classes.

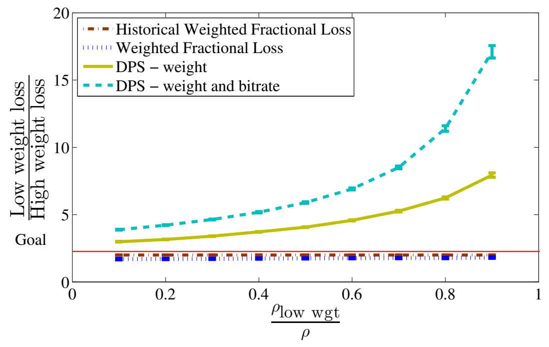
VII. Conclusions and Future Work

In this paper, we study server-side scheduling policies in which a single server must provide content to clients, subject to a capacity limit at the server, where our server drops content that cannot be delivered on time. Rather than "dropping" content, one can equivalently view the server as providing content at a reduced rate.

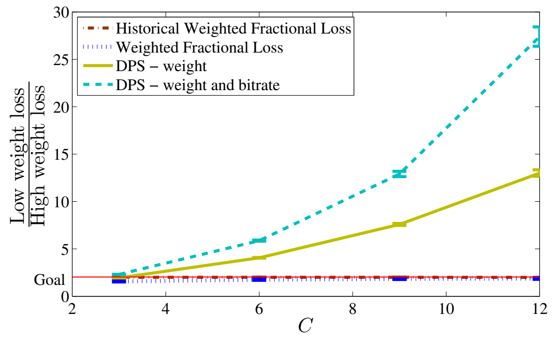
We consider two equally important objectives: 1) minimizing the total fraction of content that is dropped, and 2) distributing loss fraction in proportion to class weights. We introduce the EPDF class of policies, which we prove is optimal with respect to the first objective. Furthermore, we demonstrate that two EPDF policies, Historical Weighted Fractional Loss and Weighted Fractional Loss, can achieve the second objective. Our simulation results demonstrate that the performance of EPDF policies with respect to the second objective is largely insensitive to system parameters. In contrast, the commonly-used DPS does not minimize the total fraction



(a) Sensitivity to system load, ρ ($C = 6$, $\rho_{\text{low wgt}} = \rho_{\text{high wgt}}$)



(b) Sensitivity to relative load of low weight jobs, $\frac{\rho_{\text{low wgt}}}{\rho}$ ($C = 6$, $\rho = 0.5$)



(c) Sensitivity to server capacity, C ($\rho = 0.5$, $\rho_{\text{low wgt}} = \rho_{\text{high wgt}}$)

Fig. 8: Ratio of loss fraction for low weight jobs to that for high weight jobs as a function of increasing **(a)** system load, ρ, **(b)** relative load of low weight jobs, $\frac{\rho_{\text{low wgt}}}{\rho}$, and **(c)** server capacity, C. The DPS policies are highly sensitive to all three parameters, while the EPDF policies are insensitive to all three parameters, and consistently meet the target ratio of 2.

of content lost, cannot achieve the target loss ratio between weight classes, and is highly sensitive to system parameters.

While the policies that we consider in this paper are described in a somewhat idealized fluid framework, the absolute dominance of the EPDF class with respect to (i) overall loss fraction, (ii) the ability to provide proportional loss fractions, and (iii) insensitivity properties indicates that EPDF should serve as a good heuristic in less idealized settings.

The specific EPDF policies that we discuss are designed to achieve a particular objective: to balance the loss fraction between different classes of users. However, EPDF is a very broad class of policies, and EPDF policies can be specified for a wide variety of second objectives, depending on the particular goals of the system designer. For example, one may wish to minimize the number of users affected by loss, minimize the number of instances of loss per user, or any number of other goals. The strength of the EPDF class is that regardless of the second objective, it remains optimal with respect to minimizing overall content lost.

One direction of future work is to define a hybrid goal which combines our two goals into a single metric, e.g. the weighted overall loss fraction (WOLF), defined as the total weighted amount of content lost, divided by the total weighted amount of content requested. It would be interesting to identify which policies are optimal for the WOLF metric. Another future direction is to extend the results to time-varying bitrates $d_i(t)$.

References

[1] V. Gamini Abhaya, Z. Tari, P. Zeephongsekul, and A. Zomaya. Performance analysis of EDF scheduling in a multi-priority preemptive M/G/1 queue. *IEEE Transactions on Parallel and Distributed Systems*, July 2013.

[2] J.C.R. Bennett and H. Zhang. WF^{2}Q: worst-case fair weighted fair queueing. In *IEEE INFOCOM Proceedings*, pages 120–128, 1996.

[3] An architecture for differentiated services. IETF RFC 2475, 1998.

[4] S. Bodamer. A new scheduling mechanism to provide relative differentiation for real-time IP traffic. In *Global Telecommunications Conference*, pages 646–650, 2000.

[5] Integrated services in the internet architecture: an overview. IETF RFC 1633, 1994.

[6] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM Computer Communications Review*, 19(4):1–12, September 1989.

[7] F. Dobrian, A. Awan, D. Joseph, A. Ganjam, J. Zhan, V. Sekar, I. Stoica, and H. Zhang. Understanding the impact of video quality on user engagement. In *SIGCOMM*, pages 362–373, August 2011.

[8] G. Fayolle, I. Mitrani, and R. Iasnogorodski. Sharing a processor among many job classes. *Journal of the ACM*, 27(3):519–532, July 1980.

[9] S.J. Golestani. A self-clocked fair queueing scheme for broadband applications. In *IEEE INFOCOM Proceedings*, pages 636–646, 1994.

[10] M. Kargahi and A. Movaghar. A method for performance analysis of earliest-deadline-first scheduling policy. *Journal of Supercomputing*, 37:197–222, 2006.

[11] L. Kruk, J. Lehoczky, K. Ramanan, and S. Shreve. Heavy traffic analysis for EDF queues with reneging. *Annals of Applied Probability*, 21(2):484–545, 2011.

[12] J.P. Lehoczky. Real-time queueing theory. In *17th IEEE Real-Time Systems Symposium*, pages 186–195, March 1996.

[13] C.L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[14] P. Moyal. On queues with impatience: stability, and the optimality of earliest deadline first. *Queueing Systems*, 75:211–242, 2013.

[15] A.K. Parekh and R.G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The multiple-node case. *IEEE/ACM Transactions on Networking*, 2(2):137–150, 1994.

[16] T.Y. Tsai, Y.L. Chung, and Z. Tsai. Introduction to packet scheduling algorithms for communication networks. In Jun Peng, editor, *Communications and Networking*. Sciyo, Rijeka, Croatia, 2010.

[17] Z. Yuhua and Z. Chaochen. A formal proof of the deadline driven scheduler. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, 863:756–775, 1994.

[18] H. Zhang. Service disciplines for guaranteed performance service in packet-switching networks. In *Proceedings of the IEEE*, pages 1374–1396, October 1995.