Earliest Eligible Virtual Deadline First : A Flexible and Accurate Mechanism for Proportional Share Resource Allocation[*]

Ion Stoica, Hussein Abdel-Wahab
Department of Computer Science, Old Dominion University
Norfolk, Virginia, 23529-0162
{stoica, wahab}@cs.odu.edu

TR-95-22

Abstract

We propose and analyze a new proportional share allocation algorithm for time shared resources. We assume that the resource is allocated in time quanta of size q. To each client, we associate a weight which determines the relative share from the resource that the client should receive. We define the notion of fairness in the context of an idealized system in which the resource is assumed to be granted in arbitrarily small intervals of time. Mainly, we show that in steady conditions our algorithm guarantees that the difference between the service time that a client should receive in the idealized system and the service time it actually receives in the real system is bounded by the size q of a time quantum. The algorithm provides support for dynamic operations, such as a client joining or leaving the competition (for the resource), and changing a client's weight. By using an efficient augmented binary search tree data structure we implement these operations in $O(\log n)$, where n represents the number of clients competing for the resource.

[*]Revised January 26, 1996.

1 Introduction

One of the most challenging problems in modern operating systems is to design flexible and accurate algorithms to allocate resources among competing clients. This issue has become more important with the emergence of new types of real-time applications such as multimedia which have well defined time constraints. In order to meet these constraints the underlying operating system should allocate resources in a predictable and responsive way. In addition, a general-purpose operating system should seamlessly integrate these new types of applications with conventional interactive and batch applications.

Many schedulers have tried, and in part have succeeded, to address these requirements. Generally, these schedulers fall in two categories: *proportional share* [2, 26, 28, 29, 30], and *real-time*[1] based schedulers [20, 21, 22]. In *proportional share* algorithms each client has associated a weight which determines the share of the resource that the client should receive. The scheduler tries to allocate the resource among competing clients in proportion to their share. For example, consider two clients with weights 1 and 3, respectively that compete for the same resource. Then the first client should receive 25%, while the second client should receive 75% of the resource. *Real-time* schedulers are based on an event-driven model in which a client is characterized by a set of events arriving in a certain pattern (usually periodic), and by a predicted service time and a deadline associated to each event. By imposing strict admission policies, these schedulers guarantee that all events are processed before their deadlines. While in general the proportional share schedulers tend to be more flexible and to ensure a graceful degradation in overload situations, real-time based schedulers tend to offer better guarantees for applications with timeliness constraints, such as multimedia.

Although real-time based schedulers provide better support for multimedia, they cannot be easily extended to support batch applications. The main reason is that while multimedia and interactive applications fit the event-driven model implicitly assumed by these schedulers, batch applications do not. For example, a video application can be modeled as a periodic process, where an event represents the arrival of a frame, its service time is the time required to process and display the frame[2], and its deadline is the time at which the next frame should arrive. Similarly, an interactive application (such as an editor) can be modeled as an aperiodic process, where an event is generated as a result of the user pressing a key, the service time is the time required to process and display the corresponding character, and the deadline is given by the largest acceptable delay between the moment when the key is pressed and the moment when the character is displayed. In contrast to both multimedia and interactive applications, batch applications are usually characterized by only one parameter, i.e., the requested service time. Moreover, in many situations it is difficult to accurately determine this service time. The main reason is that, even for the same application, the service time may have large variations, and these variations are hard to predict. For example, the service time required by a compiler may vary significantly even for the same program, depending on how many modules are recompiled. For these reasons many general purpose schedulers that use real-time schemes for scheduling continuous media also employ conventional algorithms (e.g., round-robin) for scheduling batch applications. Another drawback of real-time based schedulers is that, in order to satisfy strong time constraints, they impose strict admission policies which make them fairly restrictive. Thus, a user can be in the situation in which he cannot run a new application, although he might be willing to accept a degradation in performance of other applications in order to accommodate

[1]Usually, these schedulers are based on two well-known real-time algorithms proposed and analyzed by Liu and Layland in [18]: rate monotonic and early deadline first.

[2]Depending on the desired quality this can be either the worst case, or a statistical average service time.

the new one. Finally, in a highly dynamic environment these schedulers do not provide enough flexibility; for example, when an application terminates it is difficult to distribute rapidly its share among the other applications that are still competing for the resource.

In this paper we propose a new scheduler (called Earliest Eligible Virtual Deadline First – EEVDF) which, while retaining all the advantages of the proportional share schedulers, provides strong timeliness guarantees for the service time received by a client. In this way, our algorithm provides a unified approach for scheduling continuous media, interactive and batch applications. In part, the algorithm builds on ideas found in previous network fair-queueing algorithms [23, 31], and general-purpose proportional share allocation algorithms [29, 30]. As in [23, 31], and similar to [29, 30] we use the notion of *virtual time* to track the work progress in an ideal fluid-flow based system. To each client we associate a weight which determines the relative share of the resource that the client should receive. The client requirements are uniformly translated in a sequence of requests for the resource. These requests are either issued explicitly by the clients, or by the scheduler itself on behalf of the clients. In this way batch activities are treated uniformly with multimedia and interactive activities. Based on the client share and on the service time that the client has already received, the scheduler associates to each client's request a *virtual eligible time* and a *virtual deadline* which are the corresponding starting and finishing times of servicing the request in the fluid-flow model. A request is said to be *eligible* if its virtual eligible time is less than or equal to the current virtual time. The algorithm simply allocates a new time quantum to the client that has the *eligible* request with the earliest virtual deadline. We note that while the concept of virtual deadline is also employed by other proportional-share algorithms [23, 31, 29, 30], the concept of eligible time is a unique feature of our algorithm (which, as we will show, plays a decisive role in improving the allocation accuracy).

While EEVDF implements a flexible and accurate low-level mechanism for proportional share resource allocations, higher level resource abstractions are needed to specify the applications' requirements. We note here that many of the existing abstractions, such as *tickets* and *currencies* (developed by Waldspurger and Weihl [28, 30]), and *processor capacity reserves* (proposed by Mercer, Savage and Tokuda [20]) are easily supported by EEVDF. For example, in *lottery scheduling* the number of tickets held by a client could be directly translated into the weight associated to that client in the EEVDF algorithm. By efficiently implementing dynamic operations, the EEVDF provides direct support for higher level mechanisms such as tickets transfer and inflation employed by lottery scheduling [28, 30]. Throughout this paper we will not discuss further other higher level resource abstractions; instead, we will implicitly assume that one of the existing abstractions is implemented on top of EEVDF. Therefore, in this paper we will not address problems that are usually handled by this higher level (e.g., priority inversion).

This paper is organized as follows. The next section discusses our assumptions, and Section 3 presents the basic EEVDF algorithm. Section 4 discusses the concept of fairness in dynamic systems, while Section 5 presents three strategies for implementing the EEVDF algorithm. In Section 6 we give the fairness analysis of the algorithm. Finally, in Section 7 we give an overview of the related work, and Section 8 concludes the paper.

2 Assumptions

We consider a set of clients that compete for a time shared resource (e.g., processor, communication bandwidth). We assume that the resource is allocated in time quanta of size at most q. At the beginning of each time quantum a client is selected to use the resource. Once the client acquires the resource, it may use it either for the entire time quantum, or it may release it before the time quantum expires. Although simple, this model captures the basic mechanisms traditionally used for sharing common resources, such as processor and communication bandwidth. For example, in many preemptive operating systems (e.g., UNIX, Windows-NT), the CPU scheduler allocates the processing time among competing processes in the same fashion: a process uses the CPU until its time quantum expires or another process with a higher priority becomes active, or it may voluntarily release the CPU while it is waiting for an event to occur (e.g., an I/O operation to complete). As another example, consider a communication switch that multiplexes a set of incoming sessions on a packet-by-packet basis. Since usually the transmission of a packet cannot be preempted, we take a time quantum to be the time required to send a packet on the output link. Thus, in this case, the size q of a time quantum represents the time required to send a packet of maximum length.

Further, we associate to each client a *weight* that determines the relative *share* of the resource that it should receive. The share is computed as the ratio between the client's weight and the total sum over the weights of all active clients. A client is said to be *active* while it is competing for the resource, and *passive* otherwise. More formally, let w_i denote the weight associated to client i, and let $\mathcal{A}(t)$ be the set of all clients active at time t. Then the share of client i at time t, denoted $f_i(t)$, is defined as:

$$f_i(t) = \frac{w_i}{\sum_{j \in \mathcal{A}(t)} w_j}. \tag{1}$$

Ideally, if the client share remains constant during a time interval $[t, t + \Delta t]$, then client i is entitled to use the resource for $f_i(t)\Delta t$ time units. In general, when the client share varies over time, the service time that client i should receive in a *perfect fair* system while being active during a time interval $[t_0, t_1]$ is

$$S_i(t_0, t_1) = \int_{t_0}^{t_1} f_i(\tau) d\tau \tag{2}$$

time units. The above equation corresponds to an ideal fluid-flow system in which the resource can be granted in arbitrarily small intervals of time. In our case, this is equivalent to the situation in which the size of a time quantum approaches zero ($q \rightarrow 0$).[3] Unfortunately, in many practical situations, time quanta cannot be taken arbitrarily small. One of the reasons is the overhead introduced by the scheduling algorithm and the overhead in switching from one client to another: taking time quanta of the same order of magnitude as these overheads could drastically reduce the resource utilization. For example, it would be unacceptable for a CPU to spend more time in scheduling a new process, and context switching between the processes, than doing useful computation. Another reason is that some operations cannot be interrupted, i.e., once started they must complete in the same time quanta. For example, once a communication switch begins to send a packet for one session, it cannot serve any other session until the entire packet is sent.

Due to quantization, in a system in which the resource is allocated in discrete time quanta (as it is in ours), it is not possible for a client to always receive exactly the service time it is entitled to. The difference between the service time that a client should receive at a time t, and the service time it *actually* receives is called service time *lag*. More precisely, let t_0^i be a time at which client i becomes active, and let $s_i(t_0^i, t)$ be the service time the client receives in the interval $[t_0^i, t]$ (here, we assume that client i is active in the entire interval $[t_0^i, t]$). Then the service time lag of client i at time t is

[3] A similar model was used by Demers *et al* in studying fair-queuing algorithms in communication networks [9].

$$lag_i(t) = S_i(t_0^i, t) - s_i(t_0^i, t). \tag{3}$$

Since the service time lag determines both the throughput accuracy and the system predictability, we use it as the main parameter in characterizing our proportional resource allocation algorithm.

3 The EEVDF Algorithm

In order to obtain access to the resource, a client must issue a *request* which specifies the duration of the service time it needs. Once a client's request is fulfilled, it may either issue a new request, or otherwise the client becomes passive. Notice that we can alternately define a client to be active while it has a pending request, and to be passive otherwise. For uniformity, throughout this paper we assume that the client is the sole initiator of the requests. However, in practice this is not necessarily true. For example, in the processor case, the scheduler itself could be the one to issue the requests on behalf of the client. In this case, the request duration (length) is either specified by the client, or otherwise the scheduler assumes a "default" duration. This allows us to treat all continuous media, interactive, and batch activities in a consistent way.

For flexibility we allow requests to have any duration. If the duration of the request is larger than a time quantum, then the service time might not be allocated continuously (i.e., when a time quantum expires, the client is preempted and the next time quantum can be allocated to another client). When a client requests the resource for less than one time quantum, the scheduler simply preempts the client once its time expires. Notice that a client may request the same amount of service time by generating either fewer longer requests, or many shorter ones. For example, a client may ask for 1 min computation time, either by issuing 60 requests with a duration of 1 sec each, or by issuing 600 requests with a duration of 100 msec each. As we will show in Section 6, shorter requests guarantee better allocation accuracy, while longer requests decrease the system overhead. In this way a client could trade between the allocation accuracy and the scheduling overhead.

To clarify the ideas we briefly point out the similarities and differences between our model and the well-known problem of scheduling periodic tasks in a real-time system [18]. A periodic task is characterized by a fixed interval of time between two consecutive events, called period and denoted by T, and by the maximum service time r required to process an event. Whenever an event occurs, the task simply requests r service time units for processing that event. A central requirement in real-time systems is that the current event should be processed before the next event occurs. Notice that this requirement guarantees that the task receives as much as r service time units during every period T. Consequently, in this case the task receives a share $f = \frac{r}{T}$ of the resource. Thus, by giving the share f, the requested service time r and the time t at which the event occurs, the deadline of the corresponding request can be expressed as $t + \frac{r}{f}$. Similarly, in our model, by giving the time t at which a request is made and its duration r, we obtain the time d before which the client should receive the requested service time in an ideal system by solving the equation $r = S(t, d)$. If the share f of the client does not change in the interval $[t, d)$, then from Eq. (2) follows that $S(t, d) = f \times (d - t)$, and further we obtain $d = t + \frac{r}{f}$, which is identical to the expression of the request's deadline in the case of scheduling periodic tasks. As a major difference between our model and a real-time system, we note that while in a real-time system a request is assumed to be generated as a result of an external event (e.g., a packet arrival, a time-out), in our model a request is either generated as result of an external event (when the client enters the competition), or as a result of an *internal event* (when the client generates a new request after the current one has been fulfilled). In this way our model provides integrated support for both event driven applications, such as continuous media and interactive applications, and for conventional batch-applications.

By combining Eq. (1) and (2) we can express the service time that an active client i should receive in the interval $[t_1, t_2)$ as

$$S_i(t_1, t_2) = w_i \int_{t_1}^{t_2} \frac{1}{\sum_{j \in \mathcal{A}(\tau)} w_j} d\tau. \tag{4}$$

Similarly to [31] and [23] we define the system virtual time as

$$V(t) = \int_{0}^{t} \frac{1}{\sum_{j \in \mathcal{A}(\tau)} w_j} d\tau. \tag{5}$$

We note that the virtual time increases at a rate inverse proportional to the sum of the weights of all active clients. Notice that when the competition increases the virtual time slows down, while when the competition decreases it accelerates. Intuitively, the flow of the virtual time changes to "accommodate" all active clients in one virtual time unit. That is, the size of a virtual time unit is modified such that in the corresponding fluid-flow system each active client i receives w_i real-time units during one virtual time unit. For example, consider two clients with weights $w_1 = 2$ and $w_2 = 3$. Then the rate at which the virtual time increases is $\frac{1}{w_1+w_2} = 0.2$, and therefore a virtual time unit equals five real-time units. Thus, in each virtual time unit the two clients should receive $w_1 = 2$, and $w_2 = 3$ time units. Next, from Eq. (4) and (5) it follows that

$$S_i(t_1, t_2) = w_i(V(t_2) - V(t_1)). \tag{6}$$

To better interpret the above equation it is useful to consider a much simpler model in which the number of active clients is constant and the sum of their weights is one ($\sum_{i \in \mathcal{A}} w_i = 1$), i.e., the share of a client i is $f_i = w_i$. Then the service time that client i should receive during an interval $[t_1, t_2)$ is simply $S_i(t_1, t_2) = w_i(t_2 - t_1)$. Next, notice that by replacing the real times t_1 and t_2 with the corresponding virtual times $V(t_1)$ and $V(t_2)$ we arrive at Eq. (6). Thus, Eq. (6) can be viewed as a direct generalization of computing the service time $S_i(t_1, t_2)$ in a dynamic system.

The basic idea behind our algorithm is simple. We associate to each request an *eligible time* e and a *deadline* d. A request of an active client becomes eligible at time e when the service time that the client should receive in the corresponding fluid-flow system *equals* the service time that the client has already received (in the real system) before issuing the current request. Let t_0^i be the time at which client i becomes active, and let t be the time at which it initiates a new request. Then the eligible time e of the new request is chosen such that $S_i(t_0^i, e) = s_i(t_0^i, t)$. Notice that if at time t client i has received more service time than it was supposed to receive (i.e., its lag is negative at time t), then the client should wait until time e before the new request becomes eligible. In this way a client that has received more service time than its share is "slowed down", while giving the other active clients the opportunity to "catch up". On the other hand, if at time t client i has received less service time than it was supposed to receive (i.e., its lag is positive), then we have $e < t$, and therefore the new request is immediately eligible at time t. Since in either case the service time that the client has received at time e is no greater than the service time that the client should have received at time e it follows that the client's lag at time e ($lag_i(e)$) is always positive. By using Eq. (6) we can express the virtual eligible time $V(e)$ as

$$V(e) = V(t_0^i) + \frac{s_i(t_0^i, t)}{w_i}. \tag{7}$$

Next, the *deadline* of the request is chosen such that the service time that the client should receive between the eligible time e and the deadline d *equals* the service time of the new request, i.e., $S_i(e, d) = r$, where r represents the length of the new request. Further, by using again Eq. (6), we derive the virtual deadline $V(d)$ as

$$V(d) = V(e) + \frac{r}{w_i}. \tag{8}$$

Notice that although Eq. (7) and (8) give us the virtual eligible time $V(e)$ and the virtual deadline $V(d)$, they do not *necessarily* give us the values of the real times e and d ! To see why, consider the case in which e is larger than the current time t. Then e cannot be computed exactly from Eq. (5) and (7), since we do not know how the slope of V will vary in the future. Intuitively, the situation is similar to having an empty reservoir in which we collect water from a spring. Although we can define a mark to indicate when the reservoir is full, we cannot say exactly *when* this will happen since the flow-rate of the spring may vary in the future. Therefore we will formulate our algorithm in terms of *virtual* eligible times and deadlines and not of the real times. With this the Earliest Eligible Virtual Deadline First (EEVDF) algorithm can be simply stated as follows:

EEVDF Algorithm. *A new quantum is allocated to the client that has the eligible request with the earliest virtual deadline.*

The EEVDF algorithm does not assume that a client will always use all the service time it has requested. This is an important feature that differentiates it from the fair queueing algorithms used for allocating bandwidth in communication networks [9, 12, 23], which assume that the length of a packet (and therefore the service time) is known when the packet arrives. Although in communication networks this is a realistic assumption[4], for processor scheduling it is much harder (and often impossible) to predict exactly the amount of service time a client will actually use. However, as we will show in Section 6, this does not compromise the fairness. Intuitively, this results from the definition of the virtual eligible eligible time: whenever a client uses less service time than it has requested, the virtual eligible time of the next request is pushed backwards (such that the lag to be zero). In this way EEVDF provides direct support for non-uniform quanta.[5]

Since EEVDF is formulated in terms of virtual times, in the remaining of this paper we use ve and vd to denote the virtual eligible time and virtual deadline respectively, whenever the corresponding real eligible time and the deadline are not given. Let $r^{(k)}$ denote the length of the k^{th} request made by client i, and let $ve^{(k)}$ and $vd^{(k)}$ denote the virtual eligible time and the virtual deadline associated to this request. If the client uses each time the *entire* service time it has requested, then by using Eq. (7) and (8) we obtain the following reccurence which computes both the virtual eligible time and the virtual deadline of each request:

[4]Usually, these algorithms take the packet arrival time to be the time at which the last bit from the packet has been received.

[5]Notice that EEVDF also provides support for fractional quanta by simply taking the time of the request to be equal to the desired fraction of a time quantum. The difference between fractional and non-uniform quanta is that while in the first case the fraction from the time quantum (that the client will actually use) is assumed to be known in advance, in the non-uniform quanta case this fraction is not known.

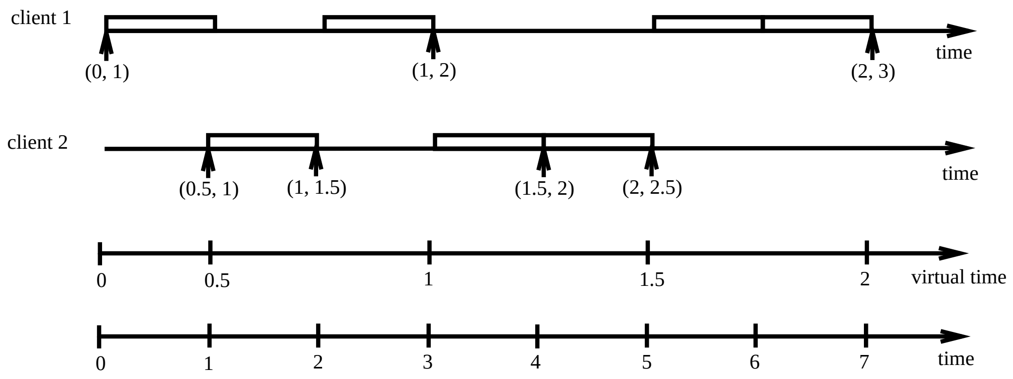


Figure 1: *An example of EEVDF scheduling involving two clients with equal weights $w_1 = w_2 = 2$. All the requests generated by client 1 have length 2, and all of the requests generated by client 2 are of length 1. Client 1 becomes active at time 0 (virtual time 0), while client 2 becomes active at time 1 (virtual time 0.5). The arrows represent the times when the requests are initiated (the pair associated to each arrow represents the virtual eligible time and the virtual deadline of the corresponding request).*

$$ve^{(1)} = V(t_0^i), \tag{9}$$

$$vd^{(k)} = ve^{(k)} + \frac{r^{(k)}}{w_i}, \tag{10}$$

$$ve^{(k+1)} = vd^{(k)}. \tag{11}$$

Next, let us consider the more general case in which the client does not use the entire service time it has requested. Since a client never receives more service time than requested, we need to consider only the case when the client uses the resource for less time than requested. Let $u^{(k)}$ denote the service time that client i actually receives during k^{th} request. Then the only change in Eq. (9)–(11), will be in computing the eligible time of a new request. Specifically, Eq. (11) is replaced by

$$ve^{(k+1)} = ve^{(k)} + \frac{u^{(k)}}{w_i}. \tag{12}$$

To clarify the ideas, let us take a simple example (see Figure 1). Consider two clients with weights $w_1 = w_2 = 2$ that issue requests with lengths $r_1 = 2$, and $r_2 = 1$, respectively. We assume that the time quantum is of unit size ($q = 1$) and that client 1 is the first one which enters competition at time $t_0 = 0$. Thus, according to Eq. (9) and (10) the virtual eligible time for the first request of client 1 is $ve = 0$, while its virtual deadline is $vd = 1$. Being the single client that has an outstanding eligible request, client 1 receives the first quantum. At time $t = 1$, client 2 enters the competition. Since the virtual time increases at a constant rate during the interval $[0, 1)$ (i.e., $\frac{1}{w_1} = 0.5$), its value at time $t = 1$ is $V(1) = \int_0^1 \frac{1}{w_1} d\tau = 0.5$. After the second client enters the competition the slope of the virtual time function becomes $\frac{1}{w_1+w_2} = 0.25$. Next, let us assume that client 2 makes its first request before the second quantum is allocated. Then at $t = 1$ there are two pending requests: one from client 1 with the virtual deadline 1 (which waits for another time quantum to fulfill its request), and one from client 2 which has the same virtual deadline i.e., 1. In this situation we arbitrarily break the tie in the favor of client 2, which therefore receives the second quantum upon termination. Since this quantum fulfills the current request of client 2, client 2 issues a new one with the virtual eligible time 1 and the virtual deadline 1.5. Thus, at time $t = 2$ the single eligible request is the one of client 1, which therefore receives the next quantum. Further, at $t = 3$ there are again two eligible requests: the one of client 2 that has just become eligible, and the new request issued by client 1. Since the deadline of the second client's request (1.5) is earlier than the one of the first client (2), the fourth quantum is allocated to the client 2. Further, Figure 1 shows how the next three quanta are allocated.

4 Fairness in Dynamic Systems

In this section we address the issue of *fairness* in dynamic systems. Throughout this paper, we assume that a dynamic system provides support for the following three operations: client joining the competition, client leaving the competition, and changing the client's weight. In an idealized fluid-flow system, supporting dynamic operations is trivial since at any moment of time the lag of any active client is zero. Unfortunately, in a system in which the service time is allocated in discrete time quanta, this is no longer true. In the remaining of this section we discuss how this affects the fairness in a dynamic system[6]. We consider the following two questions:

1. What is the impact of a client with non-zero lag leaving, joining, or changing its weight on the other clients ?
2. When a client with non-zero lag leaves the competition, what should be its lag when it rejoins the competition ?

In answering the first question, we start with a simple example. Let us assume that a client leaves the competition with a negative lag, i.e., after it has received more service time than it was entitled to. As we will show in Section 6, during any time interval, the total service time allocated to all active clients is equal to the service time that the clients should receive. Therefore, if a client leaves the competition with a negative lag, the remaining clients should have received less service time than they were entitled to. In short, a gain for one client translates into a loss for the other active clients. In this case, the question we need to answer is the following: How should the loss be distributed among the remaining active clients in order to attain fairness ? We answer this question by assuming that whenever a dynamic operation takes place, the effect (i.e., the resulting gain or loss) is *proportionally* distributed among all active clients. In other words, each active client will inherit a gain/loss proportional to its weight. Besides its intuitive appeal, as we will show, this policy has another advantage: it can be easily implemented by simply updating the virtual time. We note that although this policy is similar to the one employed by Waldspurger and Weihl in their stride scheduling algorithm [29, 30], the two policies are not equivalent. The difference consists in the way in which the virtual time is updated when a client leaves or joins the competition: while in stride scheduling only the slope of the virtual time [7] is updated, in our algorithm the value of the virtual time is updated as well (see Eq. 18, 19, 20 in this section).

To better understand our reasons in considering the above policy, let us consider the following example. Suppose that three clients with zero lags become active at time t_0, and at time t client 3 leaves the competition (see Figure 2). Further, we discuss the impact of client 3 leaving the competition on the other two clients.

[6]These issues were also addressed by Waldspurger and Weihl in the context of their stride scheduling algorithm [30].

[7]Instead virtual time, stride scheduling uses an equivalent concept, called *global pass*.

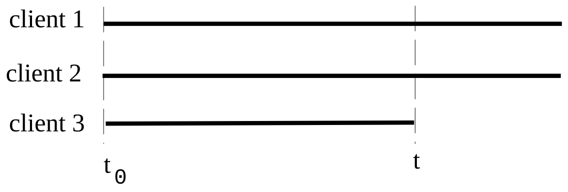


Figure 2: *The three clients become active at time t_0. At time t, client 3 leaves the competition.*

Since the number of active clients and their shares do not change during the interval $[t_0, t)$, the slope of the virtual time during this interval is constant and equal to $\frac{1}{w_1+w_2+w_3}$. Then from Eq. (3) and (6), the lag of each client at time t is

$$lag_i(t) = w_i \frac{t - t_0}{w_1 + w_2 + w_3} - s_i(t_0, t), \quad i = 1, 2, 3. \tag{13}$$

Next, we turn our attention to the clients 1 and 2 which remain active after client 3 leaves the competition. Since EEVDF is a work-conserving[8] algorithm, the total service time received by all active clients during the interval $[t_0, t)$ is equal to $t - t_0$. From here, the service time allocated to the first two clients during this interval can be expressed as $t - t_0 - s_3(t_0, t)$. Let t^+ be the time immediately *after* client 3 leaves the competition, where by neglecting the leaving operation overhead, we have $t^+ \to t$.

Now, what is the service time that clients 1 and 2 should have received at time t^+ ? A natural and intuitive approach would be to simply divide the entire service time received by both clients (i.e., $t - t_0 - s_3(t_0, t)$) proportional to the clients' weights, i.e.,

$$S_i(t_0, t^+) = (t - t_0 - s_3(t_0, t)) \frac{w_i}{w_1 + w_2}, \quad i = 1, 2, \tag{14}$$

We note that if $lag_3(t) \neq 0$, then this result is *different* from the service time each client should have received just before the departure of client 3, i.e., $(t - t_0)\frac{w_i}{w_1+w_2+w_3}$ $(i = 1, 2, 3)$. This is because once client 3 leaves, the remaining two clients will proportionally support the eventual loss or gain in the service time. Next, we show that in the EEVDF algorithm this is equivalent to a simple translation of the virtual time. By replacing $s_3(t_0, t)$ from Eq. (13) into Eq. (14), we obtain

$$\begin{aligned} S_i(t_0, t^+) &= (t - t_0)\frac{w_i}{w_1 + w_2 + w_3} + w_i \frac{lag_3(t)}{w_1 + w_2} \\ &= w_i(V(t) - V(t_0)) + w_i \frac{lag_3(t)}{w_1 + w_2}, \quad i = 1, 2. \end{aligned} \tag{15}$$

Finally, from Eq. (6) and (15) it follows that

$$V(t^+) = V(t) + \frac{lag_3(t)}{w_1 + w_2}, \quad i = 1, 2. \tag{16}$$

Thus, to maintain the fairness among the remaining clients, the virtual time should be updated according to the lag of the client which leaves the competition (as shown by the above equation). Since t^+ is asymptotically close to t, the service time received by any client at time t^+ is equal to the service time it has received at time t (i.e., $s_i(t_0, t) = s_i(t_0, t^+)$). From here, the lags of the first two clients at t^+ can be computed as

$$lag_i(t^+) \;=\; w_i(V(t^+) - V(t_0)) - s_i(t_0, t^+) = lag_i(t) + w_i \frac{lag_3(t)}{w_1 + w_2}, \quad i = 1, 2. \tag{17}$$

[8]A scheduling algorithm is said to be *work-conserving* if the resource cannot be idle while there is at least one active client (see Section 6 for details).

Therefore when client 3 leaves, its lag is *proportionally* distributed among the remaining clients, which is in accordance with our interpretation of fairness. By generalizing Eq. (16), we derive the following updating rule for the virtual time when a client j leaves the competition at time t

$$V(t) = V(t) + \frac{lag_j(t)}{\sum_{i \in \mathcal{A}(t^+)} w_i}. \tag{18}$$

Correspondingly, when a client j joins the competition at time t, the virtual time is updated as follows

$$V(t) = V(t) - \frac{lag_j(t)}{\sum_{i \in \mathcal{A}(t^+)} w_i}, \tag{19}$$

where $\mathcal{A}(t^+)$ contains all the active clients immediately after client j joins the competition, and $lag_j(t)$ represents the lag with which client j joins the competition. Although it might not be clear at this point, by updating the virtual time according to Eq. (18) and (19) we ensure that the sum over the lags of all active clients is always zero. This can be viewed as a conservation property of the service time, i.e., any time a client receives *more* service time than its share, there is at least another client that receives *less*. We note that if the lag of the client that leaves or joins the competition is zero, then according to Eq. (18) and (19) the virtual time does not change.[9]

We note that changing the weight of an active client is equivalent to a leave and a rejoin operation that take place at the same time. To be specific, suppose that at time t the weight of client j is changed from w_j to w'_j. Then this is equivalent to the following two operations: client j leaves the competition at time t, and rejoins it immediately (at the same time t) having weight w'_j. By adding Eq. (18) and (19), we obtain

$$V(t) = V(t) + \frac{lag_j(t)}{(\sum_{i \in \mathcal{A}(t)} w_i) - w_j} - \frac{lag_j(t)}{(\sum_{i \in \mathcal{A}(t)} w_i) - w_j + w'_j}. \tag{20}$$

As for join and leave operations, notice that the virtual time does not change when the weight of a client with *zero* lag is modified. Thus, in a system in which any client is allowed to join, leave, or change its weight only when its lag is zero, the variation of the virtual time is continuous.

Now let us turn our attention to the second question. We need to decide whether a client that becomes passive without using its entire share could use it when it again becomes active next time, and whether a client that leaves the competition after it has used more service time than its share should be penalized when it rejoins the competition. To be specific, consider a client that leaves the competition with a positive lag (i.e., after it has received less service time than it was entitled to). Then the question is whether the client should receive any compensation when it rejoins the competition. Unfortunately, there is no simple answer to this question. If we decide not to compensate, then the lost service time may accumulate over multiple periods of activity, and consequently, over large intervals of time the client may receive significantly less service time than it is entitled to. On the other hand, if we decide to compensate a client for the lost service when it rejoins the competition, this might hamper other clients. To see why, consider the following example. Suppose that before time t there are two active clients 1 and 2, and at time t client 2 becomes passive with a positive lag, $lag_2(t) > 0$. Next, assume that at a subsequent time t' client 2 rejoins the competition, while another client 3 is active (we assume that at this time client 1 is no longer active). Then client 2 will have to recover the service time that it has lost to client 1, at the expense of client 3 ! Consequently, client 3 will indirectly lose some service time because client 2 has not used its entire service time while it was previously active, which is not fair.

[9]However, notice that the slope of the virtual time changes.

5 Algorithm Implementation

Since, as we have seen in the previous section, there is no clear answer on what is *fair* to do with the lag of a client when it rejoins the competition, in this section we present three strategies for implementing the EEVDF algorithm. The characteristics of these strategies are dictated by the decision on whether a client could leave, join, or change its weight when its lag is non-zero, and by the decision on whether a client receives compensation or it is penalized when it rejoins the competition.

Strategy 1. In this strategy a client may leave or join the competition at any time, and depending on its lag it is either penalized or it receives compensation when it rejoins the competition. More precisely, if the client leaves the competition at time t, and rejoins at t', then $lag(t) = lag(t')$. Each time an event occurs (e.g., a client joining, leaving, or changing the weight of a client), the virtual time is updated according to Eq. (18), (19), and (20) respectively. Finally, we note that this strategy is appropriate for systems where it is desirable to maintain fairness over multiple periods of client activity. In Appendix A we give an example of how this strategy might be actually implemented.

Strategy 2. This strategy is similar to the previous one with the only difference being that the lag is not preserved after a client leaves the competition, i.e., any client that (re)joins the competition has zero lag. This strategy is appropriate for those systems in which the events that cause the clients to become active are independent. This is similar to a real-time system in which the processing time required by an event is assumed to be independent of the processing time required by any other event.

Strategy 3. In this strategy a client is allowed to leave, join, or change its weight, only when its lag is zero. Thus, in this case, there is no need to update the virtual time when a dynamic operation takes place. On the other hand, some complexity is added in ensuring that when these events occur the lag is indeed zero. In order to ensure that all events involve only clients with zero lag, we need to update the slope of the virtual time at the corresponding times in the fluid-flow system. The main problem in implementing this strategy is to update the slope of the virtual time when a client leaves the competition with a positive lag.[10] In this case the time at which the client should leave the competition in the fluid-flow system is *smaller* than the corresponding time in the real system. A solution would be to "undo" all the modifications in the system that occurred between the time when the client should leave the competition in the fluid-flow system and the time when it actually leaves. Unfortunately, this solution is very expensive to implement; it requires to store the event history and, in addition, the "undo" operation may involve a high overhead. In solving this problem, we assume that no event occurs during any time quantum. We note that this assumption is not as restrictive as it appears. For example, in the processor case, this is a realistic assumption since, in general, the scheduling algorithm executes only between time quanta. On the other hand, in communication networks we do not need to enforce this assumption since the service time (i.e., transmission time) is assumed to be known, before the request is initiated. The basic mechanisms to implement this strategy, under the assumption that no event can occur during a time quantum is given below.

First, assume that a client wants to leave the competition when its lag is negative. In this case, the idea is simply to *delay* the client until its lag becomes zero. This is done by issuing a dummy request of

[10] We note that in a system in which the client always uses the entire service time it has requested, by using Eq. (9)–(11) we can compute the virtual time when the client should leave in the fluid-flow system as the virtual deadline of the client's last request. This is the approach used by Parekh and Gallager in their Packet-by-Packet Generalized Processor Share algorithm [23].

zero length. This approach is motivated by the fact that, in this case, the eligible time is always chosen such that the lag is zero. Since a request cannot be processed before it becomes eligible, and since the virtual eligible time of the dummy request is equal to its deadline (see Eq. (10)), it follows that this request will be processed after its deadline. Thus, by using a request of zero length, we have reduced the case of a client which leaves with a negative lag to the case of a client which leaves with a nonnegative lag and therefore we further consider only the later case. As we will show in Section 6, in a system in which the virtual time varies continuously (such as in our system), a request is guaranteed to be fulfilled no latter than a time quantum after its deadline. Thus, between the moment when the lag of the client becomes zero, and the moment when the request is fulfilled, no other time quanta are allocated. Since *no* event occurs during this time quantum, it will not make any difference whether we update the virtual time after the time quantum expires, instead of exactly when the lag becomes zero.

A second question regarding this strategy is what to do with the remaining service time when the client leaves before its last request has been fulfilled. Here we take the simplest approach: the extra time quanta that are no longer used by the client are randomly allocated to other active clients without *charging* them, i.e., their received service times and theirs lags are not updated. Although more complex allocation schemes might be devised, they are not necessary to achieve the goal of this strategy which is to guarantee that as long as a client competes for the resource it will receive at *least* its share.

6 Fairness Analysis of the EEVDF Algorithm

In this section we determine bounds for the service time lag. First we show that during any time interval in which there is at least one active client, there is also at least one eligible pending request (Lemma 2). A direct consequence of this result is that the EEVDF algorithm is *work-conserving*, i.e., as long as there is at least one active client the resource cannot be idle. By using this result, in Theorem 1 we give tight bounds for the lag of any client in a steady system (see Definitions 1 and 2 below). Finally, we show that in the particular case when all the requests have durations no greater than a time quantum q, our algorithm achieves tight bounds which are optimal with respect to any proportional share allocation algorithm (Lemma 5).

Throughout this section we refer to any event that can change the state of the system, i.e., a client joining or leaving the competition, and changing the client's weight, simply as *event*. We introduce now some definitions to help us in our analysis.

Definition 1 *A system is said to be* **steady** *if all the events occurring in that system involve only clients with zero lag.*

Thus, in a steady system the lag of any client that joins, leaves, or has its weight changed, is zero. Recall that in a system in which all events involve only clients having zero lags the virtual time is continuous. As we will see, this is the basic property we use to determine tight bounds for the client lag. The following definition restricts the notion of steadiness to an interval.

Definition 2 *An interval is said to be* **steady** *if all the events occurring in that interval involve only clients with zero lag.*

We note that a steady system could be alternatively defined as a system for which any time interval is steady. The next lemma gives the condition for a client request to be eligible.

Lemma 1 *Consider an active client k with a positive lag at time t, i.e.,*

$$lag_k(t) \geq 0, \tag{21}$$

Then client k has a pending eligible request at time t.

Proof. Let r be the length of the pending request of client k at time t (recall that an active client has always a pending request), and let ve and vd denote the virtual eligible time and the virtual deadline of the request. For the sake of contradiction, assume the request is not eligible at time t, i.e.,

$$ve > V(t) \tag{22}$$

Let t' be the time when the request was initiated. Then from Eq. (7) we have

$$ve = V(t_0^k) + \frac{s_k(t_0^k, t')}{w_k}. \tag{23}$$

Since between t' and t the request was not eligible, it follows that the client has not received any service time in the interval $[t', t)$, and therefore $s_k(t_0^k, t') = s_k(t_0^k, t)$. By substituting $s_k(t_0^k, t')$ to $s_k(t_0^k, t)$ in Eq. (23) and by using Eq. (3) and (6) we obtain

$$\begin{aligned} lag_k(t) &= w_k(V(t) - V(t_0^k)) - s_k(t_0^k, t) \\ &= w_k(V(t) - V(t_0^k)) - w_k(ve - V(t_0^k)) \\ &= w_k(V(t) - ve). \end{aligned} \tag{24}$$

Finally, from Ineq. (22) it follows that $lag_k(t) < 0$, which contradicts the hypothesis and therefore proves the lemma. □

From Lemma 1 and from the fact that any zero sum has at least a nonnegative term, we have the following corollary.

Corollary 1 *Let A(t) be the set of all active clients at time t, such that*

$$\sum_{i \in \mathcal{A}(t)} lag_i(t) = 0. \tag{25}$$

Then there is at least one eligible request at time t.

The next lemma shows that at any time t the sum of the lags over all active clients is zero. An immediate consequence of this result and the above corollary is that at any time t at which there is at least one active client, there is also at least one pending eligible request in the system. Thus, in the sense of the definition given in [23], the EEVDF algorithm is *work-conserving*, i.e., as long as there are active clients the resource is busy.

Lemma 2 *At any moment of time t, the sum of the lags of all active clients is zero, i.e,*

$$\sum_{i \in \mathcal{A}(t)} lag_i(t) = 0. \tag{26}$$

Proof. The proof goes by induction. First, we assume that at time $t = 0$ there is no active client and therefore Eq. (26) is trivially true. Next, for the induction step, we show that Eq. (26) remains true after each one of the following events occurs: (i) a client joins the competition, (ii) a client leaves the competition, (iii) a client changes its weight. Finally, we show that (iv) during any interval $[t, t')$ in which none of the above events occurs if Eq. (26) holds at time t, then it also holds at time t'.

Case (i). Assume that client j joins the competition at time t with lag $lag_j(t)$. Let t^- denote the time immediately before, and let t^+ denote the time immediately after client j joins the competition, where t^+ and t^- are asymptotically close to t. Next, let $W(t)$ denote the total sum over the weights of all active clients at time t, i.e., $W(t) = \sum_{i \in \mathcal{A}(t)} w_i(t)$, and by convenience let us take $lag_j(t^-) = lag_j(t)$. Since $t^- \to t^+$ we have $s_i(t_0^i, t^-) = s_i(t_0^i, t^+)$. Then from Eq. (3) we obtain:

$$lag_i(t^+) = lag_i(t^-) + S_i(t_0^i, t^+) - S_i(t_0^i, t^-)$$

Further, by using Eq. (6) and (19), the lag of any active client i at time t^+ (including client j) is

$$lag_i(t^+) = lag_i(t^-) - lag_j(t)\frac{w_i}{W(t^+)}. \qquad (27)$$

Since $\mathcal{A}(t^+) = \mathcal{A}(t^-) \cup \{j\}$, and since from the induction hypothesis we have $\sum_{i \in \mathcal{A}(t^-)} lag_i(t^-) = 0$, by using Eq. (27), we obtain

$$\begin{aligned} \sum_{i \in \mathcal{A}(t^+)} lag_i(t^+) &= \sum_{i \in \mathcal{A}(t^+)} (lag_i(t^-) - lag_j(t)\frac{w_i}{W(t^+)}) \qquad (28) \\ &= \sum_{i \in \mathcal{A}(t^+)} lag_i(t^-) - lag_j(t)\frac{\sum_{i \in \mathcal{A}(t^+)} w_i}{W(t^+)} \\ &= \sum_{i \in \mathcal{A}(t^-)} lag_i(t^-) + lag_j(t^-) - lag_j(t) = 0. \end{aligned}$$

Case (ii). The proof of this case is very similar to the one of the previous case; therefore we omit it here.
Case (iii). Changing the weight of a client j from w_j to w_j' at time t can be viewed as a sequence of two events: first, client j leaves the competition at time t; second, it joins the competition at the same time t, but with weight w_j'. Thus, the proof of this case reduces to the previous two cases.
Case (iv). Consider an interval $[t, t')$ in which no event occurs, i.e., no client leaves or joins the competition and no weight is changed during the interval $[t, t')$. Next, assume that $\sum_{i \in \mathcal{A}(t)} lag_i(t) = 0$. Then we shall prove that $\sum_{i \in \mathcal{A}(t')} lag_i(t') = 0$. By using Eq. (3) and (6) we obtain

$$\begin{aligned} \sum_{i \in \mathcal{A}(t')} lag_i(t') &= \sum_{i \in \mathcal{A}(t')} (S_i(t_0^i, t') - s_i(t_0^i, t')) \qquad (29) \\ &= \sum_{i \in \mathcal{A}(t)} (S_i(t_0^i, t) - s_i(t_0^i, t)) + \sum_{i \in \mathcal{A}(t)} (S_i(t, t') - s_i(t, t')) \\ &= \sum_{i \in \mathcal{A}(t)} lag_i(t) + \sum_{i \in \mathcal{A}(t)} S_i(t, t') - \sum_{i \in \mathcal{A}(t)} s_i(t, t') \\ &= \sum_{i \in \mathcal{A}(t)} w_i(V(t') - V(t)) - \sum_{i \in \mathcal{A}(t)} s_i(t, t') \\ &= (t' - t) - \sum_{i \in \mathcal{A}(t)} s_i(t, t'). \end{aligned}$$

Next we show that the resource is busy during the entire interval $[t, t')$. For contradiction assume this is not true. Let l denote the earliest time in the interval $[t, t')$ when the resource is idle. Similarly to Eq. (29) we have:

$$\sum_{i \in \mathcal{A}(l)} lag_i(l) \quad = \quad (l-t) - \sum_{i \in \mathcal{A}(t)} s_i(t,l).$$

Since the resource is not idle at any time between t and l, it follows that the total service time allocated to all active clients during the interval $[t, t')$ (i.e., $\sum_{i \in \mathcal{A}(t)} s_i(t,l)$) is equal to $l - t$. Further, from the above equation we have $\sum_{i \in \mathcal{A}(l)} lag_i(l) = 0$. But then from Lemma 1 it follows that there is at least one eligible request at time l, and therefore the resource cannot be idle at time l, which proves our claim. Further, with a similar argument, it is easy to show that $\sum_{i \in \mathcal{A}(t')} lag_i(t') = 0$ which completes the proof of this case.

Since these are the only cases in which the lags of the active clients may change, the proof of the lemma follows. □

The following lemma gives the upper bound for the maximum delay of fulfilling a request in a steady system. We note that this result is similar to the one obtained by Parekh and Gallager [23] for their Generalized Processor Sharing algorithm, i.e., in a communication network, a packet is guaranteed not to miss its deadline by more than the time required to send a packet of maximum length.

Lemma 3 *In a steady system any request of any active client k is fulfilled no later than $d + q$, where d is the request's deadline, and q is the size of a time quantum.*

Proof. Let e be the eligible time associated to the request (with deadline d) of client k. Consider the partition of all the active clients at time d, into two sets B and C, where set B contains all the clients that have at least a deadline in the interval $[e, d]$, and set C contains all the other active clients (see Figure 3). Let t be the latest time no greater than d at which a client in C receives a time quantum, if any. Further we consider two cases whether such a t exists or not.

Case 1 (t exists). Here we consider two sub-cases whether $t \in [e, d)$, or $t < e$. First assume that a client in C receives a time quantum at a time $t \in [e, d)$. Since all the deadlines of the pending requests issued by clients in C are larger than d, this means that at time t the pending request of client k is already fulfilled. Consequently, in the first sub-case the request of client k is fulfilled before time d.

For the second sub-case, let us D denote all the active clients that have at least one eligible request with the deadline in the interval $[t, d)$ (see Figure 3). Further, let $D(\tau)$ denote the subset of D containing the active clients at time τ. Since a time quantum is allocated to a client in C at time t, it follows that no other client with an earlier deadline is eligible at t. For any client j belonging to $D(t)$, let e_j be the eligible time of its pending request at time t. Since the deadlines of these requests are no greater than d (and therefore smaller than any deadline of any client in C), it follows that all these pending requests are not eligible at time t, i.e., $t < e_j$. Notice that besides the clients in $D(t)$, the other clients that belong to D are those that eventually join the competition after time t. For any client j in D that joins the competition after time t, we take e_j to be the eligible time of its first request.

Next, for any client j belonging to D, let d_j denote the largest deadline no greater than d of any of its requests (notice that the eligible time e_j and the deadline d_j might not be associated to the same request). From Eq. (10) it easy to see that after client j receives $S_j(e_j, d_j)$ time units, all its requests in the interval $[e_j, d_j)$ are fulfilled. Thus, the service time needed to fulfill all the requests which have deadlines in the interval $[t, d)$ is

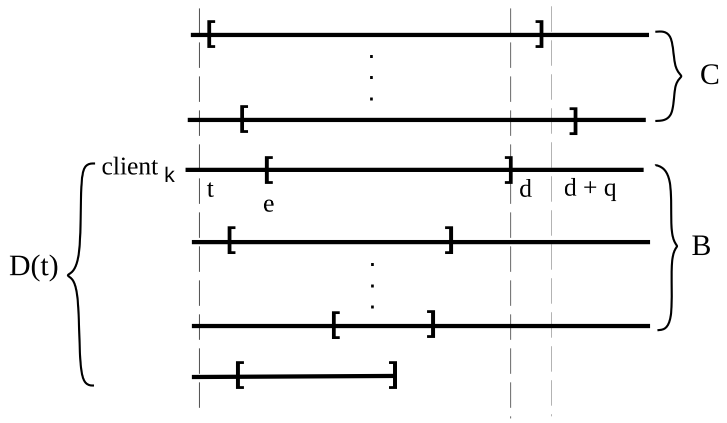


Figure 3: *The current pending request of client k has the eligible time e and deadline d. Set B contains all the active clients that have at least one request with the deadline in the interval $[e, d]$, while set C contains all the other active clients. Time t represents the largest time no greater than d at which a client from C receives a time quantum. Finally, set $D(t)$ contains all the active clients at time t that have at least one eligible request with the deadline in the interval $[t, d)$.*

$$\sum_{j \in D} S_j(e_j, d_j) = \sum_{j \in D}\left(\int_{e_j}^{d_j} \frac{w_j}{\sum_{i \in \mathcal{A}(\tau)} w_i} d\tau\right). \tag{30}$$

By decomposing the above sum over a set of disjoint intervals $J_l = [a_l, b_l)$ $(1 \leq l \leq m)$ covering $[t, d)$, such that no interval contains any eligible time or deadline of any client belonging to D, we can rewrite Eq. (30) as

$$\sum_{j \in D} S_j(e_j, d_j) = \sum_{l=1}^{m}\left(\int_{a_l}^{b_l} \frac{\sum_{i \in D(a_l)} w_i}{\sum_{i \in \mathcal{A}(a_l)} w_i} d\tau\right) < \sum_{l=1}^{m}\left(\int_{a_l}^{b_l} d\tau\right) = \sum_{l=1}^{m}(b_l - a_l) = d - t, \tag{31}$$

The above inequality results from the fact that $D(\tau)$ is a proper subset of $\mathcal{A}(\tau)$ at least for some subintervals J_i (otherwise, if $\mathcal{A}(\tau)$ is identical to $D(\tau)$ over the entire interval $[t, d)$, sets C and C' would be empty).

Assume that at time $d + q$ the request of client k (having the deadline d) is not fulfilled yet. Since no client in C can be served before the request of client k is fulfilled, it follows that the service time between $t + q$ and $d + q$ is allocated only to the clients in D. Consequently, during the entire interval $[t + q, d + q)$, there are $d - t$ service time units to be allocated to all clients in D. Next, recall that any client j belonging to D will not receive any other time quantum after its request having deadline d_j is eventually fulfilled, as long as the request of client k is not fulfilled. This is simply because the next request of client j will have a deadline greater than d. But according to Eq. (31) the service time required to fulfill *all* the requests having the deadlines in the interval $[t, d)$ is less than $d - t$, which means that at some point the resource is idle during the interval $[t + q, d + q)$. But this contradicts the fact that EEVDF is work-conserving, and therefore proves this case.

Case 2. (t does not exist) In this case we take t to be the time when the first client joins the competition. From here the proof is similar to the one for the first case, with the following difference. Since set C is empty, all the time quanta between t and d are allocated to the clients in D, and therefore, in this case, we show that in fact client k does not miss the deadline d. □

Following we give a similar result for a steady interval. Mainly, we show that for certain subintervals of a steady interval the same bound holds. This shows that a system which allows clients with non-zero lag to join, leave, or to change their weight, will eventually reach a steady state.

Lemma 4 *Let $I = [t_1, t_2)$ be a steady interval, and let d_m be the largest deadline among all the pending requests of the clients with negative lags which are active at t_1. Then any request of any active client k is fulfilled no later than $d + q$, if $d \in [d_m, t_2)$.*

Proof. Similarly to the proof of Lemma 3, we consider the partition of all the active clients at time d, into two set B and C, where set B contains all the active clients that have at least a deadline in the interval $[e, d]$, and set C contains all the other clients. Similarly, we let t denote the latest time in the interval $[t_1, d)$ when a client in C receives a time quantum, if any. Further, we consider two cases whether such t exists or not.

Case 1. (t exists) The proof proceeds similarly to the one for Case 1 in Lemma 3.

Case 2. (t does not exist) In this case we consider two sub-sets of C: C^- containing all clients in C that had negative lags at time t_1, and C^+ containing all the other clients in C. Since no client belonging to C^- receives any time quantum before d_m it follows that no pending request of any client in C^- is fulfilled before its deadline (recall that the deadlines of all the other clients with negative lags at t_1 are $\leq d_m$) and therefore all clients in C^- will have nonnegative lags at time d_m. On the other hand, since all clients in C^+ had nonnegative lags at time t_1, and since they do not receive any time quanta between t_1 and d_m, all of them will have positive lags at d_m. Thus, we have

$$\sum_{i \in C} lag_i(d) > 0 \tag{32}$$

On the other hand, we note that if the request of client k is not fulfilled before its deadline, then no other client belonging to B will receive any other time quantum after its last request with the deadline no greater than d is fulfilled. But then from Eq. (3) it follows that their lags as well as the lag of client k are positive at time d, i.e.,

$$\sum_{i \in B} lag_i(d) \geq 0 \tag{33}$$

Further, by adding Eq. (32) and (33), we obtain

$$\sum_{i \in \mathcal{A}(d)} lag_i(d) > 0 \tag{34}$$

which contradicts Lemma 2, and therefore completes the proof.□

The next theorem gives tight bounds for a client's lag in a steady system.

Theorem 1 *The lag of any active client k in a steady system is bounded as follows,*

$$-r_{max} < lag_k(d) < max(r_{max}, q), \tag{35}$$

where r_{max} represents the maximum duration of any request issued by client k. Moreover, these bounds are asymptotically tight.

Proof. Let e and d be the eligible time and the deadline of a request with duration r issued by client k. Since S_k increases monotonically with a slope no greater than one (see Eq. (4)), from Eq. (3) it follows that the lag of client k decreases as long as it receives service time, and increases otherwise. Further, since a request is not serviced before it is eligible, it is easy to see that the minimum lag is achieved when the client receives the entirely service time as soon as the request becomes eligible. In other words, the minimum lag occurs at time $e + r$, if the request is fulfilled by that time. Further, by using Eq (3) we have

$$\begin{aligned} lag_k(e+r) &= S_k(t_0^k, e+r) - s_k(t_0^k, e+r) \\ &= S_k(t_0^k, e) + S_k(e, e+r) - (s_k(t_0^k, e) + s_k(e, e+r)) \\ &= lag_k(e) + S_k(e, e+r) - s_k(e, e+r) \end{aligned} \tag{36}$$

From the definition of the eligible time (see Section 2) we have $lag_k(e) \geq 0$, and thus from the above equation we obtain

$$lag_k(e+r) \geq S_k(e, e+r) - s_k(e, e+r) > -s_k(e, e+r) \geq -r. \tag{37}$$

Since this is the lower bound for the client's lag during a request with duration r, and since r_{max} represents the maximum duration of any request issued by client k, it follows that at a any time t while client k is active we have

$$lag_k(t) \geq -r_{max}. \tag{38}$$

Similarly, the maximum lag in the interval $[e, d)$ is obtained when the entire service time is allocated as late as possible. Since according to Lemma 3, the request is fulfilled no later than $d + q$, it follows that the latest time when client k should receive the first quantum is $d + q - r$. We consider two cases: $r \geq q$ and $r < q$. In the first case $d + q - r \leq d$, and therefore we obtain $S_k(e, d+q-r) < S_k(e, d) = r$. Let t_1 be the time at which the request is issued. Further, from the definition of the eligible time, and from the fact that the client is assumed that it does not receive any time quantum during the interval $[t_1, d+q-r)$, we have for any time t while the request is pending

$$\begin{aligned} lag_k(t) &\leq S_k(t_0^k, d+q-r) - s_k(t_0^k, d+q-r) \\ &= S_k(t_0^k, e) + S_k(e, d+q-r) - s_k(t_0^k, t_1) - s_k(t_1, d+q-r) \\ &= (S_k(t_0^k, e) - s_k(t_0^k, t_1)) + S_k(e, d+q-r) - s_k(t_1, d+q-r) \\ &= S_k(e, d+q-r) < r. \end{aligned} \tag{39}$$

Since the slope of S_k is always no greater than one, in the second case we have $S_k(e, d+q-r) = S_k(e, d) + S_k(d, d+q-r) < r + q - r = q$, and from here we obtain

$$lag_k(t) \leq S_k(e, d+q-r) < q. \tag{40}$$

Finally, by combining Eq. (39) and (40) we obtain $lag_k(t) < max(q, r)$. Thus, at any time t while the client is active we have

$$lag_k(t) < max(q, r_{max}). \tag{41}$$

To show that the bound $lag_k(t) > -r_{max}$ is asymptotically tight, consider the following example. Let w_1, w_2 be the weights of two active clients, such that $w_1 \ll w_2$. Next, suppose that both clients become active at time t_0 and their first requests have the lengths r_{max} and r'_{max}, respectively. We assume that r_{max} and r'_{max} are chosen such that the virtual deadline of the first client's request is smaller than the virtual deadline of the second client's request, i.e., $t_0 + \frac{r_{max}}{w_1} < t_0 + \frac{r'_{max}}{w_2}$. Then client 1 receives the entire service time before client 2, and thus from Eq. (3) we have $lag_1(r_{max}) = S_1(t_0, t_0 + r_{max}) - r_{max}$. Next, by using Eq. (4) we obtain $S_1(t_0, t_0 + r_{max}) = \frac{w_1}{w_1+w_2}$, which approaches zero when $\frac{w_1}{w_2} \to \infty$, and consequently $lag_1(r_{max})$ approaches $-r_{max}$.

To show that the bound $lag_k(t) < max(r_{max}, q)$ is asymptotically tight, we use the same example. However, in this case we assume that the virtual deadline of the first request of client 1 is earlier than the virtual deadline of the first request of client 2, such that client 1 receives its entire service time just prior to its deadline. Since the details of the proof are similar with the previous case we do not show them here. □

Notice that the bounds given by Theorem 1 apply independently to each client and depend only on the length of their requests. While shorter requests offer a better allocation accuracy, the longer ones reduce the system overhead since for the same total service time fewer requests need to be generated. It is therefore possible to trade between the accuracy and the system overhead, depending on the client requirements. For example, for an intensive computation task it would be acceptable to take the length of the request to be in the order of seconds. On the other hand, in the case of a multimedia application we need to take the length of a request no greater than several tens of milliseconds, due to the delay constraints. Theorem 1 shows that EEVDF can accommodate clients with different requirements, while guaranteeing tight bounds for the lag of each client during a steady interval. The following corollary follows directly from Theorem 1.

Corollary 2 *Consider a steady system and a client k such that no request of client k is larger than a time quantum. Then at any time t, the lag of client k is bounded as follows:*

$$-q < lag_k(t) < q. \tag{42}$$

Next we give a simple lemma which shows that the bounds given in Corollary 3 are optimal, i.e., they hold for any proportional share algorithm.

Lemma 5 *Given any steady system with time quanta of size q and any proportional share algorithm, the lag of any client is bounded by $-q$ and q.*

Proof. Consider n clients with equal weights that become active at time 0. We consider two cases: (i) each client receives exactly one time quantum out of the first n quanta, and (ii) there is a client k which receives more than a time quanta. From Eq. (3), it is easy to see that, at time q, the lag of the client that receives the first quantum is

$$lag(q) = \frac{q}{n} - q. \tag{43}$$

Similarly, the lag of the client which receives the n^{th} time quantum is (at time $n-1$, immediately before it receives the time quantum)

$$lag(q(n-1)) = q - \frac{q}{n}. \tag{44}$$

For contradiction, assume that there is a proportional share algorithm that achieves an upper bound smaller than q, i.e., $q - \epsilon$, where ϵ is a positive real. Then by taking $n > \frac{q}{\epsilon}$, from Eq. (44), it follows that $lag(q(n-1)) > q - \epsilon$ which is not possible. Similarly, it can be shown that no algorithm can achieve a lower bound better than -q.

For the second case (ii), notice that since client j receives more than one time quanta, there must be another client k that does not receive any time quanta in the first n time units. Then it is easy to see that the lag of client j is smaller than $-q$ after it receives the second time quantum, and the lag of client k is larger than q after just before receiving its first time quantum, which completes our proof. □

7 Related Work

In this section we present a compressive overview of the related work. We classify the scheduling algorithms as follows: time-dependent priority, real-time, fair queueing, and proportional share.

7.1 Time-Dependent Priority

Many of the existing operating systems rely on the concept of *priority* to allocate processing time to competing processes [25]. In the static priority schemes, a process with higher priority has absolute precedence over a process with lower priority. Unfortunately, this schemes are inflexible and may lead to starvation [25]. In trying to overcome these problems, several solutions were proposed. One of the best-known schemes is *decay usage scheduling* [13] which tries to ensure fairness by changing the process priorities according to their recent CPU usage. This policy was implemented in many operating systems, such as Unix BSD [16] and System V [1]. The main drawback of this policy is that it offers only a crude control over resource allocation during short periods of time.

Recently, Fong and Squillante have proposed a new scheduling discipline called Time-Function Scheduling (TFS) [11]. The priority of a client in TFS is defined by a time-dependent function, i.e., the priority increases linearly with time while the client waits to be scheduled, and it is reinitialized to a predefined value whenever the client is scheduled. In TFS clients are partitioned in disjoint classes based on their characteristics and scheduling objectives. All clients belonging to the same class have associated the same time-dependent function and are organized in a FCFS queue. By serving them in a round-robin fashion, the scheduler ensures that the client with the highest priority in that class is always at the front of the queue. Thus to select the client with the highest overall priority it is enough to search among the clients which are at the front of their queues. In this way, the dispatch operation can be efficiently implemented in $O(\log c)$, where c represents the total number of classes. On the other hand, the time-complexity of updating clients' priorities is $O(c \log c)$. TFS provides effective and flexible control over resource allocation and it can be used to achieve general scheduling objectives such as relative per-class throughputs and low waiting time variance. Although somewhat indirectly, TFS can also archive proportional share allocation by assigning an equal share to each client in the same class. However, the algorithm accuracy depends on the frequency at which the clients' priorities are updated. Since the updating operation is rather expensive this limits the allocation accuracy that can be achieved.

7.2 Real-Time

Real-time systems were specifically developed for critical time tasks which require strong deadline guarantees. These tasks are characterized by a sequence of events that arrive in a certain pattern (usually periodic). Each event is described by its predicted service time and a deadline before which the event should be processed.

Two of the most popular algorithms for scheduling periodic tasks, *rate monotonic* (RM) and *earliest deadline first* (EDF), were proposed and analyzed by Liu and Layland in [18]. In RM, tasks are assigned priorities in the decreasing order of their periods, i.e., the task with the smallest period has the highest priority. While in RM the priorities are fixed, in EDF they change dynamically whenever a task initiates a new request. More precisely, the EDF algorithm assigns priorities to tasks corresponding to the deadlines of their current requests, i.e., the task which has the request with the earliest deadline is assigned the highest priority.

We note here that in a static fluid-flow system (in which the weights and the number of active clients do not change) the EEVDF and EDF algorithms are equivalent. To see why, consider how EEVDF behaves when a new request with an earlier deadline than the process that is currently executing is issued. In this case, as soon as the current time quantum expires, the new request is scheduled for execution. Since in an idealized fluid-flow model the size of a time quantum is arbitrarily small, this is equivalent to schedule the new request as soon as it arrives, which is identical to the policy employed by EDF.

In order to guarantee that all tasks will meet their deadlines, both RM and EDF impose strict admission policies. Specifically, Liu and Layland [18] have shown that under the EDF policy all tasks will meet their deadlines as long as the processor is not over-utilized (i.e., its utilization is $\leq 100\%$). Similarly, for the RM algorithm, they have given a schedulability test with the worst case processor utilization of 69%. For a specified set of tasks, this bound can be improved by using the exact analysis given by Lehoczky, Sha, and Ding in [17]. Unfortunately, this analysis is more expensive, which makes it less appealing for practical implementations. Besides ensuring a higher processor utilization, another advantage of EDF versus RM is that, for the same set of tasks, it never generates more preemptions than RM, which reduces the context-switching overhead. On the other hand, since it uses fixed priorities RM is simpler and slightly more efficient to implement than EDF. Finally, another advantage of RM is that in case of overload the tasks with higher priorities will still meet their deadlines at the expense of tasks with lower priorities, while under the EDF algorithm all tasks could miss their deadlines.

Both RM and EDF were the solutions of choice used to add support for continuous media and real-time applications to the existing operating systems. For example, in designing an application platform for distributed multimedia applications, Coulson *et al* [8] use the EDF algorithm for processor scheduling. Mercer, Savage, and Tokuda consider both RM and EDF algorithms in developing a flexible higher level abstraction, called *processor capacity reserves* [20, 21], specifically designed for measuring and controlling processor usage in a microkernel system. In their model, each client (thread) has associated a reserve to which its computation time is charged. The scheduler uses the usage measurements for each client to control and enforce its reservation.

Unlike the above approaches which try to add support for real-time applications such as multimedia by extending and/or modifying the general purpose CPU schedulers in the existing operating systems, Bollela and Jeffay [4] take a more radical approach. Their idea is to partition the processor and other shared system resources into two virtual machines: one machine running a general purpose operating system, and the other one running a real-time kernel support. Specifically, the CPU is multiplexed between the two systems, each operating system running alternatively for a predefined time slice. While this approach achieves a high level of isolation between general purpose and real-time applications, running two different operating systems increases both the overhead and the resource requirements in the system.

In general, real-time based schedulers do not provide an integrated solution for continuous media, interactive, and batch applications. For this reason, general purpose operating systems that use real-time based schedulers for supporting continuous media and interactive applications, also employ more conventional schedulers (such as round-robin) for batch activities. Compared to proportional-share schedulers, real-time schedulers are more restrictive and less flexible. As an example, when an application terminates it is difficult to efficiently redistribute its share among the applications that are still active. Finally we note that although real-time based schedulers provide stronger timeliness guarantees, the guarantees offered by the EEVDF algorithm (i.e., a deadline is never missed by more than a time quantum) are good enough to accommodate a broad range of real-time applications.

Recently, Jeffay and Bennette have proposed a new abstraction for multimedia applications, called *rate-based execution* (RBE) [13]. In RBE a process is characterized by three parameters: x, y, and d, where x represents the number of events that arrive during a time interval with the duration y, and d represents the desired maximum elapsed time between the delivery of an event and the completion of that event. Like EEVDF, RBE does not make any assumptions about the interarrival times, and about the distribution of the processing time during y time units. We note that specifying parameters x and y is equivalent to specifying the share that the process should receive during a time interval with the duration y. While RBE provides better control over the maximum elapsed time d (in the EEVDF this is implicitly determined from the client share and the request duration), the EEVDF provides more flexibility in share allocation over time intervals of arbitrary length. Moreover, although RBE generalizes the traditional real-time models, it does not address the problem of supporting batch and multimedia applications in an integrated environment.

Nieh and Lam have developed a novel integrated processor scheduler that provides support for multimedia applications in a general purpose operating system [22]. Similarly to a request in EEVDF, they associate to each client a *minimum execution rate* which is defined as the desired fraction of the processing time that the client should receive in a given interval. For continuous media and interactive applications the minimum execution rates result directly from their time constraints, while for batch applications the minimum execution rates express their minimum acceptable rates of forward progress. These rates are translated into a series of deadlines, which are similar to the deadlines of the clients' requests in EEVDF. The scheduler attempts to meet these deadlines by using an EDF algorithm. In addition, the scheduler assigns to each activity a priority. When the system is overloaded, the scheduler tries to meet the time constraints for the clients with higher priorities at the expense of the clients with lower priorities.

We note that both EEVDF and this scheduler rely on similar concepts (i.e., minimum execution rate and request respectively) in providing an integrated solution for scheduling continuous media, interactive and batch applications. However since this scheduler is based on a simple EDF policy it is not clear how it behaves in a highly dynamic environment. For example, it is not clear what is the trade-off (if any) between efficient implementation of dynamic operations (i.e., adjusting the clients' minimum execution rates) and the degree of fairness ensured by the scheduler. While the addition of priorities makes the scheduler more flexible and effective in supporting a broader range of applications, this might increase the complexity and possibly the scheduling overhead. Although in EEVDF we are not providing similar support for expressing clients' priorities, we note that this support can be integrated in higher level resource abstractions such as monetary funds [26].[11]

[11] In [26] we consider two classes of services: bounded and unbounded. A client receiving a bounded service is guaranteed to receive a share of the resources which is inferior and/or superior bounded.

7.3 Fair Queuing

The EEVDF algorithm shares many common characteristics with the *fair queueing algorithms* which were originally developed for bandwidth allocation in communication networks. These algorithms use the same notion of idealized fluid-flow model to express the concept of fairness in a dynamic system, and the notion of virtual time to track the work progress in the system. Since the idealized model cannot be applied in the context of a packet-based traffic (where a packet transmission cannot be preempted by other packets) Demers *et al* have introduced a new policy, called *packet-by-packet fair queueing* (PFQ) [9]. According to this policy, the order in which the packets are served is defined as the order in which they would finish in the corresponding ideal fluid-flow system.

Parekh and Gallager [23] have analyzed the PFQ[12] scheme when the input traffic stream conforms to the *leaky-bucket* constraints [6]. Namely, they proved that each packet is processed within t_{max} time units from the time at which the packet would be processed in the corresponding idealized system, where t_{max} is the transmission time of the largest possible packet.

We note that the computation of the virtual finishing time in PFQ is identical to the computation of the virtual deadline in EEVDF. Moreover, the PFQ policy is very similar to the one employed by the EEVDF algorithm; both select to send the packet which has the earliest virtual deadline (finishing time) in the idealized system. The major difference between the two policies is that in PFQ a packet becomes eligible as soon as it arrives, while in EEVDF a packet becomes eligible only after all the previous messages belonging to the same session have been sent. As we will show in the following example, this difference is critical in reducing the session lag from $O(n)$ to $O(1)$, where n is the number of active sessions (an active session is defined as a session that has at least one packet to send).

Let us consider a communication switch with $n+1$ active sessions, where session 1 has weight n, and all the other sessions have weights equal to 1. Further, for simplicity, assume that all the packets are of equal size and the transmission of one packet takes one time unit. From Eq. (9)–(11) it follows that the virtual deadlines of the packets belonging to session 1 are: $\frac{1}{n}$, $\frac{2}{n}$, $\frac{3}{n}$, $\ldots$, $\frac{n-1}{n}$, 1. Similarly, the deadline of the first packet of any of the other sessions is 1. Then it is easy to see that, since in PFQ all the packets are eligible at time 0, the first $n-1$ packets of session 1 will be sent without interruption during the time interval $[0, n-1)$, and therefore the total service time received by session 1 during the interval $[0, n-1)$ is $s_1(0, n-1) = n-1$. On the other hand, from Eq. (5) and (6) it follows that the total service time that session 1 should have received during the same interval is $S_1(0, n-1) = \frac{n-1}{2}$. Finally, from Eq. (3), the lag of session 1 at time $n-1$ is

$$lag_1(n-1) = S_1(0, n-1) - s_1(0, n-1) = -\frac{n-1}{2},$$

which proves our point. We note that in EEVDF this does not happen since session 1 will have *only one* packet eligible at a time. More precisely, the eligible times of the first $n-1$ packets of session 1 are: 0, $\frac{1}{n}$, $\ldots$, $\frac{n-2}{n}$. Thus, after the first packet belonging to session 1 is sent, the virtual time becomes $\frac{1}{2n}$. Since at this time the second packet of session 1 is not eligible yet, the next packet to be sent will belong to one of the other n sessions. Thus, in EEVDF the transmission of the first $n-1$ packets of session 1 is interleaved with transmissions of packets from the other sessions.

We note that guaranteeing stronger bounds for session lags helps in reducing the buffer requirements. As an example consider a receiver that processes the incoming packets at fixed periods of time. To be specific, assume that every two time units the receiver processes an incoming packet from session 1 (see

[12]In their work the PFQ is referred as *packet-by-packet generalized processor sharing* (PGPS).

Algorithm		Lottery	Charge	BITREV	Stride	PD	EEVDF
Lag		O($\sqrt{m}$)	O(n_c)	O(log w)	O(log n_c)	O(1)	O(1)
Quantum	fractional	Yes	Yes	Yes	Yes	No	Yes
	non-uniform	Yes	Yes	No	Yes	No	Yes
Operation complexity	selection	O(log n_c)	O(n_c)	O(log n_c)	O(log n_c)	O(log n_c)	O(log n_c)
	joining	O(log n_c)	O(n_c)	O(log n_c)	O(log n_c)	O(n_c)	O(log n_c)
	leaving	O(log n_c)	O(n_c)	O(log n_c)	O(log n_c)	O(n_c)	O(log n_c)
	change weight	O(log n_c)	O(n_c)	O(log n_c)	O(log n_c)	O(n_c)	O(log n_c)

Table 1: *Comparison between EEVDF and other proportional share scheduling algorithms: Lottery, Charge-based, Bit Reversal (BITREV), Stride and PD. Legend : n_c denotes the number of clients that compete for the resource, w denotes the client weight, and m denotes the number of time quanta allocated from the moment when client enters competition.*

the above example). Then when using PFQ, the size of the buffer to store the incoming packets is $O(n)$, while when using EEVDF it is only $O(1)$.[13]

Recently, Golestani has proposed a new scheme called *self clocked fair queueing* (SCFQ) [12], that approximates the PFQ scheme. In this scheme he uses the notion of virtual time to measure the progress of work in the *actual* packet-based system, not in the idealized one. The main advantage of this new scheme is that it does not need to update the virtual time whenever an event occurs in the *idealized* model, and therefore could be more efficiently implemented (although it still maintains the $O(\log n)$ complexity for insertion and deletion from priority queue, where n is the number of clients). However, this does not come for free; the bounds guaranteed by this scheme for the session lags are within a factor of two from the ones guaranteed by PFQ. As an open problem, we note that it is not clear whether our EEVDF algorithm will benefit by using the same notion of virtual time as in SCFQ. This is because in EEVDF the eligible times are computed based on the virtual times in the *idealized* system, and trivially approximating them with the virtual times in the *real* system would result in a non-work conserving algorithm, which will make the algorithm much harder to analyze, and more expensive to implement.

7.4 Proportional Share

Recently, a significant number of proportional share scheduling algorithms have been developed [2, 26, 28, 29, 30]. In comparing these algorithms to EEVDF we consider the following criteria (see Table 1):
•**Lag** – The difference between the service time the client should receive and the service time it actually receives while competing for the resource, as defined by Eq. (3).
•**Quantum** – Specifies whether or not the algorithm provides support for fractional and non-uniform time quanta.
•**Operation complexity** – Time complexity for each of the basic dynamic operations supported by the scheduler, i.e., client selection, joining, leaving, and changing the weight of a client.

Waldspurger and Weihl [28, 30] have proposed a proportional share scheduling algorithm called *lottery scheduling*. In their algorithm the resource rights are encapsulated into lottery tickets. Every client has associated a certain number of tickets that determines the share of the resource that the client should

[13] Here we assume that the receiver and sender do not use any flow control mechanism.

receive. At the beginning of every time slice a lottery is held and the client with the winning ticket is selected and granted to use the resource. Since the ticket numbers are randomly generated, this scheme ensures that, on the average, a client receives a number of time slices *proportional* to the fraction of tickets it has. As shown in Table 1, the client lag is proportional to the square root of the number of allocated quanta m, i.e., $O(\sqrt{m})$. Compared to the other algorithms which guarantee bounds independent of the number of allocated time quanta, this result is not as good. However, the advantage of the lottery scheduling lays in its simplicity; since there is no need to update any global state, dynamic operations are easy and efficient to implement. In addition, by means of tickets, lottery scheduling introduces a powerful and flexible resource abstraction for resource management.

Maheshawari has proposed a new deterministic scheme called *charged-based* proportional scheduling [19]. The scheme is based on charging threads (clients) for processor usage. Threads are considered in a round-robin order and occasionally some of them are skipped to keep their usage close to the desired proportion. The client lag is bounded by $O(n_c)$, where n_c represents the number of active clients. Although the selection operation is fast on the average, its worst case complexity is $O(n_c)$. The time complexity of the other dynamic operations is also $O(n_c)$.

In an attempt to improve the lottery scheduling, Stoica and Wahab have developed a new deterministic algorithm for generating the winning numbers [26]. Although they have initially shown [26] that the algorithm guarantees that the client lag is bounded by $O(\log m)$, where m represents the number of allocated time quanta, more recent results have reduced this bound to $O(\log w)$, where w is the client weight [27]. The main drawback of this algorithm is that it does not support non-uniform quanta. On the other hand, as lottery scheduling, it supports dynamic operations efficiently, i.e., it requires only integer comparisons, additions and subtractions.

Recently, Waldspurger and Weihl have proposed a new algorithm called stride scheduling [29, 30]. The algorithm can be viewed as an extension and a cross-application of the network fair-queueing algorithms [31, 23] to the domain of processor scheduling. Stride scheduling relays on the concept of *global pass* (which is similar to virtual time) to measure the work progress in the system. Each client has associated a *stride* which is inversely proportional to its weight, and a *pass* which measures the progress of that client. The algorithm allocates a time quantum to the client with the lowest pass. After a client receives a time quantum, its pass is updated to its stride (notice that this is similar to computing the virtual deadline by Eq. (10), when the request duration is equal to a time quantum). The basic algorithm guarantees a client lag of $O(n_c)$, where n_c is the number of active clients. By grouping the clients in a binary tree, and recursively applying the basic stride scheduling algorithm at each level, Waldspurger and Weihl have further reduced the client lag to $O(\log n_c)$. As well as the basic algorithm, the hierarchical stride implements all the dynamic operations in $O(\log n_c)$.

Baruah, Gehrke, and Plaxton have developed a powerful proportional share scheduling algorithm for *multiple* resources, called PD (from "pseudo-deadlines") [2]. Given m identical copies of a resource the PD algorithm guarantees that the client lag is bounded by one time quantum. In the single resource case, the algorithm exhibits a $O(\log n_c)$ time-complexity for client selection, the same as EEVDF. While solving the proportional share scheduling problem for multiple resources, PD does not provide explicit support for dynamic operations, and therefore their implementation would require an expensive O(n_c) pre-processing phase. Moreover, the PD algorithm lacks support for fractional and non-uniform time quanta.

Recently, Baruah claimed new results that address the problem of supporting dynamic operations in the context of fixed time quanta [3]. The new algorithm implements client leaving and joining operations in $O(n_c)$ time for multiple resource case, and in $O(\log n_c)$ time for a single resource case. Also, when a client leaves the competition, the algorithm provides an $O(\log n)$ bound on the client lag. While these results improve the ones achieved by the PD algorithm, we note that for a single resource case, by using Strategy 3, the EEVDF algorithm achieves an $O(1)$ bound on the client lag, while providing support for both non-uniform and fractional time quanta (see Corollary 2).

8 Conclusions

We have described a new proportional share resource allocation scheduler that provides a flexible control, and provides strong timeliness guarantees for the service time received by a client. In this way we provide a unified approach for scheduling multimedia, interactive, and batch applications. We achieve this by uniformly converting the application requirements regardless of their type in a sequence of requests for the resource. Our algorithm guarantees that in steady conditions the difference between the service time that a client should receive in the idealized system and the service time it actually receives in the real system is bounded by one time quantum. This result improves the previous known bounds for proportional share allocation of both processor [29, 30] and communication bandwidth [23] in dynamic systems. The algorithm provides efficient support for dynamic operations such as client joining or leaving the competition (for the resource), and changing a client's weight. The scheduler also provides support for both fractional and non-uniform time quanta.

However, many other problems remain open. First, we would like to investigate higher level resource abstractions on top of EEVDF. An example, such an abstractions should provide inferior bounded service [26], i.e., a client should be able to specify a minimum fraction of the resource for which it is able to run. This feature is important in supporting both multimedia applications that require certain rate objectives and real-time applications that have to meet specified deadlines.

Second, it will be interesting to consider both hierarchical and heterogeneous scheduling schemes. As an example, consider the case in which the clients are grouped in several classes. Then, at a higher level a proportional scheduler may be used to allocate resource shares to each class, while at a lower level, among the clients in the same class, a simple scheduler (e.g. round-robin) may be used. As a second example, consider the process-thread hierarchy. Here, similarly, a proportional scheduler may be used to allocated CPU time slices to each process, and in turn, the process may select a thread to run by using a different policy. Ultimately, we think that this may be a valuable approach to implement flexible resource management in the new generations of operating systems [5, 10] in which the applications would be able to implement their own algorithms for managing the shares of the resources allocated to them.

We have conducted extensive simulations and we are currently in the final stages of implementing a prototype under FreeBSD Unix. The experimental results will be reported in a forthcoming paper.

9 Appendix A: Example: Implementation of Strategy 1

In this appendix we give the ANSI C code for Strategy 1. For clarity of presentation we use two distinct data structures: a client data structure (*client_struct*) and a request data structure (*req_struct*). However, we note that since any client has associated at most one pending request, in practice it is enough to use one common data structure including all the information. For simplicity, we assume that the duration of any request issued by a client is no greater than one time quantum (i.e., *QuantumSize*). The clients' pending requests are stored in a tree-based data structure *ReqTree* (see Appendix B for details) that supports the following operations: insertion (*insert_req*), deletion (*delete_req*), and finding the eligible request with the earliest virtual deadline (*get_req*). All these operations are implemented in $O(\log n)$, where n represents the number of pending requests. Notice that since any active client has exactly one pending request, n also represents the number of active clients.

When a client joins the competition (*join* function), the virtual time is updated according to Eq. (19), and the client issues its first request. The request's virtual eligible time and the virtual deadline are computed according to Eq. (9) and (10). Similarly, when the client leaves the competition (*leave* function), the virtual time is updated according to Eq. (18), and its request is removed from the tree.[14] Finally, when the weight of a client is changed (*change_weight*), the virtual time is updated according to Eq. (20).

The dispatch function (*EEVDF_dispatch*) finds the eligible request with the earliest deadline (by invoking *get_req*) and allocates the resource to the corresponding client. Since the request is assumed to take at most one time quantum, when the client releases the resource the request is already fulfilled, and consequently the client's lag is updated and a new request is issued. We note that if the duration of the request is greater than one time quantum, there is no need to issue a new request before the current one is fulfilled. In this case the *delete_req* and *issue_req* functions are called less frequently, which results in a lower overhead. However, as we have shown in Section 6 this comes at the expense of decreasing the allocation accuracy. The functions used to update the virtual time (*get_current_vt*) and the client lag (*update_lag*) are not given (their implementation based on the Eq. (3), (5), and (6) being straightforward).

```
/* client data structure */
typedef struct _client {
  int     lag;  /* client lag */
  void   *req;  /* pending request */
  ...
} client_struct;

/* request data structure */
typedef struct _req {
  time_v         ve;     /* virtual eligible time */
  time_v         vd;     /* virtual deadline */
  time_v         min_vd; /* variable used in augmented data structure */
  client_struct *client; /* client which has issued request */
  struct _req   *left, *right, *parent;
} req_struct;

/* global data */
req_struct *ReqTree     = NULL; /* request tree */
time_v      VirtualTime = 0;    /* virtual time */
int         TotalWeight = 0;    /* total weight of all active clients */
```

[14]Here we assume that the client lag is updated before the *leave* function is called

```
/* join competition */
void join(client_struct *client)
{
  /* update total weight of all active clients */
  TotalWeight += client->weight;

  /* update virtual time according to client lag */
  VirtualTime = get_current_vt() - client->lag/TotalWeight;

  /* issue request */
  client->req->ve = VirtualTime;
  client->req->vd = req->ve + QuantumSize/client->weight;
  insert_req(client->req);
}

/* leave competition */
void leave(client_struct *client)
{
  /* update total weight of all active clients */
  TotalWeight -= client->weight;

  /* update virtual time according to client lag */
  VirtualTime = get_current_vt() + client->lag/TotalWeight;

  /* delete request */
  delete_req(client->req);
}

/* change client weight */
void change_weight(client_struct *client, int new_weight)
{
  /* partial update virtual time according to client lag */
  VirtualTime = get_current_vt() + client->lag/(TotalWeight - client->weight);

  /* update total weight of all active clients */
  TotalWeight += new_weight - client->weight;

  /* update client's weight */
  client->weight = new_weight;

  /* update virtual time */
  VirtualTime -= client->lag/TotalWeight;
}

/* dispatch function */
void EEVDF_dispatch()
{
  int         used;
  /* get eligible request with earliest virtual deadline */
  req = get_req(ReqTree, get_curreent_vt());

  /* allocate resource to client with earliest eligible virtual deadline */
  used = allocate(req->client);

  /* update client's lag */
  update_lag(client, used);

  /* current request has been fulfilled; delete it */
  delete_req(ReqTree, req);

  /* issue new request */
  client->req->ve += used/client->weight;
  client->req->vd = client->req->ve + QuantumSize/client->weight;
  insert_req(client->req);
}
```

10 Appendix B. Request Data Structure

In this section we propose an efficient data structure to manage the requests initiated by active clients. The data structure is based on an augmented binary search tree and supports the following operations: insertion (*insert_req*), deletion (*delete_req*), and finding the eligible request with the earliest virtual deadline (*get_req*). Each node in the tree stores the virtual eligible time *ve*, and the virtual deadline *vd* of the request it represents. In addition, in *min_vd*, each node maintains the minimum virtual deadline among all of the nodes in its subtree (a node's subtree consists of the node itself and all its descendants). The virtual eligible time *ve* is used as a key in the binary search tree. Figure 4 depicts a simple instance of our data structure containing 6 nodes. As an example, the information associated with the root (node n_1) is: $ve = 10$, $vd = 25$, and $min\_vd = 15$ (notice that, in this case, the *min_vd* represents the minimum virtual deadline among all the nodes in the tree).

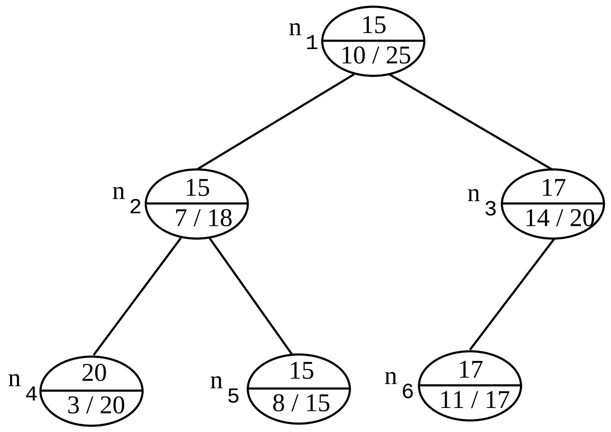
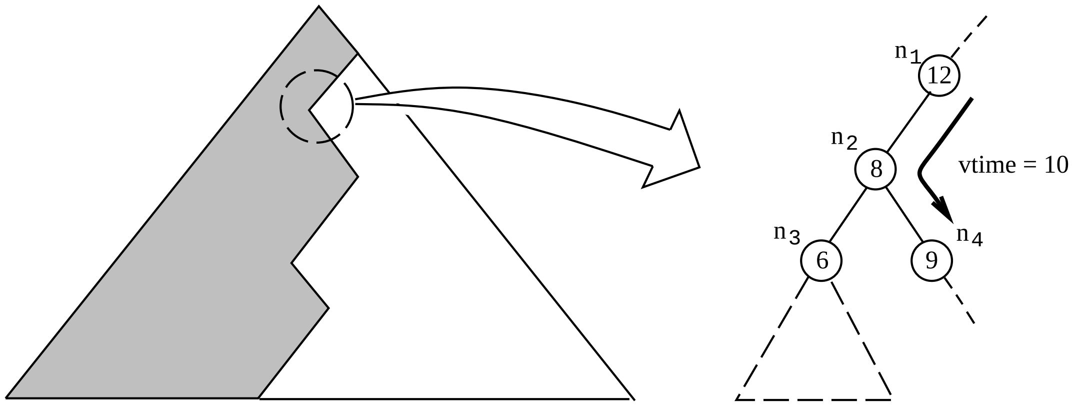


Figure 4: *The pending requests are stored in an augmented binary search tree. Each node maintains two state variables associated to the request it represents: the virtual eligible time (which is also the key in the tree) and the virtual deadline. In addition each node maintains the minimum among all the virtual deadlines stored in the nodes of its subtree.*

The remainder of this section describes the details of the three basic operations: *get_req*, *insert_req*, *delete_req*.

10.1 Searching

Figure 5 shows the ANSI C code of the *get_req* function. The function accepts as input the current virtual time *vtime*, and returns the eligible request with the earliest deadline. The algorithm starts at the root and goes down the tree as follows: whenever *vtime* is larger than the eligible time of the current node (*node*), the algorithm selects the right child as the next current node, otherwise the algorithm selects the left child. As a result (see Figure 5(a)) we obtain a path from root to one of the leaves which partitions the tree in two: all the nodes in the left (shaded) partition represent eligible requests, while all the nodes in the right partition represent requests that are not eligible yet. Notice that to obtain the entire set of eligible requests, besides the nodes in the left partition, we need to consider also the eligible nodes along the searching path. Thus to find the earliest eligible request it is enough to search only along the path and among the nodes in the left partition.



(a)

```
/* get eligible request with earliest deadline */
req_struct *get_req(req_struct *root, time_v vtime)
{
  req_struct *node = root, *st_tree = NULL, *path_req = NULL;

  while (node) {
    if (node->ve <= vtime) {

      /* update node with earliest deadline along path. */
      if ((!path_req) || (path_req->vd > node->vd))
        path_req = node;

      /* update root of subtree containing earliest deadline */
      if ((!st_tree) || (node->left && st_tree->min_vd > node->left->min_vd))
        st_tree = node->left;

      node = node->right;
    } else
      node = node->left;
  }

  /* check whether node with earliest deadline was along path */
  if ((!st_tree) || (st_tree->min_vd >= path_req->vd))
    return path_req;

  /* return node with earliest deadline from subtree */
  for (node = st_tree; node; ) {
    /* if node found, return it */
    if (st_tree->min_vd == node->vd)
      return node;
    if (node->min_vd == node->left->min_vd)
      node = node->left;
    else
      node = node->right;
  }
}
```

(b)

Figure 5: *(a) The searching path in a request tree (for each node, only the virtual eligible time is shown). (b) The code for searching the node with the earliest eligible deadline.*

Next, notice that if *vtime* is larger than the eligible time of the current node, then the current node and all its left descendants are eligible (the set of left descendants consists of all nodes belonging to the subtree of the node's left child). For example, in Figure 5(a), node n_2 and all the descendants of node n_3 are eligible. Further, it is easy to see that all these subtrees are disjoint, and their union covers the entire left partition. Thus, since the root of each of these subtrees maintains the smallest deadline among all its descendants in its *min_vd* field, for finding the node with the earliest deadline in the left partition it is enough to search only among the descendants of the root of the subtree with the smallest *min_vd*.

The searching algorithm updates two main variables: *path_req*, which maintains the eligible node with the earliest deadline on the searching path, and *st_tree*, which maintains the root of the subtree with the smallest *min_vd* in the left partition. After the final values of these variables are computed, clearly the eligible node with the earliest deadline is either *path_req*, or it belongs to *st_tree*. Thus, if the node's virtual deadline *vd* is smaller than the *min_vd* of *st_tree*, then *path_req* represents the request with the earliest deadline, and consequently the algorithm returns it. Otherwise, if the node with the earliest deadline belongs to *st_tree*, the algorithm identifies and returns it (this task is performed by the last 10 lines of the code in Figure 5(b)).

```
/* insert new request */
void insert_req(req_struct *root, req_str *req)
{
  req_struct *node = root;

  while (node) {

    /* update min_vd of current node */
    if (node->min_vd > req->vd)
      node->min_vd = req->vd;

    if (node->ve < req->ve) {
      if (!node->right) {
        /* insert new request as right child */
        node->right = req;
        req->parent = node;
        return;
      }
      node = node->right;
    } else {
      if (!node->left) {
        /* insert new request as left child */
        node->left = req;
        req->parent = node;
        return;
      }
      node = node->left;
    }
  }
}
```

Figure 6: *The code for insertion.*

10.2 Insertion

The code for inserting a new request (*insert_req* function) is straightforward. The only difference from the classical insertion algorithm for binary search trees [7] is that all the ancestors of the new node eventually need to update their *min_vd* field. As shown in Figure 6, this is simply done by updating (the *min_vd* of) all nodes which have the *min_vd* larger than the virtual deadline of the new node on the path from root to the place where the node is to be inserted.

10.3 Deletion

As shown in Figure 8(a), the deletion algorithm considers three cases depending on whether the node to be deleted is (i) a leaf, (ii) an internal node with only one child, or (iii) an internal node with two children. As in the case of insertion, the main difference from the standard algorithm (see [7]) consists in updating the *min_vd* fields of the ancestors of the deleted node, when needed. For example, in the tree from Figure 4 if node n_5 is deleted, then its ancestors n_2 and n_1 should update their *min_vd* field to 18.

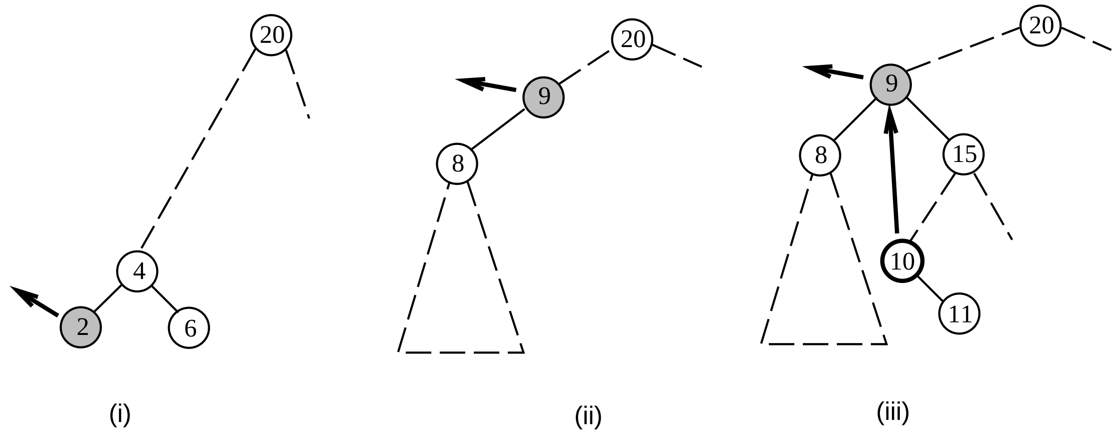
```
/* update min_vd fields of all ancestors of node */
void backward_update(req_struct *node, req_struct *root)
{
  do {
      node->min_vd = node->vd;
      /* check for right child */
      if (node->right && node->min_vd > node->right->min_vd)
        node->min_vd = node->right->min_vd;
      /* check for left child */
      if (node->left && node->min_vd > node->left->min_vd)
        node->min_vd = node->left->min_vd;
   } while ((node != root) && (node = node->parent))
}
```

Figure 7: *The code for the backward update function.*

In our algorithm this task is performed by the *backward_update* function (see Figure 7). This function takes as input a node and the root of the tree the node belongs to and updates the *min_vd* fields of all the nodes on the path from that node to the root (i.e., its ancestors). The function starts from the given node and iteratively advances to the root. At each level, it sets the *min_vd* of the current node to the minimum between the node's virtual deadline and the *min_vd*'s of its children. In this way the modification propagates recursively to the root. To see how this procedure works, consider the situation in which node n_5 is removed from the tree in Figure 4. In this case, we invoke the *backward_update* function with the following parameters: the parent of the deleted node n_2 and the root n_1. Since after the deletion of n_5, n_2 remains with only one child n_4, the *min_vd* of n_2 is set to the minimum between its virtual deadline *vd* and the *min_vd* of n_4, i.e., 18. Further, at the next level, the *min_vd* of n_1 is set to the minimum between its virtual deadline and the *min_vd*'s of its children, i.e., 17.

Figure 8 shows the code of the *delete_req* function. The algorithm takes as input the *root* of the request tree and the node to be deleted (*node*). In the first two cases ((i) and (ii)) the node is simply removed and its parent inherits its child, if any. Once this operation is accomplished, the algorithm updates the *min_vd*s' of the *node*'s ancestors by calling the *backward_update* function. The case (iii) is a bit more complex (for a complete description, see [7]). First, the algorithm finds the successor[15] of the node to be deleted (*succ*) and removes it from the tree by recursively calling the *delete_req* function. Since the successor of *node* is the leftmost node among its right descendants, it follows that the successor

[15]In a binary search tree, the successor of a node n is defined as the node with the smallest key greater than n's key. For example, in Figure 4, the successor of the root is node n_6, which has the key 11.



(a)

```
/* delete specified request */
req_struct *delete_req(req_struct *root, req_struct *req)
{
  req_struct *child, *succ;

  if (!req->right || !req->left) { /* cases (i) and (ii) */

    /* get node's child, if any */
    child = get_child(req);

    /* remove node form tree */
    if (child)
      child->parent = node->parent;

    if (req == root)
      return child;

    if (req->parent->left == req)
      req->parent->left = child;
    else
      req->parent->right = child;

    /* update min_vd of node's ancestors */
    backward_update(parent, root);
  } else { /* case (iii) */
    /* get node's successor */
    succ = get_succesor(req);

    /* delete node's successor from tree */
    req->right = delete_req(req->right, succ);

    /* swap node with its successor */
    swap_node(succ, req);

    /* update min_vd of node's ancestors */
    backward_update(req, root);
  }
  return root;
}
```

(b)

Figure 8: *(a) The generic cases for deletion. (b) The code for deletion.*

can have at most one right child. Therefore the *delete_req* function is called recursively *exactly* once. For efficiency this function operates only on the tree containing *node*'s right descendants and not on the entire tree. In this way it updates only the *min_vd* fields of the right descendants of *node*. Finally, the algorithm replaces *node* with its successor, and updates the *min_vd* fields of all its ancestors by calling the *backward_update* function.

From Figures 5(b) 6, and 8(b), it is easy to see that that all the algorithms spend a constant amount of time at each level of the tree (recall that in the case of *delete_req*, although we have a recursive call, this call may occur at most once). Thus, for a tree of height h, the time complexity of *get_req*, *insert_req*, and *delete_req* functions is $O(h)$.

10.4 Balanced Trees

Although a binary search tree provides good performances on the average [7], in the worst case (when the tree degenerates) it exhibits an $O(n)$ time complexity for insertion, deletion, and searching operations, where n represents the number of nodes of the tree. The usual solution to address this problem is to use one of the existing balanced data structures, such as red-black [7], or AVL trees [24]. The procedure to balance the tree after an insertion or deletion is based on two types of operations: left and right rotations. Figure shows the code to update the *min_vd* fields for the left rotation (the code for the right rotation being similar). Since a rotation clearly takes constant time, and since as shown in [7] the time complexity to balance a tree with n nodes is $O(\lg n)$, the overall time complexity of *get_req*, *insert_req* and *delete_req* functions is also $O(\log n)$.

```
/* left rotation */
void left_rotation(req_struct *node)
{
  /* update min_vd fields */
  left->min_vd = node->min_vd;
  node->min_vd = min(node->vd, left->right->min_vd);
  node->min_vd = min(node->min_vd, node->left->min_vd);

  /* perform rotation */
  ...
}
```

Figure 9: *The code for updating node's state for a left-rotation rotation.*

References

[1] M. J. Bach. "The Design of the Unix Operating System," Prentice-Hall, 1986.

[2] S. K. Baruah, J. E. Gehrke and C. G. Plaxton, "Fast Scheduling of Periodic Tasks on Multiple Resources", *Proc. of the 9th International Parallel Processing Symposium*, April 1995, pp. 280–288.

[3] S. K. Baruah, *Personal Communications*, June 1995.

[4] G. Bollela and K. Jeffay, "Support for Real-Time Computing Within General Purpose Operating Systems", *Proc. of the IEEE Real-Time Technology and Applications Symposium*, Chicago, May 1995.

[5] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers and C. Chambers, "Extensibility, Safety and Performance in the SPIN Operating System", *SOSP-15*, 1995.

[6] R. L. Cruz, "A calculus for network delay. Part I: Network elements in isolation", *IEEE Transaction on Information Theory*, vol. 37, pp. 114–131, 1991.

[7] T. H. Cormen, C. E. Leiserson and R. L. Rivest. "Introduction to Algorithms," MIT Press, 1992.

[8] G. Coulson A. Campbell, P. Robin, G. Blair, M. Papathomas and D. Hutchinson, "The Design of a QoS Controlled ATM Based Communication System in Chorus", *Internal Report* MPG-94-05, Department of Computer Science, Lancaster University, 1994.

[9] A. Demers, S. Keshav, and S. Shenkar, "Analysis and Simulation of a Fair Queueing Algorithm", *Proc. of ACM SIGCOM'89*, September 1989, pp. 1–12.

[10] D. R. Engler, M. Frans Kaashoek and J. O'Toole Jr, "Exokernel: An Operating System Architecture for Application-Level Resource Management", *SOSP-15*, 1995.

[11] L. L. Fong and M. S. Squillante, "Time-Function Scheduling: A General Approach for Controllable Resource Management", Working Draft, IBM T.J. Watson Research Center, New York, March 1995.

[12] S. J. Golestani, "A Self-Clocked Fair Queueing Scheme for Broadband Applications", *Proc. of IEEE INFOCOM'94*, April 1994, pp. 636–646.

[13] J. L. Hellerstein. "Achieving Service Rate Objectives with Decay Usage Scheduling," *IEEE Transactions on Software Engineering*, Vol. 19, No. 8, August 1993, pp 813-825.

[14] K. Jeffay and D. Bennett, "A Rate-Based Execution Abstraction for Multimedia Computing", *Proc. of the Fifth International Workshop on Network and Operating System Support for Digital Audio and Video*, Durham, Appril 95.

[15] J. Kay and P. Lauder. "A Fair Share Scheduler," *Communication of the ACM*, Vol. 31, No. 1, January 1988, pp 44-45.

[16] S. J. Leffler, M. K. McKusick, M. J. Karels and J. S. Quarterman. "The Design and Implementation of the 4.3BSD UNIX Operating System," Addison-Wesley, 1989.

[17] J. P. Lehoczky, L. Sha and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Behaviour", *Proc. of the IEEE Tenth Real-Time Systems Symposium*, December 1989, pp. 166–171.

[18] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *Journal of the ACM*, vol. 20, no. 1, January 1973, pp. 46–61.

[19] U. Maheshwari, "Charged-based Proportional Scheduling", Technical Memorandum MIT/LCS/TM-529, Laboratory for CS, MIT, July 1995.

[20] C. W. Mercer, S. Savage, and H. Tokuda "Processor Capacity Reserves: An Abstraction for Managing Processor Usage", *Proc. of the Fourth Workshop on Workstation Operating Systems*, October 1993

[21] C. W. Mercer, S. Savage, and H. Tokuda, "Processor Capacity Reserves: Operating System Support for Multimedia Applications", *Proc. of the IEEE International Conference on Multimedia Computing and Systems*, May 1994

[22] J. Nieh, M. S. Lam, "Integrated Processor Scheduling for Multimedia", *Proc. of the Fifth International Workshop on Network and Operating System Support for Digital Audio and Video*, Durham, N.H., April, 1995.

[23] A. K. Parekh and R. G. Gallager, "A Generalized Processor Sharing Approach To Flow Control in Integrated Services Networks-The Single Node Case", *Proc. of IEEE INFOCOM'92*, 1992, pp. 915–924.

[24] J. D. Smith, "Design and Analysis of Algorithms", PWS-KENT Publishing Company, Boston, 1989.

[25] A. Silberschatz and P. B. Galvin. "Operating Systems Concepts," - fourth edition, Addison-Wesley, 1994.

[26] I. Stoica, H. Abdel-Wahab, "A new approach to implement proportional share resource allocation", *Technical Report* TR-95-05, CS Dpt., ODU, April 1995.

[27] I. Stoica, *Work in progress*, CS Dpt., ODU, 1995.

[28] C. A. Waldspurger and W. E. Weihl. "Lottery Scheduling: Flexible Proportional-Share Resource Management," *Proc. of the First Symposium on Operating System Design and Implementation*, November 1994, pp. 1–12.

[29] C. A. Waldspurger and W. E. Weihl. "Stride Scheduling: Deterministic Proportional Share Resource Menagement," Technical Memorandum, MIT/LCS/TM-528, Laboratory for CS, MIT, July 1995.

[30] C. A. Waldspurger. "Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management," *PhD Thesis*, Laboratory for CS, MIT, September 1995.

[31] L. Zhang, "VirtualClock: A New Traffic Control Algorithm for Packet-Switched Networks", *ACM Transactions on Computer Systems*, vol. 9, no. 2, May 1991, pp. 101–124.