Exact GPS Simulation and Optimal Fair Scheduling with Logarithmic Complexity

Paolo Valente
Dipartimento di Ingegneria dell'Informazione
Università degli Studi Di Modena - Italy
paolo.valente@unimore.it

***Abstract*— Generalized Processor Sharing (GPS) is a fluid scheduling policy providing perfect fairness over both constant-rate and variable-rate links. The minimum deviation (lead/lag) with respect to the GPS service achievable by a packet scheduler is one maximum packet size. To the best of our knowledge, the only packet scheduler guaranteeing the minimum deviation is Worst-case Fair Weighted Fair Queueing (WF^{2}Q), which requires on-line GPS simulation. Existing algorithms to perform GPS simulation have $O(N)$ worst-case computational complexity per packet transmission (N being the number of competing flows). Hence WF^{2}Q has been charged for $O(N)$ complexity too. However it has been proven that the lower bound complexity to guarantee $O(1)$ deviation is $\Omega(\log N)$, yet a scheduler achieving such a result has remained elusive so far.**

In this paper we present L-GPS, an algorithm that performs exact GPS simulation with $O(\log N)$ worst-case complexity and small constants. As such it improves the complexity of all the packet schedulers based on GPS simulation. We also present L-WF^{2}Q, an implementation of WF^{2}Q based on L-GPS. L-WF^{2}Q has $O(\log N)$ complexity with small constants, and, since it achieves the minimum possible deviation, it does match the aforementioned complexity lower bound. Furthermore, both L-GPS and L-WF^{2}Q comply with constant-rate as well as variable-rate links. We assess the effectiveness of both algorithms by simulating real-world scenarios.

***Index Terms*— Computational Complexity, Data Structures, Packet Scheduling, Quality of Service.**

I. Introduction

Given a set of N flows, defined in whatever meaningful way and sharing a common transmission link, packet scheduling algorithms play a critical role in providing each flow with a predictable service.

An important reference system in packet scheduling is the Generalized Processor Sharing (GPS) server [1]. Provided that each flow has a weight assigned to it, such a system serves all flows *simultaneously*, delivering each one a service rate proportional to its weight. The GPS service discipline is not realistic: a practical system can serve a limited number of packets at a time (in this paper we consider only systems that can serve at most one packet at a time). Nevertheless, thanks to its perfectly fair allocation, the GPS service discipline is used as a reference model for evaluating the properties of more practical schedulers.

It is easy to prove that the fairness of a packet scheduler depends on its maximum per-flow *deviation* (difference) with respect to the amount of service delivered by the GPS server. In particular, $O(N)$ deviation implies $O(N)$ (un)fairness, whereas $O(1)$ deviation guarantees $O(1)$ (un)fairness (N can be quite large, as shown in [21] and discussed in more detail below).

Furthermore, as shown in [3] and [5], a scheduler with $O(N)$ deviation with respect to the GPS service may introduce *bursts* a *bursting* period, during which up to $O(N)$ packets belonging to the same flow are served back-to-back, can be followed by a *silence* period – with length equal to the preceding bursting period – during which no packet of the the flow is served.

Since packet transmission is atomic, no packet scheduling algorithm can avoid a *minimum deviation*, equal to one maximum size packet, between the amount of service provided to each flow by the real system and the amount of service provided to the same flow by the GPS server. We say that the service delivered by a real system (thanks to the adopted scheduling policy) is *optimum*, if the discrepancy with respect to the GPS service never exceeds the *minimum* deviation. It has been proven that the *lower* bound complexity to guarantee $O(1)$ deviation with respect to the GPS service is $\Omega(\log N)$ [13].

A very accurate packet scheduling algorithm, called Worst-case Fair Weighted Fair Queueing (WF^{2}Q) [3] and based on the on-line simulation of a GPS server, does achieve the optimum service. The *classical* algorithm for simulating the GPS server has been proposed more than a decade ago together with the GPS service discipline itself [1]. It has been proven to require – in the worst-case – the processing of $O(N)$ events in a single packet transmission time [9]. For this reason WF^{2}Q has been charged for $O(N)$ complexity too [9], [6], [5].

Another important measure of the cost of the GPS simulation is the number of steps performed for each arriving packet, the *per packet* complexity. This complexity has been systematically studied for the first time in [15]. The authors showed that its lower bound is $\Omega(\log N)$, and that an algorithm matching this bound was already proposed in [14]. But they also clarified that: 1) if a *heap*-type priority queue is used to implement this algorithm, the worst-case complexity per packet transmission time is still $\Omega(N)$, 2) this lower bound is due to an unavoidable problem referred to as the *mandatory lazy evaluation* problem (see Subsec. III-A). In this paper we show how, despite this problem, $O(\log N)$ complexity per packet transmission time can be achieved by using a *hierarchical tree*-type data structure.

The computational complexity of a packet scheduler is a critical issue, because links transmit packets at increasingly higher speeds, and the number of competing flows can be quite high. As reported in a recent work [21], tens of thousands flows can be *in progress* at the same time through an Internet link (in [21] a flow is denoted as in progress during any time interval in which the inter-arrival time of its packets is lower than 20 seconds).

However, in the same paper it is shown that the number of simultaneously backlogged flows at any time instant is in the order of a few hundreds under stable load conditions. With such figures, linear complexity may constitute a significant barrier to on-line scheduling in high speed applications [20], [9], [6], [5], [10], [11]. On the contrary, depending on the constants, logarithmic complexity per packet transmission time may be affordable.

Many scheduling algorithms with $O(\log N)$ complexity have been proposed, such as Self Clocked Fair Queueing (SCFQ) [9], Frame Based Fair Queueing (FFQ) [6] and Start Time Fair Queueing [10]. They are based on an *approximate* simulation of the GPS server, trading accuracy for complexity. Unfortunately, all them exhibit $O(N)$ deviation with respect to the GPS service.

A more accurate algorithm, called Worst-case Fair Weighted Fair Queueing Plus (WF^{2}Q+) [5], has been proposed to reduce the implementation complexity of WF^{2}Q while retaining several of its properties (a similar, but not identical, algorithm has been proposed in [7]). WF^{2}Q+ has $O(1)$ deviation from the *minimum* amount of service guaranteed to each flow by the GPS server. However, also WF^{2}Q+ may exhibit $O(N)$ deviation from the *actual* service delivered by the GPS server when some flows are idle [22].

Finally, several schedulers with very low complexity (ranging from $O(1)$ to $O(\log\log N)$) have been proposed [11], [8], [12], [19], but all of them exhibit $O(N)$ or, worse yet, unbounded deviation with respect to the GPS service. In the end, even though the *lower* bound complexity to guarantee the optimum service has been proven to be $\Omega(\log N)$ [13], the problem of providing $O(1)$ deviation from a perfectly fair service with sub-linear complexity was still open.

Contributions of this paper

In this paper we present Logarithmic-GPS (L-GPS), an algorithm for simulating a GPS server, based on a specially augmented balanced binary tree. Such a tree allows the state of the simulated GPS server to be computed with $O(\log N)$ complexity at any time instant. The tree must be updated *only* at each packet arrival, and at $O(\log N)$ cost.

Actually, the number of operations needed to compute the state of the GPS server and to update the tree is proportional to the depth of the tree itself, which in its turn can be implemented by *augmenting*, in the sense defined in [17] (Chapter 14), an underlying balanced binary tree. In this paper we show two possible implementations, based, respectively, on Patricia Trees [16], which guarantee $O(\log N)$ *average* depth, and on Red-black Trees [17], which guarantee $O(\log N)$ *worst-case* depth. Especially, although providing a weaker theoretical complexity bound, Patricia Trees have a much simpler structure and allow L-GPS to be implemented in a more efficient way than Red-black Trees. As we show through simulations, they achieve good performance in practical cases. In the end, depending on the specific balanced tree used, L-GPS enables the GPS service to be simulated at $O(\log N)$ – statistical or deterministic – cost per packet transmission/arrival.

We also present Logarithmic-WF^{2}Q (L-WF^{2}Q), an implementation of WF^{2}Q based on L-GPS, with $O(\log N)$ complexity and small constants. To the best of our knowledge, L-WF^{2}Q is the first scheduler with $O(\log N)$ complexity achieving $O(1)$ deviation (actually, the minimum possible deviation) with respect to the GPS service.

Both L-GPS and L-WF^{2}Q comply with constant-rate as well as variable-rate links. As an example of the second category, consider shared-media wired or wireless links. Typically, only the MAC protocol is concerned with collisions and packet losses, and it hides these details to layers 3 and above. So the latter just 'see' a time-varying capacity link.

L-GPS and L-WF^{2}Q reduce the *upper* bound complexity for simulating a GPS server and for providing the optimum service, both from $O(N)$ to $O(\log N)$. Moreover, since $\Omega(\log N)$ is the *lower* bound complexity to guarantee $O(1)$ deviation from the GPS service [13], L-WF^{2}Q achieves the *optimum* service with *optimum* complexity.

Part of the material presented in this paper appeared for the first time in a former work [22].

Organization of this paper

This paper is organized as follows. In Sec. II we provide an overview of GPS and WF^{2}Q. In Sec. III we make a survey of related work, focusing on the existing linear complexity algorithms for simulating the GPS server, and on the WF^{2}Q+ packet scheduler. In Sec. IV we present our main result, the L-GPS algorithm, whereas in Sec. V we discuss how it can be implemented using two classes of balanced trees. In Sec. VI we describe L-WF^{2}Q. In Sec. VII we show through simulations how the actual complexity of L-GPS and L-WF^{2}Q compares to the worst-case bound.

II. GPS AND WF^{2}Q

Consider a system in which N flows (defined in whatever meaningful way) share a common transmission link with a *time-varying* capacity (rate) of $C(t)$ bits/sec. We define $W(t) \equiv \int_0^t C(\tau)\cdot d\tau$ as the total amount of service provided by the system during $[0,\, t]$. We say that a packet *has arrived* in the system when its last bit has arrived in the system, we call packet *arrival time* the time at which this happens. Similarly, we say that a packet *departs* from the system when its last bit is transmitted by the system, and we call packet *finish time* the time at which this happens. We define as *backlogged* every flow owning packets not yet (completely) transmitted. Each flow has a packet FIFO queue associated with it, holding the flow's own backlog.

We define *busy period* a maximal interval of time during which the system is never idle. Finally, most of the notations used in this paper are summarized in Table I.

Each flow i has a positive number ϕ_i assigned to it, namely its *weight*. A GPS server [1] is an ideal system that serves all backlogged flows simultaneously, providing each of them a *share* of the output link capacity (i.e. ratio between the service rate provided to the flow and the link capacity), proportional to its weight. In formulas:

$$dW_i(t) = \frac{\phi_i}{\sum_{j\in B(t)}\phi_j}\cdot dW(t) = \frac{\phi_i}{\Phi(t)}\cdot dW(t)\ \forall i \in B(t) \quad (1)$$

L_{max}	Maximum packet length
ϕ_i	Weight of the *i-th* flow
$\Phi(t) \equiv \sum_{j\in B(t)} \phi_j$	Sum of the weights of the flows backlogged at time t
$S_i(t)$, $F_i(t)$, $U_i(t)$	Virtual start/finish/unbacking time of the *i-th* flow at time t
Quantities related to a generic node of the U_{tree}:	
t_{min} , t_{max}	Extremes of the time interval $[t_{min}, t_{max}]$ represented by the node
U_{max}	$U_{max} \equiv V(t_{max})$
$\Delta\Phi$	$\Delta\Phi \equiv \Phi(t^+_{max}) - \Phi(t^-_{min})$
ΔW	Correction factor to use in (5), computed as in (6)

TABLE I
NOTATIONS USED IN THIS PAPER.

where $dW(t) = C(t){\cdot}dt$ is the total amount of service provided by the system in $[t, t + dt]$ ($C(t)$ is the link capacity at time t), $dW_i(t)$ is the amount of service received by the *i-th* flow in $[t, t + dt]$, $B(t)$ is the set of the flows backlogged at time t, $\Phi(t) \equiv \sum_{j\in B(t)} \phi_j$ is the sum of the weights of the flows backlogged at time t.

Given the packet arrival pattern and the output link capacity of a real system, WF^{2}Q [3] is based on the on-line simulation of the *corresponding* GPS server, i.e. a GPS server with the same arrival pattern and the same capacity of the real system. We say that a packet is *eligible* if it has already started service in the corresponding GPS server. WF^{2}Q implements the following scheduling policy: at each time instant t in which the link is ready to transmit the next packet, choose, among all the eligible packets, the next one that finishes in the corresponding GPS server, if no packet arrives after time t.

A practical way for implementing this policy in case of variable-rate links is based on timestamping packets with the values assumed by the following function, called (*GPS*) *system virtual time* [5]:

$$V(t) \equiv \int_0^t \frac{1}{\Phi(\tau)} \cdot dW(\tau) \tag{2}$$

From (1), we have that $dV(t) = \frac{dW_i(t)}{\phi_i}$ $\forall i \in B(t)$, i.e. the variation of the system virtual time during $[t, t+dt]$ is equal to the *normalized* amount of service received by each backlogged flow during the same time interval. Each packet p_i^k ($k - th$ packet of $i - th$ flow, in order of arrival times) is associated with a packet *virtual start time* S_i^k and a packet *virtual finish time* F_i^k. S_i^k is the value assumed by the system virtual time when the corresponding GPS server starts servicing p_i^k, and F_i^k is the value assumed by the system virtual time when the corresponding GPS server finishes servicing p_i^k. Suppose p_i^k arrives at time a_i^k and its length is equal to L_i^k, it is easy to prove that its timestamps can be computed as follows [5]:

$$\begin{aligned} S_i^k &= \max(V(a_i^k),\, F_i^{k-1}) \\ F_i^k &= S_i^k + \tfrac{L_i^k}{\phi_i} \end{aligned} \tag{3}$$

At every time, only the packets at the head of the queues of the backlogged flows can be chosen for transmission, hence, as suggested in [5], it is possible to schedule packets on a per-flow basis, and to maintain only a pair of timestamps for each flow i.

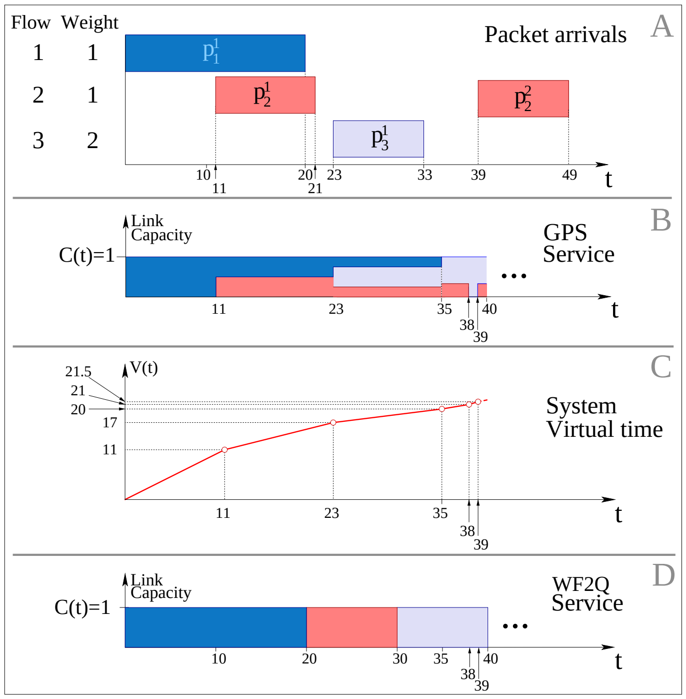

Fig. 1. Evolution of the system virtual time.

They are called, respectively, flow i virtual start time $S_i(t)$ and flow i virtual finish time $F_i(t)$, and correspond to the virtual start and finish time of the packet at the head of the queue of flow i at time t. Since the system virtual time is an increasing function of the time, it is easy to verify that the packet at the head of the $i - th$ flow is eligible at time t if and only if its virtual start time is no greater than $V(t)$. Accordingly, we say that a flow i is eligible at time t if and only if $S_i(t) \leq V(t)$.

We can now define WF^{2}Q as follows:

Definition 1: Each time the link is ready to transmit the next packet, WF^{2}Q picks the packet at the head of the queue of the eligible flow with the smallest virtual finish time.

The *maximum* per-flow deviation with respect to the corresponding GPS server guaranteed by WF^{2}Q is equal to the maximum packet length L_{max} [3] (WF^{2}Q delivers the *optimum* service). The computational complexity of WF^{2}Q is due to two major tasks: maintaining the set of the eligible flows sorted by virtual finish times, and computing the value of the system virtual time. As shown in [4] and briefly reported in Sec. VI, it is possible to maintain the eligible flows sorted by virtual finish times at $O(\log N)$ cost per packet arrival or departure. With regard to the latter task, existing algorithms with linear complexity for tracking the virtual time are presented in Sec. III. To describe both these algorithms and L-GPS we refer to the following example and definitions.

Example 1: Consider a link with a constant capacity C of 1 byte per time unit, shared by three packet flows. Flows 1 and 2 have weight 1, while flow 3 has weight 2. Fig. 1.A depicts a possible packet arrival pattern. Each arriving packet is depicted as a rectangle: the projection on the x axis of its left corner represents the packet arrival time, while the length of the base represents the time needed to serve the packet at full link capacity. Fig. 1.B shows the service delivered by the corresponding GPS server, Fig. 1.C shows the evolution of the system virtual time, Fig. 1.D shows the service provided by WF^{2}Q.

According to (2), at all times t, the *slope* of $V(t)$ against $W(t)$ is equal to $\frac{1}{\Phi(t)}$. Hence $V(t)$ is a piecewise linear function of the total amount of service $W(t)$ delivered by the system. In case of constant-rate links it is a piecewise linear function of the time too (Fig. 1.C). Hereafter, we use the term *slope* as a short for the slope of $V(t)$ against $W(t)$. Whenever $B(t)$ changes, $\Phi(t)$ and hence the slope of $V(t)$ changes, constituting a *breakpoint* in its piecewise linear form (with respect to $W(t)$). We define *break instant* every time instant $\bar{t}$ at which the slope changes (e.g. time 23 in Fig. 1.C), and *break value* the value assumed by the system virtual time at time $\bar{t}$.

We define the tuple $< V(t),\, \Phi(t^+),\, W(t) >$ as the *state* of the GPS server corresponding to the time instant t, and we say *computing the state* of the GPS server as a short for computing all the values of this tuple. Finally, given a generic function f of the time, we use the compact notations $f(t^-) = \lim_{x \to t^-} f(x)$ and $f(t^+) = \lim_{x \to t^+} f(x)$.

III. Related Work

The two main issues related to the GPS service are how to efficiently simulate it, and how to approximate it on a real system. With regard to the former issue, we present in Subsec. III-A the only two existing algorithms (according to the literature and excluding L-GPS) for tracking the system virtual time. With regard to the latter issue, in Subsec. III-B we focus on WF^{2}Q+, the only low complexity scheduler – $O(\log N)$ per packet transmission – achieving the same minimum service guarantees as WF^{2}Q.

A. Existing algorithms for tracking the virtual time

In a work-conserving scheduler, such as WFQ or WF^{2}Q, busy periods in the real system and in the corresponding GPS server coincide. Moreover, since packets arrive and are timestamped only during (or at the beginning of) busy periods, there is no need to compute the virtual time outside busy periods. Hence in what follows we consider the problem of computing $V(t_{new})$ at a generic time instant t_{new} belonging to a busy period for the GPS server. However, according to (2), the value of the virtual time is constant between two consecutive busy periods. By exploiting this property, all the algorithms described in this paper can be easily extended to compute the virtual time also at a time instant not belonging to a busy period.

Define $t_l \le t_{new}$ as the largest break instant no greater than t_{new}. $\Phi(t_l^+) > 0$ and the slope of $V(t)$ is constant and equal to $\frac{1}{\Phi(t_l^+)}$ during $(t_l,\, t_{new}]$. Hence, according to (2)

$$V(t_{new}) = V(t_l) + \frac{W(t_{new}) - W(t_l)}{\Phi(t_l^+)} \qquad (4)$$

As a consequence, if the state $< V(t_l),\, \Phi(t_l^+),\, W(t_l) >$ of the GPS server corresponding to time t_l is known, then (4) can be immediately applied to compute $V(t_{new})$.

The *classical* algorithm [1] for computing the virtual time can be defined as follows: store the state of the GPS server in three (scalar) *state variables*, and update them to $< V(t_j),\, \Phi(t_j^+),\, W(t_j) >$ at *each* break instant t_j.

Hence, at any time instant t, the state variables contain the state of the GPS server corresponding to the largest break instant no greater than t. For this reason, (4) can be immediately applied to compute $V(t_{new})$ at any time t_{new}. Furthermore, the state variables themselves can be updated upon each break instant by exploiting (4).

Break instant frequency depends on the frequency of transition of flows in and out of the set $B(t)$. Since flows are served simultaneously, packet finish times in the GPS server can be arbitrarily slightly skewed. In the worst case, $O(N)$ finish times may fall in an *arbitrarily short* time interval, and hence in the smallest packet transmission time. It is worth noting that this may happen even if the packet arrival rate is bounded to $O(1)$ packets per time unit. For example, in Fig. 1.A packet arrival times are spaced by time intervals longer than the minimum packet service time (10 time units). Nevertheless, the slope of the system virtual time changes $O(N)$ times during the service of p_3^1 (Fig. 1.D). As a conclusion, the worst-case complexity of the classical algorithm is $O(N)$ per packet transmission.

In [15] it is shown how an early algorithm proposed in [14] can be used to realize a *queue-based variant* of the classical algorithm. This variant basically allows the updating of the state variable to be postponed. For ease of exposition we discuss here a simplified version of the algorithm, which we call *sequential algorithm* and which has a computational complexity no higher than the one of the original algorithm. Suppose that at time t_{new} the state variables contain the tuple $< V(t_{old}),\, \Phi(t_{old}^+),\, W(t_{old}) >$ corresponding to a time instant $t_{old} \le t_l$, whereas a special queue holds one element for each break instant in $(t_{old},\, t_l]$. Each element contains information which enables the state of the GPS server corresponding to the represented break instant to be computed at $O(1)$ cost, provided that the state of the GPS server upon the immediately preceding break instant is known. $V(t_{new})$ is computed as follows. First, the states of the GPS server corresponding to all the break instants in $(t_{old},\, t_l]$ are computed by *sequentially* visiting each element of the queue. Once the state $< V(t_l),\, \Phi(t_l^+),\, W(t_l) >$ is computed, (4) is used to compute the state of the GPS server at time t_{new}. Finally, the latter is assigned to the state variables.

The sequential algorithm has an inherently linear worst-case complexity. According to what is previously stated, $O(N)$ break instants may in general fall in $(t_{old},\, t_{new})$, thus causing the algorithm to exhibit $O(N)$ worst-case complexity per virtual time computation. The only possibility for computing $V(t_{new})$ at time t_{new} in less than $O(N)$ steps would be knowing, at time t_{new}, the state of the GPS server corresponding to a break instant $t_f \le t_l$ such that less than $O(N)$ break instants are included in $(t_f,\, t_l]$. One way to guarantee that the state corresponding to t_f is known at time t_{new} would be computing the state of the GPS server upon each break instant (which would imply $t_f = t_l$). But this would have $O(N)$ cost. On the contrary, it is easy to prove that, to guarantee that the state corresponding to t_f is known at time t_{new} without incurring $O(N)$ complexity, it is necessary be able to pre-compute the *expected* states of the GPS server corresponding to future *expected* break instants. The sequential algorithm can be easily extended to pre-compute *expected* states as well.

Unfortunately, the expected state of the GPS server corresponding to an expected break instant t_f cannot be finalized before time t_f, because every packet p_i^j arriving at a time instant $t_a < t_f$ may change the evolution of the virtual time during $(t_a, t_f]$. For example, flow i may become backlogged, thus causing the slope to change at time t_a (detailed examples of how the expected evolution changes in consequence of packet arrivals are reported in Subsec. IV-A). The authors of [15] refer to this issue as the "mandatory lazy evaluation" problem. Since $O(N)$ expected break instants may fall in $(t_a, t_f]$ and the sequential algorithm performs one step per break instant, maintaining the expected state corresponding to t_f would have $O(N)$ cost per packet arrival.

B. *WF^{2}Q+*

WF^{2}Q+ [5] implements the same packet timestamping (3) and selection policy (Def. 1) of WF^{2}Q, but it uses a simpler system virtual time function. WF^{2}Q+ has been defined assuming that flow weights are *normalized* so that $\sum_{i=1}^{N} \phi_i = 1$ holds. Under this hypothesis and assuming also that some admission policy is used, we define the weight ϕ_i of a flow i also as its *reserved fraction* of the link capacity. We define as *reserved service* of a flow during a given time interval, the amount of service that the flow should receive during the time interval, according to its reserved fraction (for simplicity we neglect the case where the weight of a flow changes over time). It has been shown in [5] that, thanks to the properties of its system virtual time function, WF^{2}Q+ (as WF^{2}Q) guarantees to each admitted flow, and over any time interval, the minimum possible worst-case lag (less than $2 \cdot L_{max}$) with respect to its reserved service.

According to (1), the GPS server provides each flow with *at least* its reserved service over any time interval. Especially, a backlogged flow may receive much more than its reserved service during any time interval in which not all the flows are backlogged. In [22] it is shown that, due to this fact, WF^{2}Q+ may exhibit $O(N)$ deviation with respect to the GPS service if not all the admitted flows are continuously backlogged.

The following considerations can be made on the actual impact of the above shortcoming. Suppose that an application reserves the desired capacity along the nodes traversed by its flows, and that it relies only on the reserved service. In this case, an $O(1)$ lag with respect to the reserved service constitutes the most important guarantee for the application, and an $O(N)$ deviation with respect to the GPS service should cause no relevant consequences.

Conversely, consider a reservation-free scenario, as e.g. the one envisaged in [21]. First, perfect fairness is a desirable service distribution for best-effort traffic. Second, $O(N)$ deviation with respect to the GPS service results in additional burstiness, i.e. service rate oscillations, introduced by the scheduler. To the best of this author's knowledge, there is no experimental work either showing that this is not an issue, or showing to which extent adaptive (such as video streaming) and feedback-based (such as tcp) applications may benefit from the smoothest possible service.

Finally, as far as the computational cost is concerned, the main difference between WF^{2}Q and WF^{2}Q+ is that the latter is based on a simpler system virtual time function. However, as WF^{2}Q, WF^{2}Q+ must maintain the set of the eligible flows sorted by virtual finish times. To the literature, the lowest cost ($O(\log N)$) solution [4] to perform this task is provably more expensive than tracking the system virtual time of WF^{2}Q+ (more details are provided in Sec. VI). In the end the computational cost of (exact implementations of) WF^{2}Q and WF^{2}Q+ is comparable. In contrast, implementations of WF^{2}Q+ with $O(1)$ overall complexity have been devised [19] using approximate timestamps.

IV. L-GPS

In this section we concentrate on the GPS simulation effort, and we consider the following pair of systems: a real system and the *corresponding* GPS server (the GPS server for short). Hereafter we use the term virtual time assuming we are referring to the GPS system virtual time. We say that a flow is backlogged/idle if it is backlogged/idle in the GPS server, independently of its state in the real system. We define as *total backlog* at time t the sum of the backlogs of all the flows in the GPS server at time t, and we call *expected clearing time* at time t the time instant $t_C \geq t$ in which the total backlog will be cleared if no packet arrives after time t.

In the rest of this section, we always refer to the problem of computing $V(t_{new})$ at a generic time instant t_{new} belonging to a busy period for the GPS server (see the note at the beginning of Subsec. III-A), provided that the total amount of service delivered by the system is known upon each packet arrival and upon any time instant at which the value of the virtual time is to be computed. We show that L-GPS solves this problem at $O(\log N)$ cost, by using an *ad hoc* augmented balanced binary tree, called U_{tree}, that must be updated at $O(\log N)$ cost upon each packet arrival.

The approach used to compute $V(t_{new})$ is similar to the one used in the sequential algorithm [15] (Subsec. III-A). As in the sequential algorithm, when the computation of $V(t_{new})$ begins, the state of the GPS server corresponding to a time instant $t_{old} < t_{new}$ is available (as shown in Subsec. IV-C, there can be up to $O(N)$ break instants between t_{old} and t_{new}). Then L-GPS uses the information stored in the U_{tree} to reconstruct the evolution of the virtual time during $(t_{old}, t_{new}]$.

Each node of the U_{tree} contains aggregated information on a time interval ranging between two break instants. In general the extremes of the time interval are not consecutive break instants; on the contrary, up to $O(N)$ break instants can be included in it. The information stored in the nodes is organized in a hierarchical fashion: two sibling nodes contain information on two adjacent intervals, and their parent node contains aggregated information on the union of the two intervals. Whereas in the sequential algorithm the break instants included in $(t_{old}, t_{new}]$ must *all* be sequentially processed, the aggregated information stored in the U_{tree} allows L-GPS to process events *in groups* during a special visit from the root to a leaf of the U_{tree}. Up to $O(N)$ events are processed at $O(1)$ cost each time a level of the U_{tree} is descended. In the end, the maximum number of steps performed is in the order of the depth of the U_{tree}.

Finally, the main idea behind the construction of the U_{tree} is *pre-computing* and storing information on the *expected* evolution of the virtual time. As shown in detail in the next subsection, the expected evolution of the virtual time changes upon each packet arrival. The information stored in the U_{tree} is coded in such a way that it can be updated at $O(\log N)$ cost after each packet arrival.

As previously said, the nodes of the U_{tree} contain information on time intervals whose extremes are break instants. We say that a point $(\bar{t}, V(\bar{t}))$, with $\bar{t} > t$, is an *expected* breakpoint at time t if it will constitute a breakpoint if no packet arrives after time t; furthermore, we say that $\bar{t}$ is an *expected* break instant at time t, and that $V(\bar{t})$ is an *expected* break value at time t.

Expected breakpoints are obviously due *only* to flows becoming idle. We define, for each flow i, the *flow virtual unbacking time* $U_i(t)$ as the virtual finish time of the last packet of the *i-th* flow arrived up to time t. The expected break values at time t correspond to the virtual unbacking times of the flows backlogged at time t. Through (3), the virtual unbacking time of each flow can be easily computed/updated upon the arrival of each of its packets. It is important to note that from the same formula it follows that the virtual unbacking time of a flow *does not change* in consequence of the arrival of packets belonging to other flows.

In the next subsection we show in detail the data structure used by L-GPS, whereas in the successive two subsections we show, respectively, how the virtual time is computed using this data structure and how the data structure itself is updated.

A. The shape data structure

L-GPS stores information on the expected evolution of the virtual time in the following data structure:

Definition 2: Shape data structure. Union of a *base tuple* containing, at any time instant t, the state of the GPS server corresponding to a time instant $t_{old} \le t$, and a balanced binary tree, called U_{tree} and containing one leaf for each (actual or expected) break instant included in $(t_{old}, t_C]$, where t_C is the expected clearing time at time t. Each node of the U_{tree} represents a time interval $[t_{min}, t_{max}]$, where t_{min} and t_{max} are, respectively, the smallest and the largest time instant represented in the subtree rooted at the node (leaves represent time intervals of length 0). Furthermore:

1) the time interval represented by the left child of a node precedes the time interval represented by the right child;

2) the information stored in each node – *all* evaluated assuming that no packet arrives after time t – are: the (actual or expected) break value $U_{max} = V(t_{max})$, the difference $\Delta\Phi = \Phi(t^+_{max}) - \Phi(t^-_{min})$, and a *correction factor* ΔW characterized by the following property: given any time instant $t_1 \le t_{min}$ such that there is no break instant in $[t_1, t_{min})$, we have

$$W(t_1, t_{max}) = \Phi(t_1^+) \cdot (U_{max} - V(t_1)) - \Delta W \qquad (5)$$

where $W(t_1, t_{max})$ is the expected amount of total service delivered by the system during $[t_1, t_{max}]$. For a leaf, $t_{min} = t_{max} = t_j$, $U_{max} = V(t_j)$, $\Delta\Phi = \Phi(t_j^+) - \Phi(t_j^-)$, and ΔW is obviously 0.

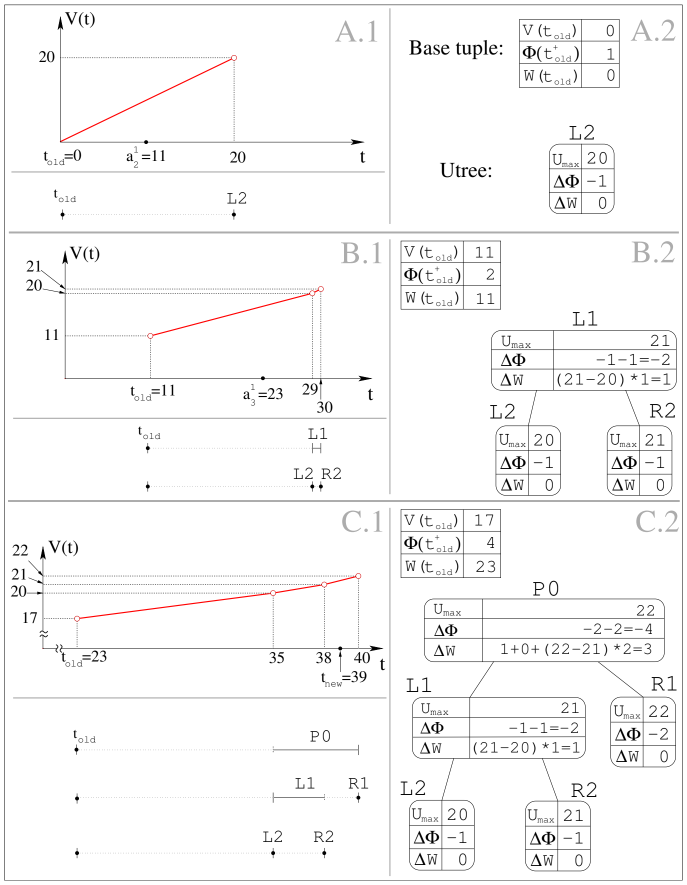

Fig. 2. Expected virtual evolution and shape data structure after the arrival of each of the first three packets in Example 1.

Neglect for a moment the correction factor ΔW. In what follows we denote as f_i^j the finish time of the packet p_i^j in the GPS server. Consider the upper part of Fig. 2.A.1: with reference to Example 1 (where $\forall t\ C(t) = C = 1$ byte/sec), it shows the *expected* evolution of $V(t)$ after the arrival of p_1^1 ($L_1^1 = 20$ bytes, $\phi_1 = 1$) at time 0, assuming that no further packet arrives. Fig. 2.A.2 shows the corresponding shape data structure. For the base tuple, we have that $\Phi(0^+) = \phi_1 = 1$, $W(0) = 0$, $V(0) = 0$. Flow 1 gets the entire capacity, and there is just one expected break instant, corresponding to the expected clearing time $f_1^1 = \frac{L_1^1}{C} = \frac{20}{1}$. From (3), the corresponding break value is $U_1(0^+) = F_1^1 = \frac{L_1^1}{\phi_1} = 20$. On this breakpoint, $\Phi(t)$ varies by a quantity $\Delta\Phi = -\phi_1 = -1$. Hence the U_{tree} consists of just the leaf $L2$, containing the above information (Fig. 2.A.2). The bottom part of Fig. 2.A.1 shows the break instant represented by $L2$.

Fig. 2.A.1 also shows the time instant 11 at which a new packet, p_2^1, arrives ($L_2^1 = 10$ bytes, $\phi_2 = 1$). The expected evolution of $V(t)$ after the arrival of p_2^1 is shown in the upper part of Fig. 2.B.1. We have that $\Phi(11^+) = \phi_1 + \phi_2 = 2$, hence the slope of $V(t)$ halves at time 11. Since both flows 1 and 2 have the same weight, they start to get half of the capacity each. 11 out of 20 bytes of p_1^1 have been already served at time 11, hence the expected break instant f_1^1 moves from time 20 to time $11 + \frac{20-11}{C/2} = 29$ (of course, the corresponding break value F_1^1 is unchanged). During $[11, 29]$, $(29-11) \cdot \frac{C}{2} = 9$ bytes of

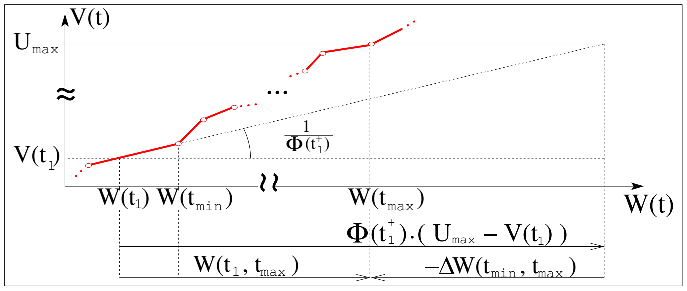

Fig. 3. The correction factor is equal to the difference between the value that $W(t_1, t_{max})$ would have if the slope of $V(t)$ was constant and equal to $\frac{1}{\Phi(t_1^+)}$ during $(t_1, t_{max}]$, and the actual/expected value of $W(t_1, t_{max})$ according to the actual/expected evolution of $V(t)$.

p_2^1 are served. Hence, after the completion of p_1^1, $10 - 9 = 1$ byte of p_2^1 is still to be served, and flow 2 starts getting all the capacity. As a consequence, there is one more expected break instant $f_2^1 = 29 + \frac{1}{C} = 30$, whose corresponding break value is equal to $U_2(11^+) = F_2^1 = V(11) + \frac{10}{1} = 21$ (see (3)). Finally, $\Phi(t)$ varies by a quantity $\Delta\Phi = -\phi_2 = -1$ on f_2^1.

Fig. 2.B.2 shows the corresponding shape data structure, assuming that the base tuple contains the state of the GPS server corresponding to time 11 (as shown in Subsec. IV-C, the base tuple may also happen to contain the state corresponding to a *lower* time instant than the current time, in consequence of the adopted *lazy* updating algorithm). The time intervals represented by the nodes of the U_{tree} (continuous lines or points) are shown in the bottom part of Fig. 2.B.1. The information on the new expected break instant $f_2^1 = 30$ is stored in the leaf $R2$. The leaf $L2$, representing the other break instant f_1^1, obviously contains *the same information* stored in the only node of the U_{tree} in Fig. 2.A.2. The root $L1$ of the U_{tree} contains aggregated information on the time interval ranging between the break instants represented by the two leaves $L2$ and $R2$; especially, it contains the break value $U_{max} = F_2^1 = 21$ and the cumulative variation $\Delta\Phi = -\phi_1 - \phi_2 = -2$ of the weight sum.

Finally, the upper part of Fig. 2.C.1 shows the expected evolution after the arrival of packet p_3^1 at time 23 ($L_3^1 = 10$ bytes, $\phi_3 = 2$). It is easy to show that the previous two expected break instants f_1^1 and f_2^1 move, respectively, from 29 to 35, and from 30 to 38, and that there is a new expected break instant $f_3^1 = 40$. The corresponding break value is $U_3(23^+) = F_3^1 = V(23) + \frac{L_3^1}{\phi_3} = 17 + 5 = 22$. Fig. 2.C.2 and the bottom part of Fig. 2.C.1 show the corresponding shape data structure and the represented time intervals, assuming that the base tuple contains the state of the GPS server corresponding to time 23.

Consider now the correction factor ΔW. Its purpose is allowing L-GPS to efficiently compute the value assumed by $W(t)$ upon break instants (as shown in the next subsection, this is crucial to reconstructing the evolution of the virtual time during $(t_{old}, t_{new}]$). A simple solution to immediately get these values while visiting the U_{tree} would have been explicitly storing the value $W(t_{max})$ in each node representing the time interval $[t_{min}, t_{max}]$ (recall that t_{max} is a break instant). In contrast, according to (5), ΔW needs to be combined with additional information to compute the amount of work done by the system during the time interval $[t_1, t_{max}]$, and then this value must be summed to $W(t_1)$ to get $W(t_{max})$. The reason for storing the correction factor ΔW instead of $W(t_{max})$ in each node is the following. Before time t_{max}, $W(t_{max})$ is actually an expected value: in general it changes after a packet arrival before t_{max} (recall the mandatory lazy evaluation problem). Updating the field $W(t_{max})$ in all the (involved) nodes can be easily proven to have $\Omega(N)$ cost. On the contrary, as shown below, the field ΔW can be updated at $O(\log N)$ cost. In the rest of this subsection we report just the properties of the correction factor, and in general of the shape data structure, whereas we show how these properties are exploited in the next two subsections.

A graphical representation of Eq. (5) is shown in Fig. 3. For each node, ΔW depends only on the information stored in the subtree rooted at the node, and it is *independent of* $\Phi(t_1^+)$, as stated by the following theorem.

Theorem 1: For any internal node P of a U_{tree}

$$\Delta W^P = \Delta W^L + \Delta W^R - \Delta\Phi^L \cdot (U_{max}^R - U_{max}^L) \qquad (6)$$

where L is the left child of node P, and R is the right one. The proof of the theorem can be found in the Appendix, whereas numerical examples are reported in Fig. 2.A.2, 2.B.2 and 2.C.2.

For ease of exposition, given any time interval $[t_{min}, t_{max}]$ represented by a node of the U_{tree}, we define as its *preceding gap* the maximal time interval $(\bar{t}, t_{min})$ containing no break instant and such that $\bar{t} \geq t_{old}$. Preceding gaps are depicted as dotted lines in the bottom parts of Fig. 2.A.1, 2.B.1 and 2.C.1. As highlighted by Fig. 2.C.1, there is a gap both between t_{old} and any of the leftmost time intervals represented by some node of the U_{tree}, and between every pair of time intervals represented by two sibling nodes.

Given the time interval represented by a generic node of the U_{tree}, Eq. (5) obviously holds for any time instant t_1 in its preceding gap. Suppose to know the state of the GPS server corresponding to a time instant t_1 in the preceding gap, and let U_{max}, $\Delta\Phi$ and ΔW be the values of the fields of the node. $W(t_1, t_{max})$ can be immediately computed through (5). Furthermore, $W(t_{max}) = W(t_1) + W(t_1, t_{max})$, $V(t_{max}) = U_{max}$ and, since $\Phi(t_{min}^-) = \Phi(t_1^+)$, $\Phi(t_{max}) = \Phi(t_1^+) + \Delta\Phi$. Hence, through the information stored in the node, the state of the GPS server corresponding to the time instant t_{max} can be computed at $O(1)$ cost, *independently* of the number of break instants in $(t_1, t_{max}]$.

It is worth noting that in case t_{max} is an expected break instant at time t_{new} ($t_{max} > t_{new}$), the above computed state is more precisely the *expected* state corresponding to the expected break instant t_{max}. As an example, consider Fig. 2.C.1 and 2.C.2: using the state stored in the base tuple $< V(t_{old}), \Phi(t_{old}^+), W(t_{old}) >$, corresponding to time $t_{old} = 23$, the values stored in the fields U_{max}^{L1}, ΔW^{L1} and $\Delta\Phi^{L1}$ of the node $L1$ allow the state corresponding to time $t_{max}^{L1} = 38$ to be computed at $O(1)$ cost.

We can now summarize the two *key features* that enable the U_{tree} to be updated and the virtual time to be computed at

```
1  // shape data structure:
2  V_old ;      // V(t_old)
3  W_old ;      // W(t_old)
4  Phi_old ;    // Phi(t_old +)
5  Utree ;      // Def. 2
6
7  function computeV( W_new )          // returns V(t_new)
8  {
9    // next three temp. variab. will store V(t_l), W(t_l),
10   // Phi(t_l +) at the end of the search (Eq. (4))
11   W_s = W_old ;
12   V_s = V_old ;
13   Phi_s = Phi_old ;
14   cur = Utree.root ;            // curr. search subtree
15
16   // at each search step we have:
17   // W_s [left gap] [left interval] W_L_max [right gap] [right interval]
18   while ( not is_leaf(cur) ) {        // search W(t_l)
19     W_L_Max = W_s + ( cur->left->Umax - V_s )*Phi_s -
20              cur->left->d_W;          // pivot: Eq. (5)
21
22     if ( W_new < W_L_Max )   // => W(t_l) < W_L_Max
23       cur = cur->left ;     //    cont. in left subtree
24     else {                  // => W(t_l) >= W_L_Max
25       // update variables to the begin. of next gap
26       V_s = cur->left->Umax ;
27       W_s = W_L_Max ;
28       Phi_s = Phi_s + cur->left->d_Phi ;
29       cur = cur->right ;   // cont. in right subtree
30     }                     // end of case W(t_l)>=W_L_Max
31   }                       // end of search loop
32
33   return V_s + ( W_new - W_s ) / Phi_s ;      // Eq. (4)
34 }
```

Fig. 4. Function `computeV`.

$O(\log N)$ cost: 1) thanks to (5), the information stored in a node representing a time interval $[t_{min}, t_{max}]$ allow the state of the GPS server corresponding to the time instant t_{max} to be computed at $O(1)$ cost, provided that the state of the GPS server corresponding to a time instant t_1 in the preceding gap is known; 2) according to Def. 2 and Th. 1, the information stored in each node depends only on its subtree.

A final remark is in order: the nodes of the U_{tree} do represent time instants/intervals, but they contain *no* information on the value of any time instant. Maintaining such information is in general a hard task in case of variable-rate links.

B. Computing the virtual time

In this subsection we show how L-GPS computes $V(t_{new})$ at $O(\log N)$ cost through the shape data structure, assuming that the U_{tree} has $O(\log N)$ depth. First we describe the algorithm, then we show an example of how it operates, finally we provide a synthetic proof of its correctness.

The algorithm is implemented by the function `computeV`, whose pseudocode is shown in Fig. 4. `computeV` takes as input $W(t_{new})$ and performs a binary search of the leaf representing the largest break instant $t_l \leq t_{new}$. During the search three temporary variables are used; they are updated in such a way that they will contain the tuple $< V(t_l),\, \Phi(t_l^+),\, W(t_l) >$ at the end of the search. Then they are used to compute $V(t_{new})$ through (4).

In more detail, the temporary variables are initialized to the tuple $< V(t_{old}),\, \Phi(t_{old}^+),\, W(t_{old}) >$ before beginning the binary search (lines 11-13). Then, upon each search step, the largest time instant t_{max}^L represented by the left child L of the node involved in the current search step is used as *pivot*. Unfortunately, as previously said, computing break instants in case of variable-rate links is a hard task. Hence t_{max}^L is *indirectly* compared against t_{new} by exploiting the following property: since the system is work-conserving, $W(t)$ is an increasing function of the time, hence the ordering between t_{new} and t_{max}^L is the same as between $W(t_{new})$ and $W(t_{max}^L)$. The last two values are the actually compared ones (line 22).

To compare it against $W(t_{new})$, $W(t_{max}^L)$ is computed by exploiting the first key feature of the U_{tree}. Upon the first iteration, L is the left child of the root node of the U_{tree}, hence there is no break instant between t_{old} and t_{min}^L (Def. 2). Therefore, through Eq. (5) the state stored in the base tuple is used to compute $W(t_{max}^L)$ at $O(1)$ cost (lines 19-20).

Consider now the right subtree of the U_{tree}, as e.g. the subtree rooted at $R1$ in Fig. 2.C.2. There is at least one break instant between t_{old} and the smallest time instant represented in this subtree. Hence, if the search continues in the right subtree upon the second iteration, the state stored in the temporary variables can no more be used to compute $W(t_{max}^L)$ at $O(1)$ cost through Eq. (5). On the contrary, as noted in the previous subsection, there is a gap (a time interval containing no break instant) between the largest time instant represented by a node and the smallest time instant represented by its right sibling. For this reason (and also to let them contain the state corresponding to time t_l at the end of the search), the state variables are updated to $< V(t_{max}^L),\, \Phi(t_{max}^{L\,+}),\, W(t_{max}^L) >$ (at $O(1)$ cost) on each search step that causes the search to continue into the right subtree (lines 26-28). Hence, they can be used to compute $W(t_{max}^L)$ at $O(1)$ cost at any iteration.

As an example, suppose to compute $V(t_{new} = a_2^2 = 39)$ (referring to Fig. 2.C.1 and 2.C.2). The temporary variables are first initialized to the base tuple, i.e. to the state corresponding to time 23. Upon the first iteration, $t_{max}^L = t_{max}^{L1} = 38$ (see the bottom part of Fig.. 2.C.1), and $W(38) = 23 + (21 - 17) * 4 - 1 = 38$ is computed at lines 19-20. Since $W(t_{new} = 39) > W(38)$, the temporary variables are updated to $< U_{max}^{L1} = F_2^1 = V(38) = 21$, $\Phi(23^+) + \Delta\Phi^{L1} = \Phi(38^+) = 2$, $W(38) = 38 >$ at lines 26-28. Then $R1$ is selected for the next search step. $R1$ is a leaf, hence the search loop ends, and, using the values stored in the temporary variables, $V(39)$ is computed as $V(38) + \frac{W(39)-W(38)}{2} = 21 + 0.5 = 21.5$.

Finally, to prove that the search ends up storing the tuple $< V(t_l),\, \Phi(t_l^+),\, W(t_l) >$ in the temporary variables, consider that: 1) $t_{new} \leq t_C$ and the U_{tree} is assumed to represent *all* the break instants included in $(t_{old}, t_C]$ (we show in the next subsection how this can be accomplished); 2) the system is *causal*, i.e. the evolution of the virtual time up to time t_{new} does not change in consequence of new packet arrivals after time t_{new}; hence all the states stored in the temporary variables during the search are *actual* states; 3) each time the search must continue in the right subtree, the temporary variables are updated to the state corresponding to the largest break instant represented by the left subtree.

Since a level of the U_{tree} is descended upon each iteration, the search terminates after a number of iterations no larger than the depth of the U_{tree}. Hence, since we assumed that the U_{tree} has $O(\log N)$ depth, the function `computeV` has $O(\log N)$ complexity.

```
1  bubble_up(P) {      // update aggr. info from node P
2    while ( is_not_null(P) ) {
3      P->Umax = P->right->Umax ;
4      P->d_Phi = P->left->d_Phi + P->right->d_Phi ;
5      P->d_W = P->left->d_W + P->right->d_W -
6       (P->right->Umax - P->left->Umax)*P->left->d_Phi;
7      P = P->father ;             // move up one level
8    }
9  }
10
11 // adds/updates a breakpoint; in: break value U, weight
12 // sum variation d_Phi, current virt. time curr_V
13 function add_break_point(U, d_Phi, curr_V) {
14   if (is_empty(Utree)) { // init base tuple
15     V_old = curr_V ;         // current value of V(t)
16     W_old = curr_W ;         // current value of W(t)
17     Phi_old = curr_Phi ;     // current value of Phi(t)
18   }
19   // next function returns the newly
20   // created or just updated leaf
21   leaf = bal_tree_insert(Utree, U, d_Phi) ;
22   bubble_up(leaf->father) ;      // update aggr. info
23   bal_tree_ins_fixup(leaf->father) ;    // rebal. tree
24
25   if (Utree.leftmost_leaf->U <= curr_V)    // stale brk
26     rem_break_point(Utree.leftmost_leaf,
27                     Utree.leftmost_leaf->d_Phi) ;
28   return leaf ;
29 }
30
31 // updates/removes a breakpoint
32 rem_break_point(leaf, d_Phi) {   // in: leaf to work on
33   if ( leaf == Utree.leftmost_leaf and
34        d_Phi == leaf->d_Phi) {
35     // Removing leftmost leaf, update base tuple:
36     W_old += Phi_old * (leaf->Umax - V_old)   // Eq. (5)
37     Phi_old = Phi_old + leaf->d_Phi ;
38     V_old = leaf->Umax ;
39   }
40   // next func. updates or removes the leaf and replaces
41   // leaf->father with the brother of the leaf
42   brother = bal_tree_remove(Utree, leaf, d_Phi) ;
43   bubble_up(brother->father) ;     // update aggr. info
44   bal_tree_rem_fixup(brother) ;    // re-balance tree
45 }
```

Fig. 5. Functions `add_break_point`, `rem_break_point` and `bubble_up`.

C. Updating the shape data structure

In this subsection we show how, by exploiting the second key feature of the U_{tree} and assuming the U_{tree} to be balanced, the shape data structure can be updated on each packet arrival at $O(\log N)$ cost. Especially, we show how nodes are automatically removed at $O(\log N)$ cost, and in such a way that the U_{tree} never contains more than N leaves. We show how balancing can be guaranteed by implementing the U_{tree} as an augmented balanced tree in the next section.

The shape data structure can be updated through two functions, `add_break_point` and `rem_break_point`, both shown in Fig. 5. `add_break_point` takes as input the break value *U* of the breakpoint to add, the variation *d_Phi* of $\Phi(t)$ on the breakpoint, and the current value of the virtual time.

When the arrival of a packet causes a flow to become backlogged at time t, `add_break_point` must be invoked twice, to add both the (actual) breakpoint corresponding to the flow becoming backlogged, and the expected breakpoint corresponding to the expected break instant at which the flow becomes idle if no packet arrives after time t. On the first invocation, the virtual start time of the just arrived packet and the weight of the flow must be assigned, respectively, to *U* and *d_Phi* ($\Phi(t)$ increases by the weight of the flow on the breakpoint); on the second invocation, the virtual unbacking time of the flow (equal to the virtual finish time of the packet) and the opposite of the weight of the flow must be assigned, respectively, to *U* and *d_Phi*.

On the contrary, if the packet causes the virtual unbacking time of an already backlogged flow to move forward, `rem_break_point` (described later) must be called to remove the old breakpoint, then `add_break_point` must be called to insert the new one.

Invoking `add_break_point` and `rem_break_point` as above shown guarantees the U_{tree} to represent, at any time instant t, *all* the (actual and expected) break instants larger than t_{old} and due to the packets arrived up to time t.

`add_break_point` calls the function `bal_tree_insert`, which descends the tree looking for a leaf containing the break value *U*. On success, `bal_tree_insert` adds *d_Phi* to the value stored in the field $\Delta\Phi$ of the leaf (a further flow becomes idle/backlogged upon the break instant represented by the leaf); otherwise it creates both a new leaf containing the tuple $< U, d\_Phi, 0 >$, and an internal node whose children are the newly created leaf and the last leaf visited during the search; hence it replaces the last leaf visited during the search with the newly created internal node.

It is worth noting that `bal_tree_insert` guarantees that each internal node of the U_{tree} has exactly two children (an internal node with just one child would represent the same time interval represented by its child).

`bal_tree_insert` does not deal with the aggregate information stored in the nodes, which are instead updated by the function `bubble_up` (lines 1-9). All the information stored in an internal node of the U_{tree} depend only on the information stored in the subtree rooted at that node (second key feature of the U_{tree}). Hence, if the information stored in a node changes, *only its ancestors* must be updated. Therefore, `bubble_up` updates only the nodes along the path from the input node to the root of the U_{tree}. The expressions used to update U_{max}, $\Delta\Phi$ and ΔW come from Def. 2 and Th. 1.

In order to preserve balancing, some types of balanced trees need a *fix up* after the insertion (removal) of a node. This is accomplished by the function `bal_tree_ins_fixup` (`bal_tree_rem_fixup`), whose code – as the one of `bal_tree_insert` – depends on the specific underlying balanced tree and is described in the next section.

It is easy to understand that the computational complexity of the functions `bal_tree_insert` and `bubble_up` is $O(d)$, where d is the depth of the U_{tree}. The complexity of the fix up functions shown in the next section is $O(d)$ as well.

After inserting a new leaf and updating the aggregate information, `add_break_point` checks whether the leftmost leaf of the U_{tree} represents a *stale* breakpoint (i.e. a breakpoint whose corresponding break value is no greater than the current value of the virtual time). If this is the case, `add_break_point` invokes `rem_break_point` to remove the leaf and to consistently update the base tuple.

Hence, on the one hand `add_break_point` does not increase the depth of the U_{tree} in case the removal of a stale breakpoint can be performed. On the other hand, when such a removal can not be performed, there is actually no stale breakpoint in the U_{tree}. In this case, the U_{tree} contains only expected breakpoints, due to flows becoming idle. But a flow whose state changes only once during a given time interval causes only one breakpoint during the time interval. Therefore, when the U_{tree} does not contain any stale breakpoint, it is representing a time interval that contains at most one break instant per flow.

As a conclusion, since there are N flows in the system and the U_{tree} is balanced, it is easy to prove that the depth of the U_{tree} never exceeds $O(\log N)$, and `add_break_point` has $O(\log N)$ complexity.

The same considerations about balancing issues made for `add_break_point`, apply also to `rem_breakpoint` (which is briefly commented in Fig. 5 too). In particular, `rem_break_point` invokes the function `bal_tree_remove`, which subtracts the value of the input argument *d_Phi* to the value stored in the field $\Delta\Phi$ of the leaf pointed by the input argument *leaf*. If this value becomes equal to zero, `bal_tree_remove` does remove the leaf, and replaces the father node of the just removed leaf with the other child (recall that `bal_tree_insert` guarantees each internal node to always have two children).

V. Balanced trees

The actual computational cost of L-GPS depends on the depth of the augmented balanced tree used to implement the U_{tree}. In the following two subsections we show two classes of balanced trees suitable for implementing the U_{tree}: Patricia Trees [16], that guarantee balancing from a statistical point of view, and Red-black Trees [17], that guarantee deterministic balancing. We also show that Patricia Trees do not need any re-balancing after insertions/extractions, and that they allow entire subtrees to be removed in $O(1)$ steps, which further improves the performance of L-GPS.

The reader interested into numerical issues (as e.g. timestamp wraparound) is referred to Subsec. 5.3 in [22].

A. Statistical balancing: Patricia Trees

Instead of the ordering between labels, a search tree can be organized as a function of the label representations as a sequence of digits. This is the main idea behind *tries* [16], a well known (and very studied) technique for storing and retrieving data. A common method to decrease the number of nodes in a trie is using a *path compression* method, known as Patricia compression [16]. A binary Digital Patricia Tree – hereafter called *DTree* for short – containing N values is a binary tree in which each leaf is labeled with the binary representation of each value (there is one leaf per value), whereas each internal node is labeled with the common *prefix* of the labels of all the leaves stored in the subtree rooted at the node.

The U_{tree} can be implemented as an augmented DTree in which each leaf is labeled with the binary representation of the break value it contains, and each internal node is labeled with the common prefix of all the break values stored in its subtree. If we imagine to add such a prefix to each internal node, then Fig. 2.A.2, 2.B.2 and 2.C.2 turn out to show three U_{tree} implemented as augmented DTrees.

The form of a DTree depends only on the values it contains, and it is independent of the order in which values are inserted. If M is the number of binary digits used to represent the values stored in a DTree, the maximum depth of the DTree is equal to M. However, the average depth of a DTree has the following interesting property. Consider a DTree containing N independent random values from a distribution with any density function $f(x)$ such that $\int f^2(x)dx < \infty$: the expected *average* depth of such a DTree is $O(\log N)$ [16], [18].

Finally, in our simulations (Sec. VII), even the *measured maximum* depth of a DTree-based U_{tree} resulted to be $O(\log N)$ with small constants (within a factor 2 with respect to the maximum depth of a perfectly balanced tree).

Thus, `bal_tree_insert` and `bal_tree_remove` have $O(\log N)$ complexity in practical cases, and they are quite efficient, because each elementary step is based on simple bit-comparisons. Finally, `bal_tree_ins_fixup` and `bal_tree_rem_fixup` are obviously empty functions.

DTrees allow a further optimization. Let node L be the root of a subtree to remove, node P be the father of node L, and node R be the other child of node P. If node R was the only child of node P, the labels and the aggregate information stored in both nodes would coincide. Hence, the removal of the subtree rooted at node L can be achieved by simply substituting node R in place of node P (suppose e.g. to remove the subtree rooted at $L1$ in Fig. 2.C.2: the content of node $P0$ will just coincide with the one of its right child $R1$). Each node of the subtree rooted at node L can be easily recycled by inserting node L in a list of *free trees*, i.e. a list whose elements are root nodes of trees removed from the U_{tree}. Whenever a new node must be added to the U_{tree} and the list is not empty, the node can be recycled from the head Z of the list. If node Z has children, they are inserted as the first and the second element of the list. Hence insertions into and extractions from the list have $O(1)$ cost.

Consider the function `computeV`: if the left subtree of the node involved in the current search step is removed from the U_{tree} each time the binary search continues in the right subtree, then *all* the stale breakpoints are pruned from the U_{tree} each time the new value of the virtual time is computed (aggregate information can be easily updated at the end of the search by invoking `bubble_up` and passing to it the last node visited).

B. Deterministic balancing: Red-black Trees

Red-black Trees [17] are balanced search trees based on comparisons between keys. Each node is labeled with one of the K values contained in the tree; furthermore, all the labels in the subtree rooted at the left/right child of a node are smaller/larger than the label of the node. Two special fix up (re-balancing) functions, invoked, respectively, after each insertion and extraction, guarantee the maximum depth of a Red-black Tree containing K nodes to be equal to $\lceil 2 \cdot \log_2(K+1) \rceil$ [17]. Furthermore, fix up operations have logarithmic complexity with small constants [17].

The U_{tree} can be implemented as an augmented Red-black Tree in which each leaf is labeled with the break value it contains, and each internal node is labeled with the maximum break value stored in the leaves of its left subtree. Since a binary

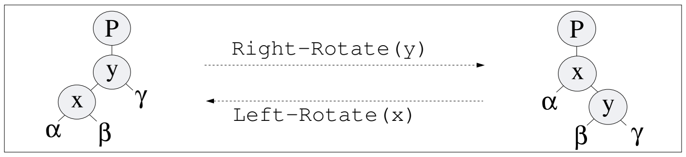

Fig. 6. The rotation operations performed by the fix up functions in a Red-black Tree. The letters α, β and γ represent arbitrary subtrees.

tree with N leaves has $2 \cdot N - 1$ nodes, the worst-case depth guaranteed by the underlying Red-black Tree for the U_{tree} is equal to $\lceil 2 \cdot (1 + \log_2 N) \rceil$.

`bal_tree_ins_fixup` and `bal_tree_rem_fixup` can be obtained with minor modifications from the fix up functions shown at pages 268 and 274 of [17]. The only critical operations performed by these functions are the two *rotations* shown in Fig. 6: each rotation does not affect the aggregate information stored in the parent node P and in the root nodes of the subtrees α, β and γ. Hence the original functions need to be modified so as to apply the inner part of the `while` loop in `bubble_up` (Fig. 5, lines 3-6) only to the nodes x and y after each rotation.

It is worth noting that both nodes x and y are assumed to be internal nodes in a rotation [17], hence the leaves of the U_{tree} can never (erroneously) become internal nodes.

VI. L-WF^{2}Q

In this section we describe L-WF^{2}Q, an implementation of WF^{2}Q with $O(\log N)$ complexity and small constants.

In addition to using L-GPS to compute the virtual time, L-WF^{2}Q exploits the following property to further reduce the computational cost. Assume that flow timestamps are immediately updated each time a new packet is enqueued or dequeued (i.e. as it starts to be served). It follows that the quantity $|V(t) - S_i(t)|$ is upper bounded by the maximum difference between the *normalized* amount of service delivered to the *i-th* flow by, respectively, the GPS server and the real system. Furthermore, as shown in Sec. II, the maximum deviation of WF^{2}Q with respect to the GPS service is equal to L_{max}. Hence,

$$|V(t) - S_i(t)| \leq \frac{L_{max}}{\phi_i} \;\; \forall i, \forall t \tag{7}$$

known as the Globally Bounded Timestamp (GBT) property [19]. As a consequence,

$$U_i(t) - S_i(t) > \frac{L_{max}}{\phi_i} \;\Rightarrow\; U_i(t) > V(t) \;\; \forall i, \forall t$$

In the end, $U_i(t_{new})$ can constitute an actual break value at time t_{new} only if $U_i(t_{new}) - S_i(t_{new}) \leq \frac{L_{max}}{\phi_i}$. We define as *near* the virtual unbacking times that meet this condition. It is easy to understand that the virtual time can be computed considering only near virtual unbacking times. Therefore, the virtual unbacking times to insert into the U_{tree} can be properly filtered, thus reducing the depth of the U_{tree}. The effectiveness of this optimization during congestion periods is shown in the next section through simulations.

The pseudocode for L-WF^{2}Q is shown in Fig. 7. Both the functions `enqueue` and `dequeue` can be divided into two parts: the first part (`enqueue` lines 3-12, `dequeue` lines 32-37) is a *vanilla* implementation of the packet timestamping and selection policy of WF^{2}Q [3] (Eq. 3 and Def. 1), whereas the second part (`enqueue` lines 13-27, `dequeue` lines 38-39) deals with the shape data structure and implements the previously defined filtering of the unbacking virtual times (`enqueue` line 14, `dequeue` line 38).

```
1  enqueue(pkt)        // invoked when a new pkt arrives
2  {
3    V = computeV(curr_W) ;
4    f = find_flow(pkt) ;      // find the flow owning pkt
5    pkt.S = max(V, f.U) ;                          // Eq. 3
6    pkt.F = pkt.S + pkt.L/f.phi ;                  // Eq. 3
7    tail_insert(f, pkt) ;        // ins. pkt into f queue
8    if (queue_head(f) == pkt) {        // flow f was idle
9      // update flow timestamps
10     f.S = pkt.S ;
11     f.F = pkt.F ;
12   }
13   f.U = pkt.F ;        // update flow unback. virt. time
14   if (f.U <= f.S + Lmax/f.phi) {  // f.U is (still) near
15     if (not_in_Utree(f.Uleaf) or f.Uleaf->Umax < V) {
16        // flow f is not present in the Utree, or its
17        // previous unbacking vtime was overcome by V
18       add_break_point(f.S, f.phi, V) ;
19       f.Uleaf = add_break_point(f.U, -f.phi, V) ;
20     }
21     else {             // move forward f.U
22       rem_break_point(f.Uleaf, f.phi) ;
23       f.Uleaf = add_break_point(f.U, -f.phi, V) ;
24     }
25   } // end of branch for near f.U
26   else if (in_Utree(f.Uleaf))     // f.U is no more near,
27     rem_break_point(f.Uleaf, f.phi) ; // rem. from Utree
28 }
29
30 packet dequeue() // invoked when the link is available
31 {
32   pkt = schedule_next() ;                      // Def. 1
33   f = find_flow(pkt) ;      // find the flow owning pkt
34   head_remove(f) ;  // rem. pkt at the head of f queue
35   if (not is_empty(f)) {      // update flow timestamps
36     f.S = head(f).S ; // may cause f.U could become near
37     f.F = head(f).F ;
38     if (not_in_Utree(f.Uleaf) and f.U <= f.S + Lmax/f.phi)
39       f.Uleaf = add_break_point(f.U, -f.phi, V) ;
40   }
41   return pkt ;
42 }
```

Fig. 7. L-WF^{2}Q.

The function `schedule_next` (line 32) returns the next packet to transmit among the eligible ones. In [4] it is shown how to perform this operation with $O(\log N)$ complexity, using a special augmented balanced binary search tree. In such a tree each node represents a backlogged flow, and contains the flow virtual start and finish times. In addition, each node contains as aggregated information the minimum virtual finish time among the ones stored in the nodes of its subtree. As suggested in [4], balancing can be guaranteed by using e.g. a Red-black Tree as underlying balanced tree. In the end, the resulting tree is similar to the U_{tree}, and keeping it up to date involves similar operations.

VII. Simulation results

As shown in Subsec. V-A, DTrees are very simple to handle, and allow *all* the stale breakpoints to be efficiently removed from the U_{tree} upon each update of the system virtual time. But, although the expected average depth of a DTree is $O(\log N)$, its worst-case depth is $O(M)$, where M is the number of bits in the labels of the nodes. Hence, to assess the actual performance of a DTree in practical cases, we simulate the operation of L-

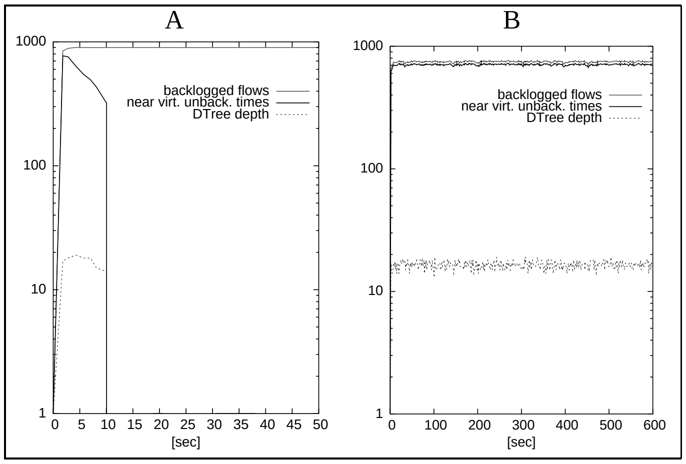

Fig. 8. System evolution in case of: A) generic scenario with offered load greater than the link capacity, B) Scenario 4.

WF^{2}Q when the U_{tree} is implemented with a DTree and only near virtual unbacking times are inserted in the U_{tree}. Finally, we compare the achieved performance with the one that would have been guaranteed by an ideal perfectly balanced tree and by a Red-Black Tree.

We use the *ns-2* network simulator [23]. The environment consists of a node with a 10 Mbps output link. We simulate the following 4 scenarios for 10 minutes each:

1) 1000 simultaneous FTP transfers.

2) 755 (asynchronous) Constant Bit Rate (CBR) traffic sources with packet length distribution equal to the one experienced in an Internet router according to [24]. Sources were divided into five (rate, weight) groups, ranging from (10 Kbps, 1) to (50 Kbps, 5), increasing in steps of (10 Kbps, 1).

3) 820 VoIP traffic sources, using CISCO [25] codec G.723 (30 bytes payload, 22 packets per sec, 40 bytes IP/UDP/RTP header).

4) A mix of the previous traffic sources: 20 FTP sources, 400 asynchronous 10Kbps CBR sources, 350 VoIP sources, plus 20 Video sources (MPEG-4 coding) transmitting real video traffic traces taken from [26].

During each simulation we take *snapshots* of the state of the system – number of backlogged flows, number of *near* virtual unbacking times and depth of the U_{tree} – at time intervals with length uniformly distributed between 1 and 2 seconds.

In a preliminary simulation run we found that the number of backlogged flows and, hence, the frequency of breakpoints is very low if the offered load is 'too' lower than the link capacity (a 5% lower offered load is enough to get a very low frequency of breakpoints). On the contrary, if the offered load is equal to or higher than the link capacity, the number of backlogged flows is high, but the number of breakpoints stored in the U_{tree} is limited by the filtering of the *near* virtual unbacking time. Especially, the more the backlog increases, the more the filtering becomes effective: Fig. 8.A shows this phenomenon in case the offered load is 20% larger than the link capacity.

As a consequence, for each of the above scenarios (except for scenario 1), the number of sources and the rate of each source come from a fine tuning aimed at achieving the maximum frequency of breakpoints. Fig. 8.B shows the evolution of the system in case of Scenario 4 (qualitatively similar to the ones for scenarios 2 and 3). Apart from a short initial transitory period, the number of near virtual unbacking times recorded in each snapshot is roughly equal to the total number of flows.

Scen.	Flows	Mean DTree	99% Conf.	Max DTree	Max Bal.	Ratio	Max RB
1	1000	0	0	0	0	-	0
2	755	14.62	0.02	17	11	1.55	21
3	820	14.84	0.10	17	11	1.55	22
4	790	16.47	0.06	20	11	1.82	22

TABLE II
STATISTICS COLLECTED FOR EACH SCENARIO

To compute statistics on the depth of the U_{tree}, each simulation is repeated 10 times and, for each simulation, only the steady time interval is considered (e.g. $[100, 600]$ in Fig. 8.B).

Table II summarizes our results: for each scenario, each column reports, respectively, the number of competing flows, the mean depth of the U_{tree}; the semi-width of the 99% confidence interval upon this value; the maximum depth of the U_{tree} (the maximum among the depths of the U_{tree} recorded in each snapshot), the depth of a perfectly balanced tree containing N_{max} leaves ($1 + \lceil \log_2 N_{max} \rceil$), where N_{max} is the maximum among the number of near virtual unbacking times recorded in each snapshot; the ratio between the maximum depth of the U_{tree} (column 5) and the maximum depth of the perfectly balanced tree (previous column); the worst-case depth of a Red-black Tree with N_{max} leaves ($\lceil 2 \cdot (1+ \ \log_2 N_{max}) \rceil$, Subsec. V-B).

Whereas the null depth of any tree for scenario 1 is a consequence of the filtering of virtual unbacking times, in all the other cases the mean depth and the (sample) maximum depth of the U_{tree} is within a factor 2 with respect to the maximum depth of a perfectly balanced tree.

VIII. Conclusions

In this paper we have shown L-GPS, a new algorithm for performing exact GPS simulation, and L-WF^{2}Q, an efficient implementation of WF^{2}Q based on L-GPS. Both algorithms have $O(\log N)$ complexity per packet transmission, and comply with constant-rate as well as variable-rate links.

To the best of our knowledge, L-WF^{2}Q is the first scheduler achieving the *optimum* service (i.e. the *minimum* deviation with respect to the GPS service) at $O(\log N)$ cost. Furthermore, analytical results and simulations demonstrate that the computational complexity of both L-GPS and L-WF^{2}Q has small constants too.

L-GPS and L-WF^{2}Q reduce the *upper* bound complexity for simulating a GPS server and the *upper* bound complexity for providing the *optimum* service, both from $O(N)$ to $O(\log N)$. Moreover, since the complexity lower bound to guarantee the minimum deviation with respect to the GPS service is $\Omega(\log N)$ [13], L-WF^{2}Q achieves the *optimum* service with *optimum* complexity.

IX. Acknowledgments

I wish to thank Ming-I Hsieh for his important fixes to the pseudocode of both L-GPS and L-WF^{2}Q. I would like to thank also Giovanni Stea and Martin Karsten for their helpful suggestions. Finally, I whish to thank the anonymous referees, whose valuable comments helped to improve the quality of this paper.

References

[1] A. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control - the single node case", in *Proceedings of INFOCOM '92*, 1992.

[2] D. Stiliadis and A.Varma, "Rate-proportional servers: A general methodology for fair queueing algorithms", *IEEE/ACM Transactions on networking*, 1996.

[3] References J. C. R. Bennett e H.Zhang, "WF^{2}Q: Worst-case fair weighted fair queueing", in *Proceedings of IEEE INFOCOM '96*, 1996.

[4] I. Stoica, H. Abdel-Wahab. "Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation", in *Technical Report 95-22*, Department of Computer Science, Old Dominion University, November 1995.

[5] J. C. R. Bennett e H.Zhang, "Hierarchical packet fair queueing algorithms", in *Proceedings of ACM SIGMETRICS '96*, 1996.

[6] D. Stiliadis and A. Varma, "Efficient Fair Queueing Algorithms for Packet Switched Networks," in *IEEE/ACM Transactions on Networking*, 1998.

[7] D. Stiliadis and A. Varma, "A general methodology for designing efficient traffic scheduling and shaping algorithms", in *IEEE INFOCOM'97*, 1997.

[8] S. Suri, G. Varghese and G. Chandramenon, "Leap Forward Virtual Clock: A New Fair Queuing Scheme with Guaranteed Delays and Throughput Fairness", in *Proceedings of IEEE INFOCOM'97*, 1997.

[9] S. Golestani. "A self-clocked fair queueing scheme for broadband applications", in *Proceedings of IEEE INFOCOM'94*, 1994.

[10] P. Goyal, H.M. Vin, and H. Chen. "Start-time Fair Queueing: A scheduling algorithm for integrated services." In *Proceedings of the SIGCOMM 96*, 1996.

[11] M. Shreedhar and G. Varghese. "Efficient fair queueing using deficit round robin", in *Proceedings of SIGCOMM'95*, 1995.

[12] C. Waldspurger. "Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management", PhD thesis, Massachusetts Inst. of Technology, 1995.

[13] J. Xu and R. J. Lipton. "On Fundamental Tradeoffs between Delay Bounds and Computational Complexity in Packet Scheduling Algorithms", in *Proceedings of ACM SIGCOMM '02*, 2002.

[14] A. G. Greenberg and N. Madras, "How Fair is Fair Queueing?", *Journal of the Association for Computing Machinery* 39, 1992.

[15] Qi Zhao, Jun Xu, "On the Computational Complexity of Maintaining GPS Clock in Packet Scheduling", in *Proceedings of IEEE INFOCOM'*04, 2004.

[16] D. E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, 1973.

[17] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. The MIT Press, 1991.

[18] L. Devroye. "A note on the average depth of tries", in *Computing*, 28:367-371, 1982.

[19] D.C. Stephens, J.C. Bennett and H. Zhang, "Implementing scheduling algorithms in high-speed networks" in *IEEE JSAC Special Issue on High Performance Switches/Routers*, 1999.

[20] V. Firoiu, J. Le Boudec, D. Towsley, Z. Zhang. "Advances in Internet Quality of Service", Technical report DSC200149, EPFL-DI-ICA, October 2001.

[21] A. Kortebi, L. Muscariello, S. Oueslati and J. Roberts, "Evaluating the Number of Active Flows in a Scheduler Realizing Fair Statistical Bandwidth Sharing", in *Proceedings of SIGMETRICS'05*, 2005.

[22] P. Valente, "Exact GPS simulation with logarithmic complexity, and its application to an optimally fair scheduler", in *Proceedings of SIGCOMM'04*, 2004.

[23] <www.isi.edu/nsnam/ns/>.

[24] <advanced.comms.agilent.com/insight/2001-08/Questions/traffic_gen.htm>.

[25] <www.cisco.com>.

[26] <www-tkn.ee.tu-berlin.de/research/trace/trace.html>.

Appendix

Proof of Theorem 1: We proceed by induction. Consider a node P of the U_{tree} and let t^L_{max} and t^R_{max} be the largest time instants represented, respectively, by the subtree rooted at the left child L, and by the subtree rooted at the right child R of the node P at time t_{new}. Possibly t^R_{max}, or both t^L_{max} and t^R_{max} are expected break instants at time t_{new}.

Let t_1 be a time instant such that there is no break instant between t_1 and the smallest break instant $t^P_{min} = t^L_{min}$ represented by the subtree rooted at P, and consider the total amount of service $W(t_1, t^P_{max})$ that the system is expected to deliver while the virtual time grows from $V(t_1)$ to $V(t^P_{max}) = U^P_{max} = U^R_{max}$ if no packet arrives after time t_{new} (Fig. 3). We can write:

$$W(t_1, t^P_{max}) \quad = \quad W(t_1, t^L_{max}) + W(t^L_{max}, t^R_{max}) \tag{8}$$

For the base case suppose that both nodes L and R are leaves: according to (2), (8) becomes

$$W(t_1, t^P_{max}) = \Phi(t_1^+) \cdot (U^L_{max} - V(t_1)) + \Phi(t^{L+}_{max}) \cdot (U^R_{max} - U^L_{max}) \tag{9}$$

According to Def. 2, we have

$$\Phi(t^{L+}_{max}) = \Phi(t^{L-}_{max}) + \Delta\Phi^L = \Phi(t_1^+) + \Delta\Phi^L \tag{10}$$

which, substituted in (9), gives

$$\begin{aligned} W(t_1, t^P_{max}) \quad &= \quad \Phi(t_1^+) \cdot (U^R_{max} - V(t_1)) + \\ &\quad -[-\Delta\Phi^L \cdot (U^R_{max} - U^L_{max})] \end{aligned}$$

For the inductive step, suppose that P is a generic internal node, and that Eq. (6) holds for both its children. Since there is no break instant between t^L_{max} and t^R_{min}, and considering (5) and (10)

$$\begin{cases} W(t_1, t^L_{max}) = \Phi(t_1^+) \cdot (U^L_{max} - V(t_1)) - \Delta W^L \\ W(t^L_{max}, t^R_{max}) = (\Phi(t_1^+) + \Delta\Phi^L) \cdot (U^R_{max} - U^L_{max}) - \Delta W^R \end{cases}$$

Substituting the above expressions in (8), we get

$$\begin{aligned} W(t_1, t^P_{max}) \quad &= \quad \Phi(t_1^+) \cdot (U^R_{max} - V(t_1)) + \\ &\quad -[\Delta W^L + \Delta W^R - \Delta\Phi^L \cdot (U^R_{max} - U^L_{max})] \end{aligned}$$

Paolo Valente Paolo Valente received the Laurea and the PhD degree in Computer Systems Engineering from the University of Pisa, Italy, in 2000 and 2004 respectively.

Starting from 2006, he is an Assistant professor (Ricercatore) at the Department of Computer Science of the University of Modena, Italy. His current research focus is the design and analysis of resource scheduling algorithms.

He was and is involved in national and European research projects.