Exact GPS Simulation with Logarithmic Complexity, and its Application to an Optimally Fair Scheduler

Paolo Valente
Dipartimento di Ingegneria dell'Informazione
Via Diotisalvi, 2
Università di Pisa, Italy
paolo.valente@iet.unipi.it

ABSTRACT

Generalized Processor Sharing (GPS) is a fluid scheduling policy providing perfect fairness. The minimum deviation (lead/lag) with respect to the GPS service achievable by a packet scheduler is one packet size. To the best of our knowledge, the only packet scheduler guaranteeing such minimum deviation is Worst-case Fair Weighted Fair Queueing (WF^2Q), that requires on-line GPS simulation. Existing algorithms to perform GPS simulation have $O(N)$ complexity per packet transmission (N being the number of competing flows). Hence WF^2Q has been charged for $O(N)$ complexity too. Schedulers with lower complexity have been devised, but at the price of at least $O(N)$ deviation from the GPS service, which has been shown to be detrimental for real-time adaptive applications and feedback based applications. Furthermore, it has been proven that the lower bound complexity to guarantee $O(1)$ deviation is $\Omega(\log N)$, yet a scheduler achieving such result has remained elusive so far.

In this paper we present 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. In particular, using our algorithm within WF^2Q, we achieve the minimum deviation from the GPS service with $O(\log N)$ complexity, thus matching the aforementioned lower bound. Furthermore, we assess the effectiveness of the proposed solution by simulating real-world scenarios.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and problem Complexity**]: Non-numerical Algorithms and Problems; C.2.6 [**Computer Communication Networks**]: Internetworking

General Terms

Algorithms, Performance

Keywords

Computational Complexity, Packet Scheduling, Quality of Service, Data Structures

1. INTRODUCTION

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

An important reference system in packet scheduling is the GPS (Generalized Processor Sharing) server [1]. Provided that each flow has a weight assigned to it, such 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 only 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, because of 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 deviation (lead/lag) with respect to the 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 (recall that, in practical networks, the number of flows can be in the tens of thousands, or more).

Furthermore, as shown in [3] and [5], a scheduler with $O(N)$ deviation with respect to the GPS service may provide a *bursty* service: 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. It has been shown that this oscillation of service rates adversely affects adaptive real time applications [20] (such as video streaming), and results in instability in case of feedback based applications [3, 5] (such as congestion control).

Hence, in the rest of this paper, we will use the maximum deviation with respect to the GPS service as a measure of the fairness and the smoothness of the service provided by a scheduler.

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 scheduling policy used) is *optimum*, if the discrepancy with respect to the GPS service never exceeds the *minimum* deviation. Finally, 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 real-time simulation of a GPS server, does achieve the optimum service. Unfortunately, to date, the best implementations of the GPS simulation [1, 15] may require processing $O(N)$ events in a single packet transmission time [9]. As the number of flows can become very large, such complexity is widely considered a significant barrier to on-line scheduling in high speed applications [20, 9, 6, 5, 10, 11].

Many scheduling algorithms with $O(\log N)$ complexity – such as Self Clocked Fair Queueing (SCFQ) [9], Frame Based Fair Queueing (FFQ) [6] and Start Time Fair Queueing [10] – perform an *approximated* simulation of a 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 was proposed in [7]). WF^{2}Q+ has $O(1)$ deviation from the *minimum* amount of service guaranteed to each flow by a GPS server. However, WF^{2}Q+ provides no guarantee on the maximum deviation from the *actual* service delivered by the GPS server when some flows are idle. As we will show in Subsection 3.1, in such a case, also WF^{2}Q+ can exhibit $O(N)$ deviation and provide a bursty service.

Finally, several schedulers with very low complexity (ranging from $O(1)$ to $O(\log\log N)$) have been proposed to achieve high efficiency in high speed applications [11, 8, 12, 19], but all 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 was proven to be $\Omega(\log N)$ [13], the problem of providing $O(1)$ deviation from fair service with sub-linear complexity was still open.

Contributions of this paper

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

Actually, the complexity of the operations depends on the depth of the tree, that can in turn be implemented by augmenting two different types of balanced binary trees: Patricia Trees [16], that guarantee $O(\log N)$ average depth, or Red-black Trees [17], that guarantee $O(\log N)$ worst-case depth. Especially, even though Patricia Trees provide a weaker theoretical complexity bound, they have a much simpler structure and they allow L-GPS to be implemented in a more efficient way than Red-black Trees. Thus, as we will show through simulations, they achieve very good performance in practical cases. In the end, depending on the specific balanced tree used, L-GPS enables the GPS service to be simulated with $O(\log N)$ – statistical or deterministic – complexity with respect to any time interval, e.g. single packet transmission time.

Furthermore, we will show how to use L-GPS to implement WF^{2}Q with $O(\log N)$ complexity and small constants. Hereafter such implementation will be referred to as Logarithmic-WF^{2}Q (L-WF^{2}Q). At our best knowledge, L-WF^{2}Q is the first scheduler with $O(\log N)$ complexity achieving $O(1)$ deviation (actually, the minimum deviation) with respect to the GPS service.

From a theoretical point of view, the contribution of this paper is twofold. i) We reduce the *upper* bound complexity for simulating a GPS server with respect to the best result in the literature. ii) By implementing WF^{2}Q with $O(\log N)$ complexity, we reduce also the *upper* bound complexity to provide the optimum service. 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.

From a practical point of view, we reduce the complexity of a scheduler, WF^{2}Q, that provides a very smooth service, suitable for real time adaptive applications (such as video streaming) and feedback based applications (such as congestion control).

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_{min}, U_{max}	$U_{min} \equiv V(t_{min})$, $U_{max} \equiv V(t_{max})$
$\Delta\Phi$	$\Delta\Phi \equiv \Phi(t^+_{max}) - \Phi(t^-_{min})$
ΔW	Correction factor to use in (6) and computed as in (7)

Table 1: Notations used in this paper.

Organization of this paper

This paper is organized as follows. In Section 2 we provide an overview of GPS and WF^{2}Q. In Section 3 we make a brief survey of related work, focusing especially on WF^{2}Q+. In Section 4 we present our main result, the L-GPS algorithm, whereas in Section 5 we discuss how it can be implemented with two classes of balanced trees. In Section 6 we describe L-WF^{2}Q. In Section 7 we show through simulations how the actual complexity of L-GPS and L-WF^{2}Q compares to the worst-case bound.

2. GPS AND WF^{2}Q

Consider a system in which N flows (defined in whatever meaningful way) share a common link with time varying capacity $R(t)$. 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 own backlog.

We define *system busy period* a maximal interval of time during which the system is never idle. At the end of a system busy period we can safely reinitialize the state of the system. Therefore, each time a system busy period is considered, without loss of generality, we assume it begins at time 0. Finally, most of the notations used in this paper are summarized in Table 1.

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:

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

where $dW(t) = R(t)\cdot dt$ is the total amount of service provided by the system in $[t,\ t+dt]$, $dW_i(t)$ is the amount of service received by the $i-th$ flow in $[t,\ t+dt]$ ($W(0) = 0$ and $W_i(0) = 0\ \forall i$), $B(t)$ 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 real-time simulation of the *corresponding* GPS server, i.e. a GPS server with the same arrival pattern and the same capacity of the real system. Especially, 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 packets that have already started service in the corresponding GPS (*eligible* packets), the next one that finishes in the corresponding GPS server, if no packet arrives after time t.

A practical way for implementing such 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]:

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

From (1), we have that $dV(t) = \frac{dW_i(t)}{\phi_i}\ \forall i \in B(t)$, i.e. the system virtual time measures the *normalized* amount of service received by each backlogged flow. 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, then its timestamps can be computed as follows [5]:

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

At every time, only the packets at the head of the queues of 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 couple of timestamps each flow i, called, respectively, flow i virtual start time $S_i(t)$ and flow i virtual finish time $F_i(t)$, and corresponding 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 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 $V(t)$, hence we say that a flow i is eligible at time t if and only if $S_i(t) \le V(t)$. 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 eligible flows sorted by virtual finish times, and computing the value of the system virtual time. As shown in [4], it is possible to maintain the eligible flows sorted by virtual finish times at $O(\log N)$ cost per packet arrival or departure. Computing the system virtual time is instead complicated by the fact that its slope can change $O(N)$ times during a single packet transmission, as shown in the following example.

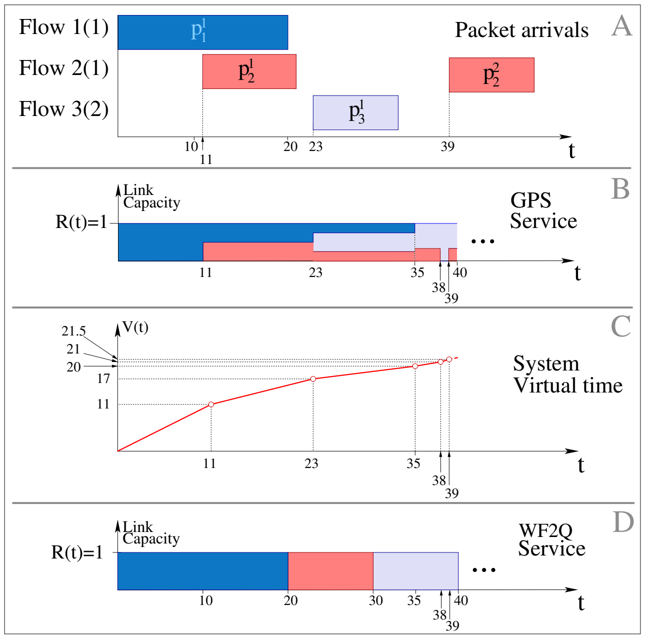

Figure 1: The evolution of the system virtual time.

EXAMPLE 1. *Consider a link with capacity of one byte per time unit. Fig. 1.A depicts a possible packet arrival pattern in such a system, 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. Flows 1 and 2 have weight 1, while flow 3 has weight 2. 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 speed.*

As shown in Fig. 1.C, system virtual time is a piecewise linear function of the total amount of service $W(t)$ delivered by the system. At each time t, the *slope* of $V(t)$ against $W(t)$ is inversely proportional to the sum of the weights of the backlogged flows (see (2)). Hereafter we will use the term *slope*, as a short for the slope of $V(t)$ against $W(t)$. Whenever $B(t)$ changes, the slope of $V(t)$ changes, constituting a *breakpoint* in its piecewise linear form. We define *break instant* every time instant $\bar{t}$ in 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 $\bar{t}$.

Suppose to compute $V(t_{new})$ at a generic time instant t_{new}, and consider the largest break instant $t_l \le t_{new}$, and the value $\Phi(t_l^+)$ assumed by $\Phi(t)$ immediately after t_l: the slope is constant and equal to $\frac{1}{\Phi(t_l^+)}$ during $(t_l,\ t_{new}]$, hence, according to (2)

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

The *classical* algorithm [1] for simulating a GPS server was defined for a constant rate server. In case of variable rate servers it can be generalized as follows: by using (4), update three state variables at *each* break instant t_j, assigning them to the tuple $< V(t_j),\ \Phi(t_j^+),\ W(t_j) >$. As a consequence, at a generic time instant t_{new} in which a scheduler based on the GPS simulation may need to know $V(t_{new})$, the state variables contain the tuple $< V(t_l), \Phi(t_l^+), W(t_l) >$ corresponding to the largest break instant $t_l \le t_{new}$. Hence (4) can be immediately applied to compute $V(t_{new})$.

In [15] it is shown how an early algorithm proposed in [14] can be used to compute the virtual time without updating the state variables at each break instant. Suppose that, at the time instant t_{new} in which $V(t_{new})$ is to be computed, 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 all the packets arrived during (t_{old}, t_{new}) are stored both in a queue ordered by packet arrival times, and in a queue ordered by packet (real) finish times in the corresponding GPS server. Using such information, the algorithm proposed in [14] can reconstruct all the breakpoints of the virtual time function in (t_{old}, t_{new}), update the state variables to the tuple $< V(t_l), \Phi(t_l^+), W(t_l) >$ and, hence, compute $V(t_{new})$ using (4). The number of steps performed by such algorithm is equal to the number of break instants in (t_{old}, t_{new}).

Breakpoints frequency depends on the frequency of transition of flows in and out of the set $B(t)$. Since in a GPS server the flows are served simultaneously, packet finish times can be arbitrarily slightly skewed. In the worst case, $O(N)$ finish times could fall in an arbitrarily small time interval. This could 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). For this reason, the complexity of the existing algorithms for simulating a GPS server is $O(N)$ per packet transmission time.

3. RELATED WORK

The two main issues related to the GPS service discipline are how to accurately approximate it on a real system, and how to efficiently compute the system virtual time.

To the best of our knowledge, apart from the initial $O(N)$ implementation proposed in [1], the computation of the system virtual time has been investigated in just one work [15], where it has been proven that maintaining the GPS virtual time has $\Omega(\log N)$ *amortized* per-packet complexity and that there is an algorithm [14] in the literature that matches such lower bound. Unfortunately, as the same authors of [15] show, such algorithm has $O(N)$ worst-case complexity per packet transmission time.

With regard to the relation between the service provided by practical packet schedulers and the GPS service discipline, several low complexity scheduling algorithms [19, 11, 8, 9, 6, 10, 5] have been proposed to solve (efficiently) the fairness problem and, in general, to provide service guarantees. As shown in the introduction, among them, WF^{2}Q+ [5] is the only candidate for providing $O(1)$ deviation with respect to the GPS service. Unfortunately, we will show in the next subsection that also WF^{2}Q+ exhibits $O(N)$ deviation and provides a bursty service when not all the flows are backlogged.

As a conclusion, to date, all packet scheduling algorithms, except for WF^{2}Q, exhibit $O(N)$ deviation with respect to the GPS service, or, worse yet, do not guarantee any bound on such deviation.

3.1 WF^{2}Q+ unfairness and burstiness

WF^{2}Q+ implements the same packet timestamping (3) and packet selection policy (Def. 1) of WF^{2}Q, but it uses a different system

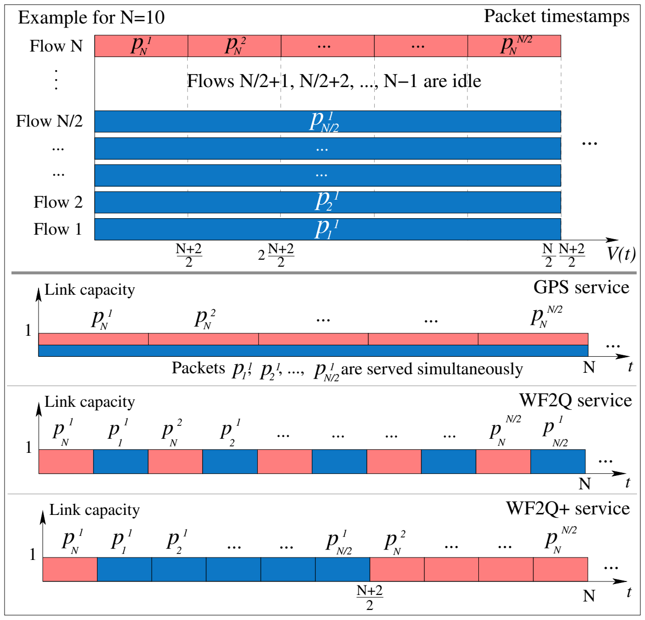

Figure 2: WF^{2}Q+ unfairness and bursty service.

virtual time function [5]:

$$V_{WF^2Q+}(t+\tau) = \max\{V_{WF^2Q+}(t)+W(t,t+\tau), \min_{i\in \hat{B}(t+\tau)}\{S_i(t+\tau)\}\} \quad (5)$$

where $W(t, t+\tau)$ is the total amount of service provided by the system during the period $[t, t+\tau]$, $\hat{B}(t+\tau)$ is the set of flows backlogged in the real system at time $t+\tau$, and, without losing generality, $\sum_{i=1}^N \phi_i = 1$ is assumed to hold. Thanks to this simpler function, WF^{2}Q+ has logarithmic complexity in the number of flows.

The *minimum slope* of $V_{WF^2Q+}(t)$ is equal to 1, i.e the slope of $V_{GPS}(t)$ when all the flows are backlogged (recall that $\sum_{i=1}^N \phi_i = 1$). It has been shown in [5] that, thanks to such property, WF^{2}Q+ achieves the minimum deviation with respect to the *minimum* service guaranteed by the GPS server to each flow. For this reasons, WF^{2}Q+ has the same performance of WF^{2}Q on several fairness indexes, such as the Worst-case Fair Index [3], that measures the minimum amount of service guaranteed by a scheduler to any flow over any time interval. But, what happens if not all the flows are continuously backlogged? Depending on the values of the weights of the flows, the slope of $V_{WF^2Q+}(t)$ can result to be arbitrarily smaller than the slope of $V_{GPS}(t)$.

Consider a system with N flows, the first $\frac{N}{2}$ with weight $\frac{2}{N} \cdot C$, the remaining $\frac{N}{2}$ with weight $1 \cdot C$, where the normalization factor $C = \frac{2}{N+2}$ guarantees that $\sum_{i=1}^N \phi_i = 1$ holds. Flows $1, 2, \ldots \frac{N}{2}$ and flow N are persistently backlogged, while the other flows are always idle. All packets have the same length and the outgoing link has capacity of 1 packet per time unit. The upper part of Fig. 2 shows the virtual start and finish timestamps of the packets of the backlogged flows, whereas the bottom one shows the service provided, respectively, by the GPS, WF^{2}Q and WF^{2}Q+ scheduling policies (the evolution of the system is periodic). We see that in the short term (the first $\frac{N}{2}+1$ packet transmissions), WF^{2}Q+ delivers $O(N)$ times less service to flow N than GPS or WF^{2}Q.

The ultimate reason for this is the following: since the virtual start time of the first packet of each of the first $\frac{N}{2}$ flows is 0, then, until all such packets have been transmitted, $\min_{i \in \widehat{B}(t)}\{S_i(t)\}$ is stuck at 0, and, hence, $V_{WF^2Q+}(t)$ grows with slope 1 (see (5)). Hence the second packet of flow N is not eligible before time $\frac{N+2}{2}$. On the contrary, after time $\frac{N+2}{2}$, a burst of $\frac{N}{2} - 1$ packets of flow N is transmitted.

4. L-GPS

In this section we will concentrate on the GPS simulation effort, and we will 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 *potential 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. We use the notation $f(t^-)$ and $f(t^+)$ to refer to the values assumed by a generic function f immediately before/after the time instant t. Finally, 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 such tuple.

In the rest of this section, we will always refer to the problem of computing $V(t_{new})$ at a given time instant t_{new}, provided that $W(t)$ is known at any time instant $t \leq t_{new}$. We will show that L-GPS solves this problem with $O(\log N)$ complexity, by using an *ad hoc* data structure that must be updated only at each packet arrival, and at $O(\log N)$ cost.

L-GPS stores the state of the GPS server in three variables we will refer to as the *base tuple*. As previously shown, the state of the GPS server can change $O(N)$ times in an arbitrarily short time interval, but certainly it is not used *so often* by a practical scheduler based on the simulation of a GPS server. For example, WF^{2}Q uses it only on each packet arrival (to timestamp the packet), and on each packet departure from the real system (to choose the next eligible packet to transmit). Basing upon this observation, L-GPS *does not* update the base tuple at each break instant, but only on packet arrivals. When $V(t_{new})$ is to be computed, L-GPS reconstructs the evolution of the virtual time during $(t_{old},\ t_{new}]$ with an approach similar to the one used in [15], and shown at the end of Section 2. But, whereas in [15] the information on the events (packet arrivals and departures) occurred during $(t_{old},\ t_{new})$ are somehow stored into two queues and they must *all* be processed sequentially, L-GPS stores them in a specially augmented binary tree, called U_{tree}, and process them in *groups* (possibly of $O(N)$ events) by navigating such tree.

Before providing the formal description of the data structure and the algorithm, we introduce them from an intuitive point of view. The main idea behind the construction of the U_{tree} is *pre-computing* the expected evolution of the virtual time.

We say that a point $(\bar{t},\ V(\bar{t}))$, with $\bar{t} > t$, is a *potential* breakpoint at time t if it will constitute a breakpoint if no packet arrives after time t; furthermore, we say that $\bar{t}$ is a *potential* break instant at time t, and that $V(\bar{t})$ is a *potential* break value at time t. Potential breakpoints are due only to flows becoming idle: if 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, then the potential break values at time t correspond to the virtual unbacking times of the flows backlogged at time t. Flows virtual unbacking times can be easily computed/updated at each packet arrival, and there is no need to update them on packet departures.

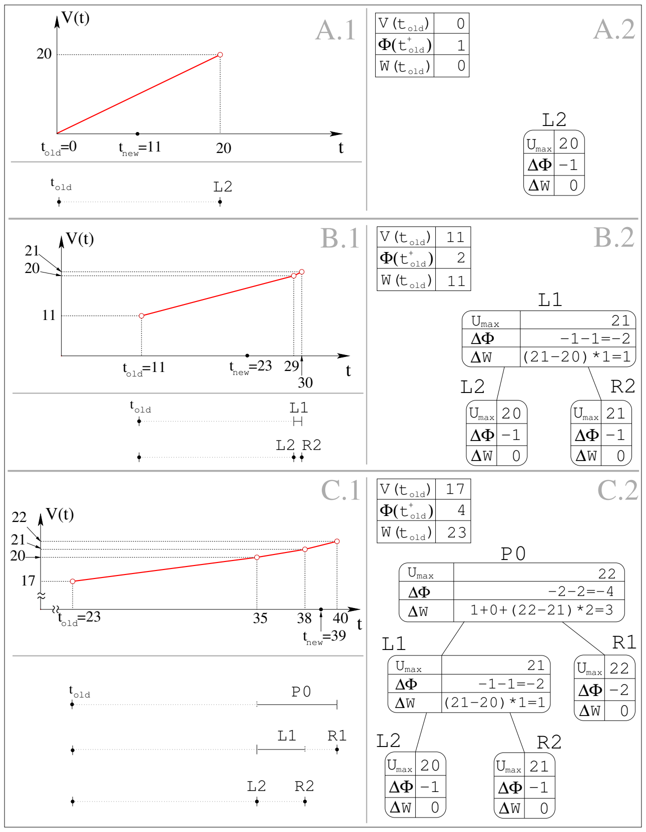

Figure 3: Expected virtual evolution and shape data structure after the arrival of each of the first three packets in Example 1.

To see how the expected evolution of the virtual time can be pre-computed, consider the upper part of Fig. 3.A.1. With reference to Example 1, it shows the *expected* evolution of the virtual time after the arrival of p_1^1 at time 0, assuming that no further packet arrives. Especially, $\Phi(0^+) = \phi_1$, whereas there is just one potential break-point – corresponding to the potential clearing time – whose break value is equal to $U_1(0^+) = F_1^1$. Furthermore, on such breakpoint, $\Phi(t)$ varies by a quantity $\Delta\Phi = -\phi_1$. Hence all the information on the expected evolution of the virtual time after p_1^1 arrival can be computed when such packet arrives, except for the potential clearing time, that is hard to compute in advance in case of variable rate links. On the contrary, the amount of service provided by the system on the clearing time does not depend on the capacity of the link, and it is equal to the length of p_1^1.

Actually, since the slope of $V(t)$ against $W(t)$ depends only on $\Phi(t)$ (see (2)), it is easy to understand that the expected evolution of the virtual time as a function of the service provided by the system is independent of the capacity of the link, and it changes *only* in consequence of packet arrivals. Fig. 3.A.1 shows also the time instant 11 in which a new packet, p_2^1, will arrive.

Since the system is *causal*, i.e. packets arriving after time t can not change the evolution of the system before t, the *actual* evolution of the virtual time during $[0,\ 11]$ coincides with the expected evolution computed at p_1^1 arrival. Hence, the information on the time interval $[0,\ 11]$ computed on p_1^1 arrival can be used to compute the state of the GPS server when p_2^1 arrives. Especially, the knowledge of the state of the GPS server at time 0 is enough to compute $V(11)$ by applying (4).

After the arrival of p_2^1, the expected evolution from time 11 changes in the following way (upper part of Fig. 3.B.1) with respect to the one computed on p_1^1 arrival: $\Phi(11^+) = \phi_1 + \phi_2$, and there is one more potential breakpoint, whose corresponding break value is equal to $U_2(11^+) = F_2^1$. Furthermore, $\Phi(t)$ varies by a quantity $\Delta\Phi = -\phi_2$ on the corresponding break instant. The amount of service provided by the system on each of the two potential break instants can be computed as a function of the service rate received by each flow, and of the fraction of p_1^1 not yet completed upon p_2^1 arrival.

When p_3^1 arrives, the information on the expected evolution computed on p_2^1 arrival can be used to compute the state of the GPS server, and to update the expected evolution of the virtual time as done on p_2^1 arrival. The new expected evolution is depicted in the upper part of Fig. 3.C.1.

In the end, if t_C is the potential clearing time at time t, then the expected evolution of the virtual time at time t, as a function of the service provided by the system during $[t_{old},\, t_C]$, is completely known if the state of the GPS server corresponding to $t_{old} \leq t$ is known, and, for every actual and potential break instant t_j included in $(t_{old},\, t_C]$, $V(t_j)$, the variation of the weight sum $\Delta\Phi = \Phi(t_j^+) - \Phi(t_j^-)$, and $W(t_j)$ are known. Obviously, the expected and the actual evolution during $(t_{old},\, t]$ coincide.

L-GPS stores the information on the expected evolution of the virtual time as a function of the service provided by the system in the base tuple and in the U_{tree}. Hence the U_{tree} must be updated *only* on each packet arrival. The U_{tree} *does not* contain any information on the values of the break instants, and L-GPS does not rely on their knowledge to compute $V(t_{new})$. However, in what follows we will mention break instants to simplify the description of the algorithm.

The U_{tree} contains one leaf for each of the (actual or potential) break instants included in $(t_{old},\, t_C]$, and each leaf stores the value of the virtual time, and the variation of the weight sum on the break instant it represents. If each leaf stored also the amount of service provided by the system on the break instant, $O(N)$ leaves may need to be updated on each packet arrival, because the arrival of a packet at time t causes the amount of service provided by the system on every potential break instant at time t to change. For this reason, the information on the service provided by the system on each break instant are not stored in the leaves, but in the internal nodes.

The way in which such information are coded is the basis of the two *key features* of the U_{tree}: i) the U_{tree} can be updated at $O(\log N)$ cost at each packet arrival (as we will show in the last subsection), ii) each node stores *aggregate* information on the time interval ranging from the smallest break instant t_{min} to the largest break instant t_{max} represented by the leaves of its subtree, and such information 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 $t_1 \leq t_{min}$ such that there is no break instant between t_1 and t_{min} is known.

As we will show in Subsection 4.2, L-GPS exploits such property to compute $V(t_{new})$: first, it computes the state of the GPS server corresponding to the largest break instant $t_l \leq t_{new}$ by performing a *binary search* [17] of the leaf representing it: before beginning the search, L-GPS initializes three temporary variables to the values contained in the base tuple; then, each time a new node is visited, it uses the aggregate information stored in such node to update the temporary variables to the state of the GPS server corresponding to the largest time instant represented in the subtree rooted at the node. When the target leaf is reached, the temporary variables contain the tuple $< V(t_l),\, \Phi(t_l^+),\, W(t_l) >$. Finally, L-GPS computes $V(t_{new})$ by applying (4). Hence the number of steps performed to compute $V(t_{new})$ is equal to the depth of the U_{tree}.

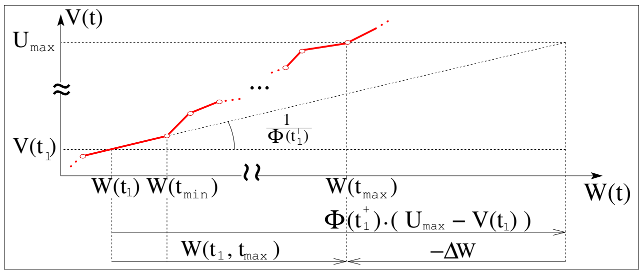

Figure 4: Graphical interpretation of ΔW.

Since the U_{tree} is balanced, and, as we will show in the last subsection, it is possible to bound the maximum number of leaves of the U_{tree} to N, L-GPS computes $V(t_{new})$ with $O(\log N)$ complexity. In the following two subsections we will describe, respectively, the data structure and the algorithm used by L-GPS to compute $V(t_{new})$, whereas, in the last subsection we will show how such data structure is updated upon each packet arrival.

4.1 The shape data structure

L-GPS computes $V(t_{new})$ through the following data structure:

DEFINITION 2. *We define* shape data structure *the union of a* base tuple *that stores, at any time instant* t*, the state of the GPS server corresponding to a time instant* $t_{old} \leq t$*, and a balanced binary tree, called* U_{tree} *and containing one leaf for each (actual or potential) break instant included in* $(t_{old},\, t_C]$*, where* t_C *is the potential 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 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) each node stores the following information – all *evaluated assuming that no packet arrives after time* t*: the break value* $U_{max} = V(t_{max})$*, the difference* $\Delta\Phi = \Phi(t_{max}^+) - \Phi(t_{min}^-)$*, and, finally, a* correction factor ΔW *such that, if there is no break instant between a time instant* $t_1 \leq t_{min}$ *and* t_{min}

$$W(t_1,\, t_{max}) = \Phi(t_1^+) \cdot (U_{max} - V(t_1)) - \Delta W \qquad (6)$$

where $W(t_1,\, t_{max})$ *is the expected amount of total service delivered by the system during the period* $[t_1,\, t_{max}]$*. 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*.*

The graphical interpretation of ΔW is shown in Fig. 4. 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}

$$\Delta W^P = \Delta W^L + \Delta W^R - \Delta\Phi^L \cdot (U_{max}^R - U_{max}^L) \qquad (7)$$

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. Figures 3.A.2, 3.B.2 and 3.C.2 show three instances of the shape data structure representing the expected evolutions of the virtual time depicted at their left side. The fields U_{max} and $\Delta\Phi$ in the leaves of the U_{tree} can be computed at each packet arrival in the way explained in the previous subsection, whereas, in case of an internal node, U_{max} is equal to the value of the corresponding field in its right child, $\Delta\Phi$ is equal to the sum of the corresponding fields in its children, and ΔW is computed according to Th. 1.

Eq. (6) has a simple graphical interpretation: the bottom parts of Figures 3.A.1, 3.B.1 and 3.C.1 show the time intervals (possibly of size 0) represented by each node of the U_{tree}; the subintervals – *gaps* – of $[t_{old},\, t_C]$ not covered by the nodes of the U_{tree} are represented by dotted lines. With reference to Fig. 3.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.

Consider then a gap and one of the consecutive time intervals $[t_{min},\, t_{max}]$ represented by the nodes of the U_{tree}, e.g. the gap $[23,\, 35]$ and the time interval represented by the node $L1$. If $V(t_1)$ and $\Phi(t_1^+)$ are known for a generic time instant t_1 belonging to the gap (extremes included), then (6) allows us to compute the amount of service $W(t_1,\, t_{max})$ delivered by the system during $[t_1,\, t_{max}]$ at $O(1)$ cost. Hence, if we know also $W(t_1)$, we can compute $W(t_{max})$ at $O(1)$ cost. Furthermore, $V(t_{max}) = U_{max}$ and, since $\Phi(t_{min}^-) = \Phi(t_1^+)$, $\Phi(t_{max}) = \Phi(t_1^+) + \Delta\Phi$. Hence, through the information stored in the node, we can compute the state of the GPS server corresponding to the time instant t_{max} at $O(1)$ cost.

For example, consider Fig. 3.C.2: since the base tuple $< V(t_{old}),\, \Phi(t_{old}^+),\, W(t_{old}) >$ is known, the fields U_{max}^{L1}, ΔW^{L1} and $\Delta\Phi^{L1}$ of the node $L1$ allows us to compute the tuple $< V(t_{max}^{L1}),\, \Phi(t_{max}^{L1\,+}),\, W(t_{max}^{L1}) >$ at $O(1)$ cost.

In case t_{max} is a potential break instant at time t_{new} ($t_{max} > t_{new}$), the state of the GPS server computed using U_{max}, $\Delta\Phi$ and ΔW is the *expected* state at time t_{max} if no packet arrives after time t_{new}.

We will describe in detail how the shape data structure is updated at each packet arrival in the last subsection, whereas we will show how the U_{tree} can be implemented by augmenting existing types of balanced trees in Section 5.

4.2 Computing Virtual Time

In this subsection we will show how L-GPS computes $V(t_{new})$ at $O(\log N)$ cost through the shape data structure.

The algorithm is implemented by the function computeV, whose pseudocode is shown in Fig. 5. 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, computeV updates three temporary variables, initially set to the tuple $< V(t_{old}),\, \Phi(t_{old}^+),\, W(t_{old}) >$. Upon each search step, it chooses as *pivot* the largest time instant t_{max}^L in the time interval represented by the left child L of the node involved in the current search step. In case the search continues on the right subtree, the temporary variables are set to $< V(t_{max}^L),\, \Phi(t_{max}^{L\,+}),\, W(t_{max}^L) >$.

The temporary variables have a noteworthy property: upon each search step they contain the tuple corresponding to the time instant at the beginning of the gap that precedes the time interval represented by the left child L of the node involved in the current search step, hence they allows the values $W(t_{max}^L)$ and $\Phi(t_{max}^L)$ to be computed at $O(1)$ cost (lines 19-20 and 29).

As previously said, computing break instants in case of variable rate links is a hard task. Hence the break instant used as pivot at each search step is compared against t_{new} *indirectly*, by exploiting the following property: since the system is work-conserving, $W(t)$ is an increasing function of time, hence the ordering between any break instant t_j represented by the U_{tree} and t_{new} is the same as between $W(t_j)$ and $W(t_{new})$.

Finally, to prove that the search ends up storing the tuple

```
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 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->d_W;                // pivot: see (6)
21
22     if ( W_new < W_L_Max ) // => W(t_l) < pivot
23       cur = cur->left ;    //   cont. in left subtree
24     else {                 // => W(t_l) >= pivot
25       cur = cur->right ;   //   cont. in right subtree
26       // update variables to the begin. of next gap
27       V_s = cur->left->Umax ;
28       W_s = W_L_Max ;
29       Phi_s = Phi_s + Cur.left.d_Phi ;
30     }                      // end of case t_l >= pivot
31   }                        // end of search loop
32
33   return V_s + ( W_new - W_s ) / Phi_s ;      // Eq. (4)
34 }
```

Figure 5: Function computeV.

$< V(t_l),\, \Phi(t_l^+),\, W(t_l) >$ in the temporary variables, consider that: 1) the U_{tree} is assumed to represent *all* the break instants included in $(t_{old},\, t_C]$ (we will show in the next subsection how this is accomplished); 2) since the system is causal, the information stored in the base tuple as well as the information stored in the nodes of the U_{tree} that represent time intervals that precede t_{new} do not change if new packets arrive after t_{new}; hence all the the values stored in the temporary variables during the search are *actual* and not *expected* values; 3) $t_{new} \leq t_C$.

Therefore, it is easy to understand that the search will stop on the leaf representing the smallest (potential) break instant no greater than t_{new}, and that, at such point, the temporary variables will contain the tuple $< V(t_l),\, \Phi(t_l^+),\, W(t_l) >$.

Since a level of the U_{tree} is descended upon each iteration, the search terminates after a number of iterations no greater than the depth of the U_{tree}. Since we assumed that the U_{tree} is balanced and, as we will show in the next subsection, it never contains more than N leaves, the function computeV has $O(\log N)$ complexity.

4.3 Handling the shape data structure

In this subsection we will show how the shape data structure is updated on each packet arrival at $O(\log N)$ cost. All the issues related to the *underlying* balanced tree used to implement the U_{tree} are the subject of the next section.

The *shape* data structure is handled by just two functions, add-break_point and rem_break_point (shown in Fig. 6). Apart from the current value of the virtual time (used for removing stale breakpoints, as we will show), add_break_point takes as arguments the break value *U* of the breakpoint to add, and the variation *d_phi* of $\Phi(t)$ on such breakpoint.

When the arrival of a packet causes a flow to become backlogged at time t, add_break_point must be invoked twice: to add the (actual) breakpoint corresponding to the flow becoming back-

```
bubble_up(P) {      // update aggr. info from node P
  while ( is_not_null(P) ) {
    P->Umax = P->right->Umax ;
    P->d_Phi = P->left->d_Phi + P->right->d_Phi ;
    P->d_W = P->left->d_W + P->right->d_W -
      (P->right->Umax - P->left->Umax)*P->left->d_Phi;
    P = P->father ;             // move up one level
  }
}

// add a brkpoint to the shape data str.; in: V(t),
// break value U, and weight sum variation d_Phi
function add_break_point(curr_V, U, d_phi) {
  // next function returns the newly
  // created or just updated leaf
  leaf = bal_tree_insert(Utree, U, d_phi) ;
  bubble_up(leaf->father) ;     // update aggr. info
  bal_tree_ins_fixup(leaf->father) ; // rebal. tree

  if (Utree.leftmost_leaf->U <= curr_V)  // stale brk
    rem_break_point(Utree.leftmost_leaf) ;
  return leaf ;
}

// removes a brkpoint from the shape data structure
rem_break_point(leaf) {       // in:  leaf to remove
  if ( leaf == Utree.leftmost_leaf ) {
    // Removing leftmost leaf, update base tuple:
    W_old += Phi_old * (leaf->Umin - V_old) -
             cur->d_W ;                 // Eq.  (6)
    Phi_old = Phi_old + leaf->d_Phi ;
    V_old = leaf->Umin ;
  }
  // next function removes the leaf and replaces
  // leaf->father with the brother of the leaf
  brother = bal_tree_remove(Utree, leaf) ;
  bubble_up(brother->father) ; // update aggr. info
  bal_tree_rem_fixup(leaf) ;      // rebalance tree
}
```

Figure 6: Functions `add_break_point`, `rem_break_point` and `bubble_up`.

logged, and the potential breakpoint corresponding to the potential break instant in which the flow becomes idle if no packet arrives after time t. On the first invocation, we assign the virtual start time of the just arrived packet to U and the weight of the flow to *d_phi* ($\Phi(t)$ increases by the weight of the flow); on the second invocation, we assign the virtual unbacking time of the flow (equal to the virtual finish time of the packet) to U and the opposite of the weight of the flow to *d_phi*.

On the contrary, if the packet causes the virtual unbacking time of an already backlogged flow to move forward, `rem_breakpoint` must be called to remove the old breakpoint, hence `add_break-point` must be called to insert the new one (passing to it the new value of the virtual unbacking time of the flow and the opposite of the weight of the flow).

By doing so, the U_{tree} represents, at any time instant t, *all* the (actual and potential) break instants larger than t_{old} and due to the packets arrived up to time t. Furthermore, since the system is causal, the actual break instants represented by the U_{tree} at time t coincide with the *only* actual break instants in $(t_{old}, t]$.

`add_break_point` calls the function `bal_tree_insert` that 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 at 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.

`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, that are instead updated by the function `bubble_up`. 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. Hence, if the information stored in a node change, 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 comes from Def. 2 and Th. 1.

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

It is easy to understand that the complexity of the functions `bal-tree_insert` and `bubble_up` is $O(d)$, where d is the depth of the U_{tree}. We will show that also the complexity of `bal_tree_ins-fixup` is $O(d)$ in the next section.

After having inserted a new leaf and having updated the aggregate information, `add_break_point` checks if 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 such leaf and to update the base tuple consistently.

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 removal can not be performed, there is actually no stale breakpoint in the U_{tree}. In such a case, the U_{tree} contains only potential breakpoints, that, in turn, are due only to flows becoming idle. A flow can cause more than one breakpoint during any time interval only if the flow becomes idle and then backlogged again at least once during such time interval. Therefore, when the U_{tree} does not contain any stale breakpoint, it is representing a time interval containing at most one break instant per flow.

In the end, since there are N flows in the system and the U_{tree} is balanced, the depth of the U_{tree} never exceeds $O(\log N)$, and `add_break_point` has $O(\log N)$ complexity.

The same comments done so far for `bal_tree_insert`, `bal-tree_rem_fixup` and `add_break_point` apply also to `bal_tree_remove`, `bal_tree_rem_fixup` and `rem_break-point` (also briefly commented in Fig. 6).

5. BALANCED TREES

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

Finally, in practical systems, timestamps are represented by a finite number of bits: in the last subsection we will show how to handle wraparound problems.

5.1 Statistical balancing: Patricia Trees

Instead of the ordering between labels, a search tree can be organized as a function of the labels representation as a sequence of digits. This is the main idea behind *tries* [16], a well know (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 (DTree for short) containing N values is a binary tree in which each leaf is labelled with the binary representation of each value (there is one leaf per value), whereas each internal node is labelled with the common *prefix* of the labels of all the leaves stored in the subtree rooted at such node.

The U_{tree} can be implemented as an augmented DTree in which each leaf is labelled with the binary representation of the break value it contains, and each internal node is labelled with the common prefix of all the break values stored in its subtree. If we imagine to add such prefix to each internal node, then Figures 3.A.2, 3.B.2 and 3.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 they 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. Furthermore, 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 DTree is $O(\log N)$ [16, 18].

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, it is easy to understand that `bal_tree_ins_fixup` and `bal_tree_rem_fixup` are empty functions.

DTrees allow a further optimization. Let node L be the root of a subtree to remove, node P be the father of such node, and node R be the other child of node P. If node R would be the only child of node P, their labels 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, whereas each node of such subtree 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, such node is recycled from the head Z of the list. If node Z has further free children, they are inserted as the first and the second element of the list. The cost of insertions into and extractions from the list is, obviously, $O(1)$.

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).

In Section 7 we will show that, when a DTree is used to implement L-GPS in a real system, also its maximum depth is $O(\log N)$ and it has small constants.

5.2 Deterministic balancing: Red-black Trees

As shown in the previous subsection, DTrees have $O(\log N)$ depth in practical cases. Anyway, if the depth of a DTree bumps up to the number of bits in the node labels in some pathological cases, then a Red-black Tree can be used instead [17].

Red-black Trees are balanced search trees based on comparisons between keys. Each node is labelled 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 such node. Two special fix up functions, invoked, respectively, after each insertion or extraction, guarantee that the maximum depth of a Red-black Tree containing K nodes is equal to $\lceil 2 \cdot \log_2(K+1) \rceil$ [17]. Furthermore, such operations have logarithmic complexity with small constants [17].

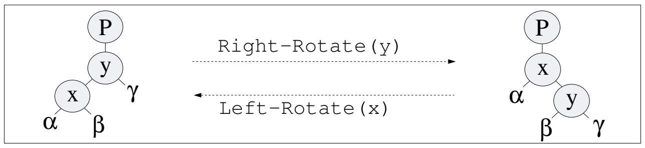

Figure 7: The rotation operations performed by the fix up functions in a Red-black Tree. The letters α, β and γ represent arbitrary subtrees.

The U_{tree} can be implemented as an augmented Red-black Tree in which each leaf is labelled with the break value it contains, and each internal node is labelled with the maximum break value stored in the leaves of its left subtree. Since a binary 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 $\lceil 2 \cdot (1 + \log_2 N) \rceil$.

Finally, `bal_tree_ins_fixup` and `bal_tree_rem_fixup` can be obtained with minor modifications to the fix up functions shown at pages 268 and 274 of [17]. The only critical operations performed by such functions are the two *rotations* shown in Fig. 7: 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. 6, lines 3-6) only to the nodes x and y after each rotation.

5.3 Handling wraparound

All the scheduling algorithms based on packet timestamping and comparison must face the problem of the *wraparound*. In practical systems, the virtual time can be represented as n bits positive integer numbers (or fixed point numbers). Using modular arithmetic, values can be compared without ambiguity even in case of wraparound, provided that the difference between them is less than 2^{n-1}.

As shown in Sec. 2, the maximum deviation of WF^{2}Q with respect to the GPS service is equal to L_{max}. In terms of normalized service, it implies

$$|S_i(t) - V(t)| \leq \frac{L_{max}}{\phi_i} \ \forall i, \forall t \tag{8}$$

known as the Globally Bounded Timestamp (GBT) property [19]. Hence, considering also the second one of Eq. (3), the maximum difference between system virtual time and flows virtual start/finish times is $2 \cdot \frac{L_{max}}{\phi_{min}}$, while the maximum difference between system virtual time and flows virtual unbacking times is $(m+1) \cdot \frac{L_{max}}{\phi_{min}}$, where m is the maximum number of packets per flow queue, and ϕ_{min} is the minimum weight in the system.

Therefore, if WF^{2}Q is implemented using the classical algorithm for GPS simulation, that deals only with virtual start and finish times, the virtual time can be represented on a number of bits n' such that $2^{n'-1} \geq 2 \cdot \frac{L_{max}}{\phi_{min}}$. On the contrary, since the U_{tree} contains virtual unbacking times, the virtual time must be represented using $n = n' + \lceil \log_2 \frac{m+1}{2} \rceil$ bits in case of L-GPS.

Once n has been properly dimensioned, timestamps wraparound causes no problem in search trees based on comparisons between

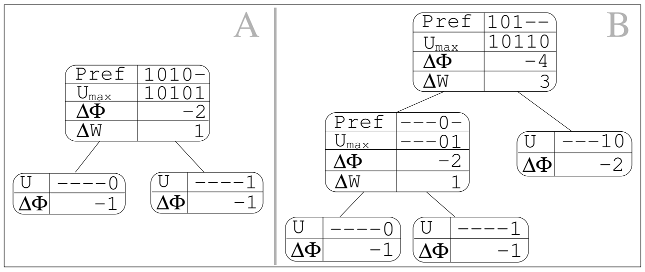

Figure 8: Modified DTrees.

keys (as Red-black Trees). In contrast, the node hierarchy in a DTree depends on the binary representation of timestamps. Consider the first virtual unbacking time U – from system startup at time 0 – whose absolute value exceeds the maximum value allowed ($2^n - 1$). Since U representation wraps, the DTree insertion algorithm would put U in the place appropriate to its binary representation, as if U was smaller than all the other values stored in the DTree.

The problem can be easily overcome in the following manner. Since the difference between all the values stored in the U_{tree} is always smaller than 2^{n-1}, then, when U representation wraps, all such values are in the interval $[2^{n-1},\ 2^n - 1]$ (i.e. the most significant bit in their binary representation is set to 1). Then, the wrapping of U representation can be easily discovered by checking if the value of U is smaller than 2^{n-1} (i.e. most significant bit set to 0).

If true, we subtract 2^{n-1} to the current value of the system virtual time and to every virtual unbacking time stored in the U_{tree}, which can be accomplished by zeroing the most significant bit of their binary representation. As a consequence, the new value of the system virtual time and of all the virtual unbacking times stored in the U_{tree} will be in the interval $[0,\ 2^{n-1} - 1]$. At this point, the right ordering between all the values can be easily preserved by adding 2^{n-1} to U.

The most significant bit in the binary representation of all the values stored in the U_{tree} can be easily zeroed in $O(1)$ steps, if, instead of the full binary representation of the prefix (the value, in case of a leaf), each node is labelled only with the less significant digits that must be added to the label of its father node to get such binary representation. Figures 8.A and 8.B show the modified versions of the augmented DTrees that implement the U_{tree} in Figures 3.B.2 and 3.C.2 (symbol '-' means non significant digit). By doing so, only the root node will contain the common prefix of the binary representation of all the values stored in the U_{tree}. As a consequence, the most significant bit in the representations of such values can be zeroed by simply zeroing the most significant bit in the label of the root node.

6. L-WF²Q

In this section we describe L-WF^{2}Q, an implementation of WF^{2}Q with $O(\log N)$ complexity and small constants. The algorithm is shown in Fig. 9. The code of the functions `enqueue` and `dequeue` can be divided into two parts: the first part (`enqueue` lines 3-12, `dequeue` lines 31-36) 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 13-26, `dequeue` lines 37-38) deals with the shape data structure and evidences a further improvement on L-GPS, based on the fact that, according to (8)

```
1  enqueue(pkt: in)    // 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.weight)         // f.U is near
15     // if non NULL, f.Uleaf points to the leaf
16     // that contains f.U
17     if (f.Uleaf == NULL ) {  // flow becomes backlogged
18       f.Uleaf = add_break_point(V, f.S, f.phi) ;
19       f.Uleaf = add_break_point(V, f.U, -f.phi) ;
20     }
21     else {               // move f.U to the right place
22       f.Uleaf = rem_break_point(f.Uleaf) ;
23       f.Uleaf = add_break_point(V, f.U, -f.phi) ;
24   }
25   else if (f.Uleaf != NULL)     // f.U is no more near,
26     rem_break_point(f.Uleaf) ;   // remove from Utree
27 }
28
29 packet dequeue() // invoked when the link is avail.
30 {
31   pkt = schedule_next() ;                      // Def.  1
32   f = find_flow(pkt) ;     // find the flow owning pkt
33   head_remove(f) ; // rem. pkt at the head of f queue
34   if (not is_empty(f)) {     // update flow timestamps
35     f.S = head(f).S ;          // f.U could become near
36     f.F = head(f).F ;
37     if (f.Uleaf == NULL and f.U <= f.S + Lmax/f.phi)
38       f.Uleaf = add_break_point(f.U, -f.phi) ;
39   }
40   return pkt ;
41 }
```

Figure 9: L-WF^{2}Q.

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

Hence $U_i(t_{new})$ can constitute an actual break value at time t_{new} only if $U_i(t_{new}) - S_i(t_{new}) \le \frac{L_{max}}{\phi_i}$.

We define as *near* the virtual unbacking times that meet the just mentioned condition. It is easy to understand that the system 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 (`enqueue` line 14, `dequeue` line 37), which reduces the depth of the U_{tree}.

The effectiveness of such improvement during high congestion periods is shown through simulations in the next section.

7. SIMULATION RESULTS

As shown in Subsection 5.1, DTrees are very simple to handle, and they allow an efficient implementation of L-GPS. But, whereas 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.

To show the actual performance of a DTree in practical cases, we simulated the operation of L-WF^{2}Q when L-GPS is implemented with a DTree, and the virtual unbacking times are filtered as shown in the previous subsection.

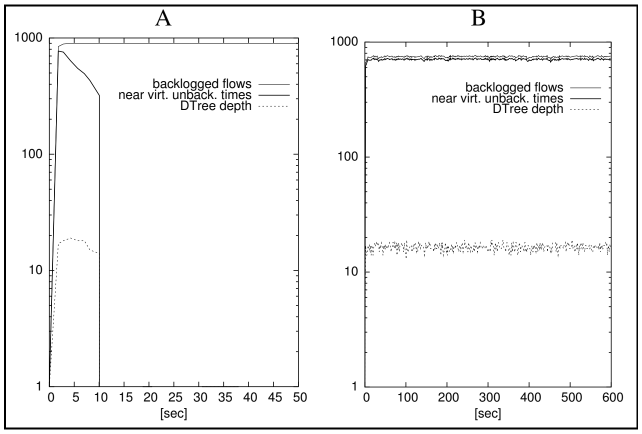

Figure 10: System evolution in case of: A) generic scenario with offered load greater than the link capacity, B) Scenario 5.

We took some statistics on the depth of the U_{tree}, and we compared our results with the ones guaranteed by the use of an ideal perfectly balanced tree or a Red-black Tree.

We used the *ns-2* network simulator [21]. The environment consisted of a node with a 10 Mbps output link. We simulated the following 5 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 that occurs in an Internet router according to [22]. 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 [23] codec 723 (30 bytes payload, 22 packets per sec, 40 bytes IP/UDP/RTP header).

4) 160 Video sources (MPEG-4 coding), transmitting real video traffic traces taken from [24].

5) A mix of the previous traffic sources: 20 FTP sources, 400 asynchronous 10Kbps CBR sources with rate 10 Kbps, 350 VoIP sources, 20 Video sources.

During each simulation we took *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.

As a general result we found that the number of backlogged flows and, hence, the frequency of breakpoints is very small if the offered load is smaller than the link capacity, whereas, if the offered load is larger 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. Besides, the more the backlog increases, the more the filtering becomes effective: Fig. 10.A shows this phenomenon in case of offered load 20% larger than the link capacity.

As a consequence, for each scenario (except for scenario 1), the number of sources and the rate of each source had to be fine tuned to achieve the maximum frequency of breakpoints. Fig. 10.B shows the evolution of the system in case of Scenario 5 (qualitatively similar to the ones of scenarios 2, 3 and 4). Apart from a very short initial transitory period, the number of near virtual unbacking times recorded in each snapshot is roughly equal to the total number of flows.

In order to calculate some statistics, we repeated each simulation 10 times and, for each simulation, we considered only the steady time interval (e.g. $[100,\ 600]$ in Fig. 10.B).

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	160	9.61	0.04	13	8	1.62	14
5	790	16.47	0.06	20	11	1.82	22

Table 2: Statistics collected for each scenario

Table 2 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 such 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 ($\lceil 2 \cdot (1 + \log_2 N_{max}) \rceil$, Subsection 5.2).

Whereas the results for scenario 1 are a consequence of the filtering of virtual unbacking times, in all the other cases the mean depth and the (sample) maximum depth of the augmented DTree is within a factor 2 with respect to the maximum depth of a perfectly balanced tree.

8. CONCLUSIONS

In this paper we showed how a GPS server can be simulated with sub-linear complexity by maintaining aggregate information in an *ad hoc* data structure. In particular, we proposed L-GPS, a new algorithm for computing the state of the simulated GPS server in $O(\log N)$ steps, and we showed that such algorithm provides a straightforward $O(\log N)$ implementation of WF^{2}Q (L-WF^{2}Q).

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

From a theoretical point of view, we reduced 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.

From a practical point of view, we reduced the complexity of a scheduler, WF^{2}Q, that provides a very smooth service, suitable for real time adaptive applications (such as video streaming), and feedback based applications (such as congestion control).

9. ACKNOWLEDGEMENTS

I would like to thank Prof. Luigi Rizzo for his helpful comments and his suggestions on the presentation of the algorithm. I would like also to thank Giovanni Stea for continuously encouraging me, and providing valuable suggestions. Finally, I would like to thank Paolo Gai, Michele Cirinei, my “shepherd” Ion Stoica, and the anonymous referees for their useful remarks.

10. 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] <www.isi.edu/nsnam/ns/>.

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

[23] <www.cisco.com>.

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

APPENDIX

Proof of Theorem 1

We will 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 potential 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. 4). We can write:

$$W(t_1, t^P_{max}) \quad = \quad W(t_1, t^L_{max}) + W(t^L_{max}, t^R_{max}) \qquad (9)$$

For the base case suppose that both nodes L and R are leaves: according to (2), (9) becomes

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

Since, according to Def. 2,

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

we have

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

For the inductive step, suppose that P is a generic internal node, and that Eq. (7) holds for both its children.

Since there is no break instant between t^L_{max} and t^R_{min}, and considering (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 (9)

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