On the Computational Complexity of Maintaining GPS Clock in Packet Scheduling

Qi (George) Zhao   Jun (Jim) Xu
College of Computing
Georgia Institute of Technology
{qzhao, jx}@cc.gatech.edu

***Abstract*— Packet scheduling is an important mechanism to provide QoS guarantees in data networks. A scheduling algorithm generally consists of two functions: one estimates how the GPS (General Processor Sharing) clock progresses with respect to the real time; the other decides the order of serving packets based on an estimation of their GPS start/finish times. In this work, we answer important open questions concerning the computational complexity of performing the first function. We systematically study the complexity of computing the GPS virtual start/finish times of the packets, which has long been believed to be $\Omega(n)$ per packet but has never been either proved or refuted. We also answer several other related questions such as "whether the complexity can be lower if the only thing that needs to be computed is the relative order of the GPS finish times of the packets rather than their exact values?"**

I. Introduction

Service discipline (i.e., packet scheduling) is an important mechanism to provide QoS guarantees such as end-to-end delay bounds and fair bandwidth allocation in computer networks [4], [8], [15]. The perfect service discipline to provide delay and fairness guarantees is General Processor Sharing (GPS) [4], [9]. In a GPS scheduler, all backlogged sessions are served simultaneously in a weighted fashion as follows. In a link of rate r served by a GPS scheduler, each session i is assigned a weight value ϕ_i. Each backlogged session i at every moment t is served simultaneously at rate $r_i = r\phi_i/(\sum_{j \in B(t)} \phi_j)$, where $B(t)$ is the set of sessions that are backlogged at time t. However, GPS is not a realistic service discipline since in a packet network, service is performed packet-by-packet, rather than "bit by bit." Nevertheless, GPS serves as a reference scheduler with which real-world packet-by-packet service disciplines (e.g., WFQ [4]) can be compared in terms of delay and fairness.

A real-world packet-by-packet service discipline typically consists of the following two functions.

1) **Tracking GPS time:** This function is to track the progress of GPS virtual time (described later) with respect to the real time. Its main objective is to estimate the GPS virtual start and finish times of a packet, which are the times that a packet should have started and finished service respectively if served by a GPS scheduler.
2) **Scheduling according to GPS clock:** This function is to schedule the packets based on the estimation of their GPS virtual finish/start times. For example, WFQ selects the one with the lowest GPS virtual finish time among the packets currently in queue to serve next.

We study an open problem concerning the complexity lower bound for computing the GPS virtual finish times of the packets. This complexity has long been believed to be $\Omega(n)$ per packe [11], [12], [3], [9], where n is the number of sessions. For this reason, many scheduling algorithms such as WF^2Q+ [2], SFQ [7], FFQ [11], and $SCFQ$ [6] only approximate the GPS clock (with certain error), trading accuracy for lower complexity. However, it has never been carefully studied whether the complexity lower bound of tracking GPS clock perfectly is indeed $\Omega(n)$ per packet. Our work not only settles this question, but also leads to other surprising discoveries.

In the sequel, we will distinguish *service discipline* from *scheduling algorithm*. The former refers to the policy as to which packet should be served next, whereas the latter refers to the actual mechanism to carry out this policy. For example, a WFQ service discipline says that packets should be scheduled in the sorted order of their GPS virtual finish times. A WFQ algorithm, on the other hand, specifies the data structure and algorithm that is used to track the GPS clock and to sort the GPS timestamps. This distinction is critical in the context of this paper, since we will pose and answer questions such as "Is this scheduling algorithm asymptotically optimal in computational complexity for carrying out the said service discipline?"

Tracking the GPS time is an important task in service disciplines. How well this task is performed directly affects the QoS guarantees provided. For example, it is shown in our prior work [14] that, among all the service disciplines, only WFQ and WF^2Q provides a tight GPS-relative delay (defined in Section II.A) of $O(1)$, because they keep perfect track of the GPS clock. All other disciplines, such as [11], [6], [2], [7], only approximate the GPS clock (with certain error), resulting in much less tight delay bounds.

This prompts us to study the following open problem: "What is the complexity lower bound for computing the GPS virtual finish times of the packets?" This complexity has long been considered as $\Omega(n)$ per packet [11], [12], [3]. For this reason, WFQ and WF^2Q are considered expensive and impractical when the number of sessions are large, and computationally less expensive algorithms such as [11], [12], [3] are proposed to approximate the GPS clock. Therefore, it is very interesting to find out whether $\Omega(n)$ per packet is indeed the complexity lower bound for tracking GPS clock, which decides whether such an unfortunate tradeoff between accuracy and complexity is justified.

This, in turn, leads to another interesting question. Note that, in WFQ and WF^2Q, one only needs to establish the relative orders among GPS virtual finish times. So we wonder whether we can establish this order relationship without explicitly computing their exact values, and hopefully the complexity can be smaller this way. Unfortunately, we prove that such a "shortcut" does not exist: establishing such order relationship is at least as asymptotically expensive as computing their exact values.

Then we establish the complexity lower bound of $\Omega(\log n)$ per packet for establishing the relative order of the GPS virtual finish times, under the linear decision tree model (described later). It seems that this lower bound is not tight since we suspect that it is $\Omega(n)$ per packet. However, we discover an existing algorithm [8] that matches the lower bound of $\Omega(\log n)$ per packet. This algorithm has seldom been mentioned later in literature, partly because it is an integral part of a theoretical exploration of a related yet different topic. Our discovery has an immediate practical implication: WFQ and WF^2Q implementations based on this algorithm are optimal among all implementations of WFQ and WF^2Q service disciplines.

Unfortunately, we discover an important subtlety in this $\Omega(\log n)$ algorithm, which has never been discussed in [8] or any other open literature. A computation timing constraint we refer to as "mandatory lazy evaluation" has to be satisfied for the algorithm to have $\Omega(\log n)$ complexity. We show that theory and practice clash on this constraint: in theory the complexity of this algorithm is strictly $\Omega(\log n)$ per packet, but in practice the worst case complexity could be as high as $\Omega(n)$ per packet due to this "mandatory laziness." This discovery completely clarifies and answers the question "$\Omega(\log n)$ or $\Omega(n)$?" (per packet) concerning the complexity of tracking the GPS clock.

Finally, we establish the most important result of this paper: the complexity lower bound of computing the GPS virtual finish times is $\Omega(\log n)$ per packet. This result is shown by establishing an interesting tradeoff between the number of comparison operations and the number of other computations (e.g., additions). We show that, for tracking the GPS clock, any algorithm has to pay either $\Omega(\log n)$ comparisons or $\Omega(\log n)$ other computations, per packet. Therefore, the overall complexity of the algorithm, which is the sum of both, has to be $\Omega(\log n)$ per packet. Our work, combined with the aforementioned $\Omega(\log n)$ algorithm [8], completely settles the long-standing open problem concerning the computational complexity of tracking the GPS clock, in theory[1].

The rest of the paper is organized as follows. In Section II, we introduce the computational models and assumptions we will use in proving our results. In Section III, we study the complexity of tracking the GPS clock. Section IV concludes the paper.

[1]In practice, we will always be "haunted" by the aforementioned subtlety of "mandatory lazy evaluation."

II. Related Work and Background

In this section, we introduce the background knowledge and related work on packet scheduling and computational complexity. Their relevance to the objective of this work will be explicitly stated.

A. Background on GPS and packet-by-packet scheduling algorithms

We have introduced GPS, an idealized service discipline with which other service disciplines can be compared in terms of QoS guarantees. One important property of GPS, proved in [9], is that it can guarantee tight end-to-end delay bounds to traffic that is leaky-bucket [13] constrained. It has been shown that scheduling algorithms can provide similar end-to-end delay bounds if their packet service does not significantly lag behind that of GPS [9]. This motivates the definition of GPS-relative delay (introduced in [9] and [8] but formally defined in [14]) and the study of existing service disciplines on this property.

For each packet p, the *GPS-relative delay* of a packet p under a particular scheduler ALG is defined as $max(0, F_p^{ALG} - F_p^{GPS})$, where F_p^{ALG} and F_p^{GPS} are the times when the packet p finishes service in the ALG scheduler and in the GPS scheduler, respectively. It has been shown in [9] and [1] respectively that WFQ and WF^2Q schedulers both have a worst-case GPS-relative delay of $\frac{L_{max}}{r}$, where L_{max} is the maximum packet size and r is the total link bandwidth. That is, for each packet p,

$$F_p^{WFQ} - F_p^{GPS} \le \frac{L_{max}}{r} \tag{1}$$

$$F_p^{WF^2Q} - F_p^{GPS} \le \frac{L_{max}}{r} \tag{2}$$

Using the convention of our prior work [14], we say that the delay bound is $O(1)$ since L_{max} and r can be viewed as constants independent of the number of sessions n. WFQ and WF^2Q achieve this $O(1)$ GPS-relative delay bound by (a) keeping perfect track of the GPS clock and (b) picking the packet with smallest GPS virtual finish time, among all (in WFQ) or all eligible (in WF^2Q) head-of-line (HOL) packets, to serve next. Here HOL packets are the first packets from each session that are currently in queue. This paper essentially studies whether steps (a) and (b) are necessary and if yes, how to carry out the first step in an optimal way.

B. On tracking the GPS clock

The GPS virtual time $V(t)$, as a function of real time t is calculated as follows.

$$\begin{aligned} V(0) &= 0 && (3)\\ V(t_{j-1}+\tau) &= V(t_{j-1}) + \frac{\tau}{\sum_{i\in B_j(t)} \phi_i} && (4)\\ \tau &\le t_j - t_{j-1}, j = 2, 3, \cdots \end{aligned}$$

Here $\{t_j\}_{j=1,2,...}$ are the times at which two types of events happen under GPS: (1) the service starts for a new packet, and (2) the service finishes for a packet currently in queue. Let A_i^k be the real time that the k^{th} packet of the i^{th} session arrives and L_i^k be its length. Let S_i^k and F_i^k be the virtual times when it should have started and finished service under GPS, respectively. Then one can show that S and F are calculated as follows.

$$S_i^k = max\{F_i^{k-1}, V(A_i^k)\} \tag{5}$$

$$F_i^k = S_i^k + \frac{L_i^k}{\phi_i} \tag{6}$$

Note that the GPS virtual finish time of a packet is calculated as soon as it arrives as above. However, the real time corresponding to this virtual finish time is not determined until later (explained later). In service disciplines, tracking of the GPS clock is to find the GPS *virtual* and *real* finish times of each packet. Although GPS virtual finish times are all that are needed in WFQ for scheduling, the GPS real finish times are equally important since they determine the GPS virtual finish times of future packets, as shown in formulas (3) and (4). In this paper, we solve the open problem concerning the complexity lower bound for computing the GPS virtual and real finish times.

C. On Computational Complexity

Computational complexity of a problem is defined under a computational model that specifies what operations are allowed in solving the problem and how they are "charged" in terms of complexity. Same as in our prior work [14], we adopt a standard and commonly-used computational model in proving lower bounds: the *decision tree*. A decision tree in general takes as input a list of real variables $\{x_i\}_{1\leq i\leq n}$. Each internal and external (leaf) node of the tree is labeled with a predicate of these inputs. The algorithm starts execution at the root node. In general, when control is centered at any internal node, the predicate labeling that node is evaluated, and the program control is passed to its left or right child when the value is "yes" or "no" respectively. Before the control is switched over, the program is allowed to execute *unlimited number* of sequential operations such as data movements and arithmetic operations. In particular, the program is allowed to store all results (i.e., no constraint on storage space) from prior computations. When program control reaches a leaf node, the predicate there is evaluated and its result is considered as the output of the program. The complexity of such an algorithm is defined as the depth of the tree, which is simply the number of predicates that needs to be evaluated in the *worst case*. Fig. 1 shows a simple decision tree with four nodes. Each P_i $(1 \leq i \leq 4)$ is a predicate of the inputs.

Allowing different types of predicates to be used in the decision tree results in models of different computational powers. *Throughout this paper, we consider the decision tree that allows the comparisons between linear functions of inputs, known as linear decision tree (LDT)* [5]. In this model, given inputs $\{x_i\}_{1\leq i\leq n}$, each predicate allowed by the decision tree

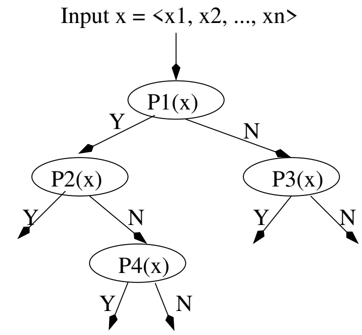

Fig. 1. Decision tree computational model

is in the form of "$h(x_1, x_2, \cdots, x_n) \geq 0$?", where h is a linear function (defined below) of the inputs $\{x_i\}_{1\leq i\leq n}$.

Definition 1 (Linear Function): A linear function f of the variables $\{x_i\}_{1\leq i\leq n}$ is defined as $f(x_1, x_2, \cdots, x_n) = \sum_{i=1}^{n} a_i x_i + a_0$, where $\{a_i\}_{0\leq i\leq n}$ are real constants.

In the context of this work, the inputs will be the lengths and the arrival times of the packets. Note that the linear decision tree model is quite generous: functions like f in the above definition may take up to $O(n)$ steps to compute, but the model charges only "1" for each of them. Due to this generosity, for certain computation problems, we may not be able to obtain a tight lower bound using this model. We, however, stick to this model since it provides mathematical tools (convexity arguments) for solving lower bounds. In addition, since the GPS virtual finish times of the packets are linear functions of the inputs, shown by the following theorem, this model captures all computation powers of a router for tracking the GPS clock and for scheduling packets.

Theorem 1: The GPS virtual finish times of the packets are all linear functions of the inputs.

Proof: Note that we only need to show that the virtual start times (S_i^k) are linear functions of the inputs since the virtual finish time (F_i^k) is equal to $S_i^k + \frac{L_i^k}{\phi_i}$ according to (6). We prove this result by induction on the packet arrival events to the system. Since the first packet arrives at time 0, according to (3), its virtual start time is 0, which is a linear function of the inputs. Now let's assume that S_i^m and F_i^m (for $m < k$) are all linear functions of the inputs. Then for the S_i^k,

$$S_i^k = max\{F_i^{k-1}, V(A_i^k)\}$$

Let t denote A_i^k and t' denote the real time when previous event (arrival or departure) happened. If $F_i^{k-1} > V(t)$, S_i^k is equal to F_i^{k-1}, which is a linear function by the induction hypothesis; otherwise, S_i^k is equal to $V(t)$. Thus,

$$S_i^k = V(t) = V(t') + \frac{t - t'}{\sum_{i\in B(t)} \phi_i}$$

In this case $V(t')$ is either the virtual finish time of a packet or the virtual start time of a packet. In both situations it is a linear function by the induction hypothesis. And note the set of sessions that are active in the interval $(t-t')$, i.e., $B(t)$, is a fixed constant. Thus S_i^k is also a linear function. Therefore the GPS virtual finish times of the packets are all linear functions of the inputs. ∎

D. Related work on the complexity lower bound of packet scheduling

Our prior paper is the first work in studying the tradeoff between delay bound and computational complexity [14]. In [14], we have established that the complexity lower bound for a service discipline to provide $O(1)$ or $O(n^a)$ $(0 < a < 1)$ GPS-relative delay bound is $\Omega(\log n)$ per packet. This bound is established under a weaker decision tree model that only allows direct comparisons between its inputs. In other words, its comparisons are all in the forms of "$x_i - x_j \geq 0$?". This model is strictly weaker than LDT since $x_i - x_j$ is clearly a linear function. However, for the particular class of instances that are used in establishing the lower bounds in [14], the weaker computational model is reasonable in the sense that existing scheduling algorithms are able to provide $O(1)$ GPS-relative delay bounds using only such predicates.

Whether the complexity lower bound of tracking the GPS clock is $\Omega(n)$ per packet was posed as an open problem by [14]. Since its focus is on the computational complexity of scheduling packets, it skillfully avoids answering this question by introducing the concept of CBFS (Continuously Backlogged Fair Sharing) condition[2]. It shows that under the CBFS condition, the GPS finish times can be computed easily in $O(1)$ and becomes a nonissue in studying the complexity of scheduling.

E. Related theorems in prior work

In the proofs of this paper, we will use a lemma from [14], which says the complexity lower bound for determining whether a given input vector $(x_1, x_2, ..., x_n)$ belongs to the following set $\mathcal{S}$ is $n \log_2 n - o(n \log_2 n)$, under the linear decision tree. Here $\mathcal{S} = \{(y_1, y_2, ..., y_n) \in R^n$: there exists a permutation π of 1, ..., n such that $\frac{iL_{max}}{n+1} - \delta < y_{\pi(i)} < \frac{iL_{max}}{n+1} + \delta$, $i = 1, 2, \cdots, n\}$. Here $L_{max} > 0$ is the maximum packet size and δ $(0 < \delta < \frac{L_{max}}{3(n+1)})$ is a "small" real constant. Here is the lemma from [14].

Lemma 1: Under the linear decision tree model, given the inputs $\{x_i\}_{1 \leq i \leq n}$, determining whether $(x_1, x_2, \cdots, x_n) \in \mathcal{S}$ requires at least $n \log_2 n$ - $o(n \log_2 n)$ linear comparisons.

III. On The Complexity of Tracking GPS Clock

The focus of this section is to clarify and establish the fundamental algorithmic complexity of tracking the GPS clock perfectly in packet scheduling. Recall that both WFQ and WF^2Q service disciplines track GPS clock perfectly, and both are able to achieve a tight GPS-relative delay bound of $O(1)$. All other service disciplines, on the other hand, only estimate GPS clock (with certain error), and it is shown in [14] that none of them can provide the same tight GPS-relative delay bound. Intuitively there is a causal relationship between the perfect tracking of GPS clock and the tightness of the delay bound.

A. GPS clock tracking and its complexity question

The computational complexity lower bound of tracking the GPS clock has long been considered $\Omega(n)$ per packet [11], [12], [3]. For this reason, both WFQ and WF^2Q are considered to be impractical, and several algorithms that approximate GPS clock at much lower complexity, such as [11], [6], [2], [7], are proposed. However, as mentioned above, none of these algorithms can achieve a tight GPS-relative delay bound of $O(1)$.

We ask the following fundamental question: is $\Omega(n)$ per packet the complexity lower bound for tracking the GPS clock perfectly? Settling this question is important in two aspects.

- First, if this complexity is indeed $\Omega(n)$ per packet, then the complexity lower bound of $\Omega(\log n)$ per packet established in [14] for scheduling the GPS timestamps to achieve tight GPS-relative delay bounds becomes less relevant, since the combined complexity of scheduling is $\Omega(n)$ per packet anyway. Note that [14] avoids answering this question, as their results are established under a CBFS condition (shown later) in which GPS tracking cost becomes $O(1)$ per packet. This complexity question leads to two other interesting questions. Is GPS clock tracking a necessary step for implementing WFQ and WF^2Q? (i.e., is there a "shortcut"?) Or is this step necessary for providing $O(1)$ GPS-relative delay? We will answer the first question in full. The second question remains an open problem.
- Second, if there is an efficient algorithm of complexity $O(\log n)$ per packet, for tracking GPS time, then WFQ and WF^2Q may become practical since their overall cost will be $O(\log n)$ per packet!

Our findings turn out to be a mixed blessing: in theory (i.e., the strict sense of computational complexity), the complexity lower bound is $\Omega(\log n)$per packet and there is an algorithm that matches the bound. In practice, however, it can be as expensive as $\Omega(n)$ per packet in the worst-case due to a "mandatory lazy evaluation" subtlety associated with the algorithm. One of our contributions is to discover and clarify this subtlety.

B. Is there a "shortcut"?

Note that in WFQ and WF^2Q, we are only concerned with the relative order of GPS virtual/real finish times rather than their exact values. This leads us to ask the following question: can the complexity of establishing such order relationships be smaller than that of computing the GPS virtual/real finish times? In other words, is there a "shortcut" that allows us to establish such orders without explicitly computing their values? Such shortcuts may be possible in some computations. For a trivial example, to know whether $x + z < y + z$ holds, we may just compare x with y without adding up $x + z$ and $y + z$. Real-world examples can be much more sophisticated since comparisons can be nested deeply in an expression.

[2] A less elegant way to avoid dealing with this question is to use the relativization arguments in theoretical computer science [10].

```
class Packet {
 double rt_arrival, rt_finish, vt_start, vt_finish;
 /* rt_arrival and rt_finish are the real times of the packet's arrival and
    departure (under GPS) respectively. vt_start and vt_finish are its GPS
    virtual start and finish times respectively. Initially rt_arrival field
    is filled by the value of arrival time of the packet and the other
    fields are initialized to 0. */
 int id, len; /* the session id of the packet and the packet length */
};

class Packet_Queue {
 Packet peek(); /* return the first packet in queue without deleting it */
 Packet pop(); /* return the first packet in queue and delete it */
 int empty(); /* return 1 if queue is empty, otherwise return 0 */
};

external Packet_Queue P, Q; /* explained in the text and Fig. 3 */

total_weight = 0;
/* the total weight of all the backlogged sessions between two events */
for (i=0; i < n; i++) backlog[i] = 0;
/* backlog[i] indicates the amount of backlog at session i queue (in GPS
   virtual time). */

while (!Q.empty()) {
  q = Q.peek(); p = P.peek();
  if (total_weight == 0) {
  /* the start of a new busy period */
    total_weight = weight[p.id]; /* weight[i] is φ_i */
    vt_last = 0; /* the virtual time when the last event happens. */
    rt_last = p.rt_arrival; /* the real time of the last event */
    continue;
  }
  vt_temp = vt_last + (p.rt_arrival - rt_last) / total_weight;
  if (q.vt_finish > vt_temp) {
  /* p starts earlier than q finishes */
    p.vt_finish = max(vt_temp, backlog[p.id]) + p.len / weight[p.id];
    vt_last = vt_temp; rt_last = p.rt_arrival;
    if (backlog[p.id] < vt_temp)
      total_weight = total_weight + weight[p.id];
    backlog[p.id] = p.vt_finish;
    P.pop();
  }
  else {
  /* q finishes first before p starts */
    q.rt_finish = rt_last + (q.vt_finish - vt_last) * total_weight;
    rt_last = q.rt_finish; vt_last = q.vt_finish;
    if (backlog[q.id] ≤ vt_last)
      total_weight = total_weight - weight[q.id];
    Q.pop();
  }
}
```

Fig. 2. Algorithm B for GPS time tracking.

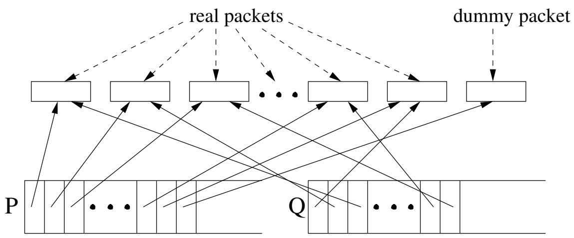

Fig. 3. Queue structure of Algorithm B

Our discovery is surprising: establishing the relative orders among the GPS finish times is at least asymptotically as expensive as computing them, as proved in the following theorem.

Theorem 2: The complexity lower bound of sorting the GPS virtual/real finish times is asymptotically at least as large as that of computing their exact values.

Proof: Our proof proceeds as follows. As before, let n denote the number of active flows. Assume that we know the relative order in which the packets finish service under GPS, which is computed using a *magic algorithm* A with complexity $C(n)$ per packet. We show there is an algorithm B that uses such order to produce the correct GPS virtual/real finish times of the packets, with complexity $O(1)$ per packet. We denote as $D(n)$ the complexity lower bound of computing the GPS finish times per packet. Clearly $D(n) \leq C(n) + O(1)$. However, since $D(n)$ is at least $O(1)$ (just to read the input), this shows that $C(n)$ is at least *asymptotically* as large as $D(n)$.

The reduction algorithm B is shown in Fig. 2. The queue Q contains pointers to these packets sorted in the increasing order of their GPS finish times. This captures our assumption that we know their relative orders. The incoming packet FIFO queue P contains pointers to these packets in the sorted order of their arrival times. Note that pointers in both P and Q refer to the same pool of actual packets, as shown in Fig. 3. For each packet, initially only the real arrival time field (rt_arrival) is set. After the execution of the program, we show that its GPS virtual finish time (vt_finish) and its real finish time (rt_finish) will also be set. This way the mission of computing the GPS real/virtual finish times of the packets is accomplished.

We have used a standard technique in programming language to make the algorithm B succinct. We assume that at the end of queue P, there is a pointer to a "dummy" packet with arrival time $+\infty$. Q, on the other hand, does not contain such a pointer. This "dummy" pointer will "flush" (line 41) all packets in Q before itself gets popped. This "flushing" considerably simplifies the inspection of boundary conditions.

We need to prove two things: (a) the complexity of the algorithm is $O(1)$ per packet, and (b) it correctly computes the GPS virtual and real finish times of the packets.

Proof of (a): Note that during each "while" iteration except for the first one, which goes through line 16 to 43, there are at least one element popped from either P (line 31) or Q (line 41). Eventually both will become empty and the program terminates. Each iteration clearly takes $O(1)$ time.

Proof of (b): We need to show that the algorithm computes the GPS real/virtual finish times as defined in formulae (3) and (4). We prove this by induction on the number of events. The GPS virtual time when the first event (the first packet arrival) happens is 0 (line 21). Suppose the values of the GPS virtual/real times and other state variables corresponding to the k^{th} event is computed correctly. We need to show that the values corresponding to the $(k+1)^{th}$ event are correctly computed. The $(k+1)^{th}$ event can be one of the following two cases: (i) the finish of a packet from a session i, (ii) the start of a new packet from a previously unbacklogged session.

- In case (i), p arrives after q gets popped, since otherwise its arrival would have been the $(k+1)^{th}$ event. So line 35 through 42 will be executed. Depending on whether the session $q.id$ queue becomes unbacklogged (line 39) or not after the departure of q, the total weight (total_weight) is either decreased or unchanged (line 40). The real finish time of q can now be finalized (line 37) since there are no new arrivals before its departure.
- In case (ii), p arrives earlier than $q's$ departure, and therefore line 26 through 34 will be executed to update the total weight and compute $p's$ GPS virtual start and finish times.

In both cases, the advance of the GPS clock $V(t)$ from the time of the k^{th} event to that of the $(k+1)^{th}$ event conforms to formulas (3) and (4). Therefore, the algorithm should compute the GPS virtual/real times and other state variables correctly for the $(k+1)^{th}$ event and all events by induction. ■

This result is important since it shows that there is no shortcut to computing the GPS virtual finish times for implementing WFQ and WF^2Q.

C. A tight lower bound

Next, we prove the lower bound of $\Omega(\log n)$ per packet for sorting GPS virtual finish times (Theorem 3) under the linear decision tree model. To prove this, we need a lemma showing that the complexity of sorting n numbers is $\Omega(n \log n)$ under the linear decision tree (Lemma 2). Note that the traditional sorting complexity we know only allows comparisons among inputs.

Lemma 2: The complexity lower bound for sorting n distinct positive real numbers x_i, $i = 1, 2, \cdots, n$, is $\Omega(n \log n)$ under the linear decision tree model.

Proof: Suppose sorting these numbers is not $\Omega(n \log n)$. Then we construct an algorithm A for deciding the membership for the aforementioned set $\mathcal{S}$ in Lemma 1, with a complexity that is not $\Omega(n \log n)$. This would contradict Lemma 1. Algorithm A first sorts these numbers using linear comparisons. Then it verifies if the sorted list is in $\mathcal{S}$, and this step apparently only needs $O(n)$ time (check if $\frac{iL_{max}}{n+1} - \delta < y_i < \frac{iL_{max}}{n+1} + \delta$, $i = 1, 2, \cdots, n$). Since $O(n) = o(n \log n)$, the complexity of algorithm A is not $\Omega(n \log n)$ if the sorting part is not $\Omega(n \log n)$. ■

Theorem 3: The complexity lower bound for computing the relative order of the GPS finish times among n packets is $\Omega(n \log n)$ under the linear decision tree model.

Proof: We reduce the problem to sorting. Given a sorting instance $x_i > 0$, $i = 1, 2, \cdots, n$ and $x_i \neq x_j$ when $i \neq j$, we convert it to a packet arrival instance as follows. Let there be n sessions sharing a link of rate 1 with equal weight. A packet of length x_i arrives at time 0 to session i, $i = 1, 2, \cdots, n$. Note that the relative order of the GPS finish times is exactly the order of x_i. Then from Lemma 2 we know that to sort the lengths of n packets requires the cost of $\Omega(n \log n)$. So computing the relative orders of the GPS finish times takes $\Omega(\log n)$ per packet. ∎

Note that the lower bound is not tight unless we can find an algorithm that matches it. Fortunately and surprisingly, we have found an existing algorithm that tracks GPS clock using only $\Omega(\log n)$ per packet complexity[8]. However, it has been seldom mentioned or referenced in current literature, including the algorithms that try to reduce the complexity of GPS tracking using approximation. Part of the reason for this oversight is that [8] is a theoretical endeavor on a related yet different topic.

The algorithm in [8] is too involved to be included here due to lack of space. Its key idea, however, can be explained relatively easily based on our algorithm in Fig. 2. Note that in Fig. 2, once we have Q, we can compute the GPS virtual finish times of the packets with complexity $O(1)$ per packet. The algorithm in [8] essentially keeps a priority queue of the HOL packets to obtain Q, which is $\Omega(\log n)$ per packet. Each session also maintains a per-session FIFO queue. When a packet in an HOL queue should have finished service in GPS, its next packet (if it exists) is moved from its corresponding per session FIFO queue to the HOL priority queue.

The following result shows that if the WFQ is performed by a priority queue and the computation of the GPS clock is performed using the algorithm in [8], then it is the asymptotically optimal implementation of the WFQ service discipline.

Corollary 1: Algorithm in [8] for tracking GPS clock combined with a priority queue to sort GPS virtual finish times is the asymptotically optimal algorithm for implementing WFQ scheduling discipline, under the linear decision tree model.

Proof: Combine the algorithm in Fig. 2, and Theorem 3 on the lower bound of comparing GPS virtual finish times. ∎

D. Mixed blessing

Now that we have an $\Omega(\log n)$ per packet algorithm for WFQ, do we still need to approximate the GPS clock as in [11], [6], [2], [7]? The answer is yes. We found that the new $\Omega(\log n)$ algorithm is a mixed blessing in the following sense. It requires a subtlety we refer to as "mandatory lazy evaluation" for its theoretical complexity to be $\Omega(\log n)$ per packet. In practice, this mandatory laziness may lead to $\Omega(n)$ worst-case complexity. To avoid this worst case, certain "approximation measures" need to be taken. Our contribution here is to have discovered and clarified this subtlety.

The algorithm in [8] implicitly requires a computational timing we refer to as "mandatory lazy evaluation." Suppose a packet p's real finish time is t, provided that there are no other packet arrivals before its departure. Then for an algorithm to be efficient, the computation of this time cannot be finalized until t has passed. The reason is that if this computation is performed at time $t' < t$, and if there is a packet arrival between t' and t, the GPS real finish time of p would have to be recomputed. In practice, this "lazy evaluation" can be implemented as a hardware timer that triggers at time t the computation of its GPS real finish time. However, in the worst case, up to $\Omega(n)$ GPS virtual finish times may be "clustered together" in a small time interval, and a new arrival immediately after the "cluster" can not be processed until all $O(n)$ departures in that cluster get processed. The GPS real finish times of all these departure events need to be computed during this interval due to lazy evaluation. Therefore, the processing and service of the new arrival triggers the computation of $\Omega(n)$ GPS real finish times. This leads to the $\Omega(n)$ worst-case complexity.

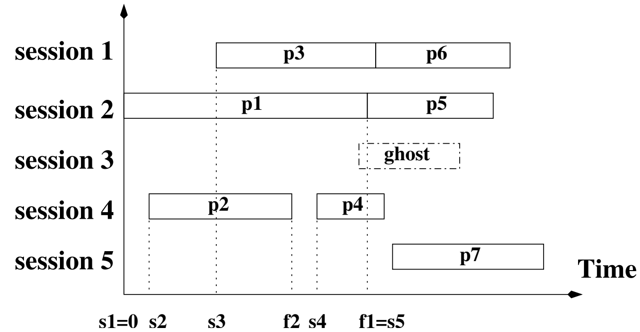

Fig. 4. Why lazy evaluation is necessary?

We illustrate this "lazy evaluation problem" by an example, shown in Fig. 4. The horizontal axis is the GPS timeline. We denote these packets as p_i and their GPS finish and start times as f_i and s_i respectively. In addtion, we define $R(t)$ as the function that returns the virtual time corresponding to virtual time t. In Fig. 4, we show that f_1, f_3 and f_4 are very close to each other and form a "cluster", followed immediately by the arrival of p_7. To be able to compute $R(s_7)$, we need to finish computing the GPS real finish times corresponding to f_1, f_3, and f_4. However, these computations need to be finished "within" the tiny interval of $[f1, s7]$ due to "mandatory lazy evaluation", which may not be possible in real-world finite-speed CPUs. On the other hand, if we compute these real times well in advance, then if the "ghost" packet in the figure does arrive, all these values have to be recomputed.

This presents a classic irony between theory and practice. From the theoretical point of view, it is $\Omega(\log n)$ per packet, since we only count the number of operations. In practice, however, CPU has finite speed and a "cluster" of $\Omega(n)$ packets may need to be processed during a short time interval. We are currently investigating how to alleviate this problem by using approximation techniques. It is not clear whether such techniques will be better than those used in [11], [6], [2], [7]. This is an interesting topic for further study.

E. Tracking GPS is $\Omega(\log n)$ per packet

In this section, we answer the most important and challenging question of this paper: "What is the complexity lower bound for computing the GPS virtual finish times of the packets?" We show that this lower bound is $\Omega(\log n)$ per packet.

Our lower bound is established based on the following set of instances that we will refer to later as $I(n)$. Suppose n sessions share a link of rate 1. There are n arrivals p_1, p_2, ..., p_n at time 0 from sessions 1, 2, ..., n, and their lengths are l_1, l_2, ..., l_n, respectively. The weights of these sessions are w_1, w_2, ..., w_n, respectively. We also assume $l_i/w_i \neq l_j/w_j$ when $i \neq j$. We refer to the set of all such instances of size n mentioned above as $I(n)$. Note also that these weights $w_i's$ are viewed as constants rather than as inputs.

We would like to compute their GPS real finish times. We let π be the unique permutation such that $l_{\pi(1)}/w_{\pi(1)} < l_{\pi(2)}/w_{\pi(2)} < ... < l_{\pi(n)}/w_{\pi(n)}$. It is easy to check that the GPS real finish times of the packets are as follows:

$$\begin{aligned} S(p_{\pi(1)}) &= l_{\pi(1)} * \frac{\sum_{j=1}^{n} w_{\pi(j)}}{w_{\pi(1)}} \\ S(p_{\pi(i)}) &= \frac{l_{\pi(1)}}{w_{\pi(1)}} * \sum_{j=1}^{n} w_{\pi}(j) + \sum_{m=2}^{i} \Big[(\frac{l_{\pi(m)}}{w_{\pi(m)}} \\ &\quad - \frac{l_{\pi(m-1)}}{w_{\pi(m-1)}}) \sum_{j=m}^{n} w_{\pi(j)} \Big], i = 2, 3, ..., n \end{aligned}$$

Let us define x_i for $i = 1, 2, \cdots, n$ as follows.

$$\begin{aligned} x_1 &= l_{\pi(1)} * \frac{\sum_{j=1}^{n} w_{\pi(j)}}{w_{\pi(1)}} \\ x_i &= (\frac{l_{\pi(i)}}{w_{\pi(i)}} - \frac{l_{\pi(i-1)}}{w_{\pi(i-1)}}) \sum_{j=i}^{n} w_{\pi(j)} \\ & \quad i = 2, 3, \cdots, n \end{aligned}$$

And $S_i = S(p_{\pi(i)})$. We need to show that computing $\{S_i\}_{i=1,...,n}$ requires $\Omega(n \log n)$ computation.

Obviously, if we know the right permutation π, we can compute $\{S_i\}_{1 \leq i \leq n}$ in $O(n)$, since $S_{i+1} = S_i + x_{i+1}$ and x_i can be computed in $O(1)$. But computing this π requires $\Omega(n \log n)$ comparisons, as shown in Lemma 2. However, this argument is not a proof for the lower bound since it may be possible to compute S_i without computing π completely (perhaps only computing some partial orders captured in π). To be able to establish the lower bound of $S_i's$, we need a computational model that captures *all possible ways to compute S_i and their costs.*

Our model is the following. We assume that each term x_i can be computed with $O(1)$ computation cost and each addition costs 1. Note we do not gain from this assumption since we are trying to establish the lower bound. If S_j is computed from S_i ($j > i$), i.e., S_j is computed as $S_i + x_{i+1} + x_{i+2} + ... + x_j$, the amount of computation is counted as $j - i$. This part captures the savings we get by using prior computation results. Also, for the computation S_j to use the result of computation S_i, we "must have" learned about the partial order $S_i < S_j$. This model is very natural since it captures all conceivable ways of computing S_j. It also captures the tradeoff between comparison and computation: either one pays for "learning" about the partial orders between $S_i's$ and $S_j's$ in the form of comparisons, or pays for the additions in the form of $S_i + x_{i+1} + x_{i+2} + ... + x_j$.

In the following, we show that no matter how the computation is carried out, either we have to pay $\Omega(n \log n)$ cost in comparison or $\Omega(n \log n)$ in computation, for the worst-case instance in $I(n)$. In other words, the total complexity (computation + comparison) has to be at least $\Omega(n \log n)$. Since there are n packets, the complexity per packet is $\Omega(\log n)$.

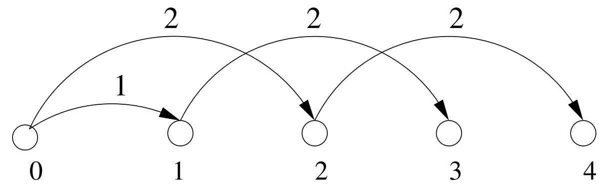

Fig. 5. An example of the GPS computation tree

For the simplicity of stating and proving our result, we use a directed graph to represent the above computational model. Any instance (of computing GPS virtual/real finish times) in $I(n)$ above corresponds to a graph containing $n+1$ points, $0, 1, ..., n$. Here points $1, 2, ..., n$ correspond to the jobs of computating S_1, S_2, ..., and S_n respectively. The point 0 denotes the "null" computation job. There is a directed edge from i to j if and only if S_j is computed from S_i. The cost of this edge is $j - i$, which captures the number of additions for computing S_j from the intermediate result S_i, as explained above. A directed edge from 0 to i means that S_i is computed "from scratch" as $x_1 + x_2 + \cdots + x_i$, and its cost is i. Note that, ignoring the direction, this directed graph is a tree because each computation job has a unique computation path from the point 0. In each such tree, there must be a directed path from 0 to any other point, since every S_i needs to be computed. We refer to this tree as the *GPS computation tree* corresponding to the instance. An example tree is shown in Fig. 5. In Fig. 5, S_3 and S_4 are computed from S_1 and S_2 respectively, and S_1 and S_2 are computed "from scratch." The numbers associated with the links are the computation costs.

We denote as $GPS(n)$ the worst case complexity for computing $S_i's$, among all instances in $I(n)$. Note that for each instance in $I(n)$, its computational complexity includes both additions and comparisons. With this graph model, our result can be formulated as the following theorem.

Theorem 4: $GPS(n) = \Omega(n \log n)$

Proof: We prove by contradiction. Suppose $GPS(n)$ is not $\Omega(n \log n)$. In other words, for any $N > 0$ and $\epsilon > 0$, there exists $n' > N$ such that, the worst case complexity (comparisons plus computations) for GPS computation trees

```
1. FIFO_PacketQueue Q ← ∅;
2. Sorted_Array P ← {0};
3. while (!P.empty()) {
4.   p ← P.min();
5.   delete p from P;
6.   if p has children
7.     Sort p's children and merge into P;
8.   Append p to Q;
9. }
```

Fig. 6. Algorithm B for getting the sorted sequence from the GPS computation tree

of size n' is no more than $\epsilon n' \log n'$. We denote this as assumption (I).

We denote as $SORT(n)$ the worst-case complexity of sorting the GPS virtual finish times for the aforementioned instance set $I(n)$. We know from Theorem 3 that $SORT(n)$ is $n \log_2 n - o(n \log_2 n)$, under the linear decision tree model. So there exists N_1 such that for all $n > N_1$, $SORT(n) > \frac{5}{6} n \log_2 n$. We also choose N_2 such that when $n > N_2$, the value of the function $2n$ is no more than $\frac{1}{6} n \log_2 n$. We let $N_3 = max(N_1, N_2)$. We know from assumption (I) that there exists $n' > N_3$ such that $GPS(n') \leq \frac{1}{6} n' \log_2 n'$, denoted as fact (II). Note that since $n' > N_3 > N_1$, $SORT(n') > \frac{5}{6} n' \log_2 n'$. This is denoted as fact (III).

We show that we can construct an algorithm A which, given any instance of size n' as above, sorts the GPS virtual finish times of these packets using no more than $\frac{5}{6} n' \log n'$ comparisons. This clearly contradicts the fact (III) above. The algorithm A runs as follows. For each packet arrival instance of size n', A first tries to compute the finish times of the packets. This results in a GPS computation tree corresponding to this instance. Since $GPS(n') \leq \frac{1}{6} n' \log_2 n'$ (fact (II) above), neither the number of comparisons nor the number of computations in this computation tree can be more than $\frac{1}{6} n' \log_2 n'$. Then A calls another algorithm B, which takes as its input a GPS computation tree, and outputs the GPS finish times in the sorted order. We will show that the complexity of algorithm B is no more than $\frac{4}{6} n' \log_2 n'$ in the worst case. So the complexity of the algorithm A is no more than $\frac{1}{6} n' \log_2 n' + \frac{4}{6} n' \log_2 n' = \frac{5}{6} n' \log_2 n'$, which is exactly what we would like to prove above.

The algorithm B is shown in Fig. 6. It sorts the GPS finish times of these n' packets when it takes as its input a GPS computation tree corresponding to an instance in $I(n')$. In Fig. 6, Q is a FIFO queue of packets and P is an arrary sorted by S_i and min() returns the element with the minimum S_i value (simply the first element of P). If a set of packets S_{j1}, S_{j2}, ... are computed from S_i, then they are all $S_i's$ children.

Two things remain to be proved: (a) the output from the algorithm is indeed sorted, and (b) the complexity of the algorithm is no more than $\frac{4}{6} n' \log_2 n'$.

Proof of (a): Clearly the "while" iterations in Fig. 6 result in a traversal of all the nodes in GPS computation tree since every point is reachable from 0. Therefore, Q will contain a permutation of all the points S_i after the execution. To prove that they are sorted, we only need to show the following invariant $\mathcal{I}$ at each iteration of the "while" loop: the point S_i is deleted from P in the $(i+1)^{th}$ iteration for $i = 0, 1, \cdots, n$. We prove it by induction on i. In the first iteration, clearly number 0 (i.e., S_0) is deleted from P. Suppose $\mathcal{I}$ holds for the number $i < k$. In the $(k+1)^{th}$ iteration, suppose $S_{k'}$ $(k' \neq k)$ will be deleted from P instead of S_k. We know that $k' > k$ due to induction hypothesis. Clearly S_k cannot be in the queue P at the moment since otherwise "min()" will choose S_k instead of $S_{k'}$. Suppose the parent of S_k is $S_{k''}$. Clearly $k'' < k$ and $S_{k''}$ is deleted during the $(k''+1)^{th}$ iteration due to the induction hypothesis. So at the $(k''+1)^{th}$ iteration, S_k must have been inserted. Therefore, the only possibility is that S_k has been deleted earlier than the $(k+1)^{th}$ iteration. This contradicts the induction hypothesis.

Proof of (b): Let E be the set of edges in the GPS computation tree and let e_{ij} be an edge from S_i to S_j $(j > i)$. By the fact (II) above, $\sum_{e_{ij} \in E}(j - i) \leq \frac{1}{6} n' \log_2 n'$. We assume that right after number S_i is deleted (line 5), there are C_i elements in P. We also assume that S_i has D_i children. Then the complexity of sorting these children and merging them with P (already sorted) is $C_i + D_i + D_i \log_2 D_i$, since sorting takes $D_i \log_2 D_i$ time and merging two sorted lists of size C_i and D_i each takes time $C_i + D_i$. The total complexity is therefore $\sum_{i=0}^{n'-1}(C_i + D_i + D_i \log_2 D_i)$. Note that the complexity of the last iteration is zero in terms of comparisons since at that time there is only one element left in the priority queue P. So in the following summations, we will not count the last iteration, and will only add up the index $n' - 1$ (instead of n'). Also note that there is no cost for "min()" operation since the P is already sorted and we only "charge" for additions and comparisons as discussed before.

Clearly, $\sum_{i=0}^{n'-1} D_i = n'$ since each of the n' nodes $0, 1, \cdots, n-1$ appears in D_i exactly once by the aforementioned invariant $\mathcal{I}$. We claim that $(\sum_{i=0}^{n'-1} C_i)$ is no more than $|E| + \sum_{e_{ij} \in E}(j - i)$. This can be shown by the technique of *double counting* in combinatorics as follows. Note that an edge e_{ij} is counted for most $j - i + 1$ times in the above summation since e_{ij} is inserted in round $i + 1$ and deleted in round $j + 1$ (i.e., counted in $C_i, C_{i+1}, ..., C_j$) by the aforementioned invariant $\mathcal{I}$. So $(\sum_{i=0}^{n'-1} C_i) \leq \sum_{e_{ij} \in E}(j - i + 1) = (\sum_{e_{ij} \in E}(j - i)) + |E|$.

So $(\sum_{i=0}^{n'-1} C_i) \leq \frac{1}{6} n' \log_2 n' + n'$ since $|E| = n'$ (property of tree).

Now we claim that $\sum_{i=0}^{n'-1} D_i \log_2 D_i \leq \sum_{i=0}^{n'-1} \frac{1}{2} D_i (D_i + 1) \leq \sum_{e_{ij} \in E} (j - i)$. The first inequality is due to the fact $\log_2 x \leq (x+1)/2$ when $x > 0$. The second equality holds, since these D_i edge from node S_i are at least of lengths 1, 2, $\cdots, D_i$ respectively, and the total length of edges coming out from S_i is at least $\frac{1}{2} D_i (D_i + 1)$. So the total complexity of the algorithm is no more than $(\sum_{i=0}^{n'-1} C_i)$ + $(\sum_{i=0}^{n'-1} D_i)$ + $(\sum_{i=0}^{n'-1} D_i \log_2 D_i) \leq 2(\frac{1}{6} n' \log_2 n' + n') \leq \frac{4}{6} n' \log_2 n'$. ■

IV. Conclusions

In this work, we answered a set of important open questions concerning the computational complexity of packet scheduling. We first studied the interesting open question as to whether the complexity lower bound of "sorting" the GPS virtual finish times, which is all that are needed in WFQ, can be smaller than computing their exact values. We showed, surprisingly, that the former is at least asymptotically as large as the latter. Then we showed that this lower bound complexity for "sorting" the GPS virtual finish times is $\Omega(\log n)$ per packet, and discovered that an existing algorithm [8] matches this bound. This had an important implication: the WFQ implementation that uses the GPS tracking algorithm in [8] is the optimal algorithm for implementing the WFQ service discipline. We discovered, however, that this GPS tracking algorithm requires "mandatory lazy evaluation," so that although its theoretical complexity is $\Omega(\log n)$ per packet, its worst-case complexity in practice can be as large as $\Omega(n)$. Finally, we proved the most important result of this paper: the complexity lower bound for tracking the GPS clock perfectly is $\Omega(\log n)$ per packet.

Acknowledgment

The authors would like to thank Dr. Li (Erran) Li and the anonymous reviewers for valuable comments. This work was made possible by NSF Grant ITR/SY ANI-0113933 and NSF CAREER award ANI-0238315.

References

[1] J. Bennett and H. Zhang. wf^2q: worst-case fair weighted fair queuing. In *IEEE INFOCOM'96*, March 1996.

[2] J. Bennett and H. Zhang. Hierarchical packet fair queuing algorithms. *IEEE/ACM Transactions on Networking*, 5:675–689, 1997.

[3] H. Chao and X. Guo. *Quality of Service Control in High-Speed Networks*. Wiley, 2001.

[4] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *Internetworking: Research and Experience*, pages 3–26, 1990.

[5] D. Dobkin and R. Lipton. A lower bound of $\frac{1}{2}n^2$ on linear search programs for the knapsack problem. *J. Comput. Syst. Sci.*, 16:413–417, 1978.

[6] S. Golestani. A self-clocked fair queueing scheme for broadband applications. In *Proc. of Infocom'94*, June 1994.

[7] P. Goyal, H. Vin, and H. Cheng. Start-time fair queuing: A scheduling algorithm for integrated servicespacket switching networks. In *CS-TR-96-02, Dpet. of Computer Science, Univ. of Texas, Austin*, 1996.

[8] A. Greenberg and N. Madras. How fair is fair queuing? *Journal of the ACM*, 39(3):568–598, 1992.

[9] A. Parekh and R. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single node case. *IEEE/ACM Transaction on Networking*, 1(3):344–357, June 1993.

[10] M. Sipser. *Introduction to the Theory of Computation*. PWS, 1997.

[11] D. Stiliadis and A. Varma. Design and analysis of frame-based fair queuing: A new traffic scheduling algorithm for packet switched networks. In *Proc. of ACM Sigmetrics'96*, pages 104–115, May 1996.

[12] D. Stiliadis and A. Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. In *Proc. of Infocom'96*, March 1996.

[13] J. Turner. New directions in communications (or which way to the information age?). *IEEE Communications Magazine*, 24:8–15, October 1986.

[14] J. Xu and R. Lipton. On fundamental tradeoffs between delay bounds and computational complexity in packet scheduling algorithms. In *Proc. of ACM SIGCOMM 2002*, August 2002.

[15] H. Zhang. Service disciplines for guaranteed performance service in packet switching networks. *Proceedings of the IEEE*, 83(10), October 1995.