QFQ: Efficient Packet Scheduling with Tight Bandwidth Distribution Guarantees

Paper # 244 - Submitted to SIGCOMM 2010

ABSTRACT

Packet scheduling, together with classification, is one of the expensive processing steps in systems providing tight bandwidth and delay guarantees at high packet rates. Schedulers with near-optimal bandwidth distribution and $O(1)$ time complexity in the number of flows have been proposed in the literature, they adopt techniques as *timestamp rounding* and *flow grouping* to keep their execution time small. However even the two best proposals in this family either have a linear complexity component in the number of groups or require a number of operations proportional to the length of the packet during transmission. Furthermore, no studies are available on the actual execution time of these algorithms.

In this paper we make two contributions. First, we present Quick Fair Queueing (QFQ), a new $O(1)$ scheduler that provides near-optimal guarantees, and is the first to achieve that goal with a constant cost also with respect to the number of groups and the packet length. The QFQ algorithm has no loops, and uses very simple instructions and data structures that lend themselves very well to a hardware implementation.

Secondly, we have developed production-quality implementations of QFQ and of its closest competitor, S-KPS, and we have used them to perform an accurate performance analysis, in terms of both run times and service guarantees. Experiments show that QFQ fulfills the expectations, with a runtime on a low end workstation of about 110 ns for an enqueue()/dequeue() pair (only twice the time of DRR, but with much better service guarantees).

1. INTRODUCTION

QoS provisioning is a long-standing problem whose solution is hindered by business and technical issues. The latter relate to the use and scalability of resource reservation protocols and packet schedulers, which are necessary whenever over-provisioning is not an option.

An IntServ approach can provide fine-grained per-flow guarantees, but the scheduler has to deal with a potentially large number of flows *in progress*[1] (up to 10^5 and more, as reported in a recent study [10]). Besides memory costs to keep per-flow state, the time complexity and service guarantees of the scheduling algorithm can be a concern.

A DiffServ approach aggregates flows into a few classes with predefined service levels, and schedules the aggregate classes without need for resource reservation signaling. This drastically reduces the space and time complexity, but, within each class, per-flow scheduling may still be needed if we want to provide fairness and guarantees to the individual flows.

The above considerations motivate the interest for packet schedulers with low complexity and tight guarantees even in presence of large number of flows. Round Robin schedulers have $O(1)$ time complexity, but all have an $O(N)$ worst-case deviation with respect to the ideal amount of service that the flow should receive over any given time interval.

More accurate schedulers have been proposed, based on flow grouping and timestamp rounding, which feature $O(1)$ time complexity and near-optimal deviation from the ideal amount of service (see Section 2). The two most efficient proposals in this class, the scheduler proposed in [14], hereafter called *Group Fair Queueing* (*GFQ*) for brevity, and *Simple KPS* (*S-KPS*) [8] use data structures with somewhat high constants hidden in the $O()$ notation. In particular, on each dequeue operation, GFQ needs to iterate on all groups in which flows are partitioned, whereas S-KPS has to do (in parallel with packet transmissions) a number of operations proportional to the length of the packet being transmitted.

Our contribution: In this paper we present Quick Fair Queueing (QFQ), a new scheduler with $O(1)$ time complexity, implementing an approximated version of WF^{2}Q+ with near-optimal service guarantees similar to GFQ and S-KPS.

The key innovation of QFQ is the partitioning of groups of flows into four sets, each represented by a machine word. All bookkeeping to implement schedul-

[1] In [10] a flow is denoted as in progress during any time interval in which the inter-arrival time of its packets is lower than 20 seconds.

ing decisions is based on manipulations of these sets. Multiple groups and flows can be moved at once between sets with simple CPU instructions such as AND, OR, XOR and *Find First bit Set* (FFS).

The major improvement of QFQ over the previous proposals is on performance: our algorithm has no loops, and the simplicity of the data structures and instructions involved makes it well suited to hardware implementations. The execution time is within two times that of DRR across a wide variety of configurations, and consistently about three times faster than S-KPS. In absolute terms, on a low end desktop machine, with 1000 backlogged flows DRR and QFQ take 50 and 110 ns per enqueue()/dequeue() pair, respectively. In these conditions, S-KPS takes 350-400 ns. As the number of flows and queue size grows, all algorithms exhibit a proportional increase in the running time, mostly due to cache misses, going up to 95 ns for DRR, 150 ns for QFQ, and 500 ns for KPS. Section 7 reports more detailed results.

Paper structure: Section 2 complements this introduction by discussing related work. In Section 3 we define the system model and other terms used in the rest of the paper. Section 4 presents the QFQ algorithm in detail, proving its correctness and illustrating its implementation. Section 5 gives an analytical evaluation of the (worst case) service guarantees. In Section 6 we present the results of some ns2 simulation to compare the delay experienced by various traffic patterns under different scheduling policies. Finally, Section 7 measures the actual performance of the algorithm on a real machine, comparing production quality implementations of QFQ, S-KPS, and several other schedulers (FIFO, DRR and WF2Q+) taken from Linux and FreeBSD distributions.

NOTE FOR THE REVIEWERS: Some of the proofs in Section 4 and Section 5 are omitted for brevity, but are included in an unpublished, full version of this paper [1] which is available on request from the TPC chairs, as well as the code used for the experiments.

2. BACKGROUND AND RELATED WORK

Packet schedulers are evaluated based on their time (and space) complexity and on their service properties. Several service metrics have been defined in the literature, including *relative fairness* [9], and *Bit-* and *Time-Worst-case Fair Index* (B-WFI and T-WFI [3, 5], see the definitions in Section 5; in the paper we will refer to both as WFI for brevity).

B-WFI^k and T-WFI^k represent the worst-case deviation (in terms of service and time, respectively) that a flow k may experience over any time interval with respect to the service it would receive from a perfect weighted bandwidth sharing server.

The WFI is somehow a more accurate service metric than just the worst-case per-flow lag or packet completion delay with respect to an ideal, perfectly fair system (hereafter we call just lag and packet delay these quantities). In fact, even a scheduler guaranteeing the minimum possible lag or packet delay may still suffer from a high WFI. It happens if the scheduler oscillates between the following two situations. First, during some time interval, the scheduler gives a flow more service than what the flow should receive according to its reserved share. Then the scheduler forces the flow to "pay back" for the extra service previously received, i.e., the scheduler gives to the flow less service than the reserved share until the balance breaks even.

The WFI measures the extent of these oscillations, which can even hinder per-flow lag and delay guarantees in a hierarchical setting [3]. In the end, a low WFI is essential to provide tight fairness and delay guarantees in such a setting. In addition, the B-WFI can also be used to predict the minimum amount of service guaranteed to each flow over any time interval, whereas, if the arrival pattern is known, the T-WFI can be used to compute the maximum completion time of each packet.

Round Robin (RR) schedulers lend naturally to $O(1)$ implementations with small constants. Several variants have been proposed, as e.g., *Deficit Round Robin* [13], *Smoothed Round Robin* [4] and its evolution *G-3* [6], *Aliquem* [11] and *Stratified Round Robin* [12], to address some of the shortcomings (burstiness, etc.) of RR schedulers. Despite their extreme simplicity and efficiency, one disadvantage of this family of schedulers is that, irrespective of the weight ϕ^k of any flow k, both the flow packet delay and the T-WFI^k have an $O(NL)$ component, where L is the maximum packet size and N is the total number of flows in the system.

To achieve a lower WFI than what is possible with RR schedulers, other scheduler families try to stay as close as possible to the service provided by an internally-tracked ideal system. We call them *timestamp based* schedulers as they typically timestamp packets with some kind of *Virtual Time* function, and try to serve them in ascending timestamp order, which has $\Omega(\log N)$ cost. Using this approach, schedulers such as WF^{2}Q [5] and WF^{2}Q+ [3] offer *optimal* lag/packet delay and optimal WFI, i.e., the lowest possible lag/packet delay and WFI for a non-preemptive system. Especially, their lag/packet delay is $O(L)$ whereas their WFI is $O(L(1 + 1/\phi^k))$. The L/ϕ^k component is unavoidable and due to the non-preemptiveness of the system. This component can still grow to $O(NL)$ for low-weight flows, but high weight flows receive better treatment.

Exact timestamp based schedulers have $O(\log N)$ time complexity. Several approximated variants have been defined that use rounded timestamps instead of exact ones to get rid of the burden of exact ordering and run in $O(1)$ time. Examples are GFQ [14], Simple KPS (S-KPS, the most efficient variant of SI-WF^{2}Q) [8],

and the QFQ algorithm described here.

The approximation has an implication, proved in [17]: differently from an optimal one (as WF^{2}Q or WF^{2}Q+), any scheduler based on approximated timestamps has a packet delay with respect to an ideal GPS server larger than $O(L)$. However, GFQ, S-KPS and QFQ guarantee the same T-WFI=$O(L(1+1/\phi^k))$ as the optimal schedulers. The difference is only in the multiplying constant, which is 1 with exact timestamps and slightly larger otherwise (e.g., 3 in the case of QFQ—see Sec.5.2). Thus, approximated timestamps still give much better guarantees than RR schedulers.

Approximated timestamp-based schedulers typically use data structures that are more complex to manipulate than those used in RR or exact timestamp-based schedulers. As a consequence, the same or better asymptotic complexity does not necessarily reflect in faster execution times. As an example, the approximated variants of WF^{2}Q+ presented in GFQ [14] use timestamp rounding, flow grouping and a calendar queue to maintain a partial ordering among flows within the same group. Each scheduling decision requires a linear scan of the groups in the system to determine the candidate for the next transmission. The actual algorithms are only outlined, and no public implementation is available; the authors claim a sustainable rate of 1 Mpps and a per-packet overhead of 500 ns for their hardware implementations.

LFVC [16] rounds timestamps to integer multiples of a fixed constant. Reducing the timestamps to a finite universe enables LFVC to use data structures like van Emde Boas priority queues, which have $O(\log\log N)$ complexity for all the basic operations they support. This is not $O(1)$ but grows very slowly with N, though the van Emde Boas priority queues have high constants hidden in the $O()$ notation. A drawback of LFVC is that its worst case complexity is $O(N)$, because the algorithm maintains separate queues for eligible and ineligible flows, and individual events may require to move most/all flows from one queue to the other.

Finally, S-KPS is based on a special data structure called Interleaved Stratified Timer Wheels (ISTW). ISTW containers have several nice properties that allow S-KPS to execute packet enqueue and dequeue operations at a worst-case cost independent of even the number of groups, provided that a number of groups proportional to the maximum packet size (in the worst-case) is moved between two containers during the service of each packet. In a system where the latter task can be performed, and completed, during the transmission of a packet, S-KPS runs at a low $O(1)$ amortized cost per packet transmission time.

3. SYSTEM MODEL AND COMMON DEFINITIONS

In this section we give some definitions commonly used in the scheduling literature, and also present the exact WF^{2}Q+ algorithm, which is used as a reference to describe QFQ. For convenience, all symbols used in the paper are listed in Table 1. Most quantities are a function of time, but we omit the time argument (t) when not ambiguous and clear from the context.

Symbol	Meaning
N	Total number of flows
L	Maximum length of any packet in the system
$B(t)$	The set of backlogged flows at time t
$W(t_1,t_2)$	Total service delivered by the system in $[t_1,t_2]$
k	Flow index
L^k	Maximum length of packets in flow k
ϕ^k	Weight of flow k
l^k	Length of the head packet in flow k; $l^k=0$ when the flow is idle
$Q^k(t)$	Backlog of flow k at time t
$W^k(t_1,t_2)$	Service received by flow k in $[t_1,t_2]$
$V(t)$	System virtual time, see Eq. (3)
S^k, F^k	Virtual start and finish times of flow k, see Eq. (2)
$\hat{S}^k, \hat{F}^k$	Approximated S^k and F^k for flow k, see Section 4.1.2
i,j	Group index (groups are defined in Sec.4.1.1)
S_i, F_i	Virtual start and finish times of group i, see Eq. (6)
σ_i	Slot size of group i (defined in Sec.4.1.1, $\sigma_i=2^i$)
ER, EB, IR, IB	The four sets in which groups are partitioned

Table 1: Definition of the symbols used in this paper.

We consider a system in which N packet flows (defined in whatever meaningful way) share a common transmission link serving one packet at a time. The link has a time-varying rate, which the system can decide to use, partially or completely, to transmit packets waiting for service. A system is called *work conserving* if the link is used at full capacity whenever there are packets queued. A scheduler sits between the flows and the link: arriving packets are immediately enqueued, and the next packet to serve is chosen and dequeued by the scheduler when the link is ready. The interface of the scheduler to the rest of the system is made of one packet *enqueue()* and one packet *dequeue()* function.

In our model, each flow k is assigned a fixed weight $\phi^k > 0$. Without losing generality, we assume that $\sum_{k=1}^{N}\phi^k \le 1$.

A flow is defined *backlogged* if it owns packets not yet completely transmitted, otherwise we say that the flow is *idle*. We denote as $B(t)$ the set of flows backlogged at time t. Inside the system each flow has a FIFO queue associated with it, holding the flow's own backlog.

We call *head packet* of a flow the packet at the head of the queue, and l^k its length; $l^k = 0$ when a flow is idle. We say that a flow is *receiving service* if one of its packets is being transmitted. Both the amount of service $W^k(t_1, t_2)$ received by a flow and the total amount of service $W(t_1, t_2)$ delivered by the system in the time interval $[t_1, t_2]$ are measured in number of bits transmitted during the interval.

3.1 WF^{2}Q+

Here we outline the original WF^{2}Q+ algorithm for a variable-rate system. See [3, 15] for a complete description. WF^{2}Q+ is a *packet scheduler* that approximates, on a packet-by-packet basis, the service provided by a work-conserving *ideal fluid system* which delivers the following, almost perfect bandwidth distribution over any time interval:

$$W^k(t_1, t_2) \geq \phi^k W(t_1, t_2) - (1 - \phi^k)L. \qquad (1)$$

The packet and the fluid system serve the same flows and deliver the same *total* amount of work $W(t)$ (systems with these features are called *corresponding* in the literature). They differ in that the fluid system may serve multiple packets in parallel, whereas the packet system has to serve one packet at a time, and is non preemptive. Because of these constraints, the allocation of work to the individual flows may differ in the two systems. WF^{2}Q+ has optimal B-/T-WFI and $O(\log N)$ complexity, which makes it of practical interest.

WF^{2}Q+ operates as follows. Each time the link is ready, the scheduler starts to serve, among the packets that have already started in the ideal fluid system, the next one that would be completed (ties are arbitrarily broken). WF^{2}Q+ is a work-conserving on-line algorithm, hence it succeeds in finishing packets in the same order as the ideal fluid system, except when the next packet to serve arrives after that one or more out-of-order packets have already started.

The WF^{2}Q+ policy is efficiently implemented by considering, for each flow, a special *flow virtual time* function $V^k(t)$ that grows as the *normalized amount of service* (i.e. actual service divided by the flow's weight) received by the flow when it is backlogged. The algorithm only needs to know the values of $V^k(t)$ when the flow becomes backlogged, or when its head packet completes transmission in the ideal fluid system. So, each flow k is timestamped with these two values, called *virtual start* and *finish time*, S^k and F^k, of the flow. Using an additional *system virtual time* function $V(t)$, at time t_p when a packet enqueue/dequeue occurs, WF^{2}Q+computes these timestamps as follows:

$$\begin{aligned} S^k &\leftarrow \begin{cases} \max(V(t_p),\, F^k) & \text{on newly backlogged flow} \\ F^k & \text{on pkt dequeue} \end{cases} \\ F^k &\leftarrow S^k + l^k/\phi^k, \end{aligned} \qquad (2)$$

where $V(t)$ is the *system virtual time* function defined as follows (assuming $\sum \phi^k \leq 1$):

$$V(t_2) \equiv \max\left(V(t_1) + W(t_1, t_2),\, \min_{k \in B(t_2)} S^k\right). \qquad (3)$$

At system start-up $V(0) = 0$, $S^k \leftarrow 0$ and $F^k \leftarrow 0$.

Flow k is said to be *eligible* at time t if $V(t) \geq S^k$. This inequality guarantees that the head packet of the flow has already started to be served in the ideal fluid system. Using this definition, WF^{2}Q+ can be implemented as follows: each time the link is ready, the scheduler selects for transmission the head packet of the eligible flow with the smallest virtual finish time. Note that the second argument of the max function in Eq. (3) guarantees that the system is work-conserving.

The implementation complexity in WF^{2}Q+ comes from three tasks: i) the computation of $V(t)$ from Eq. (3), which requires to keep track of the minimum S^k, and has $O(\log N)$ cost; ii) the selection of the next flow to serve among the eligible ones, which requires sorting on F^k, and also has $O(\log N)$ cost at each step; iii) the management of eligible flows as $V(t)$ grows. This is made complex by the fact that any change in $V(t)$ can render $O(N)$ flows eligible. With some cleverness [7], an augmented balanced tree can be used to perform the latter two tasks together in $O(\log N)$ time.

4. QUICK FAIR QUEUEING

We start by giving the intuition of the algorithm and of its underlying data structures, represented in Fig. 1. As other timestamp-based schedulers, QFQ associates to each flow (a yellow square in the figure) both exact and approximated virtual times. Flows are statically mapped into a finite number of *groups* (the grey regions in the figure). Within each group, a bucket sort algorithm is used to keep flows ordered according to their (approximate) virtual times. Structural properties of the algorithm guarantee that each group only needs a finite and small number of buckets, so both sorting and min-extraction within a group can be done in constant time with a handful of machine instructions. Depending on certain properties of its virtual times, each non-empty group belongs to one of four sets called ER, EB, IR and IB.

The management of eligibility (essential for service guarantees) and inter-group sorting is done by splitting groups between four distinct sets, called ER, EB, IR and IB. These are built so that i) groups in ER only contain eligible flows, ii) in ER, the group number reflects the ordering of the (approximated) virtual times

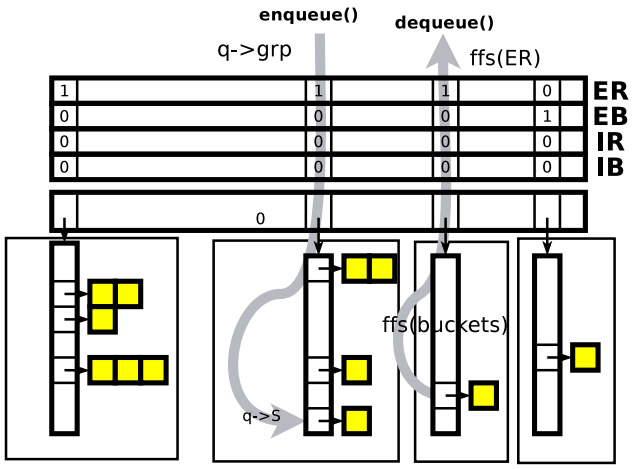

Figure 1: QFQ at a glance. The figure represents all main data structures used by the algorithm: the four groups sets on the top; the groups (rectangles on the bottom) containing the bucket lists and individual flow queues.

of the flows in the group, and iii) within each group the bucket number reflects the order of virtual times of the flows in the group.

As a result, an *enqueue()* operation can directly access the group, use the virtual time as a bucket index, and do a simple append to add the flow to the bucket's queue. Some trivial bitmap operations are then sufficient to adjust set membership.

On a *dequeue()*, the selection of the eligible flow with the smallest virtual finish time requires only a couple of Find First Set (ffs) CPU instructions to locate the candidate, once again followed by a modest amount of work to readjust set membership.

4.1 Detailed Description

QFQ approximates WF^{2}Q+, providing near-optimal service guarantees (see Section 5) but reducing the implementation complexity to $O(1)$ with small constants thanks to three main techniques described below.

4.1.1 Flow Grouping

QFQ groups flows into a small constant number of groups on which to do the scheduling. A flow k is assigned to a group i defined as

$$i = \left\lceil \log_2 \frac{L^k}{\phi^k} \right\rceil, \qquad (4)$$

where L^k is the maximum size of the packets for flow k. For any practical set of L^k's and ϕ^k's in a system, *the number of distinct groups is less than 64* (in fact, 32 groups suffice in many cases)[2] and lets us represent a set of groups with a bitmap that fits in a single machine word.

The quantity $\sigma_i \equiv 2^i$ (bits) is called the *slot size* of the group. It is easy to see that $L^k/\phi^k < \sigma_i$, hence from Eq. (2), $F^k - S^k \le \sigma_i$ for any flow k in group i.

4.1.2 Timestamp Rounding

QFQ computes S^k and F^k for each flow using the exact algorithm in Eq. (2), but for eligibility or scheduling computations it uses the approximate values[3]:

$$\hat{S}^k \leftarrow \left\lfloor \frac{S^k}{\sigma_i} \right\rfloor \sigma_i \quad , \quad \hat{F}^k \leftarrow \hat{S}^k + 2\sigma_i, \qquad (5)$$

where i is the group index.

Within each group we define

$$S_i = \min_{k \in group_i} \hat{S}^k \quad , \quad F_i = S_i + 2\sigma_i, \qquad (6)$$

which are called the group's *virtual start* and finish times. Finally, in Eq. (3), QFQ replaces $\min_{k \in B(t)} S^k$ with $\min_{k \in B(t)} \hat{S}^k$.

Same as for the exact values, $\hat{S}^k$ and $\hat{F}^k$ can assume a limited range of values around $V(t)$ (this is called Globally Bounded Timestamp property, or GBT). We can prove Theorems 3 and 4 that at any time instant $\hat{S}^k < V(t) + \sigma_i$, and the range of values for $\hat{S}^k$ is limited to $2 + \lceil L/\sigma_i \rceil$ times σ_i. This implies that at any time instant the possible values for $\hat{S}^k$ are in small and finite number. Hence, we can sort flows within a group using a constant-time bucket sort algorithm. The use of $\hat{S}^k$ in Eq. (3) also saves another sorting step, because, as we will see, the *group sets* defined in the next section let us compute it in constant time.

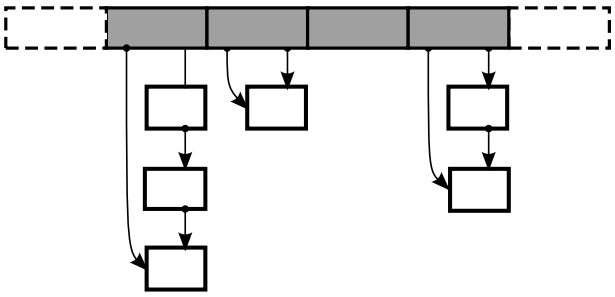

Figure 2: A representation of bucket lists. The number of buckets (grey, each corresponding to a possible value of $\hat{S}^k$), is fixed and independent of the number of flows in the group.

[2]This is trivially proven by substituting values in (4); as an example, L^k between 64 bytes and 16 Kbytes, ϕ^k between 1 and 10^{-6}) yield values between $64 = 2^6$ and $16 \cdot 10^9 \approx 2^{34}$, or 29 groups.

[3]There is only one case where the S^k for certain newly backlogged groups can be shifted backwards to preserve the ordering of **EB**; this exception is described and its correctness is proved in Lemma 4.

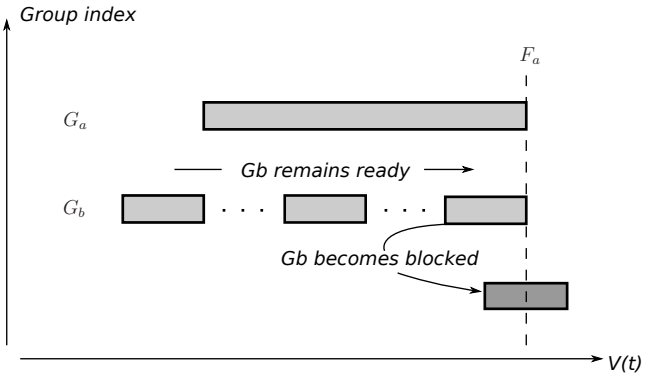

Figure 3: Transitions from/to the blocked state. See Section 4.1.3.

The data structure used to sort flows within a group is a *bucket list*—a short array with as many buckets as the number of distinct values for $\hat{S}^k$ (see Fig. 2). Each bucket contains a FIFO list of all the flows with the same $\hat{S}^k$ and $\hat{F}^k$. The number of buckets depends on the ratio L/σ_i which is at most $\max(L/L_k)$. For practical purposes, 64 buckets are largely sufficient, so we can map each bucket to a bit in a machine word; this allows us to use a constant-time Find First bit Set (FFS) instruction to locate the first non-empty bucket in order to find the next flow to serve.

4.1.3 Group Sets and Their Properties

QFQ partitions backlogged groups into four distinct sets, which reduces scheduling and bookkeeping operations to simple set manipulations, in turn performed with basic CPU instructions such as AND, OR and FFS on single machine words.

The sets are called $\mathbf{ER}, \mathbf{EB}, \mathbf{IR}, and\ \mathbf{IB}$ (from the combinations of the initials of *Eligible, Ineligible, Ready,* and *Blocked*), and the partitioning is done using two properties:

- **Eligibile:** group i is said *Eligible* at time t iff $S_i \le V(t)$, and *Ineligible* otherwise.
- **Blocked:** independent of its own eligibility, a group i is said to be *Blocked* if there is some eligible group with higher index and lower finish time. Otherwise the group is said to be *Ready*.

Fig. 3 illustrates the meaning of "Blocked". The rectangle on the top represents the first full slot of group G_a the smaller rectangles on the bottom represent some of the positions that a group G_b with an index $b < a$ can assume. If only these two groups are backlogged in the system, G_b remains **Ready** until its finish time F_a is smaller than that of G_a. Flows in G_b will be selected for service (as long as they are eligibile), and the finish time of the group increases as its flows gets serviced. Once $F_b > F_a$ then G_b becomes *Blocked*.

Introducing the blocked state gives groups in the **ER** the nice property that the group index reflects the ordering by finish time: $(\forall j, i \in \mathbf{ER}, j > i \implies F_j > F_i)$ which means that the next packet to serve is the head packet of the first flow of the first group in **ER**.

Individually, a group can enter any of the four sets when it becomes backlogged or after it is served. Moves of multiple groups from one set to the other can occur only on the paths

$$\mathbf{IB} \to \mathbf{IR}\ ,\ \mathbf{IR} \to \mathbf{ER}\ ,\ \mathbf{IB} \to \mathbf{EB}\ ,\ \mathbf{EB} \to \mathbf{ER}$$

because transitions in eligibility (driven by changes in $V(t)$), and readiness (driven by changes in the F_i of the blocking group) are not reversible until the group is served.

Moving multiple groups from one set to another can be done with basic CPU instructions (AND, OR, NOT) without iterating over the sets, because, first, the finish time ordering of **ER** actually holds for all the four sets, and second $\mathbf{IB} \cup \mathbf{IR}$ is sorted by both S_i and F_i. All the properties and their proofs are summarized below:

1. $\mathbf{IB} \cup \mathbf{IR}$ is sorted by S_i as a result of the GBT property. In fact, if a group i is ineligible, any flow k in the group has $V(t) < S^k < V(t) + \sigma_i$. Due to the rounding we can only have $S_i = \lceil V(t)/\sigma_i \rceil \sigma_i$, and if $i < j$, we have $2\sigma_i \le \sigma_j$ hence $S_i \le S_j$;
2. $\mathbf{IB} \cup \mathbf{IR}$ is also sorted by F_i because of the sorting by S_i and the fact that σ_i's are increasing with i;
3. the sorting of **ER** by F_i is proven in Theorem 6;
4. the sorting of **EB** by F_i is proven in Theorem 7;

By definition, if $i \in \mathbf{EB}$, then at least one group $j \in \mathbf{ER}$ has $F_j \le F_i$. Because **ER** is sorted by F_i, the readiness test for group i needs only to look at the lowest-order group in **ER** with an index $x > i$. Function *compute_group_state()* in Fig. 4 does the computation of set membership for a group.

4.2 Quick Fair Queueing: The Algorithm

We are now ready to describe the full algorithm for *enqueue()* and *dequeue()*, which are run respectively on packet arrivals, and when the link becomes idle.

4.2.1 Packet Enqueue

The full enqueue algorithm is shown in Fig. 4. First, the packet is appended to the flow's queue, and nothing else needs to be done if the flow is already backlogged. Next, we update the flow's timestamps (lines 8–9). In line 11 we check whether the group's state needs to be updated: this happens if the group was not backlogged (g.bucketlist.head == NULL), or if the new flow causes the group's timestamp to decrease. In this respect, from line 8 it follows that $V(t) \le S^k$. As a consequence, if $S^k < S_i$ holds at line 11, then $V(t) < S_i$ holds too.

```
1  // Enqueue the input pkt of the input flow
2  enqueue(in: pkt, in: flow)
3  {
4    append(pkt, flow.queue); // always enqueue
5    if (flow.queue.head != pkt)
6         return; // Flow already backlogged, we are done.
7    // Update flow timestamps according to Eq. (2)
8    flow.S = max(flow.F, V) ;
9    flow.F = flow.S + pkt.length/flow.weight;
10   g = flow.group; // g is our group
11   if (g.bucketlist.head==NULL || flow.S<g.S) {
12        // Group g is surely idle or not eligible.
13        // Remove from IR and IB if there, and compute
14        // new timestamps from Eq. (5).
15        set[IR] &= ~(1 << g.index) ;
16        set[IB] &= ~(1 << g.index) ;
17        g.S = flow.S & ~(g.slot_size - 1);
18        g.F = g.S + 2*g.slot_size;
19   }
20   bucket_insert(flow, g);
21
22   // If there is some backlogged group, at least one is
23   // in ER; otherwise, make sure V ≥ g.S
24   if (set[ER] == 0 && V < g.S)
25        V = g.S ;
26
27   // compute new state for g, and insert in the proper set
28   state = compute_group_state(g) ;
29   set[state] |= 1 << g.index ;
30 }
31
32 // Compute the group's state, see Section 4.1.3
33 compute_group_state(in: g) : int
34 {
35   // First, eligibility test
36   s = (g.S <= V) ? ELIGIBLE : INELIGIBLE;
37   // Find lowest order group x > group.index.
38   // This is the group that might block us.
39   // ffs_from(d, i) returns the index of the first
40   // bit set in d after position i
41   x = ffs_from(set[ER], g.index);
42   s += (x != NO_GROUP && groups[x].F < g.F) ?
43         BLOCKED : READY;
44   return s;
45 }
```

Figure 4: The *enqueue()* function, called on packet arrivals, and *compute_group_state()* that implements the tests for eligibility and readiness.

Hence the group was either idle or ineligible if we enter the block at lines 12-19, and we remove the group from the ineligible sets (lines 15–16), and update the group's timestamps (being the slot size 2^i, the start time calculation only needs to clear the last i bits of S_k, line 17). Following that, we use a constant time bucket sort (line 20) to order the flow with respect to other flows in the group. At this point we may need to update $V(t)$ as in Eq. (2). Finally (lines 28–29), we update the state of the group (which may have changed due to the new values of S_i, F_i and $V(t)$), and put the group in its new set. The computation of the new state of the group is done in function *compute_group_state()*.

It is important to note that an enqueue involves no movement of other groups across sets. There cannot be eligibility changes, because $V(t)$ changes only if all other groups are idle. The blocked/ready states cannot change if the flow was already backlogged, because its group does not change its finish time.

Finally, if the group j containing this flow just became backlogged, or its finish time decreased, we have $S^k \geq V(t)$, hence $F_j > \lfloor V(t)/\sigma_j \rfloor \sigma_j + 2\sigma_j$. Any Ready group $i < j$ will have $F_i < \lfloor V(t)/\sigma_i \rfloor \sigma_i + 3\sigma_i$ (one σ_i comes from the upper bound on S^k, the other two come from the definition of $F_i = S_i + 2\sigma_i$). Hence $F_i \leq V(t) + 3\sigma_i$. By definition $j > i \implies \sigma_j \geq 2\sigma_i$, so $F_j \geq F_i$ and the newly backlogged group j cannot block a previously Ready group, even in the worst case (largest possible F_i, smallest possible F_j).

4.2.2 Packet Dequeue

```
1  dequeue() : packet // Return the next packet to serve
2  {
3    if (set[ER] == 0)
4        return NULL;
5    // Dequeue the first packet of the first flow of the group
6    // in ER with the smallest index
7    g = groups[ffs(set[ER])];
8    flow = bucket_head_remove(g.bucketlist);
9    pkt = head_remove(flow.queue);
10
11   // Update flow timestamps according to Eq. (2)
12   flow.S = flow.F ;
13   p = flow.queue.head; // next packet in the queue
14   if (p != NULL) {
15        flow.F = flow.S + p.length/flow.weight;
16        bucket_insert(flow, g) ;
17   }
18
19   old_V = V; // Need the old value in make_eligible()
20   V += pkt.length; // Account for packet just served
21
22   old_F = g.F ; // Save for later use
23   if (g.bucketlist.headflow == NULL) {
24        state = IDLE ; // F not significant now
25   } else {
26        g.S = g.bucketlist.headflow.S ;
27        g.F = g.bucketlist.headflow.F ;
28        state = compute_group_state(g) ;
29   }
30
31   // If g becomes IDLE, or if F has grown, may need to
32   // unblock other groups and move g to its new set
33   if (state == IDLE || g.F > old_F) {
34        set[ER]    &= ~(1 << g.index);
35        set[state] |=    1 << g.index ;
36        unblock_groups(g.index, old_F);
37   }
38
39   x = set[IR] | set[IB]; // all ineligible groups
40   if (x != 0) { // Someone is ineligible, may need to
41                 // bump V up according to Eq. (3)
42        if (set[ER] == 0)
43             V = max(V, groups[ffs(x)].S) ;
44        // Move newly eligible groups from IR/IB to ER/EB
45        make_eligible(old_V, V) ;
46   }
47   return pkt ;
48 }
```

Figure 5: The *dequeue()* function, described in Section 4.2.2.

Function *dequeue()* in Fig. 5 is called whenever the link is idle and we have packets to transmit. The function returns the next packet to transmit, and updates data structures as needed.

The packet selection (lines 3–9) is straightforward. If

there are queued flows, at least one flow is eligible, so **ER** is not empty, and we just need to pick the group with the lowest index in **ER**, and from that the first packet from the first flow. Following that, the flow's timestamps are updated, and the flow is possibly reinserted in the bucket list (lines 11–17).

Virtual time is increased in line 20, to reflect the service of the packet selected for transmission. Next (lines 22–29), the group's timestamps and state are updated. If the group has increased its finish time or it has become idle (lines 33–37), we may need to unblock some other groups, using function *unblock_groups()* described in Sec. 4.2.3.

Finally, lines 39–46 make sure that at least one backlogged group is eligible by bumping up V if necessary, and moving groups between sets using function *make_eligible()* which will be discussed next.

4.2.3 Support Functions

We are left with a small set of functions to document, shown in Fig. 6, and mostly used in the *dequeue()* code.

Function *move_groups()* is trivial and just moves from set *src* to set *dest* those groups whose indexes are included in *mask*. Simple bit operations do the job.

Function *make_eligible()* determines which groups become eligible as $V(t)$ grows after serving a flow. Here again we exploit the features of timestamp rounding to make the operation simple. Fig. 7 gives a graphical representation of the possible values of S_i's and $V(t)$, and the binary representations of $V(t)$ (the vertical strings of binary digits). Since slot sizes are powers of two, the binary representation of the i-th group's slot size (and thus of all its virtual time slots, as they are aligned to the slot size) ends with i zeros; in any given slot belonging to group i, the value of the i-th bit is constant during the whole slot. Each time the i-th bit of $V(t)$ flips, the virtual time enters a new slot of size 2^i. Since slots are aligned, we only need to determine the highest bit j that changed on each $V(t)$ update to find all the ineligible groups with $i \leq j$ have become eligible.

Function *make_eligible()* computes the index j, using an XOR followed by a *Find Last Set* (fls) operation; then computes the binary mask of all indexes $i \leq j$, and calls function *move_groups* to move groups whose index is in the mask from **IR** to **ER** and from **IB** to **EB**.

Function *unblock_groups()* updates the set of blocked groups. If the finish time F_i of the group under service increases it may be possible that some groups it was blocking become ready (in other words, moving the blocked groups into the ready state would no more violate the ordering of **ER**$\cup$**IR**, because the violation was caused by the finish times of the blocked groups being lesser than F_i). Anyway, if there still are groups in **ER**, the first one of them must have a finish time greater than the last one just served, otherwise the blocked groups would remain blocked by the new head group. If there are no more groups in **ER**, or if the smallest F_i in **ER** is greater than the one last served, then Theorem 8 proves that all the groups with an index smaller than the groups last served become ready again.

```
1  // Move the groups in mask from the src to the dst set
2  move_groups(in: mask, in: src, in: dest)
3  {
4    set[dest] |= (set[src] & mask) ;
5    set[src]  &= ~(set[src] & mask) ;
6  }
7
8  // Move from IR/IB to ER/EB all groups that become
9  // eligible as V(t) grows from V1 to V2.
10 // This uses the logic described in Fig. 7
11 make_eligible(in: V1, in: V2)
12 {
13   // compute the highest bit changed in V(t) using XOR
14   i = fls(V1 ^ V2);
15   // mask contains all groups with index j ≤ i
16   mask = (1 << (i+1)) - 1 ;
17   move_groups(mask, IR, ER) ;
18   move_groups(mask, IB, EB) ;
19 }
20
21 // Unblock groups after serving group i with F=old_F
22 unblock_groups(in: i, in: old_F)
23 {
24   x = ffs(set[ER]) ;
25   if (x == NO_GROUP || groups[x].F > old_F) {
26       // Unblock all the lower order groups (Theorem 8)
27       // mask contains all groups with index j < i
28       mask = (1 << i) - 1 ;
29       move_groups(mask, EB, ER) ;
30       move_groups(mask, IB, IR) ;
31   }
32 }
```

Figure 6: Support functions to recompute the set of eligible groups after a flow has been served. These are described in Sec. 4.2.2

.

4.3 Time and Space Complexity

From the listings it is clear that QFQ has $O(1)$ time complexity on packet arrivals and departures: all operations, including insertion in the bucket list and finding the minimum timestamps, require constant time. All arithmetic operations can be done using fixed point computations, including the division by the flow weight. Detailed performance measurements will be given in Section 7.

In terms of space, the per-flow overhead is approximately 24 bytes (two timestamps, weight, group index and one pointer). Each group contains a variable number of buckets (32 in the worst case, requiring one pointer each), plus two timestamps and a bitmap. Finally, the main data structure contains five bitmaps, the sum of weights and a timestamp. Overall, even a large configuration will require 4 Kbytes of memory to hold the entire state of the scheduler.

An important property is that on each *enqueue()* or

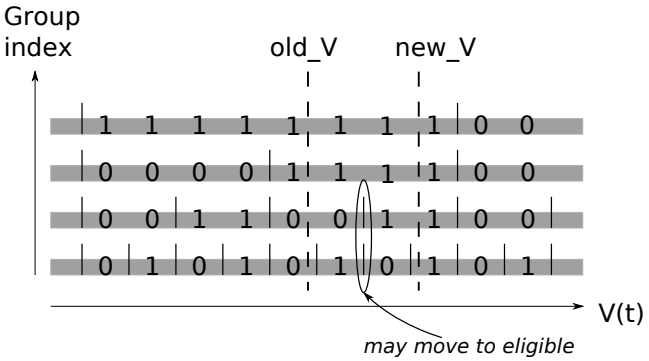

Figure 7: Tracking group eligibility. The picture represents the transition of V from old_V to new_V; the highest bit that flips across the transition is the second one, so groups zero and one are the candidates to become eligible. See Section 4.2.3 for further details.

dequeue() request, the algorithm only touches the internal memory (the 4 KB mentioned above) and the descriptor of the single flow involved in the operation. As a consequence, QFQ it is influenced very little by contention on the external memory bus.

5. SERVICE PROPERTIES

Service guarantees are an important parameter of any scheduling algorithm, so we provide here analytical bounds for the B-WFI (bit guarantees) and the T-WFI (time guarantees) of QFQ. Most of the proofs for the theorems used here are reported in the full version of the paper [1], (available from the TPC chairs to preserve anonimity.)

5.1 Bit Guarantees

The B-WFI^k guaranteed by a scheduler to a flow k is defined as:

$$\text{B-WFI}^k \equiv \max_{[t_1,t_2]} \phi^k W(t_1, t_2) - W^k(t_1, t_2), \qquad (7)$$

where $[t_1, t_2]$ is any time interval during which the flow is continuously backlogged, $\phi^k W(t_1, t_2)$ is the minimum amount of service the flow should have received according to its share of the link bandwidth and $W^k(t_1, t_2)$ is the actual amount of service provided by the scheduler to the flow. This definition is indeed slightly more general than the original one, where t_2 is constrained to the completion time of a packet.

THEOREM 1. **B-WFI for QFQ** *For a flow k belonging to group i QFQ guarantees*

$$B\text{-}WFI^k = 3\phi^k\sigma_i + 2\phi^k L. \qquad (8)$$

PROOF. In this proof we express timestamps ($V(t)$, $F^k(t)$, etc.) as functions of time to avoid ambiguities. We consider two cases. The first one is when flow k is eligible at time t_1. In this case, the virtual time $V^k(t)$ of flow k in the real system must be $V^k(t_1) \le F^k(t_1)$, and $V^k(t_2) \ge S^k(t_2)$. In addition, $\forall t, V(t) \le F_i(t) + L$ as proven in Theorem 4, then $S_i(t_2) = F_i(t_2) - 2\sigma_i > V(t_2) - L - 2\sigma_i$. Hence we have:

$$\begin{aligned}
W^k(t_1,t_2) &= \\
\phi^k V^k(t_1,t_2) &= \\
\phi^k(S^k(t_2) - V^k(t_1)) &\ge \\
\phi^k(S^k(t_2) - F^k(t_1)) &\ge \\
\phi^k(S_i(t_2) - F^k(t_1)) &> \\
\phi^k(S_i(t_2) - (S^k(t_1) + \sigma_i)) &> \\
\phi^k(V(t_2) - L - 2\sigma_i - (V(t_1) + \sigma_i)) &= \\
\phi^k(V(t_2) - V(t_1) - L - 3\sigma_i) &= \\
\phi^k(V(t_2) - V(t_1)) - \phi^k L - 3\phi^k\sigma_i &> \\
\phi^k W(t_1,t_2) - 2\phi^k L - 3\phi^k\sigma_i, &
\end{aligned} \qquad (9)$$

where the last inequality follows from the fact that, because of the immediate increment of $V(t)$ as a packet is dequeued (see *updateV()*), $V(t_2) - V(t_1) \ge W(t_1, t_2) - L$.

The other case is when flow k is not eligible at time t_1. This implies that $V^k(t_1)$ is exactly equal to $S^k(t_1)$. Hence, considering that $S^k(t_1) \le V(t_1) + \sigma_i$, we have:

$$\begin{aligned}
W^k(t_1,t_2) &\ge \\
\phi^k(S^k(t_2) - S^k(t_1)) &\ge \\
\phi^k(S_i(t_2) - S^k(t_1)) &> \\
\phi^k(V(t_2) - L - 2\sigma_i - S^k(t_1)) &> \\
\phi^k(V(t_2) - L - 2\sigma_i - (V(t_1) + \sigma_i) &> \\
\phi^k(V(t_2) - V(t_1) - L - 3\sigma_i) &> \\
\phi^k W(t_1,t_2) - 2\phi^k L - 3\phi^k\sigma_i. &
\end{aligned} \qquad (10)$$

□

As a term of comparison, in a perfectly fair ideal fluid system such as the GPS server, B-WFI$^k = 0$ (see [3]), whereas repeating the same passages of the proof in case of exact timestamps (i.e. exact WF^{2}Q+ *with stepwise* $V(t)$), the resulting B-WFI^k would be $(L^k + 2\phi^k)L$.

The B-WFIs for SI-WFQ and GFQ have not been computed by their authors. However both these algorithms and QFQ implement the same policy (WF^{2}Q+), differing only in how they approximate the timestamps. Generalizing the previous proof, and expressing again the timestamps as a function of time, it is possible to show that the B-WFI of a scheduler of this kind is the sum of two components, equal to $\max_t[S^k(t) - V(t)]$ and $\max_t[V(t) - F^k(t)]$, respectively. The first component is upper-bounded by $\max_t[S^k(t) - \hat{S}^k(t)]$ (from Eq. 5); and, Lemma 3 in [8] proves that the second component is upper-bounded by $L + \max_t(\hat{F}^k - F^k(t))$.

In QFQ, the two bounds are σ_i and $L + 2\sigma_i$, respectively. In S-KPS, these bounds are $4\sigma_i$ and $L + 2\sigma_i$, so the B-WFI of S-KPS larger than that of QFQ by $3\sigma_i$. Finally, in GFQ, setting the slot sizes as powers of two yields the bounds σ_i and $L + \sigma_i$, and hence a B-WFI lower than that of QFQ by σ_i.

5.2 Time Guarantees

Expressing the service guarantees in terms of time is only possible if the link rate is known. The T-WFI^k guaranteed by a scheduler to a flow k on a link with constant rate R is defined as:

$$\text{T-WFI}^k \equiv \max\left(t_c - t_a - \frac{Q^k(t_a^+)}{\phi^k R}\right), \qquad (11)$$

In S-KPS, these bounds are $4\sigma_i$ and $L + 2\sigma_i$, so the B-WFI of S-KPS larger than that of QFQ by $3\sigma_i$. Finally, in GFQ, setting the slot sizes as powers of two yields the bounds σ_i and $L + \sigma_i$, and hence a B-WFI lower than that of QFQ by σ_i. where t_a and t_c are, respectively, the arrival and completion time of a packet, and $Q^k(t_a^+)$ is the backlog of flow k just after the arrival of the packet.

THEOREM 2. **T-WFI for QFQ** *For a flow k belonging to group i QFQ guarantees*

$$T\text{-}WFI^k = (3\sigma_i + 2L)\,\frac{1}{R}\ . \qquad (12)$$

For the proof see Theorem 10; the proof is conceptually similar to the one of the B-WFI.

For comparison, a perfectly fair ideal fluid system would have T-WFI$^k = 0$, whereas for WF^{2}Q+, which uses exact timestamps, repeating the same passages of the proof yields T-WFI$^k = (\frac{L^k}{\phi^k} + 2L)/R$. Finally, using the same arguments as for the B-WFI, the T-WFI of S-KPS is higher than that of QFQ by $3\sigma_i/R$, and the T-WFI of GFQ is lower than that of QFQ by σ_i/R.

6. SIMULATIONS

To prove the effectiveness of the service properties guaranteed by QFQ we implemented it in the ns2 simulator [2] and we compared it to DRR, S-KPS and WF^{2}Q+. We chose DRR to represent the class of pure round robin schedulers, KPS as an example of high-efficiency timestamp-based scheduler, and WF^{2}Q+ as a reference point for its optimal service properties.

The network topology used in the simulations is inspired to the one used in [6], and is depicted in Fig. 8. The links between R_0 and R_1 and R_1 and R_2 have 10 Mbit/s bandwidth and 10 ms propagation delay, all the other links have 100 MBit/s and 1 ms. The observed flows are f_0, a CBR with rate 32 Kbit/s going from S_0 to K_0, and f_1, a CBR with rate 512 Kbit/s going from S_1 to K_1. In the network are also active a CBR flow with rate 512 Kbit/s from S_1 to K_1 (same configuration as f_1, and 50 CBR flows with rate 160 Kbit/s going from S_2 to K_2, and two best effort flows, one from S_3 to K_3 and one from S_4 to K_4, each generated from its own Pareto source with mean on and off times of 100 ms, $\alpha = 1.5$, and mean rate of 2 MBit/s (larger than the unallocated bandwidth of the links between the routers, in order to saturate their queues).

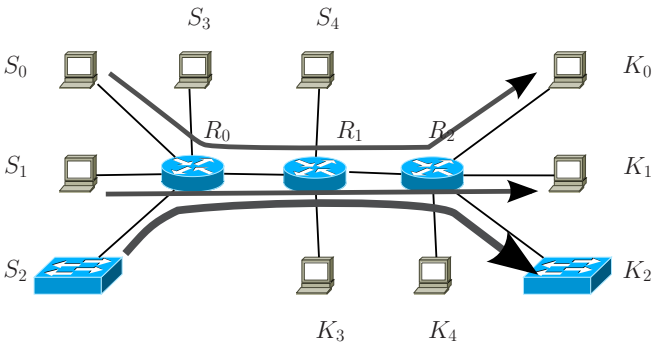

Figure 8: Simulated scenario. Individual flows f_0 and f_1 are originated by nodes S_0 and S_1, whereas S_2 generates 50 CBR flows to perturbate the traffic. The routers all run the same scheduling algorithm.

	f_0	f_1
DRR	134.87 ± 34.28	126.32 ± 30.64
QFQ	43.16 ± 5.60	22.76 ± 0.61
S-KPS	46.28 ± 5.42	22.59 ± 0.60
WF2QP	34.39 ± 0.35	22.59 ± 0.61

Table 2: Simulation results. End-to-end average delays/stddev in ms.

Table 2 shows the average end-to-end delays experienced by f_0 and f_1 during the last 15 seconds of simulation (the total simulation time was 20 s, the first five where not considered to let the values settle). Table 3 shows the maximum delays observed during the experiment. As it can be seen, QFQ performs as expected, with delays similar to the ones measured for S-KPS, given the common nature that the two schedulers share. DRR is showing delays which are an order of magnitude higher than the other schedulers.

We can also observe that, as expected, both S-KPS and QFQ are able to provide less deviation from WF^{2}Q+ for flows with higher rate, because the timestamp approximation is less precise as the flow's rate decreases. Both the average and the maximum delays for f_0, which is a low-rate flow (the lowest rate in the sys-

	f_0	f_1
DRR	216.99	206.40
QFQ	47.56	23.33
S-KPS	59.91	23.34
WF^{2}Q+	35.2	23.33

Table 3: Simulation results. Maximum end-to-end delays in ms.

tem), are substantially greater than with WF^{2}Q+, and the standard deviation indicates a slightly more pronounced jitter in the service. On the other hand, the highest priority flow, f_1 does not suffer from the same, and in these simulations it receives the same treatment it receives in WF^{2}Q+.

7. EXPERIMENTAL RESULTS

Together with the good service guarantees, the most interesting feature of QFQ is the constant (independent of the number of flows) and small per-packet execution time, which makes the algorithm extremely practical. To study the actual performance of our algorithm, and compare it with other alternatives, we have built C versions of QFQ and various other schedulers, including S-KPS. Our code is directly usable as a kernel module in the scheduling frameworks in Linux ("tc") and FreeBSD ("dummynet"), and includes all necessary (and potentially time-consuming) operations such as parameter checking and accounting.

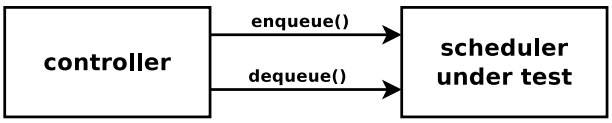

Figure 9: Our testing environment: a controller drives the scheduler module with programmable sequences of requests.

We have performed a thorough performance analysis by running the scheduler code in a test environment where we could precisely control the sequence of enqueue/dequeue requests presented to the schedulers (Fig. 9). The controller lets us decide the number and distribution of flow weights and packet sizes, as well as keep track of the number of backlogged flows and the total amount of traffic queued in the scheduler. These parameters impact the behaviour of schedulers in different ways; especially, the memory usage and access patterns are important on modern CPUs, where cached and non-cached access times differ by one order of magnitude or more.

7.1 Measurement details

Ideally we would like to take individual samples of the duration of each *enqueue()* or *dequeue()* operation. Unfortunately some of these operations are extremely fast, in the order of tens of nanoseconds and less: in fact, *enqueue()*'s in many cases boil down to a simple list append, and in many of the simplest schedulers (e.g. FIFO or DRR) *dequeue()*'s are equally simple. The measurements tools (TSC or CPU performance counters) available on modern CPUs are still not sufficiently accurate for measuring such short intervals, because of the effects of out-of-order instruction processing and deep CPU pipelines.

On top of this, even having precise timing measurement tools would not help much for performing worst case analysis, as the execution time of such small pieces of code as the ones we are trying to measure is heavily affected by cache misses and memory contention.

To achieve more reliable results, we did not pursue this granularity of measurements, and instead evaluated the total execution time for experiments where the controller generates a very large number (5 millions or more) of *enqueue()* and *dequeue()* requests. By dividing the total time the number of requests, we can estimate the *average* cost of an *enqueue()/dequeue()* pair, inclusive of the time spent by the controller for packet generation and disposal. To estimate the controller costs, for each configuration we have run experiments with the scheduler replaced by empty calls.

This approach does not allow us to separate the cost of *enqueue()* and */dequeue()* operations, but this is not particularly important for the following reasons. First, in the steady state, there is approximately the same number of calls for the two functions. Only when a link is severely overloaded the number of *enqueue()* will be much larger than its counterpart, but in this case dropping a packet is a very inexpensive operation. Second, in most algorithms it is possible to move some operations between *enqueue()* and *dequeue()*, so it is really the sum of the two costs that counts to judge the overall performance of an algorithm.

Our tests include the following schedulers:

NONE This is actually the baseline case which we use to measure the cost of packet generation and disposal, including memory-touching operation that may affect the behaviour of the cache in other experiments. In this configuration, packets generated by the controller are just stored in a FIFO queue from where they are extracted when the controller would request a *dequeue()*;

FIFO this is the simplest possible scheduler, consisting in an unbounded FIFO queue. Compared to the baseline case, here we exercise the scheduler's API, which causes one extra function calls and counter updates on each request;

DRR this implements the Deficit Round Robin scheduler, where each flow has a configurable quantum;

QFQ the algorithm described in this paper. The code follows very closely the description given here. We use 19 groups, packet sizes up to 2KBytes, and weights between 1 and 2^{16};

S-KPS the scheduler described in [8]. The code has been implemented from the paper with some minor optimizations. Internal parameters (e.g.

l_{min}, l_{max}) have been set to values similar to those used for QFQ;

WF2Q+ this is an implementation of the WF^{2}Q+ algorithm taken from the FreeBSD's dummynet code. It has $O(\log N)$ scaling properties, but it is of some interest to determine the break-even point between schedulers with different asymptotical behaviour.

In terms of traffic patterns, we ran extensive tests with different combinations and number of flows (from 1 to 128K), with various weight and packet size distributions. These configurations show how the schedulers depend on the number of flow, traffic classes and also their sensitivity to memory access times.

To emulate different load conditions for the link, we generate requests for the scheduler with three patterns: *SMALL* and *LARGE* generate bursts of 5N and 30N packets, respectively (where N is the number of active flows), and then completely drain the scheduler; *FULL* keeps the scheduler constantly busy, with a total backlog between 3N and 30N packets.

The bursty patterns try to reproduce operation on a normally unloaded link, whereas the "full' pattern mimics the behaviour of a fully loaded link driven by TCP or otherwise adaptive flows, which modify their offered load depending on available bandwidth.

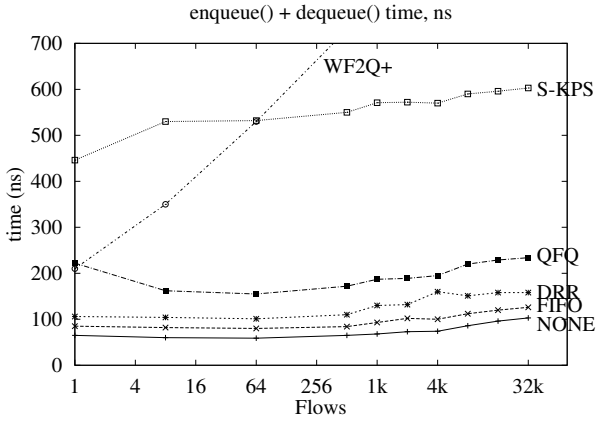

Figure 10: Scaling properties of the various algorithms. WF^{2}Q+ grows as $O(\log N)$, reaching 2000 ns for 32k flows (see Table 4).

7.2 Results

Table 4 and Figure 10 report some of the most significant test results. These experiments have been run on a low-end desktop machine (2.1GHz CPU, 32-bit OS, 667MHz memory bandwidth), with code compiled with gcc -O3. Experiments ran on different platforms have produced results that scale mostly linearly with the platform's performance. The point where cache effects become visible, however, varies on each system depending on available cache sizes.

Measurement results in ns for an *enqueue()/dequeue()* pair and packet generation. **Standard deviations are within 3% of the average**, so we do not report them to reduce the clutter in the table

	NONE	FIFO	DRR	**QFQ**	S-KPS	WF^{2}Q+
1 flow						
small	65	86	107	**221**	419	220
large	65	85	106	**222**	446	210
full	62	83	105	**221**	450	210
8 flows						
small	62	80	102	**168**	458	356
large	60	82	104	**162**	530	350
full	60	80	102	**163**	543	344
64 flows						
small	62	80	100	**162**	454	528
large	59	80	101	**155**	532	530
full	59	80	100	**158**	540	526
512 flows						
small	62	82	111	**170**	468	732
large	65	84	110	**172**	550	730
full	64	85	110	**175**	560	740
4096 flows						
small	72	100	150	**212**	507	1100
large	74	100	160	**195**	570	1090
full	74	102	157	**197**	590	1110
32768 flows						
small	90	114	185	**230**	550	1900
large	103	126	158	**234**	603	1880
full	62	117	147	**222**	601	1690
1:32k,2:4k,4:2k,8:1k,128:16,1k:1 flows						
small	91	120	167	**247**	598	1868
large	107	131	160	**250**	595	1734
full	92	119	160	**255**	612	1715

Table 4: A subset of experimental results. On top, experiment with increasing number of flows, all belonging to the same class. The last group shows a mix of flows with exponentially increasing weights, and exponentially decreasing number of flows for each class.

Figure 10 shows clearly that all $O(1)$ algorithms do not depend on the number of flows, whereas WF2Q+ shows the expected $O(\log N)$ behaviour. We see that DRR and FIFO are really inexpensive, and most of the time in the test is consumed by the packet generator (the curve labeled NONE in the figure), which accounts for approximately 60 ns per enqueue/dequeue pair. All schedulers, and the generator itself, show a modest increase of the execution time as the number of flows goes (on this particular platform) above 4k. This is likely due to the working set of the algorithm overflowing the available cache, which causes cache misses that impact

on the total execution time. In absolute terms, QFQ behaves really well, consuming about 100-110 ns (excluding the traffic generation) up to the point where cache misses start to matter. S-KPS also has reasonably good performance, taking approximately 500 ns, although this is between 2.5 and 3 times the cost of QFQ.

Finally, we would like to note that while WF2Q+ has obvious scalability issues, for configurations with a small number of flows it can still be a viable alternative.

The final block of the table reports the result of experiments with a large mix of flows using different weights. This case does not show significant differences with the case where all flows have the same parameters.

Looking at the values in Table 4, and in other experiments not reported here, we can see that algorithms can have peculiar behaviours in certain conditions.

As an example, the row with 1 flow shows that QFQ takes a modest performance hit when there is only one flow backlogged. This happens because, in the dequeue code, the removal of the flow from the group leaves the group empty and triggers unnecessary calls to *unblock_groups()* and *make_eligible()*. S-KPS seems to have slightly better performance in presence of small burst, presumably due to similar reasons (certain code paths becoming more frequent). DRR also exhibits similar differences in performance when the packet size is not matched with the quantum size, as certain packets require two rounds instead of one to be processed.

Overall, these variations tend to be small in absolute and relative terms, and we only see them because we are dealing with extremely fast algorithms where even small changes in the instruction counts can be measured.

8. AVAILABILITY

A complete implementation of QFQ and S-KPS is available at XXX (removed for anonimity purposes; please ask the TPC chairs). The code includes kernel modules for use with Linux and FreeBSD, as well as the traffic generator used for the experiments. To the best of our knowledge, this the first publicly available implementation of a scheduler of this class ($O(1)$ time and near-optimal WFI).

9. CONCLUSIONS

In this paper we presented QFQ, an approximated implementation of WF^{2}Q+ which can run in constant time, with very low constants and using extremely simple data structures. Together with a detailed description of the algorithm, we provide a theoretical analysis of its service properties, and present an accurate performance analysis, comparing QFQ with a variety of other schedulers.

The experimental results show that QFQ lives up to its promises. The scheduler only takes 110 ns per *enqueue()/dequeue()* pair, which is only twice the time taken by DRR. The algorithm is based on extremely simple instructions, and uses very small and localized data structures, which make it amenable to a hardware implementation.

10. REFERENCES

[1] *Removed for anonimity purposes, copy available from the TPC chairs.*

[2] `http://www.isi.edu/nsnam/ns/`.

[3] Jon C. R. Bennett and Hui Zhang. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on Networking*, 5(5):675–689, 1997.

[4] Guo Chuanxiong. SRR: An O(1) time complexity packet scheduler for flows in multi-service packet networks. In *In ACM SIGCOMM 2001*, pages 211–222, 2001.

[5] J. C. R. Bennett e H.Zhang. WF^{2}Q: Worst-case fair weighted fair queueing. *Proceedings of IEEE INFOCOM '96*, pages 120–128, March 1996.

[6] Chuanxiong Guo. G-3: An o(1) time complexity packet scheduler that provides bounded end-to-end delay. In *INFOCOM*, pages 1109–1117, 2007.

[7] H. Abdel-Wahab I. Stoica. Earliest eligible virtual deadline first: A flexible and accurate mechanism for proportional share resource allocation. Technical Report 95-22*, Department of Computer Science, Old Dominion University, November 1995*, Nov 1995.

[8] M. Karsten. Approximation of generalized processor sharing with stratified interleaved timer wheels. *To appear on ToN, available online at* `http://www.cs.uwaterloo.ca/~mkarsten/papers/ton2010.html`.

[9] M. Karsten. SI-WF^{2}Q: WF^{2}Q approximation with small constant execution overhead. *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1–12, April 2006.

[10] A. Kortebi, L. Muscariello, S. Oueslati, and J. Roberts. Evaluating the number of active flows in a scheduler realizing fair statistical bandwidth sharing. *SIGMETRICS Perform. Eval. Rev.*, 33(1):217–228, 2005.

[11] Luciano Lenzini, Enzo Mingozzi, and Giovanni Stea. Tradeoffs between low complexity, low latency, and fairness with deficit round-robin schedulers. *IEEE/ACM Trans. Netw.*, 12(4):681–693, 2004.

[12] Sriram Ramabhadran and Joseph Pasquale. The stratified round robin scheduler: design, analysis and implementation. *IEEE/ACM Trans. Netw.*, 14(6):1362–1373, 2006.

[13] M. Shreedhar and George Varghese. Efficient fair

queuing using deficit round robin. In *IEEE/ACM Transactions on Networking*, pages 375–385, 1995.
[14] D.C. Stephens, J.C.R. Bennett, and Hui Zhang. Implementing scheduling algorithms in high-speed networks. *Selected Areas in Communications, IEEE Journal on*, 17(6):1145–1158, Jun 1999.
[15] D. Stiliadis and A. Varma. A general methodology for designing efficient traffic scheduling and shaping algorithms. *Proc. of INFOCOM '97*, 1:326–335 vol.1, 7-12 Apr 1997.
[16] Subhash Suri, George Varghese, and Girish Chandranmenon. Leap forward virtual clock: A new fair queuing scheme with guaranteed delays and throughput fairness. In *Proceedings of INFOCOM'97*, 1997.
[17] J. Xu and R. J. Lipton. On fundamental tradeoffs between delay bounds and computational complexity in packet scheduling algorithms. In *In Proceedings of ACM SIGCOMM '02*, 2002.

APPENDIX

A. PROOFS

We prove here both the properties of the data structure used in Sec. 4, and the B/T-WFI of QFQ. In this section we explicitly indicate the time at which any timestamp is computed to avoid ambiguity, and we assume that the various quantities ($V(t)$, $S^k(t)$, ...) are computed as described in the QFQ algorithm. Given a generic function of time $f(t)$, we define $f(t_1^+) \equiv \lim_{t\to t_1^+} f(t)$. For notational convenience, we avoid writing $f(t_c^+)$ if $f(t)$ is continuous at time t_c. To further simplify the notation, if the function is discontinuous at a time instant t_d, we assume, without losing generality, that $f(t_d) \equiv \lim_{t\to t_d^-} f(t)$, i.e., that the function is left-continuous. Finally, recall that all the quantities used hereafter are also reported in Table 1.

We define the following two notations for convenience:

$$\lfloor x \rfloor_{\sigma_i} \equiv \lfloor \frac{x}{\sigma_i} \rfloor \sigma_i \ , \ \lceil x \rceil_{\sigma_i} \equiv \lceil \frac{x}{\sigma_i} \rceil \sigma_i$$

For any positive quantity $y < x + \sigma_i$, we have

$$\lfloor y \rfloor_{\sigma_i} \leq \lceil x \rceil_{\sigma_i}. \tag{13}$$

In fact, x can be written as $x = n\sigma_i + \delta$, with $0 \leq \delta < \sigma_i$. If $\delta = 0$ then $y < (n+1)\sigma_1 \implies \lfloor y \rfloor_{\sigma_i} \leq n\sigma_i$, $\lceil x \rceil_{\sigma_i} = n\sigma_i$, and the thesis holds; if $\delta > 0$ then $\lfloor y \rfloor_{\sigma_i} \leq (n+1)\sigma_i$, $\lceil x \rceil_{\sigma_i} = (n+1)\sigma_i$, and the thesis holds too.

A.1 Group GBT under QFQ

We start by proving per-group upper bounds for $S_i(t) - V(t)$ (in Theorem 3) and for $V(t) - F_i(t)$ (Theorem 4, supported by the two long Lemmas 1 and 2). The two bounds represent a group-based variant of the *Globally Bounded Timestamp* (GBT) property, normally defined for the flow timestamps in an exact virtual time-based scheduler. Lemmas 1 and 2 are an adapted version of the ones in [**?**], repeated here for convenience, with permission from the author.

We will use these bounds to prove both the properties of the data structure and the B/T-WFI of QFQ.

THEOREM 3. **Upper bound for $S_i(t) - V(t)$.** *For any backlogged group i and $\forall t$*

$$S_i(t) \leq \left\lceil \frac{V(t)}{\sigma_i} \right\rceil \sigma_i = \lceil V(t) \rceil_{\sigma_i} \tag{14}$$

PROOF. By definition (5), at any time t and for any group i, $S_i(t)$ is an integer multiple of σ_i and, for any backlogged flow k of the group, $S_i(t) \leq \hat{S}^k(t) = \lfloor S^k(t) \rfloor_{\sigma_i}$. It follows that, if $S^k(t) < V(t) + 2\sigma_i$, then (14) trivially holds. Hence, to prove (14) we actually prove the latter, i.e., that $S^k(t) < V(t) + 2\sigma_i$, and to prove it we consider only a generic time instant t_1 at which a generic packet for flow k is enqueued/dequeued, as this is the only event upon which which $S^k(t)$ may increase.

According to (2), either $S^k(t_1^+) = V(t_1)$, in which case the packet is enqueued and the thesis trivially holds, or $S^k(t_1^+) = F^k(t_1)$. In this case flow k must have had a packet previously dequeued at time $t_p < t_1$.

When the packet was dequeued at t_p flow k was certainly eligible, and $V(t)$ is immediately incremented after the dequeue at t_p, so we have $F^k(t_1) = S^k(t_p^+) = S^k(t_p) + l^k(t_p)/\phi^k \leq V(t_p) + \sigma_i + l^k(t_p)/\phi^k \leq V(t_p) + 2\sigma_i < V(t_p^+) + 2\sigma_i$, which proves the thesis. □

LEMMA 1. *Let $I(t) = \{k : k \in B(t), S^k(t) \geq V(t)\}$ be a subset of flows. Given a constant V', $\forall t : V(t) \leq V'$ we have:*

$$\sum_{k\in I(t)} \left(l^k(t) + \phi^k [V' - F^k(t)] \right) \leq V' - V(t) \tag{15}$$

where $l^k(t)$ is the size of the first packet in the queue for flow k at time t.

PROOF. By definition, $l^k(t) = \phi^k[F^k(t) - S^k(t)]$. Thus, for flows in set $I(t)$ we have $l^k(t) \leq \phi^k[F^k(t) - V(t)]$. Therefore, with simple algebraic passages:

$$\begin{aligned} \textstyle\sum_{k\in I(t)} \left\{ l^k(t) + \phi^k [V(t) - F^k(t)] \right\} &= \\ \textstyle\sum_{k\in I(t)} \left\{ l^k(t) + \phi^k [V(t) - V'] + \phi^k [V' - F^k(t)] \right\} &\leq \\ 0 & \end{aligned} \tag{16}$$

This implies:

$$\begin{aligned} \textstyle\sum_{k\in I(t)} \left\{ l^k(t) + \phi^k [V' - F^k(t)] \right\} &\leq \\ \textstyle\sum_{k\in I(t)} \phi^k [V' - V(t)] &\leq \\ V' - V(t) & \end{aligned} \tag{17}$$

where the last passage uses $\sum_{k\in I} \phi^k \leq 1$. □

LEMMA 2. *Let $X(t,M) \equiv \{k : \hat{F}^k(t) \leq M\}$ be a set of flows. Given a constant V', we have that $\forall t$: $L + V' \geq V(t)$:*

$$\sum_{k \in X(t,V')} \left(l^k(t) + \phi^k[V' - F^k(t)]\right) \leq L + V' - V(t) \quad (18)$$

PROOF. The proof is by induction over those events that change the terms in (18): packet enqueues for idle flows, packet dequeues and virtual time jumps. The base case where X is empty is true by assumption. For the inductive proof, we assume (18) to hold at some time t_1.

Packet enqueue for an idle flow: Say a packet of size l_1 of the idle flow k arrives at time t_1. $V(t)$ does not change on packet arrivals except for virtual time jumps, that are dealt with later.
If after the enqueue of the new packet $k \notin X(t_1^+, V')$, i.e. $\hat{F}^k(t_1^+) > V'$, we must consider two sub-cases. First, if $k \notin X(t_1, V')$ nothing changes. Second, if $k \in X(t_1, V')$ the positive component $\phi^k[V' - F^k(t_1)]$ is removed from the sum, so the left hand side of (18) decreases. In both sub-cases the lemma holds. The remaining case is if $k \in X(t_1^+, V')$. Since $\hat{F}^k(t_1^+) > \hat{F}^k(t_1)$, this implies $k \in X(t_1, V')$. In this case $l^k(t)$ is incremented by l_1, but $F^k(t)$ is incremented by l_1/ϕ^k, so the left hand side of (18) remains unchanged and the lemma holds.

Virtual time jump: After a virtual time jump, all backlogged flows have $S^k(t_1^+) \geq \hat{S}^k(t_1^+) \geq V(t_1^+)$. With regard to the idle flows, we assume that their virtual start and finish times are pushed to $V(t_1^+)$. By doing so we do not lose generality, as the virtual start times of these flows will be lower-bounded by $V(t)$ when they become backlogged (again). Besides, it is easy to see that pushing up their virtual finish times may only let the left side of (18) decrease. In the end $S^k(t_1^+) \geq V(t_1^+)$ for all flows and, if $V' \geq V(t_1^+)$ then Lemma 1 applies and the lemma holds. For other V' in $[V(t_1^+) - L, V(t_1^+)[$, the additional L term in (18) absorbs any decrement on the right hand side. Therefore, the lemma holds.

Packet dequeue: Flow k receives service at time t_1 for its head packet of size $l^k(t_1)$. We have to distinguish two cases, depending on V' and $\hat{F}^k(t_1)$.

Case 1 – $V' \geq \hat{F}^k(t_1)$. $V(t)$ is incremented exactly by $l^k(t_1)$, so the right side of (18) decreases exactly by $l^k(t_1)$.

With regard to the left side, the variation of $l^k(t)$ can be seen as the result of first decreasing by $l^k(t_1)$, which balances the above decrement of $V(t)$, and then increasing by $l^k(t_1^+)$, which is in turn balanced by incrementing $F^k(t)$ by $\frac{l^k(t_1^+)}{\phi^k}$. Hence the lemma holds.

Case 2 – $V' < \hat{F}^k(t_1)$. In this case all flows $h \in X(t_1, V')$ have $\hat{F}^h(t_1) < \hat{F}^k(t_1)$, so they must have been ineligible according to their rounded start time, otherwise the current flow k would have not been chosen. Therefore, $V(t_1) < \hat{S}^h(t_1) \leq S^h(t_1)$ for all flows in $X(t_1, V')$. Lemma 1 applies then for all $V' \geq V(t_1)$, i.e.

$$\sum_{k \in X(t_1^+, V')} \left(l^k(t_1) + \phi^k[V' - F^k(t_1)]\right) \leq V' - V(t_1). \quad (19)$$

Because $V(t_1^+) = V(t_1) + l^k$ and we assume $L + V' \geq V(t_1^+)$ after service, we only need to consider V' with $L + V' \geq V(t_1^+) + l^k$ before service. Therefore

$$V' - V(t_1) \leq (L - l^k) + V' - (V(t_1^+) - l^k) = L + V' - V(t_1^+) \quad (20)$$

and the lemma holds after service. □

THEOREM 4. **Upper bound for $V(t) - F_i(t)$**
For any backlogged group i

$$V(t) \leq F_i(t) + L. \quad (21)$$

PROOF. To prove the thesis we will actually prove the more general inequality $V(t) \leq \hat{F}^k(t) + L$ for a generic flow k of group i. The proof is by contradiction. The only event that could lead to a violation of the assumption is serving a packet. Assume that at $t = t1$: $V(t_1) = V_1$ the lemma holds. A packet p with rounded finish time $\hat{F}_1$ and length l_p is served and afterwards at time $t_2 : V(t_2) = V_2$, there is a packet q with finish time F_2, such that $\hat{F}_2 + L < V_2$. Denote with $\hat{S}_1$ and $\hat{S}_2$ the corresponding start times. We need to distinguish three cases.

Case 1: Packet q is eligible at time t_1 according to its rounded start time. Then, $\hat{F}_2 \geq \hat{F}_1$ (both packets were eligible at V_1 and p was chosen). Applying Lemma 2 with $t = t_1$ and $V' = \hat{F}_2$ results in

$$\sum_{k \in X(t_1, \hat{F}_2)} l^k(t_1) + \sum_{k \in X(t_1, \hat{F}_2)} (\hat{F}_2 - F^k(t_1))\phi^k \leq L + \hat{F}_2 - V(t_1) \quad (22)$$

Because $F^k(t) \leq \hat{F}^k(t)$, the second term on the left side of the inequality is non-negative and therefore

$$l_p \leq \sum_{k : \hat{F}^k(t) \leq \hat{F}_2} l^k(t) \leq L + \hat{F}_2 - V_1 \quad (23)$$

$$V_2 - V_1 \leq L + \hat{F}_2 - V_1 \quad (24)$$

$$V_2 \leq \hat{F}_2 + L \quad (25)$$

The step from (23) to (24) uses $V_1 + l_p = V_2$.

Case 2: Packet q is not eligible at V_1 according to its rounded start time, but becomes eligible between V_1 and V_2. Then, $\hat{S}_2 \geq V_1$. Virtual time advances by at most L and therefore:

$$\hat{F}_2 \geq \hat{S}_2 \geq V_1 \geq V_2 - L \quad (26)$$

Case 3: Packet q is not eligible according to its rounded start time after service to p, therefore V_2 is

reached by a virtual time jump before q can be served. In this case:

$$\hat{F}_2 \geq \hat{S}_2 \geq V_2 \geq V_2 - L \tag{27}$$

This concludes the proof. □

A.2 Proofs of the data structure properties

We can now prove the theorems used in Sec. 4 considering the only two events that can change the state of the scheduler, namely packet enqueue and packet dequeue. We start from Theorem 5, which gives the per-group slot occupancy.

THEOREM 5. *At all times, only the first* $2+\lceil \frac{L}{\sigma_i} \rceil$ *consecutive slots beginning from the head slot of a group may be non empty.*

PROOF. Consider a generic flow k belonging to a group i. A new virtual start time may be assigned to the flow (only) as a consequence of the enqueueing/dequeueing of a new packet $p^{k,l}$ at a time instant t_p. As in the proof of Lemma 3, from (2) $S^k(t_p^+)$ may be equal to either (a) $V(t_p)$, or (b) $F^k(t_p)$, where we assume $F^k(t_p) = 0$ if $p^{k,l}$ is the first packet of the flow to be enqueued/dequeued.

In the first case, according to (21), $S^k(t_p^+) = V(t_p) \leq F_i(t_p) + L \leq S_i(t_p) + 2\sigma_i + L \leq S_i(t_p) + 2\sigma_i + \lceil \frac{L}{\sigma_i} \rceil \sigma_i$. In the second case, neglecting the trivial sub-case $F^k(s^{k,l-1+}) = 0$, we can consider that flow k had to be a head flow when $p^{k,l-1}$ was served. Hence, according to (5), $S^k(t_p) < S_i(t_p) + \sigma_i$. From (2), this implies $S^k(t_p^+) = F^k(t_p) < S_i(t_p) + 2\sigma_i \leq S_i(t_p) + 2\sigma_i$.

Considering both cases, it follows that, $\forall t$ $S^k(t) - S_i(t) < (2 + \lceil \frac{L}{\sigma_i} \rceil)\sigma_i$, i.e., that at any time the virtual start times of all the backlogged flows of a group may belong only to the $2 + \lceil \frac{L}{\sigma_i} \rceil$ consecutive slots beginning from the one the head slot queue is associated to, which proves the thesis. □

Using the following lemma, we want now to prove that **ER** is ordered by virtual finish times.

LEMMA 3. *Let* $\overline{t}$ *be the time instant at which a previously idle group* i *becomes backlogged, or at which the group, previously ineligible, becomes eligible, or finally at which the virtual finish time of the group decreases. We have that* $F_h(\overline{t}) \leq F_i(\overline{t}^+)$ *for any backlogged group* h *with* $h < i$.

PROOF. For $F_i(t)$ to decrease, $S_i(t)$ must decrease as well. According to the *enqueue()* and *dequeue()*, this can happen only in consequence of the enqueueing of a packet of an empty flow of the group. As this is exactly the same event that may cause a group to become backlogged, then, from (2) we have $S_i(\overline{t}^+) \geq \lfloor V(\overline{t}) \rfloor_{\sigma_i}$ both if the group become backlogged and if $F_i(t)$ decreases. Substituting this inequality, which finally holds also if the group becomes eligible at time $\overline{t}$, and (14) in the following difference we get:

$$\begin{aligned} F_h(\overline{t}) - F_i(\overline{t}^+) &= \\ S_h(\overline{t}) + 2\sigma_h - S_i(\overline{t}^+) - 2\sigma_i &= \\ S_h(\overline{t}) - S_i(\overline{t}^+) + 2\sigma_h - 2\sigma_i &\leq \\ \lceil V(\overline{t}) \rceil_{\sigma_h} - \lfloor V(\overline{t}) \rfloor_{\sigma_i} + 2\sigma_h - 2\sigma_i &\leq \\ \lceil V(\overline{t}) \rceil_{\sigma_i} - \lfloor V(\overline{t}) \rfloor_{\sigma_i} + 2\sigma_h - 2\sigma_i &\leq \\ \sigma_i + 2\sigma_h - 2\sigma_i &= \\ 2\sigma_h - \sigma_i &\leq 0 \end{aligned} \tag{28}$$

where $\lceil V(\overline{t}) \rceil_{\sigma_h} \leq \lceil V(\overline{t}) \rceil_{\sigma_i}$ and the last inequality follow from that, as $i > h$, $\sigma_i \geq 2\sigma_h$. □

The following theorem guarantees that **ER** is always ordered by virtual finish times. Then it guarantees that this order is never broken when one or more groups are inserted into it during QFQ operation.

THEOREM 6. *Set* **ER** *is ordered by group virtual finish time.*

PROOF. We will prove the thesis by induction. In the base case **ER** $= \emptyset$ the thesis trivially holds. The ordering of **ER** may change only when one or more groups enter the set. This can happen as a consequence of 1) a group entering **ER** as it becomes backlogged, 2) one or more groups moving from **IR** to **ER**, 3) one or more groups moving from **EB** to **ER**. Let i be a group entering **ER** at time t_1 for one of the above three reasons, and let the thesis hold before time t_1.

In the first case, thanks to Lemma 3 $F_i(t_1^+)$ is not lower than the virtual finish times of the groups in **ER** with lower index. By definition of **ER**, $F_i(t_1^+)$ is also not higher than the virtual finish times of the groups in **ER** with higher index.

In the second case, given a group $h \in$ **ER** with $h < i$, $S_i(t_1) \geq S_h(t_1)$ because either group h was already in **ER** before time t_1, or group h belonged to **IR**, which is ordered by virtual start times according to [Sec.4.1.3, item 2]. This implies $F_i(t_1) \geq F_h(t_1)$ because $\sigma_i \geq 2\sigma_h$. By definition of **IR**, $F_i(t_1)$ is also not higher than the virtual finish times of the groups in **ER** with higher index.

In the third case, since group i is not blocked any more, $F_i(t_1)$ is not higher than the virtual finish times of the groups in **ER** with higher index. With regard to the groups with lower index than i, for group i to be blocked before time t_1 there had to be a group $b \in$ **ER** with $b > i$ and $F_b(t_1) < F_i(t_1)$. Since we assume that **ER** is ordered by virtual finish time before time t_1, then $F_b(t_1)$, and hence $F_i(t_1)$ is not lower than the virtual finish times of all the lower index groups in **ER**. □

To prove that **EB** enjoys the same order property as **ER**, we need first a further lemma. The validity of the lemma depends on the timestamp *back-shifting* performed under QFQ when inserting a newly backlogged

group into **EB**. Hence this is the right moment to explain in detail this operation. When an idle group i becomes blocked after enqueuing a packet of a flow k at time t_p, the timestamps of flow k are not updated using the following variant of (2):

$$\begin{array}{l} S^k(t_p^+) \leftarrow \max\left[\min(V(t_p), F_b(t_p)),\ F^k(t_p)\right] \\ F^k(t_p^+) \leftarrow S^k + l^k(t_p^+)/\phi^k \end{array} \quad (29)$$

where b is the lowest order group in **ER** such that $b > i$. Basically, with respect to the exact formula, $F_b(t_p)$ is used instead of $V(t_p)$ if $V(t_p) > F_b(t_p)$. This is done because otherwise the ordering by virtual finish time in **EB** may be broken. It would be easy to show that this would happen if an idle group becomes blocked when $V(t)$ is too higher than the virtual finish time of some other blocked group $h < i$.

With regard to worst-case service guarantees, in case $V(t_p) > F_b(t_p)$ in (29), group i just benefits from the back-shifting, whereas the guarantees of the other flows are unaffected. To prove it, consider that the guarantees provided to any flow do not depend on the actual arrival time of the packets of the other flows. Hence one can still "move" a pair of timestamps backwards, provided that this does not lead to an inconsistent schedule, i.e., provided that the resulting worst-case schedule for all the flows is the same as if the packet had actually arrived at a time instant such that the would have got exactly those timestamps without using any back-shifting. This is what happens using (29), for the following reason. Should the packet that lets group i become backlogged have arrived at a time instant $\overline{t}_p \leq t_p$ at which $V(\overline{t}_p) = F_b(t_p)$, group i would have however got a virtual finish time higher than $F_b(t_p)$. Hence group i should not have been served before group b, exacly as it happens in the schedule resulting from timestamping group i with (29) at time t_p.

We can now prove the intermediate lemma we need to finally prove the ordering in **EB**.

LEMMA 4. *If a pair of groups h and i with $h < i$ are blocked at a generic time instant t_2, then $S_h(t_2) \leq F_i(t_2)$.*

PROOF. We consider two alternative cases. The first is that $S_h(t_2)$ has been last updated at a time instant $t_1 \leq t_2$ using (29). The second is that, according to (2) and (5) there are at least one head flow k of group h and a time instant $t_1 \leq t_2$ such that $S_h(t_2) = \lfloor F^k(t_1) \rfloor_{\sigma_h}$.

In the first case we have $S_h(t_2) \leq F_b(t_1)$, where b is the lowest order group in **ER** such that $b > h$. We can consider two sub-cases. First, group i is already backlogged and eligible at time t_1. It follows that, if $i \geq b$ then $F_i(t_1) \geq F_b(t_1)$. Otherwise, from the definition of b, group i is necessarily blocked, and $F_i(t_1) > F_b(t_1)$ must hold again for group b not to be blocked. In the end, regardless of whether group i is ready or blocked, $F_i(t_2) \geq F_i(t_1) > F_b(t_1) = S_h(t_2)$ and the thesis holds. In the other sub-case, i.e., group i is not ready and eligible at time t_1, thanks to Lemma 3 group i cannot happen to have a virtual finish time lower than $F_h(t_1)$ during $(t_1, t_2]$. Hence $F_i(t_2) \geq F_h(t_1) = F_h(t_2) > S_h(t_2)$ and the thesis holds.

In the other case, i.e., $S_h(t_2) = \lfloor F^k(t_1) \rfloor_{\sigma_h}$, we prove the thesis by contradiction. Suppose that $S_h(t_2) > F_i(t_2)$. Flow k must have necessarily been served with $F^k(t_0) = F^k(t_1)$ at some time $t_0 \leq t_1$. In addition, for $S_h(t_2) > F_i(t_2)$ to hold, $F^k(t_1) > F_i(t_2)$ and hence $F^k(t_0) > F_i(t_2)$ should hold as well. As flow k had to be a head flow at time t_0, it would follow that

$$F_h(t_0) \geq F^k(t_0) > F_i(t_2). \quad (30)$$

We consider two cases.

First, group i is backlogged at time t_0. If $F_i(t_0) < F_h(t_0)$, then $S_i(t_0) = F_i(t_0) - 2\sigma_i < F_h(t_0) - 2\sigma_i < S_h(t_0)$, because $\sigma_i > \sigma_h$. Hence, both group h and i would be eligible, and group h could not be served at time t_0. In follows that $F_i(t_0) \geq F_h(t_0)$ should hold. This inequality and (30) would imply $F_i(t_0) > F_i(t_2)$. Should not $F_i(t)$ decrease during $[t_0, t_2]$, the absurd $F_i(t_2) > F_i(t_2)$ would follow. But, from *enqueue()* and *dequeue()* it follows that the only event that can let $F_i(t)$ decrease is the enqueueing of a packet of an idle flow of group i that causes $S_i(t)$ to decrease (lines 12-18 of *enqueue*). Let $F_{i,min}$ be the minimum value that $F_i(t)$ may assume in consequence of this event.

Since $\forall t \in [t_0, t_2]\ V(t) \geq S_h(t_0)$, according to (2), (5) and (30), $F_{i,min} \geq \lfloor S_h(t_0) \rfloor_{\sigma_i} + 2\sigma_i \geq S_h(t_0) - \sigma_h + 2\sigma_i = F_h(t_0) - 3\sigma_h + 2\sigma_i > F_h(t_0) > F_i(t_2)$, which again would imply the absurd $F_i(t_2) > F_i(t_2)$.

The second case is that group i is not backlogged at time t_0. As the event that would let the group become backlogged after time t_0 is the same that might have let $F_i(t)$ decrease in the other case, then, using the same arguments as above, we would get the same absurd.

In the end, $S_h(t_2) \leq F_i(t_2)$ must hold. □

The following theorem guarantees that **EB** is always ordered by virtual finish time (hence, as previously proven for **ER** this order is never broken during QFQ operations).

THEOREM 7. *Set* **EB** *is ordered by group virtual finish time.*

PROOF. We will prove the thesis by induction. In the base case $\mathbf{EB} = \emptyset$ the thesis trivially holds. The only event upon which the the ordering of **EB** may change is when one or more groups enters the set. The three events that may cause a group to become blocked are 1) the enqueueing/dequeueing of a packet of a flow of an idle group $j > i$, which lets group j get a lower virtual finish time than group i (groups with lower order than i can never block group i); 2) the enqueueing/dequeueing

of a packet of a flow of group i itself, which lets the virtual finish time of group i become higher than the virtual finish of some higher order group; 3) the growth of $V(t)$, which causes one or more groups to move from **IB** to **EB**.

With regard to the first event, it is worth noting that group j can cause group i to become blocked only if group j becomes backlogged or if $F_j(t)$ decreases. Let t_1 be the time instant at which one of these two events occurs and such that **EB** is ordered up to time t_1. Thanks to Lemma 3, $F_i(t_1) \leq F_j(t_1^+)$ and hence the event cannot let group i become blocked.

Suppose now that, at time t_1, group i enters **EB** as a consequence of either a packet of a flow of the group being enqueued/dequeued or the growth of $V(t)$. We will prove that, given two any blocked groups $h < i$ and $j > i$, $F_h(t_1) \leq F_i(t_1^+)$ and $F_i(t_1^+) \leq F_j(t_1)$ hold (where $F_i(t_1^+) = F_i(t_1)$ in case group i enters **EB** from **IB**).

With regard to a blocked group $h < i$, if group i enters **EB** as a consequence of a packet enqueue/dequeue, then, from Lemma 4 and the fact that, as $F_i(t)$ is an integer multiple of σ_i, $F_i(t_1^+) \geq F_i(t_1) + \sigma_i$, we have

$$\begin{array}{rl} F_i(t_1^+) - F_h(t_1) & \geq \\ F_i(t_1) + \sigma_i - S_h(t_1) - 2\sigma_h & \geq \\ F_i(t_1) + \sigma_i - F_i(t_1) - 2\sigma_h & \geq \\ \sigma_i - 2\sigma_h & \geq 0 \end{array} \tag{31}$$

where the last inequality follows from $\sigma_i \geq 2\sigma_h$. On the other hand, if group i enters **EB** from **IB**, then $S_i(t_1) \geq S_h(t_1)$ because either group h was already eligible before time t_1, or group h belonged to **IB**, which is ordered by virtual start time according to [Sec.4.1.3, item 2]. This implies $F_i(t_1) \geq F_h(t_1)$ because $\sigma_i \geq 2\sigma_h$.

With regard to a blocked group $j > i$, let $b > j > i$ be the highest order group that is blocking group j at time $\overline{t}$. Independently of the reason why group i enters **EB**, from Lemma 4 we have

$$S_i(\overline{t}^+) \leq F_b(\overline{t}) \leq F_j(\overline{t}) - \sigma_j \tag{32}$$

where the last inequality follows from $F_b(\overline{t}) < F_j(\overline{t})$ and the fact that both $F_j(\overline{t})$ and $F_b(\overline{t})$ are integer multiples of σ_j. Substituting (32) in what follows:

$$\begin{array}{rl} F_i(\overline{t}^+) & = \\ S_i(\overline{t}^+) + 2\sigma_i & \leq \\ F_j(\overline{t}) - \sigma_j + 2\sigma_i & \leq \\ F_j(\overline{t}) - 2\sigma_i + 2\sigma_i & \end{array} \tag{33}$$

where the penultimate inequality follows from that, since $j > i$, $\sigma_j \geq 2\sigma_i$. □

Finally, we can prove the theorem that allows QFQ to quickly choose the groups to move from **EB**/**IB** to **ER**/**IR**.

THEOREM 8. **Group unblocking** *Let i be the group that would be served upon the next packet dequeue at time $\overline{t}$, and assume that there is no group $j : j > i, F_j(\overline{t}) = F_i(\overline{t})$; in this case, if group i is actually served and $F_i(\overline{t}^+) > F_i(\overline{t})$ or if group i becomes idle at time $\overline{t}$, then all and only the groups in* **EB**/**IB** *and with order lower than i must be moved into* **ER**/**IR**.

PROOF. To prove the thesis, we first prove that group i is the only group that can block a group $h < i$. The proof is by contradiction. Suppose for a moment that a group $j > i$ blocks group h. Since $F_i(\overline{t}) < F_j(\overline{t})$ must hold for group i not to be blocked, and both $F_i(\overline{t})$ and $F_j(\overline{t})$ are integer multiples of σ_i, then

$$F_i(\overline{t}) \leq F_j(\overline{t}) - \sigma_i. \tag{34}$$

Combining this inequality with Lemma 4, we get $S_h(\overline{t}) \leq F_j(\overline{t}) - \sigma_i$ and hence, considering that $\sigma_i \geq 2\sigma_h$, $F_h(\overline{t}) = S_h(\overline{t}) + 2\sigma_h \leq F_j(\overline{t}) - \sigma_i + 2\sigma_h \leq F_j(\overline{t})$. This contradicts the fact that group j blocks group h.

As a consequence, if $F_i(t)$ increases, then, thanks to (31) and (33), all and only the blocked groups $h < i$ become ready. The same happens if group i becomes idle as a consequence of a packet dequeue. □

A.3 Proofs of the service properties

THEOREM 9. **B-WFI for QFQ** *For a flow k belonging to group i QFQ guarantees*

$$B\text{-}WFI^k = 3\phi^k\sigma_i + 2\phi^k L. \tag{35}$$

PROOF. We consider two cases. First, flow k is eligible at time t_1. In this case, we consider that, given the virtual time $V^k(t)$ of flow k in the real system, $V^k(t_1) \leq F^k(t_1)$ and $V^k(t_2) \geq S^k(t_2)$. Hence, considering also that, thanks to (21), $S_i(t_2) = F_i(t_2) - 2\sigma_i > V(t_2) - L - 2\sigma_i$, we have:

$$\begin{array}{rl} W^k(t_1,t_2) & = \\ \phi^k V^k(t_1,t_2) & = \\ \phi^k(S^k(t_2) - V^k(t_1)) & \geq \\ \phi^k(S^k(t_2) - F^k(t_1)) & \geq \\ \phi^k(S_i(t_2) - F^k(t_1)) & > \\ \phi^k(S_i(t_2) - (S^k(t_1) + \sigma_i)) & > \\ \phi^k(V(t_2) - L - 2\sigma_i - (V(t_1) + \sigma_i)) & = \\ \phi^k(V(t_2) - V(t_1) - L - 3\sigma_i) & = \\ \phi^k(V(t_2) - V(t_1)) - \phi^k L - 3\phi^k\sigma_i & > \\ \phi^k W(t_1,t_2) - 2\phi^k L - 3\phi^k\sigma_i & \end{array} \tag{36}$$

where the last inequality follows from the fact that, because of the immediate increment of $V(t)$ as a packet is dequeued (see *updateV()*), $V(t_2) - V(t_1) \geq W(t_1,t_2) - L$.

Second, flow k is not eligible at time t_1. This implies that the flow virtual time is exactly equal to $S^k(t_1)$ at time t_1 and hence, considering that, $S^k(t_1) \leq V(t_1) + \sigma_i$,

we have:

$$\begin{aligned} W^k(t_1,t_2) &\geq \\ \phi^k(S^k(t_2)-S^k(t_1)) &\geq \\ \phi^k(S_i(t_2)-S^k(t_1)) &> \\ \phi^k(V(t_2)-L-2\sigma_i-S^k(t_1)) &> \\ \phi^k(V(t_2)-L-2\sigma_i-(V(t_1)+\sigma_i) &> \\ \phi^k(V(t_2)-V(t_1)-L-3\sigma_i) &> \\ \phi^k W(t_1,t_2)-2\phi^k L-3\phi^k\sigma_i & \end{aligned} \tag{37}$$

□

THEOREM 10. **T-WFI for QFQ** *For a flow k belonging to group i, and a link with constant rate R, QFQ guarantees*

$$T\text{-}WFI^k = (3\sigma_i + 2L)\frac{1}{R}. \tag{38}$$

PROOF. Assume a generic packet arriving at t_a and completing service at t_c. Let $Q^k(t_a^+)$ be the backlog of flow k just after the arrival of the packet. Because of the immediate increment of $V(t)$ upon packet dequeue we have $V(t_a^+,t_c) \geq W(t_a,t_c) - L$. Since $W(t_a,t_c) = (t_c - t_a)R$, it follows that

$$t_c - t_a \leq \frac{V(t_c)-V(t_a^+)+L}{R} \leq \frac{F_i(t_c)+L-V(t_a^+)+L}{R} = \frac{F_i(t_c)-V(t_a^+)+2L}{R}. \tag{39}$$

To prove the theorem we will find an upper bound to $F_i(t_c)$ and a lower bound to $V(t_a^+)$. Since the approximate virtual start time of the flow increases by σ_i in any time interval $[t_1,t_2]$ during which an amount of bytes $\phi^k\sigma_i$ of the flow are transmitted, and such that there is still backlog at time t_2, it follows that

$$\begin{aligned} F_i(t_c) &= \\ S_i(t_c)+2\sigma_i &= \\ S_i(t_c)+2\sigma_i &\leq \\ S_i(t_a^+)+\sigma_i\lfloor\frac{Q^k(t_a^+)}{\phi^k\sigma_i}\rfloor+2\sigma_i &\leq \\ S_i(t_a^+)+\sigma_i\frac{Q^k(t_a^+)}{\phi^k\sigma_i}+2\sigma_i &= \\ S_i(t_a^+)+\frac{Q^k(t_a^+)}{\phi^k}+2\sigma_i & \end{aligned} \tag{40}$$

Substituting this inequality and $V(t_a^+) \geq S_i(t_a^+) - \sigma_i$ (derived from (14)) in (39), we get

$$\begin{aligned} t_c - t_a &\leq \\ \frac{S_i(t_a^+)+\frac{Q^k(t_a^+)}{\phi^k}+2\sigma_i-S_i(t_a^+)+\sigma_i+2L}{R} &= \\ \frac{\frac{Q^k(t_a^+)}{\phi^k}+3\sigma_i+2L}{R} & \end{aligned} \tag{41}$$

which proves the thesis. □