Efficient Packet Scheduling with Tight Bandwidth Distribution Guarantees

Fabio Checconi
Scuola Superiore S. Anna
Pisa, Italy

Paolo Valente
Dip. di Ingegneria dell'Informazione
Università di Modena, Italy

Luigi Rizzo
Dip. di Ingegneria dell'Informazione
Università di Pisa, Italy

***Abstract*—One of the key issues in providing tight bandwidth and delay guarantees with high speed systems is the cost of packet scheduling. Through timestamp rounding and flow grouping several schedulers have been designed that provide near-optimal bandwidth distribution with $O(1)$ time complexity. However, in one of the two lowest complexity proposals in the literature, the cost of each packet enqueue/dequeue is still proportional to the number of groups in which flows are partitioned. In the other proposal, this cost is independent of the number of groups, but a number of operations proportional to the length of the packet must be executed during each transmission.**

In this paper we present Quick Fair Queueing (QFQ), a new member of this class of schedulers that provides the same near-optimal guarantees, yet has a constant cost also with respect to the number of groups and the packet length. The peculiarity of QFQ is to partition groups so that enqueue/dequeue can be accomplished in a number of steps independent of the number of groups, without requiring extra operations during packet transmissions.

To validate the effectiveness of QFQ, we implemented it in the Linux kernel. QFQ proved able to execute each packet enqueue/dequeue in 340 instructions worst case, irrespective of number of flows and arrival patterns. Furthermore, the instructions used by QFQ are extremely well suited to a hardware implementation.

I. 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 [9]). 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 (i.e., upper bounded by a constant multiple of the maximum packet size). The two best proposals in this class, the scheduler proposed in [14], hereafter called *Group Fair Queueing—GFQ* for brevity, and SI-WF^{2}Q [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 SI-WF^{2}Q 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 SI-WF^{2}Q.

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 scheduling 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 previous proposals is on performance: the algorithm has no loops, and the simplicity of the data structures and instructions involved makes it well suited to hardware implementations. Our Linux x86 version runs in approximately 340 instructions *worst case*[2] per packet enqueue/dequeue with 32k flows. To put the number in context (all worst cases are reported): on the same platform, the DRR scheduler (which boils down to two enqueue or dequeue, but has very poor service guarantees) uses 70 instructions; a simple hash-based classifier uses 202 instructions; the HFSC [16] scheduler with 1k flows uses 965 instructions (HFSC's cost is $O(\log N)$) and a full packet processing (from input to

[1] In [9] 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.

[2] We report costs in terms of instruction counts because execution times vary by over one order of magnitude due to caching effects, see Section VI. Also, we use the (observed) worst case counts because even instruction counts vary widely depending on the traffic patterns, and average or best cases would be misleading.

output) can easily consume 2000–5000 instructions.

Paper structure: Section II complements this introduction by discussing related work. In Section III we define the system model and other terms used in the rest of the paper. Section IV presents the QFQ algorithm in detail. Section V evaluates the service guarantees. Finally, Section VI measures the performance of the algorithm on a real machine, comparing an actual implementation with the production-quality one of DRR present in Linux.

II. 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* [8], and *Bit-* and *Time- Worst-case Fair Index* (B-WFI and T-WFI [3], [5], see the definitions in Section V).

B-WFI^k and T-WFI^k (we will refer to both as WFI for brevity) represent the worst-case deviation over any time interval that a flow k may experience, in terms of service and time, with respect to the service it would receive in an ideal system providing perfect weighted bandwidth sharing.

The WFI is an interesting measure as the B-WFI can be also 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 also be used to compute the maximum delay experienced by each packet. In addition, as proven in [3], a low WFI is essential to provide tight fairness and delay guarantees in a hierarchical setting.

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], *Aliquem* [10] and *Stratified Round Robin* [12], to address some of the shortcomings (burstiness, etc.) of RR schedulers. One unfixable feature of this family of schedulers is that, irrespective of the flow's weight ϕ^k, their T-WFI has an $O(NL)$ component[3], 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* WFI, i.e., the lowest possible WFI for a non-preemptive system, and have an $O(\log N)$ time complexity.

Approximated variants of these schedulers 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], SI-WF^{2}Q [8] and the QFQ algorithm described here.

The approximation has an implication: as proved in [18], in any system using approximate timestamps, the difference between the packet completion times in the real system and in an ideal GPS system is larger than $O(1)$. However, even optimal schedulers such as WF^{2}Q and WF^{2}Q+ where this difference is $O(1)$, feature the same T-WFI=$O(L(1+1/\phi^k))$ that we find in the approximated schedulers just mentioned (GFQ, SI-WF^{2}Q, QFQ). 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.V-B). Thus, approximated timestamps still give much better guarantees than Round Robin schedulers.

Approximated timestamp-based schedulers typically use data structures that are more complex to manipulate than those used in Round Robin 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 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 [17] 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, SI-WF^{2}Q is based on a special data structure called Interleaved Stratified Timer Wheels (ISTW). ISTW containers have several nice properties that allow SI-WF^{2}Q 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, SI-WF^{2}Q runs at a low $O(1)$ amortized cost per packet transmission time.

III. 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 I. Most quantities are a function of time, but we omit the time argument (t) when not ambiguous and clear from the context.

We consider a system in which N packet flows (defined in whatever meaningful way) share a common transmission link

[3]In WF^{2}Q and WF^{2}Q+ this component is $O(L(1+1/\phi^k))$. This can still grow to $O(NL)$ for low-weight flows, but high weight flows receive better treatment.

TABLE I
DEFINITIONS.

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 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 (3)
S^k, F^k	Virtual start and finish times of flow k, see (2)
$\hat{S}^k, \hat{F}^k$	Approximated S^k and F^k see Section IV-B
i, j	Group index (groups are defined in Sec.IV-A)
S_i, F_i	Virtual start and finish times of group i, see (6)
σ_i	Slot size of group i (defined in Sec.IV-A, $\sigma_i = 2^i$)
ER, EB, IR, IB	The sets in which groups are partitioned

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 can assume that $\sum_{k=1}^{N} \phi^k \leq 1$.

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.

A. 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:

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

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.

This 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:

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

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

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

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

Flow k is said *eligible* at time t if $V(t) \geq S^k$. The 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 (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 (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 [6], an augmented balanced tree can be used to perform the latter two tasks together in $O(\log N)$ time.

IV. Quick Fair Queueing

QFQ approximates WF^{2}Q+, providing near-optimal service guarantees (see Section V) but reducing the implementation complexity to $O(1)$ with small constants thanks to three main techniques. The first two are also widely adopted in the literature [8], [12], [14]: they are *flow grouping*, which partitions flows into a constant number of groups, and *timestamp rounding*, which reduces the cardinality of sorting keys so that an $O(1)$ bucket sort suffices to sort flows within a group. The third technique, *group sets*, is peculiar to QFQ and is key to a dramatic reduction in the cost of the algorithm: groups are mapped into four sets, each represented by a single machine word. Due to the properties of the sets, finding the group to schedule only requires a Find First bit Set (FFS) CPU instruction, and all the data structure housekeeping (including changing the state of multiple groups at once) only requires a constant number of bitwise operations on these sets, independent of either the number of groups or the packet length.

A. 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

$$i = \left\lceil \log_2 \frac{L^k}{\phi^k} \right\rceil \tag{4}$$

where L^k is the maximum size of packets for flow k.

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 (2), $F^k - S^k \leq \sigma_i$ for any flow k in group i.

Because of the definition of groups, 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, even 32 groups are largely sufficient in many cases)*. This is trivially proven substituting values in the equation[4], and lets us represent a set of groups with a bitmap that fits in a single machine word.

B. Timestamp rounding

QFQ computes S^k and F^k for each flow using the exact algorithm in (2). However, when selecting flows for eligibility or scheduling the next flows to serve, QFQ uses the approximate values[5]:

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

Furthermore, QFQ replaces $\min_{k\in B(t)} S^k$ with $\min_{k\in B(t)} \hat{S}^k$ in computing (3).

A variant of the *Globally Bounded Timestamp* (GBT) property, proved in [1, Theorems 1 and 2] establishes bounds for $V(t) - \hat{S}^k(t)$ and $\hat{F}^k(t) - V(t)$ that we use to simplify eligibility computation and flow sorting. In particular, at any time $\hat{S}^k$ can never exceed $V(t)$ by more than σ_i, and the range of values for $\hat{S}^k$ is limited to $2+\lceil L/\sigma_i \rceil$ times σ_i. The limited range and the rounding to multiples of σ_i, implies that the $\hat{S}^k$ can only assume a constant number of different values. Hence, we can sort flows within a group using a constant-time bucket sort algorithm. The use of $\hat{S}^k$ in (3) also saves another sorting step, because, as we will see, the *group sets* defined in the next section will let us compute (3) in constant time.

Once flows in a group are sorted, we can easily compute

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

which are called the *group's virtual start and finish times*.

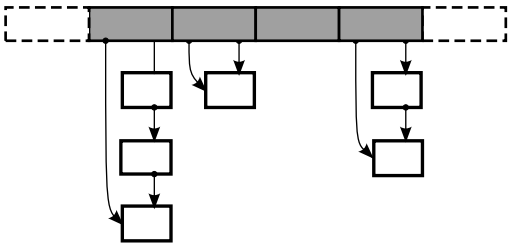

Fig. 1. A representation of bucket lists. The number of buckets (grey), i.e. possible values of S^k, is fixed and independent of the number of flows in the group. Each list member corresponds to a flow, and the only list operations required are tail insertion and head removal.

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. 1). 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, and can use a constant-time Find First bit Set (FFS) instruction to locate the first non-empty bucket, which we need to find to select the next flow to serve.

C. Group sets and their properties

QFQ partitions backlogged groups into four distinct sets, which reduce scheduling and bookkeeping operations to simple set manipulations. Given the small number of groups, each set can be represented with a single machine word, and set manipulations can be implemented with basic CPU instructions such as AND, OR and FFS.

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

- **eligibility:** group i is said *Eligible* at time t iff $S_i \leq V(t)$, and *Ineligible* otherwise.
- **readiness:** independent of its own eligibility, a group i is said to be *Ready* if there is no eligible group $j > i$ with $F_j < F_i$. Otherwise, such a group j exists, and we say that group i is *Blocked* by group j.

Readiness is not intuitive but together with eligibility gives the sets a number of interesting properties, fully exploited in QFQ. In particular, as will be demonstrated later:

[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.

[5] There is only one special case where the S^k for certain newly backlogged groups is pushed down to preserve the ordering properties of the set **EB**, described in detail in [1, near Lemma 4]. There it is also proven that this is done without violating service guarantees.

- groups within each set are ordered by group finish times ($\forall j, i \in \mathbf{X} : j > i \implies F_j > F_i$);
- the next packet to serve will always be the head packet of the first flow of the first group in $\mathbf{ER}$;
- on dequeue and on virtual time changes, we can move multiple groups at once from one set to another. The groups to be moved are located in an easy-to-find portion of the source set, as explained in Section IV-D3.

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

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

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.

The ordering properties of sets are proven as follows, and hold over any event that changes set membership:

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 $2\sigma_i \leq \sigma_j$ hence $S_i \leq S_j$;
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 are increasing with i;
3) the sorting of $\mathbf{ER}$ by F_i is proven in [1, Theorem 4];
4) the sorting of $\mathbf{EB}$ by F_i is proven in [1, Theorem 5];

By definition, if $i \in \mathbf{EB}$, then at least one group $j \in \mathbf{ER}$ has $F_j \leq F_i$. Because $\mathbf{ER}$ is sorted by F_i, the readiness test for group i needs only to look at the lowest-order group in $\mathbf{ER}$ with an index $x > i$. Function *compute_group_state()* in Fig. 2 does the computation of set membership for a group.

```
1  // Compute the group's state, see Section IV-C
2  compute_group_state(in: group) : state
3  {
4    state = 0 ; // default: not ELIGIBLE, not READY
5    // Find lowest order group x > group.index. This is the one
6    // that may block us.
7    x = ffs_from(set[ER], group.index);
8    if (x == NO_GROUP || groups[x].F > group.F)
9        state |= R; // READY
10   if (group.S <= V)
11       state |= E; // ELIGIBLE
12   return state;
13 }
```

Fig. 2. The *compute_group_state()* function implements both the eligibility and the readiness tests and returns the state of the group passed as argument. *ffs_from(d,i)* return the index of the first bit set in d after position i.

D. Quick Fair Queueing: the algorithms

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

1) The enqueue function: The full enqueue algorithm is shown in Fig. 3. First of all the packet is appended to the flow's queue, and nothing else needs to be done if the flow is already backlogged (which in turn means that the link is not idle,

```
1  // enqueue the input pkt of the input flow
2  enqueue(in: pkt, in: flow)
3  {
4    tail_insert(pkt, flow.queue) ; // always enqueue
5    if (flow.queue.head != pkt)
6        return; // Flow is already backlogged, nothing to do.
7    // Update flow timestamps according to (2)
8    flow.S = max(flow.F, V) ;
9    flow.F = flow.S + pkt.length / flow.weight ;
10   g = flow.group ; // reference to the group
11   old_state = g.state ;
12   if (old_state == IDLE || flow.S < g.S) {
13       // compute group timestamps according to (5)
14       g.S = floor(flow.S, g.slot_size) ; // clear the low i
             bits
15       g.F = group.S + 2 * g.slot_size ;
16   }
17   bucket_insert(flow, g) ; // constant time
18
19   // If there is some backlogged group, at least one is in ER;
20   // otherwise, make sure V ≥ g.S to remain work conserving.
21   if (set[ER] == 0 && V < g.S)
22       V = g.S ;
23
24   g.state = compute_group_state(g) ;
25   // Remove from old set, add to new one
26   move_groups(1 << g.index, old_state, g.state) ;
27   if (link.idle)
28       dequeue();
29 }
```

Fig. 3. The *enqueue()* function, called on packet arrivals. In the listing, IB, IR, EB, ER are the constants 0..3 corresponding to combinations of the Eligible and Blocked flags. IDLE=4 represents a non-backlogged group.

being the algorithm work-conserving). Otherwise (lines 8–16) we update the flow's timestamps and possibly the group's timestamp (because the slot size is a power of 2, the *floor()* function just returns the argument with the i least significant bits cleared). Following that, we use a constant time bucket sort (line 17) to order the flow with respect to other flows in the group. At this point we possibly update $V(t)$ as in (2), and update the state of the group, which may have changed due to the new values of S_i, F_i and $V(t)$. The computation of the new state of the group is done in function *compute_group_state()* in Fig. 2. Finally, if the group has changed state, we need to move it from one set to another, done by calling function *move_groups()* in Fig. 6. The very last step, if the link was idle, is to call *dequeue()*.

Note that an enqueue involves no movement of other groups between sets. $V(t)$ changes only if all other groups were idle, so there are no eligibility changes. If the flow was already backlogged, no group changes its finish time, so there are no readiness changes. If the group j containing this flow just became backlogged, its start time is at least as large as $V(t)$, hence $F_j > \lfloor V(t)/\sigma_j \rfloor \sigma_j + \sigma_j$. 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 F_i, smallest F_j).

2) The dequeue function: Function *dequeue()* in Fig. 4 is called whenever the link is idle and we have packets to transmit. The function returns the next packet to transmit, and

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

Fig. 4. The *dequeue* function. See Section IV-D2 for a full description.

updates data structures as needed.

As discussed, the packet selection is straightforward: upon the call to *dequeue()* at least one flow is eligible, so $\mathbf{ER}$ is not empty, and we just need to pick (lines 5–7) the group with the lowest index in $\mathbf{ER}$ and from that the first packet from the first flow. The flow's timestamps are updated, and the flow is possibly reinserted in the bucketlist (lines 10–13).

Next (lines 16–22), the group's timestamps are updated, resulting also in an update of the group's state. If the group has increased its finish time or it has become idle (lines 24–29), we may need to unblock some other groups, using function *unblock_groups()* described in the next Subsection. After that, we move the group to its new set.

Finally, we update $V(t)$ to account for the newly transmitted packet, and lines 34–41 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()*.

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

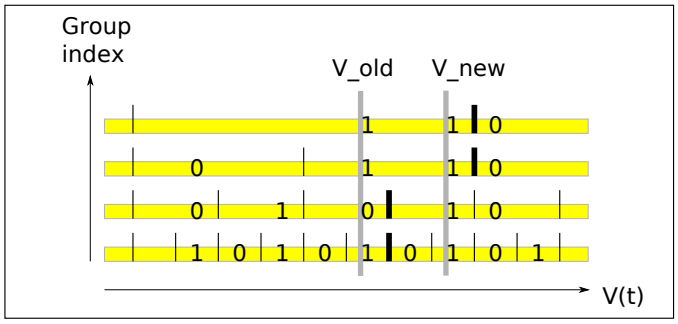

Fig. 5. How to compute changes in eligibility when $V(t)$ grows. Given $V(t) = V_{old}$, the admissible S_i's for ineligible groups can only assume the discrete values on the tick markers (spaced by $\sigma_i = 2^i$). If we label the intervals on each line as alternating 0's and 1's, the binary representation of $V(t)$ coincides with the sequence of labels of the intervals containing $V(t)$. When $V(t)$ moves from V_{old} to V_{new}, group i becomes eligible iff some bit $j \geq i$ changes in the transition.

mask. Simple bit operations do the job.

Function *make_eligible()* in Fig. 6 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. 5 gives a graphical representation of the possible values of S_i's and $V(t)$, and the binary representation of $V(t)$. As explained in the caption, if j is the index of the highest bit that changes in the binary representation of $V(t)$, then all backlogged groups with index $i \leq j$ will become eligible. Function *make_eligible()* computes the index j, using an XOR followed by a *Find Last Set* (fls) operation; then computes the set $\mathbf{new\_e}$ that includes all indexes $i \leq j$, and calls function *move_groups* to move groups whose index is in $\mathbf{new\_e}$ from $\mathbf{IR}$ to $\mathbf{ER}$ and from $\mathbf{IB}$ to $\mathbf{EB}$.

Function *unblock_groups()* is completely unintuitive and relies partly on [1, Theorem 6]. Essentially, if, upon serving group i, F_i does not change or coincides with some $F_j, j > i, j \in \mathbf{ER}$ (line 22–23), all currently blocked group will remain blocked by either group i or group j. In other cases, the theorem shows how we can find the subset of groups (all indexes smaller than i) that must be unblocked.

E. Time and space complexity

From the listings is it 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.

In terms of space, the per-flow overhead is approximately 24 bytes (two timestamps plus one pointer and a few flags), and we have 32..280 bytes per group (to store the slot pointers, two timestamps and some flags).

V. Service properties

We report here both the B-WFI (bit guarantees) and the T-WFI (time guarantees) of QFQ.

```
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 eligible
9  // as V(t) grows from V1 to V2.
10 // This uses the logic described in Fig. 5
11 make_eligible(in: V1, in: V2)
12 {
13     j = fls(V1 XOR V2) ; // highest bit changed in V(t)
14     new_e = (1 << (j+1)) - 1 ;
15     move_groups(new_e, IR, ER) ;
16     move_groups(new_e, IB, EB) ;
17 }
18
19 // Possibly unblock groups after serving group i with F=old_F
20 unblock_groups(in: i, in: old_F)
21 {
22       x = ffs_from(set[ER], i + 1)] ;
23       if (x == NO_GROUP || groups[x].F > old_F) {
24           // Unblock all the lower order groups (Theorem 6)
25           low_groups = (1 << i) - 1 ;
26           move_groups(low_groups, EB, ER) ;
27           move_groups(low_groups, IB, IR) ;
28       }
29 }
```

Fig. 6. Support functions to recompute the set of eligible groups after a flow has been served. These are described in Sec. IV-D2

A. Bit guarantees

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

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

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

$$\textit{B-WFI}^k = 3\phi^k \sigma_i + 2\phi^k L. \tag{8}$$

The proof does not differ in principle from the proof of the B-WFI for WF^{2}Q+ [3], and can be found in [1, Theorem 5]. As a term of comparison, in a perfectly fair ideal fluid system like the GPS server B-WFI$^k = 0$ [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 $(L^k + 2\phi^k)L$.

B. 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:

$$\text{T-WFI}^k \equiv \max\left(t_c - t_a - \frac{Q^k(t_a^+)}{\phi^k R}\right), \tag{9}$$

where t_a and t_c are, respectively, the arrival and completion time of a packet, and $Q^k(t)$ is the backlog of flow k.

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

$$\textit{T-WFI}^k = \left(3\left\lceil\frac{L^k}{\phi^k}\right\rceil + 2L\right)\frac{1}{R}\ . \tag{10}$$

For the proof, once again see [1, Theorem 6]. 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$^k = (\frac{L^k}{\phi^k} + 2L)/R$.

VI. Experimental results

The most interesting feature of QFQ is its bounded (independent of number of flows) and small per-packet execution time, which makes the algorithm extremely practical and amenable to really fast implementations. This section analyzes the performance of our Linux implementation of QFQ[6], and compares it with the Deficit Round Robin (DRR) scheduler available on the same system.

The choice of DRR is relevant because it represents a practical lower bound for the complexity of a scheduler. It would have been interesting to evaluate GFQ or SI-WF^{2}Q as well, but the lack of a full specification and public implementation of these algorithms prevented us from doing so. Implementing these schedulers from scratch would have required an exceeding amount of coding, well outside the scope of this work; and, especially, implementation decisions not fully covered by the respective papers could have made the results not representative of the original intentions of their authors, leading to an unfair comparison.

The experiments have been ran using three PCs acting as source, router, and sink of network traffic. Source and router were connected by a 1 Gbit/s card, whereas router and sink were connected by a 100 Mbit/s card. The source generated UDP traffic using a modified version of the in-kernel traffic generator, `pktgen` [11], with different configurations. In this paper we focus on two of them: *light load*, with packet lengths uniformly distributed between 64 and 1518 bytes, with a 100 Mbit/s link between source and router; *heavy load*, with minimum-sized packets (64 bytes) and 1 Gbit/s link between source and router. The two scenarios serve to exercise different code paths in the schedulers, as the execution times may depend on the backlog status of flows, groups and sets as well as on the content of the cache and the interleaving with other system activities). In particular, low load is a worst case scenario (instruction-wise) for QFQ, because there is a lot of bookkeeping involved in every *enqueue()* and *dequeue()*. Conversely, at high load, QFQ's *enqueue()* is as inexpensive as in DRR, and *dequeue()* is normally cheap because the current group is often backlogged.

[6]The source code of the scheduler and all the other software used to obtain the results presented in this section are available, together with all the data collected during the measurements, at [2].

When doing measurements on a machine with deep memory hierarchies (our CPU was a 3.0 GHz, dual core, Intel E8400) and code with variable execution paths, single performance numbers are insufficient to describe the behavior of the system. As a consequence the evaluation has been done by measuring three parameters across each execution of the *enqueue()* and *dequeue()* functions: the number of *retired instructions*, the total *CPU cycles*, and the number of *cache misses*. All of them were computed by reading the relevant CPU performance counters around the functions being measured. Samples were stored in a kernel ring buffer of 64k elements, and dumped to userspace every 5s. To let the system settle, no dumps were done in the first 15s of the experiment, and the first 10 dumps were ignored. We used the next 30 dumps for our evaluation. We took care of preventing major sources of imprecision, such as interrupts and preemptions/migrations to interfere with the measurements, by disabling interrupts around the measured sections of code, and pinning the relevant interrupt handlers to a single core whenever possible.

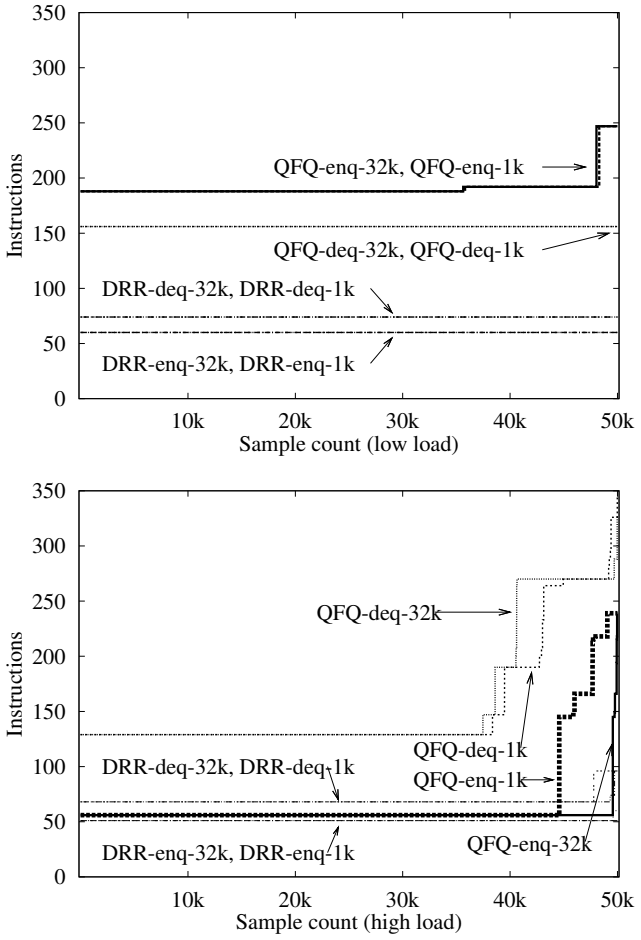

Fig. 7. Distribution of the number of instructions.

Given the potentially high variance of the samples, before computing any statistics it is important to look at the distribution of the data. Fig. 7 shows the distributions of the instruction counts for enqueue/dequeue in various configurations, accumulated across multiple experiments. We only show a total of 50k samples in these graphs, but distributions are essentially unmodified as the number of samples increases. As expected, instruction counts proved to be stable across multiple executions. and they flat regions of the curves correspond to the different code paths in the functions under test.

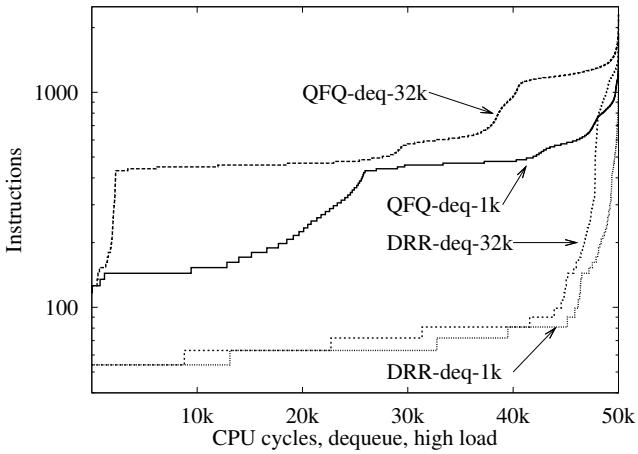

Fig. 8. Distribution of CPU cycles, dequeue.

DRR is mostly oblivious to the state of the system. Its *enqueue()* function just appends the packet to the flow's queue, and appends the flow to the FIFO list if not there yet, so the instruction count is practically flat. DRR's *dequeue()* also has a mostly constant instruction count irrespective of the load.

QFQ's *enqueue()* also starts by enqueueing the packet, but then possibly has to do some extra work depending on the flow/group/system state. The frequency of extra work is higher at low load, when flows and groups are almost always idle—in fact, the shortest code path is never exercised in our low-load scenario, and the instruction count for QFQ's *enqueue()* varies between 180 and 240. At high load, the function has an instruction count similar to DRR's *enqueue()* (55 instructions) in 80–90% of the cases, while the remaining ones exercise different code paths, up to the same 240 instructions as before.

QFQ's *dequeue()* is influenced by the load in a similar way. At low load the cost is always around 155 instructions, because in this scenario there is always only one backlogged flow. At high load the instruction count varies, again due to the existence of multiple code paths which are taken on different executions. We range from a minimum of 130 instructions to a maximum of 340 instructions (only in 2–3% of the cases; the 95-percentile value is around 270 instructions).

Note that the worst case for QFQ's *dequeue()* is the same with 1k and 32k flows—the curves have different shapes only because the relative frequency of the various code paths changes.

Moving from instruction to cycle counts the nice and flat curves we have seen so far disappear. Most of this depends on the effect of caches and of instruction latencies/dependencies. A small part also depends on measurement errors caused by the effect of hardware factors like pipelining and out-of-order

execution on short observation periods.

The effects are visible on all configurations. Due to space limitations we focus on one set of curves, namely the distribution of durations, in CPU cycles, of the *dequeue()* operation under high-load conditions. The four curves (for QFQ and DRR, 1k and 32k flows) are shown in Figure 8. With low load, and in the best runs, a warm cache allows the delivery of multiple instructions per clock cycle, but apart from some lucky runs, especially as the number of flows grows, the cache becomes less and less effective, and execution times quickly ramp up for both QFQ and DRR. The two algorithms have $O(1)$ cost, but here the curves with 1k and 32k flows are different for both algorithms. A large number of flows implies a larger memory footprint and a higher chance of cache misses. Even though misses are limited (less than 2–3 in most cases), they are so expensive that the execution time of the functions can be affected badly—near the end of the distribution, both DRR and QFQ have some executions that consume over 2000 clock cycles, compared to the best values of 54 and 117, respectively. Such a large deviation explains why performance claims should be taken with extreme care by the reader, if not backed up by actual experimental data. Moreover, given the nature of the architecture in consideration, the real worst-case execution times can hardly be estimated.

With these considerations in mind, the following table summarizes the the average per-packet execution times, and the 95% confidence intervals, measured for QFQ and DRR in the worst performing scenario (duration-wise), which proved to be the heavy load. The purpose of the table is not to provide single numbers to characterise the two scheduler, but rather to warn the reader once more on the care that must be used in analysing these measurements. As an example, note how, even though the instruction counts are relatively repeatable, some of the entries have a large variance which reflects the multi-modal results of the measurements.

Moving to execution times (our machine has a fixed 3 GHz clock speed), as expected we see that the variance explodes, and the ratio between instructions and times changes widely from one configuration to another.

	1k Flows		32k Flows	
	QFQ	DRR	QFQ	DRR
Instructions				
enqueue	70.1 ± 42.2	51.1 ± 0.8	56.9 ± 10.6	51 ± 0
dequeue	153.9 ± 51.0	68.2 ± 2.1	158.5 ± 55.3	69.2 ± 5.7
Duration, in ns				
enqueue	40.2 ± 46.9	40.9 ± 45.1	62.1 ± 63.8	56.5 ± 61.8
dequeue	117.0 ± 65.0	25.9 ± 22.2	212.6 ± 105.3	39.1 ± 67.0

TABLE II
PER-PACKET INSTRUCTIONS/EXECUTION TIMES.

VII. 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. On each packet arrival/departure, the algorithm executes only one multiplication and a small number of arithmetic and logic operations. The simplicity of QFQ is witnessed by its low instruction count (340 per packet worst case) and fast execution times.

In addition to a detailed description of the algorithm and a thorough theoretical analysis of the service guarantees, the paper also provides a detailed evaluation of the performance on real hardware, both in terms of instructions and CPU cycles. We have also developed a production quality implementation of QFQ which is, to the best of our knowledge, the first publicly available implementation of a scheduler of this class ($O(1)$ time and near-optimal WFI). The code, which works with the Linux Packet Scheduling framework, has been used for our measurements and is available at [2].

REFERENCES

[1] http://feanor.sssup.it/~fabio/linux/qfq/qfq-full.pdf.

[2] http://feanor.sssup.it/~fabio/linux/qfq/.

[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] 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.

[7] M. Karsten. Approximation of generalized processor sharing with stratified interleaved timer wheels. *Work in progress*.

[8] 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.

[9] 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.

[10] 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.

[11] Robert Olsson. Pktgen: the Linux packet generator. In *Proceedings of the 2005 Ottawa Linux Symposium*, July 2005.

[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] Ion Stoica, Hui Zhang, and T. S. Eugene Ng. A hierarchical fair service curve algorithm for link-sharing, real-time, and priority services. *IEEE/ACM Trans. Netw.*, 8(2):185–199, 2000.

[17] 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.

[18] 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

We prove here both the properties of the data structure used in Sec.IV, 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 I.

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

$$\lfloor y \rfloor_{\sigma_i} \leq \lceil x \rceil_{\sigma_i}. \tag{11}$$

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. Group GBT under QFQ

We start by proving per-group upper bounds for $S_i(t) - V(t)$ (in Theorem 1) and for $V(t) - F_i(t)$ (Theorem 2, 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 [7], 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 1. Upper bound for $S_i(t) - V(t)$.
For any backlogged group i and $\forall t$

$$S_i(t) \leq \left\lceil \frac{V(t)}{\sigma_i} \right\rceil \sigma_i = \lceil V(t) \rceil_{\sigma_i} \tag{12}$$

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 (12) trivially holds. Hence, to prove (12) 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:*

$$\sum_{k \in I(t)} \left(l^k(t) + \phi^k [V' - F^k(t)] \right) \leq V' - V(t) \tag{13}$$

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:

$$\sum_{k \in I(t)} \left\{ l^k(t) + \phi^k [V(t) - F^k(t)] \right\} \leq 0 \tag{14}$$

$$\sum_{k \in I(t)} \left\{ l^k(t) + \phi^k [V(t) - V'] + \phi^k [V' - F^k(t)] \right\} \leq 0 \tag{15}$$

$$\sum_{k \in I(t)} \left\{ l^k(t) + \phi^k [V' - F^k(t)] \right\} \leq \sum_{k \in I(t)} \phi^k [V' - V(t)] \tag{16}$$

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

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 $\forall t : L + V' \geq V(t)$:*

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

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 2. 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:

$$\hat{F}_2 \geq \hat{S}_2 \geq V_2 \geq V_2 - L \quad (27)$$

This concludes the proof. ■

B. Proofs of the data structure properties

We can now prove the theorems used in Sec.IV considering the only two events that can change the state of the scheduler, namely packet enqueue and packet dequeue. We start from Theorem 3, which gives the per-group slot occupancy.

Theorem 3. *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 1, 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* $\bar{t}$ *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 $F_h(\bar{t}) \leq F_i(\bar{t}^+)$ *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 $S_i(\bar{t}^+) \geq \lfloor V(\bar{t}) \rfloor_{\sigma_i}$ 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 $\bar{t}$, and (12) in the following difference we get:

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

where $\lceil V(\bar{t}) \rceil_{\sigma_h} \leq \lceil V(\bar{t}) \rceil_{\sigma_i}$ 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 4. *Set* **ER** *is ordered by group virtual finish time.*

Proof: We will prove the thesis by induction. In the base case $\mathbf{ER} = \emptyset$ 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 $h \in \mathbf{ER}$ 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.IV-C, 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 $b \in \mathbf{ER}$ 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):

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

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 schedule is the same as if the packet would have actually arrived at the time that would have lead to that value of the pair of timestamps according to the exact formulas. And this is what happens using (29), for the following reason. Should the packet that lets group i become backlogged had arrived at a time instant $\bar{t}_p \leq t_p$ at which $V(\bar{t}_p) = F_b(t_p)$, group i would have however got a virtual start time higher than $F_b(t_p)$. In addition, since $V(\bar{t_p}) = F_b(t_p)$, then group b must have been backlogged at time $\bar{t_p} \leq t_p$ to have a virtual finish time equal to $F_b(t_p)$ at time t_p. In the end group i would 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

$$F_h(t_0) \geq F^k(t_0) > F_i(t_2). \tag{30}$$

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 5. *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

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

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.IV-C, 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 $\bar{t}$. Independently of the reason why group i enters **EB**, from Lemma 4 we have

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

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

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

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 6. Group unblocking *Let i be the group that would be served upon the next packet dequeue at time $\bar{t}$, and assume that there is no group $j : j > i, F_j(\bar{t}) = F_i(\bar{t})$; in this case, if group i is actually served and $F_i(\bar{t}^+) > F_i(\bar{t})$ or if group i becomes idle at time $\bar{t}$, 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 $F_i(\bar{t}) < F_j(\bar{t})$ must hold for group i not

to be blocked, and both $F_i(\bar{t})$ and $F_j(\bar{t})$ are integer multiples of σ_i, then

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

Combining this inequality with Lemma 4, we get $S_h(\bar{t}) \leq F_j(\bar{t}) - \sigma_i$ and hence, considering that $\sigma_i \geq 2\sigma_h$, $F_h(\bar{t}) = S_h(\bar{t}) + 2\sigma_h \leq F_j(\bar{t}) - \sigma_i + 2\sigma_h \leq F_j(\bar{t})$. 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. ■

C. Proofs of the service properties

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

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

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:

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

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

$$\textit{T-WFI}^k = \left(3\left\lceil\frac{L^k}{\phi^k}\right\rceil + 2L\right)\frac{1}{R}. \tag{38}$$

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

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

Substituting this inequality and $V(t_a^+) \geq S_i(t_a^+) - \sigma_i$ (derived from (12)) 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. ■